Chapter 5. Learning and Prediction

In this chapter, we'll learn what our data means and how it drives our decision processes. Learning about our data gives us knowledge, and knowledge enables us to make reasonable guesses about what to expect in the future. This is the reason for the existence of data science: learning enough about the data so we can make predictions on newly arriving data. This can be as simple as categorizing data into groups or clusters. It can span a much broader set of processes that culminate (ultimately) in the path to artificial intelligence. Learning is divided into two major categories: unsupervised and supervised.

In general, we think of data as having variates **X** and responses **Y**, and our goal is to build a model using **X** so that we can predict what happens when we put in a new **X**. If we have the **Y**, we can "supervise" the building of the model. In many cases, we have only the variates **X**. The model will then have to be built in an unsupervised manner. Typical unsupervised methods include clustering, whereas supervised learning may include any of the regression methods (e.g., linear regression) or classification methods such as naive Bayes, logistic, or deep neural net classifiers. Many other methods and permutations of those methods exist, and covering them all would be impossible. Instead, here we dive into a few of the most useful ones.

Learning Algorithms

A few learning algorithms are prevalent in a large variety of techniques. In particular, we often use an iterative learning process to repeatedly optimize or update the model parameters we are searching for. Several methods are available for optimizing the parameters, and we cover the gradient descent method here.

Iterative Learning Procedure

One standard way to learn a model is to loop over a prediction state and update the state. Regression, clustering and expectation-maximization (EM) algorithms all benefit from similar forms of an iterative learning procedure. Our strategy here is to create a class that contains all the boilerplate iterative machinery, and then allow subclasses to define the explicit form of the prediction and parameter update methods.

```java
public class IterativeLearningProcess {

    private boolean isConverged;
    private int numIterations;
    private int maxIterations;
    private double loss;
    private double tolerance;
    private int batchSize; // if == 0 then uses ALL data
    private LossFunction lossFunction;

    public IterativeLearningProcess(LossFunction lossFunction) {
        this.lossFunction = lossFunction;
        loss = 0;
        isConverged = false;
```

```
            numIterations = 0;
            maxIterations = 200;
            tolerance = 10E-6;
            batchSize = 100;
        }

        public void learn(RealMatrix input, RealMatrix target) {
            double priorLoss = tolerance;
            numIterations = 0;
            loss = 0;
            isConverged = false;
            Batch batch = new Batch(input, target);
            RealMatrix inputBatch;
            RealMatrix targetBatch;
            while(numIterations < maxIterations && !isConverged) {
                if(batchSize > 0 && batchSize < input.getRowDimension()) {
                    batch.calcNextBatch(batchSize);
                    inputBatch = batch.getInputBatch();
                    targetBatch = batch.getTargetBatch();
                } else {
                    inputBatch = input;
                    targetBatch = target;
                }
                RealMatrix outputBatch = predict(inputBatch);
                loss = lossFunction.getMeanLoss(outputBatch, targetBatch);
                if(Math.abs(priorLoss - loss) < tolerance) {
                    isConverged = true;
                } else {
                    update(inputBatch, targetBatch, outputBatch);
                    priorLoss = loss;
                }
                numIterations++;
            }
        }

        public RealMatrix predict(RealMatrix input) {
            throw new UnsupportedOperationException("Implement the predict method!");
        }

        public void update(RealMatrix input, RealMatrix target, RealMatrix output) {
            throw new UnsupportedOperationException("Implement the update method!");
        }

    }
```

Gradient Descent Optimizer

One way to learn parameters is via a gradient descent (an iterative first-order optimization algorithm). This optimizes the parameters by incrementally updating them with corrective learning (the error is used). The term *stochastic* means that we add one point at a time, as opposed to using the whole batch of data at once. In practice, it helps to use a mini-batch of about 100 points at a time, chosen at random in each step of the iterative learning process. The general idea is to minimize a loss function such that parameter updates are given by the following:

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

The parameter update is related to the gradient of an objective function $f(\theta)$ such that

$$\Delta\theta \propto \nabla f(\theta)$$

For deep networks, we will need to back-propagate this error through the network. We cover this in detail in "Deep Networks".

For the purposes of this chapter, we can define an interface that returns a parameter update, given a particular gradient. Method signatures for both matrix and vector forms are included:

```
public interface Optimizer {
    RealMatrix getWeightUpdate(RealMatrix weightGradient);
    RealVector getBiasUpdate(RealVector biasGradient);
}
```

The most common case of gradient descent is to subtract the scaled gradient from the existing parameter such that

$$\Delta\theta_t = -\eta\nabla f(\theta)_t$$

The update rule is as follows:

$$\theta_{t+1} = \theta_t - \eta\nabla f(\theta)_t$$

The most common type of stochastic gradient descent (SGD) is adding the update to the current parameters by using a learning rate:

```
public class GradientDescent implements Optimizer {

    private double learningRate;

    public GradientDescent(double learningRate) {
        this.learningRate = learningRate;
    }

    @Override
    public RealMatrix getWeightUpdate(RealMatrix weightGradient) {
        return weightGradient.scalarMultiply(-1.0 * learningRate);
    }

    @Override
    public RealVector getBiasUpdate(RealVector biasGradient) {
        return biasGradient.mapMultiply(-1.0 * learningRate);
    }
}
```

One common extension to this optimizer is the inclusion of momentum, which slows the process as the optimum is reached, avoiding an overshoot of the correct parameters:

$$\Delta\theta_t = \rho\Delta\theta_{t-1} - \eta\nabla f(\theta)_t$$

The update rule is as follows:

$$\theta_{t+1} = \theta_t + \rho\Delta\theta_{t-1} - \eta\nabla f(\theta)_t$$

We see that adding momentum is easily accomplished by extending the `GradientDescent` class, making provisions for storing the most recent update to the weights and bias for calculation of the next update. Note that the first time around, no prior updates will be stored yet, so a new set is created (and initialized to zero):

```
public class GradientDescentMomentum extends GradientDescent {

    private final double momentum;
    private RealMatrix priorWeightUpdate;
    private RealVector priorBiasUpdate;

    public GradientDescentMomentum(double learningRate, double momentum) {
        super(learningRate);
        this.momentum = momentum;
        priorWeightUpdate = null;
        priorBiasUpdate = null;
    }

    @Override
    public RealMatrix getWeightUpdate(RealMatrix weightGradient) {
        // creates matrix of zeros same size as gradients if
        // one does not already exist
        if(priorWeightUpdate == null) {
            priorWeightUpdate =
                new BlockRealMatrix(weightGradient.getRowDimension(),
                                    weightGradient.getColumnDimension());
        }
        RealMatrix update = priorWeightUpdate
                            .scalarMultiply(momentum)
                            .subtract(super.getWeightUpdate(weightGradient));
        priorWeightUpdate = update;
        return update;
    }

    @Override
    public RealVector getBiasUpdate(RealVector biasGradient) {
        if(priorBiasUpdate == null) {
            priorBiasUpdate = new ArrayRealVector(biasGradient.getDimension());
        }
        RealVector update = priorBiasUpdate
                            .mapMultiply(momentum)
                            .subtract(super.getBiasUpdate(biasGradient));
        priorBiasUpdate = update;
        return update;
    }
}
```

This is an ongoing and active field. By using this methodology, it is easy to extend capabilities by using ADAM or ADADELTA algorithms, for example.

Evaluating Learning Processes

Iterative processes can operate indefinitely. We always designate a maximum number of iterations we will allow so that any process cannot just run away and compute forever. Typically, this is on the order of 10^3 to 10^6 iterations, but there's no rule. There is a way to stop the iterative process early if a certain criteria has been met. We call this *convergence*, and the idea is that our process has converged on an answer that appears to be a stable point in the computation (e.g., the free parameters are no longer changing in a large enough increment to warrant the continuation of the process). Of course, there is more than one way to do this. Although certain learning techniques lend themselves to specific convergence criteria, there is no universal method.

Minimizing a Loss Function

A *loss function* designates the loss between predicted and target outputs. It is also known as a *cost function* or *error term*. Given a singular input vector **x**, output vector **y**, and prediction vector $\hat{\mathbf{y}}$, the loss of the sample is denoted with $\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})$. The form of the loss function depends on the underlying statistical distribution of the output data. In most cases, the loss over *p*-dimensional output and prediction is the sum of the scalar losses per dimension:

$$\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = \sum_{p} \mathcal{L}(y, \hat{y})$$

Because we often deal with batches of data, we then calculate the mean loss $\langle \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) \rangle$ over the whole batch. When we talk about minimizing a loss function, we are minimizing the mean loss over the batch of data that was input into the learning algorithm. In many cases, we can use the gradient of the loss $\nabla \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})$ to apply corrective learning. Here the gradient of the loss with respect to the predicted value $\frac{\partial \mathcal{L}}{\partial \hat{y}_i}$ can usually be computed with ease. The idea is then to return a loss gradient that is the same shape as its input.

WARNING

In some texts, the output is denoted as t (for *truth* or *target*), and the prediction is denoted as y. In this text, we denote the output as y and prediction as $\hat{y}$. Note that y has different meanings in these two cases.

Many forms are dependent on the type of variables (continuous or discrete or both) and the underlying statistical distribution. However, a common theme makes using an interface ideal. A reason for leaving implementation up to a specific class is that it takes advantage of optimized algorithms for linear algebra routines.

```java
public interface LossFunction {
    public double getSampleLoss(double predicted, double target);
    public double getSampleLoss(RealVector predicted, RealVector target);
    public double getMeanLoss(RealMatrix predicted, RealMatrix target);
    public double getSampleLossGradient(double predicted, double target);
    public RealVector getSampleLossGradient(RealVector predicted,
                                            RealVector target);
    public RealMatrix getLossGradient(RealMatrix predicted, RealMatrix target);
}
```

LINEAR LOSS

Also known as the *absolute loss*, the *linear loss* is the absolute difference between the output and the prediction:

$$\mathcal{L}(y, \hat{y}) = | \hat{y} - y |$$

The gradient is misleading because of the absolute-value signs, which cannot be ignored:

$$\frac{\partial \mathcal{L}(y, \hat{y})}{\partial \hat{y}} = \frac{\hat{y} - y}{| \hat{y} - y |}$$

The gradient is not defined at $\hat{y} - y = 0$ because $\mathcal{L}$ has a discontinuity there. However, we can programmatically designate the gradient function to set its value to 0 when the gradient is zero to avoid a 1/0 exception. In this way, the gradient function returns only a –1, 0, or 1. Ideally, we then use the mathematical function *sign(x)*, which returns only –1, 0, or 1, depending on the respective input values of x < 0, x = 0 and x > 0.

```java
public class LinearLossFunction implements LossFunction {

    @Override
    public double getSampleLoss(double predicted, double target) {
        return Math.abs(predicted - target);
    }

    @Override
    public double getSampleLoss(RealVector predicted, RealVector target) {
        return predicted.getL1Distance(target);
    }
    @Override
    public double getMeanLoss(RealMatrix predicted, RealMatrix target) {
        SummaryStatistics stats = new SummaryStatistics();
        for (int i = 0; i < predicted.getRowDimension(); i++) {
            double dist = getSampleLoss(predicted.getRowVector(i),
            target.getRowVector(i));
            stats.addValue(dist);
        }
        return stats.getMean();
    }

    @Override
    public double getSampleLossGradient(double predicted, double target) {
        return Math.signum(predicted - target); // -1, 0, 1
    }

    @Override
    public RealVector getSampleLossGradient(RealVector predicted,
        RealVector target) {
        return predicted.subtract(target).map(new Signum());
    }

    //YOUDO SparseToSignum would be nice!!! only process elements of the iterable
    @Override
    public RealMatrix getLossGradient(RealMatrix predicted, RealMatrix target) {
        RealMatrix loss = new Array2DRowRealMatrix(predicted.getRowDimension(),
        predicted.getColumnDimension());
        for (int i = 0; i < predicted.getRowDimension(); i++) {
            loss.setRowVector(i, getSampleLossGradient(predicted.getRowVector(i),
            target.getRowVector(i)));
        }
        return loss;
    }

}
```

QUADRATIC LOSS

A generalized form for computing the error of a predictive process is by minimizing a distance metric such as L1 or L2 over the entire dataset. For a particular prediction-target pair, the quadratic error is as follows:

$$\mathcal{L}(y, \hat{y}) = \frac{1}{2}(\hat{y} - y)^2$$

An element of the sample loss gradient is then as follows:

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = (\hat{y} - y)$$

An implementation of a quadratic loss function follows:

```
public class QuadraticLossFunction implements LossFunction {

    @Override
    public double getSampleLoss(double predicted, double target) {
        double diff = predicted - target;
        return 0.5 * diff * diff;
    }

    @Override
    public double getSampleLoss(RealVector predicted, RealVector target) {
        double dist = predicted.getDistance(target);
```

```
            return 0.5 * dist * dist;
        }

        @Override
        public double getMeanLoss(RealMatrix predicted, RealMatrix target) {
            SummaryStatistics stats = new SummaryStatistics();
            for (int i = 0; i < predicted.getRowDimension(); i++) {
                double dist = getSampleLoss(predicted.getRowVector(i),
                                            target.getRowVector(i));
                stats.addValue(dist);
            }
            return stats.getMean();
        }

        @Override
        public double getSampleLossGradient(double predicted, double target) {
            return predicted - target;
        }

        @Override
        public RealVector getSampleLossGradient(RealVector predicted,
            RealVector target) {
            return predicted.subtract(target);
        }

        @Override
        public RealMatrix getLossGradient(RealMatrix predicted, RealMatrix target) {
            return predicted.subtract(target);
        }
    }
```

CROSS-ENTROPY LOSS

Cross entropy is great for classification (e.g., logistics regression or neural nets). We discussed the origins of cross entropy in Chapter 3. Because cross entropy shows similarity between two samples, it can be used for measuring agreement between known and predicted values. In the case of learning algorithms, we equate p with the known value y, and q with the predicted value $\hat{y}$. We set the loss equal to the cross entropy $\mathscr{H}(p, q)$ such that $L(y, \hat{y}) = \mathscr{H}(p, q)$ where $y_{ik} = p_{ik}$ is the target (label) and $\hat{y}_{ik} = q_{ik}$ is the i-th predicted value for each class k in a K multiclass output. The cross entropy (the loss per sample) is then as follows:

$$\mathscr{L}(y, \hat{y}) = -\sum_{k}^{K} y_k \log(\hat{y}_k)$$

There are several common forms for cross entropy and its associated loss function.

Bernoulli

In the case of Bernoulli output variates, the known outputs y are binary, where the prediction probability is $\hat{y}$, giving a cross-entropy loss:

$$\mathscr{L}(y, \hat{y}) = -(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y}))$$

The sample loss gradient is then as follows:

$$\frac{\partial \mathscr{L}}{\partial \hat{y}} = \frac{\hat{y} - y}{\hat{y}(1 - \hat{y})}$$

Here is an implementation of the Bernoulli cross-entropy loss:

```
public class CrossEntropyLossFunction implements LossFunction {

    @Override
    public double getSampleLoss(double predicted, double target) {
        return -1.0 * (target * ((predicted>0)?FastMath.log(predicted):0)
            + (1.0 - target)*(predicted<1?FastMath.log(1.0-predicted):0));
    }

    @Override
    public double getSampleLoss(RealVector predicted, RealVector target) {
        double loss = 0.0;
        for (int i = 0; i < predicted.getDimension(); i++) {
            loss += getSampleLoss(predicted.getEntry(i), target.getEntry(i));
        }
        return loss;
    }

    @Override
    public double getMeanLoss(RealMatrix predicted, RealMatrix target) {
        SummaryStatistics stats = new SummaryStatistics();
        for (int i = 0; i < predicted.getRowDimension(); i++) {
            stats.addValue(getSampleLoss(predicted.getRowVector(i),
            target.getRowVector(i)));
        }
        return stats.getMean();
    }

    @Override
    public double getSampleLossGradient(double predicted, double target) {
        // NOTE this blows up if predicted = 0 or 1, which it should never be
        return (predicted - target) / (predicted * (1 - predicted));
    }

    @Override
    public RealVector getSampleLossGradient(RealVector predicted,
                                            RealVector target) {
        RealVector loss = new ArrayRealVector(predicted.getDimension());
        for (int i = 0; i < predicted.getDimension(); i++) {
            loss.setEntry(i, getSampleLossGradient(predicted.getEntry(i),
            target.getEntry(i)));
        }
        return loss;
    }

    @Override
    public RealMatrix getLossGradient(RealMatrix predicted, RealMatrix target) {
        RealMatrix loss = new Array2DRowRealMatrix(predicted.getRowDimension(),
        predicted.getColumnDimension());
        for (int i = 0; i < predicted.getRowDimension(); i++) {
            loss.setRowVector(i, getSampleLossGradient(predicted.getRowVector(i),
            target.getRowVector(i)));
        }
        return loss;
    }
}
```

This expression is most often used with the logistic output function.

Multinomial

When the output is multiclass ($k = 0,1,2 \dots K - 1$) and transformed to a set of binary outputs via one-hot-encoding, the cross entropy loss is the sum over all possible classes:

$$\mathscr{L}(y, \hat{y}) = -\sum_k y_k \log\left(\hat{y}_k\right)$$

However, in one-hot encoding, only one dimension has $y = 1$ and the rest are $y = 0$ (a sparse matrix). Therefore, the sample loss is also a sparse matrix. Ideally, we could simplify this calculation by taking that into account.

The sample loss gradient is as follows:

$$\frac{\partial \mathscr{L}}{\partial \hat{y}} = -\frac{y}{\hat{y}}$$

Because most of the loss matrix will be zeros, we need to calculate the gradient only for locations where $y = 1$. This form is used primarily with the softmax output function:

```
public class OneHotCrossEntropyLossFunction implements LossFunction {

    @Override
    public double getSampleLoss(double predicted, double target) {
        return predicted > 0 ? -1.0 * target * FastMath.log(predicted) : 0;
    }

    @Override
    public double getSampleLoss(RealVector predicted, RealVector target) {
        double sampleLoss = 0.0;
        for (int i = 0; i < predicted.getDimension(); i++) {
            sampleLoss += getSampleLoss(predicted.getEntry(i),
                                        target.getEntry(i));
        }
        return sampleLoss;
    }

    @Override
    public double getMeanLoss(RealMatrix predicted, RealMatrix target) {
        SummaryStatistics stats = new SummaryStatistics();
        for (int i = 0; i < predicted.getRowDimension(); i++) {
            stats.addValue(getSampleLoss(predicted.getRowVector(i),
            target.getRowVector(i)));
        }
        return stats.getMean();
    }

    @Override
    public double getSampleLossGradient(double predicted, double target) {
        return -1.0 * target / predicted;
    }

    @Override
    public RealVector getSampleLossGradient(RealVector predicted,
                                            RealVector target) {
        return target.ebeDivide(predicted).mapMultiplyToSelf(-1.0);
    }

    @Override
    public RealMatrix getLossGradient(RealMatrix predicted, RealMatrix target) {
        RealMatrix loss = new Array2DRowRealMatrix(predicted.getRowDimension(),
        predicted.getColumnDimension());
        for (int i = 0; i < predicted.getRowDimension(); i++) {
            loss.setRowVector(i, getSampleLossGradient(predicted.getRowVector(i),
            target.getRowVector(i)));
        }
        return loss;
    }
}
```

Two-Point

When the output is binary but takes on the values of –1 and 1 instead of 0 and 1, we can rescale for use with the Bernoulli expression with the substitutions $y^* = (y+1)/2$ and $\hat{y}^* = (\hat{y}+1)/2$:

$$\mathscr{L}(y^*, \hat{y}^*) = -\left(\left(\frac{y+1}{2}\right)\log\left(\frac{\hat{y}+1}{2}\right) + \left(\frac{1-y}{2}\right)\log\left(\frac{1-\hat{y}}{2}\right)\right)$$

The sample loss gradient is as follows:

$$\frac{\partial \mathscr{L}}{\partial \hat{y}} = \frac{\hat{y} - y}{1 - \hat{y}^2}$$

The Java code is shown here:

```
public class TwoPointCrossEntropyLossFunction implements LossFunction {

    @Override
    public double getSampleLoss(double predicted, double target) {
        // convert -1:1 to 0:1 scale
        double y = 0.5 * (predicted + 1);
        double t = 0.5 * (target + 1);
        return -1.0 * (t * ((y>0)?FastMath.log(y):0) +
                (1.0 - t)*(y<1?FastMath.log(1.0-y):0));
    }

    @Override
    public double getSampleLoss(RealVector predicted, RealVector target) {
        double loss = 0.0;
        for (int i = 0; i < predicted.getDimension(); i++) {
            loss += getSampleLoss(predicted.getEntry(i), target.getEntry(i));
        }
        return loss;
    }

    @Override
    public double getMeanLoss(RealMatrix predicted, RealMatrix target) {
        SummaryStatistics stats = new SummaryStatistics();
        for (int i = 0; i < predicted.getRowDimension(); i++) {
            stats.addValue(getSampleLoss(predicted.getRowVector(i),
            target.getRowVector(i)));
        }
        return stats.getMean();
    }

    @Override
    public double getSampleLossGradient(double predicted, double target) {
        return (predicted - target) / (1 - predicted * predicted);
    }

    @Override
    public RealVector getSampleLossGradient(RealVector predicted,
                                            RealVector target) {
        RealVector loss = new ArrayRealVector(predicted.getDimension());
        for (int i = 0; i < predicted.getDimension(); i++) {
            loss.setEntry(i, getSampleLossGradient(predicted.getEntry(i),
            target.getEntry(i)));
        }
        return loss;
    }

    @Override
    public RealMatrix getLossGradient(RealMatrix predicted, RealMatrix target) {
        RealMatrix loss = new Array2DRowRealMatrix(predicted.getRowDimension(),
        predicted.getColumnDimension());
        for (int i = 0; i < predicted.getRowDimension(); i++) {
            loss.setRowVector(i, getSampleLossGradient(predicted.getRowVector(i),
            target.getRowVector(i)));
        }
        return loss;
    }
}
```

This form of loss is compatible with a tanh activation function.

Minimizing the Sum of Variances

When data is split into more than one group, we can monitor the spread of the group from its mean position via the variance. Because variances add, we can define a metric *s* over *n* groups, where σ_i^2 is the variance of each group:

$$s = \sum_{i=1}^{n} \sigma_i^2$$

As *s* decreases, it signifies that the overall error of the procedure is also decreasing. This works great for clustering techniques, such as *k*-means, which are based on finding the mean value or center point of each cluster.

Silhouette Coefficient

In unsupervised learning techniques such as clustering, we seek to discover how closely packed each group of points is. The *silhouette coefficient* is a metric that relates the difference between the minimum distance inside any given cluster and its nearest cluster. The silhouette coefficient, *s*, is the average over all distances s_i for each sample; *a* = the mean distance between that sample and all other points in the class, and *b* = the mean distance between that sample and all the points in the next nearest cluster:

$$s_i = \frac{b_i - a_i}{max(a_i, b_i)}$$

Then the silhouette score is the mean of all the sample silhouette coefficients:

$$s = \frac{1}{n} \sum_{i}^{n} s_i$$

The silhouette score is between –1 and 1, where –1 is incorrect clustering, 1 is highly dense clustering, and 0 indicates overlapping clusters. *s* increases as clusters are dense and well separated. The goal is in monitor processes for a maximal value of *s*. Note that the silhouette coefficient is defined only for $2 <= n_{\text{labels}} <= n_{\text{samples}} - 1$. Here is the Java code:

```
public class SilhouetteCoefficient {

    List<Cluster<DoublePoint>> clusters;
    double coefficient;
    int numClusters;
    int numSamples;

    public SilhouetteCoefficient(List<Cluster<DoublePoint>> clusters) {
        this.clusters = clusters;
        calculateMeanCoefficient();
    }

    private void calculateMeanCoefficient() {
        SummaryStatistics stats = new SummaryStatistics();
        int clusterNumber = 0;
        for (Cluster<DoublePoint> cluster : clusters) {
            for (DoublePoint point : cluster.getPoints()) {
                double s = calculateCoefficientForOnePoint(point, clusterNumber);
                stats.addValue(s);
            }
            clusterNumber++;
        }
        coefficient = stats.getMean();
    }

    private double calculateCoefficientForOnePoint(DoublePoint onePoint,
    int clusterLabel) {

        /* all other points will compared to this one */
        RealVector vector = new ArrayRealVector(onePoint.getPoint());
        double a = 0;
        double b = Double.MAX_VALUE;
        int clusterNumber = 0;
        for (Cluster<DoublePoint> cluster : clusters) {
            SummaryStatistics clusterStats = new SummaryStatistics();
            for (DoublePoint otherPoint : cluster.getPoints()) {
                RealVector otherVector =
                    new ArrayRealVector(otherPoint.getPoint());
                double dist = vector.getDistance(otherVector);
                clusterStats.addValue(dist);
            }
            double avgDistance = clusterStats.getMean();
            if(clusterNumber==clusterLabel) {
                /* we have included a 0 distance of point with itself */
                /* and need to subtract it out of the mean */
                double n = new Long(clusterStats.getN()).doubleValue();
                double correction = n / (n - 1.0);
                a = correction * avgDistance;
            } else {
                b = Math.min(avgDistance, b);
            }
            clusterNumber++;
        }
        return (b-a) / Math.max(a, b);
    }
}
```

Log-Likelihood

In unsupervised learning problems for which each outcome prediction has a probability associated with it, we can utilize the log-likelihood. One particular example is the Gaussian clustering example in this chapter. For this expectation-maximization algorithm, a mixture of multivariate normal distributions are optimized to fit the data. Each data point has a probability density p_i associated with it, given the overlying model, and the log-likelihood can be computed as the mean of the log of the probabilities for each point:

$$\mathscr{L}(\mathbf{p}) = \sum_i \log\left(p_i\right)$$

We can then accumulate the average log-likelihood over all the data points $\langle L(\mathbf{p}) \rangle$. In the case of the Gaussian clustering example, we can obtain this parameter directly via the `MultivariateNormalMixtureExpectationMaximization.getLogLikelihood()` method.

Classifier Accuracy

How do we know how accurate a classifier really is? A binary classification scheme has four possible outcomes:

1. true positive (TP)—both data and prediction have the value of 1
2. true negative (TN)—both data and prediction have a value of 0
3. false positive (FP)—the data is 0 and prediction is 1
4. false negative (FN)—the data is 1 and the prediction is 0

Given a tally of each of the four possible outcomes, we can calculate, among other things, the accuracy of the classifier.

Accuracy is calculated as follows:

$$accuracy = \frac{tp + tn}{tp + tn + fp + fn}$$

Or, considering that the denominator is the total number of rows in the dataset *N*, the expression is equivalent to the following:

$$accuracy = \frac{tp + tn}{N}$$

We can then calculate the accuracy for each dimension. The average of the accuracy vector is the average accuracy of the classifier. This is also the Jaccard score.

In the special case that we are using one-hot-encoding we require only true positives and the accuracy per dimension is then as folllows:

$$accuracy = \frac{tp}{N_t}$$

N_t is the total class count (of 1s) for that dimension. The accuracy score for the classifier is then as follows:

$$accuracy = \frac{\sum tp}{N}$$

In this implementation, we have two use cases. In one case, there is one-hot encoding. In the other case, the binary, multilabel outputs are independent. In that case, we can choose a threshold (between 0 and 1) at which point to decide whether the class is 1 or 0. In the most basic sense, we can choose the threshold to be 0.5, where all probabilities below 0.5 are classified as 0, and probabilities greater than or equal to 0.5 are classified as 1. Examples of this class's use are in "Supervised Learning".

```
public class ClassifierAccuracy {

    RealMatrix predictions;
    RealMatrix targets;
    ProbabilityEncoder probabilityEncoder;
    RealVector classCount;

    public ClassifierAccuracy(RealMatrix predictions, RealMatrix targets) {
        this.predictions = predictions;
        this.targets = targets;
        probabilityEncoder = new ProbabilityEncoder();
        //tally the binary class occurrences per dimension
        classCount = new ArrayRealVector(targets.getColumnDimension());
        for (int i = 0; i < targets.getRowDimension(); i++) {
            classCount = classCount.add(targets.getRowVector(i));
        }
    }

    public RealVector getAccuracyPerDimension() {
        RealVector accuracy =
            new ArrayRealVector(predictions.getColumnDimension());
        for (int i = 0; i < predictions.getRowDimension(); i++) {
            RealVector binarized = probabilityEncoder.getOneHot(
                predictions.getRowVector(i));
            // 0*0, 0*1, 1*0 = 0 and ONLY 1*1 = 1 gives true positives
            RealVector decision = binarized.ebeMultiply(targets.getRowVector(i));
            // append TP counts to accuracy
            accuracy = accuracy.add(decision);
        }
        return accuracy.ebeDivide(classCount);
    }

    public double getAccuracy() {
        // convert accuracy_per_dim back to counts
        // then sum and divide by total rows
        return getAccuracyPerDimension().ebeMultiply(classCount).getL1Norm() /
        targets.getRowDimension();
    }

    // implements Jaccard similarity scores
    public RealVector getAccuracyPerDimension(double threshold) {
    // assumes un-correlated multi-output
        RealVector accuracy = new ArrayRealVector(targets.getColumnDimension());
        for (int i = 0; i < predictions.getRowDimension(); i++) {
            //binarize the row vector according to the threshold
            RealVector binarized = probabilityEncoder.getBinary(
            predictions.getRowVector(i), threshold);
            // 0-0 (TN) and 1-1 (TP) = 0 while 1-0 = 1 and 0-1 = -1
            RealVector decision = binarized.subtract(
            targets.getRowVector(i)).map(new Abs()).mapMultiply(-1).mapAdd(1);
            // append either TP and TN counts to accuracy
            accuracy = accuracy.add(decision);
        }
        return accuracy.mapDivide((double) predictions.getRowDimension());
        // accuracy for each dimension, given the threshold
    }

    public double getAccuracy(double threshold) {
        // mean of the accuracy vector
        return getAccuracyPerDimension(threshold).getL1Norm() /
```

```
                targets.getColumnDimension();
        }
    }
```

Unsupervised Learning

When we have only independent variables, we must discern patterns in the data without the aid of dependent variables (responses) or labels. The most common of the unsupervised techniques is clustering. The goal of all clustering is to classify each data point **X** into a series of **K** sets, $S = S_1, \ldots S_K$, where the number of sets is less than the number of points. Typically, each point X_i will belong to only one subset S_k. However, we can also designate each point X_i to belong to all sets with a probability $p(X_i) = p_1, p_2, \ldots p_K$ such that the sum = 1. Here we explore two varieties of *hard assignment*, *k*-means and DBSCAN clustering; and a *soft assignment* type, mixture of Gaussians. They all vary widely in their assumptions, algorithms, and scope. However, the result is generally the same: to classify a point **X** into one or more subsets, or clusters.

k-Means Clustering

k-means is the simplest form of clustering and uses hard assignment to find the cluster centers for a predetermined number of clusters. Initially, an integer number of *K* clusters is chosen to start with, and the centroid location μ_k of each is chosen by an algorithm (or at random). A point *x* will belong to a cluster of set S_k if its Euclidean distance (can be others, but usually L2) is closest to μ_k. Then the objective function to minimize is as follows:

$$\mathscr{L} = \sum_{k=1}^{K} \sum_{\mathbf{x} \in S_k} \| \mathbf{x} - \boldsymbol{\mu}_\mathbf{k} \|^2$$

Then we update the new centroid (the mean position of all **x** in a cluster) via this equation:

$$\boldsymbol{\mu}_k = \frac{1}{N} \sum_{\mathbf{x} \in S_k} \mathbf{x}$$

We can stop when L does not change anymore, and therefore the centroids are not changing. How do we know what number of clusters is optimal? We can keep track of the sum of all cluster variances and vary the number of clusters. When plotting the sum-of-variances versus the number of clusters, ideally the shape will look like a hockey stick, with a sharp bend in the plot indicating the ideal number of clusters at the point.

$$\sigma_K^2 = \sum_{k=1}^{K} \frac{1}{N_k - 1} \sum_{\mathbf{x} \in S_k} \| \mathbf{x} - \boldsymbol{\mu}_\mathbf{k} \|^2$$

The algorithm used by Apache Commons Math is the *k*-means++, which does a better job of picking out random starting points. The class `KMeansPlusPlusClusterer<T>` takes several arguments in its constructor, but only one is required: the number of clusters to search for. The data to be clustered must be a `List` of `Clusterable` points. The class `DoublePoint` is a convenient wrapper around an array of doubles that implements `Clusterable`. It takes an array of doubles in its constructor.

```
double[][] rawData = ...

List<DoublePoint> data = new ArrayList<>();

for (double[] row : rawData) {
    data.add(new DoublePoint(row));
}

/* num clusters to search for */
int numClusters = 1;

/* the basic constructor */
KMeansPlusPlusClusterer<DoublePoint> kmpp =
    new KMeansPlusPlusClusterer<>(numClusters);

/* this performs the clustering and returns a list with length numClusters */
List<CentroidCluster<DoublePoint>> results = kmpp.cluster(data);

/* iterate the list of Clusterables */
for (CentroidCluster<DoublePoint> result : results) {

    DoublePoint centroid = (DoublePoint) result.getCenter();

    System.out.println(centroid); // DoublePoint has toString() method

    /* we also have access to all the points in only this cluster */
    List<DoublePoint> clusterPoints = result.getPoints();

}
```

In the *k*-means scheme, we want to iterate over several choices of `numClusters`, keeping track of the sum of variances for each cluster. Because variances add, this gives us a measure of total error. Ideally, we want to minimize this number. Here we keep track of the cluster variances as we iterate through various cluster searches:

```
/* search for 1 through 5 clusters */
for (int i = 1; i < 5; i++) {

    KMeansPlusPlusClusterer<DoublePoint> kmpp = new KMeansPlusPlusClusterer<>(i);
    List<CentroidCluster<DoublePoint>> results = kmpp.cluster(data);

    /* this is the sum of variances for this number of clusters */
    SumOfClusterVariances<DoublePoint> clusterVar =
    new SumOfClusterVariances<>(new EuclideanDistance());

    for (CentroidCluster<DoublePoint> result : results) {
        DoublePoint centroid = (DoublePoint) result.getCenter());
    }
}
```

One way we can improve the *k*-means is to try several starting points and take the best result—that is, lowest error. Because the starting points are random, at times the clustering algorithm takes a wrong turn, which even our strategies for handling empty clusters can't handle. It's a good idea to repeat each clustering attempt and choose the one with the best results. The class `MultiKMeansPlusPlusClusterer<T>` performs the same clustering operation `numTrials` times and uses only the best result. We can combine these with the previous code:

```
/* repeat each clustering trial 10 times and take the best */
int numTrials = 10;

/* search for 1 through 5 clusters */
for (int i = 1; i < 5; i++) {

    /* we still need to create a cluster instance ... */
    KMeansPlusPlusClusterer<DoublePoint> kmpp = new KMeansPlusPlusClusterer<>(i);
```

```
        /* ... and pass it to the constructor of the multi */
        MultiKMeansPlusPlusClusterer<DoublePoint> multiKMPP =
            new MultiKMeansPlusPlusClusterer<>(kmpp, numTrials);

        /* NOTE this clusters on multiKMPP NOT kmpp */
        List<CentroidCluster<DoublePoint>> results = multikKMPP.cluster(data);

        /* this is the sum of variances for this number of clusters */
        SumOfClusterVariances<DoublePoint> clusterVar =
            new SumOfClusterVariances<>(new EuclideanDistance());

        /* the sumOfVariance score for 'i' clusters */
        double score = clusterVar.score(results)

        /* the 'best' centroids */
        for (CentroidCluster<DoublePoint> result : results) {
            DoublePoint centroid = (DoublePoint) result.getCenter());
        }
    }
```

DBSCAN

What if clusters have irregular shapes? What if clusters are intertwined? The DBSCAN (density-based spatial clustering of applications with noise) algorithm is ideal for finding hard-to-classify clusters. It does not assume the number of clusters, but rather optimizes itself to the number of clusters present. The only input parameters are the maximum radius of capture and the minimum number of points per cluster. It is implemented as follows:

```
/* constructor takes eps and minpoints */
double eps = 2.0;
int minPts = 3;
DBSCANClusterer clusterer = new DBSCANClusterer(eps, minPts);
List<Cluster<DoublePoint>> results = clusterer.cluster(data);
```

Note that unlike the previous *k*-means++, DBSCAN does not return a `CentroidCluster` type because the centroids of the irregularly shaped clusters may not be meaningful. Instead, you can access the clustered points directly and use them for further processing. But also note that if the algorithm cannot find any clusters, the `List<Cluster<T>>` instance will comprise an empty `List` with a size of 0:

```
if(results.isEmpty()) {
    System.out.println("No clusters were found");
} else {

    for (Cluster<DoublePoint> result : results) {
        /* each clusters points are in here */
        List<DoublePoint> points = result.getPoints();
        System.out.println(points.size());
        // TODO do something with the points in each cluster
    }
}
```

In this example, we have created four random multivariate (two-dimensional) normal clusters. Of note is that two of the clusters are close enough to be touching and could even be considered one angular-shaped cluster. This demonstrates a trade-off in the DBSCAN algorithm.

In this case, we need to set the radius of capture small enough ($\epsilon = 0.225$) to allow detection of the separate clusters, but there are outliers. A larger radius ($\epsilon = 0.8$) here would combine the two leftmost clusters into one, but there would be almost no outliers. As we decrease ϵ, we are enabling a finer resolution for cluster detection, but we also are increasing the likelihood of outliers. This may become less of an issue in higher-dimensional space in which clusters in close proximity to each other are less likely. An example of four Gaussian clusters that fit the DBSCAN algorithm is shown in Figure 5-1.

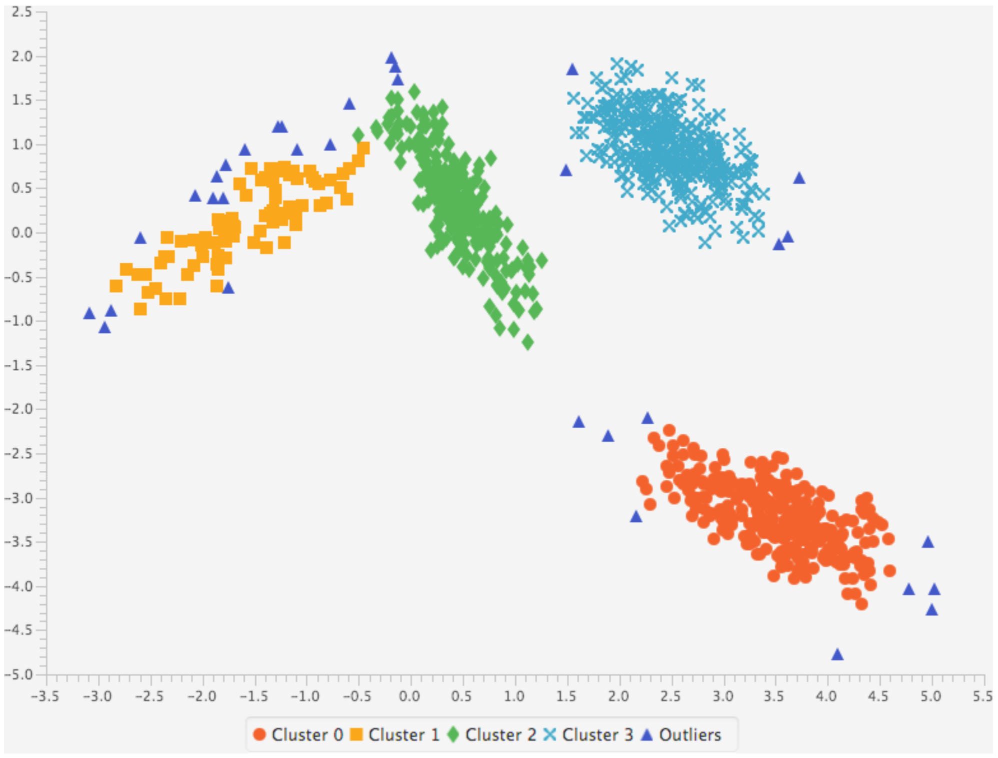

Figure 5-1. DBSCAN on simulation of four Gaussian clusters

DEALING WITH OUTLIERS

The DBSCAN algorithm is well suited for dealing with outliers. How do we access them? Unfortunately, the current Math implementation does not allow access to the points labeled as noise in the DBSCAN algorithm. But we can try to keep track of that like this:

```
/* we are going to keep track of outliers */
// NOTE need a completely new list, not to reference same object
// e.g., outliers = data is not a good idea

// List<DoublePoint> outliers = data; // will remove points from data as well

List<DoublePoint> outliers = new ArrayList<>();

for (DoublePoint dp : data) {
    outliers.add(new DoublePoint(dp.getPoint()));
}
```

Then when we are iterating through the results clusters, we can remove each cluster from the complete dataset data, which will become the outliers after we remove everything else:

```
for (Cluster<DoublePoint> result : results) {

    /* each clusters points are in here */
    List<DoublePoint> points = result.getPoints();


    /* remove these cluster points from the data copy "outliers"
       which will contain ONLY the outliers after all of the
       cluster points are removed
     */

    outliers.removeAll(points);
}

// now the LIST outliers only contains points NOT in any cluster
```

OPTIMIZING RADIUS OF CAPTURE AND MINPOINTS

The radius of capture is easy to see in 2D, but how do you know what is optimal? Clearly, this is entirely subjective and will depend on your use case. In general, the number of minimum points should follow this relation:

$$n_{min} \geq p + 1$$

So in a 2D case, we at least want three minimum points per cluster. The radius of capture, ϵ, can be estimated at the bend in hockey stick of the *k*-distance graph. Both the number of minimum points and the radius of capture can be grid-searched against the silhouette score as a metric. First, find the silhouette coefficient, *s*, for each sample; *a* = the mean distance between that sample and all other points in the class, and *b* = the mean distance between that sample and all the points in the next nearest cluster:

$$s = \frac{b - a}{max(a,\ b)}$$

Then the silhouette score is the mean of all the sample silhouette coefficients. The silhouette score is between –1 and 1: –1 is incorrect clustering, 1 is highly dense clustering, and 0 indicates overlapping clusters. *s* increases as clusters are dense and well separated. As in the case for *k*-means previously, we can vary the ϵ value and output the silhouette score:

```
double[] epsVals = {0.15, 0.16, 0.17, 0.18, 0.19, 0.20,
                    0.21, 0.22, 0.23, 0.24, 0.25};

for (double epsVal : epsVals) {

    DBSCANClusterer clusterer = new DBSCANClusterer(epsVal, minPts);
    List<Cluster<DoublePoint>> results = clusterer.cluster(dbExam.clusterPoints);

    if(results.isEmpty()) {

        System.out.println("No clusters where found");

     } else {

        SilhouetteCoefficient s = new SilhouetteCoefficient(results);
        System.out.println("eps = " + epsVal +
                           " numClusters = " + results.size() +
                           " s = " + s.getCoefficient());
     }
}
```

This gives the following output:

```
eps = 0.15 numClusters = 7 s = 0.54765
eps = 0.16 numClusters = 7 s = 0.53424
eps = 0.17 numClusters = 7 s = 0.53311
eps = 0.18 numClusters = 6 s = 0.68734
eps = 0.19 numClusters = 6 s = 0.68342
eps = 0.20 numClusters = 6 s = 0.67743
eps = 0.21 numClusters = 5 s = 0.68348
eps = 0.22 numClusters = 4 s = 0.70073 // best one!
eps = 0.23 numClusters = 3 s = 0.68861
eps = 0.24 numClusters = 3 s = 0.68766
eps = 0.25 numClusters = 3 s = 0.68571
```

We see a bump in the silhouette score at ϵ = 0.22, where s = 0.7, indicating that the ideal ϵ is approximately 0.22. At this particular ϵ, the DBSCAN routine also converged on four clusters, which is the number we simulated. In practical situations, of course, we won't know the number of clusters beforehand. But this example does indicate that s should approach a maximal value of 1 if we have the right number of clusters and therefore the right ϵ.

INFERENCE FROM DBSCAN

DBSCAN is not for predicting membership of new points as in the *k*-means algorithm. It is for segmenting the data for further use. If you want a predictive model based on DBSCAN, you can assign class values to the clustered data points and try a classification scheme such as Gaussian, naive Bayes, or others.

Gaussian Mixtures

A similar concept to DBSCAN is to cluster based on the density of points, but use the multivariate normal distribution N(μ, Σ) because it comprises a mean and covariance. Data points located near the mean have the highest probability of belonging to that cluster, whereas the probability drops off to almost nothing as the data point is located very far from the mean.

GAUSSIAN MIXTURE MODEL

A Gaussian mixture model is expressed mathematically as a weighted mixture of k multivariate Gaussian distributions (as discussed in Chapter 3).

$$f(\mathbf{x}) = \sum_{i=1}^{k} \alpha_i \mathcal{N}(\boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i)$$

Here the weights satisfy the relation $\sum_i^k \alpha_i = 1$. We must create a `List` of `Pair` objects, where the first member of `Pair` is the weight, and the second member is the distribution itself:

```
List<Pair<Double, MultivariateNormalDistribution>> mixture = new ArrayList<>();

/* mixture component 1 */
double alphaOne = 0.70;
double[] meansOne = {0.0, 0.0};
double[][] covOne = {{1.0, 0.0},{0.0, 1.0}};
MultivariateNormalDistribution distOne =
    new MultivariateNormalDistribution(meansOne, covOne);
Pair pairOne = new Pair(alphaOne, distOne);
mixture.add(pairOne);

/* mixture component 2 */
double alphaTwo = 0.30;
double[] meansTwo = {5.0, 5.0};
double[][] covTwo = {{1.0, 0.0},{0.0, 1.0}};
MultivariateNormalDistribution distTwo =
    new MultivariateNormalDistribution(meansTwo, covTwo);
Pair pairTwo = new Pair(alphaTwo, distTwo);
mixture.add(pairTwo);

/* add the list of pairs to the mixture model and sample the points */
MixtureMultivariateNormalDistribution dist =
    new MixtureMultivariateNormalDistribution(mixture);

/* we don't need a seed, but it helps if we want to recall the same data */
dist.reseedRandomGenerator(0L);

/* generate 1000 random data points from the mixture */
double[][] data = dist.sample(1000);
```

Note that the data sampled from the distribution mixture model does not keep track of what component the sampled data point comes from. In other words, you will not be able to tell what `MultivariateNormal` each sampled data point belongs to. If you require this feature, you can always sample from the individual distributions and then add them together later.

For purposes of testing, creating mixture models can be tedious and is fraught with problems. If you are not building a dataset from existing, real data, it is best to try simulating data with some known problems averted. A method for generating random mixture-model, is presented in Appendix A. In Figure 5-2, a plot of a multivariate Gaussian mixture model is demonstrated. There are two clusters in two dimensions.

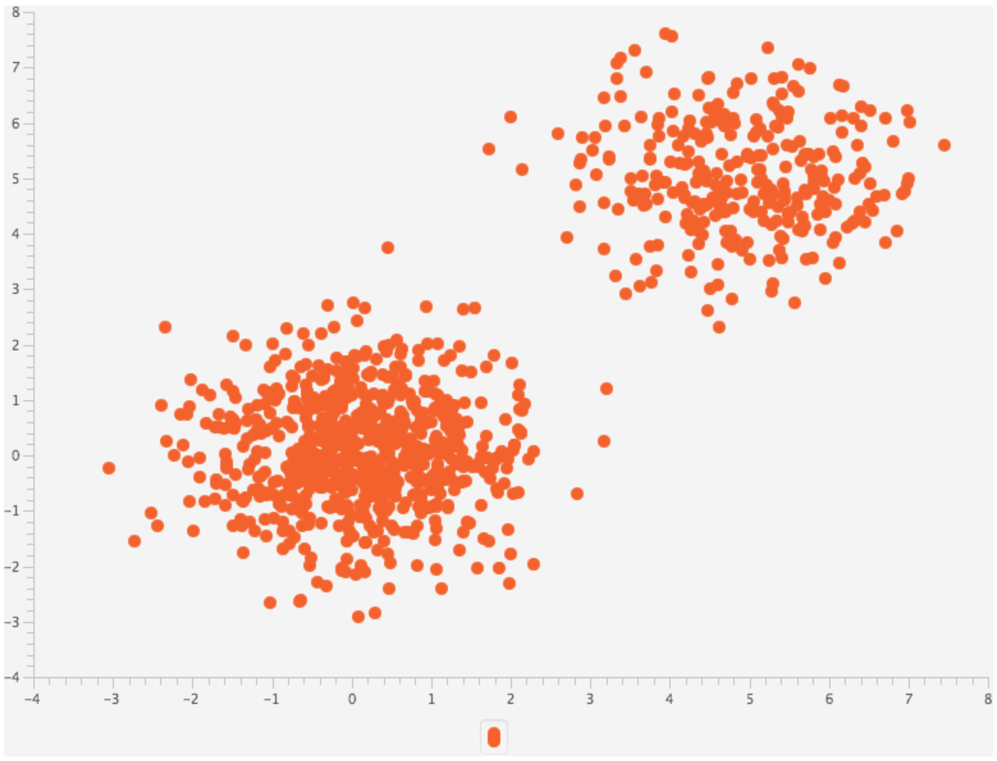

Figure 5-2. Gaussian clusters in 2D

The data can be generated with the example code:

```
int dimension = 5;
int numClusters = 7;
double boxSize = 10;
long seed = 0L;
int numPoints = 10000;

/* see Appendix for this dataset */
MultiNormalMixtureDataset mnd = new MultiNormalMixtureDataset(dimension);
mnd.createRandomMixtureModel(numClusters, boxSize, 0L);
double[][] data = mnd.getSimulatedData(numPoints);
```

FITTING WITH THE EM ALGORITHM

The expectation maximization algorithm is useful in many other places. Essentially, what is the maximum likelihood that the parameters we've chosen are correct? We iterate until they don't change anymore, given a certain tolerance. We need to provide a starting guess of what the mixture is. Using the method in the preceding section, we can create a mixture with known components. However, the static method `MultivariateNormalMixtureExpectationMaximization.estimate(data, numClusters)` is used to estimate the starting point given the dataset and number of clusters as input:

```
MultivariateNormalMixtureExpectationMaximization mixEM =
    new MultivariateNormalMixtureExpectationMaximization(data);

/* need a guess as where to start */
MixtureMultivariateNormalDistribution initialMixture =
    MultivariateNormalMixtureExpectationMaximization.estimate(data, numClusters);

/* perform the fit */
mixEM.fit(initialMixture);

/* this is the fitted model */
MixtureMultivariateNormalDistribution fittedModel = mixEM.getFittedModel();

for (Pair<Double, MultivariateNormalDistribution> pair :
    fittedModel.getComponents()) {
    System.out.println("************ cluster *****************");
    System.out.println("alpha: " + pair.getFirst());
    System.out.println("means: " + new ArrayRealVector(
        pair.getSecond().getMeans()));
    System.out.println("covar: " + pair.getSecond().getCovariances());
}
```

OPTIMIZING THE NUMBER OF CLUSTERS

Just as in *k*-means clustering, we would like to know the optimal number of clusters needed to describe our data. In this case, though, each data point belongs to all clusters with finite probability (soft assignment). How do we know when the number of clusters is good enough? We start with a low number (e.g., 2) and work our way up, calculating the log-likelihood for each trial. To make things easier, we can plot the loss (the negative of the log-likelihood), and watch as it hopefully drops toward zero. Realistically, it never will, but the idea is to stop when the loss becomes somewhat constant. Usually, the best number of clusters will be at the elbow of the hockey stick.

Here is the code:

```
MultivariateNormalMixtureExpectationMaximization mixEM =
    new MultivariateNormalMixtureExpectationMaximization(data);

int minNumClusters = 2;
int maxNumClusters = 10;

for(int i = minNumCluster; i <= maxNumClusters; i++) {

    /* need a guess as where to start */
    MixtureMultivariateNormalDistribution initialMixture =
        MultivariateNormalMixtureExpectationMaximization.estimate(data, i);

    /* perform the fit */
    mixEM.fit(initialMixture);

    /* this is the fitted model */
    MixtureMultivariateNormalDistribution fittedModel = mixEM.getFittedModel();

    /* print out the log-likelihood */
    System.out.println(i + " ll: " + mixEM.getLogLikelihood());
}
```

This outputs the following:

```
2 ll: -6.370643787350135
3 ll: -5.907864928786343
4 ll: -5.5789246749261014
5 ll: -5.366040927493185
6 ll: -5.093391683217386
7 ll: -5.1934910558216165
8 ll: -4.984837507547836
9 ll: -4.9817765145490664
10 ll: -4.981307556011888
```

When plotted, this shows a characteristic hockey-stick shape with the inflection point at `numClusters` = 7, the number of clusters we simulated. Note that we could have stored the loglikelihoods in an array, and fit the results in a `List` for later retrieval programatically. In Figure 5-3, the log-likelihood loss is plotted versus the number of clusters. Note the sharp decline in loss and the bend around seven clusters, the original number of clusters in the simulated dataset.

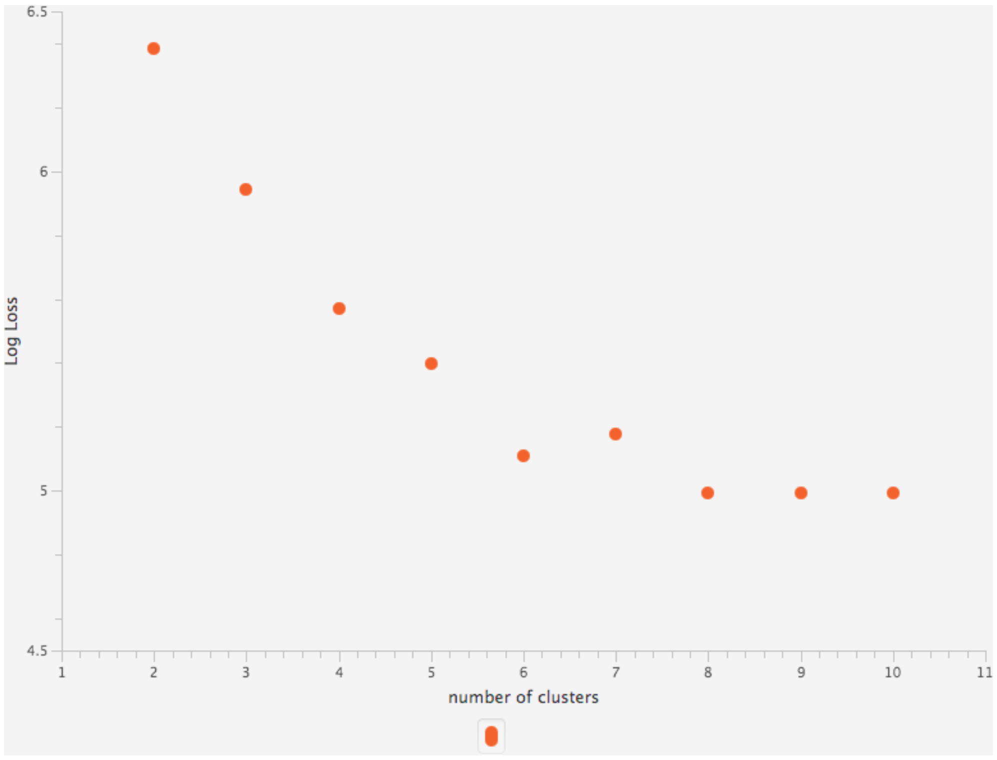

Figure 5-3. Log loss of 7, 5 dimensional clusters

Supervised Learning

Given numeric variates **X** and potentially non-numeric responses **Y**, how can we formulate mathematical models to learn and predict? Recall that linear regression models rely on both **X** and **Y** to be continuous variables (e.g., real numbers). Even when **Y** contains 0s or 1s, (and any other integers), a linear regression would most likely fail.

Here we examine methods specifically designed for the common use cases that collect numeric data as variates and their associated labels. Most classification schemes lend themselves easily to a multidimensional variate **X** and a single dimension of classes **Y**. However, several techniques, including neural networks, can handle multiple output classes **Y** in a way analogous to the multiresponse models of linear regression.

Naive Bayes

Naive Bayes is perhaps the most elementary of classification schemes and is a logical next step after clustering. Recall that in clustering, our goal is to separate or classify data into distinct groups. We can then look at each group individually and try to learn something about that group, such as its center position, its variance, or any other statistical measure.

In a naive Bayes classification scheme, we split the data into groups (classes) for each label type. We then learn something about the variates in each group. This will depend on the type of variable. For example, if the variables are real numbers, we can assume that each dimension (variate) of the data is a sample from a normal distribution.

For integer data (counts), we can assume a multinomial distribution. If the data is binary (0 or 1), we can assume a Bernoulli distributed dataset. In this way, we can estimate statistical quantities such as mean and variance for each of the datasets belonging to only the class it was labeled for. Note that unlike more sophisticated classification schemes, we never use the labels themselves in any computation or error propagation. They serve the purpose only of splitting our data into groups.

According to Bayes' theorem (posterior = prior × likelihood / evidence), the joint probability is the prior × likelihood. In our case, the evidence is the sum of joint probabilities over all classes. For a set of *K* classes, where *k* = {1, 2 … *K*}, the probability of a particular class k given an input vector **x** is determined as follows:

$$p(k \mid \mathbf{x}) = \frac{p(k)p(\mathbf{x} \mid k)}{p(\mathbf{x})}$$

Here, the naive independence assumption allows us to express the likelihood as the product of probabilities for each dimension of the *n*-dimensional variate **x**:

$$p(\mathbf{x} \mid k) = p(x_1 \mid k)p(x_2 \mid k) \cdots p(x_n \mid k)$$

This is expressed more compactly as follows:

$$p(\mathbf{x} \mid k) = \prod_{i=1}^{n} p(x_i \mid k)$$

The normalization is the sum over all terms in the numerator expressed:

$$p(\mathbf{x}) = \sum_{k=1}^{K} p(k)p(\mathbf{x} \mid k)$$

The probability of any class is the number of times it occurs divided by the total: $p(k) = n_k / N$. Here we take the product for each class *k* over each feature x_i. The form of $p(x_i \mid C_K)$, is probability density function we choose based on our assumptions of the data. In the following sections, we explore normal, multinomial, and Bernoulli distributions.

WARNING

Note that if any one calculation $p(x_i \mid k) = 0$, the entire expression will be $p(k \mid \mathbf{x}) = 0$. For some conditional probability models such as Gaussian or Bernoulli distributions, this will never be the case. But for a multinomial distribution, this can occur, so we include a small factor α to avoid this.

After we calculate posterior probabilities for each class, a Bayes classifier is then a decision rule on the posterior probabilities, where we take the maximum position as the most likely class:

$$\hat{k} = \underset{k \in \{1,2,\cdots K\}}{\arg\max}\, p(k \mid \mathbf{x})$$

We can use the same class for all types, because training the model relies on the types of quantities that are easily accumulated with `MultivariateSummaryStatistics` per class. We can then use a strategy pattern to implement whichever type of conditional probability we require and pass it directly into the constructor:

```java
public class NaiveBayes {

    Map <Integer, MultivariateSummaryStatistics> statistics;
    ConditionalProbabilityEstimator conditionalProbabilityEstimator;
    int numberOfPoints; // total number of points the model was trained on

    public NaiveBayes(
        ConditionalProbabilityEstimator conditionalProbabilityEstimator) {
        statistics = new HashMap<>();
        this.conditionalProbabilityEstimator = conditionalProbabilityEstimator;
        numberOfPoints = 0;
    }

    public void learn(RealMatrix input, RealMatrix target) {
        // if numTargetCols == 1 then multiclass e.g. 0, 1, 2, 3
        // else one-hot e.g. 1000, 0100, 0010, 0001
        numberOfPoints += input.getRowDimension();
        for (int i = 0; i < input.getRowDimension(); i++) {
            double[] rowData = input.getRow(i);
            int label;
            if (target.getColumnDimension()==1) {
                label = new Double(target.getEntry(i, 0)).intValue();
            } else {
                label = target.getRowVector(i).getMaxIndex();
            }

            if(!statistics.containsKey(label)) {
                statistics.put(label, new MultivariateSummaryStatistics(
                rowData.length, true));
            }
            statistics.get(label).addValue(rowData);
        }
    }

    public RealMatrix predict(RealMatrix input) {

        int numRows = input.getRowDimension();
        int numCols = statistics.size();
        RealMatrix predictions = new Array2DRowRealMatrix(numRows, numCols);

        for (int i = 0; i < numRows; i++) {
            double[] rowData = input.getRow(i);
            double[] probs = new double[numCols];
            double sumProbs = 0;
            for (Map.Entry<Integer, MultivariateSummaryStatistics> entrySet :
            statistics.entrySet()) {

                Integer classNumber = entrySet.getKey();
                MultivariateSummaryStatistics mss = entrySet.getValue();

                /* prior prob n_k / N ie num points in class / total points */
                double prob = new Long(mss.getN()).doubleValue()/numberOfPoints;

                /* depends on type ... Gaussian, Multinomial, or Bernoulli */
                prob *= conditionalProbabilityEstimator.getProbability(mss,
                                                                      rowData);

                probs[classNumber] = prob;
                sumProbs += prob;
            }
            /* L1 norm the probs */
            for (int j = 0; j < numCols; j++) {
                probs[j] /= sumProbs;
            }
            predictions.setRow(i, probs);
        }
        return predictions;
    }
}
```

All that is needed then is an interface that designates the form of the conditional probability:

```
public interface ConditionalProbabilityEstimator {
    public double getProbability(MultivariateSummaryStatistics mss,
                                 double[] features);
}
```

In the following three subsections, we explore three kinds of naive Bayes classifiers, each of which implements the `ConditionalProbabilityEstimator` interface for use in the `NaiveBayes` class.

GAUSSIAN

If the features are continuous variables, we can use the Gaussian naive Bayes classifier:

$$p(\mathbf{x} \mid k) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi}\sigma_{ki}} \exp\left(- \frac{(x_i - \mu_{ki})^2}{2\sigma_{ki}^2} \right)$$

We can then implement a class like this:

```
import org.apache.commons.math3.distribution.NormalDistribution;
import org.apache.commons.math3.stat.descriptive.MultivariateSummaryStatistics;

public class GaussianConditionalProbabilityEstimator
implements ConditionalProbabilityEstimator{

    @Override
    public double getProbability(MultivariateSummaryStatistics mss,
                                 double[] features) {
        double[] means = mss.getMean();
        double[] stds = mss.getStandardDeviation();
        double prob = 1.0;
        for (int i = 0; i < features.length; i++) {
            prob *= new NormalDistribution(means[i], stds[i])
                .density(features[i]);
        }
        return prob;
    }

}
```

And test it like this:

```
double[][] features = {{6, 180, 12},{5.92, 190, 11}, {5.58, 170, 12},
                       {5.92, 165, 10}, {5, 100, 6}, {5.5, 150, 8},
                       {5.42, 130, 7}, {5.75, 150, 9}};
String[] labels = {"male", "male", "male", "male",
                   "female", "female", "female", "female"};
NaiveBayes nb = new NaiveBayes(new GaussianConditionalProbabilityEstimator());
nb.train(features, labels);
```

```
double[] test = {6, 130, 8};
String inference = nb.inference(test); // "female"
```

This will yield the correct result, `female`.

MULTINOMIAL

Features are integer values—for example, counts. However, continuous features such as TFIDF also can work. The likelihood of observing any feature vector **x** for class k is as follows:

$$p(\mathbf{x} \mid k) = \frac{(\sum_{i=1}^{n} x_i)\,!}{\prod_{i=1}^{n} x_i\,!} \prod_{i=1}^{n} p_{ki}^{x_i}$$

We note that the front part of the term depends only on the input vector **x**, and therefore is equivalent for each calculation of $p(\mathbf{x}|k)$. Fortunately, this computationally intense term will drop out in the final, normalized expression for $p(k|\mathbf{x})$, allowing us to use the much simpler formulation:

$$p(\mathbf{x} \mid k) = \prod_{i=1}^{n} p_{ki}^{x_i}$$

We can easily calculate the required probability $p_{ki} = N_{ik} / N_k$ where $N_i k$ is the sum of values for each feature, given class *k*, and N_k is the total count for all features, given class *k*. When estimating the conditional probabilities, any zeros will cancel out the entire calculation. It is therefore useful to estimate the probabilities with a small additive factor α—known as *Lidstone smoothing* for generalized α, and *Laplace smoothing* when $p(\mathbf{x} \mid k) = \prod_{i=1}^{n} \left(\frac{N_{ik} + \alpha}{N_k + \alpha n}\right)^{x_i}$. As a result of this L1 normalization of the numerator, the factor n is just the dimension of the feature vector.

$$p_{ki} = \frac{N_{ik} + \alpha}{N_k + \alpha n}$$

The final expression is as follows:

$$p(\mathbf{x} \mid k) = \prod_{i=1}^{n} \left(\frac{N_{ik} + \alpha}{N_k + \alpha n} \right)^{x_i}$$

For large x_i (large counts of words, for example), the problem may become numerically intractable. We can solve the problem in log space and convert it back by using the relation $z = \exp(\ln(z))$. The preceding expression can be written as follows:

$$p(\mathbf{x} \mid k) = \exp\left(\sum_{i=1}^{n} x_i \ln\left(\frac{N_{ik} + \alpha}{N_k + \alpha n} \right) \right)$$

In this strategy implementation, the smoothing coefficient is designated in the constructor. Note the use of the logarithmic implementation to avoid numerical instability. It would be wise to add an assertion (in the constructor) that the smoothing constant alpha hold the relation $0 > \alpha \geq 1$.

```
public class MultinomialConditionalProbabilityEstimator
        implements ConditionalProbabilityEstimator {

    private double alpha;

    public MultinomialConditionalProbabilityEstimator(double alpha) {
        this.alpha = alpha; // Lidstone smoothing 0 > alpha > 1
    }

    public MultinomialConditionalProbabilityEstimator() {
        this(1); // Laplace smoothing
    }

    @Override
    public double getProbability(MultivariateSummaryStatistics mss,
                                 double[] features) {
        int n = features.length;
        double prob = 0;
        double[] sum = mss.getSum(); // array of x_i sums for this class
        double total = 0.0; // total count of all features
        for (int i = 0; i < n; i++) {
            total += sum[i];
        }
        for (int i = 0; i < n; i++) {
            prob += features[i] * Math.log((sum[i] + alpha) /(total+alpha*n));
        }
        return Math.exp(prob);
    }

}
```

BERNOULLI

Features are binary values—for example, occupancy status. The probability per feature is the mean value for that column. For an input feature, we can then calculate the probability:

$$p(\mathbf{x} \mid k) = \prod_{i=1}^{n} (p_{ki} x_i + (1 - p_{ki})(1 - x_i))$$

In other words, if the input feature is a 1, the probability for that feature is the mean value for the column. If the input feature is a 0, the probability for that feature is 1 – mean of that column. We implement the Bernoulli conditional probability as shown here:

```
public class BernoulliConditionalProbabilityEstimator
    implements ConditionalProbabilityEstimator {

    @Override
    public double getProbability(MultivariateSummaryStatistics mss,
                                 double[] features) {
        int n = features.length;
        double[] means = mss.getMean();
        // this is actually the prob per features e.g. count / total
        double prob = 1.0;
        for (int i = 0; i < n; i++) {
            // if x_i = 1, then p, if x_i = 0 then 1-p, but here x_i is a double
            prob *= (features[i] > 0.0) ? means[i] : 1-means[i];
        }
        return prob;
    }
}
```

IRIS EXAMPLE

Try the Iris dataset by using a Gaussian conditional probability estimator:

```
Iris iris = new Iris();
MatrixResampler mr = new MatrixResampler(iris.getFeatures(), iris.getLabels());
mr.calculateTestTrainSplit(0.4, 0L);

NaiveBayes nb = new NaiveBayes(new GaussianConditionalProbabilityEstimator());
nb.learn(mr.getTrainingFeatures(), mr.getTrainingLabels());

RealMatrix predictions = nb.predict(mr.getTestingFeatures());

ClassifierAccuracy acc = new ClassifierAccuracy(predictions,
                                                mr.getTestingLabels());
System.out.println(acc.getAccuracyPerDimension()); // {1; 1; 0.9642857143}
System.out.println(acc.getAccuracy()); // 0.9833333333333333
```

Linear Models

If we rotate, translate, and scale a dataset **X**, can we relate it to the output **Y** by mapping a function? In general, these all seek to solve the problem in which an input matrix **X** is the data, and **W** and **b** are the free parameters we want to optimize for. Using the notation developed in Chapter 2, for a weighted input matrix and intercept $\mathbf{Z} = \mathbf{XW} + \mathbf{hb}^T$, we apply a function $\varphi(\mathbf{Z})$ to each element of **Z** to compute a prediction matrix $\hat{\mathbf{Y}}$ such that

$$\hat{\mathbf{Y}} = \varphi(\mathbf{XW} + \mathbf{hb}^T)$$

We can view a linear model as a box with input **X** and predicted output $\hat{\mathbf{Y}}$. When optimizing the free parameters **W** and **b**, the error on the output can be sent back through the box, providing incremental updates dependent on the algorithm chosen. Of note is that we can even pass the error back to the input, calculating the error on the input. For linear models, this is not necessary, but as we will see in "Deep Networks", this is essential for the back-propagation algorithm. A generalized linear model is shown in Figure 5-4.

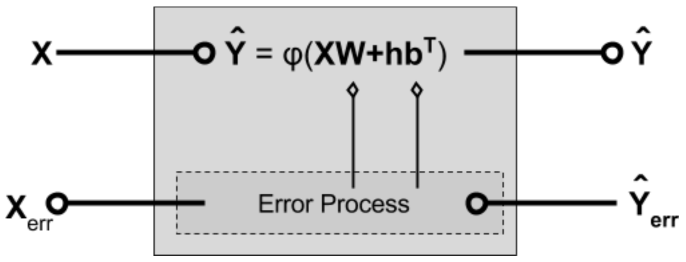

Figure 5-4. Linear model

We can then implement a `LinearModel` class that is responsible only for holding the type of output function, the state of the free parameters, and simple methods for updating the parameters:

```
public class LinearModel {

    private RealMatrix weight;
    private RealVector bias;
    private final OutputFunction outputFunction;

    public LinearModel(int inputDimension, int outputDimension,
    OutputFunction outputFunction) {
        weight = MatrixOperations.getUniformRandomMatrix(inputDimension,
        outputDimension, 0L);
        bias = MatrixOperations.getUniformRandomVector(outputDimension, 0L);
        this.outputFunction = outputFunction;
    }

    public RealMatrix getOutput(RealMatrix input) {
        return outputFunction.getOutput(input, weight, bias);
    }

    public void addUpdateToWeight(RealMatrix weightUpdate) {
        weight = weight.add(weightUpdate);
    }

    public void addUpdateToBias(RealVector biasUpdate) {
        bias = bias.add(biasUpdate);
    }
}
```

The interface for an output function is shown here:

```
public interface OutputFunction {
    RealMatrix getOutput(RealMatrix input, RealMatrix weight, RealVector bias);
    RealMatrix getDelta(RealMatrix error, RealMatrix output);
}
```

In most cases, we can never precisely determine **W** and **b** such that the relation between **X** and **Y** is exact. The best we can do is to estimate **Y**, calling it $\hat{\mathbf{Y}}$, and then proceed to minimize a loss function of our choice $\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})$. The goal is then to incrementally update the values of **W** and **b** over a set of iterations (annotated by *t*) according to the following:

$$\mathbf{W}_{t+1} = \mathbf{W}_t + \Delta\mathbf{W}_t$$

$$\mathbf{b}_{t+1} = \mathbf{b}_t + \Delta\mathbf{b}_t$$

In this section, we will focus on the use of the gradient descent algorithm for determining the values of **W** and **b**. Recalling that the loss function is ultimately a function of both **W** and **b**, we can use the gradient descent optimizer to make the incremental updates with the gradient of the loss:

$$\mathbf{W}_{t+1} = \mathbf{W}_t - \eta \nabla \mathscr{L}(\mathbf{W})_t$$
$$\mathbf{b}_{t+1} = \mathbf{b}_t - \eta \nabla \mathscr{L}(\mathbf{b})_t$$

The objective function to be optimized is the mean loss $\langle \mathscr{L}(\mathbf{y}, \hat{\mathbf{y}}) \rangle$, where the gradient of any particular term with respect to a parameter $w_{i}j$ and b_j can be expressed as follows:

$$\frac{\partial \mathscr{L}}{\partial w} = \frac{\partial \mathscr{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} \frac{\partial z}{\partial w}$$

The first term is the derivative of the loss function, which we covered in the prior section. The second part of the term is the derivative of the output function:

$$\frac{\partial \hat{y}}{\partial z} = \varphi'(z)$$

The third term is simply the derivative of *z* with respect to either *w* or *b*:

$$\frac{\partial z}{\partial w} = x$$
$$\frac{\partial z}{\partial b} = 1$$

As we will see, the choice of the appropriate pair of loss function and output function will lead to a mathematical simplification that leads to the *delta rule*. In this case, the updates to the weights and bias are always as follows:

$$\Delta \mathbf{W} = -\eta \mathbf{X}^T (\mathbf{Y} - \mathbf{T})$$
$$\Delta \mathbf{b} = -\eta \mathbf{h}^T (\mathbf{Y} - \mathbf{T})$$

When the mean loss $\langle \mathscr{L}(\mathbf{y}, \hat{\mathbf{y}}) \rangle$ stops changing within a certain numerical tolerance (e.g., 10E – 6), the process can stop, and we assume **W** and **b** are at their optimal values. However, iterative algorithms are prone to iterate forever because of numerical oddities. Therefore, all iterative solvers will set a maximum number of iterations (e.g., 1,000), after which the process will terminate. It is good practice to always check whether the maximum number of iterations was reached, because the change in loss may still be high, indicating that the optimal values of the free parameters have not yet been attained. The form of both the transformation function $\varphi(\mathbf{Z})$ and the loss function $L(\hat{\mathbf{Y}}, \mathbf{Y})$ will depend on the problem at hand. Several common scenarios are detailed next.

LINEAR

In the case of linear regression, $\varphi(\mathbf{Z})$ is set to the identity function, and the output is equivalent to the input:

$$\varphi(\mathbf{Z}) = \mathbf{Z}$$

This provides the familiar form of a linear regression model:

$$\hat{\mathbf{Y}} = \mathbf{XW} + \mathbf{hb}^T$$

We solved this problem in both Chapters 2 and 3 by using different methods. In the case of Chapter 2, we solved for the free parameters by posing the problem in matrix notation and then using a back-solver, whereas in Chapter 3 we took the least-squares approach. There are, however, even more ways to solve this problem! Ridge regression, lasso regression, and elastic nets are just a few examples. The idea is to eliminate variables that are not useful by penalizing their parameters during the optimization process:

```java
public class LinearOutputFunction implements OutputFunction {

    @Override
    public RealMatrix getOutput(RealMatrix input, RealMatrix weight,
    RealVector bias) {
        return MatrixOperations.XWplusB(input, weight, bias);
    }

    @Override
    public RealMatrix getDelta(RealMatrix errorGradient, RealMatrix output) {
        // output gradient is all 1's ... so just return errorGradient
        return errorGradient;
    }

}
```

LOGISTIC

Solve the problem where y is a 0 or 1 that can also be multidimensional, such as $\mathbf{y}$ = 0,1,1,0,1. The nonlinear function $\varphi(z) = \frac{1}{1 + \exp(-z)}$

For gradient descent, we need the derivative of the function:

$$\varphi'(z) = \frac{\exp(-z)}{(1 + \exp(-z))^2}$$

It is convenient to note that the derivative may also be expressed in terms of the original function. This is useful, because it allows us to reuse the calculated values of φ rather than having to recompute all the computationally costly matrix algebra:

$$\varphi'(z) = \varphi(z)(1 - \varphi(z))$$

In the case of gradient descent, we can then implement it as follows:

```
public class LogisticOutputFunction implements OutputFunction {

    @Override
    public RealMatrix getOutput(RealMatrix input, RealMatrix weight,
    RealVector bias) {
        return MatrixOperations.XWplusB(input, weight, bias, new Sigmoid());
    }

    @Override
    public RealMatrix getDelta(RealMatrix errorGradient, RealMatrix output) {

        // this changes output permanently
        output.walkInOptimizedOrder(new UnivariateFunctionMapper(
        new LogisticGradient()));

        // output is now the output gradient
        return MatrixOperations.ebeMultiply(errorGradient, output);
    }

    private class LogisticGradient implements UnivariateFunction {

        @Override
        public double value(double x) {
            return x * (1 - x);
        }
    }
}
```

When using the cross-entropy loss function to calculate the loss term, note that $\hat{y} = \varphi(z)$ such that:

$$\frac{\partial \mathscr{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} = \frac{\hat{y} - y}{\hat{y}(1 - \hat{y})} \hat{y}(1 - \hat{y})$$

So it then reduces to this:

$$\frac{\partial \mathscr{L}}{\partial z} = \hat{y} - y$$

And considering

$$\frac{\partial z}{\partial w} = x$$

the gradient of the loss with respect to the weight is as follows:

$$\frac{\partial \mathscr{L}}{\partial w} = (\hat{y} - y)x$$

We can include a learning rate η to slow the update process. The final formulas, adapted for use with matrices of data, are given as follows:

$$\Delta \mathbf{W} = -\eta \mathbf{X}^T(\mathbf{Y} - \mathbf{T})$$

$$\Delta \mathbf{b} = -\eta \mathbf{h}^T(\mathbf{Y} - \mathbf{T})$$

Here, **h** is an *m*-dimensional vector of 1s. Notice the inclusion of the learning rate η, which usually takes on values between 0.0001 and 1 and limits how fast the parameters converge. For small values of η, we are more likely to find the correct values of the weights, but at the cost of performing many more time-consuming iterations. For larger values of η, we will complete the algorithmic learning task much quicker. However, we may inadvertently skip over the best solution, giving nonsensical values for the weights.

SOFTMAX

Softmax is similar to logistic regression, but the target variable can be multinomial (an integer between 0 and `numClasses` – 1). We then transform the output with one-hot encoding such that **Y** = {0,0,1,0}. Note that unlike multi-output logistic regression, only one position in each row can be set to 1, and all others must be 0. Each element of the transformed matrix is then exponentiated and then L1 normalized row-wise:

$$\varphi(z_{ij}) = \frac{\exp(z_{ij})}{\sum_j \exp(z_{ij})}$$

Because the derivative involves more than one variable, the Jacobian takes the place of the gradient with terms:

$$\varphi'(z) = \begin{cases} \varphi_i(z)(1 - \varphi_i(z)) & \text{for } i = j \\ -\varphi_i(z)\varphi_j(z) & \text{for } i \neq j \end{cases}$$

Then for a single *p*-dimensional output and prediction, we can calculate the quantity:

$$\frac{\partial \mathscr{L}}{\partial \hat{\mathbf{y}}}\frac{\partial \hat{\mathbf{y}}}{\partial z} = \begin{pmatrix} \frac{-y_1}{\hat{y}_1} & \frac{-y_2}{\hat{y}_2} & \cdots & \frac{-y_p}{\hat{y}_p} \end{pmatrix} \begin{pmatrix} \hat{y}_1(1-\hat{y}_1) & -\hat{y}_1\hat{y}_2 & \cdots & -\hat{y}_1\hat{y}_p \\ -\hat{y}_2\hat{y}_1 & \hat{y}_2(1-\hat{y}_2) & \cdots & -\hat{y}_2\hat{y}_p \\ \vdots & \vdots & \ddots & \vdots \\ -\hat{y}_p\hat{y}_1 & -\hat{y}_p\hat{y}_2 & \cdots & \hat{y}_p(1-\hat{y}_p) \end{pmatrix}$$

This simplifies to the following:

$$\frac{\partial \mathcal{L}}{\partial z} = ((\hat{y}_1 - y_1) \quad (\hat{y}_2 - y_2) \cdots (\hat{y}_p - y_p))$$

Each term has the exact same update rule as the other linear models under gradient descent:

$$\frac{\partial \mathcal{L}}{\partial w} = (\hat{y} - y)x$$

As a practical matter, it will take two passes over the input to compute the softmax output. First, we raise each argument by the exponential function, keeping track of the running sum. Then we iterate over that list again, dividing each term by the running sum. If (and only if) we use the softmax cross entropy as an error, the update formula for the coefficients is identical to that of logistic regression. We show this calculation explicitly in "Deep Networks".

```
public class SoftmaxOutputFunction implements OutputFunction {

    @Override
    public RealMatrix getOutput(RealMatrix input, RealMatrix weight,
                                RealVector bias) {
        RealMatrix output = MatrixOperations.XWplusB(input, weight, bias,
            new Exp());
        MatrixScaler.l1(output);
        return output;
    }

    @Override
    public RealMatrix getDelta(RealMatrix error, RealMatrix output) {

        RealMatrix delta = new BlockRealMatrix(error.getRowDimension(),
        error.getColumnDimension());

        for (int i = 0; i < output.getRowDimension(); i++) {
            delta.setRowVector(i, getJacobian(output.getRowVector(i)).
            preMultiply(error.getRowVector(i)));
        }

        return delta;
    }

    private RealMatrix getJacobian(RealVector output) {

        int numRows = output.getDimension();
        int numCols = output.getDimension();
        RealMatrix jacobian = new BlockRealMatrix(numRows, numCols);
        for (int i = 0; i < numRows; i++) {
            double output_i = output.getEntry(i);
            for (int j = i; j < numCols; j++) {
                double output_j = output.getEntry(j);
                if(i==j) {
                    jacobian.setEntry(i, i, output_i*(1-output_i));
                } else {
                    jacobian.setEntry(i, j, -1.0 * output_i * output_j);
                    jacobian.setEntry(j, i, -1.0 * output_j * output_i);
                }
            }
        }
        return jacobian;
    }
}
```

TANH

Another common activation function utilizes the hyperbolic tangent $\tanh(z)$ with the form shown here:

$$\varphi(z) = \tanh(z)$$
$$\varphi'(z) = 1 - \tanh^2(z) = 1 - \varphi(z)^2$$

Once again, the derivative $\varphi'(z)$ reuses the value calculated from $\varphi(z)$:

```
public class TanhOutputFunction implements OutputFunction {

    @Override
    public RealMatrix getOutput(RealMatrix input, RealMatrix weight,
                                RealVector bias) {
        return MatrixOperations.XWplusB(input, weight, bias, new Tanh());
    }

    @Override
    public RealMatrix getDelta(RealMatrix errorGradient, RealMatrix output) {
        // this changes output permanently
        output.walkInOptimizedOrder(
            new UnivariateFunctionMapper(new TanhGradient()));

        // output is now the output gradient
        return MatrixOperations.ebeMultiply(errorGradient, output);
    }

    private class TanhGradient implements UnivariateFunction {
        @Override
        public double value(double x) {
            return (1 - x * x);
        }
    }
}
```

LINEAR MODEL ESTIMATOR

Using the gradient descent algorithm and the appropriate loss functions, we can build a simple linear estimator that updates the parameters iteratively using the delta rule. This applies only if the correct pairing of output function and loss function are used, as shown in Table 5-1.

Table 5-1. Delta rule pairings

Output function	Loss function
Linear	Quadratic
Logistic	Bernoulli cross-entropy
Softmax	Multinomial cross-entropy
Tanh	Two-point cross-entropy

We can then extend the `IterativeLearningProcess` class and add code for the output function prediction and updates:

```
public class LinearModelEstimator extends IterativeLearningProcess {

    private final LinearModel linearModel;
    private final Optimizer optimizer;

    public LinearModelEstimator(
            LinearModel linearModel,
            LossFunction lossFunction,
            Optimizer optimizer) {
        super(lossFunction);
        this.linearModel = linearModel;
        this.optimizer = optimizer;
    }

    @Override
    public RealMatrix predict(RealMatrix input) {
        return linearModel.getOutput(input);
    }

    @Override
    protected void update(RealMatrix input, RealMatrix target,
                          RealMatrix output) {
        RealMatrix weightGradient =
            input.transpose().multiply(output.subtract(target));
        RealMatrix weightUpdate = optimizer.getWeightUpdate(weightGradient);
        linearModel.addUpdateToWeight(weightUpdate);

        RealVector h = new ArrayRealVector(input.getRowDimension(), 1.0);
        RealVector biasGradient = output.subtract(target).preMultiply(h);
        RealVector biasUpdate = optimizer.getBiasUpdate(biasGradient);
        linearModel.addUpdateToBias(biasUpdate);

    }

    public LinearModel getLinearModel() {
        return linearModel;
    }

    public Optimizer getOptimizer() {
        return optimizer;
    }
}
```

IRIS EXAMPLE

The Iris dataset is a great example to explore a linear classifier:

```
/* get data and split into train / test sets */
Iris iris = new Iris();
MatrixResampler resampler = new MatrixResampler(iris.getFeatures(),
iris.getLabels());
resampler.calculateTestTrainSplit(0.40, 0L);

/* set up the linear estimator */
LinearModelEstimator estimator = new LinearModelEstimator(
    new LinearModel(4, 3, new SoftmaxOutputFunction()),
    new SoftMaxCrossEntropyLossFunction(),
    new DeltaRule(0.001));

estimator.setBatchSize(0);
estimator.setMaxIterations(6000);
estimator.setTolerance(10E-6);

/* learn the model parameters */
estimator.learn(resampler.getTrainingFeatures(), resampler.getTrainingLabels());

/* predict on test data */
RealMatrix prediction = estimator.predict(resampler.getTestingFeatures());

/* results */
ClassifierAccuracy accuracy = new ClassifierAccuracy(prediction,
 resampler.getTestingLabels());

estimator.isConverged();            // true
estimator.getNumIterations();       // 3094
estimator.getLoss();                // 0.0769
accuracy.getAccuracy();             // 0.983
accuracy.getAccuracyPerDimension(); // {1.0, 0.92, 1.0}
```

Deep Networks

Feeding the output of a linear model into another linear model creates a nonlinear system capable of modeling complicated behavior. A system with multiple layers is known as a deep network. While a linear model has an input and output, a deep network adds multiple “hidden” layers between the input and output. Most explanations of deep networks address the input, hidden, and output layers as separate quantities. In this book, however, we take the alternative viewpoint that a deep network is nothing more than a composition of linear models. We can then view a deep network purely as a linear algebra problem. Figure 5-5 demonstrates how a multilayer neural network can be viewed as a chain of linear models.

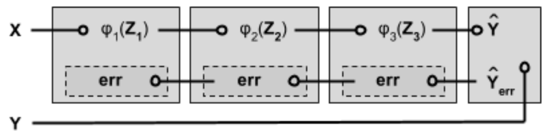

Figure 5-5. Deep network

A NETWORK LAYER

We can extend the concept of a linear model to the form of a network layer where we must persist the input, output, and errors. The code for the network layer is then an extension of the `LinearModel` class:

```
public class NetworkLayer extends LinearModel {

    RealMatrix input;
    RealMatrix inputError;
    RealMatrix output;
    RealMatrix outputError;
    Optimizer optimizer;

    public NetworkLayer(int inputDimension, int outputDimension,
            OutputFunction outputFunction, Optimizer optimizer) {
        super(inputDimension, outputDimension, outputFunction);
        this.optimizer = optimizer;
    }

    public void update() {

        //back propagate error
        /* D = eps o f'(XW) where o is Hadamard product
        or J f'(XW) where J is Jacobian */
        RealMatrix deltas = getOutputFunction().getDelta(outputError, output);

        /* E_out = D W^T */
        inputError = deltas.multiply(getWeight().transpose());

        /* W = W - alpha * delta * input */
        RealMatrix weightGradient = input.transpose().multiply(deltas);

        /* w_{t+1} = w_{t} + \delta w_{t} */
        addUpdateToWeight(optimizer.getWeightUpdate(weightGradient));

        // this essentially sums the columns of delta and that vector is grad_b
        RealVector h = new ArrayRealVector(input.getRowDimension(), 1.0);
        RealVector biasGradient = deltas.preMultiply(h);
        addUpdateToBias(optimizer.getBiasUpdate(biasGradient));
    }
}
```

FEED FORWARD

To calculate the network output, we must feed the input through each layer of the network in the forward direction. The network input $\mathbf{X}_1$ is used to compute the output of the first layer:

$$\hat{\mathbf{Y}}_1 = \varphi\left(\mathbf{X}_1\mathbf{W}_1 + \mathbf{h}\mathbf{b}_1^T\right)$$

We set the first-layer output to the second-layer input:

$$\mathbf{X}_2 = \hat{\mathbf{Y}}_1$$

The output of the second layer is shown here:

$$\hat{\mathbf{Y}}_2 = \varphi\left(\mathbf{X}_2\mathbf{W}_2 + \mathbf{h}\mathbf{b}_2^T\right) = \varphi\left(\hat{\mathbf{Y}}_1\mathbf{W}_2 + \mathbf{h}\mathbf{b}_2^T\right)$$

In general, the output of each layer *l* after the first layer is expressed in terms of the prior-layer output:

$$\hat{\mathbf{Y}}_l = \varphi\left(\hat{\mathbf{Y}}_{l-1}\mathbf{W}_l\right) + \mathbf{h}\mathbf{b}_l^T)$$

The feed-forward process for *L* layers then appears as a series of nested linear models:

$$\hat{\mathbf{Y}}_L = \varphi_L\left(\cdots \varphi_2\left(\varphi_1\left(\mathbf{X}_1\mathbf{W}_1 + \mathbf{h}\mathbf{b}_1^T\right)\mathbf{W}_2 + \mathbf{h}\mathbf{b}_2^T\right) \cdots \mathbf{W}_L + \mathbf{h}\mathbf{b}_L^T\right)$$

Another simpler way to write this expression is as a composition of functions:

$$\hat{\mathbf{Y}}_L = \varphi_L \circ \cdots \circ \varphi_2 \circ \varphi_1(\mathbf{Z})$$

In addition to unique weights, each layer can take a different form for the activation function. In this way, it is clear that a feed-forward, deep neural network (a.k.a. multilayer perceptron) is nothing more than a composition of arbitrary linear models. The result, however, is a rather complex nonlinear model.

BACK PROPAGATION

At this point, we need to back-propagate the network output error. For the last layer (the output layer), we back-propagate the loss gradient $\nabla\mathscr{L}(\mathbf{Y}, \hat{\mathbf{Y}})$. For compatible loss function–output function pairs, this is identical to a linear model estimator. As formulated in “Gradient Descent Optimizer”, it is then convenient to define a new quantity, the layer delta, **D**, which is the matrix multiplication of $\mathbf{Y}_{err}$ with the tensor of output function Jacobians:

$$\mathbf{D} = \hat{\mathbf{Y}}_{err}\mathbf{J}_{\varphi}^{(m)}$$

In most cases, the gradient of the output function will suffice and the prior expression can be simplified as as follows:

$$\mathbf{D} = \hat{\mathbf{Y}}_{err} \circ \varphi'(\mathbf{Z})$$

We store the quantity **D** since it is used in two places that must be calculated in order. The back-propagated error is updated first:

$$\mathbf{X}_{err} = \mathbf{D}\mathbf{W}^T$$

The weight and bias gradients are then calculated as follows, where **h** is an m-length vector of 1s.

$$\nabla\mathbf{W} = \mathbf{X}^T\mathbf{D}$$

$$\nabla\mathbf{b} = \mathbf{h}^T\mathbf{D}$$

Note that the expression $\mathbf{h}^T\mathbf{D}$ is the equivalent of summing each column of **D**. The layer weights can then be updated using the optimization rule of choice (usually gradient descent of some variety). That completes all the calculations needed for the network layer! Then set the $\mathbf{Y}_{err}$ of the next layer down to the freshly calculated $\mathbf{X}_{err}$ and repeat the process until the first layer's parameters are updated.

WARNING

Make sure to calculate the back-propagated error before updating the weight!

DEEP NETWORK ESTIMATOR

Learning the parameters of a deep network is accomplished with the same iterative process as a linear model. In this case, the entire feed-forward process acts as one prediction step and the back propagation process acts as one update step. We implement a deep network estimator by extending the `IterativeLearningProcess` and building layers of linear models subclassed as `NetworkLayers`:

```
public class DeepNetwork extends IterativeLearningProcess {

    private final List<NetworkLayer> layers;

    public DeepNetwork() {
        this.layers = new ArrayList<>();
    }

    public void addLayer(NetworkLayer networkLayer) {
        layers.add(networkLayer);
    }

    @Override
    public RealMatrix predict(RealMatrix input) {

        /* the initial input MUST BE DEEP COPIED or is overwritten */
        RealMatrix layerInput = input.copy();

        for (NetworkLayer layer : layers) {
            layer.setInput(layerInput);

            /* calc the output and set to next layer input*/
            RealMatrix output = layer.getOutput(layerInput);
            layer.setOutput(output);
            /*
                does not need a deep copy, but be aware that
                every layer input shares memory of prior layer output
            */
            layerInput = output;
        }
        /* layerInput is holding the final output ... get a deep copy */
        return layerInput.copy();
    }

    @Override
    protected void update(RealMatrix input, RealMatrix target,
                          RealMatrix output) {

        /* gradient of the network error and starts the back prop process */
        RealMatrix layerError = getLossFunction()
                                   .getLossGradient(output, target).copy();

        /* create list iterator and set cursor to last! */
        ListIterator li = layers.listIterator(layers.size());

        while (li.hasPrevious()) {
            NetworkLayer layer = (NetworkLayer) li.previous();
            /* get error input from higher layer */
            layer.setOutputError(layerError);
            /* this back propagates the error and updates weights */
            layer.update();
            /* pass along error to next layer down */
            layerError = layer.getInputError();
        }
    }
}
```

MNIST EXAMPLE

MNIST, the classic handwritten digits dataset, is often used for testing learning algorithms. Here we get 94 percent accuracy by using a simple network with two hidden layers:

```
MNIST mnist = new MNIST();

DeepNetwork network = new DeepNetwork();

/* input, hidden and output layers */
network.addLayer(new NetworkLayer(784, 500, new TanhOutputFunction(),
    new GradientDescentMomentum(0.0001, 0.95)));

network.addLayer(new NetworkLayer(500, 300, new TanhOutputFunction(),
    new GradientDescentMomentum(0.0001, 0.95)));

network.addLayer(new NetworkLayer(300, 10, new SoftmaxOutputFunction(),
    new GradientDescentMomentum(0.0001, 0.95)));

/* runtime parameters */
network.setLossFunction(new SoftMaxCrossEntropyLossFunction());
network.setMaxIterations(6000);
network.setTolerance(10E-9);
network.setBatchSize(100);

/* learn */
network.learn(mnist.trainingData, mnist.trainingLabels);

/* predict */
RealMatrix prediction = network.predict(mnist.testingData);

/* compute accuracy */
ClassifierAccuracy accuracy =
    new ClassifierAccuracy(prediction, mnist.testingLabels);

/* results */
network.isConverged(); // false
network.getNumIterations(); // 10000
network.getLoss(); // 0.00633
accuracy.getAccuracy(); // 0.94
```

PREV
4. Data Operations

NEXT
6. Hadoop MapReduce