XGBoost is a popular implementation of Gradient Boosting because of its speed and performance.

Internally, XGBoost models represent all problems as a regression predictive modeling problem that only takes numerical values as input. If your data is in a different form, it must be prepared into the expected format.

In this post you will discover how to prepare your data for using with gradient boosting with the XGBoost library in Python.

After reading this post you will know:

* How to encode string output variables for classification.
* How to prepare categorical input variables using one hot encoding.
* How to automatically handle missing data with XGBoost.

Let’s get started.



Data Preparation for Gradient Boosting with XGBoost in Python  
Photo by [Ed Dunens](https://www.flickr.com/photos/blachswan/14990404869/), some rights reserved.

## The Algorithm that is Winning Competitions ...XGBoost for fast gradient boosting

XGBoost is the high performance implementation of gradient boosting that you can now access directly in Python.

Your PDF Download and Email Course.

#### FREE 7-Day Mini-Course on  XGBoost With Python

[**Download Your FREE Mini-Course**](https://machinelearningmastery.leadpages.co/leadbox/1431f1e73f72a2%3A164f8be4f346dc/5662543133540352/)

  Download your PDF containing all 7 lessons.

Daily lesson via email with tips and tricks.

## Label Encode String Class Values

The iris flowers classification problem is an example of a problem that has a string class value.

This is a prediction problem where given measurements of iris flowers in centimeters, the task is to predict to which species a given flower belongs.

Below is a sample of the raw dataset. You can learn more about this dataset and download the raw data in CSV format from the [UCI Machine Learning Repository](http://archive.ics.uci.edu/ml/datasets/Iris).



|  |  |
| --- | --- |
| 1  2  3  4  5 | 5.1,3.5,1.4,0.2,Iris-setosa  4.9,3.0,1.4,0.2,Iris-setosa  4.7,3.2,1.3,0.2,Iris-setosa  4.6,3.1,1.5,0.2,Iris-setosa  5.0,3.6,1.4,0.2,Iris-setosa |

XGBoost cannot model this problem as-is as it requires that the output variables be numeric.

We can easily convert the string values to integer values using the [LabelEncoder](http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html). The three class values (Iris-setosa, Iris-versicolor, Iris-virginica) are mapped to the integer values (0, 1, 2).



|  |  |
| --- | --- |
| 1  2  3  4 | # encode string class values as integers  label\_encoder = LabelEncoder()  label\_encoder = label\_encoder.fit(Y)  label\_encoded\_y = label\_encoder.transform(Y) |

We save the label encoder as a separate object so that we can transform both the training and later the test and validation datasets using the same encoding scheme.

Below is a complete example demonstrating how to load the iris dataset. Notice that Pandas is used to load the data in order to handle the string class values.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | # multiclass classification  import pandas  import xgboost  from sklearn import cross\_validation  from sklearn.metrics import accuracy\_score  from sklearn.preprocessing import LabelEncoder  # load data  data = pandas.read\_csv('iris.csv', header=0)  dataset = data.values  # split data into X and y  X = dataset[:,0:4]  Y = dataset[:,4]  # encode string class values as integers  label\_encoder = LabelEncoder()  label\_encoder = label\_encoder.fit(Y)  label\_encoded\_y = label\_encoder.transform(Y)  seed = 7  test\_size = 0.33  X\_train, X\_test, y\_train, y\_test = cross\_validation.train\_test\_split(X, label\_encoded\_y, test\_size=test\_size, random\_state=seed)  # fit model no training data  model = xgboost.XGBClassifier()  model.fit(X\_train, y\_train)  print(model)  # make predictions for test data  y\_pred = model.predict(X\_test)  predictions = [round(value) for value in y\_pred]  # evaluate predictions  accuracy = accuracy\_score(y\_test, predictions)  print("Accuracy: %.2f%%" % (accuracy \* 100.0)) |

Running the example produces the following output:



|  |  |
| --- | --- |
| 1  2  3  4  5  6 | XGBClassifier(base\_score=0.5, colsample\_bylevel=1, colsample\_bytree=1,         gamma=0, learning\_rate=0.1, max\_delta\_step=0, max\_depth=3,         min\_child\_weight=1, missing=None, n\_estimators=100, nthread=-1,         objective='multi:softprob', reg\_alpha=0, reg\_lambda=1,         scale\_pos\_weight=1, seed=0, silent=True, subsample=1)  Accuracy: 92.00% |

Notice how the XGBoost model is configured to automatically model the multiclass classification problem using the **multi:softprob** objective, a variation on the softmax loss function to model class probabilities. This suggests that internally, that the output class is converted into a one hot type encoding automatically.

## One Hot Encode Categorical Data

Some datasets only contain categorical data, for example the breast cancer dataset.

This dataset describes the technical details of breast cancer biopsies and the prediction task is to predict whether or not the patient has a recurrence of cancer, or not.

Below is a sample of the raw dataset. You can learn more about this dataset at the [UCI Machine Learning Repository](http://archive.ics.uci.edu/ml/datasets/Breast+Cancer) and download it in CSV format from [mldata.org](http://mldata.org/repository/data/viewslug/datasets-uci-breast-cancer/).



|  |  |
| --- | --- |
| 1  2  3  4  5 | '40-49','premeno','15-19','0-2','yes','3','right','left\_up','no','recurrence-events'  '50-59','ge40','15-19','0-2','no','1','right','central','no','no-recurrence-events'  '50-59','ge40','35-39','0-2','no','2','left','left\_low','no','recurrence-events'  '40-49','premeno','35-39','0-2','yes','3','right','left\_low','yes','no-recurrence-events'  '40-49','premeno','30-34','3-5','yes','2','left','right\_up','no','recurrence-events' |

We can see that all 9 input variables are categorical and described in string format. The problem is a binary classification prediction problem and the output class values are also described in string format.

We can reuse the same approach from the previous section and convert the string class values to integer values to model the prediction using the LabelEncoder. For example:



|  |  |
| --- | --- |
| 1  2  3  4 | # encode string class values as integers  label\_encoder = LabelEncoder()  label\_encoder = label\_encoder.fit(Y)  label\_encoded\_y = label\_encoder.transform(Y) |

We can use this same approach on each input feature in X, but this is only a starting point.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | # encode string input values as integers  features = []  for i in range(0, X.shape[1]):  label\_encoder = LabelEncoder()  feature = label\_encoder.fit\_transform(X[:,i])  features.append(feature)  encoded\_x = numpy.array(features)  encoded\_x = encoded\_x.reshape(X.shape[0], X.shape[1]) |

XGBoost may assume that encoded integer values for each input variable have an ordinal relationship. For example that ‘left-up’ encoded as 0 and ‘left-low’ encoded as 1 for the breast-quad variable have a meaningful relationship as integers. In this case, this assumption is untrue.

Instead, we must map these integer values onto new binary variables, one new variable for each categorical value.

For example, the breast-quad variable has the values:



|  |  |
| --- | --- |
| 1  2  3  4  5 | left-up  left-low  right-up  right-low  central |

We can model this as 5 binary variables as follows:



|  |  |
| --- | --- |
| 1  2  3  4  5  6 | left-up, left-low, right-up, right-low, central  1,0,0,0,0  0,1,0,0,0  0,0,1,0,0  0,0,0,1,0  0,0,0,0,1 |

This is called [one hot encoding](https://en.wikipedia.org/wiki/One-hot). We can one hot encode all of the categorical input variables using the [OneHotEncoder](http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html) class in scikit-learn.

We can one hot encode each feature after we have label encoded it. First we must transform the feature array into a 2-dimensional NumPy array where each integer value is a feature vector with a length 1.



|  |  |
| --- | --- |
| 1 | feature = feature.reshape(X.shape[0], 1) |

We can then create the OneHotEncoder and encode the feature array.



|  |  |
| --- | --- |
| 1  2 | onehot\_encoder = OneHotEncoder(sparse=False)  feature = onehot\_encoder.fit\_transform(feature) |

Finally, we can build up the input dataset by concatenating the one hot encoded features, one by one, adding them on as new columns (axis=2). We end up with an input vector comprised of 43 binary input variables.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | # encode string input values as integers  encoded\_x = None  for i in range(0, X.shape[1]):  label\_encoder = LabelEncoder()  feature = label\_encoder.fit\_transform(X[:,i])  feature = feature.reshape(X.shape[0], 1)  onehot\_encoder = OneHotEncoder(sparse=False)  feature = onehot\_encoder.fit\_transform(feature)  if encoded\_x is None:  encoded\_x = feature  else:  encoded\_x = numpy.concatenate((encoded\_x, feature), axis=1)  print("X shape: : ", encoded\_x.shape) |

Ideally, we may experiment with not one hot encode some of input attributes as we could encode them with an explicit ordinal relationship, for example the first column age with values like ’40-49′ and ’50-59′. This is left as an exercise, if you are interested in extending this example.

Below is the complete example with label and one hot encoded input variables and label encoded output variable.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45 | # binary classification, breast cancer dataset, label and one hot encoded  import numpy  from pandas import read\_csv  from xgboost import XGBClassifier  from sklearn.cross\_validation import train\_test\_split  from sklearn.metrics import accuracy\_score  from sklearn.preprocessing import LabelEncoder  from sklearn.preprocessing import OneHotEncoder  # load data  data = read\_csv('datasets-uci-breast-cancer.csv', header=0)  dataset = data.values  # split data into X and y  X = dataset[:,0:9]  Y = dataset[:,9]  # encode string input values as integers  encoded\_x = None  for i in range(0, X.shape[1]):  label\_encoder = LabelEncoder()  feature = label\_encoder.fit\_transform(X[:,i])  feature = feature.reshape(X.shape[0], 1)  onehot\_encoder = OneHotEncoder(sparse=False)  feature = onehot\_encoder.fit\_transform(feature)  if encoded\_x is None:  encoded\_x = feature  else:  encoded\_x = numpy.concatenate((encoded\_x, feature), axis=1)  print("X shape: : ", encoded\_x.shape)  # encode string class values as integers  label\_encoder = LabelEncoder()  label\_encoder = label\_encoder.fit(Y)  label\_encoded\_y = label\_encoder.transform(Y)  # split data into train and test sets  seed = 7  test\_size = 0.33  X\_train, X\_test, y\_train, y\_test = train\_test\_split(encoded\_x, label\_encoded\_y, test\_size=test\_size, random\_state=seed)  # fit model no training data  model = XGBClassifier()  model.fit(X\_train, y\_train)  print(model)  # make predictions for test data  y\_pred = model.predict(X\_test)  predictions = [round(value) for value in y\_pred]  # evaluate predictions  accuracy = accuracy\_score(y\_test, predictions)  print("Accuracy: %.2f%%" % (accuracy \* 100.0)) |

Running this example we get the following output:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | ('X shape: : ', (285, 43))  XGBClassifier(base\_score=0.5, colsample\_bylevel=1, colsample\_bytree=1,         gamma=0, learning\_rate=0.1, max\_delta\_step=0, max\_depth=3,         min\_child\_weight=1, missing=None, n\_estimators=100, nthread=-1,         objective='binary:logistic', reg\_alpha=0, reg\_lambda=1,         scale\_pos\_weight=1, seed=0, silent=True, subsample=1)  Accuracy: 69.47% |

You may get a warning like the following, that you can ignore for now:



|  |  |
| --- | --- |
| 1 | FutureWarning: numpy not\_equal will not check object identity in the future |

Again we can see that the XGBoost framework chose the ‘**binary:logistic**‘ objective automatically, the right objective for this binary classification problem.

## Support for Missing Data

XGBoost can automatically learn how to best handle missing data.

In fact, XGBoost was designed to work with sparse data, like the one hot encoded data from the previous section, and missing data is handled the same way that sparse or zero values are handled, by minimizing the loss function.

For more information on the technical details for how missing values are handled in XGBoost, see Section 3.4 “Sparsity-aware Split Finding” in the paper [XGBoost: A Scalable Tree Boosting System](https://arxiv.org/abs/1603.02754).

The Horse Colic dataset is a good example to demonstrate this capability as it contains a large percentage of missing data, approximately 30%.

You can learn more about the Horse Colic dataset and download the raw data file from the [UCI Machine Learning repository](https://archive.ics.uci.edu/ml/datasets/Horse+Colic).

The values are separated by whitespace and we can easily load it using the Pandas function[read\_csv](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html).



|  |  |
| --- | --- |
| 1 | dataframe = read\_csv("horse-colic.csv", delim\_whitespace=True, header=None) |

Once loaded, we can see that the missing data is marked with a question mark character (‘?’). We can change these missing values to the sparse value expected by XGBoost which is the value zero (0).



|  |  |
| --- | --- |
| 1  2 | # set missing values to 0  X[X == '?'] = 0 |

Because the missing data was marked as strings, those columns with missing data were all loaded as string data types. We can now convert the entire set of input data to numerical values.



|  |  |
| --- | --- |
| 1  2 | # convert to numeric  X = X.astype('float32') |

Finally, this is a binary classification problem although the class values are marked with the integers 1 and 2. We model binary classification problems in XGBoost as logistic 0 and 1 values. We can easily convert the Y dataset to 0 and 1 integers using the LabelEncoder, as we did in the iris flowers example.



|  |  |
| --- | --- |
| 1  2  3  4 | # encode Y class values as integers  label\_encoder = LabelEncoder()  label\_encoder = label\_encoder.fit(Y)  label\_encoded\_y = label\_encoder.transform(Y) |

The full code listing is provided below for completeness.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | # binary classification, missing data  from pandas import read\_csv  from xgboost import XGBClassifier  from sklearn.cross\_validation import train\_test\_split  from sklearn.metrics import accuracy\_score  from sklearn.preprocessing import LabelEncoder  # load data  dataframe = read\_csv("horse-colic.csv", delim\_whitespace=True, header=None)  dataset = dataframe.values  # split data into X and y  X = dataset[:,0:27]  Y = dataset[:,27]  # set missing values to 0  X[X == '?'] = 0  # convert to numeric  X = X.astype('float32')  # encode Y class values as integers  label\_encoder = LabelEncoder()  label\_encoder = label\_encoder.fit(Y)  label\_encoded\_y = label\_encoder.transform(Y)  # split data into train and test sets  seed = 7  test\_size = 0.33  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, label\_encoded\_y, test\_size=test\_size, random\_state=seed)  # fit model no training data  model = XGBClassifier()  model.fit(X\_train, y\_train)  print(model)  # make predictions for test data  y\_pred = model.predict(X\_test)  predictions = [round(value) for value in y\_pred]  # evaluate predictions  accuracy = accuracy\_score(y\_test, predictions)  print("Accuracy: %.2f%%" % (accuracy \* 100.0)) |

Running this example produces the following output.



|  |  |
| --- | --- |
| 1  2  3  4  5  6 | XGBClassifier(base\_score=0.5, colsample\_bylevel=1, colsample\_bytree=1,         gamma=0, learning\_rate=0.1, max\_delta\_step=0, max\_depth=3,         min\_child\_weight=1, missing=None, n\_estimators=100, nthread=-1,         objective='binary:logistic', reg\_alpha=0, reg\_lambda=1,         scale\_pos\_weight=1, seed=0, silent=True, subsample=1)  Accuracy: 83.84% |

We can tease out the effect of XGBoost’s automatic handling of missing values, by marking the missing values with a non-zero value, such as 1.



|  |  |
| --- | --- |
| 1 | X[X == '?'] = 1 |

Re-running the example demonstrates a drop in accuracy for the model.



|  |  |
| --- | --- |
| 1 | Accuracy: 79.80% |

We can also impute the missing data with a specific value.

It is common to use a mean or a median for the column. We can easily impute the missing data using the scikit-learn [Imputer](http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.Imputer.html) class.



|  |  |
| --- | --- |
| 1  2  3 | # impute missing values as the mean  imputer = Imputer()  imputed\_x = imputer.fit\_transform(X) |

Below is the full example with missing data imputed with the mean value from each column.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | # binary classification, missing data, impute with mean  import numpy  from pandas import read\_csv  from xgboost import XGBClassifier  from sklearn.cross\_validation import train\_test\_split  from sklearn.metrics import accuracy\_score  from sklearn.preprocessing import LabelEncoder  from sklearn.preprocessing import Imputer  # load data  dataframe = read\_csv("horse-colic.csv", delim\_whitespace=True, header=None)  dataset = dataframe.values  # split data into X and y  X = dataset[:,0:27]  Y = dataset[:,27]  # set missing values to 0  X[X == '?'] = numpy.nan  # convert to numeric  X = X.astype('float32')  # impute missing values as the mean  imputer = Imputer()  imputed\_x = imputer.fit\_transform(X)  # encode Y class values as integers  label\_encoder = LabelEncoder()  label\_encoder = label\_encoder.fit(Y)  label\_encoded\_y = label\_encoder.transform(Y)  # split data into train and test sets  seed = 7  test\_size = 0.33  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, label\_encoded\_y, test\_size=test\_size, random\_state=seed)  # fit model no training data  model = XGBClassifier()  model.fit(X\_train, y\_train)  print(model)  # make predictions for test data  y\_pred = model.predict(X\_test)  predictions = [round(value) for value in y\_pred]  # evaluate predictions  accuracy = accuracy\_score(y\_test, predictions)  print("Accuracy: %.2f%%" % (accuracy \* 100.0)) |

Running this example in fact shows a small lift over the unoptimized XGBoost model with it’s automatic handling of missing data.



|  |  |
| --- | --- |
| 1 | Accuracy: 85.86% |

It is a good lesson to try both approaches (automatic handling and imputing) on your data when you have missing values.

## Want to Systematically Learn How To Use XGBoost?

### [XGBoost With Python](https://machinelearningmastery.com/xgboost-with-python/)

You can develop and evaluate XGBoost models in just a few lines of Python code. You need:

[>> XGBoost With Python](http://machinelearningmastery.com/xgboost-with-python)

Take the next step with **15 self-study tutorial lessons**.

Covers building large models on Amazon Web Services, feature importance, tree visualization, hyperparameter tuning, and much more...

Ideal for machine learning practitioners already familiar with the Python ecosystem.

[Bring XGBoost To Your Machine Learning Projects](http://machinelearningmastery.com/xgboost-with-python/)

## Summary

In this post you discovered how you can prepare your machine learning data for gradient boosting with XGBoost in Python.

Specifically, you learned:

* How to prepare string class values for binary classification using label encoding.
* How to prepare categorical input variables using a one hot encoding to model them as binary variables.
* How XGBoost automatically handles missing data and how you can mark and impute missing values.

Do you have any questions about how to prepare your data for XGBoost or about this post? Ask your questions in the comments and I will do my best to answer.