

ML Tuning: model selection and hyperparameter tuning

»

This section describes how to use MLlib's tooling for tuning ML algorithms and Pipelines. Built-in Cross-Validation and other tooling allow users to optimize hyperparameters in algorithms and Pipelines.

Table of contents

- [Model selection \(a.k.a. hyperparameter tuning\)](#)
- [Cross-Validation](#)
- [Train-Validation Split](#)

Model selection (a.k.a. hyperparameter tuning)

An important task in ML is *model selection*, or using data to find the best model or parameters for a given task. This is also called *tuning*. Tuning may be done for individual Estimators such as `LogisticRegression`, or for entire Pipelines which include multiple algorithms, featurization, and other steps. Users can tune an entire Pipeline at once, rather than tuning each element in the Pipeline separately.

MLlib supports model selection using tools such as `CrossValidator` and `TrainValidationSplit`. These tools require the following items:

- **Estimator**: algorithm or Pipeline to tune
- Set of ParamMaps: parameters to choose from, sometimes called a “parameter grid” to search over
- **Evaluator**: metric to measure how well a fitted Model does on held-out test data

At a high level, these model selection tools work as follows:

- They split the input data into separate training and test datasets.
- For each (training, test) pair, they iterate through the set of ParamMaps:
 - For each ParamMap, they fit the Estimator using those parameters, get the fitted Model, and evaluate the Model's performance using the Evaluator.
- They select the Model produced by the best-performing set of parameters.

The Evaluator can be a `RegressionEvaluator` for regression problems, a `BinaryClassificationEvaluator` for binary data, or a `MulticlassClassificationEvaluator` for multiclass problems. The default metric used to choose the best ParamMap can be overridden by the `setMetricName` method in each of these evaluators.

To help construct the parameter grid, users can use the `ParamGridBuilder` utility.

Cross-Validation

`CrossValidator` begins by splitting the dataset into a set of *folds* which are used as separate training and test datasets. E.g., 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. To evaluate a particular ParamMap, `CrossValidator` computes the average evaluation metric for the 3 Models produced by fitting the Estimator on the 3 different (training, test) dataset pairs.

After identifying the best ParamMap, CrossValidator finally re-fits the Estimator using the best ParamMap and the entire dataset.

Examples: model selection via cross-validation

The following example demonstrates using CrossValidator to select from a grid of parameters.

Note that cross-validation over a grid of parameters is expensive. E.g., in the example below, the parameter grid has 3 values for hashingTF.numFeatures and 2 values for lr.regParam, and CrossValidator uses 2 folds. This multiplies out to $(3 \times 2) \times 2 = 12$ different models being trained. In realistic settings, it can be common to try many more parameters and use more folds ($k = 3$ and $k = 10$ are common). In other words, using CrossValidator can be very expensive. However, it is also a well-established method for choosing parameters which is more statistically sound than heuristic hand-tuning.

Scala

Java

Python

Refer to the [CrossValidator Python docs](#) for more details on the API.

```
from pyspark.ml import Pipeline
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.ml.feature import HashingTF, Tokenizer
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder

# Prepare training documents, which are labeled.
training = spark.createDataFrame([
    (0, "a b c d e spark", 1.0),
    (1, "b d", 0.0),
    (2, "spark f g h", 1.0),
    (3, "hadoop mapreduce", 0.0),
    (4, "b spark who", 1.0),
    (5, "g d a y", 0.0),
    (6, "spark fly", 1.0),
    (7, "was mapreduce", 0.0),
    (8, "e spark program", 1.0),
    (9, "a e c l", 0.0),
    (10, "spark compile", 1.0),
    (11, "hadoop software", 0.0)
], ["id", "text", "label"])

# Configure an ML pipeline, which consists of tree stages: tokenizer, hashingTF, and lr.
tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(), outputCol="features")
lr = LogisticRegression(maxIter=10)
pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])

# We now treat the Pipeline as an Estimator, wrapping it in a CrossValidator instance.
# This will allow us to jointly choose parameters for all Pipeline stages.
# A CrossValidator requires an Estimator, a set of Estimator ParamMaps, and an Evaluator.
# We use a ParamGridBuilder to construct a grid of parameters to search over.
# With 3 values for hashingTF.numFeatures and 2 values for lr.regParam,
# this grid will have 3 x 2 = 6 parameter settings for CrossValidator to choose from.
```

»

```

paramGrid = ParamGridBuilder() \
    .addGrid(hashingTF.numFeatures, [10, 100, 1000]) \
    .addGrid(lr.regParam, [0.1, 0.01]) \
    .build()

crossval = CrossValidator(estimator=pipeline,
                          estimatorParamMaps=paramGrid,
                          evaluator=BinaryClassificationEvaluator(),
                          numFolds=2) # use 3+ folds in practice

# Run cross-validation, and choose the best set of parameters.
cvModel = crossval.fit(training)

# Prepare test documents, which are unlabeled.
test = spark.createDataFrame([
    (4, "spark i j k"),
    (5, "l m n"),
    (6, "mapreduce spark"),
    (7, "apache hadoop")
], ["id", "text"])

# Make predictions on test documents. cvModel uses the best model found (lrModel).
prediction = cvModel.transform(test)
selected = prediction.select("id", "text", "probability", "prediction")
for row in selected.collect():
    print(row)

```

Find full example code at "examples/src/main/python/ml/cross_validator.py" in the Spark repo.

Train-Validation Split

In addition to `CrossValidator` Spark also offers `TrainValidationSplit` for hyper-parameter tuning.

`TrainValidationSplit` 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 when the training dataset is not sufficiently large.

Unlike `CrossValidator`, `TrainValidationSplit` creates a single (training, test) dataset pair. It splits the dataset into these two parts using the `trainRatio` parameter. For example with `trainRatio = 0.75`, `TrainValidationSplit` will generate a training and test dataset pair where 75% of the data is used for training and 25% for validation.

Like `CrossValidator`, `TrainValidationSplit` finally fits the Estimator using the best `ParamMap` and the entire dataset.

Examples: model selection via train validation split

Scala

Java

Python

Refer to the [TrainValidationSplit Python docs](#) for more details on the API.

```

from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.regression import LinearRegression
from pyspark.ml.tuning import ParamGridBuilder, TrainValidationSplit

```

»

```
# Prepare training and test data.
data = spark.read.format("libsvm")\
    .load("data/mllib/sample_linear_regression_data.txt")
train, test = data.randomSplit([0.9, 0.1], seed=12345)

lr = LinearRegression(maxIter=10)

# We use a ParamGridBuilder to construct a grid of parameters to search over.
# TrainValidationSplit will try all combinations of values and determine best model using
# the evaluator.
paramGrid = ParamGridBuilder()\
    .addGrid(lr.regParam, [0.1, 0.01]) \
    .addGrid(lr.fitIntercept, [False, True])\
    .addGrid(lr.elasticNetParam, [0.0, 0.5, 1.0])\
    .build()

# In this case the estimator is simply the linear regression.
# A TrainValidationSplit requires an Estimator, a set of Estimator ParamMaps, and an Evaluator.
tvsv = TrainValidationSplit(estimator=lr,
                           estimatorParamMaps=paramGrid,
                           evaluator=RegressionEvaluator(),
                           # 80% of the data will be used for training, 20% for validation.
                           trainRatio=0.8)

# Run TrainValidationSplit, and choose the best set of parameters.
model = tvsv.fit(train)

# Make predictions on test data. model is the model with combination of parameters
# that performed best.
model.transform(test)\
    .select("features", "label", "prediction")\
    .show()
```

Find full example code at "examples/src/main/python/ml/train_validation_split.py" in the Spark repo.