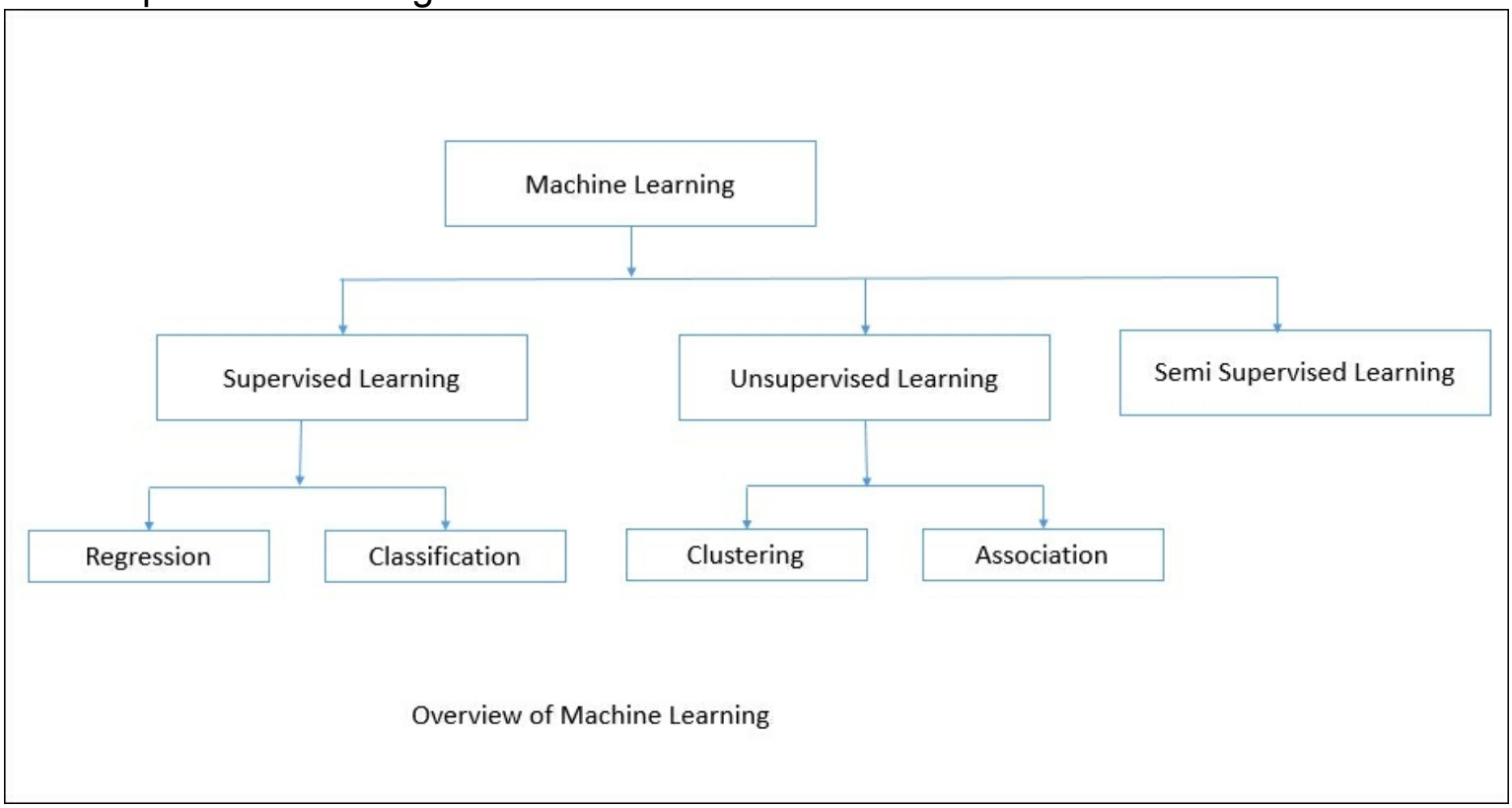
Chapter 6. Machine Learning Using FlinkML

In the previous chapter, we talked about how to solve complex event-processing problems using the Flink CEP library. In this chapter, we are going to see how to do machine learning using Flink's machine learning library, called FlinkML. FlinkML consists of a set of supported algorithms, which can be used to solve real-life use cases. Throughout this chapter, we will look at what algorithms are available in FlinkML and how to apply them.
Before diving deep into FlinkML, let us first try to understand basic machine learning principles.

What is machine learning?

Machine learning is a stream of engineering which uses mathematics to allow machines to make classifications, predictions, recommendations, and so on, based on the data provided to them. This area is vast, and we could spend years discussing it. But in order to keep our discussion focused, we will discuss only what is required for the scope of this book.
Very broadly, machine learning can be divided into three big categories:

- Supervised learning
- Unsupervised learning
- Semi supervised learning



Overview of Machine Learning

The preceding diagram shows a broad classification of machine learning algorithms. Now let's discuss these in detail.

Supervised learning

In supervised learning, we are generally given an input dataset, which is a historical record of actual events. We are also given what the expected output should look like. Using the

historical data, we choose which factors contributed to the results. Such attributes are called features. Using the historical data, we understand how the previous results were calculated and apply that same understanding to the data on which we want to make predictions.
Supervised learning can be again subdivided into:

- Regression
- Classification

Regression

In regression problems, we try to predict results using inputs from a continuous function. Regression means predicting the score of one variable based on the scores of another variable. The variable we will be predicting is called the criterion variable, and the variable from which we will be doing our predictions is called the predictor variable. There can be more than one predictor variable; in this case, we need to find the best fitting line, called the regression line.

> **Note**
> You can read more about regression at https://en.wikipedia.org/wiki/Regression_analysis.

Some very common algorithms used for solving regression problem are as follows:

- Logistic regression
- Decision trees
- Support Vector Machine (SVM)
- Naive Bayes
- Random forest
- Linear regression
- Polynomial regression

Classification

In classification, we predict the output in discrete results. Classification, being a part of supervised learning, also needs the input data and sample output to be given. Here, based on the features, we try to classify the results into sets of defined categories. For instance, based on the features given, classify records of people into male or female. Or, based on customer behavior, predict if he/she would buy a product or not. Or based on the e-mail content and sender, predict if the e-mail is spam or not. Refer to https://en.wikipedia.org/wiki/Statistical_classification.
In order to understand the difference between regression and classification, consider the example of stock data. Regression algorithms can help to predict the value of stock in upcoming days, while classification algorithms can help decide whether to buy the stock or not.

Unsupervised learning

Unsupervised learning does not give us any idea about how our results should look. Instead, it allows us to group data based on the features of the attributes. We derive the clustering based on the relationships among the records.

Unlike supervised learning, there is no validation we can do to verify our results, which means there is no feedback method to teach us whether we did right or wrong. Unsupervised learning is primarily based on clustering algorithms.

Clustering

In order to understand clustering more easily, let's consider an example; let's say we have 20,000 news articles on various topics and we have to group them based on their content . In this case, we can use clustering algorithms, which would group set of articles into small groups.

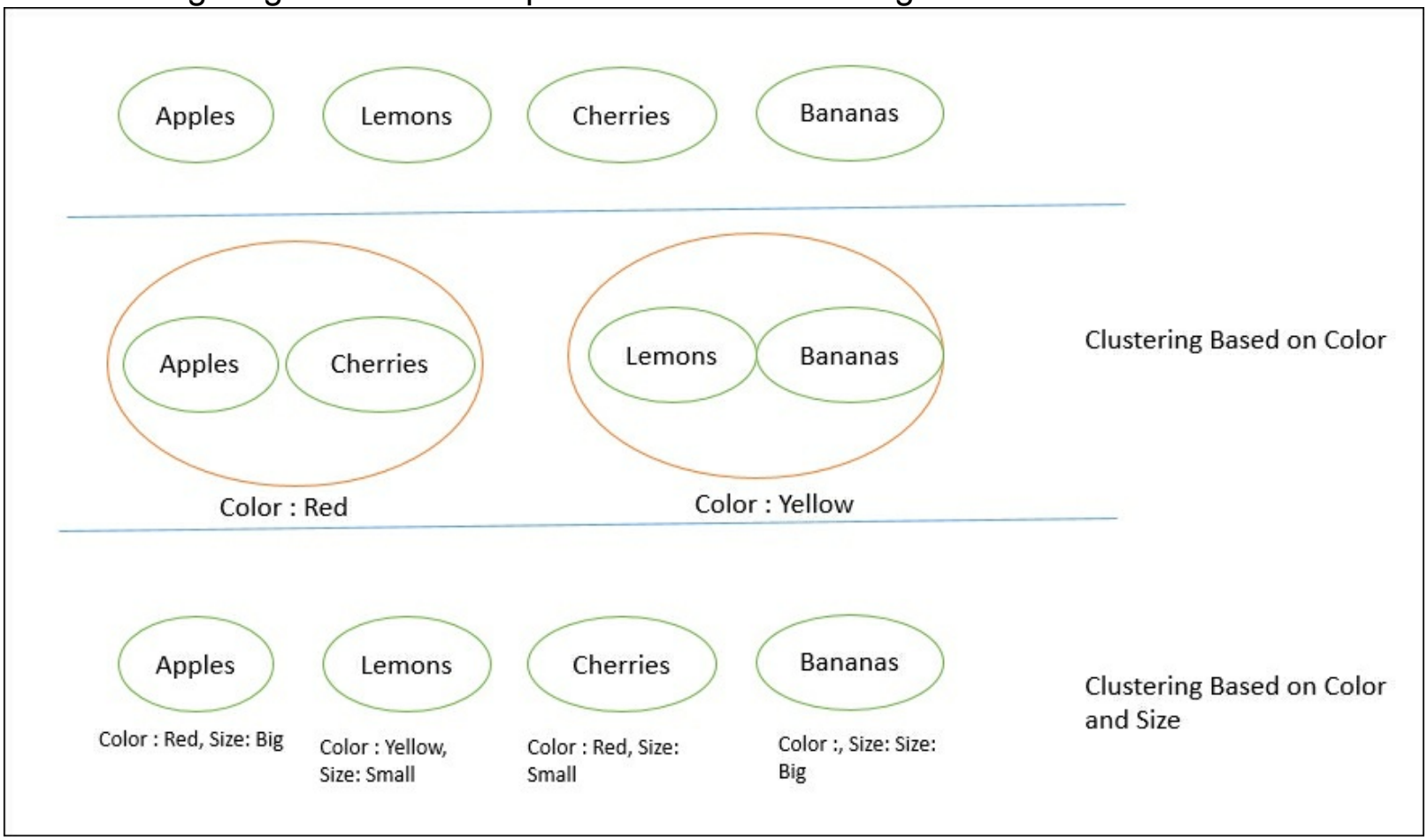
We can also consider the basic example of fruits. Let's say we have apples, bananas, lemons, and cherries in a fruit basket and we need to classify them into groups. If we look at their colors, we can classify them into two groups:

- **Red color group**: Apples and cherries
- **Yellow color group**: Bananas and lemons

Now we can do more grouping based on another feature, its size:

- **Red color and large size**: Apples
- **Red color and small size**: Cherries
- **Yellow color and large size**: Banana
- **Yellow color and small size**: Lemons

The following diagram shows a representation of clustering:



This way, by looking at more features, we can also do more clustering. Here, we don't have any training data and a variable to be predicted, unlike in supervised learning. Our only task is to learn more about the features and cluster the records based on inputs.

The following are some of the algorithms commonly used for clustering:

- K-means clustering
- Hierarchical clustering
- Hidden Markov models

Association

Association problems are more about learning, and making recommendations by defining association rules. Association rules could, for example, refer to the assumption that people who bought an iPhone are more likely to buy an iPhone case.

These days, many retail companies use these algorithms to make personalized recommendations. For instance, on www.amazon.com, if I tend to purchase product *X* and then Amazon recommends me product *Y* as well, there must be some association between the two.

Some of the algorithms based on these principles are as follows:

- Apriori algorithm
- Eclat algorithm
- FDP growth algorithm

Semi-supervised learning

Semi-supervised learning is a sub-class of supervised learning that considers unlabeled data for training. Generally, while training, it has a good amount of unlabeled data and only a very small amount of labeled data. Many researchers and machine learning practitioners have found that, when labeled data is used in conjunction with unlabeled data, the results are likely to be more accurate.

> **Note**
>
> More details on semi-supervised learning can be found at https://en.wikipedia.org/wiki/Semi-supervised_learning.

FlinkML

FlinkML is a library of sets of algorithms supported by Flink that can be used to solve real-life use cases. The algorithms are built so that they can use the distributed computing power of Flink and make predictions or do clustering and so on with ease. Right now, there are only a few sets of algorithms supported, but the list is growing.

FlinkML is being built with the focus on ML developers needing to write minimal glue code. Glue code is code that helps bind various components together. Another goal of FlinkML is to keep the use of algorithms simple.

Flink exploits in-memory data streaming and executes iterative data processing natively. FlinkML allows data scientists to test their models locally, with a subset of data, and then execute them in cluster mode on bigger data.

FlinkML is inspired by scikit-learn and Spark's MLlib, which allows you to define data pipelines cleanly and solve machine learning problems in a distributed manner.

The following is the road map Flink's development team is aiming to build:

- Pipelines of transformers and learners
- Data pre-processing:
  - Feature scaling
  - Polynomial feature base mapper
  - Feature hashing
  - Feature extraction for text
  - Dimensionality reduction
- Model selection and performance evaluation:
  - Model evaluation using a variety of scoring functions
  - Cross-validation for model selection and evaluation
  - Hyper-parameter optimization
- Supervised learning:
  - Optimization framework
  - Stochastic Gradient Descent
  - L-BFGS
  - Generalized Linear Models
  - Multiple linear regression
  - LASSO, Ridge regression
  - Multi-class Logistic regression
  - Random forests
  - Support Vector Machines
  - Decision trees
- Unsupervised learning:
  - Clustering
  - K-means clustering
  - Principal Components Analysis
- Recommendation:
  - ALS

- Text analytics:
  - LDA
- Statistical estimation tools
- Distributed linear algebra
- Streaming ML

The algorithms highlighted are already part of the existing Flink source code. In the following section, we will look at how we can use those in practice.

Supported algorithms

To get started with FlinkML, we first need to add the following Maven dependency:

```
<!-- https://mvnrepository.com/artifact/org.apache.flink/flink-
ml_2.11 -->
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-ml_2.11</artifactId>
    <version>1.1.4</version>
</dependency>
```

Now let's try to understand the supported algorithms and how to use those.

Supervised learning

Flink supports three algorithms in the supervised-learning category. They are as follows:

- Support Vector Machine (SVM)
- Multiple linear regression
- Optimization framework

Let's get started learning about them one at a time.

Support Vector Machine

Support Vector Machines (**SVMs**) are supervised learning models, which analyze the data solving classification and regression problems. It helps classify objects into one category or another. It is non-probabilistic linear classification. There are various examples in which SVM can be used, such as the following:

- Regular data classification problems
- Text and hypertext classification problems
- Image classification problems
- Biological and other science problems

Flink supports SVM based on a soft-margin using a communication-efficient distributed dual-coordinate ascent algorithm.

Details on this algorithm are available at https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/libs/ml/svm.html#description.

Flink uses **Stochastic Dual Coordinate Ascent** (**SDCA**) to solve the minimization problem. To make this algorithm efficient in a distributed environment, Flink uses the CoCoA algorithm, which calculates the SDCA on a local data block and then merges it into global state.

> **Note**
>
> The implementation of this algorithm is based on the following paper: https://arxiv.org/pdf/1409.1458v2.pdf.

Now let's look at how we can solve a real-life problem using this algorithm. We will take the example of the Iris dataset (https://en.wikipedia.org/wiki/Iris_flower_data_set), consisting of four attributes which decide the species of Iris. The following is some sample data:

Sepal length	Sepal width	Petal length	Petal width	Species
5.1	3.5	1.4	0.2	1

5.6	2.9	3.6	1.3	2
5.8	2.7	5.1	1.9	3

Here, it is important to use categories in number format to be used as input to SVM:

Species code	**Species name**
1	Iris Setosa
2	Iris Versicolor
3	Iris Virginica

One more thing we need to do before using data for Flink's SVM algorithm is to convert this CSV data into LibSVM data.

> **Note**
> LibSVM data is a special format used for specifying SVM datasets. More information on LibSVM is available at https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/.

To convert CSV data to LibSVM data, we will use some open-source Python code available at https://github.com/zygmuntz/phraug/blob/master/csv2libsvm.py.
To convert CSV to LibSVM, we need to execute the following command:

```
    csv2libsvm.py <input file> <output file> [<label index = 0>]
[<skip
    headers = 0>]
```

Now let's get started with writing the program:

```
package com.demo.chapter06

import org.apache.flink.api.scala._
import org.apache.flink.ml.math.Vector
import org.apache.flink.ml.common.LabeledVector
import org.apache.flink.ml.classification.SVM
import org.apache.flink.ml.RichExecutionEnvironment

object MySVMApp {
  def main(args: Array[String]) {
    // set up the execution environment
    val pathToTrainingFile: String = "iris-train.txt"
    val pathToTestingFile: String = "iris-train.txt"
    val env = ExecutionEnvironment.getExecutionEnvironment

    // Read the training dataset, from a LibSVM formatted file
    val trainingDS: DataSet[LabeledVector] =
    env.readLibSVM(pathToTrainingFile)

    // Create the SVM learner
    val svm = SVM()
      .setBlocks(10)

    // Learn the SVM model
    svm.fit(trainingDS)

    // Read the testing dataset
    val testingDS: DataSet[Vector] =
    env.readLibSVM(pathToTestingFile).map(_.vector)
```

```
        // Calculate the predictions for the testing dataset
        val predictionDS: DataSet[(Vector, Double)] =
        svm.predict(testingDS)
        predictionDS.writeAsText("out")

        env.execute("Flink SVM App")
      }
    }
```

So, now we are all set to run the program, and you will the see the predicted output in the output folder.
The following is the code:

```
(SparseVector((0,5.1), (1,3.5), (2,1.4), (3,0.2)),1.0)
(SparseVector((0,4.9), (1,3.0), (2,1.4), (3,0.2)),1.0)
(SparseVector((0,4.7), (1,3.2), (2,1.3), (3,0.2)),1.0)
(SparseVector((0,4.6), (1,3.1), (2,1.5), (3,0.2)),1.0)
(SparseVector((0,5.0), (1,3.6), (2,1.4), (3,0.2)),1.0)
(SparseVector((0,5.4), (1,3.9), (2,1.7), (3,0.4)),1.0)
(SparseVector((0,4.6), (1,3.4), (2,1.4), (3,0.3)),1.0)
(SparseVector((0,5.0), (1,3.4), (2,1.5), (3,0.2)),1.0)
(SparseVector((0,4.4), (1,2.9), (2,1.4), (3,0.2)),1.0)
(SparseVector((0,4.9), (1,3.1), (2,1.5), (3,0.1)),1.0)
(SparseVector((0,5.4), (1,3.7), (2,1.5), (3,0.2)),1.0)
(SparseVector((0,4.8), (1,3.4), (2,1.6), (3,0.2)),1.0)
(SparseVector((0,4.8), (1,3.0), (2,1.4), (3,0.1)),1.0)
```

We can also fine-tune the results by setting various parameters:

Parameter	Description
`Blocks`	Sets the number of blocks into which the input data will be split. It is ideal to set this number equal to the parallelism you want to achieve. On each block, local stochastic dual-coordinate ascent is performed. The default value is `None`.
`Iterations`	Sets the number of iterations of the outer loop method, for example, the amount of time the SDCA method should applied on blocked data. The default value is `10`.
`LocalIterations`	Defines the maximum number of SDCA iterations that need to be executed locally. The default value is `10`.
`Regularization`	Sets the regularization constant of the algorithm. The higher you set the value, the smaller the 2 norm of the weighted vector be. The default value is `1`.
`StepSize`	Defines the initial step size for the updates of weight vector. This value needs to be set up in case the algorithm becomes unstable. The default value is `1.0`.
`ThresholdValue`	Defines the limiting value for the decision function. The default value is `0.0`.
`OutputDecisionFunction`	Setting this to true will return the hyperplane distance for each

	example. Setting it to false will return the binary label.
`Seed`	Sets the random long integer. This will be used to initialize the random number generator.

Multiple Linear Regression

Multiple Linear Regression (**MLR**) is an extension of simple linear regression where more than one independent variable (*X*) is used to determine the single independent variable (*Y*). The predicted value is a linear transformation of input variables such that the sum of squared deviations of the observed and predicted is minimum.
MLR tries to model the relationship between multiple explanatory variables and response variables by fitting a linear equation.

> **Note**
> A more detailed explanation of MLR can be found on this link http://www.stat.yale.edu/Courses/1997-98/101/linmult.htm.

Let's try solving the same classification problem of Iris dataset using MLR now. First we need the training dataset on which we can train our mode.
Here we will be using the same data files we used in previous section on SVM. So now we have `iris-train.txt` and `iris-test.txt` which are converted into LibSVM format.
The following code snippet shows how MLR can be used:

```
package com.demo.flink.ml

import org.apache.flink.api.scala._
import org.apache.flink.ml._
import org.apache.flink.ml.common.LabeledVector
import org.apache.flink.ml.math.DenseVector
import org.apache.flink.ml.math.Vector
import org.apache.flink.ml.preprocessing.Splitter
import org.apache.flink.ml.regression.MultipleLinearRegression

object MLRJob {
  def main(args: Array[String]) {
    // set up the execution environment
    val env = ExecutionEnvironment.getExecutionEnvironment
    val trainingDataset = MLUtils.readLibSVM(env, "iris-train.txt")
    val testingDataset = MLUtils.readLibSVM(env, "iris-
test.txt").map {
    lv => lv.vector }
    val mlr = MultipleLinearRegression()
      .setStepsize(1.0)
      .setIterations(5)
      .setConvergenceThreshold(0.001)

    mlr.fit(trainingDataset)

    // The fitted model can now be used to make predictions
    val predictions = mlr.predict(testingDataset)

    predictions.print()

  }
```

```
}
```

The complete code and the data files are available for download on https://github.com/deshpandetanmay/mastering-flink/tree/master/chapter06. We can also fine-tune the results by setting various parameters:

Parameter	Description
`Iterations`	Sets the maximum number of iterations. The default value is `10`.
`Stepsize`	The step size of the gradient descent method. This value controls how far the gradient descent method can move in the opposite direction. Tuning this parameter is very important to get better results. The default value is `0.1`.
`Convergencethreshold`	The threshold for the relative change of the sum of squared residuals until the iteration is stopped. The default value is `None`.
`Learningratemethod`	`Learningratemethod` is used to calculate the learning rate of each iteration.

Optimization framework

Optimization framework in Flink is a developer-friendly package which can be used to solve optimization problems. This is not a specific algorithm to solve exact problems, but it is the basis of every machine learning problem.

Generally, it is about finding the model, with a set of parameters, with a minimization function. FlinkML supports **Stochastic Gradient Descent** (**SGD**), with the following types of regularizations:

Regularization function	Class name
L1 regularization	`GradientDescentL1`
L2 regularization	`GradientDescentL2`
No regularization	`SimpleGradient`

The following code snippet shows how to use SGD using FlinkML:

```
// Create SGD solver
val sgd = GradientDescentL1()
  .setLossFunction(SquaredLoss())
  .setRegularizationConstant(0.2)
  .setIterations(100)
  .setLearningRate(0.01)
  .setLearningRateMethod(LearningRateMethod.Xu(-0.75))


// Obtain data
val trainingDS: DataSet[LabeledVector] = ...

// Optimize the weights, according to the provided data
val weightDS = sgd.optimize(trainingDS)
```

We can also use parameters to fine-tune the algorithm:

Parameter	Description

`LossFunction`	Flink supports the following loss functions: • Squared Loss • Hinged Loss • Logistic Loss • The default is `None`
`RegularizationConstant`	The weight of regularization to be applied. The default value is `0.1`.
`Iterations`	The maximum number of iterations to be performed. The default is `10`.
`ConvergenceThreshold`	The threshold for relative change of the sum of squared residuals until the iteration is stopped. The default value is `None`.
`LearningRateMethod`	This method is used to calculate the learning rate of each iteration.
`LearningRate`	This is the initial learning rate for the gradient descent method.
`Decay`	The default value is `0.0`.

Recommendations

Recommendation engines are one of the most interesting and heavily used machine learning techniques to provide user-based and item-based recommendations. E-commerce companies such as Amazon use recommendation engines to personalize recommendations based on the purchasing patterns and review ratings of its customers.
Flink also supports ALS-based recommendations. Let's look at ALS in more detail.

Alternating Least Squares

The **Alternating Least Squares** (**ALS**) algorithm factorizes a given matrix, *R*, into two factors, *U* and *V*, such that $R \approx U^T V$
In order to better understand the application of this algorithm, let's assume that we have a dataset which contains the rating, *r*, provided by user *u* for book *b*.
Here is a sample data format (`user_id`, `book_id`, `rating`):

```
1  10 1
1  11 2
1  12 5
1  13 5
1  14 5
1  15 4
1  16 5
1  17 1
1  18 5
2  10 1
2  11 2
2  15 5
2  16 4.5
2  17 1
2  18 5
3  11 2.5
```

```
3  12 4.5
3  13 4
3  14 3
3  15 3.5
3  16 4.5
3  17 4
3  18 5
4  10 5
4  11 5
4  12 5
4  13 0
4  14 2
4  15 3
4  16 1
4  17 4
4  18 1
```

Now we can feed this information to the ALS algorithm and start getting recommendations from it. The following is a code snippet for using ALS:

```
package com.demo.chapter06

import org.apache.flink.api.scala._
import org.apache.flink.ml.recommendation._
import org.apache.flink.ml.common.ParameterMap

object MyALSApp {
  def main(args: Array[String]): Unit = {

    val env = ExecutionEnvironment.getExecutionEnvironment
    val inputDS: DataSet[(Int, Int, Double)] =
env.readCsvFile[(Int,
    Int, Double)]("input.csv")

    // Setup the ALS learner
    val als = ALS()
      .setIterations(10)
      .setNumFactors(10)
      .setBlocks(100)
      .setTemporaryPath("tmp")

    // Set the other parameters via a parameter map
    val parameters = ParameterMap()
      .add(ALS.Lambda, 0.9)
      .add(ALS.Seed, 42L)

    // Calculate the factorization
    als.fit(inputDS, parameters)

    // Read the testing dataset from a csv file
    val testingDS: DataSet[(Int, Int)] = env.readCsvFile[(Int,
Int)]
    ("test-data.csv")

    // Calculate the ratings according to the matrix factorization
    val predictedRatings = als.predict(testingDS)

    predictedRatings.writeAsCsv("output")
```

```
        env.execute("Flink Recommendation App")
      }
    }
```

Once you execute the application, you'll get the results as recommendations. Like with other algorithms, you can fine-tune the parameters to get better results:

Parameter	Description
`NumFactors`	The number of latent factors to use for the underlying model. The default value is `10`.
`Lambda`	This is a regularization factor; we can tune this parameter for better results. The default is `1`.
`Iterations`	The maximum number of iterations to be performed. The default is `10`.
`Blocks`	The number of blocks in which user and item matrix are grouped. The fewer the blocks, the less data is sent redundantly. The default value is `None`.
`Seed`	The seed value to initiate the item matrix generator. The default is `0`.
`TemporaryPath`	This is a path to be used for storing intermediate results.

Unsupervised learning

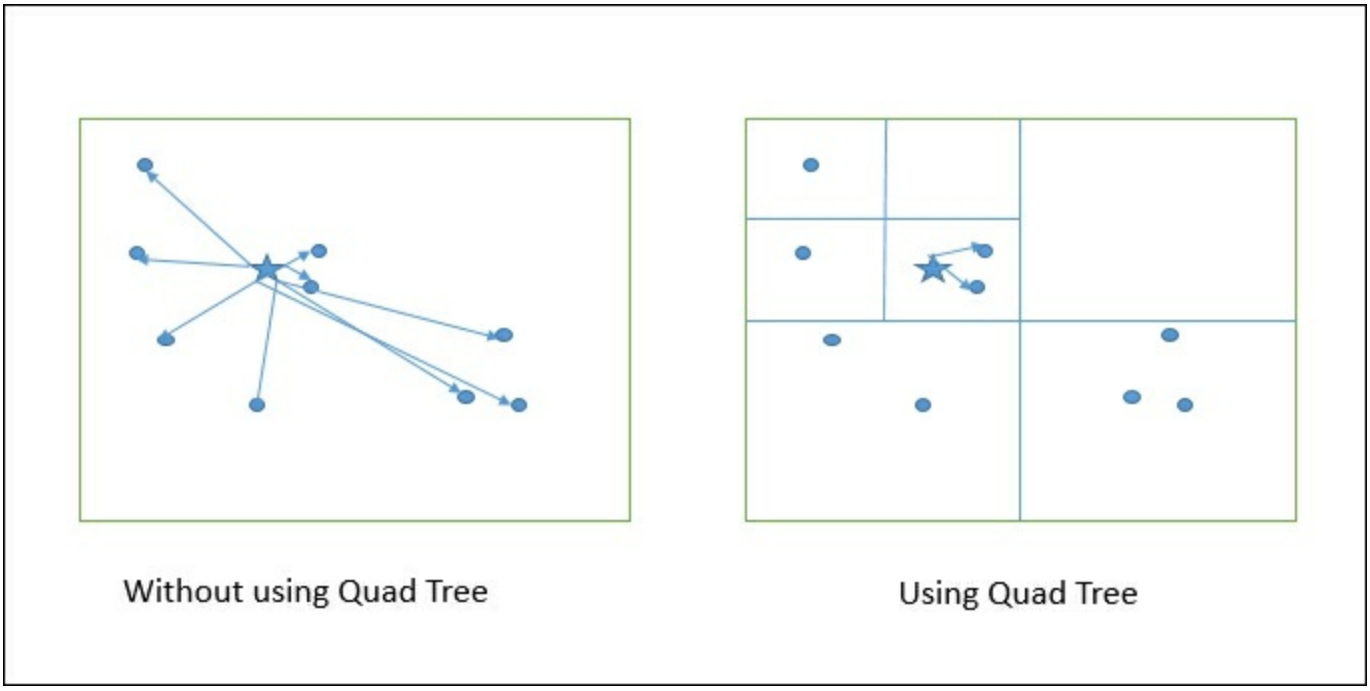
Now let's try to understand what FinkML offers for unsupervised learning. For now, it supports only one algorithm, called the k Nearest Neighbor join algorithm.

k Nearest Neighbour join

The **k Nearest Neighbor** (**kNN**) algorithm is designed to find the k Nearest Neighbour from a dataset for every object in another dataset. It is one of the most widely used solutions in many data-mining algorithms. The kNN is an expensive operation, as it is a combination of finding the k nearest neighbor and performing a join. Considering the volume of the data, it is very difficult to perform this operation on a centralized single machine, hence it is always good to have solutions that can work on distributed architecture. The FlinkML algorithm provides kNN on a distributed environment.

> **Note**
> A research paper describing the implementation of kNN on a distributed environment can be found here: https://arxiv.org/pdf/1207.0141v1.pdf.

Here, the idea is to compute the distance between every training and testing point and then find the nearest points for a given point. Computing the distance between each point is a time-consuming activity, which is eased out in Flink by implementing quad trees.
Using quad trees reduces the computation by partitioning the dataset. This reduces the computation to the subset of data only. The following diagram shows computation with and without quad trees:



You can find a detailed discussion on using quad trees to calculate the nearest neighbors here: http://danielblazevski.github.io/assets/player/KeynoteDHTMLPlayer.html.

It's not always the case that quad trees will perform better. If the data is spatial, the quad trees might be the worst choice. But as a developer, we don't need to worry about it as FlinkML takes care of deciding whether to use quad tree or not based on the data available.

The following code snippet shows how to use kNN join in FlinkML:

```
import
org.apache.flink.api.common.operators.base.CrossOperatorBase.CrossH
int
import org.apache.flink.api.scala._
import org.apache.flink.ml.nn.KNN
import org.apache.flink.ml.math.Vector
import
org.apache.flink.ml.metrics.distances.SquaredEuclideanDistanceMetri
c

val env = ExecutionEnvironment.getExecutionEnvironment

// prepare data
val trainingSet: DataSet[Vector] = ...
val testingSet: DataSet[Vector] = ...

val knn = KNN()
  .setK(3)
  .setBlocks(10)
  .setDistanceMetric(SquaredEuclideanDistanceMetric())
  .setUseQuadTree(false)
  .setSizeHint(CrossHint.SECOND_IS_SMALL)

// run knn join
knn.fit(trainingSet)
val result = knn.predict(testingSet).collect()
```

The following are some parameters we can use to fine-tune the results:

Parameter	Description

`K`	The number of nearest neighbours to search for. The default is `5`.
`DistanceMetric`	Sets the distance metric to be used to calculate the distance between two points. By default, Euclidian Distance Metric is used.
`Blocks`	The number of blocks into which the input data should be split. It is ideal to set this number equal to the degree of parallelism.
`UseQuadTree`	Sets whether to use quad tree for processing or not. The default value is `None`. If nothing is specified, the algorithm decides on its own.

Utilities

FlinkML supports various extensible utilities, which can be handy while doing data analysis and predictions. One such utility is distance metrics. Flink supports a set of distance metrics which can be used. The following link shows Flink-supported distance metrics: https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/libs/ml/distance_metrics.html.
If any of the previously mentioned algorithms do not satisfy your needs, you can think about writing your own custom distance algorithm. The following code snippet shows how to do so:

```
class MyDistance extends DistanceMetric {
  override def distance(a: Vector, b: Vector) = ... // your
implementation
}

object MyDistance {
  def apply() = new MyDistance()
}

val myMetric = MyDistance()
```

A good application of using distance metrics is the kNN join algorithm, where you can set the distance metric to use.
Another important utility is `Splitter`, which can be used for cross validation. In some cases, we may not have a test dataset to validate our results. In such cases, we can split the training dataset using `Splitter`.
The following is an example:

```
// A Simple Train-Test-Split
val dataTrainTest: TrainTestDataSet = Splitter.trainTestSplit(data,
0.6, true)
```

In the preceding example, we are splitting the training dataset into portions of 60% and 40% of the actual data.
There is another method to fetch better results, called `TrainTestHoldout` split. Here, we use some portion of the data for training, some for testing, and another set for final result validations. The following snippet shows how to do it:

```
// Create a train test holdout DataSet
val dataTrainTestHO: trainTestHoldoutDataSet =
Splitter.trainTestHoldoutSplit(data, Array(6.0, 3.0, 1.0))
```

We can use another strategy, called K fold splits. In this method, the training set is split into

k equal size folds. Here, an algorithm is created for each fold and then validated against its testing set. The following code shows how to do k-fold splits:

```
// Create an Array of K TrainTestDataSets
val dataKFolded: Array[TrainTestDataSet] =
Splitter.kFoldSplit(data, 10)
```

We can also use **Multi Random Splits**; here we can specify how many datasets to create and of what portion of the original:

```
// create an array of 5 datasets of 1 of 50%, and 5 of 10% each
val dataMultiRandom: Array[DataSet[T]] =
Splitter.multiRandomSplit(data, Array(0.5, 0.1, 0.1, 0.1, 0.1))
```

Data pre processing and pipelines

Flink supports Python scikit-learn style pipeline. A pipeline in FlinkML is feature to chain multiple transformers and predictors in one go. In general, many data scientists would like to see and build the flow of machine learning application with ease. Flink allows them to do so using the concept of pipelines.

In general, there are three building blocks of ML pipelines:

- **Estimator:** Estimator performs the actual training of a model using a `fit` method. For example, finding correct weights in a linear regression model.
- **Transformer:** Transformer as the name suggests have a `transform` method which can help in scaling the input.
- **Predictor:** Predictors have the `predict` method which applies the algorithm for generating predictions, for example, SVM or MLR.

A pipeline is a chain of estimator, transformers and predictor. The predictor is the end of a pipeline and nothing can be chained after that.

Flink supports various data pre-processing tools which would help us advance the results. Let's start understanding the details.

Polynomial features

The polynomial feature is a transformer which maps a vector into the polynomial feature space of degree *d*. Polynomial feature helps in solving classification problems by changing the graph of the function. Let's try to understand this by an example:

- Consider a linear formula: *F(x,y) = 1*x + 2*y;*
- Imagine we have two observations:
  - *x=12* and *y=2*
  - *x=5* and *y =5.5*

In both cases, we get *f() = 16*. If these observations belong to two different classes then we cannot differentiate between the two. Now if we add one more feature called *z* which is combination of previous two features *z = x+y*.

So now *f(x,y,z) = 1*x + 2*y + 3*z*

Now the same observations would be

- *(1*12)+ (2*2) + (3*24) = 88*
- *(1*5)+ (2*5.5) + (3*27.5) = 98.5*

This way adding a new feature using existing features can help us get better results. Flink

polynomial features allows us to do the same with pre-build functions.
In order to use polynomial features in Flink, we have the following code:

```
val polyFeatures = PolynomialFeatures()
      .setDegree(3)
```

Standard scaler

Standard scaler helps scale the input data using the user-specified mean and variance. If the user does not specify any values then default mean is `0` and standard deviation would be `1`. Standard scaler is a transformer which has `fit` and `transform` method.
First we need to define the values for mean and standard deviation as shown in the following code snippet:

```
val scaler = StandardScaler()
    .setMean(10.0)
    .setStd(2.0)
```

Next we need to let it learn about mean and standard deviation of training dataset as shown in the following code snippet:

```
scaler.fit(trainingDataset)
```

And finally we scale the provided data using the user-defined mean and standard deviation as shown in the following code snippet:

```
val scaledDS = scaler.transform(trainingDataset)
```

Now we can use this scaled input data to do further transformation and analysis.

MinMax scaler

MinMax scaler is like standard scaler but the only difference is it makes sure that scaling of each feature lies between user-defined `min` and `max` values.
The following code snippet shows how to use this:

```
val minMaxscaler = MinMaxScaler()
.setMin(1.0)
.setMax(3.0)
minMaxscaler.fit(trainingDataset)
val scaledDS = minMaxscaler.transform(trainingDataset)
```

Thus, we can use these data pre-processing operations to enhance the results. These can also be combined in pipelines to create the workflow.
The following code snippet shows how to use these data pre-processing operations in pipeline:

```
// Create pipeline PolynomialFeatures -> MultipleLinearRegression
val pipeline = polyFeatures.chainPredictor(mlr)
// train the model
pipeline.fit(scaledDS)
// The fitted model can now be used to make predictions
val predictions = pipeline.predict(testingDataset)
predictions.print()
```

The complete code is available on GitHub at https://github.com/deshpandetanmay/mastering-flink/tree/master/chapter06.

Summary

In this chapter, we learned about the different types of machine learning algorithm. We looked at various supervised and unsupervised algorithms, and their respective examples. We also looked at various utilities provided by FlinkML, which can be very handy during data analysis. Later we looked at data pre-processing operations and how to use them in pipelines.

In the following chapter, we will look at the graph-processing capabilities of Flink.