Machine Learning in Java

As the amount of data continues to grow at an incomprehensible rate, the main challenge is how to transform data into actionable knowledge.

Machine Learning in Java will provide you with the techniques and tools you need to quickly gain insight from complex data. You will start by learning how to apply machine learning methods to a variety of common tasks, including classification, prediction, forecasting, market basket analysis, and clustering. Moving on, you will discover how to detect anomalies and fraud, and ways to perform activity recognition, image recognition, and text analysis. By the end of the book, you will explore related web resources and technologies that will help you take your learning to the next level.

By applying the most effective machine learning methods to real-world problems, you will gain hands-on experience that will transform the way you think about data.

Who this book is written for

If you want to learn how to use Java's machine learning libraries to gain insight from your data, this book is for you. You should be familiar with Java programming and data mining concepts, but no prior experience with data mining packages is necessary.

What you will learn from this book

- Understand the basic steps of applied machine learning and how to differentiate between various machine learning approaches
- Discover key Java machine learning libraries, what each library brings to the table, and what kinds of problem each are able to solve
- Learn how to implement classification, regression, and clustering
- Develop a sustainable strategy for customer retention by predicting likely churn candidates
- Build a scalable recommendation engine with Apache Mahout
- Apply machine learning to fraud, anomaly, and outlier detection
- Experiment with deep learning concepts, algorithms, and the toolbox for deep learning
- Write your own activity recognition model for eHealth applications using mobile sensors


7.50 x 9.25
235 mm x 191 mm

7.50 x 9.25
250 mm x 191 mm

Content Type: Black & White
Paper Type: White
Page Count: 258
File Type: InDesign
Request ID: CSS1592189

Boštjan Kaluža

Design, build, and deploy your own machine learning applications by leveraging key Java machine learning libraries.
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 7 'Fraud and Anomaly Detection'
- A synopsis of the book’s content
- More information on Machine Learning in Java
About the Author

Boštjan Kaluža, PhD, is a researcher in artificial intelligence and machine learning. Boštjan is the chief data scientist at Evolven, a leading IT operations analytics company, focusing on configuration and change management. He works with machine learning, predictive analytics, pattern mining, and anomaly detection to turn data into understandable relevant information and actionable insight.

Prior to Evolven, Boštjan served as a senior researcher in the department of intelligent systems at the Jozef Stefan Institute, a leading Slovenian scientific research institution, and led research projects involving pattern and anomaly detection, ubiquitous computing, and multi-agent systems. Boštjan was also a visiting researcher at the University of Southern California, where he studied suspicious and anomalous agent behavior in the context of security applications. Boštjan has extensive experience in Java and Python, and he also lectures on Weka in the classroom.

Focusing on machine learning and data science, Boštjan has published numerous articles in professional journals, delivered conference papers, and authored or contributed to a number of patents. In 2013, Boštjan published his first book on data science, Instant Weka How-to, Packt Publishing, exploring how to leverage machine learning using Weka. Learn more about him at http://bostjankaluza.net.
Machine learning is a subfield of artificial intelligence. It helps computers to learn and act like human beings with the help of algorithms and data. With a given set of data, an ML algorithm learns different properties of the data and infers the properties of the data that it may encounter in future.

This book will teach the readers how to create and implement machine learning algorithms in Java by providing fundamental concepts as well as practical examples. In this process, it will also talk about some machine learning libraries that are frequently used, such as Weka, Apache Mahout, Mallet, and so on. This book will help the user to select appropriate approaches for particular problems and compare and evaluate the results of different techniques. This book will also cover performance improvement techniques, including input preprocessing and combining output from different methods.

Without shying away from the technical details, you will explore machine learning with Java libraries using clear and practical examples. You will also explore how to prepare data for analysis, choose a machine learning method, and measure the success of the process.

What this book covers

Chapter 1, Applied Machine Learning Quick Start, introduces the basics of machine learning, laying down the common concepts, machine learning principles, and applied machine learning workflow.

Chapter 2, Java Libraries and Platforms for Machine Learning, reviews the various Java libraries and platforms dedicated to machine learning, what each library brings to the table, and what kind of problems it is able to solve. The review includes Weka, Java-ML, Apache Mahout, Apache Spark, deeplearning4j, and Mallet.
Chapter 3, Basic Algorithms – Classification, Regression, and Clustering, starts with basic machine learning tasks, introducing the key algorithms for classification, regression, and clustering, using small, easy-to-understand datasets.

Chapter 4, Customer Relationship Prediction with Ensembles, dives into a real-world marketing database, where the task is to predict the customer that will churn, upsell, and cross-sell. The problem is attacked with ensemble methods, following the steps of KDD Cup-winning solution.

Chapter 5, Affinity Analysis, discusses how to analyze co-occurrence relationships using association rule mining. We will look into market basket analysis to understand the purchasing behavior of customers and discuss applications of the approach to other domains.

Chapter 6, Recommendation Engine with Apache Mahout, explains the basic concepts required to understand recommendation engine principles, followed by two applications leveraging Apache Mahout to build content-based filtering and collaborative recommender.

Chapter 7, Fraud and Anomaly Detection, introduces the background to anomalous and suspicious pattern detection, followed by two practical applications on detecting frauds in insurance claims and detecting anomalies in website traffic.

Chapter 8, Image Recognition with Deeplearning4j, introduces image recognition and reviews fundamental neural network architectures. We will then discuss how to implement various deep learning architectures with deeplearning4j library to recognize handwritten digits.

Chapter 9, Activity Recognition with Mobile Phone Sensors, tackles the problem of recognizing patterns from sensor data. This chapter introduces the activity recognition process, explains how to collect data with an Android device, and presents a classification model to recognize activities of daily living.

Chapter 10, Text Mining with Mallet – Topic Modeling and Spam Detection, explains the basics of text mining, introduces the text processing pipeline, and shows how to apply this to two real-world problems: topic modeling and document classification.

Chapter 11, What is Next?, concludes the book with practical advice about how to deploy models and gives you further pointers about where to find additional resources, materials, venues, and technologies to dive deeper into machine learning.
Fraud and Anomaly Detection

Outlier detection is used to identify exceptions, rare events, or other anomalous situations. Such anomalies may be hard-to-find needles in a haystack, but their consequences may nonetheless be quite dramatic, for instance, credit card fraud detection, identifying network intrusion, faults in a manufacturing processes, clinical trials, voting activities, and criminal activities in e-commerce. Therefore, discovered anomalies represent high value when they are found or high costs if they are not found. Applying machine learning to outlier detection problems brings new insight and better detection of outlier events. Machine learning can take into account many disparate sources of data and find correlations that are too obscure for human analysis to identify.

Take the example of e-commerce fraud detection. With machine learning algorithm in place, the purchaser's online behavior, that is, website browsing history, becomes a part of the fraud detection algorithm rather than simply considering the history of purchases made by the cardholder. This involves analyzing a variety of data sources, but it is also a far more robust approach to e-commerce fraud detection.

In this chapter, we will cover the following topics:

- Problems and challenges
- Suspicious pattern detection
- Anomalous pattern detection
- Working with unbalanced datasets
- Anomaly detection in time series
Suspicious and anomalous behavior detection

The problem of learning patterns from sensor data arises in many applications, including e-commerce, smart environments, video surveillance, network analysis, human-robot interaction, ambient assisted living, and so on. We focus on detecting patterns that deviate from regular behaviors and might represent a security risk, health problem, or any other abnormal behavior contingency.

In other words, deviant behavior is a data pattern that either does not conform to the expected behavior (anomalous behavior) or matches a previously defined unwanted behavior (suspicious behavior). Deviant behavior patterns are also referred to as outliers, exceptions, peculiarities, surprise, misuse, and so on. Such patterns relatively occur infrequently; however, when they do occur, their consequences can be quite dramatic, and often negatively so. Typical examples include credit card fraud detection, cyber-intrusions, and industrial damage. In e-commerce, fraud is estimated to cost merchants more than $200 billion a year; in healthcare, fraud is estimated to cost taxpayers $60 billion a year; for banks, the cost is over $12 billion.

Unknown-unknowns

When Donald Rumsfeld, US Secretary of Defense, had a news briefing on February 12, 2002, about the lack of evidence linking the government of Iraq to the supply of weapons of mass destruction to terrorist groups, it immediately became a subject of much commentary. Rumsfeld stated (DoD News, 2012):

"Reports that say that something hasn't happened are always interesting to me, because as we know, there are known knowns; there are things we know we know. We also know there are known unknowns; that is to say we know there are some things we do not know. But there are also unknown unknowns – the ones we don't know we don't know. And if one looks throughout the history of our country and other free countries, it is the latter category that tend to be the difficult ones."

The statement might seem confusing at first, but the idea of unknown unknowns was well studied among scholars dealing with risk, NSA, and other intelligence agencies. What the statement basically says is the following:

- **Known-knowns**: These are well-known problems or issues we know how to recognize them and how deal with them
- **Known-unknowns**: These are expected or foreseeable problems, which can be reasonably anticipated, but have not occurred before
• **Unknown-unknowns**: These are unexpected and unforeseeable problems, which pose significant risk as they cannot be anticipated, based on previous experience.

In the following sections, we will look into two fundamental approaches dealing with the first two types of knowns and unknowns: suspicious pattern detection dealing with known-knowns and anomalous pattern detection targeting known-unknowns.

**Suspicious pattern detection**

The first approach assumes a behavior library that encodes negative patterns shown as red minus signs in the following image, and thus recognizing that observed behavior corresponds to identifying a match in the library. If a new pattern (blue circle) can be matched against negative patterns, then it is considered suspicious:

For example, when you visit a doctor, she inspects various health symptoms (body temperature, pain levels, affected areas, and so on) and matches the symptoms to a known disease. In machine learning terms, the doctor collects attributes and performs classifications.

An advantage of this approach is that we immediately know what is wrong; for example, assuming we know the disease, we can select appropriate treatment procedure.

A major disadvantage of this approach is that it can detect only suspicious patterns that are known in advance. If a pattern is not inserted into a negative pattern library, then we will not be able to recognize it. This approach is, therefore, appropriate for modeling known-knowns.
Anomalous pattern detection

The second approach uses the pattern library in an inverse fashion, meaning that the library encodes only positive patterns marked with green plus signs in the following image. When an observed behavior (blue circle) cannot be matched against the library, it is considered anomalous:

![Pattern Library Example]

This approach requires us to model only what we have seen in the past, that is, normal patterns. If we return to the doctor example, the main reason we visited the doctor in the first place was because we did not feel fine. Our perceived state of feelings (for example, headache, sore skin) did not match our usual feelings, therefore, we decided to seek doctor. We don't know which disease caused this state nor do we know the treatment, but we were able to observe that it doesn't match the usual state.

A major advantage of this approach is that it does not require us to say anything about non-normal patterns; hence, it is appropriate for modeling known-unknowns and unknown-unknowns. On the other hand, it does not tell us what exactly is wrong.

Analysis types

Several approaches have been proposed to tackle the problem either way. We broadly classify anomalous and suspicious behavior detection in the following three categories: pattern analysis, transaction analysis, and plan recognition. In the following sections, we will quickly look into some real-life applications.

Pattern analysis

An active area of anomalous and suspicious behavior detection from patterns is based on visual modalities such as camera. Zhang et al (2007) proposed a system for a visual human motion analysis from a video sequence, which recognizes unusual behavior based on walking trajectories; Lin et al (2009) described a video surveillance system based on color features, distance features, and a count feature, where evolutionary techniques are used to measure observation similarity. The system tracks each person and classifies their behavior by analyzing their trajectory patterns. The system extracts a set of visual low-level features in different parts of the image, and performs a classification with SVMs to detect aggressive, cheerful, intoxicated, nervous, neutral, and tired behavior.
Transaction analysis

Transaction analysis assumes discrete states/transactions in contrast to continuous observations. A major research area is Intrusion Detection (ID) that aims at detecting attacks against information systems in general. There are two types of ID systems, signature-based and anomaly-based, that broadly follow the suspicious and anomalous pattern detection as described in the previous sections. A comprehensive review of ID approaches was published by Gyanchandani et al (2012).

Furthermore, applications in ambient-assisted living that are based on wearable sensors also fit to transaction analysis as sensing is typically event-based. Lymberopoulos et al (2008) proposed a system for automatic extraction of the users’ spatio-temporal patterns encoded as sensor activations from the sensor network deployed inside their home. The proposed method, based on location, time, and duration, was able to extract frequent patterns using the Apriori algorithm and encode the most frequent patterns in the form of a Markov chain. Another area of related work includes Hidden Markov Models (HMMs) (Rabiner, 1989) that are widely used in traditional activity recognition for modeling a sequence of actions, but these topics are already out of scope of this book.

Plan recognition

Plan recognition focuses on a mechanism for recognizing the unobservable state of an agent, given observations of its interaction with its environment (Avrahami-Zilberbrand, 2009). Most existing investigations assume discrete observations in the form of activities. To perform anomalous and suspicious behavior detection, plan recognition algorithms may use a hybrid approach, a symbolic plan recognizer is used to filter consistent hypotheses, passing them to an evaluation engine, which focuses on ranking.

These were advanced approaches applied to various real-life scenarios targeted at discovering anomalies. In the following sections, we’ll dive into more basic approaches for suspicious and anomalous pattern detection.

Fraud detection of insurance claims

First, we’ll take a look at suspicious behavior detection, where the goal is to learn known patterns of frauds, which correspond to modeling known-knowns.
Dataset

We’ll work with a dataset describing insurance transactions publicly available at Oracle Database Online Documentation (2015), as follows:

http://docs.oracle.com/cd/B28359_01/datamine.111/b28129/anomalies.htm

The dataset describes insurance vehicle incident claims for an undisclosed insurance company. It contains 15,430 claims; each claim comprises 33 attributes describing the following components:

- Customer demographic details (Age, Sex, MartialStatus, and so on)
- Purchased policy (PolicyType, VehicleCategory, number of supplements, agent type, and so on)
- Claim circumstances (day/month/week claimed, policy report filed, witness present, past days between incident-policy report, incident-claim, and so on)
- Other customer data (number of cars, previous claims, DriverRating, and so on)
- Fraud found (yes and no)

A sample of the database shown in the following screenshot depicts the data loaded into Weka:
Now the task is to create a model that will be able to identify suspicious claims in future. The challenging thing about this task is the fact that only 6% of claims are suspicious. If we create a dummy classifier saying no claim is suspicious, it will be accurate in 94% cases. Therefore, in this task, we will use different accuracy measures: precision and recall.

Recall the outcome table from *Chapter 1, Applied Machine Learning Quick Start*, where there are four possible outcomes denoted as true positive, false positive, false negative, and true negative:

<table>
<thead>
<tr>
<th>Actual</th>
<th>Classified as</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fraud</td>
<td>TP—true positive</td>
</tr>
<tr>
<td>No fraud</td>
<td>FP—false positive</td>
</tr>
<tr>
<td>TP</td>
<td>TN—true negative</td>
</tr>
</tbody>
</table>

Precision and recall are defined as follows:

- Precision is equal to the proportion of correctly raised alarms, as follows:

\[
Pr = \frac{TP}{TP + FP}
\]

- Recall is equal to the proportion of deviant signatures, which are correctly identified as such:

\[
Re = \frac{TP}{TP + FN}
\]

- With these measures, our dummy classifier scores \( Pr = 0 \) and \( Re = 0 \) as it never marks any instance as fraud (\( TP = 0 \)). In practice, we want to compare classifiers by both numbers, hence we use *F-measure*. This is a de-facto measure that calculates a harmonic mean between precision and recall, as follows:

\[
F - \text{measure} = \frac{2 \times Pr \times Re}{Pr + Re}
\]

Now let's move on to designing a real classifier.
Modeling suspicious patterns

To design a classifier, we can follow the standard supervised learning steps as described in Chapter 1, Applied Machine Learning Quick Start. In this recipe, we will include some additional steps to handle unbalanced dataset and evaluate classifiers based on precision and recall. The plan is as follows:

- Load the data in the .csv format
- Assign the class attribute
- Convert all the attributes from numeric to nominal in order to make sure there are no incorrectly loaded numerical values
- **Experiment 1**: Evaluate models with k-fold cross validation
- **Experiment 2**: Rebalance dataset to a more balanced class distribution and manually perform cross validation
- Compare classifiers by recall, precision, and f-measure

First, let's load the data using the CSVLoader class, as follows:

```java

CSVLoader loader = new CSVLoader();
loader.setFieldSeparator(",");
loader.setSource(new File(filePath));
Instances data = loader.getDataSet();
```

Next, we need to make sure all the attributes are nominal. During the data import, Weka applies some heuristics to guess the most probable attribute type, that is, numeric, nominal, string, or date. As heuristics cannot always guess the correct type, we can set types manually, as follows:

```java
NumericToNominal toNominal = new NumericToNominal();
toNominal.setInputFormat(data);
data = Filter.useFilter(data, toNominal);
```

Before we continue, we need to specify the attribute that we will try to predict. We can achieve this by calling the `setClassIndex(int)` function:

```java
int CLASS_INDEX = 15;
data.setClassIndex(CLASS_INDEX);
```
Next, we need to remove an attribute describing the policy number as it has no predictive value. We simply apply the `Remove` filter, as follows:

```java
Remove remove = new Remove();
remove.setInputFormat(data);
remove.setOptions(new String[] {"-R", "+POLICY_INDEX"});
data = Filter.useFilter(data, remove);
```

Now we are ready to start modeling.

**Vanilla approach**

The vanilla approach is to directly apply the lesson as demonstrated in *Chapter 3, Basic Algorithms – Classification, Regression, Clustering*, without any pre-processing and not taking into account dataset specifics. To demonstrate drawbacks of vanilla approach, we will simply build a model with default parameters and apply k-fold cross validation.

First, let's define some classifiers that we want to test:

```java
ArrayList<Classifier> models = new ArrayList<Classifier>();
models.add(new J48());
models.add(new RandomForest());
models.add(new NaiveBayes());
models.add(new AdaBoostM1());
models.add(new Logistic());
```

Next, we create an `Evaluation` object and perform k-fold cross validation by calling the `crossValidate(Classifier, Instances, int, Random, String[])` method, outputting precision, recall, and F-measure:

```java
int FOLDS = 3;
Evaluation eval = new Evaluation(data);
for(Classifier model : models){
    eval.crossValidateModel(model, data, FOLDS,
    new Random(1), new String[] {});
    System.out.println(model.getClass().getName() + "\n" +
    "\tRecall: " + eval.recall(FRAUD) + "\n" +
    "\tPrecision: " + eval.precision(FRAUD) + "\n" +
    "\tF-measure: " + eval.fMeasure(FRAUD));
}
The evaluation outputs the following scores:

```java
weka.classifiers.trees.J48
   Recall: 0.03358613217768147
   Precision: 0.9117647058823529
   F-measure: 0.06478578892371996
...
weka.classifiers.functions.Logistic
   Recall: 0.037486457204767065
   Precision: 0.2521865889212828
   F-measure: 0.06527070364082249
```

We can see the results are not very promising. Recall, that is, the share of discovered frauds among all frauds is only 1-3%, meaning that only 1-3/100 frauds are detected. On the other hand, precision, that is, the accuracy of alarms is 91%, meaning that in 9/10 cases, when a claim is marked as fraud, the model is correct.

### Dataset rebalancing

As the number of negative examples, that is, frauds, is very small, compared to positive examples, the learning algorithms struggle with induction. We can help them by giving them a dataset, where the share of positive and negative examples is comparable. This can be achieved with dataset rebalancing.

Weka has a built-in filter, `Resample`, which produces a random subsample of a dataset using either sampling with replacement or without replacement. The filter can also bias distribution towards a uniform class distribution.

We will proceed by manually implementing k-fold cross validation. First, we will split the dataset into k equal folds. Fold k will be used for testing, while the other folds will be used for learning. To split dataset into folds, we'll use the `StratifiedRemoveFolds` filter, which maintains the class distribution within the folds, as follows:

```java
StratifiedRemoveFolds kFold = new StratifiedRemoveFolds();
kFold.setInputFormat(data);

double measures[][] = new double[models.size()][3];

for(int k = 1; k <= FOLDS; k++){

   // Split data to test and train folds
   kFold.setOptions(new String[]{...
```
"-N", "+FOLDS", "-F", "+k", "-S", "1"});
Instances test = Filter.useFilter(data, kFold);

kFold.setOptions(new String[]{
  "-N", "+FOLDS", "-F", "+k", "-S", "1", "-V"});
  // select inverse "-V"
Instances train = Filter.useFilter(data, kFold);

Next, we can rebalance train dataset, where the –Z parameter specifies the percentage of dataset to be resampled, and –B bias the class distribution towards uniform distribution:

Resample resample = new Resample();
resample.setInputFormat(data);
resample.setOptions(new String[]{"-Z", "100", "-B", "1"});  //with replacement
Instances balancedTrain = Filter.useFilter(train, resample);

Next, we can build classifiers and perform evaluation:

for(ListIterator<Classifier>it = models.listIterator();
  it.hasNext();){
  Classifier model = it.next();
  model.buildClassifier(balancedTrain);
  eval = new Evaluation(balancedTrain);
  eval.evaluateModel(model, test);
  // save results for average
  measures[it.previousIndex()] [0] += eval.getRecall(FRAUD);
  measures[it.previousIndex()] [1] += eval.getPrecision(FRAUD);
  measures[it.previousIndex()] [2] += eval.getFMeasure(FRAUD);
}

Finally, we calculate the average and output the best model:

  // calculate average
  for(int i = 0; i < models.size(); i++){
    measures[i][0] /= 1.0 * FOLDS;
    measures[i][1] /= 1.0 * FOLDS;
    measures[i][2] /= 1.0 * FOLDS;
  }

  // output results and select best model
  Classifier bestModel = null; double bestScore = -1;
  for(ListIterator<Classifier> it = models.listIterator();
    it.hasNext();){
    Classifier model = it.next();
    model.buildClassifier(balancedTrain);
    eval = new Evaluation(balancedTrain);
    eval.evaluateModel(model, test);
    // save results for average
    measures[it.previousIndex()] [0] += eval.getRecall(FRAUD);
    measures[it.previousIndex()] [1] += eval.getPrecision(FRAUD);
    measures[it.previousIndex()] [2] += eval.getFMeasure(FRAUD);
    }
Fraud and Anomaly Detection

Classifier model = it.next();
double fMeasure = measures[it.previousIndex()][2];
System.out.println(
    model.getClass().getName() + "\n"
   + "Recall:    "+measures[it.previousIndex()][0] + "\n"
   + "Precision: "+measures[it.previousIndex()][1] + "\n"
   + "F-measure: "+fMeasure);
if(fMeasure > bestScore){
    bestScore = fMeasure;
    bestModel = model;
}
System.out.println("Best model:"+bestModel.getClass().getName());

Now the performance of the models has significantly improved, as follows:

weka.classifiers.trees.J48
Recall: 0.44204845100610574
Precision: 0.14570766048577555
F-measure: 0.21912423640160392
...
weka.classifiers.functions.Logistic
Recall: 0.7670657247204478
Precision: 0.13507459756495374
F-measure: 0.22969038530557626
Best model: weka.classifiers.functions.Logistic

What we can see is that all the models have scored significantly better; for instance, the best model, Logistic Regression, correctly discovers 76% of frauds, while producing a reasonable amount of false alarms—only 13% of claims marked as fraud are indeed fraudulent. If an undetected fraud is significantly more expensive than investigation of false alarms, then it makes sense to deal with an increased number of false alarms.

The overall performance has most likely still some room for improvement; we could perform attribute selection and feature generation and apply more complex model learning that we discussed in Chapter 3, Basic Algorithms – Classification, Regression, Clustering.
Anomaly detection in website traffic

In the second example, we'll focus on modeling the opposite of the previous example. Instead of discussing what typical fraud-less cases are, we'll discuss the normal expected behavior of the system. If something cannot be matched against our expected model, it will be considered anomalous.

Dataset

We'll work with a publicly available dataset released by Yahoo Labs that is useful for discussing how to detect anomalies in time series data. For Yahoo, the main use case is in detecting unusual traffic on Yahoo servers.

Even though Yahoo announced that their data is publicly available, you have to apply to use it, and it takes about 24 hours before the approval is granted. The dataset is available here:

http://webscope.sandbox.yahoo.com/catalog.php?datatype=s&did=70

The data set comprises real traffic to Yahoo services, along with some synthetic data. In total, the dataset contains 367 time series, each of which contain between 741 and 1680 observations, recorded at regular intervals. Each series is written in its own file, one observation per line. A series is accompanied by a second column indicator with a one if the observation was an anomaly, and zero otherwise. The anomalies in real data were determined by human judgment, while those in the synthetic data were generated algorithmically. A snippet of the synthetic times series data is shown in the following table:

<table>
<thead>
<tr>
<th>timestamp</th>
<th>value</th>
<th>anomaly</th>
<th>change point</th>
<th>trend</th>
<th>noise</th>
<th>12 hour seasonality</th>
<th>daily seasonality</th>
<th>weekly seasonality</th>
</tr>
</thead>
<tbody>
<tr>
<td>1422237800</td>
<td>4333.4</td>
<td>0</td>
<td>0</td>
<td>4599</td>
<td>1.81</td>
<td>-190.95</td>
<td>-128.88</td>
<td>52.44</td>
</tr>
<tr>
<td>1422241200</td>
<td>4316.1</td>
<td>0</td>
<td>0</td>
<td>4602</td>
<td>-14.65</td>
<td>-220.5</td>
<td>-105.21</td>
<td>54.51</td>
</tr>
<tr>
<td>1422244800</td>
<td>4403.2</td>
<td>0</td>
<td>0</td>
<td>4605</td>
<td>7.04</td>
<td>-190.95</td>
<td>-74.39</td>
<td>56.51</td>
</tr>
<tr>
<td>1422249400</td>
<td>4531.2</td>
<td>0</td>
<td>0</td>
<td>4608</td>
<td>13.52</td>
<td>-110.25</td>
<td>-38.51</td>
<td>58.43</td>
</tr>
<tr>
<td>1422252000</td>
<td>4957.5</td>
<td>1</td>
<td>0</td>
<td>4911</td>
<td>-3.77</td>
<td>-8.91</td>
<td>-2.33</td>
<td>80.27</td>
</tr>
</tbody>
</table>

Snippet of the synthetic time-series data
In the following section, we'll learn how to transform time series data to attribute presentation that allows us to apply machine learning algorithms.

**Anomaly detection in time series data**

Detecting anomalies in raw, streaming time series data requires some data transformations. The most obvious way is to select a time window and sample time series with fixed length. In the next step, we want to compare a new time series to our previously collected set to detect if something is out of the ordinary.

The comparison can be done with various techniques, as follows:

- Forecasting the most probable following value, as well as confidence intervals (for example, Holt-Winters exponential smoothing). If a new value is out of forecasted confidence interval, it is considered anomalous.
- Cross correlation compares new sample to library of positive samples, it looks for exact match. If the match is not found, it is marked as anomalous.
- Dynamic time wrapping is similar to cross correlation, but allows signal distortion in comparison.
- Discretizing signal to bands, where each band corresponds to a letter. For example, \( A = [\text{min}, \text{mean}/3] \), \( B = [\text{mean}/3, \text{mean} \times 2/3] \), and \( C = [\text{mean} \times 2/3, \text{max}] \) transforms the signal to a sequence of letters such as \( a\text{A}\text{A}\text{A}\text{C}\text{A}\text{A}\text{A}\text{B}\text{A} \ldots \). This approach reduces the storage and allows us to apply text-mining algorithms that we will discuss in Chapter 10, *Text Mining with Mallet – Topic Modeling and Spam Detection*.
- Distribution-based approach estimates distribution of values in a specific time window. When we observe a new sample, we can compare whether distribution matches to the previously observed one.

This list is, by no means, exhaustive. Different approaches are focused on detecting different anomalies (for example, in value, frequency, and distribution). We will focus on a version of distribution-based approaches.

**Histogram-based anomaly detection**

In histogram-based anomaly detection, we split signals by some selected time window as shown in the following image.

For each window, we calculate the histogram, that is, for a selected number of buckets, we count how many values fall into each bucket. The histogram captures basic distribution of values in a selected time window as shown at the center of the diagram.
Histograms can be then directly presented as instances, where each bin corresponds to an attribute. Further, we can reduce the number of attributes by applying a dimensionality-reduction technique such as Principal Component Analysis (PCA), which allows us to visualize the reduced-dimension histograms in a plot as shown at the bottom-right of the diagram, where each dot corresponds to a histogram.

In our example, the idea is to observe website traffic for a couple of days and then to create histograms, for example, four-hour time windows to build a library of positive behavior. If a new time window histogram cannot be matched against positive library, we can mark it as an anomaly:
For comparing a new histogram to a set of existing histograms, we will use a density-based k-nearest neighbor algorithm, **Local Outlier Factor (LOF)** (Breunig et al, 2000). The algorithm is able to handle clusters with different densities as shown in the following image. For example, the upper-right cluster is large and widespread as compared to the bottom-left cluster, which is smaller and denser:

![Image showing Local Outlier Factor (LOF) clusters](image)

Let's get started!

**Loading the data**

In the first step, we'll need to load the data from text files to a Java object. The files are stored in a folder, each file contains one-time series with values per line. We'll load them into a list of `Double`:

```java
String filePath = "chap07/ydata/A1Benchmark/real";
List<List<Double>> rawData = new ArrayList<List<Double>>();
```
We will need the min and max value for histogram normalization, so let's collect them in this data pass:

```java
double max = Double.MIN_VALUE;
double min = Double.MAX_VALUE;

for(int i = 1; i<= 67; i++){
    List<Double> sample = new ArrayList<Double> ();
    BufferedReader reader = new BufferedReader(new FileReader(filePath+i+".csv");
    
    boolean isAnomaly = false;
    reader.readLine();
    while(reader.ready()){
        String line[] = reader.readLine().split(",");
        double value = Double.parseDouble(line[1]);
        sample.add(value);
        
        max = Math.max(max, value);
        min = Double.min(min, value);
        
        if(line[2] == "1")
            isAnomaly = true;
    }
    System.out.println(isAnomaly);
    reader.close();
    
    rawData.add(sample);
}
```

The data is loaded, now let's move on to histograms.

**Creating histograms**

We will create a histogram for a selected time window with the $WIN_SIZE$ width. The histogram will hold the $HIST_BINS$ value buckets. The histograms consisting of list of doubles will be stored into an array list:

```java
int WIN_SIZE = 500;
int HIST_BINS = 20;
int current = 0;

List<double[]> dataHist = new ArrayList<double[]>();
for(List<Double> sample : rawData){
```
Fraud and Anomaly Detection

double[] histogram = new double[HIST_BINS];
for(double value : sample){
    int bin = toBin(normalize(value, min, max), HIST_BINS);
    histogram[bin]++;
    current++;
    if(current == WIN_SIZE){
        current = 0;
        dataHist.add(histogram);
        histogram = new double[HIST_BINS];
    }
}
dataHist.add(histogram);

Histograms are now completed. The last step is to transform them into Weka's Instance objects. Each histogram value will correspond to one Weka attribute, as follows:

ArrayList<Attribute> attributes = new ArrayList<Attribute>();
for(int i = 0; i<HIST_BINS; i++){
    attributes.add(new Attribute("Hist-"+i));
}
Instances dataset = new Instances("My dataset", attributes, dataHist.size());
for(double[] histogram: dataHist){
    dataset.add(new Instance(1.0, histogram));
}

The dataset is now loaded and ready to be plugged into an anomaly-detection algorithm.

Density based k-nearest neighbors

To demonstrate how LOF calculates scores, we'll first split the dataset into training and testing set using the testCV(int, int) function. The first parameter specifies the number of folds, while the second parameter specifies which fold to return.

// split data to train and test
Instances trainData = dataset.testCV(2, 0);
Instances testData = dataset.testCV(2, 1);

The LOF algorithm is not a part of the default Weka distribution, but it can be downloaded through Weka's package manager:

http://weka.sourceforge.net/packageMetaData/localOutlierFactor/index.html
LOF algorithm has two implemented interfaces: as an unsupervised filter that calculates LOF values (known-unknowns) and as a supervised k-nn classifier (known-knowns). In our case, we want to calculate the outlier-ness factor, therefore, we'll use the unsupervised filter interface:

```java
import weka.filters.unsupervised.attribute.LOF;
```

The filter is initialized the same way as a usual filter. We can specify the \( k \) number of neighbors, for example, \( k=3 \), with \(-\text{min} \) and \(-\text{max} \) parameters. LOF allows us to specify two different \( k \) parameters, which are used internally as the upper and lower bound to find the minimal/maximal number of values:

```java
LOF lof = new LOF();
lof.setInputFormat(trainData);
lof.setOptions(new String[]{"-min", "3", "-max", "3"});
```

Next, we load training instances into the filter that will serve as a positive example library. After we complete the loading, we call the `batchFinished()` method to initialize internal calculations:

```java
for(Instance inst : trainData){
    lof.input(inst);
}
lof.batchFinished();
```

Finally, we can apply the filter to test data. Filter will process the instances and append an additional attribute at the end containing the LOF score. We can simply output the score on the console:

```java
Instances testDataLofScore = Filter.useFilter(testData, lof);

for(Instance inst : testDataLofScore){
    System.out.println(inst.value(inst.numAttributes()-1));
}
```

The LOF score of the first couple of test instances is as follows:

1.306740014927325
1.318239332210458
1.0294812291949587
1.1715039094530768
To understand the LOF values, we need some background on the LOF algorithm. It compares the density of an instance to the density of its nearest neighbors. The two scores are divided, producing the LOF score. The LOF score around 1 indicates that the density is approximately equal, while higher LOF values indicate that the density of the instance is substantially lower than the density of its neighbors. In such cases, the instance can be marked as anomalous.

**Summary**

In this chapter, we looked into detecting anomalous and suspicious patterns. We discussed the two fundamental approaches focusing on library encoding either positive or negative patterns. Next, we got our hands on two real-life datasets, where we discussed how to deal with unbalanced class distribution and perform anomaly detection in time series data.

In the next chapter, we’ll dive deeper into patterns and more advanced approaches to build pattern-based classifier, discussing how to automatically assign labels to images with deep learning.
Where to buy this book

You can buy Machine Learning in Java from the Packt Publishing website. Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.