Chapter 4. Data Operations

Now that we know how to input data into a useful data structure, we can operate on that data by using what we know about statistics and linear algebra. There are many operations we perform on data before we subject it to a learning algorithm. Often called *preprocessing*, this step comprises data cleaning, regularizing or scaling the data, reducing the data to a smaller size, encoding text values to numerical values, and splitting the data into parts for model training and testing. Often our data is already in one form or another (e.g., `List` or `double[][]`), and the learning routines we will use may take either or both of those formats. Additionally, a learning algorithm may need to know whether the labels are binary or multiclass or even encoded in some other way such as text. We need to account for this and prepare the data before it goes in the learning algorithm. The steps in this chapter can be part of an automated pipeline that takes raw data from the source and prepares it for either learning or prediction algorithms.

Transforming Text Data

Many learning and prediction algorithms require numerical input. One of the simplest ways to achieve this is by creating a vector space model in which we define a vector of known length and then assign a collection of text snippets (or even words) to a corresponding collection of vectors. The general process of converting text to vectors has many options and variations. Here we will assume that there exists a large body of text (corpus) that can be divided into sentences or lines (documents) that can in turn be divided into words (tokens). Note that the definitions of *corpus*, *document*, and *token* are user-definable.

Extracting Tokens from a Document

For each document, we want to extract all the tokens. Because there are many ways to approach this problem, we can create an interface with a method that takes in a document string and returns an array of `String` tokens:

```
public interface Tokenizer {
    String[] getTokens(String document);
}
```

The tokens may have lots of characters that are undesirable, such as punctuation, numbers, or other characters. Of course, this will entirely depend on your application. In this example, we are concerned only with the actual content of regular English words, so we can clean the tokens to accept only lowercase alphabetical characters. Including a variable for minimum token size enables us to skip words such as *a*, *or*, and *at*.

```
public class SimpleTokenizer implements Tokenizer {

    private final int minTokenSize;

    public SimpleTokenizer(int minTokenSize) {
        this.minTokenSize = minTokenSize;
    }
    public SimpleTokenizer() {
        this(0);
    }

    @Override
    public String[] getTokens(String document) {
        String[] tokens = document.trim().split("\\s+");
        List<String> cleanTokens = new ArrayList<>();
        for (String token : tokens) {
            String cleanToken = token.trim().toLowerCase()
              .replaceAll("[^A-Za-z\']+", "");
            if(cleanToken.length() > minTokenSize) {
                cleanTokens.add(cleanToken);
            }
        }
        return cleanTokens.toArray(new String[0]);
    }
}
```

Utilizing Dictionaries

A *dictionary* is a list of terms that are relevant (i.e., a "vocabulary"). There is more than one strategy to implement a dictionary. The important feature is that each term needs to be associated with an integer value that corresponds to its location in a vector. Of course, this can be an array that is searched by position, but for large dictionaries, this is inefficient and a `Map` is better. For much larger dictionaries, we can skip the term storage and use the hashing trick. In general, we need to know the number of terms in the dictionary for creating vectors as well as a method that returns the index of particular term. Note that int cannot be `null`, so by using the boxed type `Integer`, a returned index can either be an `int` or `null` value.

```
public interface Dictionary {
    Integer getTermIndex(String term);
    int getNumTerms();
}
```

We can build a dictionary of the exact terms collected from the `Tokenizer` instance. Note that the strategy is to add a term and integer for each item. New items will increment the counter, and duplicates will be discarded without incrementing the counter. In this case, the `TermDictionary` class needs methods for adding new terms:

```
public class TermDictionary implements Dictionary {

    private final Map<String, Integer> indexedTerms;
    private int counter;

    public TermDictionary() {
        indexedTerms = new HashMap<>();
        counter = 0;
    }

    public void addTerm(String term) {
        if(!indexedTerms.containsKey(term)) {
            indexedTerms.put(term, counter++);
        }
    }

    public void addTerms(String[] terms) {
        for (String term : terms) {
            addTerm(term);
        }
    }

    @Override
    public Integer getTermIndex(String term) {
        return indexedTerms.get(term);
    }

    @Override
    public int getNumTerms() {
        return indexedTerms.size();
    }
}
```

For a large number of terms, we can use the hashing trick. Essentially, we use the hash code of the `String` value for each term and then take the modulo of the number of terms that will be in the dictionary. For a large number of terms (about 1 million), collisions are unlikely. Note that unlike with `TermDictionary`, we do not need to add terms or keep track of terms. Each term index is calculated on the fly. The number of terms is a constant that we set. For efficiency in hash table retrieval, it's a good idea to make the number of terms equal to 2^n. For around 2^{20}, it will be approximately 1 million terms.

```
public class HashingDictionary implements Dictionary {

    private int numTerms; // 2^n is optimal

    public HashingDictionary() {
        // 2^20 = 1048576
        this(new Double(Math.pow(2,20)).intValue());
    }

    public HashingDictionary(int numTerms) {
        this.numTerms = numTerms;
    }

    @Override
    public Integer getTermIndex(String term) {
        return Math.floorMod(term.hashCode(), numTerms);
    }

    @Override
    public int getNumTerms() {
        return numTerms;
    }
}
```

Vectorizing a Document

Now that we have a tokenizer and a dictionary, we can turn a list of words into numeric values that can be passed into machine-learning algorithms. The most straightforward way is to first decide on what the dictionary is, and then count the number of occurrences that are in the sentence (or text of interest). This is often called *bag of words*. In some cases, we want to know only whether a word occurred. In such a case, a 1 is placed in the vector as opposed to a count:

```
public class Vectorizer {

    private final Dictionary dictionary;
    private final Tokenizer tokenzier;
    private final boolean isBinary;

    public Vectorizer(Dictionary dictionary, Tokenizer tokenzier,
        boolean isBinary) {
        this.dictionary = dictionary;
        this.tokenzier = tokenzier;
        this.isBinary = isBinary;
    }

    public Vectorizer() {
        this(new HashingDictionary(), new SimpleTokenizer(), false);
    }

    public RealVector getCountVector(String document) {
        RealVector vector = new OpenMapRealVector(dictionary.getNumTerms());
        String[] tokens = tokenzier.getTokens(document);
        for (String token : tokens) {
            Integer index = dictionary.getTermIndex(token);
            if(index != null) {
                if(isBinary) {
                    vector.setEntry(index, 1);
                } else {
                    vector.addToEntry(index, 1); // increment !
                }
            }
        }
        return vector;
    }

    public RealMatrix getCountMatrix(List<String> documents) {
        int rowDimension = documents.size();
        int columnDimension = dictionary.getNumTerms();
        RealMatrix matrix = new OpenMapRealMatrix(rowDimension, columnDimension);
        int counter = 0;
        for (String document : documents) {
            matrix.setRowVector(counter++, getCountVector(document));
        }
        return matrix;
    }
}
```

In some cases, we will want to reduce the effects of common words. The term frequency—inverse document frequency (TFIDF) vector does just that. The TFIDF component is highest when a term occurs many times within a small number of documents but lowest when the term occurs in nearly all documents. Note that TFIDF is just term frequency times the inverse document frequency: $TFIDF = TF \times IDF$. Here TF is the number of times a term has appeared in a document (its count vector). IDF is the (pseudo)inverse of the document frequency, DF, the number of documents the term has appeared in. In general, we can compute TF by counting terms per document, and DF by computing a binary vector over each document and cumulatively summing those vectors as we process each document. The most common form of the TFIDF is shown here, where N is the total number of documents processed:

$$TFIDF_{t,d} = TF_{t,d} \log \left(N / DF_t \right)$$

This is just one strategy for TFIDF. Note that the log function will cause trouble if either N or DF has zero values. Some strategies avoid this by adding in small factors or 1. We can handle it in our implementation by setting `log(0)` to 0. In general, our implementation here is to first create a matrix of counts and then operate over that matrix, converting each term into its weighted TFIDF value. Because these matrices are usually sparse, it's a good idea to use the optimized order-walking operator:

```
public class TFIDF implements RealMatrixChangingVisitor {

    private final int numDocuments;
    private final RealVector termDocumentFrequency;
    double logNumDocuments;

    public TFIDF(int numDocuments, RealVector termDocumentFrequency) {
        this.numDocuments = numDocuments;
        this.termDocumentFrequency = termDocumentFrequency;
        this.logNumDocuments = numDocuments > 0 ? Math.log(numDocuments) : 0;
    }

    @Override
    public void start(int rows, int columns, int startRow, int endRow,
            int startColumn, int endColumn) {
        //NA
    }

    @Override
    public double visit(int row, int column, double value) {
        double df = termDocumentFrequency.getEntry(column);
        double logDF = df > 0 ? Math.log(df) : 0.0;
        // TFIDF = TF_i * log(N/DF_i) = TF_i * ( log(N) - log(DF_i) )
        return value * (logNumDocuments - logDF);
    }

    @Override
    public double end() {
        return 0.0;
    }

}
```

Then `TFIDFVectorizer` uses both counts and binary counts:

```
public class TFIDFVectorizer {

    private Vectorizer vectorizer;
    private Vectorizer binaryVectorizer;
    private int numTerms;

    public TFIDFVectorizer(Dictionary dictionary, Tokenizer tokenzier) {
        vectorizer = new Vectorizer(dictionary, tokenzier, false);
        binaryVectorizer = new Vectorizer(dictionary, tokenzier, true);
        numTerms = dictionary.getNumTerms();
    }

    public TFIDFVectorizer() {
       this(new HashingDictionary(), new SimpleTokenizer());
    }

    public RealVector getTermDocumentCount(List<String> documents) {
        RealVector vector = new OpenMapRealVector(numTerms);
        for (String document : documents) {
            vector.add(binaryVectorizer.getCountVector(document));
        }
        return vector;
    }

    public RealMatrix getTFIDF(List<String> documents) {
        int numDocuments = documents.size();
        RealVector df = getTermDocumentCount(documents);
        RealMatrix tfidf = vectorizer.getCountMatrix(documents);
        tfidf.walkInOptimizedOrder(new TFIDF(numDocuments, df));
        return tfidf;
    }
}
```

Here’s an example using the sentiment dataset described in Appendix A:

```
/* sentiment data ... see appendix */
Sentiment sentiment = new Sentiment();

/* create a dictionary of all terms */
TermDictionary termDictionary = new TermDictionary();

/* need a basic tokenizer to parse text */
SimpleTokenizer tokenizer = new SimpleTokenizer();

/* add all terms in sentiment dataset to dictionary */
for (String document : sentiment.getDocuments()) {
    String[]tokens = tokenizer.getTokens(document);
    termDictionary.addTerms(tokens);
}

/* create of matrix of word counts for each sentence */
Vectorizer vectorizer = new Vectorizer(termDictionary, tokenizer, false);
RealMatrix counts = vectorizer.getCountMatrix(sentiment.getDocuments());

/* ... or create a binary counter */
Vectorizer binaryVectorizer = new Vectorizer(termDictionary, tokenizer, true);
RealMatrix binCounts = binaryVectorizer.getCountMatrix(sentiment.getDocuments());

/* ... or create a matrix TFIDF  */
TFIDFVectorizer tfidfVectorizer = new TFIDFVectorizer(termDictionary, tokenizer);
RealMatrix tfidf = tfidfVectorizer.getTFIDF(sentiment.getDocuments());
```

Scaling and Regularizing Numeric Data

Should we pull data from our classes or use the arrays as is? Our goal is to apply some transform of each element in the dataset such that $f(x_{i,j}) \rightarrow x^{*}_{i,j}$. There are two basic ways to scale data: either by column or row. For column scaling, we just need to collect the statistics for each column of data. In particular, we need the min, max, mean, and standard deviation. So if we add the entire dataset to a `MultivariateSummaryStatistics` instance, we will have all of that. In the other case of row scaling, we need to collect the L1 or L2 normalization of each row. We can store those in a `RealVector` instance, which can be sparse.

WARNING

If you scale the data to train the model, retain any mins, maxes, means, or standard deviations you have used! You must use the same technique, including the stored parameters, when transforming a *new* dataset that you will use for prediction. Note that if you are splitting data into Train/Validate/Test sets, then scale the training data and use those values (e.g., means) to scale the validation and test sets so they will be unbiased.

Scaling Columns

The general form for scaling columns is to use a `RealMatrixChangingVisitor` with precomputed column statistics passed into the constructor. As the operation visits each matrix entry, the appropriate column statistics can be utilized.

```
public class MatrixScalingOperator implements RealMatrixChangingVisitor {

    MultivariateSummaryStatistics mss;

    public MatrixScalingOperator(MultivariateSummaryStatistics mss) {
        this.mss = mss;
    }

    @Override
    public void start(int rows, int columns, int startRow, int endRow,
      int startColumn, int endColumn) {
        // nothing
    }

    @Override
    public double visit(int row, int column, double value) {
        // implement specific type here
    }

    @Override
    public double end() {
        return 0.0;
    }
}
```

MIN-MAX SCALING

Min-max scaling ensures that the smallest value is 0 and the largest value is 1 for each column independently. We can transform each element *i* with min and max for that column *j*:

$$x_{i,j}^{*} = \frac{x_{i,j} - x_j^{min}}{x_j^{max} - x_j^{min}}$$

This can be implemented as follows:

```
public class MatrixScalingMinMaxOperator implements RealMatrixChangingVisitor {
...
    @Override
    public double visit(int row, int column, double value) {
            double min = mss.getMin()[column];
            double max = mss.getMax()[column];
            return (value - min) / (max - min);
        }
    ...
    }
```

At times we want to specify the lower *a* and upper *b* limits (instead of 0 and 1). In this case, we calculate the 0:1 scaled data first and then apply a second round of scaling:

$$x_{i,j}^{a,b} = x_{i,j}^{*}(b - a) + a$$

CENTERING THE DATA

Centering the data around the mean value ensures that the average of the data column will be zero. However, there can still be extreme mins and maxes because they are unbounded. Every value in one column is translated by that column's mean:

$$x_{i,j}^{*} = x_{i,j} - \bar{x}_j$$

This can be implemented as follows:

```
@Override
public double visit(int row, int column, double value) {
    double mean = mss.getMean()[column];
    return value - mean;
}
```

UNIT NORMAL SCALING

Unit normal scaling is also known as a *z-score*. It rescales every data point in a column such that it is a member of unit normal distribution by centering it about the mean and dividing by the standard deviation. Each column will then have an average value of zero, and its distribution of values will mostly be smaller than 1, although as a distribution, this is not guaranteed because the values are unbounded.

$$x_{i,j}^{*} = \frac{x_{i,j} - \bar{x}_j}{\sigma_j}$$

This can be implemented as follows:

```
@Override
public double visit(int row, int column, double value) {
    double mean = mss.getMean()[column];
    double std = mss.getStandardDeviation()[column];
    return (value - mean) / std;
}
```

Scaling Rows

When each row of data is a record across all variables, scaling by row is typically to perform an L1 or L2 regularization:

```
public class MatrixScalingOperator implements RealMatrixChangingVisitor {

    RealVector normals;

    public MatrixScalingOperator(RealVector normals) {
        this.normals = normals;
    }

    @Override
    public void start(int rows, int columns, int startRow, int endRow,
       int startColumn, int endColumn) {
        // nothing
    }

    @Override
    public double visit(int row, int column, double value) {
        //implement
    }

    @Override
    public double end() {
        return 0.0;
    }
}
```

L1 REGULARIZATION

In this case, we are normalizing each row of data such that the sum of (absolute) values is equal to 1, because we divide each element *j* of row *i* by the row L1 normal:

$$x^*_{i,j} = \frac{x_{i,j}}{|\mathbf{x_i}|}$$

We implement this as follows:

```
@Override
public double visit(int row, int column, double value) {
    double rowNormal = normals.getEntry(row);
    return ( rowNormal > 0 ) ? value / rowNormal : 0;
}
```

L2 REGULARIZATION

L2 regularization scales by row, not column. In this case, we are normalizing each row of data as we divide each element *j* of row *i* by the row L2 normal. The length of each row will now be equal to 1:

$$x^*_{i,j} = \frac{x_{i,j}}{\|\mathbf{x_i}\|}$$

We implement this with the following:

```
@Override
public double visit(int row, int column, double value) {
    double rowNormal = normals.getEntry(row);
    return ( rowNormal > 0 ) ? value / rowNormal : 0;
}
```

Matrix Scaling Operator

We can collect the scaling algorithms in static methods because we are altering the matrix in place:

```
public class MatrixScaler {

    public static void minmax(RealMatrix matrix) {
        MultivariateSummaryStatistics mss = getStats(matrix);
        matrix.walkInOptimizedOrder(new MatrixScalingMinMaxOperator(mss));
    }

    public static void center(RealMatrix matrix) {
        MultivariateSummaryStatistics mss = getStats(matrix);
        matrix.walkInOptimizedOrder(
          new MatrixScalingOperator(mss, MatrixScaleType.CENTER));
    }

    public static void zscore(RealMatrix matrix) {
        MultivariateSummaryStatistics mss = getStats(matrix);
        matrix.walkInOptimizedOrder(
          new MatrixScalingOperator(mss, MatrixScaleType.ZSCORE));
    }

    public static void l1(RealMatrix matrix) {
        RealVector normals = getL1Normals(matrix);
        matrix.walkInOptimizedOrder(
          new MatrixScalingOperator(normals, MatrixScaleType.L1));
    }

    public static void l2(RealMatrix matrix) {
        RealVector normals = getL2Normals(matrix);
        matrix.walkInOptimizedOrder(
          new MatrixScalingOperator(normals, MatrixScaleType.L2));
    }

    private static RealVector getL1Normals(RealMatrix matrix) {
        RealVector normals = new OpenMapRealVector(matrix.getRowDimension());
        for (int i = 0; i < matrix.getRowDimension(); i++) {
            double l1Norm = matrix.getRowVector(i).getL1Norm();
            if (l1Norm > 0) {
                normals.setEntry(i, l1Norm);
            }
        }
        return normals;
    }

    private static RealVector getL2Normals(RealMatrix matrix) {
        RealVector normals = new OpenMapRealVector(matrix.getRowDimension());
        for (int i = 0; i < matrix.getRowDimension(); i++) {
            double l2Norm = matrix.getRowVector(i).getNorm();
            if (l2Norm > 0) {
                normals.setEntry(i, l2Norm);
            }
        }
        return normals;
    }

    private static MultivariateSummaryStatistics getStats(RealMatrix matrix) {
        MultivariateSummaryStatistics mss =
        new MultivariateSummaryStatistics(matrix.getColumnDimension(), true);
        for (int i = 0; i < matrix.getRowDimension(); i++) {
            mss.addValue(matrix.getRow(i));
        }
        return mss;
    }
}
```

Now it is really easy to use it:

```
RealMatrix matrix = new OpenMapRealMatrix(10, 3);
        matrix.addToEntry(0, 0, 1.0);
        matrix.addToEntry(0, 2, 2.0);
        matrix.addToEntry(1, 0, 1.0);
        matrix.addToEntry(2, 0, 3.0);
        matrix.addToEntry(3, 1, 5.0);
        matrix.addToEntry(6, 2, 1.0);
        matrix.addToEntry(8, 0, 8.0);
        matrix.addToEntry(9, 1, 3.0);

/* scale matrix in-place */
MatrixScaler.minmax(matrix);
```

Reducing Data to Principal Components

The goal of a *principal components analysis* (PCA) is to transform a dataset into another dataset with fewer dimensions. We can envision this as applying a function *f* to an $m \times n$ matrix X such that the result will be an $m \times k$ matrix X_k, where $k < n$:

$$\mathbf{X}_k = f(\mathbf{X})$$

This is achieved by finding the eigenvectors and eigenvalues via linear algebra algorithms. One benefit of this type of transformation is that the new dimensions are ordered from most significant to least. For multidimensional data, we can sometimes gain insight into any significant relationships by plotting the first two dimensions of the principal components. In Figure 4-1, we have plotted the first two principal components of the Iris dataset (see Appendix A). The Iris dataset is a four-dimensional set of features with three possible labels. In this image, we note that by plotting the original data projected onto the first two principal components, we can see a separation of the three classes. This distinction does not occur when plotting any of the two dimensions from the original dataset.

However, for high-dimensional data, we need a more robust way of determining the number of principal components to keep. Because the principal components are ordered from most significant to least, we can formulate the explained variance of the principal components by computing the normalized, cumulative sum of the eigenvalues λ:

$$\sigma^2(k) = \frac{1}{\sigma^2_{max}} \sum_{i=1}^{k} \lambda_i$$

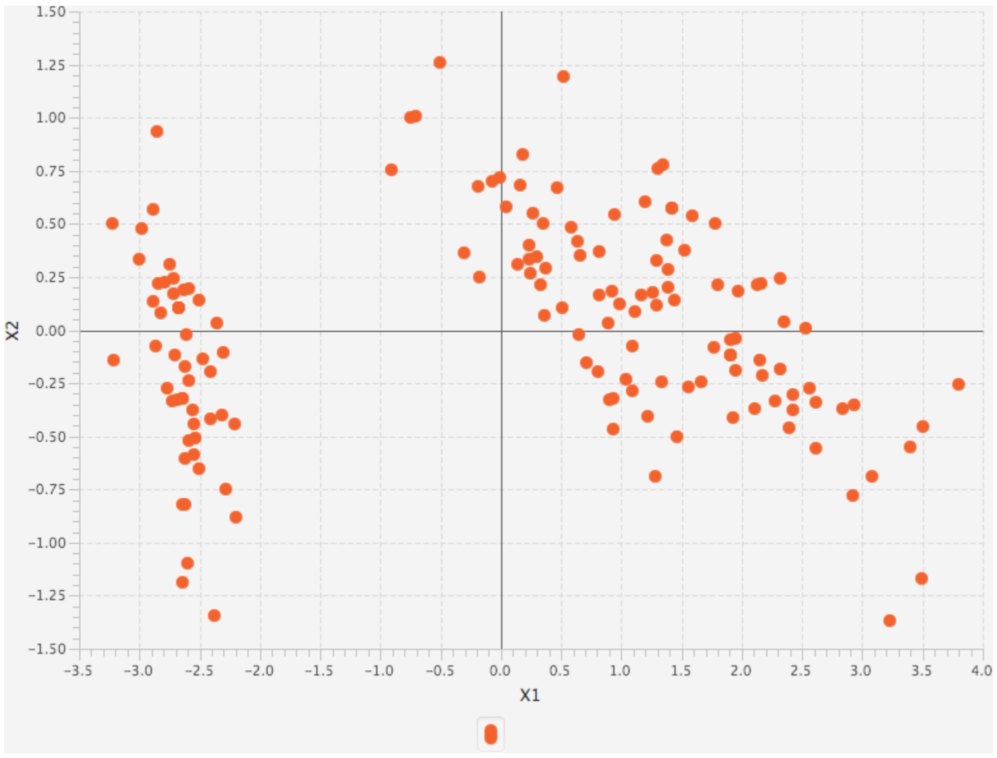

Figure 4-1. IRIS data's first two principal components

Here, each additional component explains an additional percentage of the data. There are then two uses for the explained variance. When we explicitly choose a number of principal components, we can calculate how much of the original dataset is explained by this new transformation. In the other case, we can iterate through the explained variance vector and stop at a particular number of components, *k*, when we have reached a desired coverage.

When implementing a principal components analysis, there are several strategies for computing the eigenvalues and eigenvectors. In the end, we just want to retrieve the transformed data. This is a great case for the strategy pattern in which the implementation details can be contained in separate classes, while the main PCA class is mostly just a shell:

```
public class PCA {

    private final PCAImplementation pCAImplementation;

    public PCA(RealMatrix data, PCAImplementation pCAImplementation) {
        this.pCAImplementation = pCAImplementation;
        this.pCAImplementation.compute(data);
    }

    public RealMatrix getPrincipalComponents(int k) {
        return pCAImplementation.getPrincipalComponents(k);
    }

    public RealMatrix getPrincipalComponents(int k, RealMatrix otherData) {
        return pCAImplementation.getPrincipalComponents(k, otherData);
    }

    public RealVector getExplainedVariances() {
        return pCAImplementation.getExplainedVariances();
    }

    public RealVector getCumulativeVariances() {
        RealVector variances = getExplainedVariances();
        RealVector cumulative = variances.copy();
        double sum = 0;
        for (int i = 0; i < cumulative.getDimension(); i++) {
            sum += cumulative.getEntry(i);
            cumulative.setEntry(i, sum);
        }
        return cumulative;
    }

    public int getNumberOfComponents(double threshold) {
        RealVector cumulative = getCumulativeVariances();
        int numComponents=1;
        for (int i = 0; i < cumulative.getDimension(); i++) {
            numComponents = i + 1;
            if(cumulative.getEntry(i) >= threshold) {
                break;
            }
        }
        return numComponents;
    }

    public RealMatrix getPrincipalComponents(double threshold) {
        int numComponents = getNumberOfComponents(threshold);
        return getPrincipalComponents(numComponents);
    }

    public RealMatrix getPrincipalComponents(double threshold,
        RealMatrix otherData) {
        int numComponents = getNumberOfComponents(threshold);
        return getPrincipalComponents(numComponents, otherData);
    }

}
```

We can then provide an interface `PCAImplementation` for the following methods of decomposing the input data into its principal components:

```
public interface PCAImplementation {

    void compute(RealMatrix data);

    RealVector getExplainedVariances();

    RealMatrix getPrincipalComponents(int numComponents);

    RealMatrix getPrincipalComponents(int numComponents, RealMatrix otherData);
}
```

Covariance Method

One method for calculating the PCA is by finding the eigenvalue decomposition of the covariance matrix of **X**. The principal components of a centered matrix **X** are the eigenvectors of the covariance:

$$\mathbf{C} = \frac{1}{n-1}(\mathbf{X} - \overline{\mathbf{X}})^T(\mathbf{X} - \overline{\mathbf{X}})$$

This method of covariance calculation can be computationally intensive because it requires multiplying together two potentially large matrices. However, in Chapter 3, we explored an efficient update formula for computing covariance that does not require matrix transposition. When using the Apache Commons Math class `Covariance`, or other classes that implement it (e.g., `MultivariateSummaryStatistics`), the efficient update formula is used. Then the covariance **C** can be decomposed into the following:

$$\mathbf{C} = \mathbf{V}\mathbf{D}\mathbf{V}^T$$

The columns of **V** are the eigenvectors, and the diagonal components of **D** are the eigenvalues. The Apache Commons Math implementation orders the eigenvalues (and corresponding eigenvectors) from largest to smallest. Typically, we want only *k* components, and therefore we need only the first *k* columns of **V**. The mean-centered data can be projected onto the new components with a matrix multiplication:

$$\mathbf{X}_k = (\mathbf{X} - \bar{\mathbf{X}})\mathbf{V}_k$$

Here is an implementation of a principal components analysis using the covariance method:

```java
public class PCAEIGImplementation implements PCAImplementation {

    private RealMatrix data;
    private RealMatrix d; // eigenvalue matrix
    private RealMatrix v; // eigenvector matrix
    private RealVector explainedVariances;
    private EigenDecomposition eig;
    private final MatrixScaler matrixScaler;

    public PCAEIGImplementation() {
        matrixScaler = new MatrixScaler(MatrixScaleType.CENTER);
    }

    @Override
    public void compute(RealMatrix data) {
        this.data = data;
        eig = new EigenDecomposition(new Covariance(data).getCovarianceMatrix());
        d = eig.getD();
        v = eig.getV();
    }

    @Override
    public RealVector getExplainedVariances() {
        int n = eig.getD().getColumnDimension(); //colD = rowD
        explainedVariances = new ArrayRealVector(n);
        double[] eigenValues = eig.getRealEigenvalues();
        double cumulative = 0.0;
        for (int i = 0; i < n; i++) {
            double var = eigenValues[i];
            cumulative += var;
            explainedVariances.setEntry(i, var);
        }
        /* dividing the vector by the last (highest) value maximizes to 1 */
        return explainedVariances.mapDivideToSelf(cumulative);
    }

    @Override
    public RealMatrix getPrincipalComponents(int k) {
        int m = eig.getV().getColumnDimension(); // rowD = colD
        matrixScaler.transform(data);
        return data.multiply(eig.getV().getSubMatrix(0, m-1, 0, k-1));
    }

    @Override
    public RealMatrix getPrincipalComponents(int numComponents,
        RealMatrix otherData) {
        int numRows = v.getRowDimension();
        // NEW data transformed under OLD means
        matrixScaler.transform(otherData);
        return otherData.multiply(
            v.getSubMatrix(0, numRows-1, 0, numComponents-1));
    }
}
```

Then it can be used, for example, to get the first three principal components, or to get all the components that provide 50 percent explained variance:

```
/* use the eigenvalue decomposition implementation */
PCA pca = new PCA(data, new PCAEIGImplementation());

/* get first three components */
RealMatrix pc3 = pca.getPrincipalComponents(3);

/* get however many components are needed to satisfy 50% explained variance */
RealMatrix pct = pca.getPrincipalComponents(.5);
```

SVD Method

If $\mathbf{X} - \bar{\mathbf{X}}$ is a mean-centered dataset with *m* rows and *n* columns, the principal components are calculated from the following:

$$\mathbf{X} - \bar{\mathbf{X}} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$$

Note the familiar form for a singular value decomposition, $\mathbf{A} = \mathbf{U\Sigma V}^T$, in which the column vectors of **V** are the eigenvectors, and the eigenvalues are derived from the diagonal of **Σ** via $\lambda_i = \Sigma_{i,i}^2 / (m-1)$; *m* is the number of rows of data. After performing the singular value decomposition on the mean-centered **X**, the projection is then as follows:

$$\mathbf{X}_k = \mathbf{U}_k\mathbf{\Sigma}_k$$

We've kept only the first *k* columns of **U** and the $k \times k$ upper-left submatrix of Σ. It is also correct to compute the projection with the original, mean-centered data and the eigenvectors:

$$\mathbf{X}_k = (\mathbf{X} - \bar{\mathbf{X}})\mathbf{V}_k$$

Here we keep only the first *k* columns of **V**. In particular, this expression is used when we are transforming a new set of data with the existing eigenvectors and means. Note that the means are those that the PCA was trained on, not the means of the input data. This is the same form as in the eigenvalue method in the preceding section.

Apache Commons Math implementation is *compact SVD* because there are at most *p = min(m,n)* singular values, so there is no need to calculate the full SVD as discussed in Chapter 2. Following is the SVD implementation of a principal components analysis and is the preferred method:

```
public class PCASVDImplementation implements PCAImplementation {

    private RealMatrix u;
    private RealMatrix s;
    private RealMatrix v;
    private MatrixScaler matrixScaler;
    private SingularValueDecomposition svd;

    @Override
    public void compute(RealMatrix data) {
        MatrixScaler.center(data);
        svd = new SingularValueDecomposition(data);
        u = svd.getU();
        s = svd.getS();
        v = svd.getV();
    }

    @Override
    public RealVector getExplainedVariances() {
        double[] singularValues = svd.getSingularValues();
        int n = singularValues.length;
        int m = u.getRowDimension(); // number of rows in U is same as in data
        RealVector explainedVariances = new ArrayRealVector(n);
        double sum = 0.0;
        for (int i = 0; i < n; i++) {
            double var = Math.pow(singularValues[i], 2) / (double)(m-1);
            sum += var;
            explainedVariances.setEntry(i, var);
        }
        /* dividing the vector by the last (highest) value maximizes to 1 */
        return explainedVariances.mapDivideToSelf(sum);

    }

    @Override
    public RealMatrix getPrincipalComponents(int numComponents) {
        int numRows = svd.getU().getRowDimension();
        /* submatrix limits are inclusive */
        RealMatrix uk = u.getSubMatrix(0, numRows-1, 0, numComponents-1);
        RealMatrix sk = s.getSubMatrix(0, numComponents-1, 0, numComponents-1);
        return uk.multiply(sk);
    }

    @Override
    public RealMatrix getPrincipalComponents(int numComponents,
        RealMatrix otherData) {
        // center the (new) data on means from original data
        matrixScaler.transform(otherData);
        int numRows = v.getRowDimension();
        // subMatrix indices are inclusive
        return otherData.multiply(v.getSubMatrix(0, numRows-1, 0, numComponents-1));
    }
}
```

Then to implement it, we use the following:

```
/* use the singular value decomposition implementation */
PCA pca = new PCA(data, new PCASVDImplementation());

/* get first three components */
RealMatrix pc3 = pca.getPrincipalComponents(3);

/* get however many components are needed to satisfy 50% explained variance */
RealMatrix pct = pca.getPrincipalComponents(.5);
```

Creating Training, Validation, and Test Sets

For supervised learning, we build models on one part of the dataset, and then make a prediction using the test set and see whether we were right (using the known labels of the test set). Sometimes we need a third set during the training process for validating model parameters, called the *validation set*.

The training set is used to train the model, whereas the validation set is used for model selection. A test set is used once at the very end to calculate the model error. We have at least two options. First, we can sample random integers and pick lines out of an array or matrix. Second, we can reshuffle the data itself as a `List` and pull off the sublists of length we need for each type of set.

Index-Based Resampling

Create an index for each point in the dataset:

```
public class Resampler {

    RealMatrix features;
    RealMatrix labels;
    List<Integer> indices;
    List<Integer> trainingIndices;
    List<Integer> validationIndices;
    List<Integer> testingIndices;
    int[] rowIndices;
    int[] test;
    int[] validate;
    public Resampler(RealMatrix features, RealMatrix labels) {
        this.features = features;
        this.labels = labels;
        indices = new ArrayList<>();
    }

    public void calculateTestTrainSplit(double testFraction, long seed) {
        Random rnd = new Random(seed);
        for (int i = 1; i <= features.getRowDimension(); i++) {
            indices.add(i);
        }
        Collections.shuffle(indices, rnd);
        int testSize = new Long(Math.round(
        testFraction * features.getRowDimension())).intValue();
        /* subList has inclusive fromIndex and exclusive toIndex */
        testingIndices = indices.subList(0, testSize);
        trainingIndices = indices.subList(testSize, features.getRowDimension());
    }

    public RealMatrix getTrainingFeatures() {
        int numRows = trainingIndices.size();
        rowIndices = new int[numRows];
        int counter = 0;
        for (Integer trainingIndex : trainingIndices) {
            rowIndices[counter] = trainingIndex;
        }
        counter++;
        int numCols = features.getColumnDimension();
        int[] columnIndices = new int[numCols];
        for (int i = 0; i < numCols; i++) {
            columnIndices[i] = i;
        }
        return features.getSubMatrix(rowIndices, columnIndices);
    }
}
```

Here is an example using the Iris dataset:

```
Iris iris = new Iris();

Resampler resampler = new Resampler(iris.getFeatures(), iris.getLabels());
resampler.calculateTestTrainSplit(0.40, 0L);

RealMatrix trainFeatures = resampler.getTrainingFeatures();
RealMatrix trainLabels = resampler.getTrainingLabels();
RealMatrix testFeatures = resampler.getTestingFeatures();
RealMatrix testLabels = resampler.getTestingLabels();
```

List-Based Resampling

In some cases, we may have defined our data as a collection of objects. For example, we may a have `List` of type `Record` that holds the data for each record (row) of data. It is straightforward then to build a `List`-based resampler that takes a generic type `T`:

```
public class Resampling<T> {

    private final List<T> data;
    private final int trainingSetSize;
    private final int testSetSize;
    private final int validationSetSize;

    public Resampling(List<T> data, double testFraction, long seed) {
        this(data, testFraction, 0.0, seed);
    }

    public Resampling(List<T> data, double testFraction,
    double validationFraction, long seed) {
        this.data = data;
        validationSetSize = new Double(
            validationFraction * data.size()).intValue();
        testSetSize = new Double(testFraction * data.size()).intValue();
        trainingSetSize = data.size() - (testSetSize + validationSetSize);
        Random rnd = new Random(seed);
        Collections.shuffle(data, rnd);
    }

    public int getTestSetSize() {
        return testSetSize;
    }

    public int getTrainingSetSize() {
        return trainingSetSize;
    }

    public int getValidationSetSize() {
        return validationSetSize;
    }

    public List<T> getValidationSet() {
        return data.subList(0, validationSetSize);
    }

    public List<T> getTestSet() {
        return data.subList(validationSetSize, validationSetSize + testSetSize);
    }

    public List<T> getTrainingSet() {
        return data.subList(validationSetSize + testSetSize, data.size());
    }
}
```

Given a predefined class `Record`, we can use the resampler like this:

```
Resampling<Record> resampling = new Resampling<>(data, 0.20, 0L);
//Resampling<Record> resampling = new Resampling<>(data, 0.20, 0.20, 0L);
List<Record> testSet = resampling.getTestSet();
List<Record> trainingSet = resampling.getTrainingSet();
List<Record> validationSet = resampling.getValidationSet();
```

Mini-Batches

In several learning algorithms, it is advantageous to input small batches of data (on the order of 100 data points) randomly sampled from a much larger dataset. We can reuse the code from our `MatrixResampler` for this task. The important thing to remember is that when designating batch size, we are specifically implying the test set, not the training set, as implemented in the `MatrixResampler`:

```
public class Batch extends MatrixResampler {

    public Batch(RealMatrix features, RealMatrix labels) {
        super(features, labels);
    }

    public void calcNextBatch(int batchSize) {
        super.calculateTestTrainSplit(batchSize);
    }

    public RealMatrix getInputBatch() {
        return super.getTestingFeatures();
    }

    public RealMatrix getTargetBatch() {
        return super.getTestingLabels();
    }
}
```

Encoding Labels

When labels arrive to us as a text field, such as *red* or *blue*, we convert them to integers for further processing.

NOTE

When dealing with classification algorithms, we refer to each unique instance of the outcome's variables as a *class*. Recall that `class` is a Java keyword, and we will have to use other terms instead, such as `className`, `classLabel`, or `classes` for plural. When using `classes` for the name of a `List`, be aware of your IDE's code completion when building a for...each loop.

A Generic Encoder

Here is an implementation of a label encoder for a generic type `T`. Note that this system creates classes starting at 0 through *n* - 1 classes. In other words, the resulting class is the position in the `ArrayList`:

```
public class LabelEncoder<T> {

    private final List<T> classes;

    public LabelEncoder(T[] labels) {
        classes = Arrays.asList(labels);
    }

    public List<T> getClasses() {
        return classes;
    }

    public int encode(T label) {
        return classes.indexOf(label);
    }

    public T decode(int index) {
        return classes.get(index);
    }
}
```

Here is an example of how you might use label encoding with real data:

```
String[] stringLabels = {"Sunday", "Monday", "Tuesday"};

LabelEncoder<String> stringEncoder = new LabelEncoder<>(stringLabels);

/* note that classes are in the order of the original String array */
System.out.println(stringEncoder.getClasses()); //[Sunday, Monday, Tuesday]

for (Datum datum : data) {
    int classNumber = stringEncoder.encode(datum.getLabel);
    // do something with classes i.e. add to List or Matrix
}
```

Note that in addition to `String` types, this also works for any of the boxed types, but most likely your labels will take on values suitable for `Short`, `Integer`, `Long`, `Boolean`, and `Character`. For example, `Boolean` labels could be true/false bools, `Character` could be Y/N for yes/no or M/F for male/female or even T/F for true/false. It all depends on how someone else originally coded the labels in the data file you are reading from. Labels are unlikely to be in the form of a floating-point number. If this is the case, you probably have a regression problem instead of a classification problem (that is, you are mistakenly confusing a continuous variable for a discrete one). An example using `Integer` type labels is shown in the next section.

One-Hot Encoding

In some cases, it will be more efficient to convert a multinomial label into a multivariate binomial. This is analogous to converting an integer to binary form, except that we have the requirement that only one position can be *hot* (equal to 1) at a time. For example, we can encode three string labels as integers, or represent each string as a position in a binary string:

```
Sunday  0  100
Monday  1  010
Tuesday 2  001
```

When using a `List` for encoding the labels, we use the following:

```
public class OneHotEncoder {

    private int numberOfClasses;

    public OneHotEncoder(int numberOfClasses) {
        this.numberOfClasses = numberOfClasses;
    }

    public int getNumberOfClasses() {
        return numberOfClasses;
    }

    public int[] encode(int label) {
        int[] oneHot = new int[numberOfClasses];
        oneHot[label] = 1;
        return oneHot;
    }

    public int decode(int[] oneHot) {
        return Arrays.binarySearch(oneHot, 1);
    }
}
```

In the case where the labels are strings, first encode the labels into integers by using a `LabelEncoder` instance, and then convert the integer labels to one hot by using a `OneHotEncoder` instance.

```
String[] stringLabels = {"Sunday", "Monday", "Tuesday"};

LabelEncoder<String> stringEncoder = new LabelEncoder<>(stringLabels);

int numClasses = stringEncoder.getClasses.size();

OneHotEncoder oneHotEncoder = new oneHotEncoder(numClasses);

for (Datum datum : data) {
    int classNumber = stringEncoder.encode(datum.getLabel);
    int[] oneHot    = oneHotEncoder.encode(classNumber);
    // do something with classes i.e. add to List or Matrix
}
```

Then what about the reverse? Say we have a predictive model that returns the classes we have designated in a learning process. (Usually, a learning process outputs probabilities, but we can assume that we have converted those to classes here.) First we need to convert the one-hot output to its class. Then we need to convert the class back to the original label, as shown here:

```
[1, 0, 0]
[0, 0, 1]
[1, 0, 0]
[0, 1, 0]
```

Then we need to convert output predictions from one hot:

```
for(Integer[] prediction: predictions) {
    int classLabel = oneHotEncoder.decode(prediction);
    String label = labelEncoder.decode(classLabel);
}

// predicted labels are Sunday, Tuesday, Sunday, Monday
```

PREV
3. Statistics

NEXT
5. Learning and Prediction