xgboost-2-regression

May 8, 2019

1 Using XGBoost to perform regression

Miao Cai

This is based on the datacamp online tutorial *Extreme Gradient Boosting with XGBoost*.

1.1 Common regression metrics

- Root mean squared error (RMSE)
- Mean absolute error (MAE)

RMSE tends to punish larger differences between predicted and actual values much more than smaller ones. **MSE** tends to sum absolute differences across all of the samples we build our model on. Although MAE is not affected by large differences as much as RMSE, it lacks some nice matchematical properties that make it much less frequently used as an evaluation metric.

Common regression algorithms

- Linear regression
- Decision trees

Decision trees can be both used as regression or classification tools, whihc is an important property that makes them prime candidates to be building blocks for XGBoost models.

1.2 Objective (loss) functions and base learners

An objective function (loss function) quantifies how far off a prediction is from the actual results for a given data point. It measures the difference between estimated and true value for some collection of data. The goal of building a machine learning model is to find the model that yields the minimum value of the loss function.

Common loss functions and XGBoost

- Loss function names in xgboost:
  - reg:linear - use for regression problems
  - reg:logistic - use for classification problems when you want just decision, not probability
  - binary:logistic - use when you want probability rather than just decision

Base learners and why we need them

- XGBoost involves creating a meta-model that is composed of **many individual models** that **combine** to give a final prediction.
- Individual models = base learners
- We want base learners that when combined create final prediction that is **non-linear**
- Each base learner should be good at distinguishing or predicting different part of the dataset

The goal of XGBoost is to have base learners that is slightly better than random guessing on certain subsets of training examples and uniformly bad at the remainder, so that when all of the predictions are combined, the uniformly bad predictions cancel out and those slightly better than chance combine into a single very good prediction.

By default, XGBoost use trees as base learners, which is `booster="gbtree"`

Non-linear learners

Now that you've used trees as base models in XGBoost, let's use the other kind of base model that can be used with XGBoost - a linear learner. This model, although not as commonly used in XGBoost, allows you to create a regularized linear regression using XGBoost's powerful learning API. However, because it's uncommon, you have to use XGBoost's own non-scikit-learn compatible functions to build the model, such as `xgb.train()`.

In order to do this you must create the parameter dictionary that describes the kind of booster you want to use (similarly to how you created the dictionary in Chapter 1 when you used `xgb.cv()`). The key-value pair that defines the booster type (base model) you need is `"booster":"gblinear"`. Once you've created the model, you can use the .train() and .predict() methods of the model just like you've done in the past.

```
In [1]: # Convert the training and testing sets into DMatrixes: DM_train, DM_test
        #  DM_train = xgb.DMatrix(data=X_train, label=y_train)
        #  DM_test =  xgb.DMatrix(data=X_test, label=y_test)

        # Create the parameter dictionary: params
        #  params = {"booster":"gblinear", "objective":"reg:linear"}

        # Train the model: xg_reg
        #  xg_reg = xgb.train(params = params, dtrain=DM_train, num_boost_round=5)

        # Predict the labels of the test set: preds
        #  preds = xg_reg.predict(DM_test)

        # Compute and print the RMSE
        #  rmse = np.sqrt(mean_squared_error(y_test,preds))
        #  print("RMSE: %f" % (rmse))

In [5]: ## Evaluating model quality

        # Create the DMatrix: housing_dmatrix
        #  housing_dmatrix = xgb.DMatrix(data=X, label=y)

        # Create the parameter dictionary: params
        #  params = {"objective":"reg:linear", "max_depth":4}
        # Perform cross-validation: cv_results
        #  cv_results = xgb.cv(dtrain=housing_dmatrix, params=params, nfold=4, num_boost_round

        # Print cv_results
        #  print(cv_results)

        # Extract and print final boosting round metric
        #  print((cv_results["params"]).tail(1))
```

1.3 Regularization in XGBoost

- Regularization is a control on model complexity
- Want models that are both accurate and as simple as possible
- Regularization parameters in XGBoost:
  - gamma - minimum loss reduction allowed for a split to occur
  - alpha - l1 regularization on **leaf weights** (not on feature weights), larger values mean more regularization, which causes many leaf weights in the base learner to go to 0.
  - lambda - l2 regularization on leaf weights, l2 regularization is a much smoother penalty than l1 and causes leaf weights to smoothly decrease, instead of strong sparsity constrains on the leaf weights as l1.

```
In [ ]: ## Base learners in XGBoost

        - Linear Base Learners:
                + Sum of linear terms
                + Boosted model is weighted sum of linear models
                (you do not get any non-linear combinations of )
```