XGBoost With Python

Gradient Boosted Trees With XGBoost and scikit-learn

Jason Brownlee

MACHINE LEARNING MASTERY

Jason Brownlee

XGBoost With Python

Gradient Boosted Trees With XGBoost and scikit-learn

XGBoost With Python

Edition: v1.4

Contents

Part I

Introduction

Chapter 1

Welcome

Welcome to XGBoost With Python. This book is your guide to fast gradient boosting in Python. You will discover the XGBoost Python library for gradient boosting and how to use it to develop and evaluate gradient boosting models. In this book you will discover the techniques, recipes and skills with XGBoost that you can then bring to your own machine learning projects.

Gradient Boosting does have a some fascinating math under the covers, but you do not need to know it to be able to pick it up as a tool and wield it on important projects to deliver real value. From the applied perspective, gradient boosting is quite a shallow field and a motivated developer can quickly pick it up and start making very real and impactful contributions. This is my goal for you and this book is your ticket to that outcome.

1.1 Book Organization

The tutorials in this book are divided into three parts:

- **XGBoost Basics**.
- **XGBoost Advanced**.
- **XGBoost Tuning**.

In addition to these three parts, the Conclusions part at the end of the book includes a list of resources for getting help and diving deeper into the field.

1.1.1 XGBoost Basics

This part provides a gentle introduction to the XGBoost library for use with the scikit-learn library in Python. After completing the tutorials in this section, you will know your way around basic XGBoost models. Specifically, this part covers:

- A gentle introduction to the gradient boosting algorithm.
- A gentle introduction to the XGBoost library and why it is so popular.
- How to develop your first XGBoost model from scratch.

- How to prepare data for use with XGBoost.
- How to evaluate the performance of trained XGBoost models.
- How to visualize boosted trees within an XGBoost model.

1.1.2 XGBoost Advanced

This part provides an introduction to some of the more advanced features and uses of the XGBoost library. After completing the tutorials in this section you will know how to implement some of the more advanced capabilities of gradient boosting and scale up your models for bigger hardware platforms. Specifically, this part covers:

- How to serialize trained models to file and later load and use them to make predictions.
- How to calculate importance scores and use them for feature selection.
- How to monitor the performance of a model during training and set conditions for early stopping.
- How to harness the parallel features of the XGBoost library for training models faster.
- How to rapidly speed up model training of XGBoost models using Amazon cloud infrastructure.

1.1.3 XGBoost Tuning

This part provides tutorials detailing how to configure and tune XGBoost hyperparameters. After completing the tutorials in this part, you will know how to design parameter tuning experiments to get the most from your models. Specifically, this part covers:

- An introduction to XGBoost parameters and heuristics for good parameter values.
- How to tune the number and size of trees in a model.
- How to tune the learning rate and number of trees in a model.
- How to tune the sampling rates in stochastic variation of the algorithm.

1.1.4 Python Recipes

Building up a catalog of code recipes is an important part of your XGBoost journey. Each time you learn about a new technique or new problem type, you should write up a short code recipe that demonstrates it. This will give you a starting point to use on your next machine learning project.

As part of this book you will receive a catalog of XGBoost recipes. This includes recipes for all of the tutorials presented in this book. You are strongly encouraged to add to and build upon this catalog of recipes as you expand your use and knowledge of XGBoost in Python.

1.2 Requirements For This Book

1.2.1 Python and SciPy

You do not need to be a Python expert, but it would be helpful if you knew how to install and setup Python and SciPy. The tutorials assume that you have a Python and SciPy environment available. This may be on your workstation or laptop, it may be in a VM or a Docker instance that you run, or it may be a server instance that you can configure in the cloud as taught in Part III of this book.

Technical Requirements: The technical requirements for the code and tutorials in this book are as follows:

- Python version 2 or 3 installed. This book was developed using Python version 2.7.11.
- SciPy and NumPy installed. This book was developed with SciPy version 0.17.0 and NumPy version 1.11.0.
- Matplotlib installed. This book was developed with Matplotlib version 1.5.1.
- Pandas installed. This book was developed with Pandas version 0.18.0.
- scikit-learn installed. This book was developed with scikit-learn 0.18.1.

You do not need to match the version exactly, but if you are having problems running a specific code example, please ensure that you update to the same or higher version as the library specified.

1.2.2 Machine Learning

You do not need to be a machine learning expert, but it would be helpful if you knew how to navigate a small machine learning problem using scikit-learn. Basic concepts like cross-validation and one hot encoding used in tutorials are described, but only briefly. There are resources to go into these topics in more detail at the end of the book, but some knowledge of these areas might make things easier for you.

1.2.3 Gradient Boosting

You do not need to know the math and theory of gradient boosting algorithms, but it would be helpful to have some basic idea of the field. You will get a crash course in gradient boosting terminology and models, but we will not go into much technical detail. Again, there will be resources for more information at the end of the book, but it might be helpful if you can start with some idea about the technical details of the technique.

1.3 Your Outcomes From Reading This Book

This book will lead you from being a developer who is interested in XGBoost with Python to a developer who has the resources and capabilities to work through a new dataset end-to-end using Python and develop accurate gradient boosted models. Specifically, you will know:

- How to prepare data in scikit-learn for gradient boosting.
- How to evaluate and visualize gradient boosted models.
- How to save and load trained gradient boosted models.
- How to visualize, evaluate the importance of input variables.
- How to configure and tune hyperparameters of gradient boosted models.

There are a few ways you can read this book. You can dip into the tutorials as your need or interests motivate you. Alternatively, you can work through the book end-to-end and take advantage of how the tutorials build in complexity and range. I recommend the latter approach.

To get the very most from this book, I recommend taking each tutorials and build upon them. Attempt to improve the results, apply the method to a similar but different problem, and so on. Write up what you tried or learned and share it on your blog, social media or send me an email at `jason@MachineLearningMastery.com`. This book is really what you make of it and by putting in a little extra, you can quickly become a true force in applied gradient boosting.

1.4 What This Book is Not

This book solves a specific problem of getting you, a developer, up to speed applying XGBoost to your own machine learning projects in Python. As such, this book was not intended to be everything to everyone and it is very important to calibrate your expectations. Specifically:

- **This is not a gradient boosting textbook**. We will not cover the basic theory of how algorithms and related techniques work. There will be no equations. You are also expected to have some familiarity with machine learning basics, or be able to dive deeper into the theory yourself, if needed.
- **This is not a Python programming book**. We will not be spending a lot of time on Python syntax and programming (e.g. basic programming tasks in Python). You are expected to already be familiar with Python or a developer who can pick up a new C-like language relatively quickly.

You can still get a lot out of this book if you are weak in one or two of these areas, but you may struggle picking up the language or require some more explanation of the techniques. If this is the case, see the *Getting More Help* chapter at the end of the book and seek out a good companion reference text.

1.5 Summary

It is a special time right now. The tools for fast gradient boosting have never been so good and XGBoost is the top of the stack. The pace of change with applied machine learning feels like it has never been so fast, spurred by the amazing results that the methods are showing in such a broad range of fields. This is the start of your journey into using XGBoost and I am excited for you. Take your time, have fun and I'm so excited to see where you can take this amazing new technology.

Next in Part II, you will get a gentle introduction to the gradient boosting algorithm as described by primary sources. This will lay the foundation for understanding and using the XGBoost library in Python.

Part II

XGBoost Basics

Chapter 2

A Gentle Introduction to Gradient Boosting

Gradient boosting is one of the most powerful techniques for building predictive models. In this tutorial you will discover the gradient boosting machine learning algorithm and get a gentle introduction into where it came from and how it works. After reading this tutorial, you will know:

- The origin of boosting from learning theory and AdaBoost.
- How gradient boosting works including the loss function, weak learners and the additive model.
- How to improve performance over the base algorithm with various regularization schemes.

Let's get started.

2.1 Origin of Boosting

The idea of boosting came out of the idea of whether a weak learner can be modified to become better. Michael Kearns articulated the goal as the *Hypothesis Boosting Problem* stating the goal from a practical standpoint as:

> ... an efficient algorithm for converting relatively poor hypotheses into very good hypotheses
>
> – *Thoughts on Hypothesis Boosting*, 1988.

A weak hypothesis or weak learner is defined as one whose performance is at least slightly better than random chance. These ideas built upon Leslie Valiant's work on distribution free or Probability Approximately Correct (PAC) learning, a framework for investigating the complexity of machine learning problems. Hypothesis boosting was the idea of filtering observations, leaving those observations that the weak learner can handle and focusing on developing new weak learns to handle the remaining difficult observations.

> The idea is to used the weak learning method several times to get a succession of hypotheses, each one refocused on the examples that the previous ones found difficult and misclassified. [...] Note, however, it is not obvious at all how this can be done
>
> – *Probably Approximately Correct*, page 152, 2013.

2.2 AdaBoost the First Boosting Algorithm

The first realization of boosting that saw great success in application was Adaptive Boosting or AdaBoost for short.

> Boosting refers to this general problem of producing a very accurate prediction rule by combining rough and moderately inaccurate rules-of-thumb.
>
> – *A decision-theoretic generalization of on-line learning and an application to boosting*, 1995.

The weak learners in AdaBoost are decision trees with a single split, called decision stumps for their shortness. AdaBoost works by weighting the observations, putting more weight on difficult to classify instances and less on those already handled well. New weak learners are added sequentially that focus their training on the more difficult patterns.

> This means that samples that are difficult to classify receive increasing larger weights until the algorithm identifies a model that correctly classifies these samples
>
> – *Applied Predictive Modeling*, 2013.

Predictions are made by majority vote of the weak learners' predictions, weighted by their individual accuracy. The most successful form of the AdaBoost algorithm was for binary classification problems and was called AdaBoost.M1.

2.3 Generalization of AdaBoost as Gradient Boosting

AdaBoost and related algorithms were recast in a statistical framework first by Breiman calling them ARCing algorithms.

> Arcing is an acronym for Adaptive Reweighting and Combining. Each step in an arcing algorithm consists of a weighted minimization followed by a recomputation of [the classifiers] and [weighted input].
>
> – *Prediction Games and Arching Algorithms*, 1997.

This framework was further developed by Friedman and called Gradient Boosting Machines. Later called just gradient boosting or gradient tree boosting. The statistical framework cast boosting as a numerical optimization problem where the objective is to minimize the loss of the model by adding weak learners using a gradient descent like procedure. This class of algorithms were described as a stage-wise additive model. This is because one new weak learner is added at a time and existing weak learners in the model are frozen and left unchanged.

> Note that this stagewise strategy is different from stepwise approaches that readjust previously entered terms when new ones are added.
>
> – *Greedy Function Approximation: A Gradient Boosting Machine*, 1999.

The generalization allowed arbitrary differentiable loss functions to be used, expanding the technique beyond binary classification problems to support regression, multiclass classification and more.

2.4 How Gradient Boosting Works

Gradient boosting involves three elements:

1. A loss function to be optimized.
2. A weak learner to make predictions.
3. An additive model to add weak learners to minimize the loss function.

2.4.1 Loss Function

The loss function used depends on the type of problem being solved. It must be differentiable, but many standard loss functions are supported and you can define your own. For example, regression may use a squared error and classification may use logarithmic loss. A benefit of the gradient boosting framework is that a new boosting algorithm does not have to be derived for each loss function that may want to be used, instead, it is a generic enough framework that any differentiable loss function can be used.

2.4.2 Weak Learner

Decision trees are used as the weak learner in gradient boosting. Specifically regression trees are used that output real values for splits and whose output can be added together, allowing subsequent models outputs to be added and *correct* the residuals in the predictions. Trees are constructed in a greedy manner, choosing the best split points based on purity scores like Gini or to minimize the loss.

Initially, such as in the case of AdaBoost, very short decision trees were used that only had a single split, called a decision stump. Larger trees can be used generally with 4-to-8 levels. It is common to constrain the weak learners in specific ways, such as a maximum number of layers, nodes, splits or leaf nodes. This is to ensure that the learners remain weak, but can still be constructed in a greedy manner.

2.4.3 Additive Model

Trees are added one at a time, and existing trees in the model are not changed. A gradient descent procedure is used to minimize the loss when adding trees. Traditionally, gradient descent is used to minimize a set of parameters, such as the coefficients in a regression equation or

weights in a neural network. After calculating error or loss, the weights are updated to minimize that error.

Instead of parameters, we have weak learner sub-models or more specifically decision trees. After calculating the loss, to perform the gradient descent procedure, we must add a tree to the model that reduces the loss (i.e. follow the gradient). We do this by parameterizing the tree, then modify the parameters of the tree and move in the right direction by (reducing the residual loss. Generally this approach is called functional gradient descent or gradient descent with functions.

> One way to produce a weighted combination of classifiers which optimizes [the cost] is by gradient descent in function space
>
> – *Boosting Algorithms as Gradient Descent in Function Space*, 1999.

The output for the new tree is then added to the output of the existing sequence of trees in an effort to correct or improve the final output of the model. A fixed number of trees are added or training stops once loss reaches an acceptable level or no longer improves on an external validation dataset.

2.5 Improvements to Basic Gradient Boosting

Gradient boosting is a greedy algorithm and can overfit a training dataset quickly. It can benefit from regularization methods that penalize various parts of the algorithm and generally improve the performance of the algorithm by reducing overfitting. In this this section we will look at 4 enhancements to basic gradient boosting:

1. Tree Constraints.
2. Shrinkage.
3. Random Sampling.
4. Penalized Learning.

2.5.1 Tree Constraints

It is important that the weak learners have skill but remain weak. There are a number of ways that the trees can be constrained. A good general heuristic is that the more constrained tree creation is, the more trees you will need in the model, and the reverse, where less constrained individual trees, the fewer trees that will be required. Below are some constraints that can be imposed on the construction of decision trees:

- Number of trees, generally adding more trees to the model can be very slow to overfit. The advice is to keep adding trees until no further improvement is observed.
- Tree depth, deeper trees are more complex trees and shorter trees are preferred. Generally, better results are seen with 4-8 levels.

- Number of nodes or number of leaves, like depth, this can constrain the size of the tree, but is not constrained to a symmetrical structure if other constraints are used.
- Number of observations per split imposes a minimum constraint on the amount of training data at a training node before a split can be considered
- Minimum improvement to loss is a constraint on the improvement of any split added to a tree.

2.5.2 Weighted Updates

The predictions of each tree are added together sequentially. The contribution of each tree to this sum can be weighted to slow down the learning by the algorithm. This weighting is called a shrinkage or a learning rate.

> Each update is simply scaled by the value of the "learning rate parameter v"
>
> – *Greedy Function Approximation: A Gradient Boosting Machine*, 1999.

The effect is that learning is slowed down, in turn require more trees to be added to the model, in turn taking longer to train, providing a configuration trade-off between the number of trees and learning rate.

> Decreasing the value of v [the learning rate] increases the best value for M [the number of trees].
>
> – *Greedy Function Approximation: A Gradient Boosting Machine*, 1999.

It is common to have small values in the range of 0.1 to 0.3, as well as values less than 0.1.

> Similar to a learning rate in stochastic optimization, shrinkage reduces the influence of each individual tree and leaves space for future trees to improve the model.
>
> – *Stochastic Gradient Boosting*, 1999.

2.5.3 Stochastic Gradient Boosting

A big insight into bagging ensembles and random forest was allowing trees to be greedily created from subsamples of the training dataset. This same benefit can be used to reduce the correlation between the trees in the sequence in gradient boosting models. This variation of boosting is called stochastic gradient boosting.

> at each iteration a subsample of the training data is drawn at random (without replacement) from the full training dataset. The randomly selected subsample is then used, instead of the full sample, to fit the base learner.
>
> – *Stochastic Gradient Boosting*, 1999.

A few variants of stochastic boosting that can be used:

- Subsample rows before creating each tree.
- Subsample columns before creating each tree
- Subsample columns before considering each split.

Generally, aggressive sub-sampling such as selecting only 50% of the data has shown to be beneficial.

> According to user feedback, using column sub-sampling prevents over-fitting even more so than the traditional row sub-sampling
>
> – *XGBoost: A Scalable Tree Boosting System*, 2016.

2.5.4 Penalized Gradient Boosting

Additional constraints can be imposed on the parameterized trees in addition to their structure. Classical decision trees like CART are not used as weak learners, instead a modified form called a regression tree is used that has numeric values in the leaf nodes (also called terminal nodes). The values in the leaves of the trees can be called weights in some literature. As such, the leaf weight values of the trees can be regularized using popular regularization functions, such as:

- L1 regularization of weights.
- L2 regularization of weights.

> The additional regularization term helps to smooth the final learnt weights to avoid over-fitting. Intuitively, the regularized objective will tend to select a model employing simple and predictive functions.
>
> – *XGBoost: A Scalable Tree Boosting System*, 2016.

2.6 Summary

In this tutorial you discovered the gradient boosting algorithm for predictive modeling in machine learning. You now have an understanding of the gradient boosting algorithm in general as well as common variations of the technique. Specifically you learned:

- The history of boosting in learning theory and AdaBoost.
- How the gradient boosting algorithm works with a loss function, weak learners and an additive model.
- How to improve the performance of gradient boosting with regularization.

In the next section you will start using the XGBoost library, beginning with a gentle introduction to the XGBoost library itself.

Chapter 3

A Gentle Introduction to XGBoost

XGBoost is an algorithm that has recently been dominating applied machine learning and Kaggle competitions for structured or tabular data. XGBoost is an implementation of gradient boosted decision trees designed for speed and performance. In this tutorial you will discover XGBoost and get a gentle introduction to what is, where it came from and how you can learn more. After reading this tutorial you will know:

- What XGBoost is and the goals of the project.
- Why XGBoost must be apart of your machine learning toolkit.
- Where you can learn more to start using XGBoost on your next machine learning project.

Let's get started.

3.1 What is XGBoost?

XGBoost stands for eXtreme Gradient Boosting.

> The name xgboost, though, actually refers to the engineering goal to push the limit of computations resources for boosted tree algorithms. Which is the reason why many people use xgboost.
>
> – *Tianqi Chen*, on Quora.com.

It is an implementation of gradient boosting machines created by Tianqi Chen, now with contributions from many developers. It belongs to a broader collection of tools under the umbrella of the Distributed Machine Learning Community or DMLC[1] who are also the creators of the popular `mxnet` deep learning library. Tianqi Chen provides a brief and interesting back story on the creation of XGBoost in the tutorial *Story and Lessons Behind the Evolution of XGBoost*[2]. XGBoost is a software library that you can download and install on your machine, then access from a variety of interfaces. Specifically, XGBoost supports the following main interfaces:

[1] http://dmlc.ml
[2] http://goo.gl/QiIq99

- Command Line Interface (CLI).
- C++ (the language in which the library is written).
- Python interface as well as a model in scikit-learn.
- R interface as well as a model in the caret package.
- Julia support.
- Java and JVM languages like Scala and platforms like Hadoop.

3.2 XGBoost Features

The library is laser focused on computational speed and model performance, as such there are few frills. Nevertheless, it does offer a number of advanced features.

3.2.1 Model Features

The implementation of the model supports the features of the scikit-learn and R implementations, with new additions like regularization. Three main forms of gradient boosting are supported:

- Gradient Boosting algorithm also called gradient boosting machine including the learning rate.
- Stochastic Gradient Boosting with sub-sampling at the row, column and column per split levels.
- Regularized Gradient Boosting with both L1 and L2 regularization.

3.2.2 System Features

The library provides a system for use in a range of computing environments, not least:

- Parallelization of tree construction using all of your CPU cores during training.
- Distributed Computing for training very large models using a cluster of machines.
- Out-of-Core Computing for very large datasets that don't fit into memory.
- Cache Optimization of data structures and algorithm to make best use of hardware.

3.2.3 Algorithm Features

The implementation of the algorithm was engineered for efficiency of compute time and memory resources. A design goal was to make the best use of available resources to train the model. Some key algorithm implementation features include:

- Sparse Aware implementation with automatic handling of missing data values.

- Block Structure to support the parallelization of tree construction.
- Continued Training so that you can further boost an already fitted model on new data.

XGBoost is free open source software available for use under the permissive Apache-2 license.

3.3 Why Use XGBoost?

The two reasons to use XGBoost are also the two goals of the project:

1. Execution Speed.
2. Model Performance.

3.3.1 XGBoost Execution Speed

Generally, XGBoost is fast. Really fast when compared to other implementations of gradient boosting. Szilard Pafka performed some objective benchmarks comparing the performance of XGBoost to other implementations of gradient boosting and bagged decision trees. He wrote up his results in May 2015 in the blog tutorial titled *Benchmarking Random Forest Implementations*[3].

He also provides all the code on GitHub[4] and a more extensive report of results with hard numbers. His results showed that XGBoost was almost always faster than the other benchmarked implementations from R, Python Spark and H2O. From his experiment, he commented:

> I also tried xgboost, a popular library for boosting which is capable to build random forests as well. It is fast, memory efficient and of high accuracy
>
> – *Benchmarking Random Forest Implementations*, Szilard Pafka.

3.3.2 XGBoost Model Performance

XGBoost dominates structured or tabular datasets on classification and regression predictive modeling problems. The evidence is that it is the go-to algorithm for competition winners on the Kaggle competitive data science platform. For example, there is an incomplete list of first, second and third place competition winners that used titled: *XGBoost: Machine Learning Challenge Winning Solutions*[5]. To make this point more tangible, below are some insightful quotes from Kaggle competition winners:

> As the winner of an increasing amount of Kaggle competitions, XGBoost showed us again to be a great all-round algorithm worth having in your toolbox.
>
> – *Dato Winners' Interview*, Mad Professors[6].

[3] http://datascience.la/benchmarking-random-forest-implementations/
[4] https://github.com/szilard/benchm-ml
[5] https://github.com/dmlc/xgboost/tree/master/demo
[6] http://goo.gl/AHkmWx

> When in doubt, use xgboost.
>
> – *Avito Winner's Interview*, Owen Zhang[7].

> I love single models that do well, and my best single model was an XGBoost that could get the 10th place by itself.
>
> – *Caterpillar Winners' Interview*[8].

> I only used XGBoost.
>
> – *Liberty Mutual Property Inspection Winner's Interview*, Qingchen Wang[9].

> The only supervised learning method I used was gradient boosting, as implemented in the excellent xgboost package.
>
> – *Recruit Coupon Purchase Winner's Interview*, Halla Yang[10].

3.4 What Algorithm Does XGBoost Use?

The XGBoost library implements the gradient boosting decision tree algorithm. This algorithm goes by lots of different names such as gradient boosting, multiple additive regression trees, stochastic gradient boosting or gradient boosting machines. Boosting is an ensemble technique where new models are added to correct the errors made by existing models. Models are added sequentially until no further improvements can be made. A popular example is the AdaBoost algorithm that weights data points that are hard to predict.

Gradient boosting is an approach where new models are created that predict the residuals or errors of prior models and then added together to make the final prediction. It is called gradient boosting because it uses a gradient descent algorithm to minimize the loss when adding new models. This approach supports both regression and classification predictive modeling problems.

3.5 Summary

In this tutorial you discovered the XGBoost including why it is so popular in applied machine learning. You learned:

- That XGBoost is a library for developing fast and high performance gradient boosting tree models.
- That XGBoost is achieving the best performance on a range of difficult machine learning tasks.
- That you can use this library from the command line, Python and R and how to get started.

In the next tutorial you will develop your first XGBoost model from scratch.

[7] http://goo.gl/sGyGtu
[8] http://goo.gl/Sku8vw
[9] http://goo.gl/0LTOBl
[10] http://goo.gl/wTUH7y

Chapter 4

Develop Your First XGBoost Model in Python with scikit-learn

XGBoost is an implementation of gradient boosted decision trees designed for speed and performance that is dominative competitive machine learning. In this tutorial you will discover how you can install and create your first XGBoost model in Python. After reading this tutorial you will know:

- How to install XGBoost on your system for use in Python.
- How to prepare data and train your first XGBoost model.
- How to make predictions using your XGBoost model.

Let's get started.

4.1 Install XGBoost for Use in Python

Assuming you have a working SciPy environment, XGBoost can be installed easily using pip. For example:

```
sudo pip install xgboost
```

Listing 4.1: Install XGBoost using pip.

To update your installation of XGBoost you can type:

```
sudo pip install --upgrade xgboost
```

Listing 4.2: Update XGBoost using pip.

An alternate way to install XGBoost if you cannot use pip or you want to run the latest code from GitHub requires that you make a clone of the XGBoost project and perform a manual build and installation. For example to build XGBoost without multithreading on Mac OS X (with GCC already installed via macports or homebrew), you can type:

```
git clone --recursive https://github.com/dmlc/xgboost
cd xgboost
cp make/minimum.mk ./config.mk
make -j4
```

```
cd python-package
sudo python setup.py install
```

Listing 4.3: Build and Install XGBoost on Mac OS X.

Below are some resources to help you with the installation of the XGBoost library on your platform.

- You can learn more about how to install XGBoost for different platforms on the *XGBoost Installation Guide*.
  http://xgboost.readthedocs.io/en/latest/build.html
- For up-to-date instructions for installing XGBoost for Python see the *XGBoost Python Package*.
  https://github.com/dmlc/xgboost/tree/master/python-package

XGBoost v0.6 is the latest at the time of writing and is used in this book.

4.2 Problem Description: Predict Onset of Diabetes

In this tutorial we are going to use the Pima Indians onset of diabetes dataset. This dataset is comprised of 8 input variables that describe medical details of patients and one output variable to indicate whether the patient will have an onset of diabetes within 5 years. You can learn more about this dataset on the UCI Machine Learning Repository website[1].

This is a good dataset for a first XGBoost model because all of the input variables are numeric and the problem is a simple binary classification problem. It is not necessarily a good problem for the XGBoost algorithm because it is a relatively small dataset and an easy problem to model.

```
6,148,72,35,0,33.6,0.627,50,1
1,85,66,29,0,26.6,0.351,31,0
8,183,64,0,0,23.3,0.672,32,1
1,89,66,23,94,28.1,0.167,21,0
0,137,40,35,168,43.1,2.288,33,1
```

Listing 4.4: Sample of the Pima Indians dataset.

Download this dataset[2] and place it into your current working directory with the file name `pima-indians-diabetes.csv`.

4.3 Load and Prepare Data

In this section we will load the data from file and prepare it for use for training and evaluating an XGBoost model. We will start off by importing the classes and functions we intend to use in this tutorial.

[1] https://archive.ics.uci.edu/ml/datasets/Pima+Indians+Diabetes
[2] https://goo.gl/1I4ntS

```
from numpy import loadtxt
from xgboost import XGBClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
```

Listing 4.5: Import Classes and Functions.

Next, we can load the CSV file as a NumPy array using the NumPy function `loadtext()`.

```
# load data
dataset = loadtxt('pima-indians-diabetes.csv', delimiter=",")
```

Listing 4.6: Load the Dataset.

We must separate the columns (attributes or features) of the dataset into input patterns (`X`) and output patterns (`Y`). We can do this easily by specifying the column indices in the NumPy array format.

```
# split data into X and y
X = dataset[:,0:8]
Y = dataset[:,8]
```

Listing 4.7: Separate Dataset into X and Y.

Finally, we must split the `X` and `Y` data into a training and test dataset. The training set will be used to prepare the XGBoost model and the test set will be used to make new predictions, from which we can evaluate the performance of the model. For this we will use the `train_test_split()` function from the scikit-learn library. We also specify a seed for the random number generator so that we always get the same split of data each time this example is executed.

```
# 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, Y, test_size=test_size,
    random_state=seed)
```

Listing 4.8: Separate Dataset into Train and Test sets.

We are now ready to train our model.

4.4 Train the XGBoost Model

XGBoost provides a wrapper class to allow models to be treated like classifiers or regressors in the scikit-learn framework. This means we can use the full scikit-learn library with XGBoost models. The XGBoost model for classification is called `XGBClassifier`. We can create and fit it to our training dataset. Models are fit using the scikit-learn API and the `model.fit()` function. Parameters for training the model can be passed to the model in the constructor. Here, we use the sensible defaults.

```
# fit model no training data
model = XGBClassifier()
model.fit(X_train, y_train)
```

Listing 4.9: Fit the XGBoost model.

You can see the parameters used in a trained model by printing the model, for example:

```
print(model)
```

Listing 4.10: Summarize the XGBoost model.

We are now ready to use the trained model to make predictions.

4.5 Make Predictions with XGBoost Model

We can make predictions using the fit model on the test dataset. To make predictions we use the scikit-learn function `model.predict()`. By default, the predictions made by XGBoost are probabilities. Because this is a binary classification problem, each prediction is the probability of the input pattern belonging to the first class. We can easily convert them to binary class values by rounding them to 0 or 1.

```
# make predictions for test data
y_pred = model.predict(X_test)
predictions = [round(value) for value in y_pred]
```

Listing 4.11: Make Predictions wit the XGBoost model.

Now that we have used the fit model to make predictions on new data, we can evaluate the performance of the predictions by comparing them to the expected values. For this we will use the built in `accuracy_score()` function in scikit-learn.

```
# evaluate predictions
accuracy = accuracy_score(y_test, predictions)
print("Accuracy: %.2f%%" % (accuracy * 100.0))
```

Listing 4.12: Evaluate Accuracy of XGBoost Predictions.

4.6 Tie it All Together

We can tie all of these pieces together, below is the full code listing.

```
# First XGBoost model for Pima Indians dataset
from numpy import loadtxt
from xgboost import XGBClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
# load data
dataset = loadtxt('pima-indians-diabetes.csv', delimiter=",")
# split data into X and y
X = dataset[:,0:8]
Y = dataset[:,8]
# 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, Y, test_size=test_size,
    random_state=seed)
# fit model no training data
model = XGBClassifier()
model.fit(X_train, y_train)
```

```
# 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))
```

Listing 4.13: Complete Working Example of Your First XGBoost Model.

Running this example produces the following output.

```
Accuracy: 77.95%
```

Listing 4.14: Sample Output From First XGBoost Model.

This is a good accuracy score on this problem[3], which we would expect, given the capabilities of the model and the modest complexity of the problem.

4.7 Summary

In this tutorial you discovered how to develop your first XGBoost model in Python. Specifically, you learned:

- How to install XGBoost on your system ready for use with Python.
- How to prepare data and train your first XGBoost model on a standard machine learning dataset.
- How to make predictions and evaluate the performance of a trained XGBoost model using scikit-learn.

In the next tutorial, you will build upon these skills and learn how to best prepare your data for modeling with XGBoost.

[3] http://www.is.umk.pl/projects/datasets.html#Diabetes

Chapter 5

Data Preparation for Gradient Boosting

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

5.1 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[1].

```
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
```

Listing 5.1: Sample of the Iris dataset.

[1] http://archive.ics.uci.edu/ml/datasets/Iris

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`. The three class values (`Iris-setosa`, `Iris-versicolor`, `Iris-virginica`) are mapped to the integer values (0, 1, 2).

```
# encode string class values as integers
label_encoder = LabelEncoder()
label_encoder = label_encoder.fit(Y)
label_encoded_y = label_encoder.transform(Y)
```

Listing 5.2: Example of LabelEncoder.

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.

```
# multiclass classification
from pandas import read_csv
from xgboost import XGBClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import LabelEncoder
# load data
data = read_csv('iris.csv', header=None)
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 = 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))
```

Listing 5.3: Example of Using LabelEncoder on the Iris Dataset.

Running the example produces the following output:

```
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%
```

Listing 5.4: Sample output of LabelEncoder and XGBoost on the Iris Dataset.

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.

5.2 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[2] and download it in CSV format from mldata.org[3].

```
'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'
```

Listing 5.5: Sample of the Breast Cancer Dataset.

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:

```
# encode string class values as integers
label_encoder = LabelEncoder()
label_encoder = label_encoder.fit(Y)
label_encoded_y = label_encoder.transform(Y)
```

Listing 5.6: Example of Using the LabelEncoder on the Output Variable.

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

```
# 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])
```

Listing 5.7: Example of Using the LabelEncoder on all Input Variables.

[2] http://archive.ics.uci.edu/ml/datasets/Breast+Cancer
[3] http://mldata.org/repository/data/viewslug/datasets-uci-breast-cancer/

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:

```
left-up
left-low
right-up
right-low
central
```

Listing 5.8: Sample Categorical Values for breast-quad Variable.

We can model this as 5 binary variables as follows:

```
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
```

Listing 5.9: One Hot Encoding of the breast-quad Variable.

This is called one hot encoding. We can one hot encode all of the categorical input variables using the `OneHotEncoder` 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.

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

Listing 5.10: Example of Reshaping an Encoded Variable.

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

```
onehot_encoder = OneHotEncoder(sparse=False)
feature = onehot_encoder.fit_transform(feature)
```

Listing 5.11: Example of using the OneHotEncoder on a Variable.

Finally, we can build up the input dataset by concatenating the one hot encoded features, one by one. We end up with an input vector comprised of 43 binary input variables.

```
# encode string input values as integers
columns = []
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)
  columns.append(feature)
# collapse columns into array
encoded_x = numpy.column_stack(columns)
```

Listing 5.12: Example of using the OneHotEncoder on all Input Variables.

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.

```
# binary classification, breast cancer dataset, label and one hot encoded
from numpy import column_stack
from pandas import read_csv
from xgboost import XGBClassifier
from sklearn.model_selection 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=None)
dataset = data.values
# split data into X and y
X = dataset[:,0:9]
Y = dataset[:,9]
# encode string input values as integers
columns = []
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)
  columns.append(feature)
# collapse columns into array
encoded_x = column_stack(columns)
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))
```

Listing 5.13: Example of Using OneHotEncoder on the Breast Cancer Dataset.

Running this example we get the following output:

```
('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: 71.58%
```

Listing 5.14: Sample output of OneHotEncoder and XGBoost on the Breast Cancer Dataset.

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

```
FutureWarning: numpy not_equal will not check object identity in the future
```

Listing 5.15: Example of a Warning You May See.

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

5.3 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*[4].

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[5]. The values are separated by whitespace and we can easily load it using the Pandas function `read_csv()`.

```
dataframe = read_csv("horse-colic.csv", delim_whitespace=True, header=None)
```

Listing 5.16: Load the Horse Colic Dataset.

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).

```
# set missing values to 0
X[X == '?'] = 0
```

Listing 5.17: Mark Missing Values with a Zero Value.

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.

```
# convert to numeric
X = X.astype('float32')
```

Listing 5.18: Convert Input Variables to Floats.

[4] https://arxiv.org/abs/1603.02754

[5] https://archive.ics.uci.edu/ml/datasets/Horse+Colic

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.

```
# encode Y class values as integers
label_encoder = LabelEncoder()
label_encoder = label_encoder.fit(Y)
label_encoded_y = label_encoder.transform(Y)
```

Listing 5.19: Apply LabelEncoder to Output Variable.

The full code listing is provided below for completeness.

```
# binary classification, missing data
from pandas import read_csv
from xgboost import XGBClassifier
from sklearn.model_selection 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))
```

Listing 5.20: Example of Missing Data Handling on the Horse Colic Dataset.

Running this example produces the following output.

```
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%
```

Listing 5.21: Sample output of Missing Data Handling with XGBoost on the Horse Colic Dataset.

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.

```
X[X == '?'] = 1
```

Listing 5.22: Mark Missing Values with a One Value.

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

```
Accuracy: 79.80%
```

Listing 5.23: Model Accuracy With Missing Values Set to One.

We can also mark the values as a NaN and let the XGBoost framework treat the missing values as a distinct value for the feature.

```
import numpy
...
X[X == '?'] = numpy.nan
```

Listing 5.24: Mark Missing Values with a NaN Value.

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

```
Accuracy: 85.86%
```

Listing 5.25: Model Accuracy With Missing Values Set to NaN.

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` class.

```
# impute missing values as the mean
imputer = Imputer()
imputed_x = imputer.fit_transform(X)
```

Listing 5.26: Example Usage of the Imputer Class.

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

```
# binary classification, missing data, impute with mean
import numpy
from pandas import read_csv
from xgboost import XGBClassifier
from sklearn.model_selection 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 NaN
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(imputed_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))
```

Listing 5.27: Example of Missing Data Handling With Imputing on the Horse Colic Dataset.

Running this example we see results equivalent to the fixing the value to one (1). This suggests that at least in this case we are better off marking the missing values with a distinct value of NaN or zero (0) rather than a valid value (1) or an imputed value.

```
Accuracy: 79.80%
```

Listing 5.28: Model Accuracy With Imputed Missing Values.

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

5.4 Summary

In this tutorial 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.

In the next tutorial you will further build upon these capabilities and discover how you can evaluate the performance of your XGBoost models.

Chapter 6

How to Evaluate XGBoost Models

The goal of developing a predictive model is to develop a model that is accurate on unseen data. This can be achieved using statistical techniques where the training dataset is carefully used to estimate the performance of the model on new and unseen data. In this tutorial you will discover how you can evaluate the performance of your gradient boosting models with XGBoost in Python. After completing this tutorial, you will know.

- How to evaluate the performance of your XGBoost models using train and test datasets.
- How to evaluate the performance of your XGBoost models using k-fold cross-validation.

Let's get started.

6.1 Evaluate Models With Train and Test Sets

The simplest method that we can use to evaluate the performance of a machine learning algorithm is to use different training and testing datasets. We can take our original dataset and split it into two parts. Train the algorithm on the first part, then make predictions on the second part and evaluate the predictions against the expected results. The size of the split can depend on the size and specifics of your dataset, although it is common to use 67% of the data for training and the remaining 33% for testing.

This algorithm evaluation technique is fast. It is ideal for large datasets (millions of records) where there is strong evidence that both splits of the data are representative of the underlying problem. Because of the speed, it is useful to use this approach when the algorithm you are investigating is slow to train. A downside of this technique is that it can have a high variance. This means that differences in the training and test dataset can result in meaningful differences in the estimate of model accuracy. We can split the dataset into a train and test set using the `train_test_split()` function from the scikit-learn library. For example, we can split the dataset into a 67% and 33% split for training and test sets as follows:

```
# split data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.33, random_state=7)
```

Listing 6.1: Example of Splitting a Dataset into Train and Test Sets.

The full code listing is provided below using the Pima Indians onset of diabetes dataset, assumed to be in the current working directory (see Section 4.2). An XGBoost model with default configuration is fit on the training dataset and evaluated on the test dataset.

```
# train-test split evaluation of xgboost model
from numpy import loadtxt
from xgboost import XGBClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
# load data
dataset = loadtxt('pima-indians-diabetes.csv', delimiter=",")
# split data into X and y
X = dataset[:,0:8]
Y = dataset[:,8]
# split data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.33, random_state=7)
# fit model no training data
model = XGBClassifier()
model.fit(X_train, y_train)
# 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))
```

Listing 6.2: XGBoost Evaluated on a Train and Test Set.

Running this example summarizes the performance of the model on the test set.

```
Accuracy: 77.95%
```

Listing 6.3: Sample of the XGBoost model with Train and Test Sets.

6.2 Evaluate Models With k-Fold Cross-Validation

Cross-validation is an approach that you can use to estimate the performance of a machine learning algorithm with less variance than a single train-test set split. It works by splitting the dataset into k-parts (e.g. $k = 5$ or $k = 10$). Each split of the data is called a fold. The algorithm is trained on $k - 1$ folds with one held back and tested on the held back fold. This is repeated so that each fold of the dataset is given a chance to be the held back test set. After running cross-validation you end up with k-different performance scores that you can summarize using a mean and a standard deviation.

The result is a more reliable estimate of the performance of the algorithm on new data given your test data. It is more accurate because the algorithm is trained and evaluated multiple times on different data. The choice of k must allow the size of each test partition to be large enough to be a reasonable sample of the problem, whilst allowing enough repetitions of the train-test evaluation of the algorithm to provide a fair estimate of the algorithms performance on unseen data. For modest sized datasets in the thousands or tens of thousands of observations, k values of 3, 5 and 10 are common.

We can use k-fold cross-validation support provided in scikit-learn. First we must create the `KFold` object specifying the number of folds and the size of the dataset. We can then use this scheme with the specific dataset. The `cross_val_score()` function from scikit-learn allows us to evaluate a model using the cross-validation scheme and returns a list of the scores for each model trained on each fold.

```
kfold = KFold(n_splits=10, random_state=7)
results = cross_val_score(model, X, Y, cv=kfold)
```

Listing 6.4: Example of Preparing k-Fold Cross-Validation.

The full code listing for evaluating an XGBoost model with k-fold cross-validation is provided below for completeness.

```
# k-fold cross validation evaluation of xgboost model
from numpy import loadtxt
from xgboost import XGBClassifier
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
# load data
dataset = loadtxt('pima-indians-diabetes.csv', delimiter=",")
# split data into X and y
X = dataset[:,0:8]
Y = dataset[:,8]
# CV model
model = XGBClassifier()
kfold = KFold(n_splits=10, random_state=7)
results = cross_val_score(model, X, Y, cv=kfold)
print("Accuracy: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
```

Listing 6.5: XGBoost Evaluated With k-Fold Cross-Validation.

Running this example summarizes the performance of the default model configuration on the dataset including both the mean and standard deviation classification accuracy.

```
Accuracy: 76.69% (7.11%)
```

Listing 6.6: Output From Evaluating XGBoost with k-Fold Cross-Validation.

If you have many classes for a classification type predictive modeling problem or the classes are imbalanced (there are a lot more instances for one class than another), it can be a good idea to create stratified folds when performing cross-validation. This has the effect of enforcing the same distribution of classes in each fold as in the whole training dataset when performing the cross-validation evaluation. The scikit-learn library provides this capability in the `StratifiedKFold` class. Below is the same example modified to use stratified cross-validation to evaluate an XGBoost model.

```
# stratified k-fold cross validation evaluation of xgboost model
from numpy import loadtxt
from xgboost import XGBClassifier
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import cross_val_score
# load data
dataset = loadtxt('pima-indians-diabetes.csv', delimiter=",")
# split data into X and y
X = dataset[:,0:8]
Y = dataset[:,8]
# CV model
model = XGBClassifier()
kfold = StratifiedKFold(n_splits=10, random_state=7)
results = cross_val_score(model, X, Y, cv=kfold)
print("Accuracy: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
```

Listing 6.7: XGBoost Evaluated With Stratified k-Fold Cross-Validation.

Running this example produces the following output.

```
Accuracy: 76.95% (5.88%)
```

Listing 6.8: Output From Evaluating XGBoost with Stratified k-Fold Cross-Validation.

You will notice that the performance is slightly higher and the variance is smaller in the result.

6.3 What Techniques to Use When

- Generally k-fold cross-validation is the gold-standard for evaluating the performance of a machine learning algorithm on unseen data with k set to 3, 5, or 10.
- Use stratified cross-validation to enforce class distributions when there are a large number of classes or an imbalance in instances for each class.
- Using a train/test split is good for speed when using a slow algorithm and produces performance estimates with lower bias when using large datasets.

The best advice is to experiment and find a technique for your problem that is fast and produces reasonable estimates of performance that you can use to make decisions. If in doubt, use 10-fold cross-validation for regression problems and stratified 10-fold cross-validation on classification problems.

6.4 Summary

In this tutorial you discovered how you can evaluate your XGBoost models by estimating how well they are likely to perform on unseen data. Specifically, you learned:

- How to split your dataset into train and test subsets for training and evaluating the performance of your model.
- How you can create k XGBoost models on different subsets of the dataset and average the scores to get a more robust estimate of model performance.
- Heuristics to help choose between train-test split and k-fold cross-validation for your problem.

In the next tutorial you will discover how you can use visualization to better understand the boosted trees inside a trained mode.

Chapter 7

Visualize Individual Trees Within A Model

Plotting individual decision trees can provide insight into the gradient boosting process for a given dataset. In this tutorial you will discover how you can plot individual decision trees from a trained gradient boosting model using XGBoost in Python. Let's get started.

7.1 Plot a Single XGBoost Decision Tree

The XGBoost Python API provides a function for plotting decision trees within a trained XGBoost model. This capability is provided in the `plot_tree()` function that takes a trained model as the first argument, for example:

```
plot_tree(model)
```

Listing 7.1: Example of Plotting The First Tree in the Model.

This plots the first tree in the model (the tree at index 0). This plot can be saved to file or shown on the screen using Matplotlib and `pyplot.show()`. This plotting capability requires that you have the `graphviz` library installed[1]. We can create an XGBoost model on the Pima Indians onset of diabetes dataset and plot the first tree in the model (see Section 4.2). The full code listing is provided below.

```
# plot decision tree
from numpy import loadtxt
from xgboost import XGBClassifier
from xgboost import plot_tree
from matplotlib import pyplot
# load data
dataset = loadtxt('pima-indians-diabetes.csv', delimiter=",")
# split data into X and y
X = dataset[:,0:8]
y = dataset[:,8]
# fit model no training data
model = XGBClassifier()
model.fit(X, y)
# plot single tree
```

[1] http://www.graphviz.org/

```
plot_tree(model)
pyplot.show()
```

Listing 7.2: Example of Plotting the First Tree in the XGBoost Model.

Running the code creates a plot of the first decision tree in the model (index 0), showing the features and feature values for each split as well as the output leaf nodes.

Figure 7.1: XGBoost Plot of Single Decision Tree

You can see that variables are automatically named like `f1` and `f5` corresponding with the feature indices in the input array. You can see the split decisions within each node and the different colors for left and right splits (blue and red). The `plot_tree()` function takes some parameters. You can plot specific graphs by specifying their index to the `num_trees` argument. For example, you can plot the 5^{th} boosted tree in the sequence as follows:

```
plot_tree(model, num_trees=4)
```

Listing 7.3: Plot a Specific Decision Tree within an XGBoost Model.

You can also change the layout of the graph to be left to right (easier to read) by changing the `rankdir` argument as `LR` (left-to-right) rather than the default top to bottom (UT). For example:

```
plot_tree(model, num_trees=0, rankdir='LR')
```

Listing 7.4: Use Left-to-Right Layout When Plotting Trees.

The result of plotting the tree in the left-to-right layout is shown below.

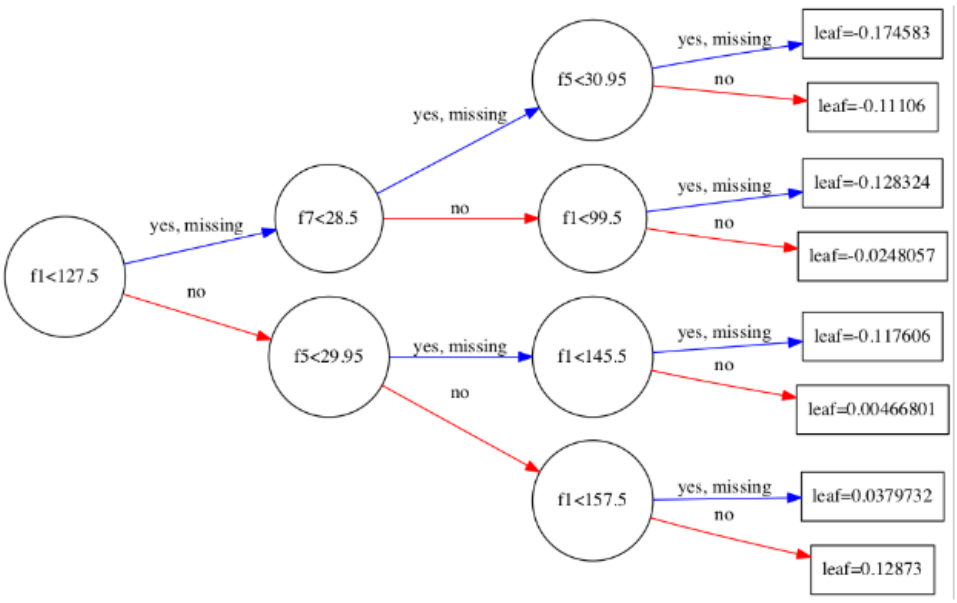

Figure 7.2: XGBoost Plot of Single Decision Tree Left-To-Right

7.2 Summary

In this tutorial you learned how to plot individual decision trees from a trained XGBoost gradient boosted model in Python. This concludes Part II of this book. Next in Part III you will build upon these basic skills with XGBoost, starting with how to save and load an XGBoost model to file.

Part III

XGBoost Advanced

Chapter 8

Save and Load Trained XGBoost Models

XGBoost can be used to create some of the most performant models for tabular data using the gradient boosting algorithm. Once trained, it is often a good practice to save your model to file for later use in making predictions new test and validation datasets and entirely new data. In this tutorial you will discover how to save your XGBoost models to file using the standard Python pickle API. After completing this tutorial, you will know:

- How to save and later load your trained XGBoost model using pickle.
- How to save and later load your trained XGBoost model using joblib.

Let's get started.

8.1 Serialize Models with Pickle

Pickle is the standard way of serializing objects in Python. You can use the Python pickle API[1] to serialize your machine learning algorithms and save the serialized format to a file, for example:

```
# save model to file
pickle.dump(model, open("pima.pickle.dat", "wb"))
```

Listing 8.1: Example Saving an XGBoost Model using Pickle.

Later you can load this file to deserialize your model and use it to make new predictions, for example:

```
# load model from file
loaded_model = pickle.load(open("pima.pickle.dat", "rb"))
```

Listing 8.2: Example Loading an XGBoost Model using Pickle.

The example below demonstrates how you can train a XGBoost model on the Pima Indians onset of diabetes dataset (see Section 4.2), save the model to file and later load it to make predictions. The full code listing is provided below for completeness.

[1] https://docs.python.org/2/library/pickle.html

```
# Train XGBoost model, save to file using pickle, load and make predictions
from numpy import loadtxt
from xgboost import XGBClassifier
import pickle
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
# load data
dataset = loadtxt('pima-indians-diabetes.csv', delimiter=",")
# split data into X and y
X = dataset[:,0:8]
Y = dataset[:,8]
# 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, Y, test_size=test_size,
    random_state=seed)
# fit model no training data
model = XGBClassifier()
model.fit(X_train, y_train)
# save model to file
pickle.dump(model, open("pima.pickle.dat", "wb"))
print("Saved model to: pima.pickle.dat")

# some time later...

# load model from file
loaded_model = pickle.load(open("pima.pickle.dat", "rb"))
print("Loaded model from: pima.pickle.dat")
# make predictions for test data
y_pred = loaded_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))
```

Listing 8.3: Worked Example of Saving and Loading an XGBoost Model with Pickle.

Running this example saves your trained XGBoost model to the `pima.pickle.dat` pickle file in the current working directory. After loading the model and making predictions on the training dataset, the accuracy of the model is printed.

```
Saved model to: pima.pickle.dat
Loaded model from: pima.pickle.dat
Accuracy: 77.95%
```

Listing 8.4: Sample Output From Worked Example of Saving and Loading an XGBoost Model with Pickle.

8.2 Serialize Models with Joblib

Joblib is part of the SciPy ecosystem and provides utilities for pipelining Python jobs. The Joblib API[2] provides utilities for saving and loading Python objects that make use of NumPy

[2] https://pypi.python.org/pypi/joblib

data structures, efficiently. It may be a faster approach for you to use with very large models. The API looks a lot like the pickle API, for example, you may save your trained model as follows:

```
# save model to file
joblib.dump(model, "pima.joblib.dat")
```

Listing 8.5: Example Saving an XGBoost Model using Joblib.

You can later load the model from file and use it to make predictions as follows:

```
# load model from file
loaded_model = joblib.load("pima.joblib.dat")
```

Listing 8.6: Example Loading an XGBoost Model using Joblib.

The example below demonstrates how you can train an XGBoost model for classification on the Pima Indians onset of diabetes dataset, save the model to file using Joblib and load it at a later time in order to make predictions.

```
# Train XGBoost model, save to file using joblib, load and make predictions
from numpy import loadtxt
from xgboost import XGBClassifier
from sklearn.externals import joblib
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
# load data
dataset = loadtxt('pima-indians-diabetes.csv', delimiter=",")
# split data into X and y
X = dataset[:,0:8]
Y = dataset[:,8]
# 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, Y, test_size=test_size,
    random_state=seed)
# fit model no training data
model = XGBClassifier()
model.fit(X_train, y_train)
# save model to file
joblib.dump(model, "pima.joblib.dat")
print("Saved model to: pima.joblib.dat")

# some time later...

# load model from file
loaded_model = joblib.load("pima.joblib.dat")
print("Loaded model from: pima.joblib.dat")
# make predictions for test data
y_pred = loaded_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))
```

Listing 8.7: Worked Example of Saving and Loading an XGBoost Model with Joblib.

Running the example saves the model to file as `pima.joblib.dat` in the current working directory and also creates one file for each NumPy array within the model (in this case two additional files).

```
pima.joblib.dat
pima.joblib.dat_01.npy
pima.joblib.dat_02.npy
```

Listing 8.8: List of Files Created When Serializing a Model with Joblib.

After the model is loaded, it is evaluated on the training dataset and the accuracy of the predictions is printed.

```
Saved model to: pima.joblib.dat
Loaded model from: pima.joblib.dat
Accuracy: 77.95%
```

Listing 8.9: Sample Output From Worked Example of Saving and Loading an XGBoost Model with Joblib.

8.3 Summary

In this tutorial you discovered how to serialize your trained XGBoost models and later load them in order to make predictions. Specifically, you learned:

- How to serialize and later load your trained XGBoost model using the pickle API.
- How to serialize and later load your trained XGBoost model using the Joblib API.

In the next tutorial you will discover how you can use feature importance calculated by an XGBoost model to better understand your dataset and even perform feature selection.

Chapter 9

Feature Importance With XGBoost and Feature Selection

A benefit of using ensembles of decision tree methods like gradient boosting is that they can automatically provide estimates of feature importance from a trained predictive model. In this tutorial you will discover how you can estimate the importance of features for a predictive modeling problem using the XGBoost library in Python. After reading this tutorial you will know:

- How feature importance is calculated using the gradient boosting algorithm.
- How to plot feature importance in Python calculated by the XGBoost model.
- How to use feature importance calculated by XGBoost to perform feature selection.

Let's get started.

9.1 Feature Importance in Gradient Boosting

A benefit of using gradient boosting is that after the boosted trees are constructed, it is relatively straightforward to retrieve importance scores for each attribute. Generally, importance provides a score that indicates how useful or valuable each feature was in the construction of the boosted decision trees within the model. The more an attribute is used to make key decisions with decision trees, the higher its relative importance. This importance is calculated explicitly for each attribute in the dataset, allowing attributes to be ranked and compared to each other.

Importance is calculated for a single decision tree by the amount that each attribute split point improves the performance measure, weighted by the number of observations the node is responsible for. The performance measure may be the purity (Gini index) used to select the split points or another more specific error function. The feature importances are then averaged across all of the decision trees within the model. For more technical information on how feature importance is calculated in boosted decision trees, see Section 10.13.1 *Relative Importance of Predictor Variables* of the book *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, page 367.

9.2 Manually Plot Feature Importance

A trained XGBoost model automatically calculates feature importance on your predictive modeling problem. These importance scores are available in the `feature_importances_` member variable of the trained model. For example, they can be printed directly as follows:

```
print(model.feature_importances_)
```

Listing 9.1: Example of Printing Feature Importances from an XGBoost Model.

We can plot these scores on a bar chart directly to get a visual indication of the relative importance of each feature in the dataset. For example:

```
# plot
pyplot.bar(range(len(model.feature_importances_)), model.feature_importances_)
pyplot.show()
```

Listing 9.2: Example of Plotting Feature Importances from an XGBoost Model.

We can demonstrate this by training an XGBoost model on the Pima Indians onset of diabetes dataset (see Section 4.2) and creating a bar chart from the calculated feature importances.

```
# plot feature importance manually
from numpy import loadtxt
from xgboost import XGBClassifier
from matplotlib import pyplot
# load data
dataset = loadtxt('pima-indians-diabetes.csv', delimiter=",")
# split data into X and y
X = dataset[:,0:8]
y = dataset[:,8]
# fit model no training data
model = XGBClassifier()
model.fit(X, y)
# feature importance
print(model.feature_importances_)
# plot
pyplot.bar(range(len(model.feature_importances_)), model.feature_importances_)
pyplot.show()
```

Listing 9.3: Worked Example of Manually Plotting Feature Importance.

Running this example first outputs the importance scores:

```
[ 0.089701    0   .17109634  0.08139535  0.04651163  0.10465116  0.2026578  0.1627907
    0 .14119601]
```

Listing 9.4: Print of Raw Importance Scores.

We also get a bar chart of the relative importances.

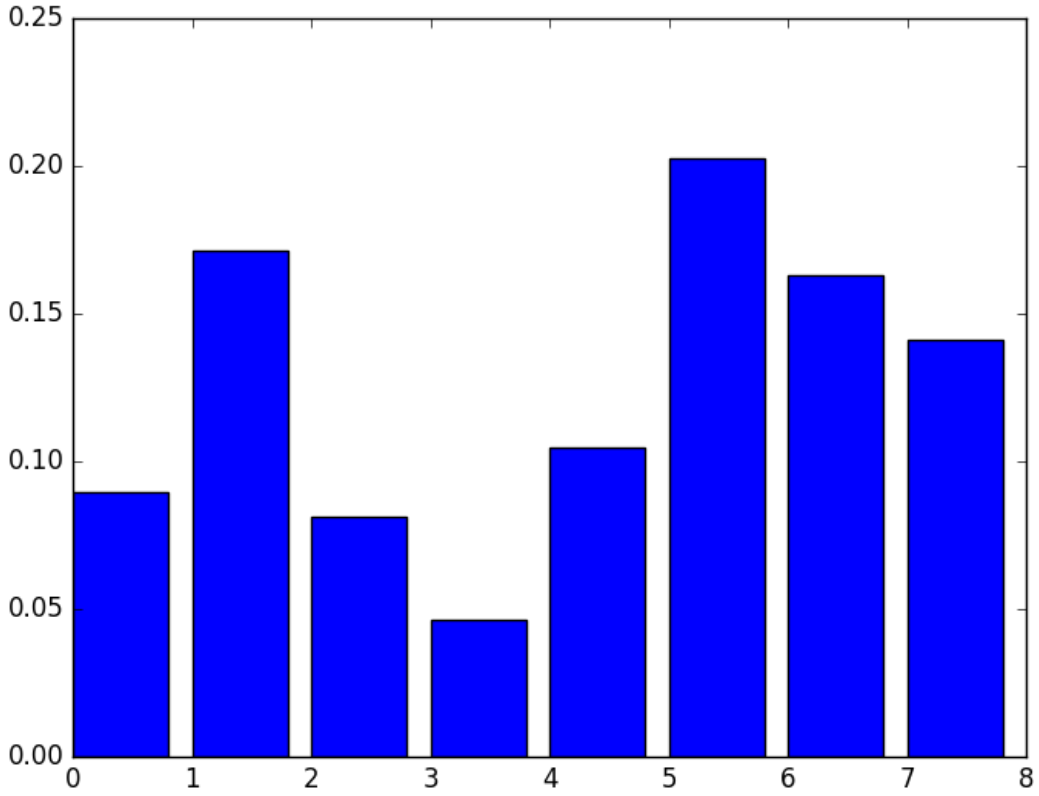

Figure 9.1: Manual Bar Chart of XGBoost Feature Importance

A downside of this plot is that the features are ordered by their input index rather than their importance. We could sort the features before plotting. Thankfully, there is a built in plot function to help us.

9.3 Using theBuilt-in XGBoost Feature Importance Plot

The XGBoost library provides a built-in function to plot features ordered by their importance. The function is called `plot_importance()` and can be used as follows:

```
# plot feature importance
plot_importance(model)
pyplot.show()
```

Listing 9.5: Example of Built-in Function To Plot Feature Importances.

For example, below is a complete code listing plotting the feature importance for the Pima Indians dataset using the built-in `plot_importance()` function.

```
# plot feature importance using built-in function
from numpy import loadtxt
from xgboost import XGBClassifier
from xgboost import plot_importance
```

```
from matplotlib import pyplot
# load data
dataset = loadtxt('pima-indians-diabetes.csv', delimiter=",")
# split data into X and y
X = dataset[:,0:8]
y = dataset[:,8]
# fit model no training data
model = XGBClassifier()
model.fit(X, y)
# plot feature importance
plot_importance(model)
pyplot.show()
```

Listing 9.6: Worked Example of Automatically Plotting Feature Importance.

Running the example gives us a more useful bar chart.

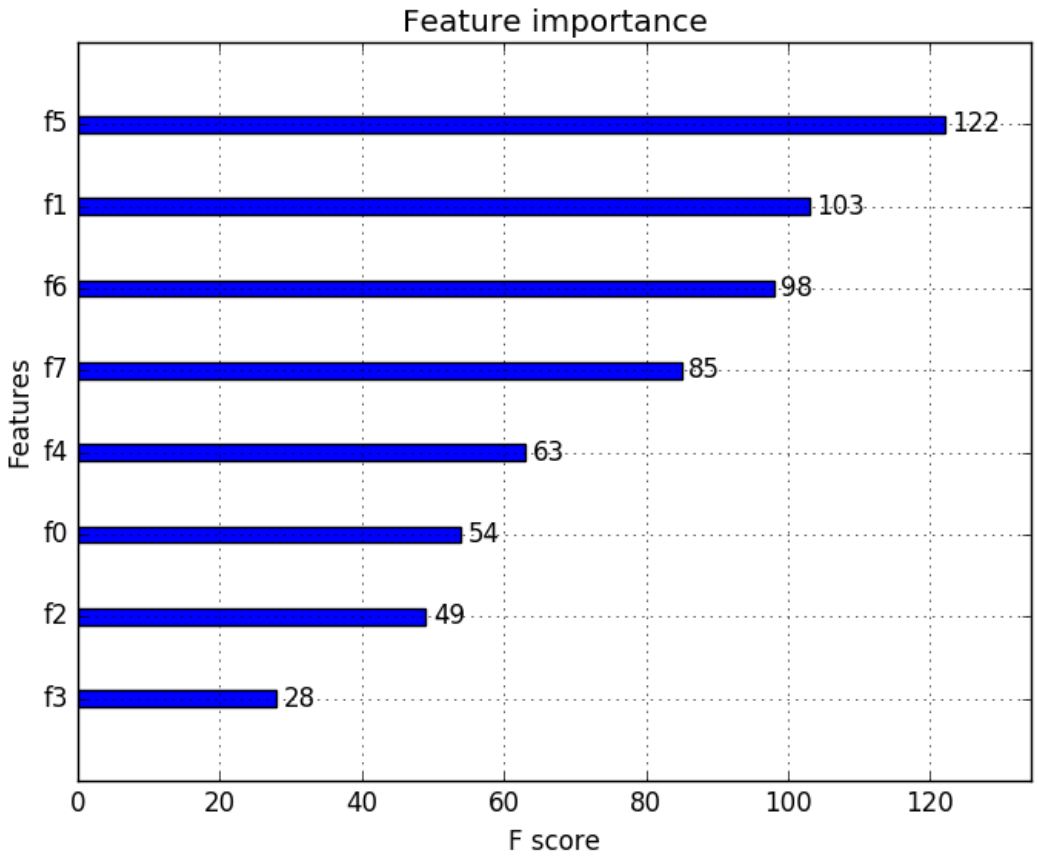

Figure 9.2: Bar Chart of XGBoost Feature Importance

You can see that features are automatically named according to their index in the input array (`X`) from `F0` to `F7`. Manually mapping these indices to names in the problem description[1], we can see that the plot shows `F5` (body mass index) has the highest importance and `F3` (skin fold thickness) has the lowest importance.

[1] https://goo.gl/OhIfBw

9.4 Feature Selection with XGBoost Feature Importance Scores

Feature importance scores can be used for feature selection in scikit-learn. This is done using the `SelectFromModel` class that takes a model and can transform a dataset into a subset with selected features. This class can take a pre-trained model, such as one trained on the entire training dataset. It can then use a threshold to decide which features to select. This threshold is used when you call the `transform()` function on the `SelectFromModel` instance to consistently select the same features on the training dataset and the test dataset.

In the example below we first train and then evaluate an XGBoost model on the entire training dataset and test datasets respectively. Using the feature importances calculated from the training dataset, we then wrap the model in a `SelectFromModel` instance. We use this to select features on the training dataset, train a model from the selected subset of features, then evaluate the model on the testset, subject to the same feature selection scheme. For example:

```
# select features using threshold
selection = SelectFromModel(model, threshold=thresh, prefit=True)
select_X_train = selection.transform(X_train)
# train model
selection_model = XGBClassifier()
selection_model.fit(select_X_train, y_train)
# eval model
select_X_test = selection.transform(X_test)
y_pred = selection_model.predict(select_X_test)
```

Listing 9.7: Example of Feature Selection Using XGBoost Feature Importance.

For interest, we can test multiple thresholds for selecting features by feature importance. Specifically, the feature importance of each input variable, essentially allowing us to test each subset of features by importance, starting with all features and ending with a subset with the most important feature. The complete code listing is provided below.

```
# use feature importance for feature selection
from numpy import loadtxt
from numpy import sort
from xgboost import XGBClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.feature_selection import SelectFromModel
# load data
dataset = loadtxt('pima-indians-diabetes.csv', delimiter=",")
# split data into X and y
X = dataset[:,0:8]
Y = dataset[:,8]
# split data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.33, random_state=7)
# fit model on all training data
model = XGBClassifier()
model.fit(X_train, y_train)
# make predictions for test data and evaluate
y_pred = model.predict(X_test)
predictions = [round(value) for value in y_pred]
accuracy = accuracy_score(y_test, predictions)
print("Accuracy: %.2f%%" % (accuracy * 100.0))
```

```
# Fit model using each importance as a threshold
thresholds = sort(model.feature_importances_)
for thresh in thresholds:
  # select features using threshold
  selection = SelectFromModel(model, threshold=thresh, prefit=True)
  select_X_train = selection.transform(X_train)
  # train model
  selection_model = XGBClassifier()
  selection_model.fit(select_X_train, y_train)
  # eval model
  select_X_test = selection.transform(X_test)
  y_pred = selection_model.predict(select_X_test)
  predictions = [round(value) for value in y_pred]
  accuracy = accuracy_score(y_test, predictions)
  print("Thresh=%.3f, n=%d, Accuracy: %.2f%%" % (thresh, select_X_train.shape[1],
      accuracy*100.0))
```

Listing 9.8: Worked Example of Feature Selection Using XGBoost Feature Importance.

Running this example prints the following output:

```
Accuracy: 77.95%
Thresh=0.071, n=8, Accuracy: 77.95%
Thresh=0.073, n=7, Accuracy: 76.38%
Thresh=0.084, n=6, Accuracy: 77.56%
Thresh=0.090, n=5, Accuracy: 76.38%
Thresh=0.128, n=4, Accuracy: 76.38%
Thresh=0.160, n=3, Accuracy: 74.80%
Thresh=0.186, n=2, Accuracy: 71.65%
Thresh=0.208, n=1, Accuracy: 63.78%
```

Listing 9.9: Output of Model Performance With Feature Subsets by Importance Scores.

We can see that the performance of the model generally decreases with the number of selected features. On this problem there is a trade-off of features to test set accuracy and we could decide to take a less complex model (fewer attributes such as n=4) and accept a modest decrease in estimated accuracy from 77.95% down to 76.38%. This is likely to be a wash on such a small dataset, but may be a more useful strategy on a larger dataset and using cross-validation as the model evaluation scheme.

9.5 Summary

In this tutorial you discovered how to access features and use importance in a trained XGBoost gradient boosting model. Specifically, you learned:

- What feature importance is and generally how it is calculated in XGBoost.
- How to access and plot feature importance scores from an XGBoost model.
- How to use feature importance from an XGBoost model for feature selection.

In the next tutorial you will discover how to can monitor the performance of a model as it is being trained and configure training to stop early under specific criteria.

Chapter 10

Monitor Training Performance and Early Stopping

Overfitting is a problem with sophisticated nonlinear learning algorithms like gradient boosting. In this tutorial you will discover how you can use early stopping to limit overfitting with XGBoost in Python. After reading this tutorial, you will know:

- About early stopping as an approach to reducing overfitting of training data.
- How to monitor the performance of an XGBoost model during training and plot the learning curve.
- How to use early stopping to prematurely stop the training of an XGBoost model at an optimal epoch.

Let's get started.

10.1 Early Stopping to Avoid Overfitting

Early stopping is an approach to training complex machine learning models to avoid overfitting. It works by monitoring the performance of the model that is being trained on a separate test dataset and stopping the training procedure once the performance on the test dataset has not improved after a fixed number of training iterations.

It avoids overfitting by attempting to automatically select the inflection point where performance on the test dataset starts to decrease while performance on the training dataset continues to improve as the model starts to overfit. The performance measure may be the loss function that is being optimized to train the model (such as logarithmic loss), or an external metric of interest to the problem in general (such as classification accuracy).

10.2 Monitoring Training Performance With XGBoost

The XGBoost model can evaluate and report on the performance on a test set for the model during training. It supports this capability by specifying both an test dataset and an evaluation metric on the call to `model.fit()` when training the model and specifying verbose output. For

example, we can report on the binary classification error rate (`error`) on a standalone test set (`eval_set`) while training an XGBoost model as follows:

```
eval_set = [(X_test, y_test)]
model.fit(X_train, y_train, eval_metric="error", eval_set=eval_set, verbose=True)
```

Listing 10.1: Example of Evaluating a Validation Dataset During Training.

XGBoost supports a suite of evaluation metrics not limited to:

- `rmse` for root mean squared error.
- `mae` for mean absolute error.
- `logloss` for binary logarithmic loss and `mlogloss` for multiclass log loss (cross entropy).
- `error` for classification error.
- `auc` for area under ROC curve.

The full list is provided in the *Learning Task Parameters section of the XGBoost Parameters* webpage[1]. For example, we can demonstrate how to track the performance of the training of an XGBoost model on the Pima Indians onset of diabetes dataset (see Section 4.2). The full example is provided below:

```
# monitor training performance
from numpy import loadtxt
from xgboost import XGBClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
# load data
dataset = loadtxt('pima-indians-diabetes.csv', delimiter=",")
# split data into X and y
X = dataset[:,0:8]
Y = dataset[:,8]
# split data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.33, random_state=7)
# fit model no training data
model = XGBClassifier()
eval_set = [(X_test, y_test)]
model.fit(X_train, y_train, eval_metric="error", eval_set=eval_set, verbose=True)
# 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))
```

Listing 10.2: Worked Example of Evaluating a Validation Dataset During Training.

Running this example trains the model on 67% of the data and evaluates the model every training epoch on a 33% test dataset. The classification error is reported each iteration and finally the classification accuracy is reported at the end. The output is provided below, truncated for brevity. We can see that the classification error is reported each training iteration (after each boosted tree is added to the model).

[1] http://xgboost.readthedocs.io/en/latest/parameter.html

```
...
[89]  validation_0-error:0.204724
[90]  validation_0-error:0.208661
[91]  validation_0-error:0.208661
[92]  validation_0-error:0.208661
[93]  validation_0-error:0.208661
[94]  validation_0-error:0.208661
[95]  validation_0-error:0.212598
[96]  validation_0-error:0.204724
[97]  validation_0-error:0.212598
[98]  validation_0-error:0.216535
[99]  validation_0-error:0.220472
Accuracy: 77.95%
```

Listing 10.3: Sample Output of Worked Example of Evaluating a Validation Dataset During Training.

Reviewing all of the output, we can see that the model performance on the test set sits flat and even gets worse towards the end of training.

10.3 Evaluate XGBoost Models With Learning Curves

We can retrieve the performance of the model on the evaluation dataset and plot it to get insight into how learning unfolded while training. We provide an array of X and y pairs to the `eval_metric` argument when fitting our XGBoost model. In addition to a test set, we can also provide the training dataset. This will provide a report on how well the model is performing on both training and test sets during training. For example:

```
eval_set = [(X_train, y_train), (X_test, y_test)]
model.fit(X_train, y_train, eval_metric="error", eval_set=eval_set, verbose=True)
```

Listing 10.4: Example of Evaluating a Validation Dataset During Training.

In addition, the performance of the model on each evaluation set is stored and made available by the model after training by calling the `model.evals_result()` function. This returns a dictionary of evaluation datasets and scores, for example:

```
results = model.evals_result()
print(results)
```

Listing 10.5: Retrieve and Print Validation Results Collected During Training.

This will print results like the following (truncated for brevity):

```
{
  'validation_0': {'error': [0.259843, 0.26378, 0.26378, ...]},
  'validation_1': {'error': [0.22179, 0.202335, 0.196498, ...]}
}
```

Listing 10.6: Sample Output of Validation Performance Results.

Each of `validation_0` and `validation_1` correspond to the order that datasets were provided to the `eval_set` argument in the call to `fit()`. A specific array of results, such as for the first dataset and the error metric can be accessed as follows:

```
results['validation_0']['error']
```

Listing 10.7: Access Validation Results By Dataset and Metric.

Additionally, we can specify more evaluation metrics to evaluate and collect by providing an array of metrics to the `eval_metric` argument of the `fit()` function. We can then use these collected performance measures to create a line plot and gain further insight into how the model behaved on train and test datasets over training epochs. Below is the complete code example showing how the collected results can be visualized on a line plot.

```
# plot learning curve
from numpy import loadtxt
from xgboost import XGBClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from matplotlib import pyplot
# load data
dataset = loadtxt('pima-indians-diabetes.csv', delimiter=",")
# split data into X and y
X = dataset[:,0:8]
Y = dataset[:,8]
# split data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.33, random_state=7)
# fit model no training data
model = XGBClassifier()
eval_set = [(X_train, y_train), (X_test, y_test)]
model.fit(X_train, y_train, eval_metric=["error", "logloss"], eval_set=eval_set,
    verbose=True)
# 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))
# retrieve performance metrics
results = model.evals_result()
epochs = len(results['validation_0']['error'])
x_axis = range(0, epochs)
# plot log loss
fig, ax = pyplot.subplots()
ax.plot(x_axis, results['validation_0']['logloss'], label='Train')
ax.plot(x_axis, results['validation_1']['logloss'], label='Test')
ax.legend()
pyplot.ylabel('Log Loss')
pyplot.title('XGBoost Log Loss')
pyplot.show()
# plot classification error
fig, ax = pyplot.subplots()
ax.plot(x_axis, results['validation_0']['error'], label='Train')
ax.plot(x_axis, results['validation_1']['error'], label='Test')
ax.legend()
pyplot.ylabel('Classification Error')
pyplot.title('XGBoost Classification Error')
pyplot.show()
```

Listing 10.8: Worked Example of Plotting Training and Validation Performance.

Running this code reports the classification error on both the train and test datasets each epoch. We can turn this off by setting `verbose=False` (the default) in the call to the `fit()` function. Two plots are created. The first shows the logarithmic loss of the XGBoost model for each epoch on the training and test datasets.

Figure 10.1: XGBoost Learning Curves For Log Loss

The second plot shows the classification error of the XGBoost model for each epoch on the training and test datasets.

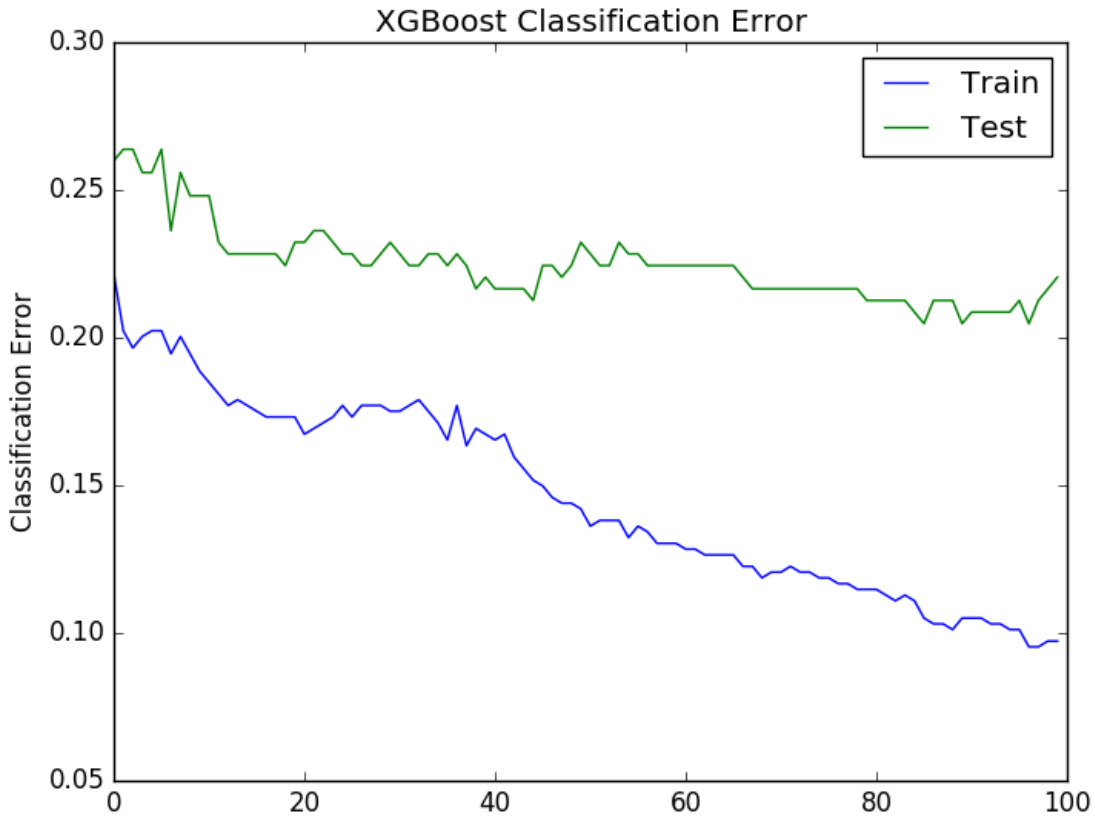

Figure 10.2: XGBoost Learning Curves For Classification Error

From reviewing the logloss plot, it looks like there is an opportunity to stop the learning early, perhaps somewhere around epoch 20 to epoch 40. We see a similar story for classification error, where error appears to go back up at around epoch 40.

10.4 Early Stopping With XGBoost

XGBoost supports early stopping after a fixed number of iterations. In addition to specifying a metric and test dataset for evaluation each epoch, you must specify a window of the number of epochs over which no improvement is observed. This is specified in the `early_stopping_rounds` parameter. For example, we can check for no improvement in logarithmic loss over the 10 epochs as follows:

```
eval_set = [(X_test, y_test)]
model.fit(X_train, y_train, early_stopping_rounds=10, eval_metric="logloss",
    eval_set=eval_set, verbose=True)
```

Listing 10.9: Example of Configuring Early Stopping.

If multiple evaluation datasets or multiple evaluation metrics are provided, then early stopping will use the last in the list. Below provides a full example for completeness with early stopping.

```
# early stopping
from numpy import loadtxt
from xgboost import XGBClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
# load data
dataset = loadtxt('pima-indians-diabetes.csv', delimiter=",")
# split data into X and y
X = dataset[:,0:8]
Y = dataset[:,8]
# 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, Y, test_size=test_size,
    random_state=seed)
# fit model no training data
model = XGBClassifier()
eval_set = [(X_test, y_test)]
model.fit(X_train, y_train, early_stopping_rounds=10, eval_metric="logloss",
    eval_set=eval_set, verbose=True)
# 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))
```

Listing 10.10: Worked Example of Early Stopping During Training.

Running the example provides the following output, truncated for brevity:

```
...
[35]  validation_0-logloss:0.487962
[36]  validation_0-logloss:0.488218
[37]  validation_0-logloss:0.489582
[38]  validation_0-logloss:0.489334
[39]  validation_0-logloss:0.490969
[40]  validation_0-logloss:0.48978
[41]  validation_0-logloss:0.490704
[42]  validation_0-logloss:0.492369
Stopping. Best iteration:
[32]  validation_0-logloss:0.487297
```

Listing 10.11: Sample Output of Worked Example of Early Stopping During Training.

We can see that the model stopped training at epoch 42 (close to what we expected by our manual judgment of learning curves) and that the model with the best loss was observed at epoch 32. It is generally a good idea to select the `early_stopping_rounds` as a reasonable function of the total number of training epochs (10% in this case) or attempt to correspond to the period of inflection points as might be observed on a plots of learning curves.

10.5 Summary

In this tutorial you discovered how to monitor the performance of an XGBoost model being trained and to support early stopping to limit overfitting. You learned:

- About the early stopping technique to stop model training before the model overfits the training data.
- How to monitor the performance of XGBoost models during training and to plot learning curves.
- How to configure early stopping when training XGBoost models.

In the next tutorial, you will discover how XGBoost can be configured to use all of the CPU cores of your system and how to best configure it to get the most from your hardware.

Chapter 11

Tune Multithreading Support for XGBoost

The XGBoost library for gradient boosting uses is designed for efficient multi-core parallel processing. This allows it to efficiently use all of the CPU cores in your system when training. In this tutorial you will discover the parallel processing capabilities of the XGBoost in Python. After reading this tutorial you will know:

- How to confirm that XGBoost multi-threading support is working on your system.
- How to evaluate the effect of increasing the number of threads on XGBoost.
- How to get the most out of multithreaded XGBoost when using cross-validation and grid search.

Let's get started.

11.1 Problem Description: Otto Dataset

In this tutorial we will use the Otto Group Product Classification Challenge[1] dataset. This dataset is available from Kaggle (you will need to sign-up to Kaggle to be able to download this dataset). You can download the training dataset `train.csv.zip` from the Data page[2] and place the unzipped `trian.csv` file into your working directory.

This dataset describes the 93 obfuscated details of more than 61,000 products grouped into 10 product categories (e.g. fashion, electronics, etc.). Input attributes are counts of different events of some kind. The goal is to make predictions for new products as an array of probabilities for each of the 10 categories and models are evaluated using multiclass logarithmic loss (also called cross entropy). This competition completed in May 2015 and this dataset is a good challenge for XGBoost because of the nontrivial number of examples and the difficulty of the problem and the fact that little data preparation is required (other than encoding the string class variables as integers).

[1] https://www.kaggle.com/c/otto-group-product-classification-challenge
[2] https://www.kaggle.com/c/otto-group-product-classification-challenge/data

11.2 Impact of the Number of Threads

XGBoost is implemented in C++ to explicitly make use of the OpenMP API[3] for parallel processing. The parallelism in gradient boosting can be implemented in the construction of individual trees, rather than in creating trees in parallel like random forest. This is because in boosting, trees are added to the model sequentially. The speed of XGBoost is both in adding parallelism in the construction of individual trees, and in the efficient preparation of the input data to aid in the speed up in the construction of trees.

Depending on your platform, you may need to compile XGBoost specifically to support multithreading. See the XGBoost installation instructions for more details[4]. The `XGBClassifier` and `XGBRegressor` wrapper classes for XGBoost for use in scikit-learn provide the `nthread` parameter to specify the number of threads that XGBoost can use during training.

By default this parameter is set to -1 to make use of all of the cores in your system.

```
model = XGBClassifier(nthread=-1)
```

Listing 11.1: Example of Setting the Number of Threads used by XGBoost.

Generally, you should get multithreading support for your XGBoost installation without any extra work. Depending on your Python environment (e.g. Python 3) you may need to explicitly enable multithreading support for XGBoost. The XGBoost library provides an example if you need help[5]. You can confirm that XGBoost multi-threading support is working by building a number of different XGBoost models, specifying the number of threads and timing how long it takes to build each model. The trend will both show you that multi-threading support is enabled and give you an indication of the effect it has when building models. For example, if your system has 4 cores, you can train 8 different models and time how long in seconds it takes to create each, then compare the times.

```
# evaluate the effect of the number of threads
results = []
num_threads = [1, 2, 3, 4]
for n in num_threads:
  start = time.time()
  model = XGBClassifier(nthread=n)
  model.fit(X_train, y_train)
  elapsed = time.time() - start
  print(n, elapsed)
  results.append(elapsed)
```

Listing 11.2: Example of Evaluating the Effect of the Number of Threads on Model Training.

We can use this approach on the Otto dataset. The full example is provided below for completeness. You can change the `num_threads` array to meet the number of cores on your system.

```
# Otto, tune number of threads
from pandas import read_csv
from xgboost import XGBClassifier
from sklearn.preprocessing import LabelEncoder
from time import time
```

[3] https://en.wikipedia.org/wiki/OpenMP

[4] https://github.com/dmlc/xgboost/blob/master/doc/build.md

[5] https://github.com/dmlc/xgboost/blob/master/demo/guide-python/sklearn_parallel.py

```
from matplotlib import pyplot
# load data
data = read_csv('train.csv')
dataset = data.values
# split data into X and y
X = dataset[:,0:94]
y = dataset[:,94]
# encode string class values as integers
label_encoded_y = LabelEncoder().fit_transform(y)
# evaluate the effect of the number of threads
results = []
num_threads = [1, 2, 3, 4]
for n in num_threads:
  start = time()
  model = XGBClassifier(nthread=n)
  model.fit(X, label_encoded_y)
  elapsed = time() - start
  print(n, elapsed)
  results.append(elapsed)
# plot results
pyplot.plot(num_threads, results)
pyplot.ylabel('Speed (seconds)')
pyplot.xlabel('Number of Threads')
pyplot.title('XGBoost Training Speed vs Number of Threads')
pyplot.show()
```

Listing 11.3: Worked Example of Timing Training With Different Numbers of Threads.

Running this example summarizes the execution time in seconds for each configuration, for example:

```
(1, 115.51652717590332)
(2, 62.7727689743042)
(3, 46.042901039123535)
(4, 40.55334496498108)
```

Listing 11.4: Sample Output of Worked Example of Timing Training With Different Numbers of Threads.

A plot of these timings is provided below.

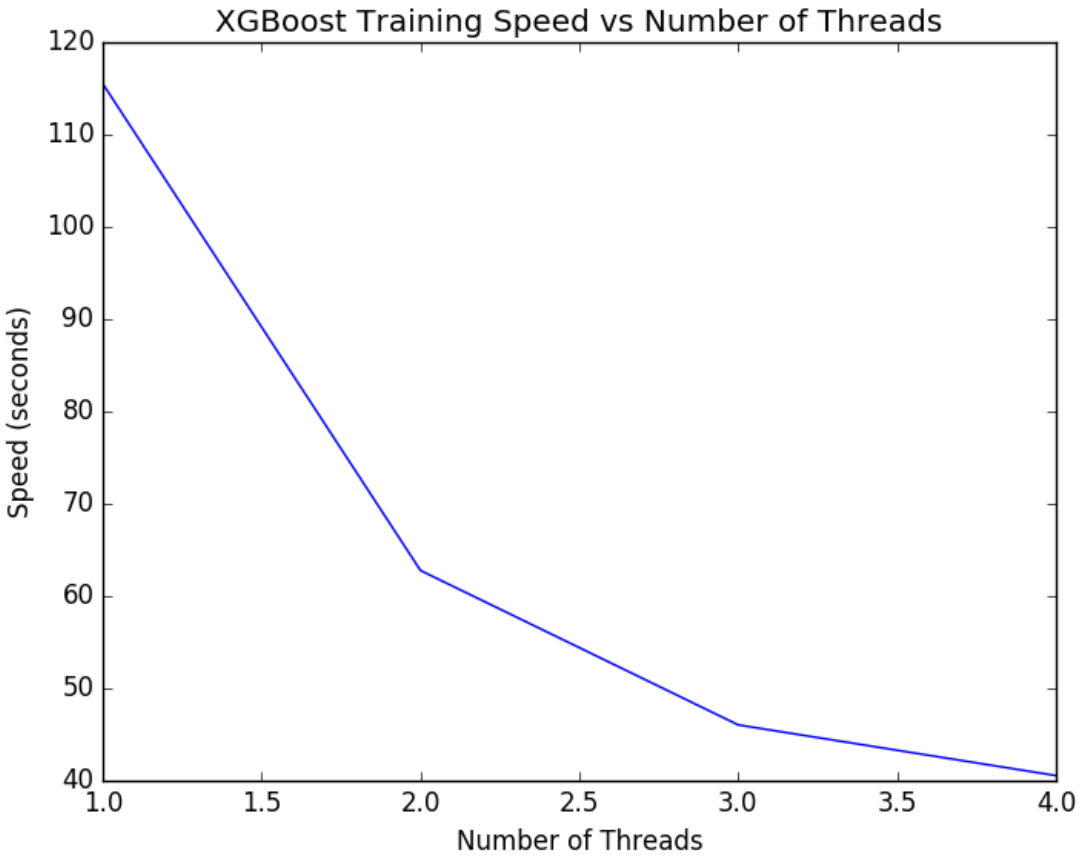

Figure 11.1: XGBoost Tune Number of Threads for Single Model

We can see a nice trend in the decrease in execution time as the number of threads is increased. If you do not see an improvement in running time for each new thread, you may want to investigate how to enable multithreading support in XGBoost as part of your install or at runtime. We can run the same code on a machine with a lot more cores. The large Amazon Web Services EC2 instance is reported to have 32 cores (see Chapter 12). We can adapt the above code to time how long it takes to train the model with 1 to 32 cores. The results are plotted below.

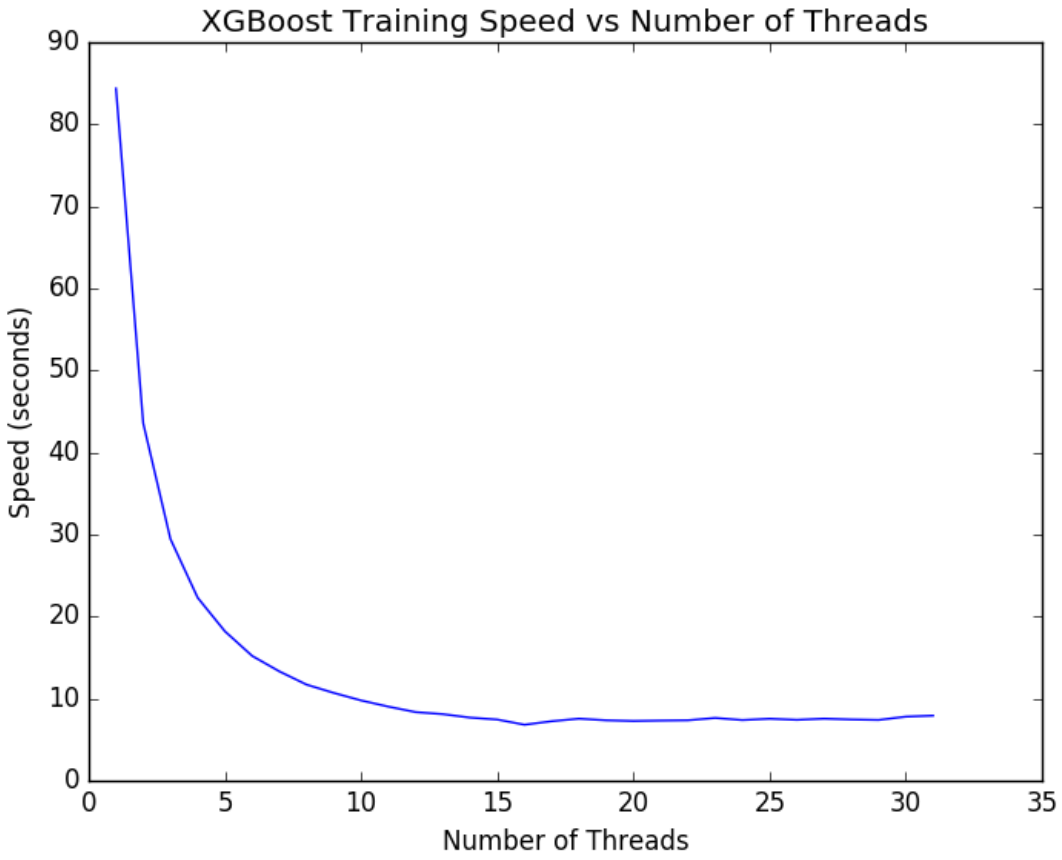

Figure 11.2: XGBoost Time to Train Model on 1 to 32 Cores

It is interesting to note that we do not see much improvement beyond 16 threads (at about 7 seconds). I expect the reason for this is that the Amazon instance only provides 16 cores in hardware and the additional 16 cores are available by hyperthreading. The results suggest that if you have a machine with hyperthreading, you may want to set `nthread` to equal the number of physical CPU cores in your machine. The low-level optimized implementation of XGBoost with OpenMP squeezes every last cycle out of a large machine like this.

11.3 Parallelism When Cross Validating XGBoost Models

The k-fold cross-validation support in scikit-learn also supports multithreading. For example, the `n_jobs` argument on the `cross_val_score()` function used to evaluate a model on a dataset using k-fold cross-validation allows you to specify the number of parallel jobs to run. By default, this is set to 1, but can be set to -1 to use all of the CPU cores on your system, which is good practice. For example:

```
results = cross_val_score(model, X, label_encoded_y, cv=kfold, scoring='neg_log_loss',
    n_jobs=-1, verbose=1)
```

Listing 11.5: Example of k-fold Cross-Validation Using all CPU Cores.

This raises the question as to how cross-validation should be configured:

- Disable multi-threading support in XGBoost and allow cross-validation to run on all cores.
- Disable multi-threading support in cross-validation and allow XGBoost to run on all cores.
- Enable multi-threading support for both XGBoost and cross-validation.

We can get an answer to this question by simply timing how long it takes to evaluate the model in each circumstance. In the example below we use 10-fold cross-validation to evaluate the default XGBoost model on the Otto training dataset. Each of the above scenarios is evaluated and the time taken to evaluate the model is reported. The full code example is provided below.

```
# Otto, parallel cross validation
from pandas import read_csv
from xgboost import XGBClassifier
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import cross_val_score
from sklearn.preprocessing import LabelEncoder
import time
# load data
data = read_csv('train.csv')
dataset = data.values
# split data into X and y
X = dataset[:,0:94]
y = dataset[:,94]
# encode string class values as integers
label_encoded_y = LabelEncoder().fit_transform(y)
# prepare cross validation
kfold = StratifiedKFold(n_splits=10, shuffle=True, random_state=7)
# Single Thread XGBoost, Parallel Thread CV
start = time.time()
model = XGBClassifier(nthread=1)
results = cross_val_score(model, X, label_encoded_y, cv=kfold, scoring='neg_log_loss',
    n_jobs=-1)
elapsed = time.time() - start
print("Single Thread XGBoost, Parallel Thread CV: %f" % (elapsed))
# Parallel Thread XGBoost, Single Thread CV
start = time.time()
model = XGBClassifier(nthread=-1)
results = cross_val_score(model, X, label_encoded_y, cv=kfold, scoring='neg_log_loss',
    n_jobs=1)
elapsed = time.time() - start
print("Parallel Thread XGBoost, Single Thread CV: %f" % (elapsed))
# Parallel Thread XGBoost and CV
start = time.time()
model = XGBClassifier(nthread=-1)
results = cross_val_score(model, X, label_encoded_y, cv=kfold, scoring='neg_log_loss',
    n_jobs=-1)
elapsed = time.time() - start
print("Parallel Thread XGBoost and CV: %f" % (elapsed))
```

Listing 11.6: Worked Example of Timing Training With Multithreading of CV and XGBoost.

Running the example prints the following results:

```
Single Thread XGBoost, Parallel Thread CV: 359.854589
Parallel Thread XGBoost, Single Thread CV: 330.498101
Parallel Thread XGBoost and CV: 313.382301
```

Listing 11.7: Sample Output of Worked Example of Timing Training With Multithreading of CV and XGBoost.

We can see that there is a benefit from parallelizing XGBoost over the cross-validation folds. This makes sense as 10 sequential fast tasks is better than (10 divided by `nthread`) slow tasks. Interestingly we can see that the best result is achieved by enabling both multi-threading within XGBoost and in cross-validation. This is surprising because it means that `nthread` number of parallel XGBoost models are competing for the same `nthread` in the construction of their models. Nevertheless, this achieves the fastest results and is the suggested usage of XGBoost for cross-validation. Because grid search uses the same underlying approach to parallelism, we expect the same finding to hold for optimizing the hyperparameters for XGBoost.

11.4 Summary

In this tutorial you discovered how to get the most from your hardware when using the XGBoost model. You learned:

- How to check that the multi-threading support in XGBoost is enabled on your system.
- How increasing the number of threads affects the performance of training XGBoost models.
- How to best configure XGBoost and Cross-Validation in Python for minimum running time.

In the next tutorial you will discover how you can scale up your model to use many more cores using Amazon cloud infrastructure.

Chapter 12

Train XGBoost Models in the Cloud with Amazon Web Services

The XGBoost library provides an implementation of gradient boosting designed for speed and performance. It is implemented to make best use of your computing resources, including all CPU cores and memory. In this tutorial you will discover how you can setup a server on Amazon's cloud service to quickly and cheaply create very large models. After reading this tutorial you will know:

- How to setup and configure an Amazon EC2 server instance for use with XGBoost.
- How to confirm the parallel capabilities of XGBoost are working on your server.
- How to transfer data and code to your server and train a very large model.

Let's get started.

12.1 Tutorial Overview

The process is quite simple. Below is an overview of the steps we are going to complete in this tutorial.

1. Setup Your AWS Account (if needed).
2. Launch Your AWS Instance.
3. Login and Run Your Code.
4. Train an XGBoost Model.
5. Close Your AWS Instance.

Note, it costs money to use a virtual server instance on Amazon. The cost is very low for ad hoc model development (e.g. less than one US dollar per hour), which is why this is so attractive, but it is not free. The server instance runs Linux. It is desirable although not required that you know how to navigate Linux or a Unix-like environment. We're just running our Python scripts, so no advanced skills are needed.

12.2 Setup Your AWS Account (if needed)

You need an account on Amazon Web Services.

- 1. You can create an account using the Amazon Web Services portal by clicking *Sign in to the Console*. From there you can sign in using an existing Amazon account or create a new account.

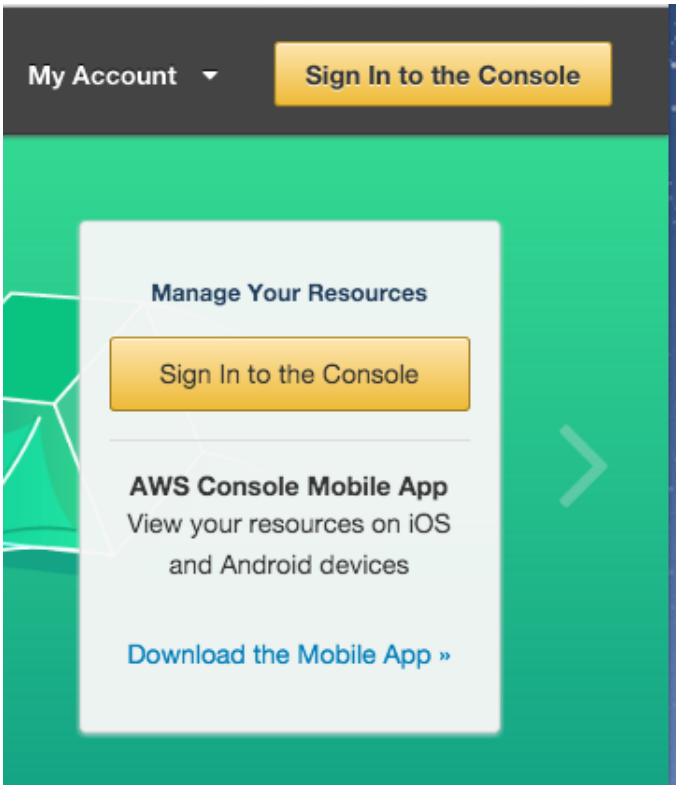

Figure 12.1: AWS Sign-in Button

- 2. If creating an account, you will need to provide your details as well as a valid credit card that Amazon can charge. The process is a lot quicker if you are already an Amazon customer and have your credit card on file.

Note: If you have created a new account, you may have to request to Amazon support in order to be approved to use larger (non-free) server instance in the rest of this tutorial.

12.3 Launch Your Server Instance

Now that you have an AWS account, you want to launch an EC2 virtual server instance on which you can run XGBoost. Launching an instance is as easy as selecting the image to load and starting the virtual server. se an existing Fedora Linux image and install Python and XGBoost manually.

- 1. Login to your AWS console if you have not already[1].

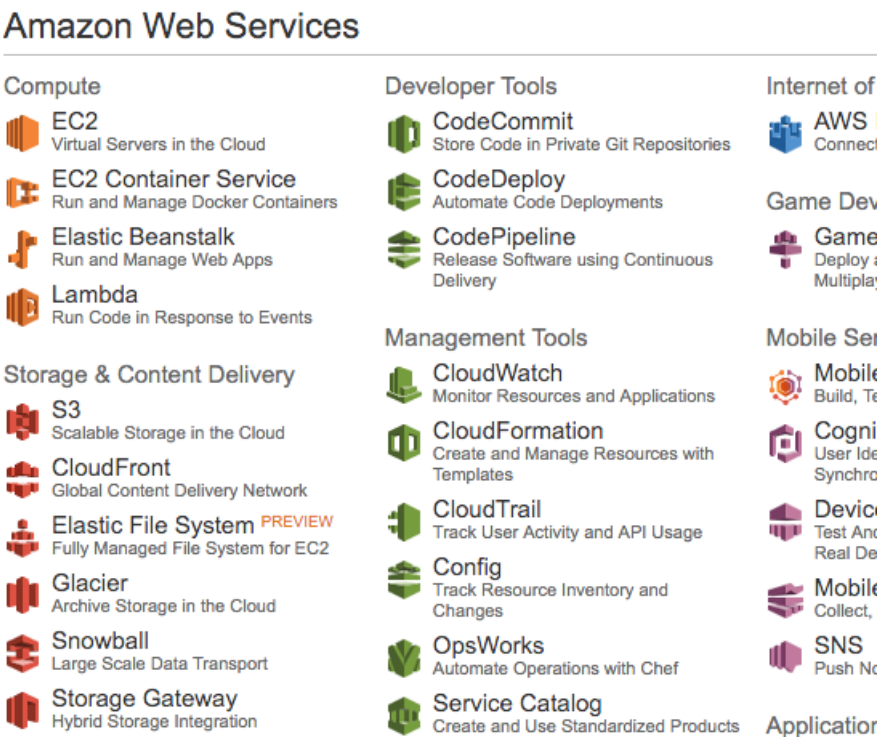

Figure 12.2: AWS Console

- 2. Click on EC2 for launching a new virtual server.
- 3. Select *N. California* from the drop-down in the top right hand corner. This is important otherwise you may not be able to find the image (called an AMI) that we plan to use.

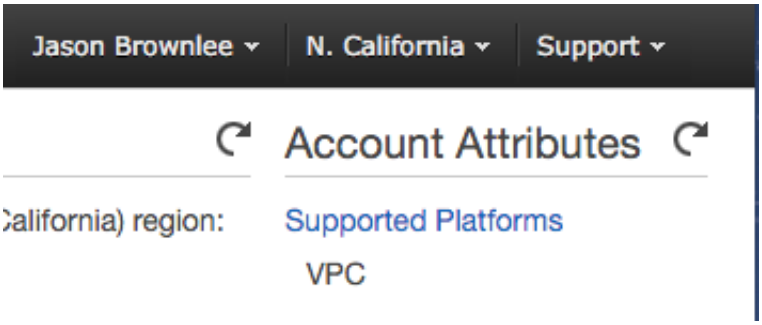

Figure 12.3: Select N California

- 4. Click the *Launch Instance* button.
- 5. Click *Community AMIs*. An AMI is an Amazon Machine Image. It is a frozen instance of a server that you can select and instantiate on a new virtual server.

[1] https://console.aws.amazon.com/console/home

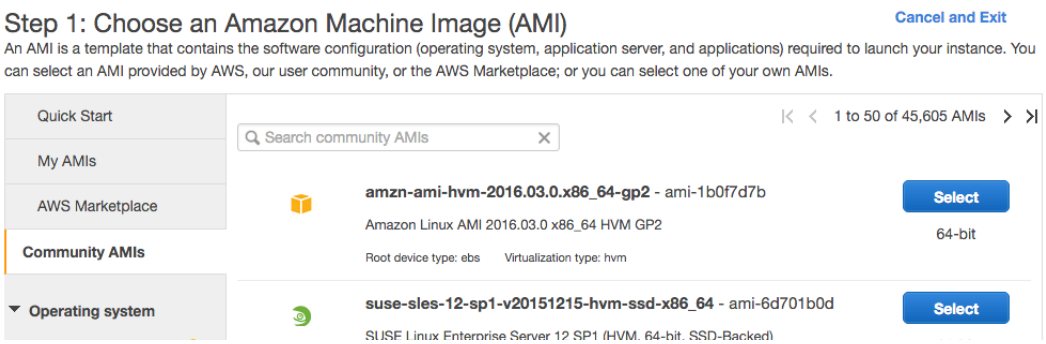

Figure 12.4: Community AMIs

- 6. Enter AMI: `ami-02d09662` in the *Search community AMIs* search box and press enter. You should be presented with a single result.

This is an image for a base installation of Fedora Linux version 24[2]. A very easy to use Linux distribution.

Figure 12.5: Select the c3.8xlarge Instance Type

- 9. Click *Review and Launch* to finalize the configuration of your server instance.

You will see a warning like *Your instance configuration is not eligible for the free usage tier*. This is just indicating that you will be charged for your time on this server. We know this, ignore this warning.

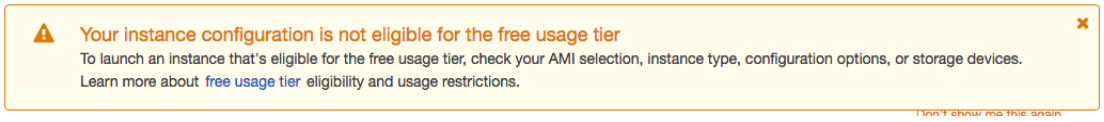

Figure 12.6: Warning Message That You Will be Charged.

- 10. Click the *Launch* button.
- 11. Select your SSH key pair.
  - If you have a key pair because you have used EC2 before, select *Choose an existing key pair* and choose your key pair from the list. Then check *I acknowledge....*

[2] https://getfedora.org/

  - If you do not have a key pair, select the option *Create a new key pair* and enter a *Key pair name* such as `xgboost-keypair`. Click the *Download Key Pair* button.

- 12. Open a Terminal and change directory to where you downloaded your key pair.
- 13. If you have not already done so, restrict the access permissions on your key pair file. This is required as part of the SSH access to your server. For example, on your console you can type:

```
cd Downloads
chmod 600 xgboost-keypair.pem
```

Listing 12.1: Set Permissions on Your SSH Keys.

- 14. Click *Launch Instances*.

Note: If this is your first time using AWS, Amazon may have to validate your request and this could take up to 2 hours (often just a few minutes).

- 15. Click *View Instances* to review the status of your instance.

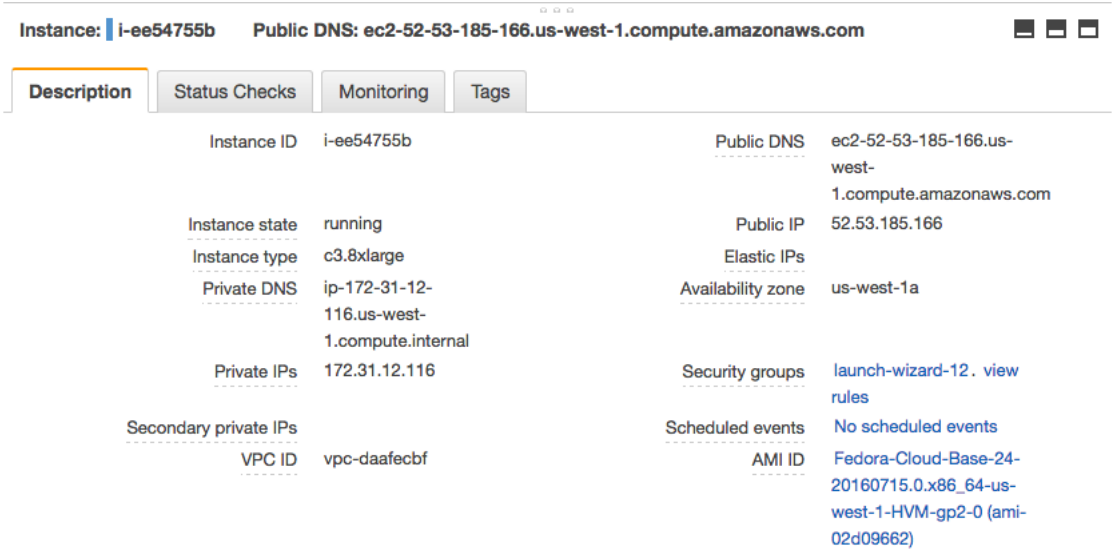

Figure 12.7: Review Your Running Instance and Note its IP Address.

Your server is now running and ready for you to log in.

12.4 Login and Configure

Now that you have launched your server instance, it is time to login and configure it for use. You will need to configure your server each time you launch it. Therefore, it is a good idea to batch all work so you can make good use of your configured server. Configuring the server will not take long, perhaps 10 minutes total.

- 1. Click *View Instances* in your Amazon EC2 console if you have not already.
- 2. Copy *Public IP* (down the bottom of the screen in Description) to clipboard.

In this example my IP address is `52.53.185.166`.
Do not use this IP address, your IP address will be different.

- 3. Open a Terminal and change directory to where you downloaded your key pair. Login to your server using SSH, for example you can type:

```
ssh -i xgboost-keypair.pem fedora@52.53.185.166
```

Listing 12.2: Log Into Your Server Using SSH from the Command Line.

- 4. You may be prompted with a warning the first time you log into your server instance. You can ignore this warning, just type *yes* and press enter.

You are now logged into your server. Double check the number of CPU cores on your instance, by typing:

```
cat /proc/cpuinfo | grep processor | wc -l
```

Listing 12.3: Check the Number of Available CPU Cores.

You should see:

```
32
```

Listing 12.4: Sample Output from Checking the Number of Available CPU Cores.

12.4.1 Install Supporting Packages

The first step is to install all of the packages to support XGBoost. This includes GCC, Python and the SciPy stack. We will use the Fedora package manager dnf (the new yum). This is a single line:

```
sudo dnf install gcc gcc-c++ make git unzip python python2-numpy python2-scipy
    python2-scikit-learn python2-pandas python2-matplotlib
```

Listing 12.5: Install Packages Required for XGBoost.

Type `y` and press Enter when prompted to confirm the packages to install. This will take a few minutes to download and install all of the required packages. Once completed, we can confirm the environment was installed successfully.

Check GCC

Type:

```
gcc --version
```

Listing 12.6: Print the GCC Version.

You should see:

```
gcc (GCC) 6.1.1 20160621 (Red Hat 6.1.1-3)
Copyright (C) 2016 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Listing 12.7: Sample Output From Printing the GCC Version.

Check Python

Type:

```
python --version
```

Listing 12.8: Print the Python Version.

You should see:

```
Python 2.7.12
```

Listing 12.9: Sample Output From Printing the Python Version.

Check SciPy

Type:

```
python -c "import scipy;print(scipy.__version__)"
python -c "import numpy;print(numpy.__version__)"
python -c "import pandas;print(pandas.__version__)"
python -c "import sklearn;print(sklearn.__version__)"
```

Listing 12.10: Print the SciPy Stack Versions.

You should see:

```
0.16.1
1.11.0
0.18.0
0.17.1
```

Listing 12.11: Sample Output From Printing the SciPy Stack Versions.

Note: If any of these checks failed, stop and correct any errors. You must have a complete working environment before moving on.

We are now ready to install XGBoost.

12.4.2 Build and Install XGBoost

The installation instructions for XGBoost are complete and we can follow them directly[3]. First we need to download the project on the server.

```
git clone --recursive https://github.com/dmlc/xgboost
cd xgboost
```

Listing 12.12: Download the XGBoost Project.

[3] https://xgboost.readthedocs.io/en/latest/build.html

Next we need to compile it. The `-j` argument can be used to specify the number of cores to expect. We can set this to 32 for the 32 cores on our AWS instance. If you chose different AWS hardware, you can set this appropriately.

```
make -j32
```

Listing 12.13: Compile the XGBoost Project.

The XGBoost project should build successfully (e.g. no errors). We are now ready to install the Python version of the library.

```
cd python-package
sudo python setup.py install
```

Listing 12.14: Install the Python XGBoost Wrappers.

That is it. We can confirm the installation was successful by typing:

```
python -c "import xgboost;print(xgboost.__version__)"
```

Listing 12.15: Print the XGBoost Version.

This should print something like:

```
0.4
```

Listing 12.16: Sample Output From Printing the XGBoost Version.

12.5 Train an XGBoost Model

Let's test out your large AWS instance by running XGBoost with a lot of cores. In this tutorial we will use the Otto Group Product Classification Challenge dataset (see Section 11.1). Create a new directory called `work/` on your workstation. You can download the training dataset `train.csv.zip` from the Data page[4] and place it in your `work/` directory on your workstation.

We will evaluate the time taken to train an XGBoost on this dataset using different numbers of cores. We will try 1 core, half the cores 16 and all of the 32 cores. We can specify the number of cores used by the XGBoost algorithm by setting the `nthread` parameter in the XGBClassifier class (the scikit-learn wrapper for XGBoost). The complete example is listed below. Save it in a file with the name `work/script.py`. Now, we can copy your `work/` directory with the data and script to your AWS server. From your workstation in the current directory where the `work/` directory is located, type:

```
scp -r -i xgboost-keypair.pem work fedora@52.53.185.166:/home/fedora/
```

Listing 12.17: Copy the work Directory to your Server.

Of course, you will need to use your key file and the IP address of your server. This will create a new `work/` directory in your home directory on your server. Log back onto your server instance (if needed):

```
ssh -i xgboost-keypair.pem fedora@52.53.185.166
```

Listing 12.18: Log Into Your Server With SSH.

[4] https://www.kaggle.com/c/otto-group-product-classification-challenge/data

Change directory to your work directory and unzip the training data.

```
cd work
unzip ./train.csv.data
```

Listing 12.19: Unzip The Otto Dataset.

Now we can run the script and train our XGBoost models and calculate how long it takes using different numbers of cores:

```
python script.py
```

Listing 12.20: Run the Python Script on the Server.

You should see output like:

```
(1, 84.26896095275879)
(16, 6.597043037414551)
(32, 7.6703619956970215)
```

Listing 12.21: Output of Evaluating Training Time With Different Numbers of Threads.

You can see little difference between 16 and 32 cores. I believe the reason for this is that AWS is giving access to 16 physical cores with hyperthreading, offering an additional virtual cores. Nevertheless, building a large XGBoost model in 7 seconds is great. You can use this as a template for your copying your own data and scripts to your AWS instance. A good tip is to run your scripts as a background process and forward any output to a file. This is just in case your connection to the server is interrupted or you want to close it down and let the server run your code all night. You can run your code as a background process and redirect output to a file by typing:

```
nohup python script.py >script.py.out 2>&1 &
```

Listing 12.22: Example of Running a Python Script as a Background Process.

Now that we are done, we can shut down the AWS instance.

12.6 Close Your AWS Instance

When you are finished with your work you must close your instance. Remember you are charged by the amount of time that you use the instance. It is cheap, but you do not want to leave an instance on if you are not using it.

- 1. Log out of your instance at the terminal, for example you can type: `exit`
- 2. Log in to your AWS account with your web browser.
- 3. Click EC2.
- 4. Click *Instances* from the left-hand side menu.
- 5. Select your running instance from the list (it may already be selected if you only have one running instance).

- 6. Click the *Actions* button and select *Instance State* and choose *Terminate*. Confirm that you want to terminate your running instance.

It may take a number of seconds for the instance to close and to be removed from your list of instances. That's it.

12.7 Summary

In this tutorial you discovered how to train large XGBoost models on Amazon cloud infrastructure. Specifically, you learned:

- How to start and configure a Linux server instance on Amazon EC2 for XGBoost.
- How to install all of the required software needed to run the XGBoost library in Python.
- How to transfer data and code to your server and train a large model making use of all of the cores on the server.

This concludes Part III of this book. Next, in Part IV you will discover how to configure gradient boosting models and tune the hyperparameters, starting with an introduction to common configuration heuristics.

Part IV

XGBoost Tuning

Chapter 13

How to Configure the Gradient Boosting Algorithm

Gradient boosting is one of the most powerful techniques for applied machine learning and as such is quickly becoming one of the most popular. But how do you configure gradient boosting on your problem? In this tutorial you will discover how you can configure gradient boosting on your machine learning problem by looking at configurations reported in books, papers and as a result of competitions. After reading this tutorial, you will know:

- How to configure gradient boosting according to the original sources.
- Ideas for configuring the algorithm from defaults and suggestions in standard implementations.
- Rules of thumb for configuring gradient boosting and XGBoost from a top Kaggle competitors.

Let's get started.

13.1 Configuration Advice from Primary Sources

In the 1999 paper *Greedy Function Approximation: A Gradient Boosting Machine*, Jerome Friedman comments on the trade-off between the number of trees (M) and the learning rate (v):

> The v-M trade-off is clearly evident; smaller values of v give rise to larger optimal M-values. They also provide higher accuracy, with a diminishing return for $v < 0.125$. The misclassification error rate is very flat for $M > 200$, so that optimal M-values for it are unstable. [...] the qualitative nature of these results is fairly universal.

He suggests to first set a large value for the number of trees, then tune the shrinkage parameter to achieve the best results. Studies in the paper preferred a shrinkage value of 0.1, a number of trees in the range 100 to 500 and the number of terminal nodes in a tree between 2 and 8. In the 1999 paper *Stochastic Gradient Boosting*, Friedman reiterated the preference for the shrinkage parameter:

> The "shrinkage" parameter $0 < v < 1$ controls the learning rate of the procedure. Empirically [...], it was found that small values ($v \leq 0.1$) lead to much better generalization error.

In the paper, Friedman introduces and empirically investigates stochastic gradient boosting (row-based sub-sampling). He finds that almost all subsampling percentages are better than so-called deterministic boosting and perhaps 30%-to-50% is a good value to choose on some problems and 50%-to-80% on others.

> ... the best value of the sampling fraction [...] is approximately 40% ($f = 0.4$) [...] However, sampling only 30% or even 20% of the data at each iteration gives considerable improvement over no sampling at all, with a corresponding computational speed-up by factors of 3 and 5 respectively.

He also studied the effect of the number of terminal nodes in trees finding values like 3 and 6 better than larger values like 11, 21 and 41.

> In both cases the optimal tree size as averaged over 100 targets is $L = 6$. Increasing the capacity of the base learner by using larger trees degrades performance through "over-fitting".

In his talk titled *Gradient Boosting Machine Learning* at H2O[1], Trevor Hastie made the comment that in general gradient boosting performs better than random forest, which in turn performs better than individual decision trees.

> Gradient Boosting > Random Forest > Bagging > Single Trees

Chapter 10 titled *Boosting and Additive Trees* of the book *The Elements of Statistical Learning: Data Mining, Inference, and Prediction* is dedicated to boosting. In it they provide both heuristics for configuring gradient boosting as well as some empirical studies. They comment that a good value the number of nodes in the tree (J) is about 6, with generally good values in the range of 4-to-8.

> Although in many applications $J = 2$ will be insufficient, it is unlikely that $J > 10$ will be required. Experience so far indicates that $4 \leq J \leq 8$ works well in the context of boosting, with results being fairly insensitive to particular choices in this range.

They suggest monitoring the performance on a validation dataset in order to calibrate the number of trees and to use an early stopping procedure once performance on the validation dataset begins to degrade. As in Friedman's first gradient boosting paper, they comment on the trade-off between the number of trees (M) and the learning rate (v) and recommend a small value for the learning rate < 0.1.

> Smaller values of v lead to larger values of M for the same training risk, so that there is a tradeoff between them. [...] In fact, the best strategy appears to be to set v to be very small ($v < 0.1$) and then choose M by early stopping.

Also, as in Friedman's stochastic gradient boosting paper, they recommend a subsampling percentage (n) without replacement with a value of about 50%.

> A typical value for n can be $\frac{1}{2}$, although for large N, n can be substantially smaller than $\frac{1}{2}$.

[1] https://www.youtube.com/watch?v=wPqtzj5VZus

13.2 Configuration Advice From R

The gradient boosting algorithm is implemented in R as the gbm package. Reviewing the package documentation[2], the `gbm()` function specifies sensible defaults:

- `n.trees` = 100 (number of trees).
- `interaction.depth` = 1 (number of leaves).
- `n.minobsinnode` = 10 (minimum number of samples in tree terminal nodes).
- `shrinkage` = 0.001 (learning rate).

It is interesting to note that a smaller shrinkage factor is used and that stumps are the default. The small shrinkage is explained by Ridgeway next. In the vignette for using the gbm package in R titled *Generalized Boosted Models: A guide to the gbm package*[3], Greg Ridgeway provides some usage heuristics. He suggest firs setting the learning rate (`lambda`) to as small as possible then tuning the number of trees (iterations or T) using cross-validation.

> In practice I set lambda to be as small as possible and then select T by cross-validation. Performance is best when lambda is as small as possible performance with decreasing marginal utility for smaller and smaller lambda.

He comments on his rationale for setting the default shrinkage to the small value of 0.001 rather than 0.1.

> It is important to know that smaller values of shrinkage (almost) always give improved predictive performance. That is, setting shrinkage=0.001 will almost certainly result in a model with better out-of-sample predictive performance than setting shrinkage=0.01. [...] The model with shrinkage=0.001 will likely require ten times as many iterations as the model with shrinkage=0.01

Ridgeway also uses quite large numbers of trees (called iterations here), thousands rather than hundreds.

> I usually aim for 3,000 to 10,000 iterations with shrinkage rates between 0.01 and 0.001.

13.3 Configuration Advice From scikit-learn

The Python library provides an implementation of gradient boosting for classification called the `GradientBoostingClassifier` class and regression called the `GradientBoostingRegressor` class. It is useful to review the default configuration for the algorithm in this library. There are many parameters, but below are a few key defaults.

- `learning_rate` = 0.1 (shrinkage).

[2] https://cran.r-project.org/web/packages/gbm/gbm.pdf
[3] http://www.saedsayad.com/docs/gbm2.pdf

- `n_estimators` = 100 (number of trees).
- `max_depth` = 3.
- `min_samples_split` = 2.
- `min_samples_leaf` = 1.
- `subsample` = 1.0.

It is interesting to note that the default shrinkage does match Friedman and that the tree depth is not set to stumps like the R package. A tree depth of 3 (if the created tree was symmetrical) will have 8 leaf nodes, matching the upper bound of the preferred number of terminal nodes in Friedman's studies (alternately `max_leaf_nodes` can be set). In the scikit-learn user guide under the section titled *Gradient Tree Boosting*[4] the authors comment that setting the maximum leaf nodes has a similar effect to setting the max depth to one minus the maximum leaf nodes, but results in worse performance.

> We found that `max_leaf_nodes`=k gives comparable results to `max_depth`=$k-1$ but is significantly faster to train at the expense of a slightly higher training error.

In a small study demonstrating regularization methods for gradient boosting titled *Gradient Boosting regularization*[5], the results show the benefit of using both shrinkage and sub-sampling.

13.4 Configuration Advice From XGBoost

The XGBoost library is dedicated to the gradient boosting algorithm. It too specifies default parameters that are interesting to note, firstly the XGBoost Parameters page[6]:

- `eta` = 0.3 (a.k.a `learning_rate`).
- `max_depth` = 6.
- `subsample` = 1.

This shows a higher learning rate and a larger max depth than we see in most studies and other libraries. Similarly, we can summarize the default parameters for XGBoost in the Python API Reference[7].

- `max_depth` = 3.
- `learning_rate` = 0.1.
- `n_estimators` = 100.
- `subsample` = 1.

These defaults are generally more in-line with scikit-learn defaults and recommendations from the papers. In a talk to TechEd Europe titled *xgboost: An R package for Fast and Accurate Gradient Boosting*[8], when asked how to configure XGBoost, Tong He suggested the three most

[4] http://scikit-learn.org/stable/modules/ensemble.html#gradient-tree-boosting
[5] http://goo.gl/aHSSCA
[6] https://github.com/dmlc/xgboost/blob/master/doc/parameter.md
[7] https://xgboost.readthedocs.io/en/latest/python/python_api.html
[8] https://www.youtube.com/watch?v=0IhraqUVJ_E

important parameters to tune are: the number of trees, tree depth and learning rate. He also provide a terse configuration strategy for new problems:

1. Run the default configuration (and presumably review learning curves?).
2. If the system is overlearning, slow the learning down (using shrinkage?).
3. If the system is underlearning, speed the learning up to be more aggressive (using shrinkage?).

In Owen Zhang's talk to the NYC Data Science Academy in 2015 titled *Winning Data Science Competitions*[9], he provides some general tips for configuring gradient boost with XGBoost. Owen is a heavy user of gradient boosting.

> My confession: I (over)use GBM. When in doubt, use GBM.

He provides some tips for configuring gradient boosting:

- Target 500-to-1000 trees and then tune the learning rate (`n_estimators`).
- Set the number of samples in the leaf nodes to enough observations needed to make a good mean estimate (`min_child_weight`).
- Configure the interaction depth to about 10 or more (`max_depth`).

In an updated slide deck for the same talk[10], he provides a table of common parameters he uses for XGBoost, summarized as follows:

- **Number of Trees** (`n_estimators`) set to a fixed value between 100 and 1000, depending on the dataset size.
- **Learning Rate** (`learning_rate`) simplified to the ratio: $\frac{[2\ to\ 10]}{trees}$, depending on the number of trees.
- **Row Sampling** (`subsample`) grid searched values in the range [0.5, 0.75, 1.0].
- **Column Sampling** (`colsample_bytree` and maybe `colsample_bylevel`) grid searched values in the range [0.4, 0.6, 0.8, 1.0].
- **Min Leaf Weight** (`min_child_weight`) simplified to the ratio $\frac{3}{rare\_events}$, where `rare_events` is the percentage of rare event observations in the dataset.
- **Tree Size** (`max_depth`) grid searched values in the rage [4, 6, 8, 10].
- **Min Split Gain** (`gamma`) fixed with a value of zero.

[9] https://www.youtube.com/watch?v=LgLcfZjNF44
[10] http://goo.gl/OqIRIc

What is interesting to note is that this world class practitioner does not fiddle with `gamma` or the terms for the regularization penalty (`reg_alpha` and `reg_lambda`). In a similar talk by Owen at ODSC Boston 2015 titled *Open Source Tools and Data Science Competitions*[11], he again summarized common parameters he uses. We can see some minor differences that may be relevant:

- **Number of Trees and Learning Rate**: Fix the number of trees at around 100 (rather than 1000) and then tune the learning rate.
- **Max Tree Depth**: Start with a value of 6 and presumably tune from there.
- **Min Leaf Weight**: Use a modified ratio of $\frac{1}{sqrt(rare\_events)}$, where `rare_events` is the percentage of rare event observations in the dataset.
- **Column Sampling**: Grid search values in the range of 0.3 to 0.5 (more constrained).
- **Row Sampling**: Fixed at the value 1.0.
- **Min Split Gain**: Fixed at the value 0.0.

Finally, Abhishek Thakur, in his tutorial titled *Approaching (Almost) Any Machine Learning Problem*[12] provided a similar table listing out key XGBoost parameters and suggestions for tuning. The tuning ranges for each parameter are much the same with some notable differences. Specifically, he suggests grid searching values for the Min Split Gain (`gamma`) and the regularization penalty terms (`reg_alpha` and `reg_lambda`). He also explore large values for tree size (`max_depth`) values above 10 as well as fixed Min Leaf Weight (`min_child_weight`) values in the range of about 1 to 10.

13.5 Summary

In this tutorial you got insight into how to configure gradient boosting for your own machine learning problems. Specifically you learned:

- About the trade-off in the number of trees and the shrinkage and good defaults for sub-sampling.
- Different ideas on limiting tree size and good defaults for tree depth and the number of terminal nodes.
- Grid search strategies used by a top Kaggle competition winner.

In the next tutorial you will learn how to design controlled experiments to tune the number and size of trees in an XGBoost model.

[11] https://www.youtube.com/watch?v=7YnVZrabTA8
[12] https://goo.gl/fOyhF3

Chapter 14

Tune the Number and Size of Decision Trees with XGBoost

Gradient boosting involves the creation and addition of decision trees sequentially, each attempting to correct the mistakes of the learners that came before it. This raises the question as to how many trees (weak learners or estimators) to configure in your gradient boosting model and how big each tree should be. In this tutorial you will discover how to design a systematic experiment to select the number and size of decision trees to use on your problem. After reading this tutorial you will know:

- How to evaluate the effect of adding more decision trees to your XGBoost model.
- How to evaluate the effect of creating larger decision trees to your XGBoost model.
- How to investigate the relationship between the number and depth of trees on your problem.

Let's get started.

14.1 Tune the Number of Decision Trees

Most implementations of gradient boosting are configured by default with a relatively small number of trees, such as hundreds or thousands. The general reason is that on most problems, adding more trees beyond a limit does not improve the performance of the model. The reason is in the way that the boosted tree model is constructed, sequentially where each new tree attempts to model and correct for the errors made by the sequence of previous trees. Quickly, the model reaches a point of diminishing returns.

We can demonstrate this point of diminishing returns easily on the Otto dataset (see Section 11.1). The number of trees (or rounds) in an XGBoost model is specified to the `XGBClassifier` or `XGBRegressor` class in the `n_estimators` argument. The default in the XGBoost library is 100. Using scikit-learn we can perform a grid search of the `n_estimators` model parameter, evaluating a series of values from 50 to 350 with a step size of 50 (50, 150, 200, 250, 300, 350).

```
# grid search
model = XGBClassifier()
```

```
n_estimators = range(50, 400, 50)
param_grid = dict(n_estimators=n_estimators)
kfold = StratifiedKFold(n_splits=10, shuffle=True, random_state=7)
grid_search = GridSearchCV(model, param_grid, scoring="neg_log_loss", n_jobs=-1, cv=kfold)
result = grid_search.fit(X, label_encoded_y)
```

Listing 14.1: Example of Grid Searching the Number of Trees.

We can perform this grid search on the Otto dataset, using 10-fold cross-validation, requiring 60 models to be trained (6 configurations $\times$ 10 folds). The full code listing is provided below for completeness.

```
# XGBoost on Otto dataset, Tune n_estimators
from pandas import read_csv
from xgboost import XGBClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import StratifiedKFold
from sklearn.preprocessing import LabelEncoder
import matplotlib
matplotlib.use('Agg')
from matplotlib import pyplot
# load data
data = read_csv('train.csv')
dataset = data.values
# split data into X and y
X = dataset[:,0:94]
y = dataset[:,94]
# encode string class values as integers
label_encoded_y = LabelEncoder().fit_transform(y)
# grid search
model = XGBClassifier()
n_estimators = range(50, 400, 50)
param_grid = dict(n_estimators=n_estimators)
kfold = StratifiedKFold(n_splits=10, shuffle=True, random_state=7)
grid_search = GridSearchCV(model, param_grid, scoring="neg_log_loss", n_jobs=-1, cv=kfold)
grid_result = grid_search.fit(X, label_encoded_y)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
  print("%f (%f) with: %r" % (mean, stdev, param))
# plot
pyplot.errorbar(n_estimators, means, yerr=stds)
pyplot.title("XGBoost n_estimators vs Log Loss")
pyplot.xlabel('n_estimators')
pyplot.ylabel('Log Loss')
pyplot.savefig('n_estimators.png')
```

Listing 14.2: Worked Example of Tuning the Number of Trees.

Running this example prints the following results.

```
Best: -0.001152 using {'n_estimators': 250}
-0.010970 (0.001083) with: {'n_estimators': 50}
-0.001239 (0.001730) with: {'n_estimators': 100}
```

```
-0.001163 (0.001715) with: {'n_estimators': 150}
-0.001153 (0.001702) with: {'n_estimators': 200}
-0.001152 (0.001702) with: {'n_estimators': 250}
-0.001152 (0.001704) with: {'n_estimators': 300}
-0.001153 (0.001706) with: {'n_estimators': 350}
```

Listing 14.3: Sample Output of Worked Example of Tuning the Number of Trees.

We can see that the cross-validation log loss scores are negative. This is because the scikit-learn cross-validation framework inverted them. The reason is that internally, the framework requires that all metrics that are being optimized are to be maximized, whereas log loss is a minimization metric. It can easily be made maximizing by inverting the scores.

The best number of trees was `n_estimators=250` resulting in a log loss of 0.001152, but really not a significant difference from `n_estimators=200`. In fact, there is not a large relative difference in the number of trees between 100 and 350 if we plot the results. Below is line graph showing the relationship between the number of trees and mean (inverted) logarithmic loss, with the standard deviation shown as error bars.

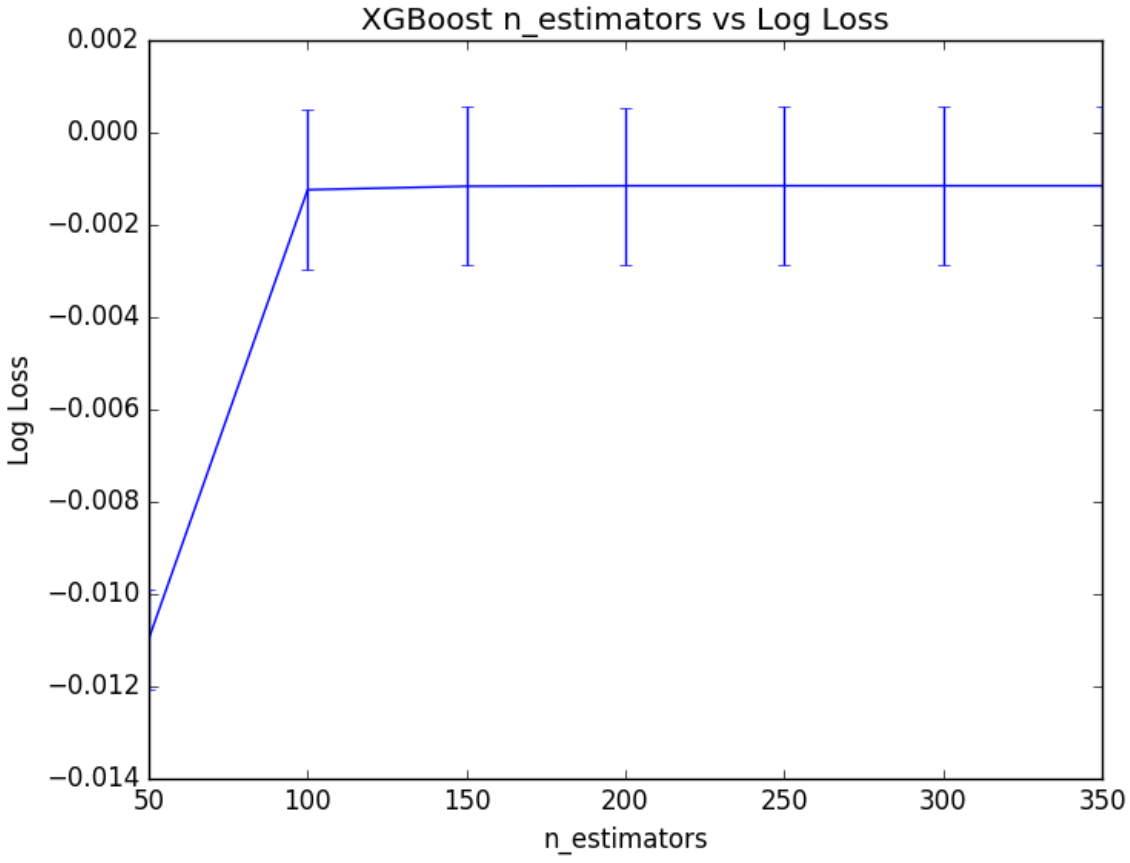

Figure 14.1: Plot of the Results from Tuning the Number of Trees in XGBoost

14.2 Tune the Size of Decision Trees

In gradient boosting, we can control the size of decision trees, also called the number of layers or the depth. Shallow trees are expected to have poor performance because they capture few details of the problem and are generally referred to as weak learners. Deeper trees generally capture too many details of the problem and overfit the training dataset, limiting the ability to make good predictions on new data. Generally, boosting algorithms are configured with weak learners, decision trees with few layers, sometimes as simple as just a root node, also called a decision stump rather than a decision tree. The maximum depth can be specified in the `XGBClassifier` and `XGBRegressor` wrapper classes for XGBoost in the `max_depth` parameter. This parameter takes an integer value and defaults to a value of 3.

```
model = XGBClassifier(max_depth=3)
```

Listing 14.4: Example of Setting the Max Tree Depth.

We can tune this hyperparameter of XGBoost using the grid search infrastructure in scikit-learn on the Otto dataset. Below we evaluate odd values for `max_depth` between 1 and 9 (1, 3, 5, 7, 9). Each of the 5 configurations is evaluated using 10-fold cross-validation, resulting in 50 models being constructed. The full code listing is provided below for completeness.

```
# XGBoost on Otto dataset, Tune max_depth
from pandas import read_csv
from xgboost import XGBClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import StratifiedKFold
from sklearn.preprocessing import LabelEncoder
import matplotlib
matplotlib.use('Agg')
from matplotlib import pyplot
# load data
data = read_csv('train.csv')
dataset = data.values
# split data into X and y
X = dataset[:,0:94]
y = dataset[:,94]
# encode string class values as integers
label_encoded_y = LabelEncoder().fit_transform(y)
# grid search
model = XGBClassifier()
max_depth = range(1, 11, 2)
print(max_depth)
param_grid = dict(max_depth=max_depth)
kfold = StratifiedKFold(n_splits=10, shuffle=True, random_state=7)
grid_search = GridSearchCV(model, param_grid, scoring="neg_log_loss", n_jobs=-1, cv=kfold,
    verbose=1)
grid_result = grid_search.fit(X, label_encoded_y)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
  print("%f (%f) with: %r" % (mean, stdev, param))
# plot
```

```
pyplot.errorbar(max_depth, means, yerr=stds)
pyplot.title("XGBoost max_depth vs Log Loss")
pyplot.xlabel('max_depth')
pyplot.ylabel('Log Loss')
pyplot.savefig('max_depth.png')
```

Listing 14.5: Worked Example of Tuning the Tree Depth.

Running this example prints the log loss for each `max_depth`. The optimal configuration was `max_depth=5` resulting in a log loss of 0.001236.

```
Best: -0.001236 using {'max_depth': 5}
-0.026235 (0.000898) with: {'max_depth': 1}
-0.001239 (0.001730) with: {'max_depth': 3}
-0.001236 (0.001701) with: {'max_depth': 5}
-0.001237 (0.001701) with: {'max_depth': 7}
-0.001237 (0.001701) with: {'max_depth': 9}
```

Listing 14.6: Sample Output of Worked Example of Tuning the Tree Depth.

Reviewing the plot of log loss scores, we can see a marked jump from `max_depth=1` to `max_depth=3` then pretty even performance for the rest the values of `max_depth`. Although the best score was observed for `max_depth=5`, it is interesting to note that there was practically little difference between using `max_depth=3` or `max_depth=7`. This suggests a point of diminishing returns in `max_depth` on a problem that you can tease out using grid search. A graph of `max_depth` values is plotted against (inverted) logarithmic loss below.

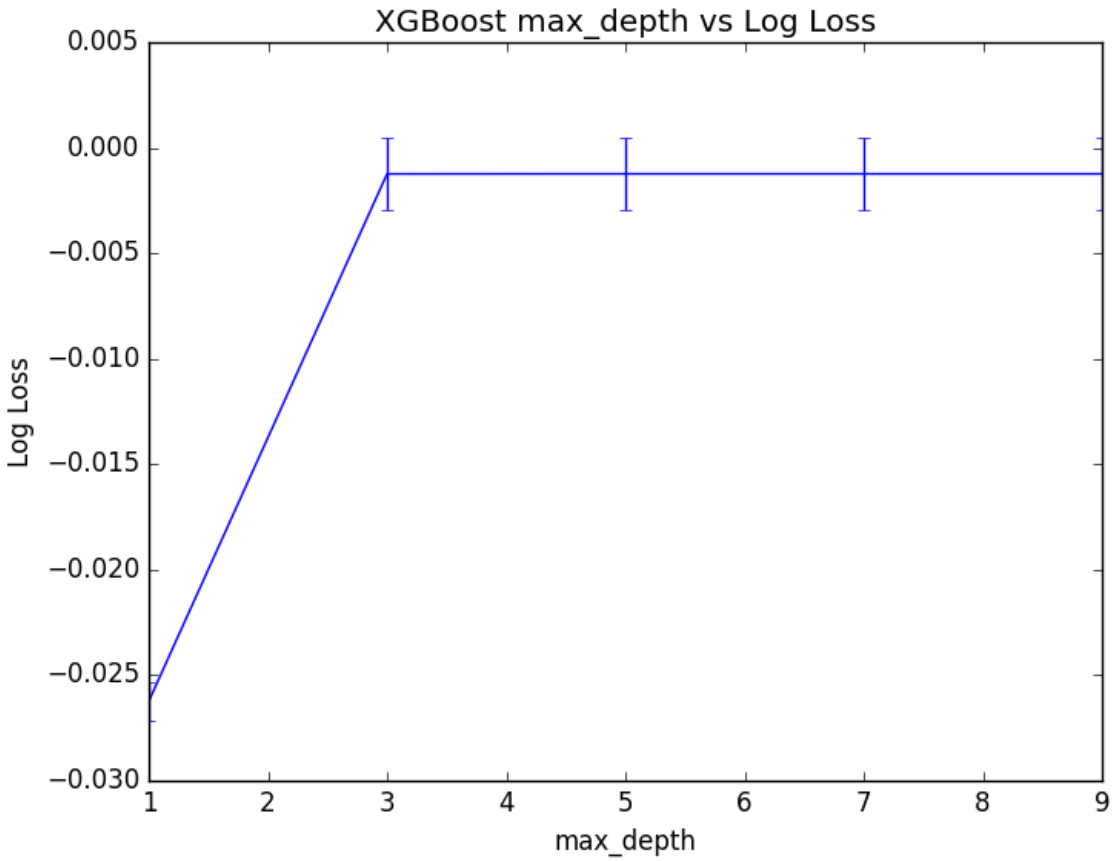

Figure 14.2: Plot of the Results from Tuning the Tree Size in XGBoost

14.3 Tune The Number and Size of Trees

There is a relationship between the number of trees in the model and the depth of each tree. We would expect that deeper trees would result in fewer trees being required in the model, and the inverse where simpler trees (such as decision stumps) require many more trees to achieve similar results. We can investigate this relationship by evaluating a grid of `n_estimators` and `max_depth` configuration values. To avoid the evaluation taking too long, we will limit the total number of configuration values evaluated. Parameters were chosen to tease out the relationship rather than optimize the model.

We will create a grid of 4 different `n_estimators` values (50, 100, 150, 200) and 4 different `max_depth` values (2, 4, 6, 8) and each combination will be evaluated using 10-fold cross-validation. A total of $4 \times 4 \times 10$ or 160 models will be trained and evaluated. The full code listing is provided below.

```
# XGBoost on Otto dataset, Tune n_estimators and max_depth
from pandas import read_csv
from xgboost import XGBClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import StratifiedKFold
from sklearn.preprocessing import LabelEncoder
import matplotlib
```

```
matplotlib.use('Agg')
from matplotlib import pyplot
import numpy
# load data
data = read_csv('train.csv')
dataset = data.values
# split data into X and y
X = dataset[:,0:94]
y = dataset[:,94]
# encode string class values as integers
label_encoded_y = LabelEncoder().fit_transform(y)
# grid search
model = XGBClassifier()
n_estimators = [50, 100, 150, 200]
max_depth = [2, 4, 6, 8]
print(max_depth)
param_grid = dict(max_depth=max_depth, n_estimators=n_estimators)
kfold = StratifiedKFold(n_splits=10, shuffle=True, random_state=7)
grid_search = GridSearchCV(model, param_grid, scoring="neg_log_loss", n_jobs=-1, cv=kfold,
    verbose=1)
grid_result = grid_search.fit(X, label_encoded_y)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
  print("%f (%f) with: %r" % (mean, stdev, param))
# plot results
scores = numpy.array(means).reshape(len(max_depth), len(n_estimators))
for i, value in enumerate(max_depth):
    pyplot.plot(n_estimators, scores[i], label='depth: ' + str(value))
pyplot.legend()
pyplot.xlabel('n_estimators')
pyplot.ylabel('Log Loss')
pyplot.savefig('n_estimators_vs_max_depth.png')
```

Listing 14.7: Worked Example of Tuning the Number of Trees and Tree Depth.

Running the code produces a listing of the logloss for each parameter pair.

```
Best: -0.001141 using {'n_estimators': 200, 'max_depth': 4}
-0.012127 (0.001130) with: {'n_estimators': 50, 'max_depth': 2}
-0.001351 (0.001825) with: {'n_estimators': 100, 'max_depth': 2}
-0.001278 (0.001812) with: {'n_estimators': 150, 'max_depth': 2}
-0.001266 (0.001796) with: {'n_estimators': 200, 'max_depth': 2}
-0.010545 (0.001083) with: {'n_estimators': 50, 'max_depth': 4}
-0.001226 (0.001721) with: {'n_estimators': 100, 'max_depth': 4}
-0.001150 (0.001704) with: {'n_estimators': 150, 'max_depth': 4}
-0.001141 (0.001693) with: {'n_estimators': 200, 'max_depth': 4}
-0.010341 (0.001059) with: {'n_estimators': 50, 'max_depth': 6}
-0.001237 (0.001701) with: {'n_estimators': 100, 'max_depth': 6}
-0.001163 (0.001688) with: {'n_estimators': 150, 'max_depth': 6}
-0.001154 (0.001679) with: {'n_estimators': 200, 'max_depth': 6}
-0.010342 (0.001059) with: {'n_estimators': 50, 'max_depth': 8}
-0.001237 (0.001701) with: {'n_estimators': 100, 'max_depth': 8}
-0.001161 (0.001688) with: {'n_estimators': 150, 'max_depth': 8}
```

```
-0.001153 (0.001679) with: {'n_estimators': 200, 'max_depth': 8}
```

Listing 14.8: Sample Output of Worked Example of Tuning the Number of Trees and Tree Depth.

We can see that the best result was achieved with a `n_estimators=200` and `max_depth=4`, similar to the best values found from the previous two rounds of standalone parameter tuning (`n_estimators=250`, `max_depth=5`). We can plot the relationship between each series of `max_depth` values for a given `n_estimators`.

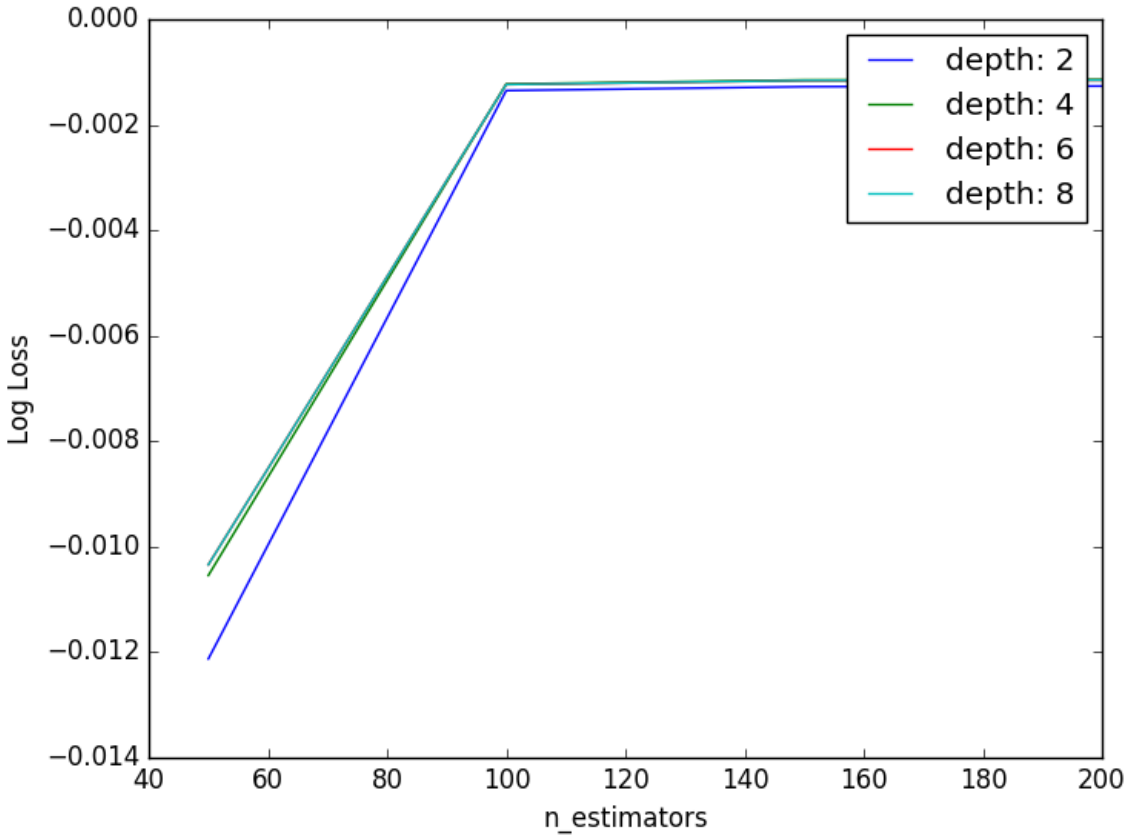

Figure 14.3: Plot of the Results from Tining the Number of Trees and Max Tree Depth in XGBoost.

The lines overlap making it hard to see the relationship, but generally we can see the interaction we expect. Fewer boosted trees are required with increased tree depth. Further, we would expect the increase complexity provided by deeper individual trees to result in greater overfitting of the training data which would be exacerbated by having more trees, in turn resulting in a lower cross-validation score. We don't see this here as our trees are not that deep nor do we have too many. Exploring this expectation is left as an exercise you could explore yourself.

14.4 Summary

In this tutorial you discovered how to tune the number and depth of decision trees when using gradient boosting with XGBoost in Python. Specifically you learned:

- How to tune the number of decision trees in an XGBoost model.
- How to tune the depth of decision trees in an XGBoost model.
- How to jointly tune the number of trees and tree depth in an XGBoost model.

In the next tutorial you will discover how you can tune the learning rate and number of trees in an XGBoost model.

Chapter 15

Tune Learning Rate and Number of Trees with XGBoost

A problem with gradient boosted decision trees is that they are quick to learn and overfit training data. One effective way to slow down learning in the gradient boosting model is to use a learning rate, also called shrinkage (or eta in XGBoost documentation). In this tutorial you will discover the effect of the learning rate in gradient boosting and how to tune it on your machine learning problem using the XGBoost library in Python. After reading this tutorial you will know:

- The effect learning rate has on the gradient boosting model.
- How to tune learning rate on your machine learning on your problem.
- How to tune the trade-off between the number of boosted trees and learning rate on your problem.

Let's get started.

15.1 Slow Learning in Gradient Boosting with a Learning Rate

Gradient boosting involves creating and adding trees to the model sequentially. New trees are created to correct the residual errors in the predictions from the existing sequence of trees. The effect is that the model can quickly fit, then overfit the training dataset. A technique to slow down the learning in the gradient boosting model is to apply a weighting factor for the corrections by new trees when added to the model. This weighting is called the shrinkage factor or the learning rate, depending on the literature or the tool.

Naive gradient boosting is the same as gradient boosting with shrinkage where the shrinkage factor is set to 1.0. Setting values less than 1.0 has the effect of making less corrections for each tree added to the model. This in turn results in more trees that must be added to the model. It is common to have small values in the range of 0.1 to 0.3, as well as values less than 0.1. Let's investigate the effect of the learning rate on a standard machine learning dataset.

15.2 Tuning Learning Rate

When creating gradient boosting models with XGBoost using the scikit-learn wrapper, the `learning_rate` parameter can be set to control the weighting of new trees added to the model. We can use the grid search capability in scikit-learn to evaluate the effect on logarithmic loss of training a gradient boosting model with different learning rate values. We will hold the number of trees constant at the default of 100 and evaluate of suite of standard values for the learning rate on the Otto dataset (see Section 11.1).

```
learning_rate = [0.0001, 0.001, 0.01, 0.1, 0.2, 0.3]
```

Listing 15.1: Example of Learning Rates to Evaluate.

There are 6 variations of learning rate to be tested and each variation will be evaluated using 10-fold cross-validation, meaning that there is a total of 6×10 or 60 XGBoost models to be trained and evaluated. The log loss for each learning rate will be printed as well as the value that resulted in the best performance.

```
# XGBoost on Otto dataset, Tune learning_rate
from pandas import read_csv
from xgboost import XGBClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import StratifiedKFold
from sklearn.preprocessing import LabelEncoder
import matplotlib
matplotlib.use('Agg')
from matplotlib import pyplot
# load data
data = read_csv('train.csv')
dataset = data.values
# split data into X and y
X = dataset[:,0:94]
y = dataset[:,94]
# encode string class values as integers
label_encoded_y = LabelEncoder().fit_transform(y)
# grid search
model = XGBClassifier()
learning_rate = [0.0001, 0.001, 0.01, 0.1, 0.2, 0.3]
param_grid = dict(learning_rate=learning_rate)
kfold = StratifiedKFold(n_splits=10, shuffle=True, random_state=7)
grid_search = GridSearchCV(model, param_grid, scoring="neg_log_loss", n_jobs=-1, cv=kfold)
grid_result = grid_search.fit(X, label_encoded_y)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
  print("%f (%f) with: %r" % (mean, stdev, param))
# plot
pyplot.errorbar(learning_rate, means, yerr=stds)
pyplot.title("XGBoost learning_rate vs Log Loss")
pyplot.xlabel('learning_rate')
pyplot.ylabel('Log Loss')
pyplot.savefig('learning_rate.png')
```

Listing 15.2: Worked Example of Tuning the Learning Rate.

Running this example prints the best result as well as the log loss for each of the evaluated learning rates.

```
Best: -0.001156 using {'learning_rate': 0.2}
-2.155497 (0.000081) with: {'learning_rate': 0.0001}
-1.841069 (0.000716) with: {'learning_rate': 0.001}
-0.597299 (0.000822) with: {'learning_rate': 0.01}
-0.001239 (0.001730) with: {'learning_rate': 0.1}
-0.001156 (0.001684) with: {'learning_rate': 0.2}
-0.001158 (0.001666) with: {'learning_rate': 0.3}
```

Listing 15.3: Sample Output of Worked Example of Tuning the Learning Rate.

Interestingly, we can see that the best learning rate was 0.2. This is a high learning rate and it suggest that perhaps the default number of trees of 100 is too low and needs to be increased. We can also plot the effect of the learning rate of the (inverted) log loss scores, although the log10-like spread of chosen `learning_rate` values means that most are squashed down the left-hand side of the plot near zero.

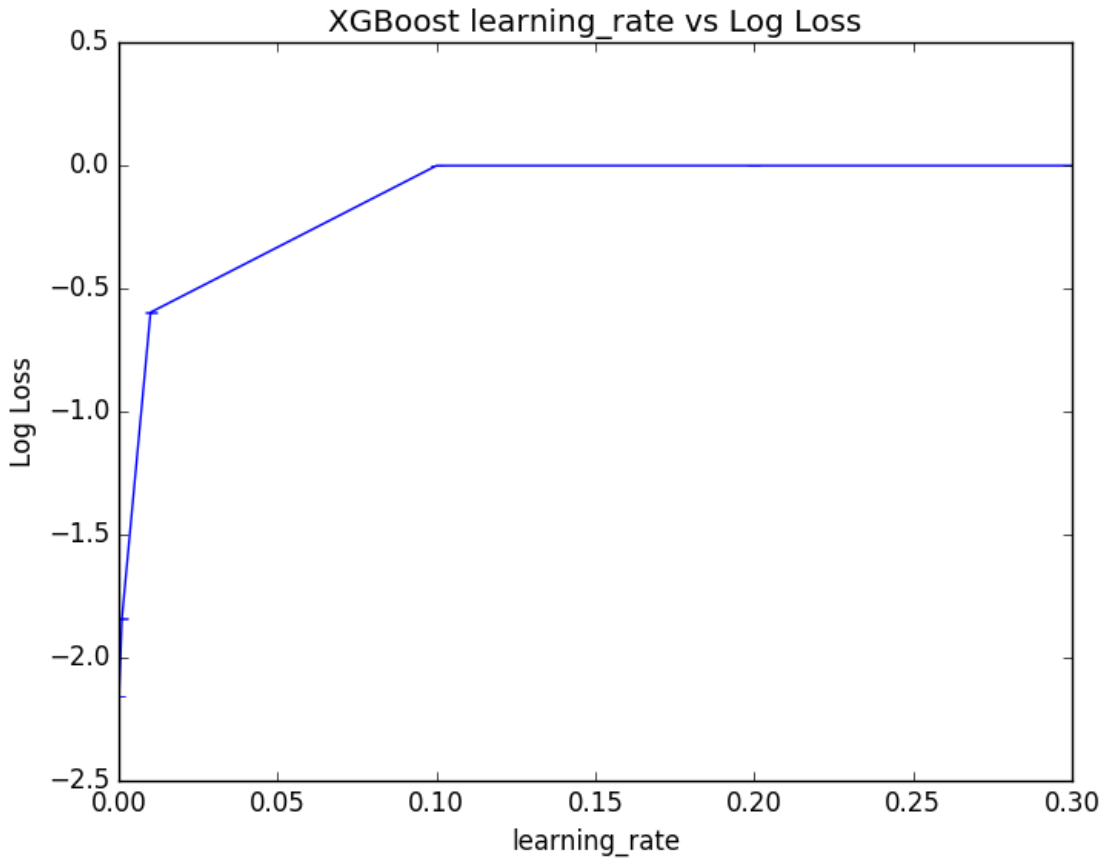

Figure 15.1: Plot of the Results from Tuning the Learning Rate in XGBoost

Next, we will look at varying the number of trees whilst varying the learning rate.

15.3 Tuning Learning Rate and the Number of Trees

Smaller learning rates generally require more trees to be added to the model. We can explore this relationship by evaluating a grid of parameter pairs. The number of decision trees will be varied from 100 to 500 and the learning rate varied on a log10 scale from 0.0001 to 0.1.

```
n_estimators = [100, 200, 300, 400, 500]
learning_rate = [0.0001, 0.001, 0.01, 0.1]
```

Listing 15.4: Example of Learning Rates and Number of Trees to Evaluate.

There are 5 variations of `n_estimators` and 4 variations of `learning_rate`. Each combination will be evaluated using 10-fold cross-validation, so that is a total of $4 \times 5 \times 10$ or 200 XGBoost models that must be trained and evaluated. The expectation is that for a given learning rate, performance will improve and then plateau as the number of trees is increased. The full code listing is provided below.

```
# XGBoost on Otto dataset, Tune learning_rate and n_estimators
from pandas import read_csv
from xgboost import XGBClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import StratifiedKFold
from sklearn.preprocessing import LabelEncoder
import matplotlib
matplotlib.use('Agg')
from matplotlib import pyplot
import numpy
# load data
data = read_csv('train.csv')
dataset = data.values
# split data into X and y
X = dataset[:,0:94]
y = dataset[:,94]
# encode string class values as integers
label_encoded_y = LabelEncoder().fit_transform(y)
# grid search
model = XGBClassifier()
n_estimators = [100, 200, 300, 400, 500]
learning_rate = [0.0001, 0.001, 0.01, 0.1]
param_grid = dict(learning_rate=learning_rate, n_estimators=n_estimators)
kfold = StratifiedKFold(n_splits=10, shuffle=True, random_state=7)
grid_search = GridSearchCV(model, param_grid, scoring="neg_log_loss", n_jobs=-1, cv=kfold)
grid_result = grid_search.fit(X, label_encoded_y)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
  print("%f (%f) with: %r" % (mean, stdev, param))
# plot results
scores = numpy.array(means).reshape(len(learning_rate), len(n_estimators))
for i, value in enumerate(learning_rate):
    pyplot.plot(n_estimators, scores[i], label='learning_rate: ' + str(value))
pyplot.legend()
pyplot.xlabel('n_estimators')
```

```
pyplot.ylabel('Log Loss')
pyplot.savefig('n_estimators_vs_learning_rate.png')
```

Listing 15.5: Worked Example of Tuning the Learning Rate and Number of Trees.

Running the example prints the best combination as well as the log loss for each evaluated pair.

```
Best: -0.001152 using {'n_estimators': 300, 'learning_rate': 0.1}
-2.155497 (0.000081) with: {'n_estimators': 100, 'learning_rate': 0.0001}
-2.115540 (0.000159) with: {'n_estimators': 200, 'learning_rate': 0.0001}
-2.077211 (0.000233) with: {'n_estimators': 300, 'learning_rate': 0.0001}
-2.040386 (0.000304) with: {'n_estimators': 400, 'learning_rate': 0.0001}
-2.004955 (0.000373) with: {'n_estimators': 500, 'learning_rate': 0.0001}
-1.841069 (0.000716) with: {'n_estimators': 100, 'learning_rate': 0.001}
-1.572384 (0.000692) with: {'n_estimators': 200, 'learning_rate': 0.001}
-1.364543 (0.000699) with: {'n_estimators': 300, 'learning_rate': 0.001}
-1.196490 (0.000713) with: {'n_estimators': 400, 'learning_rate': 0.001}
-1.056687 (0.000728) with: {'n_estimators': 500, 'learning_rate': 0.001}
-0.597299 (0.000822) with: {'n_estimators': 100, 'learning_rate': 0.01}
-0.214311 (0.000929) with: {'n_estimators': 200, 'learning_rate': 0.01}
-0.080729 (0.000982) with: {'n_estimators': 300, 'learning_rate': 0.01}
-0.030533 (0.000949) with: {'n_estimators': 400, 'learning_rate': 0.01}
-0.011769 (0.001071) with: {'n_estimators': 500, 'learning_rate': 0.01}
-0.001239 (0.001730) with: {'n_estimators': 100, 'learning_rate': 0.1}
-0.001153 (0.001702) with: {'n_estimators': 200, 'learning_rate': 0.1}
-0.001152 (0.001704) with: {'n_estimators': 300, 'learning_rate': 0.1}
-0.001153 (0.001708) with: {'n_estimators': 400, 'learning_rate': 0.1}
-0.001153 (0.001708) with: {'n_estimators': 500, 'learning_rate': 0.1}
```

Listing 15.6: Sample Output of Worked Example of Tuning the Learning Rate and Number of Trees.

We can see that the best result observed was a learning rate of 0.1 with 300 trees. It is hard to pick out trends from the raw data and small negative log loss results. Below is a plot of each learning rate as a series showing log loss performance as the number of trees is varied.

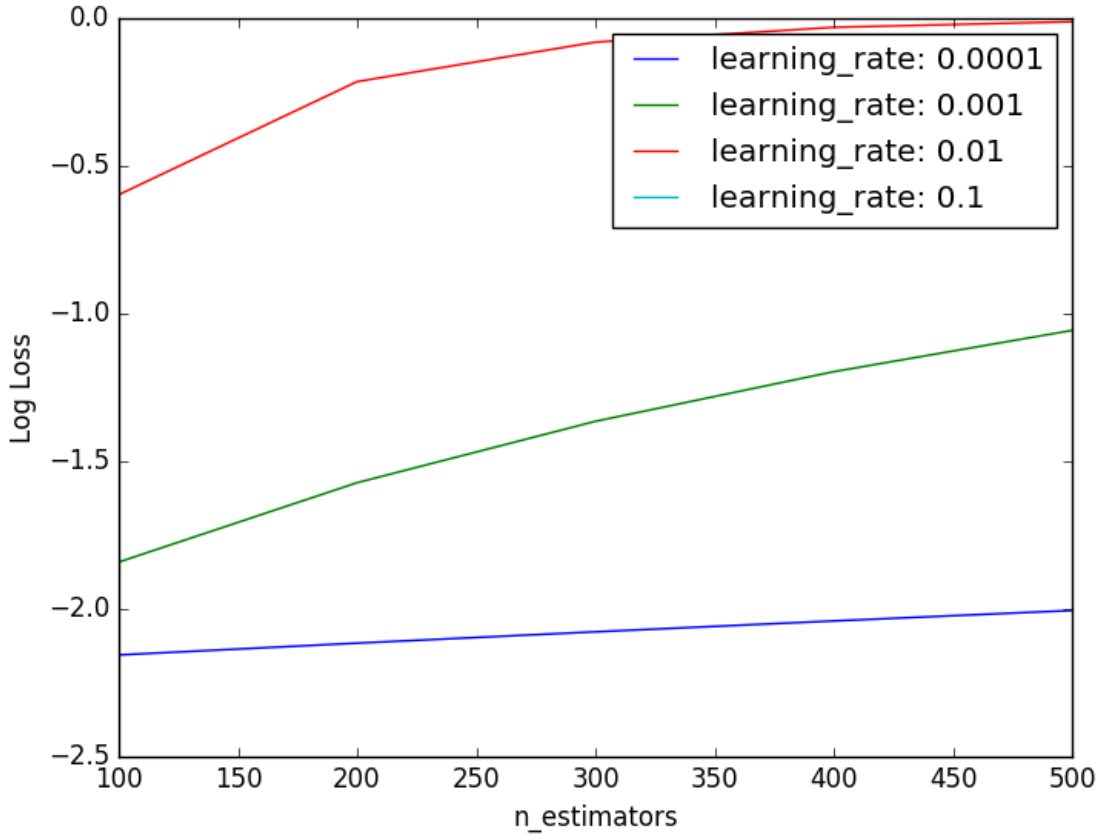

Figure 15.2: Plot of the Results from Tuning the Learning Rate and Number of Trees in XGBoost

We can see that the expected general trend holds, where the performance (inverted log loss) improves as the number of trees is increased. Performance is generally poor for the smaller learning rates, suggesting that a much larger number of trees may be required. We may need to increase the number of trees to many thousands which may be quite computationally expensive. The results for `learning_rate=0.1` are obscured due the large y-axis scale of the graph. We can extract the performance measure for just `learning_rate=0.1` and plot them directly.

```
# Plot performance for learning_rate=0.1
from matplotlib import pyplot
n_estimators = [100, 200, 300, 400, 500]
loss = [-0.001239, -0.001153, -0.001152, -0.001153, -0.001153]
pyplot.plot(n_estimators, loss)
pyplot.xlabel('n_estimators')
pyplot.ylabel('Log Loss')
pyplot.title('XGBoost learning_rate=0.1 n_estimators vs Log Loss')
pyplot.show()
```

Listing 15.7: Plot Performance of learning_rate

Running this code shows the increased performance as the number of trees are added, followed by a plateau in performance across 400 and 500 trees.

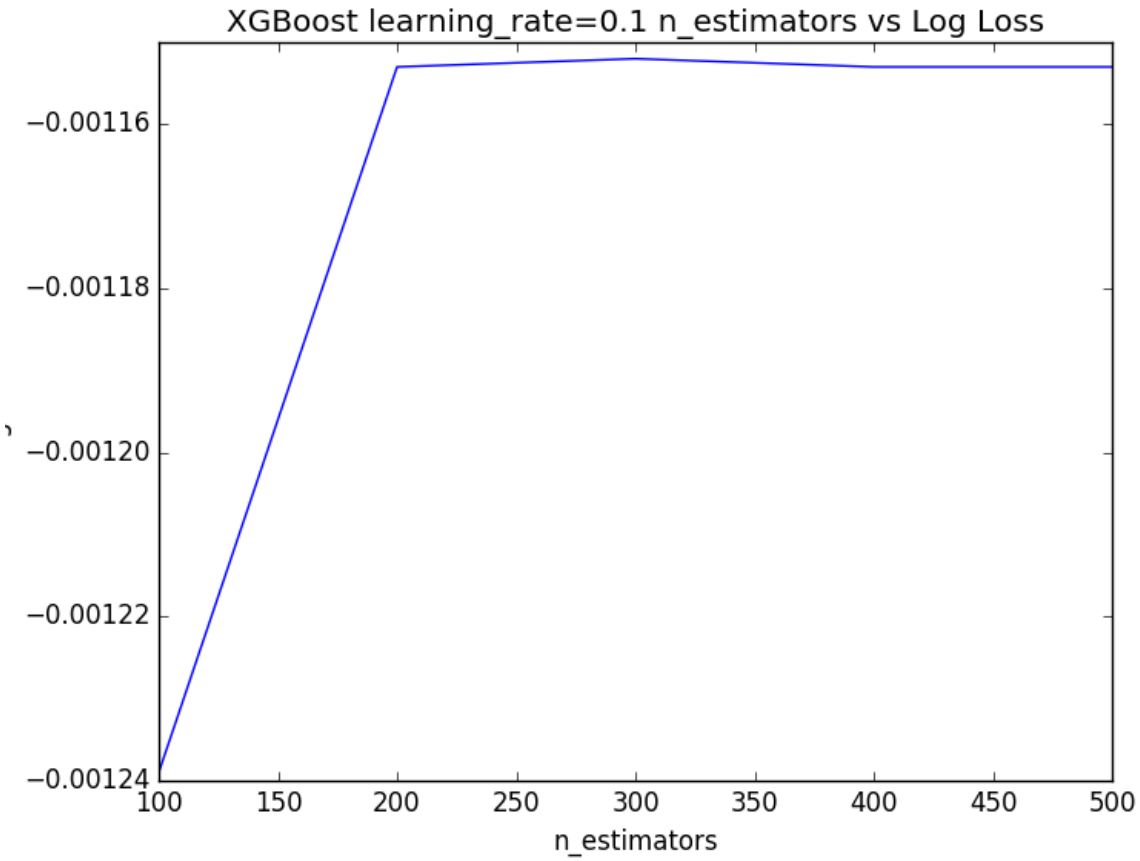

Figure 15.3: Plot of Learning Rate=0.1 and varying the Number of Trees in XGBoost.

15.4 Summary

In this tutorial you discovered the effect of weighting the addition of new trees to a gradient boosting model, called shrinkage or the learning rate. Specifically, you learned:

- That adding a learning rate is intended to slow down the adaptation of the model to the training data.
- How to evaluate a range of learning rate values on your machine learning problem.
- How to evaluate the relationship of varying both the number of trees and the learning rate on your problem.

In the next tutorial you will discover how to tune the sampling methods for stochastic gradient boosting models in XGBoost.

Chapter 16

Tuning Stochastic Gradient Boosting with XGBoost

A simple technique for ensembling decision trees involves training trees on subsamples of the training dataset. Subsets of the rows in the training data can be taken to train individual trees called bagging. When subsets of rows of the training data are also taken when calculating each split point, this is called random forest. These techniques can also be used in the gradient tree boosting model in a technique called stochastic gradient boosting. In this tutorial you will discover stochastic gradient boosting and how to tune the sampling parameters using XGBoost with scikit-learn in Python. After reading this tutorial you will know:

- The rationale behind training trees on subsamples of data and how this can be used in gradient boosting.
- How to tune row-based subsampling in XGBoost using scikit-learn.
- How to tune column-based subsampling by both tree and split-point in XGBoost.

Let's get started.

16.1 Stochastic Gradient Boosting

Gradient boosting is a greedy procedure. New decision trees are added to the model to correct the residual error of the existing model. Each decision tree is created using a greedy search procedure to select split points that best minimize an objective function. This can result in trees that use the same attributes and even the same split points again and again. Bagging is a technique where a collection of decision trees are created, each from a different random subset of rows from the training data. The effect is that better performance is achieved from the ensemble of trees because the randomness in the sample allows slightly different trees to be created, adding variance to the ensembled predictions.

Random forest takes this one step further, by allowing the features (columns) to be subsampled when choosing split points, adding further variance to the ensemble of trees. These same techniques can be used in the construction of decision trees in gradient boosting in a variation called stochastic gradient boosting. It is common to use aggressive sub-samples of the training data such as 40% to 80%.

16.2 Tutorial Overview

In this tutorial we are going to look at the effect of different subsampling techniques in gradient boosting. We will tune three different flavors of stochastic gradient boosting supported by the XGBoost library in Python, specifically:

1. Subsampling of rows in the dataset when creating each tree.
2. Subsampling of columns in the dataset when creating each tree.
3. Subsampling of columns for each split in the dataset when creating each tree.

16.3 Tuning Row Subsampling

Row subsampling involves selecting a random sample of the training dataset without replacement. Row subsampling can be specified in the scikit-learn wrapper of the XGBoost class in the `subsample` parameter. The default is 1.0 which is no sub-sampling. We can use the grid search capability built into scikit-learn to evaluate the effect of different `subsample` values from 0.1 to 1.0 on the Otto dataset (see Section 11.1).

```
[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 1.0]
```

Listing 16.1: Example of Row Sampling Rates To Evaluate.

There are 9 variations of `subsample` and each model will be evaluated using 10-fold cross-validation, meaning that 9×10 or 90 models need to be trained and tested. The complete code listing is provided below.

```
# XGBoost on Otto dataset, tune subsample
from pandas import read_csv
from xgboost import XGBClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import StratifiedKFold
from sklearn.preprocessing import LabelEncoder
import matplotlib
matplotlib.use('Agg')
from matplotlib import pyplot
# load data
data = read_csv('train.csv')
dataset = data.values
# split data into X and y
X = dataset[:,0:94]
y = dataset[:,94]
# encode string class values as integers
label_encoded_y = LabelEncoder().fit_transform(y)
# grid search
model = XGBClassifier()
subsample = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 1.0]
param_grid = dict(subsample=subsample)
kfold = StratifiedKFold(n_splits=10, shuffle=True, random_state=7)
grid_search = GridSearchCV(model, param_grid, scoring="neg_log_loss", n_jobs=-1, cv=kfold)
grid_result = grid_search.fit(X, label_encoded_y)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
```

```
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
  print("%f (%f) with: %r" % (mean, stdev, param))
# plot
pyplot.errorbar(subsample, means, yerr=stds)
pyplot.title("XGBoost subsample vs Log Loss")
pyplot.xlabel('subsample')
pyplot.ylabel('Log Loss')
pyplot.savefig('subsample.png')
```

Listing 16.2: Worked Example of Tuning the Row Sampling Rate.

Running this example prints the best configuration as well as the log loss for each tested configuration. We can see that the best results achieved were 0.3, or training trees using a 30% sample of the training dataset.

```
Best: -0.000647 using {'subsample': 0.3}
-0.001156 (0.000286) with: {'subsample': 0.1}
-0.000765 (0.000430) with: {'subsample': 0.2}
-0.000647 (0.000471) with: {'subsample': 0.3}
-0.000659 (0.000635) with: {'subsample': 0.4}
-0.000717 (0.000849) with: {'subsample': 0.5}
-0.000773 (0.000998) with: {'subsample': 0.6}
-0.000877 (0.001179) with: {'subsample': 0.7}
-0.001007 (0.001371) with: {'subsample': 0.8}
-0.001239 (0.001730) with: {'subsample': 1.0}
```

Listing 16.3: Sample Output of Worked Example of Tuning the Row Sampling Rate.

We can plot these mean and standard deviation log loss values to get a better understanding of how performance varies with the `subsample` value.

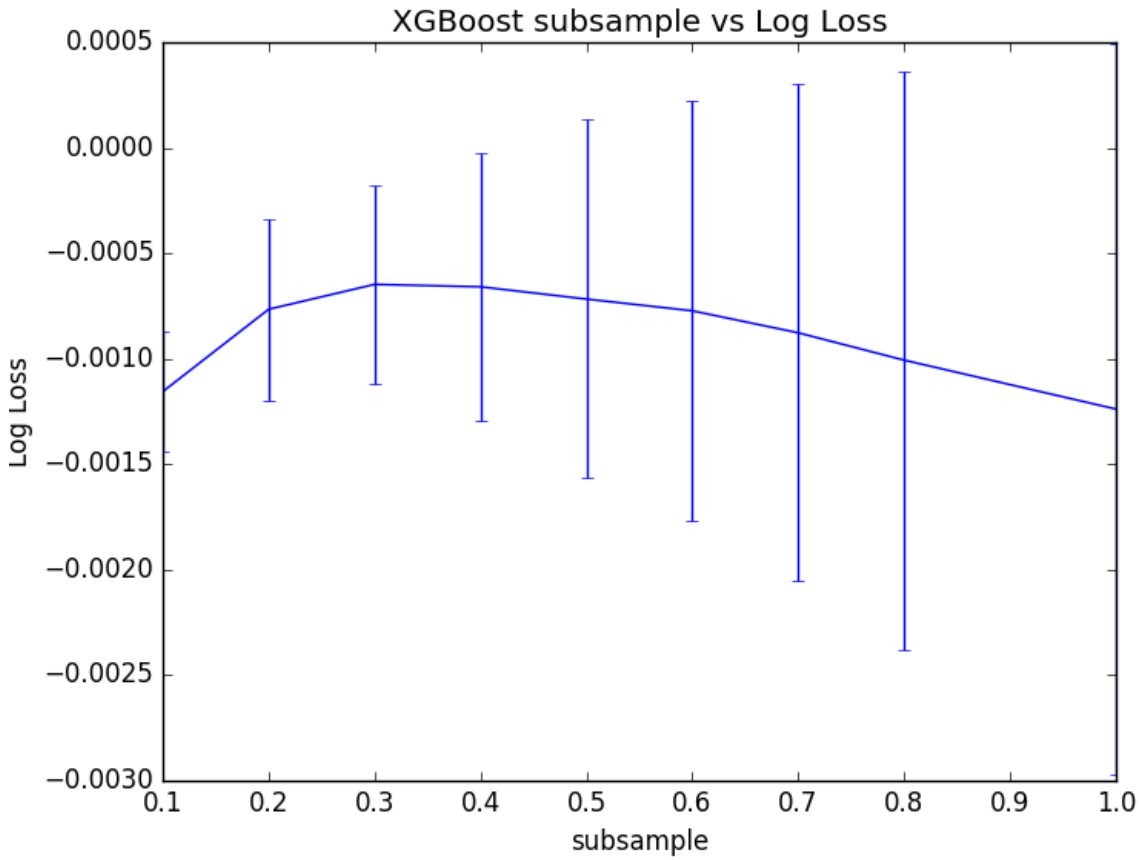

Figure 16.1: Plot of the Results from Tuning the Row Sample Rate in XGBoost

We can see that indeed 30% has the best mean performance, but we can also see that as the ratio increased, the variance in performance grows quite markedly. It is interesting to note that the mean performance of all `subsample` values outperforms the mean performance without subsampling (`subsample=1.0`).

16.4 Tuning Column Subsampling By Tree

We can also create a random sample of the features (or columns) to use prior to creating each decision tree in the boosted model. In the XGBoost wrapper for scikit-learn, this is controlled by the `colsample_bytree` parameter. The default value is 1.0 meaning that all columns are used in each decision tree. We can evaluate values for `colsample_bytree` between 0.1 and 1.0 incrementing by 0.1.

```
[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 1.0]
```

Listing 16.4: Example of Column Sampling Rates By Tree To Evaluate.

The full code listing is provided below.

```
# XGBoost on Otto dataset, tune colsample_bytree
from pandas import read_csv
from xgboost import XGBClassifier
```

```
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import StratifiedKFold
from sklearn.preprocessing import LabelEncoder
import matplotlib
matplotlib.use('Agg')
from matplotlib import pyplot
# load data
data = read_csv('train.csv')
dataset = data.values
# split data into X and y
X = dataset[:,0:94]
y = dataset[:,94]
# encode string class values as integers
label_encoded_y = LabelEncoder().fit_transform(y)
# grid search
model = XGBClassifier()
colsample_bytree = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 1.0]
param_grid = dict(colsample_bytree=colsample_bytree)
kfold = StratifiedKFold(n_splits=10, shuffle=True, random_state=7)
grid_search = GridSearchCV(model, param_grid, scoring="neg_log_loss", n_jobs=-1, cv=kfold)
grid_result = grid_search.fit(X, label_encoded_y)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
  print("%f (%f) with: %r" % (mean, stdev, param))
# plot
pyplot.errorbar(colsample_bytree, means, yerr=stds)
pyplot.title("XGBoost colsample_bytree vs Log Loss")
pyplot.xlabel('colsample_bytree')
pyplot.ylabel('Log Loss')
pyplot.savefig('colsample_bytree.png')
```

Listing 16.5: Worked Example of Tuning the Column Sampling Rate By Tree.

Running this example prints the best configuration as well as the log loss for each tested configuration. We can see that the best performance for the model was `colsample_bytree=1.0`. This suggests that subsampling columns on this problem does not add value.

```
Best: -0.001239 using {'colsample_bytree': 1.0}
-0.298955 (0.002177) with: {'colsample_bytree': 0.1}
-0.092441 (0.000798) with: {'colsample_bytree': 0.2}
-0.029993 (0.000459) with: {'colsample_bytree': 0.3}
-0.010435 (0.000669) with: {'colsample_bytree': 0.4}
-0.004176 (0.000916) with: {'colsample_bytree': 0.5}
-0.002614 (0.001062) with: {'colsample_bytree': 0.6}
-0.001694 (0.001221) with: {'colsample_bytree': 0.7}
-0.001306 (0.001435) with: {'colsample_bytree': 0.8}
-0.001239 (0.001730) with: {'colsample_bytree': 1.0}
```

Listing 16.6: Sample Output of Worked Example of Tuning the Column Sampling Rate By Tree.

Plotting the results, we can see the performance of the model plateau (at least at this scale) with values between 0.5 to 1.0.

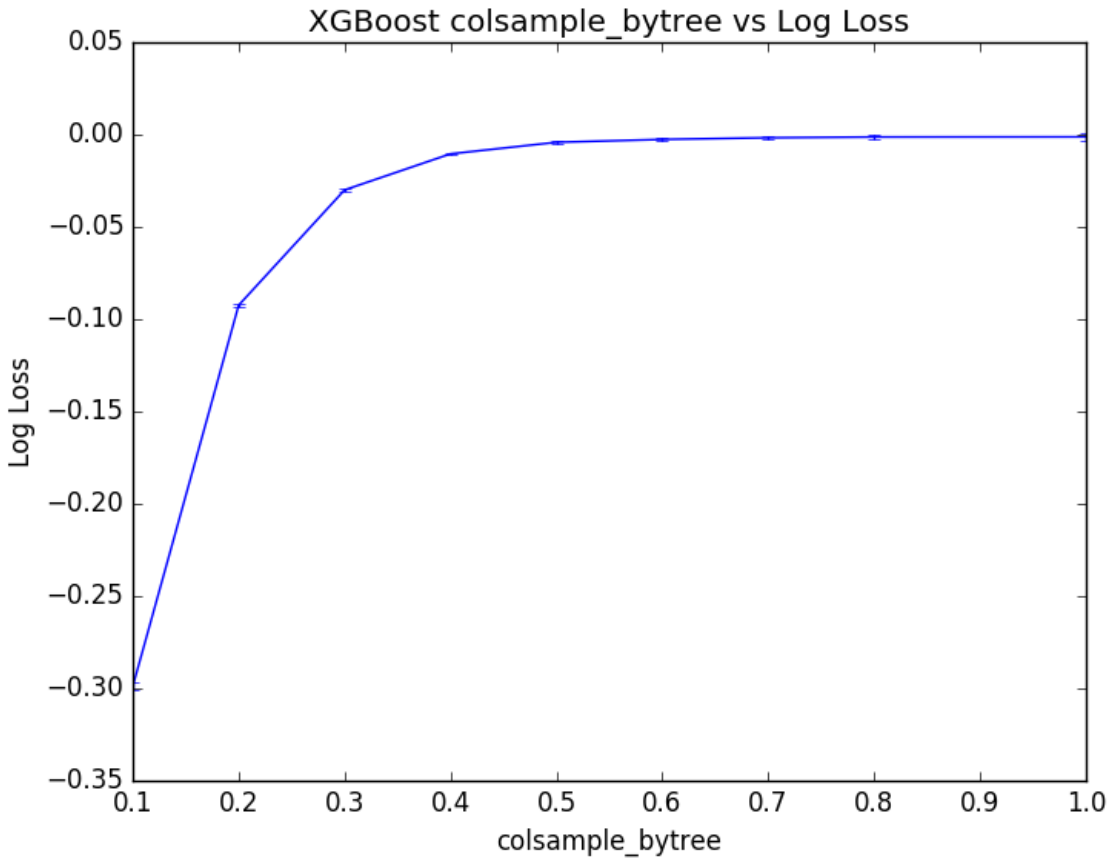

Figure 16.2: Plot of the Results from Tuning the Column Sample Rate By Tree in XGBoost

16.5 Tuning Column Subsampling By Split

Rather than subsample the columns once for each tree, we can subsample them at each split in the decision tree. In principle, this is the approach used in random forest. We can set the size of the sample of columns used at each split in the `colsample_bylevel` parameter in the XGBoost wrapper classes for scikit-learn. As before, we will vary the ratio from 10% to the default of 100%. The full code listing is provided below.

```
# XGBoost on Otto dataset, tune colsample_bylevel
from pandas import read_csv
from xgboost import XGBClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import StratifiedKFold
from sklearn.preprocessing import LabelEncoder
import matplotlib
matplotlib.use('Agg')
from matplotlib import pyplot
# load data
data = read_csv('train.csv')
dataset = data.values
# split data into X and y
X = dataset[:,0:94]
```

```
y = dataset[:,94]
# encode string class values as integers
label_encoded_y = LabelEncoder().fit_transform(y)
# grid search
model = XGBClassifier()
colsample_bylevel = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 1.0]
param_grid = dict(colsample_bylevel=colsample_bylevel)
kfold = StratifiedKFold(n_splits=10, shuffle=True, random_state=7)
grid_search = GridSearchCV(model, param_grid, scoring="neg_log_loss", n_jobs=-1, cv=kfold)
grid_result = grid_search.fit(X, label_encoded_y)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
  print("%f (%f) with: %r" % (mean, stdev, param))
# plot
pyplot.errorbar(colsample_bylevel, means, yerr=stds)
pyplot.title("XGBoost colsample_bylevel vs Log Loss")
pyplot.xlabel('colsample_bylevel')
pyplot.ylabel('Log Loss')
pyplot.savefig('colsample_bylevel.png')
```

Listing 16.7: Worked Example of Tuning the Column Sampling Rate By Split.

Running this example prints the best configuration as well as the log loss for each tested configuration. We can see that the best results were achieved by setting `colsample_bylevel` to 70%, resulting in an (inverted) log loss of -0.001062, which is better than -0.001239 seen when setting the per-tree column sampling to 100%. This suggest to not give up on column subsampling if per-tree results suggest using 100% of columns, and to instead try per-split column subsampling.

```
Best: -0.001062 using {'colsample_bylevel': 0.7}
-0.159455 (0.007028) with: {'colsample_bylevel': 0.1}
-0.034391 (0.003533) with: {'colsample_bylevel': 0.2}
-0.007619 (0.000451) with: {'colsample_bylevel': 0.3}
-0.002982 (0.000726) with: {'colsample_bylevel': 0.4}
-0.001410 (0.000946) with: {'colsample_bylevel': 0.5}
-0.001182 (0.001144) with: {'colsample_bylevel': 0.6}
-0.001062 (0.001221) with: {'colsample_bylevel': 0.7}
-0.001071 (0.001427) with: {'colsample_bylevel': 0.8}
-0.001239 (0.001730) with: {'colsample_bylevel': 1.0}
```

Listing 16.8: Sample Output of Worked Example of Tuning the Column Sampling Rate By Split.

We can plot the performance of each `colsample_bylevel` variation. The results show relatively low variance and seemingly a plateau in performance after a value of 0.3 at this scale.

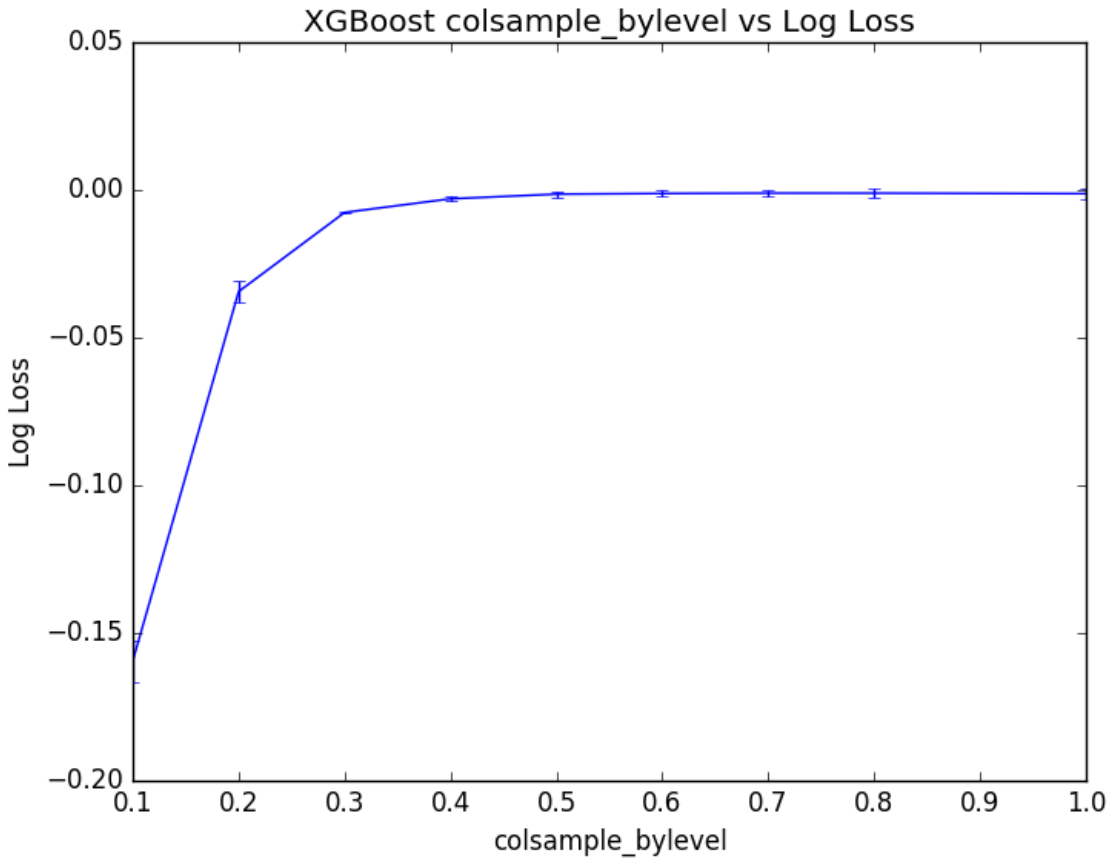

Figure 16.3: Plot of the Results from Tuning the Column Sample Rate By Split in XGBoost

16.6 Summary

In this tutorial you discovered stochastic gradient boosting with XGBoost in Python. Specifically, you learned:

- About stochastic boosting and how you can subsample your training data to improve the generalization of your model
- How to tune row subsampling with XGBoost in Python and scikit-learn.
- How to tune column subsampling with XGBoost both per-tree and per-split.

This concludes Part IV of this book. Next in Part V the book will be concluded and you will discover resources that you can use to get help with gradient boosting and the XGBoost library.

Part V

Conclusions

Chapter 17

How Far You Have Come

You made it. Well done. Take a moment and look back at how far you have come.

- You learned about the gradient boosting algorithm and variations such as the addition of shrinkage, penalties and stochastic gradient boosting.
- You developed your first XGBoost model, learned how to best prepare data for modeling with XGBoost and discovered how to effectively evaluate the performance of trained models.
- You learned how to serialized trained models for later use and to evaluate the importance of input variables. You also learned how to scale up XGBoost models to use all of the cores on your system, and how to best use a large number of scores on very hardware systems in the cloud.
- You learned how to configure gradient boosted models including common configuration heuristics. You also learned how to design controlled experiments to tune important hyperparameters in the model.

Don't make light of this. You have come a long way in a short amount of time. You have developed the important and valuable skill of being able to work through machine learning problems with gradient boosting end-to-end using Python. This is a platform that is used by a majority of working data scientist professionals. The sky is the limit.

I want to take a moment and sincerely thank you for letting me help you start your XGBoost journey with in Python. I hope you keep learning and have fun as you continue to master machine learning.

Chapter 18

Getting More Help

This book has given you a foundation for applying XGBoost in your own machine learning projects, but there is still a lot more to learn. In this chapter you will discover the places that you can go to get more help with the XGBoost library as well as gradient boosting in general.

18.1 Gradient Boosting Papers

The seminal papers on gradient boosting are quite readable, containing lots of useful tips for configuring and applying the algorithm to your problem. Below are a few select papers you may consider reading.

- *Arcing the edge*, 1998.
  http://goo.gl/Kedl7V
- *Boosting Algorithms as Gradient Descent in Function Space*, 1999.
  http://goo.gl/Tz9hNg
- *Greedy Function Approximation: A Gradient Boosting Machine*, 1999.
  https://goo.gl/5dbi4V
- *Stochastic Gradient Boosting*, 1999.
  https://goo.gl/LHKp4T
- *XGBoost: A Scalable Tree Boosting System*, 2016 (XGBoost paper).
  http://goo.gl/aFfSef

18.2 Gradient Boosting in Textbooks

Most good machine learning textbooks cover the gradient boosting algorithm in great detail. These can provide a good resource if you are looking to better understand how the algorithm works.

- Section 8.2.3 Boosting, page 321, *An Introduction to Statistical Learning: with Applications in R*.
  http://www.amazon.com/dp/1461471370?tag=inspiredalgor-20

- Section 8.6 Boosting, page 203 and Section 14.5 Stochastic Gradient Boosting, page 390, in *Applied Predictive Modeling*.
  http://www.amazon.com/dp/1461468485?tag=inspiredalgor-20
- Section 16.4 Boosting, page 556, *Machine Learning: A Probabilistic Perspective*.
  http://www.amazon.com/dp/0262018020?tag=inspiredalgor-20
- Chapter 10 Boosting and Additive Trees, page 337, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*.
  http://www.amazon.com/dp/0387848576?tag=inspiredalgor-20

18.3 Python Machine Learning

Python is a growing platform for applied machine learning. The strong attraction is because Python is a fully featured programming language (unlike R) and as such you can use the same code and libraries in developing your model as you use to deploy the model into operations. The premier machine learning library in Python is scikit-learn built on top of SciPy.

- Visit the scikit-learn home page to learn more about the library and it's capabilities.
  http://scikit-learn.org
- Visit the SciPy home page to learn more about the SciPy platform for scientific computing in Python.
  http://scipy.org
- Machine Learning Mastery with Python, the precursor to this book.
  https://machinelearningmastery.com/machine-learning-with-python

18.4 XGBoost Library

XGBoost is a fantastic but fast moving library. Large updates are still being made to the API and it is good to stay abreast of changes. You also need to know where you can ask questions to get more help with the platform.

- XGBoost GitHub repository containing code, documentation and demos.
  https://github.com/dmlc/xgboost
- XGBoost Documentation Homepage.
  https://xgboost.readthedocs.io
- XGBoost Python API.
  http://xgboost.readthedocs.io/en/latest/python/python_api.html
- XGBoost User Group for asking detailed questions about the library.
  https://groups.google.com/forum/#!forum/xgboost-user/
- Awesome XGBoost, listing useful resources.
  https://github.com/dmlc/xgboost/blob/master/demo/README.md

I am always here to help if have any questions. You can email me directly via `jason@MachineLearningMastery.com` and put this book title in the subject of your email.