Institute for ADVANCED ANALYTICS

Apache Spark for Distributed Machine Learning

Dan Zaratsian
AI/ML Architect, Gaming Solutions @ Google
d.zaratsian@gmail.com
https://github.com/zaratsian

TOPICS

- Session 1: Course Intro, Trends, and Approach to AI/ML
- Session 2: SQL and NoSQL
- Session 3: Cloud Machine Learning Services
- Session 4: Distributed ML with Spark and Tensorflow
- Session 5: Cloud Generative AI Services and Architectures
- Session 6: Serverless ML, Architectures, and Deploying ML

Why do we need Distributed ML & Distributed Compute?

Process Large Datasets

When Speed is Required

Storage and I/O Requirements

Data Redundancy and/or Scale

A few options...

Spark

beam

RAY

TensorFlow

TF.DISTRIBUTE.STRATEGY

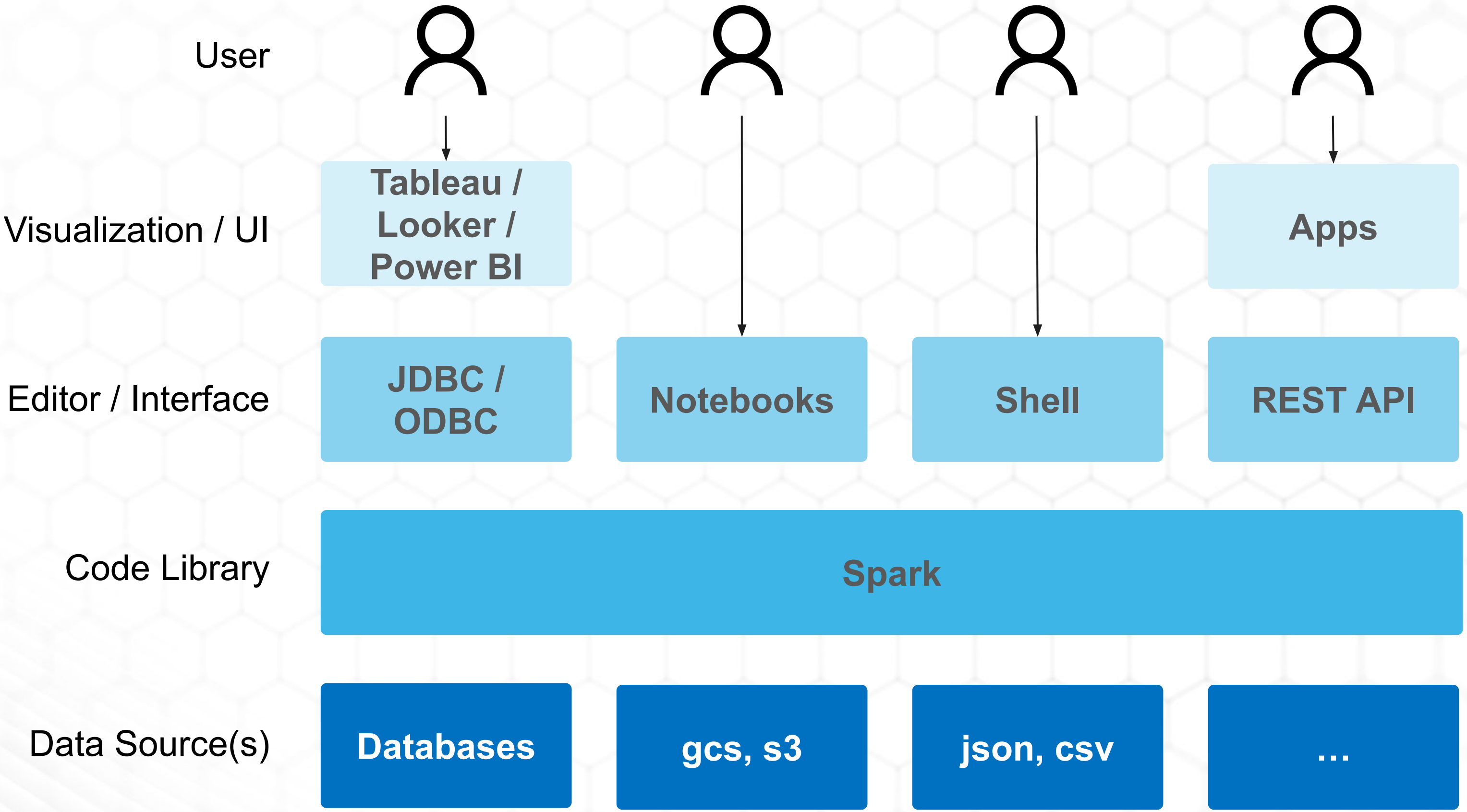
TFX

PyTorch

TORCH.DISTRIBUTED

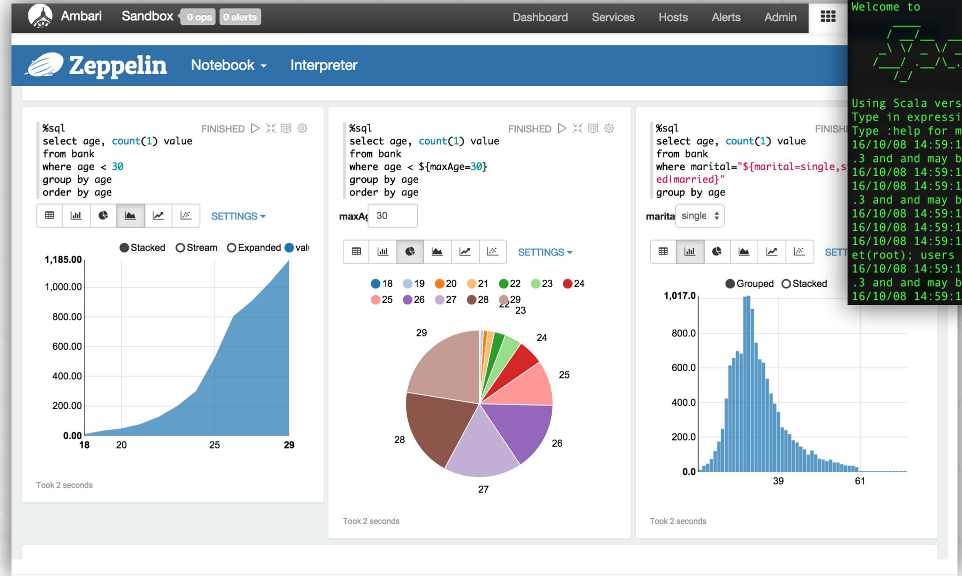
- In-Memory computing (speed)
- Distributed, cluster computing
- Machine Learning
- Flexible Data Processing
- Multiple APIs

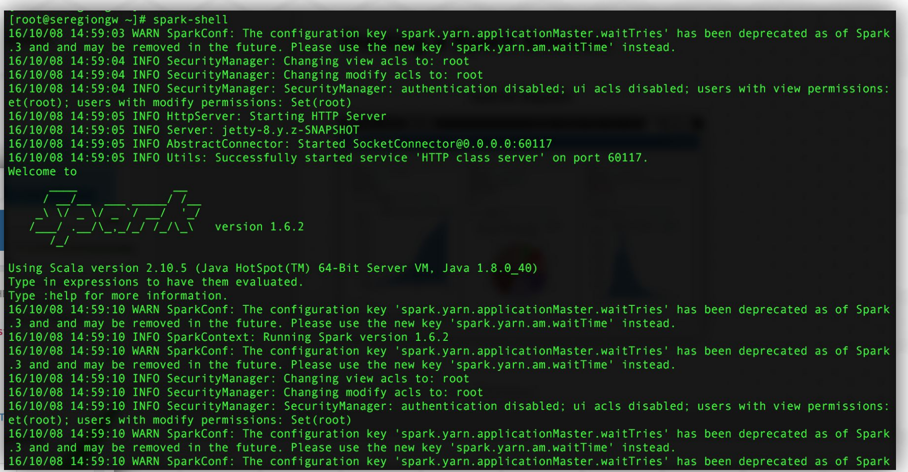
- 2009 (UC Berkeley)
- 2010 (Open Sourced)
- 2013 (Apache Project)
- 2015-03 (Spark 1.3)
- …
- 2016-07 (Spark 2.0)
- 2020-06 (Spark 3.0)
- 2024-02 (Spark 3.5.1)


User
Visualization / UI
Tableau / Looker / Power BI
Apps
Editor / Interface
JDBC / ODBC
Notebooks
Shell
REST API
Code Library
Spark
Data Source(s)
Databases
gcs, s3
json, csv
...

Starting a Spark Session

Apache Zeppelin



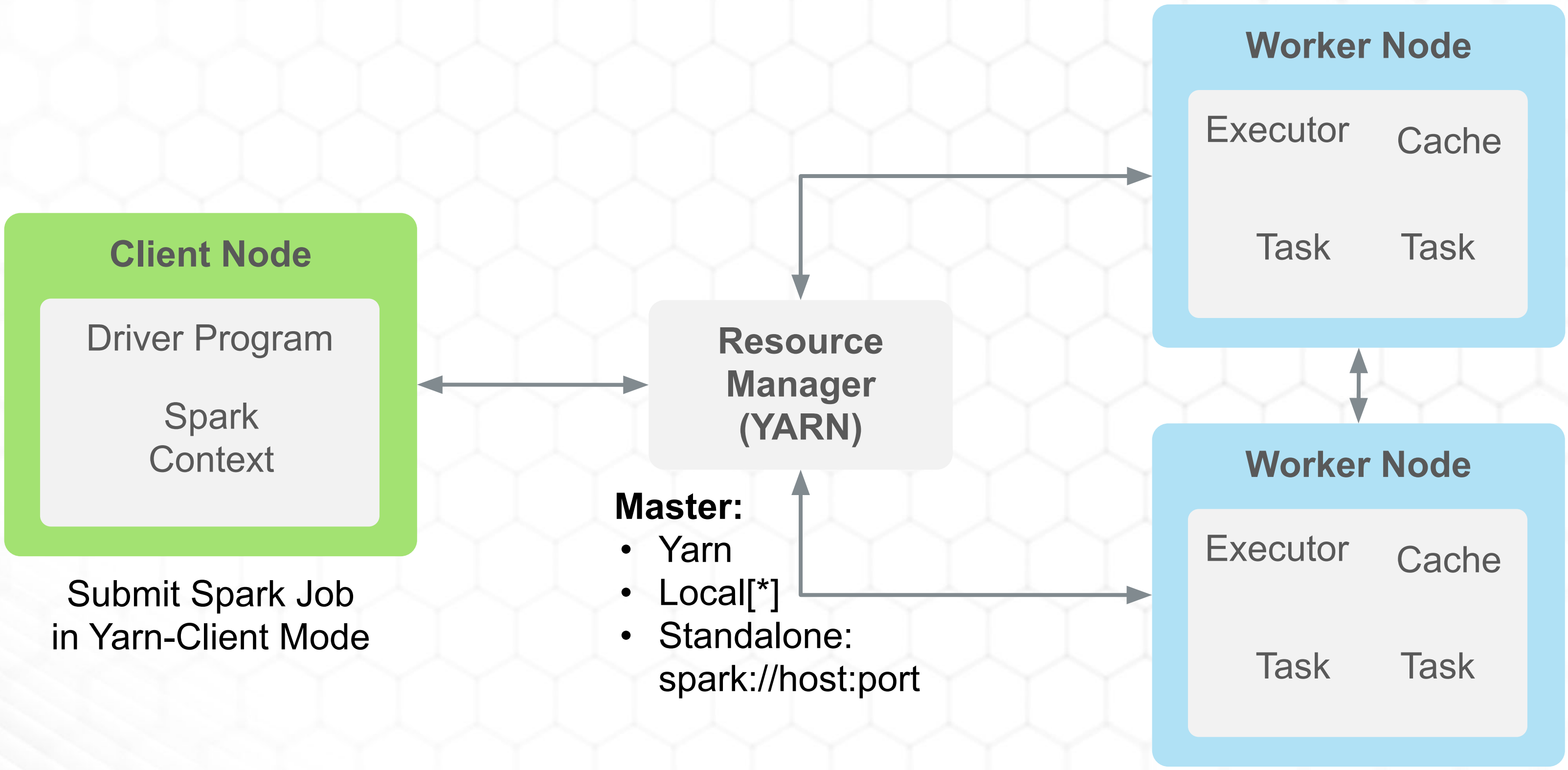


./bin/spark-shell
./bin/pyspark
./bin/sparkr
./bin/spark-sql

./bin/spark-submit

./bin/spark-submit --master yarn --deploy-mode cluster --executor-memory 20G --num-executors 50 /path/to/my_pyspark.py

What happens when you submit a Spark Job



Spark ML

Spark Machine Learning

- Machine Learning Library (MLlib)
- **Library Includes:**
  - **ML Algorithms:** Classification, regression, clustering, and collaborative filtering
  - **Featurization:** Feature extraction, transformation, dimensionality reduction
  - **Pipelines:** Tools for constructing, evaluating, and tuning ML Pipelines
  - **Persistence:** Saving and load algorithms, models, and Pipelines
  - **Utilities:** Linear algebra, statistics, data handling, etc.

Spark ML Vectors

Dense Vector is backed by a double array representing its values
Sparse Vector is backed by two parallel arrays (used when many values are zero)

Example: (1.0, 0.0, 3.0)

Dense: Vectors.dense(1.0, 0.0, 3.0)

Sparse: Vectors.sparse(3, [(0, 2), (1.0,3.0)])

Dense Vectors:

- Most elements are non-zero
- Computation efficiency
- Small/Medium Vector size

Sparse Vectors:

- Most elements are zero (ie. NLP use case)
- High dimensional data (ie. text and image processing)
- Memory efficiency

Spark ML Pipelines

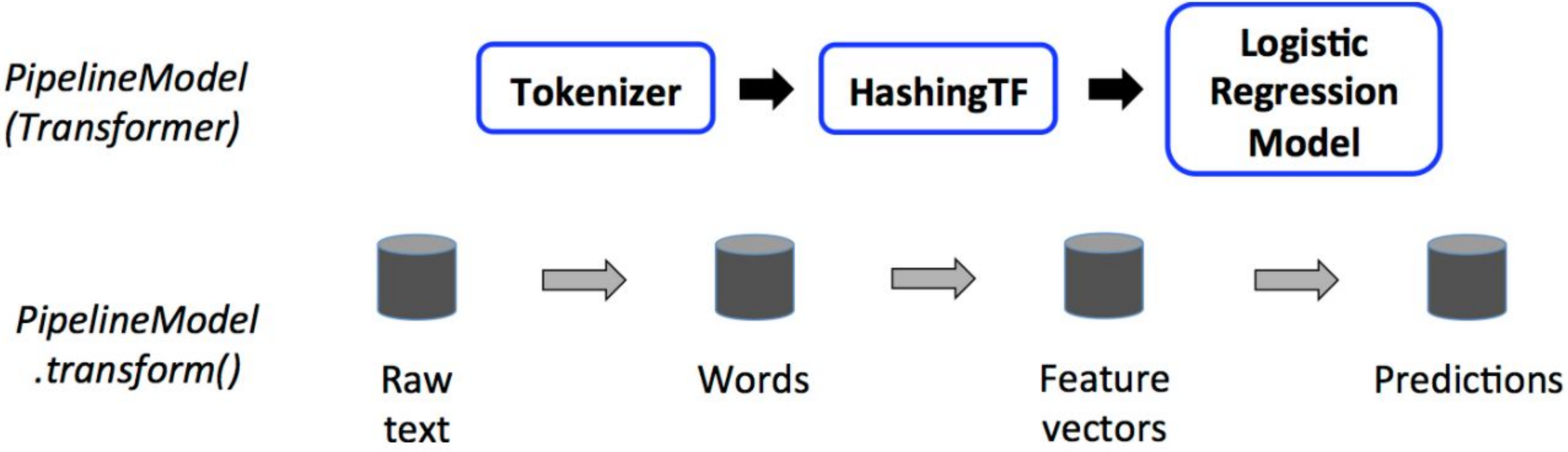
Make it easier to combine multiple algorithms into a single workflow.
Sequence of stages, and each stage is either a Transformer or an Estimator.

Core Components of a Pipeline:

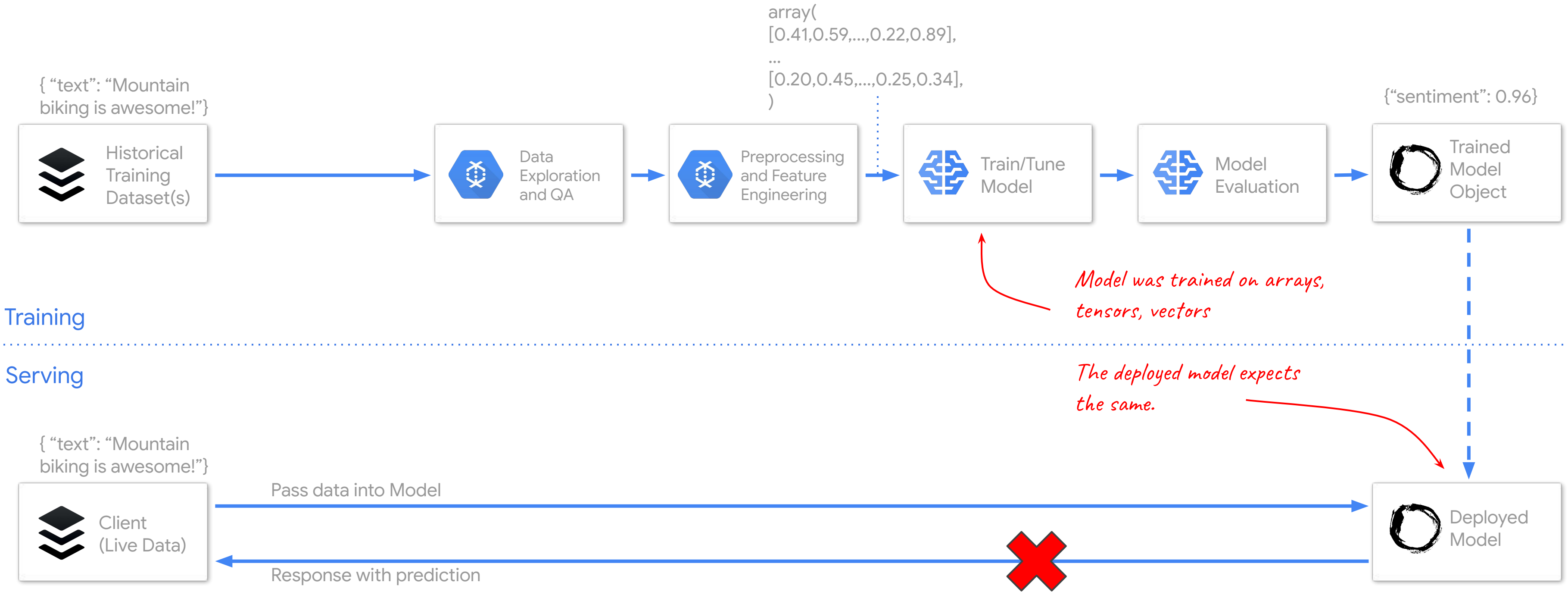
- **DataFrame**: ML dataset
- **Transformer**: Transforms one DataFrame into another DataFrame
- **Estimator**: Algorithm which can be fit on a DataFrame to produce a Transformer. E.g., a learning algorithm is an Estimator which trains on a DataFrame and produces a model.
- **Pipeline**: A Pipeline chains multiple Transformers and Estimators together to specify an ML workflow.
- **Parameter**: All Transformers and Estimators now share a common API for passing/specifying parameters.

APACHE Spark ML

Spark Machine Learning



Example:



Model was trained on arrays, tensors, vectors

The deployed model expects the same.

Spark ML Pipeline

In machine learning, it is common to run a sequence of algorithms. We can bundle these algorithms and data processing steps into a Spark ML Pipeline.

A Pipeline is specified as a sequence of stages, and each stage is either a **Transformer** or an **Estimator**. The stages are specified as an ordered array (or DAG).

It's often a best practice to save a model or a pipeline to disk for later use.

Reference Code (in Colab Notebook): Spark ML Pipeline Example

Spark Pipeline Demo

sparkml_simple_pipeline.ipynb

File Edit View Insert Runtime Tools Help Last edited on March 11

Comment Share

+ Code + Text Connect

PySpark Machine Learning

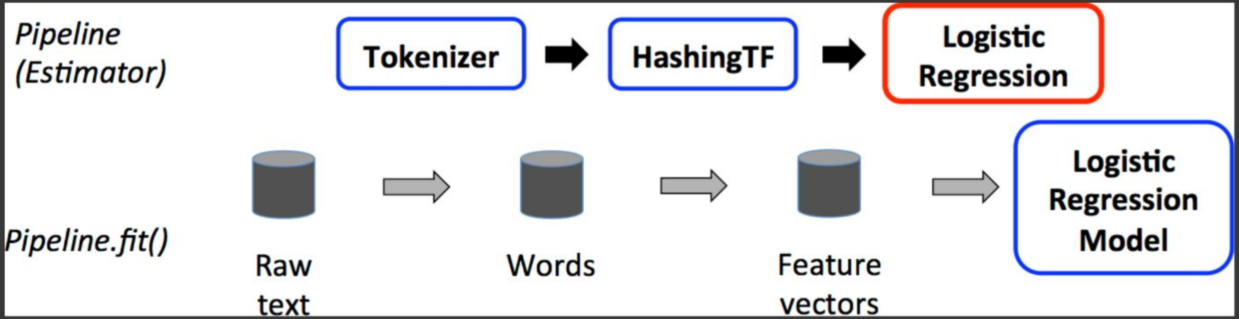
Pipeline Example (Docs)

In machine learning, it's common to run a series of steps for data prep, cleansing, feature engineering, and then ultimately model training (among several other potential steps).

Spark ML Pipelines sequences these steps into an ordered array (or DAG). A Pipeline is specified as a sequence of stages, and each stage is either a **Transformer** or an **Estimator**.

It's often a best practice to save a model or a pipeline to disk for later use.

Below is an example Spark ML Pipeline that shows two Transformers (Tokenizer and HashingTF) and one Estimator (Logistic Regression).



Install Spark Dependencies

Spark ML Feature Engineering

Extracting, transforming and selecting features:

- **Feature Extractors**: TF-IDF, Word2Vec
- **Feature Transformers**: PCA, StopWordsRemover, StringIndexer, OneHotEncoderEstimator, Bucketizer, VectorAssember
- **Feature Selectors**: VectorSlicer, RFormula, ChiSqSelector
- **Locality Sensitive Hashing**: Approximate Nearest Neighbor Search, MinHash for Jaccard Distance

Apache Spark ML

Spark VectorAssembler

Transformer that combines a list of columns into a single vector column.

Useful for combining raw features and features generated by different feature

```
 id | hour | mobile | userFeatures     | clicked
----|------|--------|------------------|---------
 0  | 18   | 1.0    | [0.0, 10.0, 0.5] | 1.0
```

VectorAssembler(inputCols=["hour", "mobile", "userFeatures"], outputCol="features")

```
 id | hour | mobile | userFeatures     | clicked | features
----|------|--------|------------------|---------|-----------------------------
 0  | 18   | 1.0    | [0.0, 10.0, 0.5] | 1.0     | [18.0, 1.0, 0.0, 10.0, 0.5]
```

NFL_Predictions_Jupyter.ipynb

File Edit View Insert Runtime Tools Help All changes saved

Comment Share

+ Code + Text

Connect Editing

Building Model Pipeline

```
[ ] # Prepare string variables so that they can be used by the decision tree algorithm
    # StringIndexer encodes a string column of labels to a column of label indices
    si1 = StringIndexer(inputCol="PlayType", outputCol="PlayType_index")
    si2 = StringIndexer(inputCol="PlayType_lag", outputCol="PlayType_lag_index")
    si3 = StringIndexer(inputCol="PassLength", outputCol="PassLength_index")
    si4 = StringIndexer(inputCol="PassLocation", outputCol="PassLocation_index")
    si5 = StringIndexer(inputCol="RunLocation", outputCol="RunLocation_index")

    target   = 'Yards_Gained'
    features = ['qtr','down','TimeSecs','yrdline100','ydstogo','ydsnet','month_day','PlayType_lag_index']

    #encode the Label column: feature indexer
    fi = StringIndexer(inputCol='Yards_Gained', outputCol='label').fit(training_run)

    # Pipelines API requires that input variables are passed in  a vector
    va  = VectorAssembler(inputCols=features, outputCol="features")
```

```
[ ] # run the algorithm and build model taking the default settings
    rfr = RandomForestRegressor(featuresCol="label", labelCol=target, predictionCol="prediction", maxDepth=5, maxBins=350, seed=12345)
    gbr = GBTRegressor(featuresCol="features", labelCol=target, predictionCol="prediction", maxDepth=5, maxBins=350, seed=12345)

    # Convert indexed labels back to original labels, label converter
    labelConverter = IndexToString(inputCol="prediction", outputCol="predictedLabel", labels=fi.labels)
```

Training the Model

```
  ] # Build the machine learning pipeline
    pipeline_run  = Pipeline(stages=[si2, fi, va, gbr, labelConverter])

    # Build model.
```

ML Algorithms

Classification Techniques

- Logistic regression
  - Binomial logistic regression
  - Multinomial logistic regression
- Decision tree classifier
- Random forest classifier
- Gradient-boosted tree classifier
- Multilayer perceptron classifier
- Linear Support Vector Machine
- One-vs-Rest classifier (a.k.a. One-vs-All)
- Naive Bayes

Regression Techniques

- Linear regression
- Generalized linear regression
  - Available families
- Decision tree regression
- Random forest regression
- Gradient-boosted tree regression
- Survival regression
- Isotonic regression

Mixed Techniques

- Decision trees
  - Inputs and Outputs
    - Input Columns
    - Output Columns
- Random Forests
  - Inputs and Outputs
    - Input Columns
    - Output Columns
- Gradient-Boosted Trees (GBTs)
  - Inputs and Outputs
    - Input Columns
    - Output Columns

Apache Spark ML

Algorithms

Clustering Techniques

- K-means
  - Input Columns
  - Output Columns
- Latent Dirichlet allocation (LDA)
- Bisecting k-means
- Gaussian Mixture Model (GMM)
  - Input Columns
  - Output Columns

Collaborative Filtering / Recommendation Techniques

- Explicit vs. implicit feedback
- Scaling of the regularization parameter
- Cold-start strategy

Advanced Algorithms

- FP-Growth (Pattern Mining)
- PrefixSpan

Optimization of linear methods (developer)

- Limited-memory BFGS (L-BFGS)
- Normal equation solver for weighted least squares
- Iteratively reweighted least squares (IRLS)

Apache Spark ML

Model Selection and Hyperparameter Tuning

Techniques:

- Model selection (hyperparameter tuning)
    - ParamGridBuilder - Used to construct the parameter grid. By default, sets of parameters from the parameter grid are evaluated in serial. Parameter evaluation can be done in parallel by setting parallelism with a value of 2 or more. (10 should be sufficient for most clusters).
- Cross-Validation - Splits the dataset into a set of folds which are used as separate training and test datasets. For example, with k=3 folds, CrossValidator will generate 3 (training, test) dataset pairs, each of which uses 2/3 of the data for training and 1/3 for testing.
- Train-Validation Split - Only evaluates each combination of parameters once, as opposed to k times in the case of CrossValidator. It is, therefore, less expensive, but will not produce as reliable results.

Best Practices: Dev

- Understand how Spark works (this takes time):
    - Spark transformations create new RDDs, but not executed until action is called (lazy loading)
    - Data is distributed, so predefine your partitions or understand how your computations are executed
- Use schema inference (if speed is not critical)
- Scala, Java, Python, R… Which one to use?
- Read release notes - APIs change frequently
- Spark is memory sensitive (dev against small samples)
- Cache/persist commonly used data
- Use Broadcast variables (read-only variable cached on each machine rather than shipping a copy of it with tasks)

Why is my code slow…

Memory Intensive:

- Loading data into memory
- Training Spark (in-memory) models
- Preprocessing data
- Training deep learning models
- Performing feature engineering
- Running clustering algorithms

CPU Intensive:

- Training machine learning models
- Performing hyperparameter tuning
- Running simulations
- Conducting Monte Carlo analysis
- Performing optimization algorithms

I/O Intensive:

- Loading and storing data
- Reading and writing to disk
- Transfer of data over the network
- Distributed shuffles & data sharing
- Running distributed training and data processing

Best Practices: Tuning

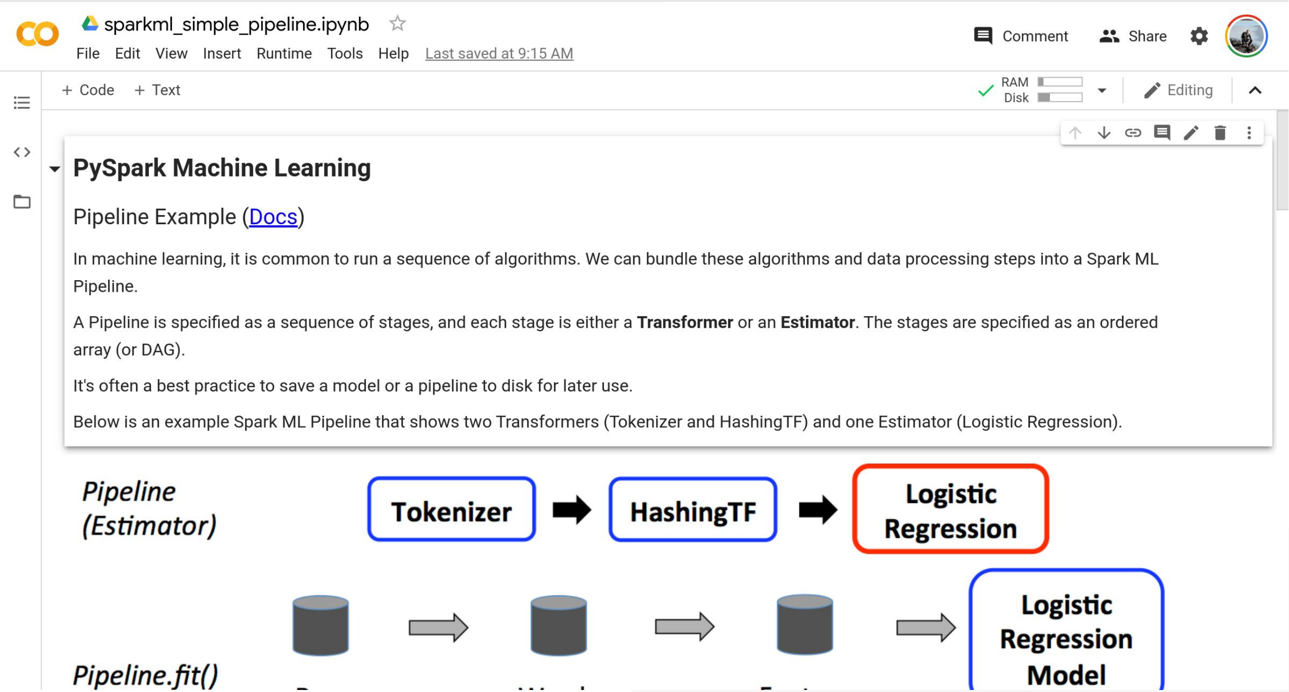
- Avoid .collect() where possible
  - use .take(x) instead, or write results directly to HDFS, Hive, etc.
- Executor-memory
  - Anything over ~20GB could cause garbage collection issues
  - Increase num-executors instead
- Num-executors
  - Do not exceed your total core count
  - Spark History UI -> Exectors -> Grid
- Enable Dynamic Allocation (disabled by default)
  - Dynamically adjusts the resources your application occupies based on the workload

Demos

Demos

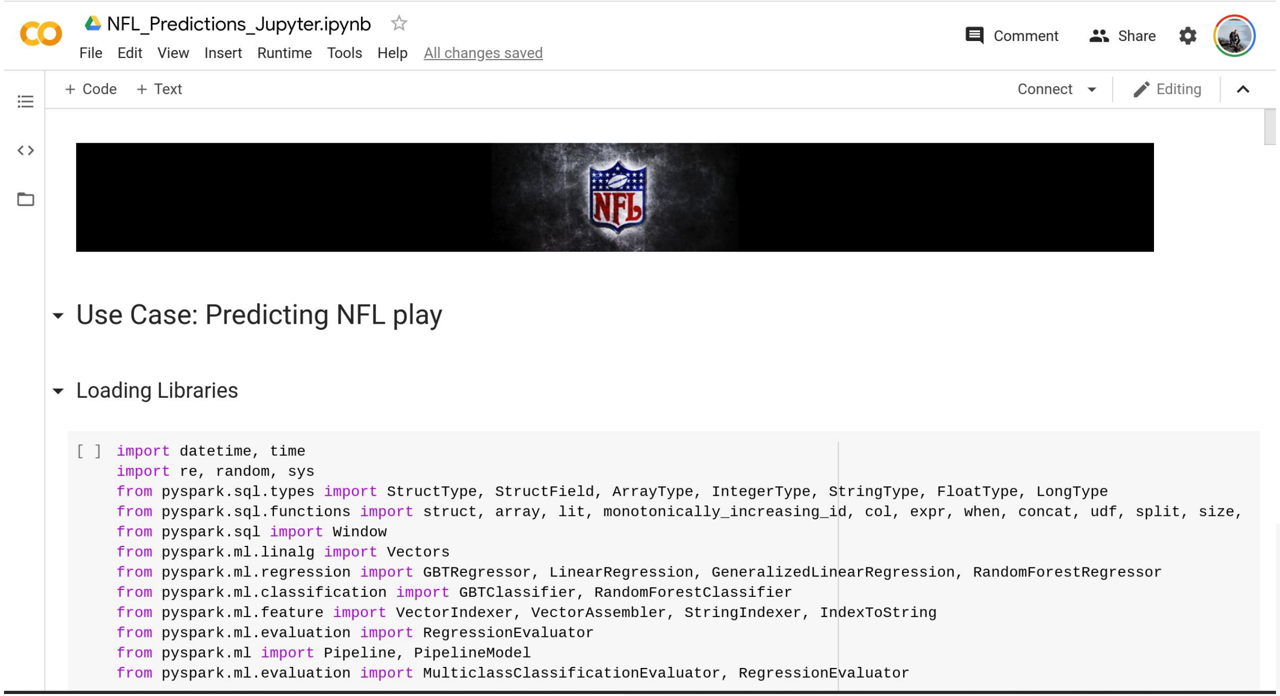
[Google Colab](#)

[ML Pipelines Basic Example](#)



Demos

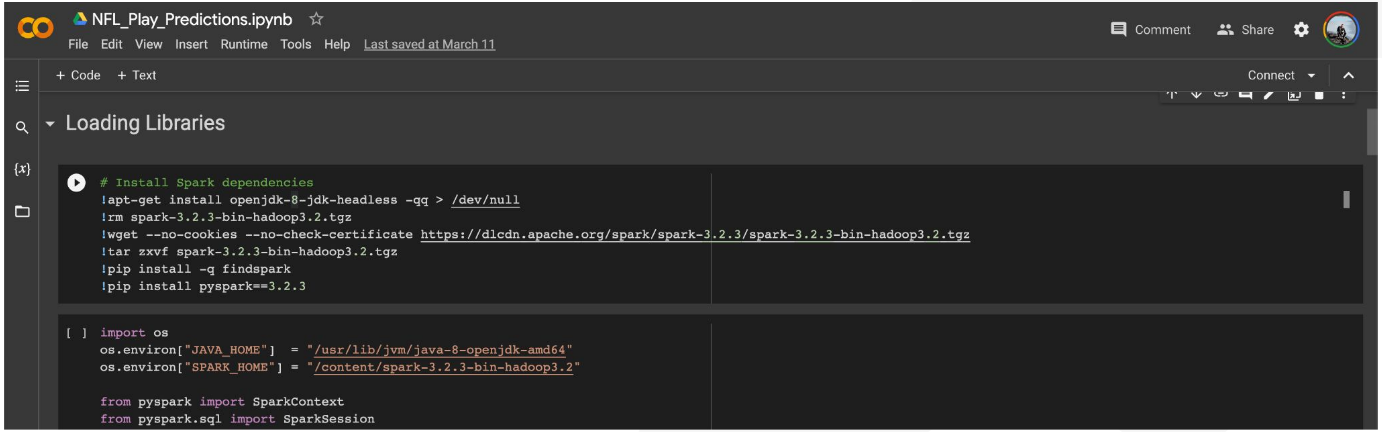
Google Colab
NFL Predictions Notebook



Google Cloud

Spark Environments

Colab Notebooks or local notebooks with custom installation (Dev)



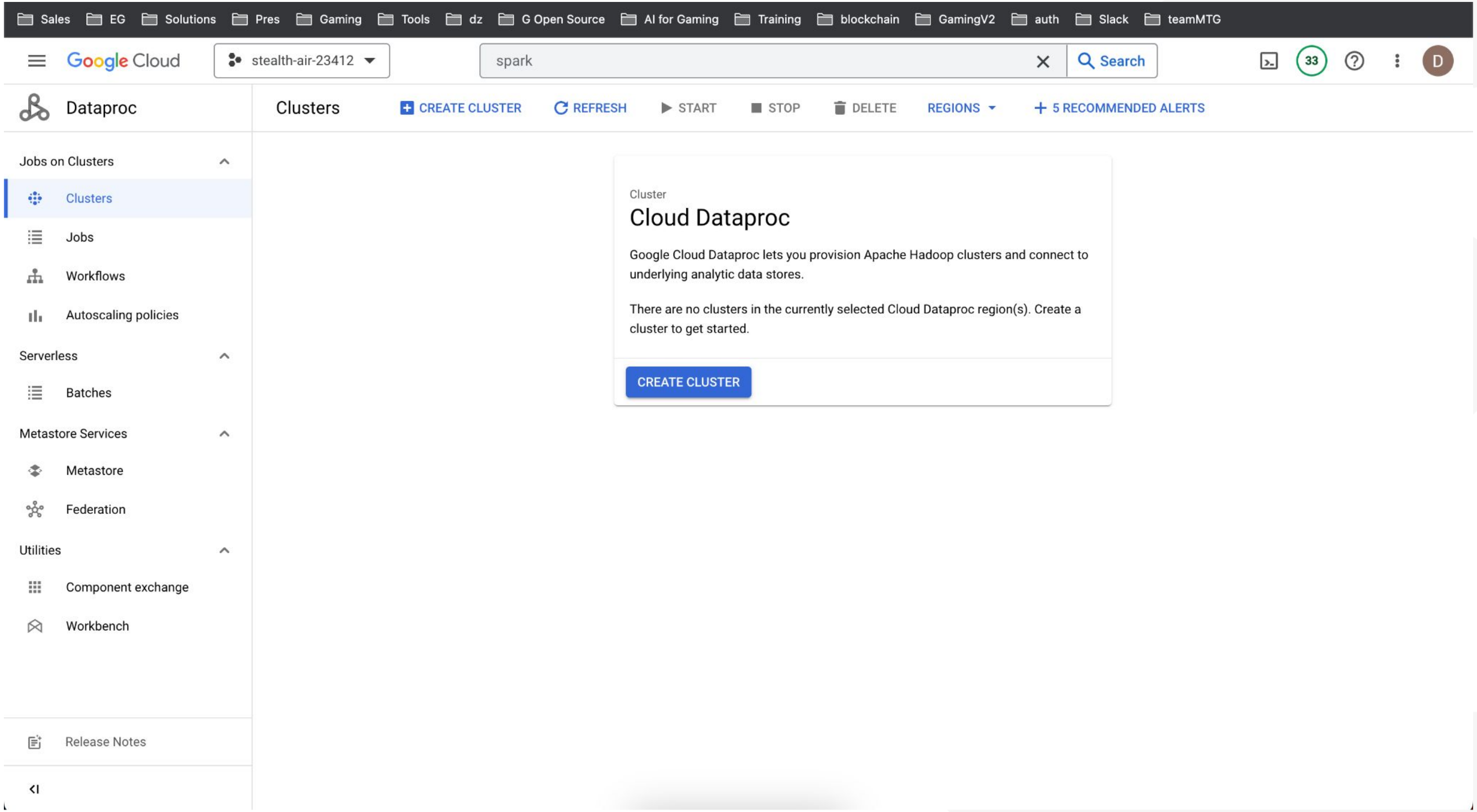
Docker (Dev)

docker run -it -p 8888:8888 --name notebook jupyter/all-spark-notebook

Cloud Environments (Dev and Prod)

Google Dataproc ● Apache Spark on Amazon EMR ● Apache Spark on Azure Databricks

Spark Environments



Thanks!

Advanced Big Data

Dan Zaratsian

AI/ML Architect, Gaming Solutions @ Google

d.zaratsian@gmail.com

https://github.com/zaratsian

Google Cloud

Misc Slides - Appendix

Apache Spark - Example

```
mm_season = spark.read.load(
        "hdfs://sandbox.hortonworks.com:8020/tmp/marchmadness/SeasonResults/SeasonResults.csv",
        format="csv",
        header=True)

mm_season.show()

mm_season.count()

mm_season.dtypes

mm_season.createOrReplaceTempView('mm_season_sql')

spark.sql('''SELECT * FROM mm_season_sql ''').show(10)
```

Apache Spark - Recommendations

1. Caching:
- MEMORY_ONLY: (default/recommended) Store RDD as deserialized objects in JVM Heap
- MEMORY_ONLY_SER: (2nd option) Store RDD as serialized Kryo objects. Trade CPU time for memory savings
- MEMORY_AND_DISK: Spill to disk if can't fit in memory
- MEMORY_AND_DISK_SER: Spill serialized RDD to disk if it can't fit in memory

2. Data Serialization Performance:
- Reduces data size, so less data transfer
- Use Kyro over Java (Kyro is up to 10x faster)
- conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
- sparkConf.set("spark.sql.tungsten.enabled", "true")
- sparkConf.set("spark.io.compression.codec", "snappy")
- sparkConf.set("spark.rdd.compress", "true")

3. Memory and Garbage Collection Tuning:
- GC is a problem for Spark apps which churn RDDs
- Measure time spent in GC by logging: -verbose:gc –XX:+PrintGCDetails –XX:+PrintGCTimeStamps
- If there's excessive GC per task, use the MEMORY_ONLY_SER storage level to limit just one object per RDD partition (one byte array) and reduce the spark.storage.memoryFraction value from 0.6 to 0.5 or less.

4. Set Correct Level of Parallelism:
- set spark.default.parallelism = 2-3 tasks per CPU core in your cluster
- Normally 3 - 6 executors per node is a reasonable, depends on the CPU cores and memory size per executor
- sparkConf.set("spark.cores.max", "4")
- 5 or less cores per executor (per node) (ie. 24-core node could run 24/4cores = 6 executors)

Best Practices: Tuning

- Avoid .collect() where possible
  - use .take(x) instead, or write results directly to HDFS, Hive, etc.
- Executor-memory
  - Anything over ~20GB could cause garbage collection issues
  - Increase num-executors instead
- Num-executors
  - Do not exceed your total core count
  - Spark History UI -> Exectors -> Grid
- Enable Dynamic Allocation (disabled by default)
  - Dynamically adjusts the resources your application occupies based on the workload

Configuring Executors

- `executor-memory`
  - Should be between 8GB and 64GB
- `executor-cores`
  - At least 2, max 4
- `num-executors`
  - This is the most flexible
  - If caching data, desirable to have datasize * 2 as the total application memory

- EXAMPLE: YARN nodes with 128GB and 16 cores available would support a relatively common 16GB-memory / 2-core executor size
  - If caching a 100GB dataset, 13 executors could be ideal

Which Storage Level to Choose?

- If the RDD fits in memory, use the default MEMORY_ONLY
- If RDDs are too big, try MEMORY_ONLY_SER with a fast serialization library (Scala only)
- If the RDDs are still too big:
  - Consider the time to compute this RDD from parent RDD vs the time to load it from disk
  - Re-computing an RDD may sometimes be faster than reading it from disk
- Replicated storage is good for fast fault recovery, but…
  - Usually this is overkill, and not a good idea if you're using a lot of data relative to total memory
- For DataFrames, use `cache()` instead of `persist(StorageLevel)`

When things go wrong…

- Where to look:
  - yarn application –list (get the list of running application)
  - yarn logs -applicationId <app_id>
  - Check Spark: http://<host>:8088/proxy/<job_id>/environment/
- Common Issues:
  - Submitted a Job but nothing happens
    - Job stays in accepted state when allocated more memory/cores than is available
    - May need to kill unresponsive/stale jobs
  - Insufficient HDFS access
    - Grant user/group necessary HDFS access
    - May lead to failure such as:
      `“Loading data to table default.testtable Failed with exception Unable to move sourcehdfs://red1:8020/tmp/hive-spark/hive_2015-03-04_12-45-42_404_3643812080461575333-1/-ext-10000/kv1.txt to destination hdfs://red1:8020/apps/hive/warehouse/testtable/kv1.txt”`
  - **Wrong host in Beeline, shows error as invalid URL**
    - `“Error: Invalid URL: jdbc:hive2://localhost:10001 (state=08S01,code=0)”`
  - **Error about closed SQLContext, restart Thirft Server**