Data 603 – Big Data Platforms

Lecture 10

Apache Spark MLlib - Part 2

Hyperparameter Tuning

Hyper parameters vs Parameters

- A hyperparameter is an attribute defined about the model prior to training.
- It is not learned during the training process. Parameters are learned during the training process.

Tree-Based Models

Tree-based models

- Decision trees
- Gradient boosted trees
- Random forests

Decision Trees

Decision Trees are …

- Sequence of if-then-else rules learned from the data
- Can be used for classification or regression
- Relatively fast to build
- Highly interpretable
- Scale-invariant – standardizing or scaling the numeric features do not change the performance of the tree.
- Series of if-then-else rules learning from the data for classification or regression tasks.
- Not always the most accurate model
- Sensitive to the specific data they were trained on

Decision Trees

- The depth of a decision tree is the longest path from the root node to any given leaf node.
- Trees that are very deep are prone to overfitting, or memorizing noise in the training data set.
- Trees that are too shallow will underfit the data set (could pick up more signal from the data).
- For decision trees there is no need to worry about standardizing or scaling the input features (no impact on the splits).
  - Note: A care has to be given about how to prepare the categorical features.

Decision Trees

spark.ml decision tree implementation:

- Supports binary and multiclass classification and regression.
- Both continuous and categorical features can be used.
- Partitions data by rows, allowing distributed training with millions or even billions of instances.

Difference between ML and MLlib implementations

- ML supports ML Pipelines
- ML separates Decision Trees for classification vs. regression
- ML uses DataFrame metadata to distinguish continuous and categorical features

References:

- ML implementation: https://spark.apache.org/docs/3.1.1/ml-classification-regression.html#decision-trees
- MLlib implementation: https://spark.apache.org/docs/3.1.1/mllib-decision-tree.html

Decision Trees - Input Columns

Param name	Type(s)	Default	Description
labelCol	Double	"label"	Label to predict
featuresCol	Vector	"features"	Feature vector

Decision Trees - Output Columns

Param name	Type(s)	Default	Description	Notes
predictionCol	Double	"prediction"	Predicted label	
rawPredictionCol	Vector	"rawPrediction"	Vector of length # classes, with the counts of training instance labels at the tree node which makes the prediction	Classification only
probabilityCol	Vector	"probability"	Vector of length # classes equal to rawPrediction normalized to a multinomial distribution	Classification only
varianceCol	Double		The biased sample variance of prediction	Regression only

Decision Trees - Classification

```
from pyspark.ml import Pipeline
from pyspark.ml.classification import DecisionTreeClassifier
from pyspark.ml.feature import StringIndexer, VectorIndexer
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
data = spark.read.format("libsvm").load("data/mllib/sample_libsvm_data.txt")
labelIndexer = StringIndexer(inputCol="label",
outputCol="indexedLabel").fit(data)
# Automatically identify categorical features, and index them.
# We specify maxCategories so features with > 4 distinct values are treated as
continuous.
featureIndexer = \
    VectorIndexer(inputCol="features", outputCol="indexedFeatures",
          maxCategories=4).fit(data)

(trainingData, testData) = data.randomSplit([0.7, 0.3])
dt = DecisionTreeClassifier(labelCol="indexedLabel",
     featuresCol="indexedFeatures")
pipeline = Pipeline(stages=[labelIndexer, featureIndexer, dt])
model = pipeline.fit(trainingData)
```

Decision Trees - Regression

```
from pyspark.ml.regression import DecisionTreeRegressor
dt = DecisionTreeRegressor(labelCol="price")

# Filter for just numeric columns (and exclude price, our label)
numericCols = [field for (field, dataType) in trainDF.dtypes
  if ((dataType == "double") & (field != "price"))]

# Combine output of StringIndexer defined above and numeric columns
assemblerInputs = indexOutputCols + numericCols
vecAssembler = VectorAssembler(inputCols=assemblerInputs,
   outputCol="features")

# Combine stages into pipeline
stages = [stringIndexer, vecAssembler, dt]
pipeline = Pipeline(stages=stages)
pipelineModel = pipeline.fit(trainDF)
```

Decision Trees - maxBins Parameter

maxBins Parameter

- Determines the number of bins into which the continuous features are discretized (split)
  - Discretization step is crucial for performing distributed training.
  - There is no maxBins parameter in *scikit-learn* because all of the data and the model reside on a single machine.
  - In Spark, workers have all the columns of the data, but only a subset of rows. It is important to use the same split values (features and values to split on) from the common discretization set up at training time.
  - Every worker has to compute summary statistics for every feature possible split point which are then aggregated across the workers.
  - maxBins need to be large enough to handle the discretization of the categorical columns (default 32).

Decision Trees - Numeric vs Categorical Features

- It is possible to split on the same feature more than once, but at different split values
- Difference between splits on numeric features versus categorical features
  - For numeric features it checks if the value is less than or equal to the threshold
  - For categorical features it checks if the value is in the set or not

Decision Trees

```
# Printing if-then-else rules
dtModel = piplineModel.stages[-1]
print(dtModel.toDebugString)

# Extracting feature importance scores from the model
import pandas as pd
featureImp = pd.DataFrame(
    list(zip(vecAssembler.getInputCols(),
             dtModel.featureImportances)),
columns=["feature", "importance"])
featureImp.sort_values(by="importance", ascending=False)
```

Tree Ensembles

- DataFrame API supports two major tree ensemble algorithms
  - Random Forests
  - Gradient-Boosted Trees
  - Both use spark.ml decision trees as their base models
- Difference between ML and MLlib implementations
  - Support for DataFrames and ML Pipelines
  - Separation of classification vs. regression
  - Use of DataFrame metadata to distinguish continuous and categorical features
  - More functionality for random forests: estimates of feature importance, as well as the predicted probability of each class (a.k.a. class conditional probabilities) for classification.

References:

- https://spark.apache.org/docs/3.1.1/ml-classification-regression.html#tree-ensembles
- https://spark.apache.org/docs/3.1.1/mllib-ensembles.html

Tree Ensembles - GBT vs Random Forests

- GBTs train one tree at a time
  - Take longer to train than random forests. Random Forests can train multiple trees in parallel.
  - On the other hand, it is often reasonable to use smaller (shallower) trees with GBTs than with Random Forests, and training smaller trees takes less time.
- Random Forests can be less prone to overfitting. Training more trees in a Random Forest reduces the likelihood of overfitting, but training more trees with GBTs increases the likelihood of overfitting.
- Random Forests can be easier to tune since performance improves monotonically with the number of trees (whereas performance can start to decrease for GBTs if the number of trees grows too large).

Random Forests

- Random forests are an ensemble of decision trees.
- In statistics and machine learning, ensemble methods use multiple learning algorithms to obtain better predictive performance than could be obtained from any of the constituent learning algorithms alone. (https://en.wikipedia.org/wiki/Ensemble_learning)
- Building many models and combining/averaging their prediction produce more robust results than the ones produced by an individual model.

Random Forests - Bagging

Bootstrapping samples by rows

- Bootstrapping: a technique for simulating new data by sampling with replacement from the original data.
- Each decision tree is trained on a different bootstrap sample of the data set
  - This produces slightly different decision trees.
  - The predictions are aggregated => *bootstrap aggregating*, or *bagging*
  - Each tree samples the same number of data points controlled through the *subsamplingRate* parameter

Random Forests - Bagging

- Decision trees are sensitive to the specific data sets on which they are trained.
- Bagging is the application of the Bootstrap procedure for high-variance algorithms.
- Bagging of Classification And Regression Trees (CART)
  1. Create many random sub-samples of the dataset with replacement
  2. Train a CART model using each sample
  3. For a new dataset, calculate the average prediction from each model

Random Forests - Random Feature Selection

Random feature selection by columns

- Issue with the bagging: The trees are highly correlated, and learn similar patterns in the data.
- Solution: Each time a split is made, a random subset of the columns are selected.
  - Due to the randomness, each tree will need to be shallow.
  - Each tree learns something different from the data set
  - Combining the collection of “weak” learners into an ensemble makes the forest much more robust than a single decision tree.

Random Forests - Random Feature Selection

A problem with decision trees (CART) is that they are greedy – it uses a greedy algorithm to minimize the error when choosing a variable to split on.

Random forest makes a simple tweak …

- It changes the algorithms for the way sub-trees are learned to reduce the correlation between prediction results.
- In CART, when selecting a split point, the algorithm looks through all variables and values to select the most optimal split-point.
- In Random forest, the learning algorithm is limited to a random sample of features
- Each tree learns something new about the data set. Collection of “weak” learners into an ensemble makes the forest much more robust.

Random Forests

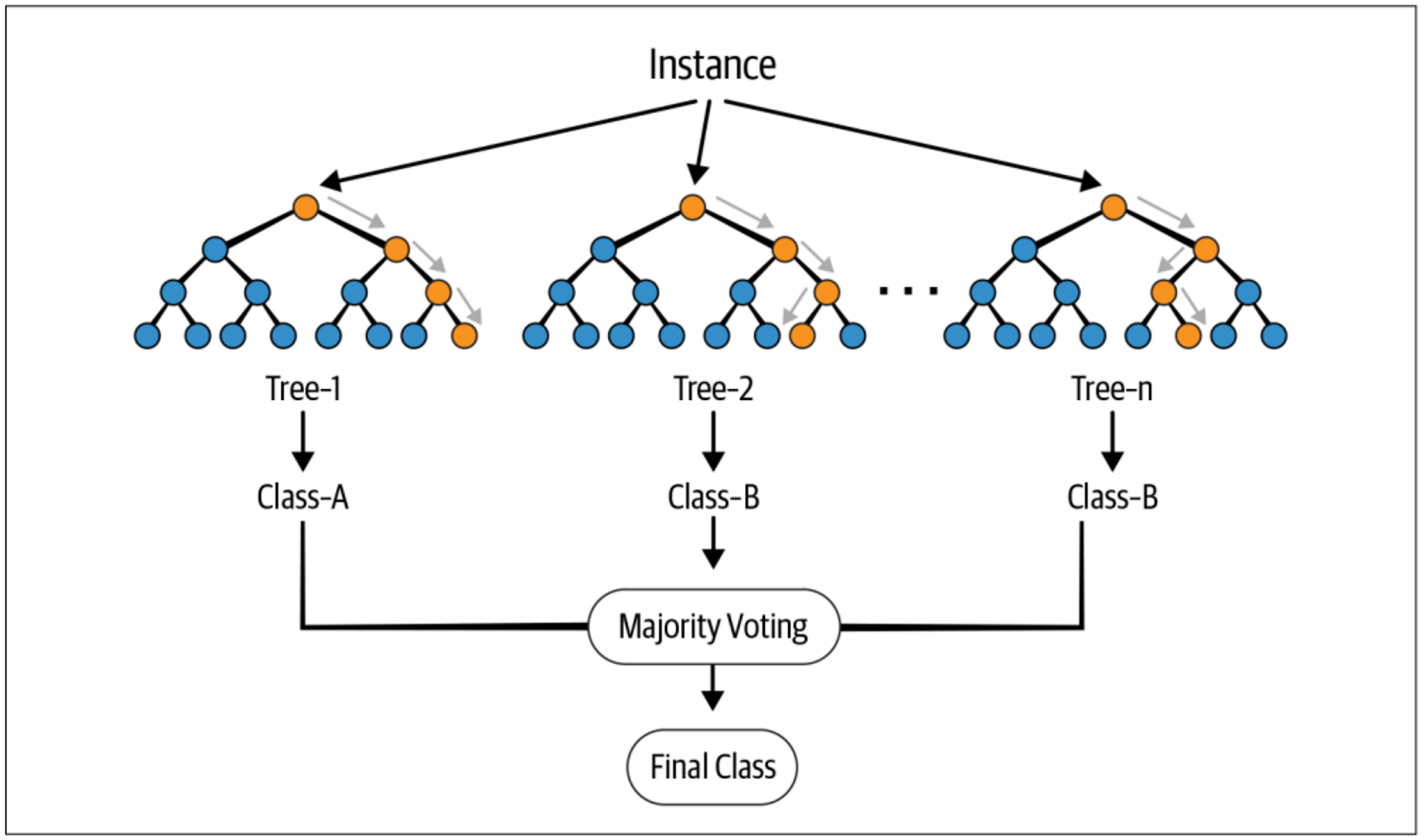


Figure 10-12. Random forest predictions

Random Forests

```
from pyspark.ml.regression import RandomForestRegressor
rf = RandomForestRegressor(labelCol="price", maxBins=40, seed=42)
```

K-Fold Cross-Validation

Hyperparameters for Random Forest

- The number of trees to train
- Maximum depth of the trees

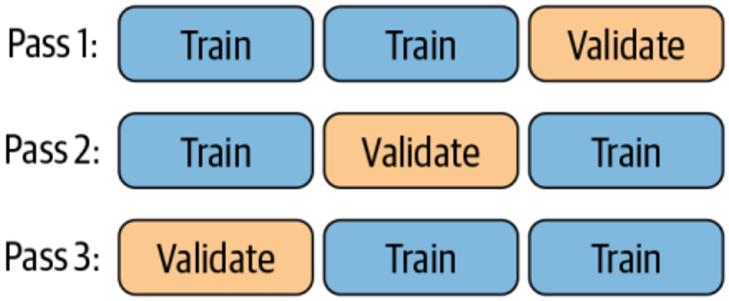
Which data set to use to determine the optimal hyperparameter values?

- Using training data set - Overfit
- Using testing data set – No longer “unseen” data, will not be able to verify how well the model generalizes.
- We can use another data set – Validation data set
  - 60/20/20 split for training, testing and validation
  - Validation data set is used to evaluate the performance to select the best hyperparameter configuration.
  - Issue? 25% of the training data is lost

K-Fold Cross-Validation

Using k-fold cross-validation, we are able to determine the best hyperparameter configuration without losing the training data.

- Split the data into training and testing
- Use the training data for both training and validation
  1. Split the training data into k subsets (folds)
  2. Given a hyperparameter configuration, train the model on k-1 folds and evaluation on the remaining fold.
  3. Repeat the process k times.
  4. Use the average of the performance of the validation as the proxy of how well the model performs with unseen data
  5. Repeat the process for all different parameter configuration to identify the optimal one.



Hyperparameter Search

1. Define the estimator to evaluate.
2. Specify which hyperparameter to vary along with the values using the *ParamGridBuilder.*
3. Define an evaluation to specify which metric to use to compare the various models.
4. Use the *CrossValidator* to perform cross-validation, evaluating each of the various models.

Hyperparameter Search

```
pipeline = Pipeline(stages = [stringIndexer, vecAssembler, rf])

from pyspark.ml.tuning import ParamGridBuilder
paramGrid = (ParamGridBuilder()
     .addGrid(rf.maxDepth, [2, 4, 6])
     .addGrid(rf.numTrees, [10, 100])
     .build())
evaluator = RegressionEvaluator(labelCol="price",
     predictionCol="prediction",
     metricName="rmse")

from pyspark.ml.tuning import CrossValidator
cv = CrossValidator(estimator=pipeline,
           evaluator=evaluator,
           estimatorParamMaps=paramGrid,
           numFolds=3,
           seed=42)
cvModel = cv.fit(trainDF)
list(zip(cvModel.getEstimatorParamMaps(), cvModel.avgMetrics))
```

Optimizing Pipelines

For *CrossValidator*, *parallelism* parameter determines the number of models to train in parallel.

- The value of parallelism should be chosen to maximize parallelism without exceeding cluster resources
- Larger values do not always lead to improved performance
- A value of up to 10 should be sufficient for most clusters.

```
cvModel = cv.setParallelism(4).fit(trainDF)
```

Optimizing Pipelines

Putting the cross-validator inside the pipeline instead of putting the pipeline inside the cross-validator.

- This eliminates the need to remove steps that don't change. E.g. There is no need to reevaluate StringIndexer.

```
cv = CrossValidator(estimator=rf,
       evaluator=evaluator,
       estimatorParamMaps=paramGrid,
       numFolds=3,
       parallelism=4,
       seed=42)
pipeline = Pipeline(stages=[stringIndexer, vecAssembler, cv])
pipelineModel = pipeline.fit(trainDF)
```

Managing Models

End-to-end reproducibility of machine learning solutions - Ability to reproduce …

- The code that generated a model
- The environment used in training
- The data used in training
- The model

Managing Models

- Library versioning
  - Which libraries and which versions of them are used?
  - Using the latest version of library can cause the code to break
- Data evolution
  - Drafts in data (shape of the data and tendency of the data) makes reproducing of the results difficult
- Order of execution
  - Running the cells in order!
  - Checking in the code
- Parallel operations
  - GPUs running operations in parallel can exacerbate the possibility of producing different results, leading to nondeterministic outputs. This is because the order of the execution is not always guaranteed.

MLflow

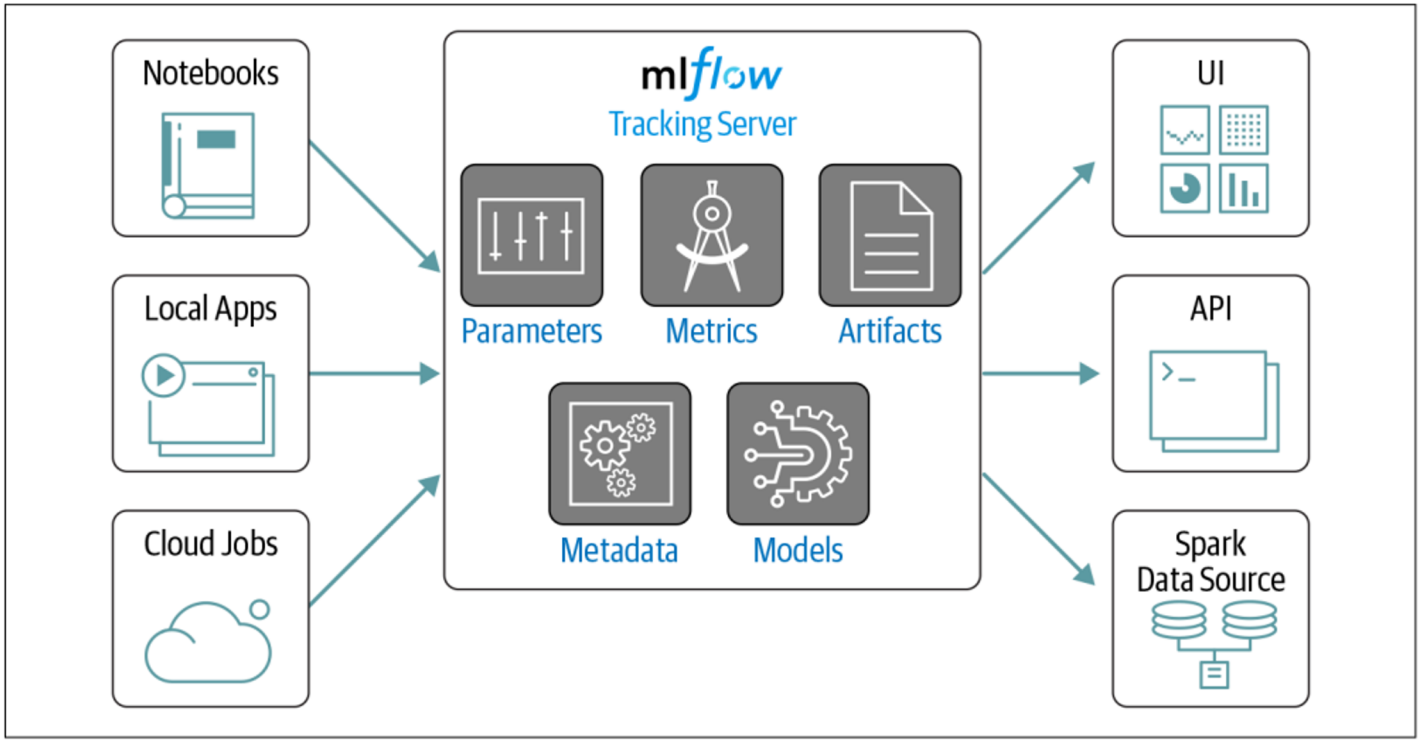
MLflow: An open source MLOps platform, which helps developers to manage, reproduce and share models. It has four components:

- Tracking – provides APIs to record parameters, metrics, code versions, models, and artifacts (plots, text, etc)
- Projects – A standardized format to package data science projects and their dependencies to run on other platforms. Helps to manage the model training process
- Models – A standardized format to package models to deploy to diverse execution environments. It provides a consistent API for loading and applying models, regardless of the algorithms or library used to build the model
- Registry – A repository to keep track of model lineage, model versions, stage transitions, and annotations

MLflow - Tracking

- Logging API agnostic to the libraries and environments that actually do the training.
  - Organized around the concept of runs (execution of data science code).
- Runs are aggregated into experiments
  - Many runs can be part of a given experiment.
- Things that can be logged:
  - Parameters: key/value pairs including hyperparameters (e.g. num_trees, max_depth)
  - Metrics: Numeric values (e.g. RMSE or accuracy values)
  - Artifacts: Files, data, and models (e.g. matplotlib images, Parquet files)
  - Metadata: Information about the run (e.g. source code and version of the code)
  - Models: The model that has been trained.

MLflow - Tracking



Links and Labs

Labs

- Binary Classification
- Automated MLflow tracking in Mllib
- Distributed Hyperopt + Automated MLflow Tracking

Links

- https://medium.com/data-design/visiting-categorical-features-and-encoding-in-decision-trees-53400fa65931

Questions