Large-Scale Machine Learning with Apache Spark - Comprehensive Exam Notes

Table of Contents

---

1. Introduction to Large-Scale ML {#introduction}

Key Question

How can we apply machine learning algorithms to massive datasets that don't fit in memory using distributed computing frameworks like MapReduce and Spark?

Two Main Challenges

- **Big Data**: Input dataset is too large to hold in main memory
- **Big Model**: The model itself is too large to hold in main memory

Parallelization Strategies

- **Data Parallelism**: Distribute data across multiple workers
- **Model Parallelism**: Distribute model parameters across multiple workers

---

2. Linear Regression Fundamentals {#linear-regression}

Mathematical Foundation

Linear regression builds a linear function that maps input data to predicted values:

Basic Formula: $\hat{y} = \mathbf{w} \cdot \mathbf{x} + \mathbf{b} = \mathbf{w}^T\mathbf{x} + \mathbf{b}$

Where:

- $\mathbf{x}$ = input data (features)
- $\mathbf{w}$ = vector of parameters (weights)
- $\mathbf{b}$ = bias term
- $\hat{y}$ = predicted value
- $\mathbf{w}^T$ = transpose of $\mathbf{w}$ (column vector → row vector)

Simplified Form (without bias)

For simplicity: $\hat{y} = \mathbf{w}^T\mathbf{x}$

Mean Squared Error (MSE)

For a set of m samples:

MSE $= (1/m) \times \Sigma_{j=1}^{m} (\hat{y}^{(j)} - y^{(j)})^2 = (1/m) \times \Sigma_{j=1}^{m} (\mathbf{w}^T\mathbf{x}^{(j)} - y^{(j)})^2$

Training Objective

Find weights **w** that minimize the MSE.

Important Note: RMSE and MSE make no difference for minimization purposes - minimizing one means minimizing the other.

---

3. Gradient Descent {#gradient-descent}

Algorithm Steps

1. **Initialize**: Start at a random point
2. **Repeat**:
   - Determine descent direction
   - Choose step size
   - Update parameters
3. **Stop**: When stopping criterion is satisfied

Gradient Calculation

The gradient of MSE for linear regression:

∇**MSE** $= \Sigma_{j=1}^{m} (\mathbf{w}^T\mathbf{x}^{(j)} - y^{(j)})\, \mathbf{x}^{(j)}$

Weight Update Rule

At timestep i+1:

$$\mathbf{w}_{i+1} = \mathbf{w}_i - \alpha \times \Sigma_{j=1}^{m} (\mathbf{w}_i^T\mathbf{x}^{(j)} - y^{(j)})\, \mathbf{x}^{(j)}$$

Where α is the learning rate (step size).

---

4. Distributed Computing Challenges {#distributed-challenges}

Data Parallelism Implementation

- Compute summands in parallel across workers
- Each worker receives all weights $\mathbf{w}_i$ at every iteration
- Example: m = 6 samples, 3 workers → each worker processes 2 samples

Major Bottleneck

Communication Overhead: Transferring weights $\mathbf{w}_i$ between driver (parameter server) and workers at each iteration.

Solutions

- Reduce communication frequency
- Use more efficient gradient approximation methods
- Implement Stochastic Gradient Descent

---

5. Stochastic Gradient Descent (SGD) {#sgd}

Motivation

- Traditional gradient descent processes all m samples per iteration
- The gradient is an expectation that can be approximated using smaller samples
- If dataset is highly redundant, gradient in first half ≈ gradient in second half

SGD Approach

- Update model with only one sample instead of m samples
- Can use mini-batches (small batch sizes)

Mini-batch Example

For batch size of 16:

$\mathbf{w}_{i+1} = \mathbf{w}_i - \alpha_i \times \Sigma_{j=1}^{16} (\mathbf{w}_i^T\mathbf{x}^{(j)} - y^{(j)}) \mathbf{x}^{(j)}$

Advantages

- Reduced computational cost per iteration
- Better scalability for large datasets
- Faster convergence in practice

---

6. Spark MLlib Overview {#mllib-overview}

Conceptual Similarity

- MLlib is conceptually similar to Scikit-Learn
- Leverages Spark's powerful distributed computing engine
- Designed for large-scale machine learning

Key Advantage

Seamless integration with Spark's distributed computing capabilities while maintaining familiar ML workflow patterns.

---

7. MLlib Core Concepts {#mllib-concepts}

1. Transformers

- **Function**: Convert raw data in some way
- **Examples**:
  - Create interaction variables
  - Convert categorical strings to numerical values
- **Usage**: Pre-processing and feature engineering
- **Input/Output**: DataFrame → DataFrame

2. Estimators

- **Function**: When provided with data, result in transformers
- **Purpose**: Algorithms used to train models
- **Examples**: Classification and regression algorithms

3. Evaluators

- **Function**: Evaluate model performance
- **Metrics**: Accuracy, ROC, AUC, etc.
- **Usage**: Model selection and validation

4. Pipeline

- **Level**: MLlib's highest-level data type
- **Components**: Transformers, estimators, and evaluators as stages
- **Similarity**: Similar to Scikit-Learn's pipeline API
- **Benefit**: Streamlined workflow management

---

8. Practical Implementation {#practical-implementation}

Dataset Requirements

MLlib requires:

- **Labels**: Type Double
- **Features**: Type Vector[Double]

Feature Engineering with RFormula

RFormula Operators

- `~`: Separate target and terms
- `+`: Concatenate terms
- `-`: Remove terms
- `:`: Interaction terms
- `.`: All columns except target/dependent variable

Example Usage

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

Model Training Workflow

1. Data Preparation

python

```python
# Fit and transform data
fittedRF = supervised.fit(df)
preparedDF = fittedRF.transform(df)

# Split data
train, test = preparedDF.randomSplit([0.7, 0.3])
```

2. Model Creation

python

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

3. Model Training

python

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

4. Prediction

python

```python
predictions = fittedLR.transform(train)
```

Pipeline Implementation

Pipeline Setup

python

```python
from pyspark.ml import Pipeline

# Define stages
rForm = RFormula()
dt = DecisionTreeClassifier().setLabelCol("label").setFeaturesCol("features")
stages = [rForm, dt]

# Create pipeline
pipeline = Pipeline().setStages(stages)
```

Hyperparameter Tuning

Grid Search Setup

python

```python
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()
```

Evaluation Setup

python

```python
from pyspark.ml.evaluation import BinaryClassificationEvaluator

evaluator = BinaryClassificationEvaluator()\
  .setMetricName("areaUnderROC")\
  .setRawPredictionCol("prediction")\
  .setLabelCol("label")
```

Validation Strategy

python

```python
from pyspark.ml.tuning import TrainValidationSplit

tvs = TrainValidationSplit()\
    .setTrainRatio(0.75)\
    .setEstimatorParamMaps(params)\
    .setEstimator(pipeline)\
    .setEvaluator(evaluator)
```

Model Training and Evaluation

python

```python
# Train model
tvsFitted = tvs.fit(train)

# Evaluate on test set
test_score = evaluator.evaluate(tvsFitted.transform(test))
# or
test_score = evaluator.evaluate(tvsFitted.bestModel.transform(test))
```

---

9. Advanced Topics {#advanced-topics}

Model Persistence

python

```python
# Save model
tvsFitted.bestModel.write().save("path/to/model")

# Load model
from pyspark.ml.pipeline import PipelineModel
myModel = PipelineModel.load("path/to/model")
```

Multiple Algorithm Comparison

- Single pipeline typically includes one ML algorithm
- For multiple competing algorithms (e.g., Decision Tree vs Logistic Regression)
- Need to specify multiple pipelines manually

Data Splitting Strategies

- **Training/Testing Split**: Direct split of data
- **Training/Validation/Testing**: Three-way split
- **K-fold Cross-validation**: More robust validation

Available Algorithms

MLlib supports various algorithms:

- DecisionTreeClassifier
- LogisticRegression
- NaiveBayes
- Random Forest
- Gradient Boosted Trees
- And more...

---

10. Key Formulas and Equations {#formulas}

Linear Regression

- **Prediction**: $\hat{y} = \mathbf{w}^T\mathbf{x} + \mathbf{b}$
- **MSE**: MSE = $(1/m) \times \Sigma_{j=1}^{m} (\hat{y}^{(j)} - y^{(j)})^2$

Gradient Descent

- **Gradient**: $\nabla MSE = \Sigma_{j=1}^{m} (\mathbf{w}^T\mathbf{x}^{(j)} - y^{(j)})\ \mathbf{x}^{(j)}$
- **Update Rule**: $\mathbf{w}_{i+1} = \mathbf{w}_i - \alpha \times \nabla MSE$

Stochastic Gradient Descent

- **Mini-batch Update**: $\mathbf{w}_{i+1} = \mathbf{w}_i - \alpha \times \Sigma_{j=1}^{b} (\mathbf{w}_i^T\mathbf{x}^{(j)} - y^{(j)})\ \mathbf{x}^{(j)}$

Where b is the batch size.

Study Tips for Exam

Theoretical Understanding

1. **Understand the mathematical foundations** of linear regression and gradient descent
2. **Know the difference** between batch and stochastic gradient descent
3. **Understand distributed computing challenges** and solutions

Practical Implementation

1. **Memorize key MLlib components**: Transformers, Estimators, Evaluators, Pipelines
2. **Understand RFormula syntax** and operators
3. **Know the typical workflow**: Data preparation → Model creation → Training → Evaluation
4. **Understand hyperparameter tuning** process with grid search and validation

Code Patterns

1. **Pipeline creation** and stage setup
2. **Model training and evaluation** workflow
3. **Model persistence** and loading
4. **Data splitting** strategies

Common Exam Topics

- Mathematical derivations of gradient descent
- Distributed computing bottlenecks and solutions
- MLlib component relationships
- Pipeline implementation
- Hyperparameter tuning strategies
- Model evaluation metrics

Quick Reference

Important Classes

- `RFormula`: Feature engineering
- `DecisionTreeClassifier`: Classification algorithm
- `Pipeline`: Workflow management
- `ParamGridBuilder`: Hyperparameter grid creation
- `TrainValidationSplit`: Model validation
- `BinaryClassificationEvaluator`: Model evaluation

Key Methods

- `.fit()`: Train model/transformer

- `.transform()`: Apply transformation/prediction
- `.randomSplit()`: Split data
- `.setStages()`: Set pipeline stages
- `.build()`: Build parameter grid
- `.evaluate()`: Evaluate model performance