3.5 Deal with missing values

February 22, 2017

1 Deal with missing values

The following notebook demonstrate XGBoost resilience to missing values. Two approaches - native interface, and Sklearn wrapper were tested against missing datasets.

Missing value is commonly seen in real-world data sets. Handling missing values has no rule to apply to all cases, since there could be various reasons for the values to be missing.

What you will learn: - how to prepare data with missing elements, - handling missing values in native interface, - handling missing values in Sklearn interface

1.0.1 Prepare data

First begin with loading all libraries and assuring reproducibility

```
In [1]: import numpy as np
        import xgboost as xgb

        from xgboost.sklearn import XGBClassifier

        from sklearn.cross_validation import cross_val_score

        # reproducibility
        seed = 123
```

Let's prepare a valid dataset with no missing values. There are 10 samples, each one will contain 5 randomly generated features and will be assigned to one of two classes.

```
In [2]: # create valid dataset
        np.random.seed(seed)

        data_v = np.random.rand(10,5) # 10 entities, each contains 5 features
        data_v
```

```
Out[2]: array([[ 0.69646919,  0.28613933,  0.22685145,  0.55131477,  0.71946897],
               [ 0.42310646,  0.9807642 ,  0.68482974,  0.4809319 ,  0.39211752],
               [ 0.34317802,  0.72904971,  0.43857224,  0.0596779 ,  0.39804426],
               [ 0.73799541,  0.18249173,  0.17545176,  0.53155137,  0.53182759],
               [ 0.63440096,  0.84943179,  0.72445532,  0.61102351,  0.72244338],
```

```
       [ 0.32295891,  0.36178866,  0.22826323,  0.29371405,  0.63097612],
       [ 0.09210494,  0.43370117,  0.43086276,  0.4936851 ,  0.42583029],
       [ 0.31226122,  0.42635131,  0.89338916,  0.94416002,  0.50183668],
       [ 0.62395295,  0.1156184 ,  0.31728548,  0.41482621,  0.86630916],
       [ 0.25045537,  0.48303426,  0.98555979,  0.51948512,  0.61289453]])
```

In the second example we are going to add some missing values

```
In [3]: # add some missing values
        data_m = np.copy(data_v)

        data_m[2, 3] = np.nan
        data_m[0, 1] = np.nan
        data_m[0, 2] = np.nan
        data_m[1, 0] = np.nan
        data_m[4, 4] = np.nan
        data_m[7, 2] = np.nan
        data_m[9, 1] = np.nan

        data_m

Out[3]: array([[ 0.69646919,         nan,         nan,  0.55131477,  0.71946897],
               [        nan,  0.9807642 ,  0.68482974,  0.4809319 ,  0.39211752],
               [ 0.34317802,  0.72904971,  0.43857224,         nan,  0.39804426],
               [ 0.73799541,  0.18249173,  0.17545176,  0.53155137,  0.53182759],
               [ 0.63440096,  0.84943179,  0.72445532,  0.61102351,         nan],
               [ 0.32295891,  0.36178866,  0.22826323,  0.29371405,  0.63097612],
               [ 0.09210494,  0.43370117,  0.43086276,  0.4936851 ,  0.42583029],
               [ 0.31226122,  0.42635131,         nan,  0.94416002,  0.50183668],
               [ 0.62395295,  0.1156184 ,  0.31728548,  0.41482621,  0.86630916],
               [ 0.25045537,         nan,  0.98555979,  0.51948512,  0.61289453]])
```

Also generate target variables. Each sample will be assigned to one of two classes - so we are dealing with binary classification problem

```
In [4]: np.random.seed(seed)

        label = np.random.randint(2, size=10) # binary target
        label

Out[4]: array([0, 1, 0, 0, 0, 0, 0, 1, 1, 0])
```

1.0.2 Native interface

In this case we will check how does the native interface handles missing data. Begin with specifing default parameters.

```
In [5]: # specify general training parameters
        params = {
```

```
    'objective':'binary:logistic',
    'max_depth':1,
    'silent':1,
    'eta':0.5
}

num_rounds = 5
```

In the experiment first we will create a valid `DMatrix` (with all values), see if it works ok, and then repeat the process with lacking one.

```
In [6]: dtrain_v = xgb.DMatrix(data_v, label=label)
```

Cross-validate results

```
In [7]: xgb.cv(params, dtrain_v, num_rounds, seed=seed)
```

Out[7]:

	test-error-mean	test-error-std	train-error-mean	train-error-std
0	0.333333	0	0.333333	0
1	0.333333	0	0.333333	0
2	0.333333	0	0.333333	0
3	0.333333	0	0.333333	0
4	0.333333	0	0.333333	0

The output obviously doesn't make sense, because the data is completely random.

When creating `DMatrix` holding missing values we have to explicitly tell what denotes that it's missing. Sometimes it might be `0`, `999` or others. In our case it's Numpy's `NAN`. Add `missing` argument to `DMatrix` constructor to handle it.

```
In [8]: dtrain_m = xgb.DMatrix(data_m, label=label, missing=np.nan)
```

Cross-validate results:

```
In [9]: xgb.cv(params, dtrain_m, num_rounds, seed=seed)
```

Out[9]:

	test-error-mean	test-error-std	train-error-mean	train-error-std
0	0.333333	0	0.333333	0
1	0.333333	0	0.333333	0
2	0.333333	0	0.333333	0
3	0.333333	0	0.333333	0
4	0.333333	0	0.333333	0

It looks like the algorithm works also with missing values.

In XGBoost chooses a soft way to handle missing values.

When using a feature with missing values to do splitting, XGBoost will assign a direction to the missing values instead of a numerical value.

Specifically, XGBoost guides all the data points with missing values to the left and right respectively, then choose the direction with a higher gain with regard to the objective.

1.0.3 Sklearn wrapper

The following section shows how to validate the same behaviour using Sklearn interface.

Begin with defining parameters and creating an estimator object.

```
In [10]: params = {
             'objective': 'binary:logistic',
             'max_depth': 1,
             'learning_rate': 0.5,
             'silent': 1.0,
             'n_estimators': 5
         }

In [11]: clf = XGBClassifier(**params)
         clf

Out[11]: XGBClassifier(base_score=0.5, colsample_bylevel=1, colsample_bytree=1,
                gamma=0, learning_rate=0.5, max_delta_step=0, max_depth=1,
                min_child_weight=1, missing=None, n_estimators=5, nthread=-1,
                objective='binary:logistic', reg_alpha=0, reg_lambda=1,
                scale_pos_weight=1, seed=0, silent=1.0, subsample=1)
```

Cross-validate results with full dataset. Because we have only 10 samples, we will perform 2-fold CV.

```
In [12]: cross_val_score(clf, data_v, label, cv=2, scoring='accuracy')

Out[12]: array([ 0.66666667,  0.75      ])
```

Some score was obtained, we won't dig into it's interpretation.

See if things work also with missing values

```
In [13]: cross_val_score(clf, data_m, label, cv=2, scoring='accuracy')

Out[13]: array([ 0.66666667,  0.75      ])
```

Both methods works with missing datasets. The Sklearn package by default handles data with `np.nan` as missing (so you will need additional pre-precessing if using different convention).