Documentación de modelos

Modelo base Benchmark, clasificación o regresión en base a nuestros objetivos. Qué cosas pueden correr más rápido.

https://spark.apache.org/docs/latest/ml-classification-regression.html#classification

Binomial logistic regression

Logistic regression is widely used to predict a binary response. It is a linear method as described above in equation (1), with the loss function in the formulation given by the logistic loss:

$$L(\mathbf{w};\mathbf{x},y) := \log(1+\exp(-y\mathbf{w}^T\mathbf{x})).$$

For binary classification problems, the algorithm outputs a binary logistic regression model. Given a new data point, denoted by x, the model makes predictions by applying the logistic function

$$\mathrm{f}(z) = \frac{1}{1+e^{-z}}$$

where z=wTx. By default, if f(wTx)>0.5, the outcome is positive, or negative otherwise, though unlike linear SVMs, the raw output of the logistic regression model, f(z), has a probabilistic interpretation (i.e., the probability that x is positive).

Binary logistic regression can be generalized into multinomial logistic regression to train and predict multiclass classification problems. For example, for K possible outcomes, one of the outcomes can be chosen as a "pivot", and the other K−1 outcomes can be separately regressed against the pivot outcome. In spark.mllib, the first class 0 is chosen as the "pivot" class. See Section 4.4 of The Elements of Statistical Learning for references. Here is a detailed mathematical derivation.

For multiclass classification problems, the algorithm will output a multinomial logistic regression model, which contains K−1 binary logistic regression models regressed against the first class. Given a new data points, K−1 models will be run, and the class with largest probability will be chosen as the predicted class.

Python API documentation.

Random forest classifier

Random forests are ensembles of decision trees. Random forests combine many decision trees in order to reduce the risk of overfitting. The `spark.ml` implementation supports random forests for binary and multiclass classification and for regression, using both continuous and categorical features.

Random Forest learning algorithm for classification. It supports both binary and multiclass labels, as well as both continuous and categorical features.

Python API docs

Gradient-boosted tree classifier

Gradient-Boosted Trees (GBTs) are ensembles of decision trees. GBTs iteratively train decision trees in order to minimize a loss function. Like decision trees, GBTs handle categorical features, extend to the multiclass classification setting, do not require feature scaling, and are able to capture non-linearities and feature interactions.

Gradient boosting iteratively trains a sequence of decision trees. On each iteration, the algorithm uses the current ensemble to predict the label of each training instance and then compares the prediction with the true label. The dataset is re-labeled to put more emphasis on training instances with poor predictions. Thus, in the next iteration, the decision tree will help correct for previous mistakes.

The specific mechanism for re-labeling instances is defined by a loss function (discussed below). With each iteration, GBTs further reduce this loss function on the training data.

Gradient boosting can overfit when trained with more trees. In order to prevent overfitting, it is useful to validate while training. The method runWithValidation has been provided to make use of this option. It takes a pair of RDD's as arguments, the first one being the training dataset and the second being the validation dataset.

The training is stopped when the improvement in the validation error is not more than a certain tolerance (supplied by the `validationTol` argument in `BoostingStrategy`). In practice, the validation error decreases initially and later increases. There might be cases in which the validation error does not change monotonically, and the user is advised to set a large enough negative tolerance and examine the validation curve using `evaluateEachIteration` (which gives the error or loss per iteration) to tune the number of iterations.

Python API docs

One-vs-Rest classifier (a.k.a. One-vs-All)

Strategy involves training a single classifier per class, with the samples of that class as positive samples and all other samples as negatives. This strategy requires the base classifiers to produce a real-valued confidence score for its decision, rather than just a class label; discrete class labels alone can lead to ambiguities, where multiple classes are predicted for a single sample.

Reduction of Multiclass Classification to Binary Classification. Performs reduction using one against all strategy. For a multiclass classification with k classes, train k models (one per class). Each example is scored against all k models and the model with highest score is picked to label the example.

Python API docs

Naive Bayes Classification

Naive Bayes classifiers are a family of simple probabilistic, multiclass classifiers based on applying Bayes' theorem with strong (naive) independence assumptions between every pair of features.

Naive Bayes can be trained very efficiently. With a single pass over the training data, it computes the conditional probability distribution of each feature given each label. For prediction, it applies Bayes' theorem to compute the conditional probability distribution of each label given an observation.

Input data: These Multinomial, Complement and Bernoulli models are typically used for document classification. Within that context, each observation is a document and each feature represents a term. A feature's value is the frequency of the term (in Multinomial or Complement Naive Bayes) or a zero or one indicating whether the term was found in the document (in Bernoulli Naive Bayes). Feature values for Multinomial and Bernoulli models must be *non-negative*. The model type is selected with an optional parameter "multinomial", "complement", "bernoulli" or "gaussian", with "multinomial" as the default. For document classification, the input feature vectors should usually be sparse vectors. Since the training data is only used once, it is not necessary to cache it.
Python API docs

XGBoost: Gradient-boosted tree Classification

XGBoost4J-Spark is a project aiming to seamlessly integrate XGBoost and Apache Spark by fitting XGBoost to Apache Spark's MLLIB framework. With the integration, user can not only uses the high-performant algorithm implementation of XGBoost, but also leverages the powerful data processing engine of Spark for:

- Feature Engineering: feature extraction, transformation, dimensionality reduction, and selection, etc.
- Pipelines: constructing, evaluating, and tuning ML Pipelines
- Persistence: persist and load machine learning models and even whole Pipelines

XGBoost4J-Spark requires Apache Spark 2.4+.

Gradient boosted trees are a supervised learning algorithm, which use the predictions of a collection of simple tree models for classification or regression. XGBoost uses a regularized loss function (the difference between the labels and the predictions) and another cost function for model complexity. These loss terms are minimized using gradient descent to reduce error and complexity of the model to move towards the simplest and most accurate model. This is a very broad generalization of the algorithm. For more detail, check out the XGBoost documentation on its powerful algorithm.