Spark Machine Learning

CMPT 732, Fall 2018

Recap: Machine Learning

We have input columns (*features*), x, and *parameters* θ.

Functions $y(x;\theta)$ produce predictions, y.

We want θ that minimizes (some measure of) error between the predictions y and target *labels* t.

If we have a discrete set of labels/predictions, then we have a *classification* problem: we are trying to predict the class or category.

If we have a continuout set of labels/predictions, then we have a *regression* problems: we are trying to predict a numeric value.

Or maybe we don't have labels are are trying to partition our inputs into unknown categories: *clustering*.

Spark ML

As with other components: there's an older RDD-based API, and a newer DataFrame-based API. We'll talk only about the DataFrame ML tools: the `pyspark.ml` package.

See also the [Spark ML Guide].

Complete code for the example to follow: [`ml_pipeline.py`].

Pieces we'll have:

DataFrame
: As before, store data: features, feature vectors, true labels, and generated predictions.

Transformers
: Feature extraction and transformation. Generally manipulating the data to get the featuers you need. Implemented with `Transformer` instances.

Estimator
: Implementations of ML algorithms that make predictions: regressors, classifiers, clustering. Implemented with `Estimator` instances.

Pipelines

[Concepts may be familiar to you from Scikit-Learn, but also maybe not…]

We are often going to want to take the data we have, and manipulate it before wassing it into the model: feature engineering, but also just reformatting and tidying the data.

That's what the `Transformer`s are for. Common pattern: data → `Transformer` → ⋯ → `Transformer` → `Estimator` → predictions.

You *could* apply transformations manually, but that's going to be tedious and error-prone: you have to apply the same ones for training, validation, testing, predicing.

A *pipeline* describes a series of transformations to its input, finishing with an estimator to make predictions. Implemented with `Pipeline` instances.

A pipeline can be trained as a unit: some transforms need training (PCA, indexer, etc), and the estimator certainly does.

Estimators need a single column of all features put together into a vector, so minimal pipeline might be:

```
assemble_features = VectorAssembler(
    inputCols=['length', 'width', 'height'],
    outputCol='features')
classifier = GBTRegressor(
    featuresCol='features', labelCol='volume')
pipeline = Pipeline(stages=[assemble_features, classifier])
```

Models

When a Spark estimator is trained, it produces a *model* object: a trained estimator that can actually make predictions; a `Model` (or probably `PipelineModel`) instance.

If we have some training data:

```
model = pipeline.fit(training)
```

Once trained, we can predict, possibly on some validation data:

```
predictions = model.transform(validation)
predictions.show()
```

Evaluation

Once you have a trained model, you probably want to come up with a score to see how it's working. This is done with `Estmator` instances.

Regression and classification are evaluated differently, but the API is the same.

```
r2_evaluator = RegressionEvaluator(
    predictionCol='prediction', labelCol='volume',
    metricName='r2')
r2 = r2_evaluator.evaluate(predictions)
print(r2)
```

ML Algorithms

Lots of learning algorithms to choose from. All of them implement the `Estimator` interface.

More Topics

Regularization

Hyperparameter tuning (Model selection)

