EBOOK

GridGain

Using In-Memory Computing for Continuous Machine and Deep Learning

Part 2: Hands-On Machine Learning Using Apache Ignite

Many software developers view machine learning as rocket science: too complicated and not living up to its full potential. Without an understanding of machine learning's underlying principles, it can seem too complex to tackle. For those who do leap in and get started, it can be hard to determine what tools to use. For those already implementing some form of machine learning, the concept of continuous learning (what we humans do as a matter of course in our daily lives) seems out of reach: it takes too long to manage, move, and (re) train models, given the amount of data needed to create a reasonably accurate model.

But continuous machine and deep learning IS an attainable goal. This machine learning eBook series is designed to give developers a basic understanding of machine and deep learning, hands-on machine learning experience with Apache® Ignite®, help getting machine learning up and running quickly, and tips to avoid some of the more common machine learning challenges.

INTRODUCTION

Apache Ignite supports algorithms for regression, classification, and clustering. This eBook explains how to use Apache Ignite for machine learning by walking through three of the most common and useful of these algorithms—linear regression, k-nearest neighbors, and k-means—as applied to sample datasets.

The General Machine Learning Approach

In each of these exercises follow the following general process:

1. Split the raw data into training data and test data.
2. Read the training data and the test data.
3. Store the training data and the test data in Ignite.
4. Use the training data to create the model.
5. Apply the model to the test data.
6. Determine the accuracy of the model.

Splitting the Raw Data

When building a model using machine learning, first find source data for both model training and testing. Split the source data into a larger subset for training the model, and then a smaller subset for testing the trained model. In general, the larger the training subset, the more accurate the model will be. The model will be fit to the training subset, so the testing subset should be different data than what the model has seen before. Using a new set of data gives an unbiased read on whether the model has learned enough from the training set to work against any data.

THIS EBOOK SERIES

The series is broken into five parts:

- Part 1: A Machine and Deep Learning Primer
- **Part 2:** Hands-on Machine Learning Using Apache Ignite (this ebook)
- **Part 3:** Hands-on Machine Learning for Fraud Detection at Scale Using Apache Ignite
- **Part 4:** Hands-on Deep Learning Using the Apache Ignite Genetic Algorithm
- **Part 5:** Hands-on Deep Learning Using TensorFlow with Apache Ignite

The series is designed to be flexible and useful. Part 1, A Machine and Deep Learning Primer, provides a basic understanding of the concepts. This eBook, Part 2, is the first hands-on eBook. It covers the built-in machine learning algorithms included with Apache Ignite and how to use them.

How much data should the training and testing sets have? One popular approach is an 80:20 split: 80 percent of the raw data for training and 20 percent for testing. The usual range is from 60 to 80 percent of the data for training and from 40 to 20 percent for testing. The specific ratio is influenced by a variety of factors.

The two sets must be equally representative of the whole: that is, the characteristics of the data (the features being examined, in machine learning terms) ought not to be distributed differently across the two sets. Data selection must also be randomized. Fortunately, randomized selection is built into many of the open source tools designed for data splitting (such as scikit-learn or the built-in splitting within Ignite).

Note that while earlier examples assumed the use of scikit-learn, Ignite and GridGain now have their own built-in splitting. Make sure to use these versions with the examples.

Following Along with the Examples

To follow along with these exercises, download the examples from GitHub. The following examples are all in Java. More recently a Python client was added to Apache Ignite and GridGain. Below are links to the Python client resources:

- https://github.com/gridgain/ml-python-api/ (GridGain repository with Python Plugin with examples)
- https://machine-learning-python-api.readthedocs.io/ (documentation for Python API)

GridGain

LINEAR REGRESSION

Linear regression is one of the most common machine learning algorithms. All the regression algorithms (simple, multiple, and so on) examine the relationship between a set of one or more variables (or features) and a value (the target outcome) dependent on those features. It is that target outcome that is the discoverable goal.

Although linear regression is relatively simple, it is also versatile. Use it to predict any value within a continuous output, such as the revenue a customer generates based on their demographics and previous purchases along with the products to recommend to them, or the expected sales and required inventory on a given day based on historical sales. This exercise walks through the process of training a linear regression model and calculating its R2 score. The R2 score shows how far the variance of actual sample values are from the linear model (a straight line). The term "standard deviation" as a measure of distribution from the mean refers to the square root of the variance.

The Sample Dataset

A standard candidate for linear regression is the estimation of housing prices based on different attributes. It is the "Hello, world" of linear regression training. This walkthrough uses the housing prices dataset available at the UC Irvine machine learning repository.

Now, since the dataset is small it is possible to just load it into standard Java data structures and run the linear regression on it directly within Java. By loading the data into Apache Ignite storage instead, however, the data is distributed across an entire cluster. This means that running the algorithm on the stored data will perform distributed training. When working with larger datasets, as with real machine learning problems, using Ignite storage improves speed by keeping all data in memory, and adding linear, horizontal scale to processing. Ignite becomes very valuable when working with terabytes of training and testing data because it runs the computing in parallel across small segments of the data without having to move the data to the algorithm, and runs model training, testing, and execution against streaming data all at the same time. This is how companies can achieve near real-time continuous learning.

Split the Raw Data into Training Data and Test Data

For this exercise, split the source data into 80 percent for training and 20 percent for testing. Here is the example code for reading the data and creating storage:

```
dataCache = new SandboxMLCache(ignite).fillCacheWith(MLSandboxDatasets.BOSTON_HOUSE_PRICES);
DatasetTrainer<LinearRegressionModel, Double> trainer = new LinearRegressionLSQRTrainer()
   .withEnvironmentBuilder(LearningEnvironmentBuilder.defaultBuilder().withRNGSeed(0));
// Splits dataset to train and test samples with 80/20 proportion.
TrainTestSplit<Integer, Vector> split = new TrainTestDatasetSplitter<Integer, Vector>().split(0.8);
```

Use the Training Data to Create the Linear Regression Model

With the data stored, create the trainer:

```
DatasetTrainer<LinearRegressionModel, Double> trainer = new LinearRegressionLSQRTrainer();
```

And then fit a linear model to the training data:

```
// This vectorizer works with values in cache of Vector class.
Vectorizer<Integer, Vector, Integer, Double> vectorizer = new DummyVectorizer<Integer>()
   .labeled(Vectorizer.LabelCoordinate.FIRST); // FIRST means "label are stored at first coordinate of vector"
LinearRegressionModel model = trainer.fit(
   ignite, dataCache,
   split.getTrainFilter(),
   vectorizer
);
```

Ignite stores data in a key-value (K-V) format. Here, the target value is Price and the contributing features are in the other columns.

Apply the Model to the Test Data

Now it is ready to check the test data against the trained linear model. Use the following code to calculate the residual sum of squares (u), the amount of variance not explained by the model, and the total sum of squares (v) of the test and sample data together. Together, these scores give a measure of the relative accuracy of the model. If the residual sum of squares is a large part of the total sum of squares, or total distribution of the data away from the linear model, the model might not be very useful.

```
double u = 0.0; // Parameters for R^2 score evaluation.
double v = 0.0;
double meanPrice = computeMeanPrice(dataCache, split.getTestFilter(), vectorizer);
ScanQuery<Integer, Vector> qry = new ScanQuery<>(split.getTestFilter());
try (QueryCursor<Cache.Entry<Integer, Vector>> cursor = dataCache.query(qry)) {
   for (Cache.Entry<Integer, Vector> entry : cursor) {
       LabeledVector<Double> vec = vectorizer.apply(entry.getKey(), entry.getValue());

       double realPrice = vec.label();
       double predictedPrice = model.predict(vec.features());

       u += Math.pow(realPrice - predictedPrice, 2);
       v += Math.pow(realPrice - meanPrice, 2);
}
```

Where computeMeanPrice method is defined as:

```
double computeMeanPrice(IgniteCache<Integer, Vector> cache,
   IgniteBiPredicate<Integer, Vector> filter,
   Vectorizer<Integer, Vector, Integer, Double> vectorizer) {

   long countOfExamples = 0;
   double sumOfPrices = 0.0;
   try (QueryCursor<Cache.Entry<Integer, Vector>> cursor = cache.query(new ScanQuery<>(filter))) {
       for (Cache.Entry<Integer, Vector> ent : cursor) {
           sumOfPrices += vectorizer.apply(ent.getKey(), ent.getValue()).label();
           countOfExamples += 1;

   return sumOfPrices / Math.max(countOfExamples, 1);

}
```

Determine the R2 Score of the Model

Now, find the value of R2 directly from the model in Ignite as 1 - u / v:

```
double score = Evaluator.evaluate(
   dataCache, split.getTestFilter(),
   model, vectorizer,
   new RegressionMetrics().withMetric(RegressionMetricValues::r2)
);

System.out.println(">>> R^2 score: " + score);
```


In this example with the sample data, the value is approximately 0.41288 or 41.3 percent, which is low. If R2 is 0, it means the model does not explain any of the variation. If R2 is 1, it means it explains all the variation. 0.41 means linear regression might not be the best model. As a next step try new variables based on combinations of already defined variables, nonlinear regression, or other models such as the Decision Tree regressor.

K-NEAREST NEIGHBOR (K-NN)

Another common machine learning algorithm is k-Nearest Neighbor (k-NN), which is a classification algorithm. While a regression is used to predict results with a continuous output, a classification tries to predict results in a discrete output. That is, it maps input variables into discrete categories. Determining whether images are huskies or wolves, whether a cancer is malignant or benign, or whether a product interests a specific customer are all classification problems.

k-NN classifies an object based upon the class of its k nearest neighbors. In other words, in the loosely clustered effect of the output, the location of the object (which demonstrates similarity with other instances) determines its class. For a two-class problem, always choose an odd number for k, and never let k be a multiple of the number of classes.

This exercise walks through the process of training a classification model with k-NN and calculating the accuracy of that model.

The Sample Dataset

A good candidate for k-NN classification is the Iris flower dataset. The UC Irvine machine learning repository contains the Iris dataset as well.

The Iris flower dataset consists of 150 samples, with 50 each from three different species of Iris flowers (Iris Setosa, Iris Versicolour, and Iris Virginica). The following four features are available for each sample:

- Sepal length (cm)
- Sepal width (cm)
- Petal length (cm)
- Petal width (cm)

With k-NN, create a model that distinguishes between the different species using these four features.

Split the Raw Data into Training Data and Test Data

For this exercise, split the source data into 60 percent for training and 40 percent for testing. The more training data you use, the more accurate the model will be. Try increasing the size of the test sample to 40% to see what happens.

The example code for this process is:

```
// Splits dataset to train and test samples with 60/40 proportion.
TrainTestSplit<Integer, Vector> split = new TrainTestDatasetSplitter<Integer, Vector>().split(0.6);
```

Use the Training Data to Create the k-NN Classification Model

With the data stored, create the trainer:

```
KNNClassificationTrainer trainer = new KNNClassificationTrainer();
```

And then fit a classification model to the training data:


```
// This vectorizer works with values in cache of Vector class.
Vectorizer<Integer, Vector, Integer, Double> vectorizer = new DummyVectorizer<Integer>()
   .labeled(Vectorizer.LabelCoordinate.FIRST); // FIRST means "label are stored at first coordinate of vector"
NNClassificationModel model = trainer.fit(
   ignite, dataCache,
   split.getTrainFilter(),
   vectorizer
)
   .withK(3)
   .withDistanceMeasure(new EuclideanDistance())
   .withStrategy(NNStrategy.WEIGHTED);
```

Ignite stores data in a Key-Value (K-V) format. Here, the target value is the Flower class and the features are in the other columns. This code sets the value of k to 3 to represent the three species. For distance measure, of the several options available (Euclidean, Hamming, Manhattan), use Euclidean. Finally, k-NN can be used as SIMPLE or WEIGHTED. Here, WEIGHTED is specified.

Apply the Model to the Test Data and Determine Accuracy of Model

Now, check the test data against the trained classification model and calculate the accuracy of the model over the test dataset. Use the following code:

```
double accuracy = Evaluator.evaluate(
   dataCache,
   split.getTestFilter(),
   model,
   vectorizer,
   new Accuracy<>()
);
```

Which gives the following value:

```
>>> Model accuracy: 0.95
```

Ignite was able to correctly classify 95 percent of the test data into the three different species.

K-MEANS CLUSTERING

The third algorithm to examine is k-means—the data clustering algorithm. Unlike the two previous ones, this algorithm refers to unsupervised learning. It can even be applied on unlabeled data, which is useful for the first data example. This algorithm can try to group k similar groups of records in our dataset.

This exercise walks through the process of training a clustering model with k-means and then evaluates the model.

The Sample Dataset

This demonstration of clustering algorithm operation uses data about the customers of a wholesale distributor. This data set includes the annual spending in monetary units (m.u.) on a broad range of product categories. Examples are here. Read more about this dataset here.

Split the Raw Data into Training Data and Test Data

For this exercise, first split the source data into 80 percent for training and 20 percent for testing using Ignite's built-in splitter:

```
TrainTestSplit<Integer, Vector> split = new TrainTestDatasetSplitter<Integer, Vector>().split(0.8);
```


Create a Cache in Ignite and Load the Data

Since this dataset is included in the list that comes with examples for the Apache ML module, load it into the cache with a simple command:

```
dataCache = new SandboxMLCache(ignite).fillCacheWith(MLSandboxDatasets.WHOLESALE_CUSTOMERS);
```

Use the Training Data to Create the K-Means Clustering Model

As mentioned earlier, k-means refers to unsupervised learning. So in this case the number of clusters that the records can be split into is not known. In this example, iterate through the number of clusters in a cycle from 1 to 10.

```
for(int amountOfClusters = 1; amountOfClusters < 10; amountOfClusters++) {
   KMeansTrainer trainer = new KMeansTrainer()
       .withAmountOfClusters(amountOfClusters)
       .withDistance(new EuclideanDistance())
       .withEnvironmentBuilder(LearningEnvironmentBuilder.defaultBuilder().withRNGSeed(0))
       .withMaxIterations(50);
```

For this trainer, explicitly set the distance metric, iterate over the number of clusters, fix the seed, and limit the maximum number of iterations of the algorithm.

After initializing the trainer, create a vectorizer:

```
Vectorizer<Integer, Vector, Integer, Double> vectorizer = new DummyVectorizer<Integer>()
   .labeled(Vectorizer.LabelCoordinate.FIRST); // FIRST means "label are stored at first coordinate of vector"
```

This vectorizer sets the rule according to convert the records in the dataset into vectors suitable for models training. In this case, they are clustered by the first attribute in the dataset, the sales channel. It can be retail or HoReCa (see the data for more details.)

It then fits a k-means clustering model to the training data:

```
KMeansModel mdl = trainer.fit(
   ignite, dataCache,
   split.getTrainFilter(),
   vectorizer
);
```

Apply the Model to the Test Data

Now evaluate the quality of our clusterizer. To do this, evaluate the average entropy as a measure of the alignment randomness.

```
double entropy = computeMeanEntropy(dataCache, split.getTestFilter(), vectorizer, mdl);
System.out.println(String.format(">> Clusters mean entropy [%d clusters]: %.2f", amountOfClusters, entropy));
```

The result is as follows:

```
>> Clusters mean entropy [1 clusters]: 0.64
>> Clusters mean entropy [2 clusters]: 0.58
>> Clusters mean entropy [3 clusters]: 0.51
>> Clusters mean entropy [4 clusters]: 0.44
>> Clusters mean entropy [5 clusters]: 0.29
>> Clusters mean entropy [6 clusters]: 0.34
>> Clusters mean entropy [7 clusters]: 0.37
>> Clusters mean entropy [8 clusters]: 0.32
>> Clusters mean entropy [9 clusters]: 0.36
```

Based on this data, it appears that there are five clusters in our dataset, since five clusters has the lowest entropy.

COMPUTE MEAN ENTROPY

Calculate the average entropy for all clusters as follows:

```
Map<Integer, Map<Integer, AtomicInteger>> clusterUniqueLabelCounts = new HashMap<>();
    try (QueryCursor<Cache.Entry<Integer, Vector>> cursor = cache.query(new ScanQuery<>(filter))) {
        for (Cache.Entry<Integer, Vector> ent : cursor) {
            LabeledVector<Double> vec = vectorizer.apply(ent.getKey(), ent.getValue());
            int cluster = model.predict(vec.features());
            int channel = vec.label().intValue();

            if (!clusterUniqueLabelCounts.containsKey(cluster))
                clusterUniqueLabelCounts.put(cluster, new HashMap<>());

            if (!clusterUniqueLabelCounts.get(cluster).containsKey(channel))
                clusterUniqueLabelCounts.get(cluster).put(channel, new AtomicInteger());

            clusterUniqueLabelCounts.get(cluster).get(channel).incrementAndGet();

    double sumOfClusterEntropies = 0.0;
    for (Integer cluster : clusterUniqueLabelCounts.keySet()) {
        Map<Integer, AtomicInteger> labelCounters = clusterUniqueLabelCounts.get(cluster);
        int sizeOfCluster = labelCounters.values().stream().mapToInt(AtomicInteger::get).sum();
        double entropyInCluster = labelCounters.values().stream()
            .mapToDouble(AtomicInteger::get)
            .map(lblsCount -> lblsCount / sizeOfCluster)
            .map(lblProb -> -lblProb * Math.log(lblProb))
            .sum();

        sumOfClusterEntropies += entropyInCluster;

    return sumOfClusterEntropies / clusterUniqueLabelCounts.size();
}
```

Where filter is the test data of the dataset. Code for this example is available on GitHub at this link.

WHERE TO GO NEXT

This eBook showed three of the built-in algorithms—linear regression, k-NN and k-means. After having used these first three, it should be easier to use the rest of them, which are here. The list of built-in algorithms continues to grow:

- Linear regression
- K-means
- Decision tree classification and regression
- k-NN (nearest neighbors) classification, and k-NN regression
- SVM binary classification, and multi-class classification
- Model cross validation
- Logistic regression
- Random forest
- Gradient boosting
- ANN (approximate nearest neighbor)

This series will cover deep learning later, including:

- Genetic algorithms
- Multilayer perceptron
- Integration with TensorFlow

The next eBook, Part 3, will cover a real-world example of fraud detection. Part 4 will cover how to use the genetic algorithm. Part 5 will focus on using TensorFlow. Go online and get started now at the machine and deep learning resource center.

Contact GridGain Systems

To learn more about how GridGain can help your business, please email our sales team at sales@gridgain.com, call us at +1 (650) 241-2281 (US) or +44 (0)208 610 0666 (Europe), or complete the form at https://www.gridgain.com/contact to have us contact you.

About GridGain Systems

GridGain Systems is revolutionizing real-time data access and processing with the GridGain in-memory computing platform built on Apache® Ignite™. GridGain and Apache Ignite are used by tens of thousands of global enterprises in financial services, fintech, software, e-commerce, retail, online business services, healthcare, telecom and other major sectors, with a client list that includes ING, Raymond James, American Express, Societe Generale, Finastra, IHS Markit, ServiceNow, Marketo, RingCentral, American Airlines, Agilent, and UnitedHealthcare. GridGain delivers unprecedented speed and massive scalability to both legacy and greenfield applications. Deployed on a distributed cluster of commodity servers, GridGain software can reside between the application and data layers (RDBMS, NoSQL and Apache® Hadoop®), requiring no rip-and-replace of the existing databases, or it can be deployed as an in-memory transactional SQL database. GridGain is the most comprehensive in-memory computing platform for high-volume ACID transactions, real-time analytics, web-scale applications, continuous learning and hybrid transactional/analytical processing (HTAP). For more information on GridGain products and services, visit www.gridgain.com.