Apache Mahout

- This lecture covers
  - What Apache Mahout is, and where it came from
  - A review of recommender engines, clustering, and classification in the real world
  - Setting up Mahout

Mahout has three defining qualities

- **First, Mahout is an open source machine learning library from Apache.**
  - The algorithms it implements fall under the broad umbrella of machine learning or collective intelligence.
  - This can mean many things, but at the moment for Mahout it means primarily recommender engines (collaborative filtering), clustering, and classification.
- **It's also scalable.**
  - Mahout aims to be the machine learning tool of choice when the collection of data to be processed is very large, perhaps far too large for a single machine.
  - In its current incarnation, these scalable machine learning implementations in Mahout are written in Java, and some portions are built upon Apache's Hadoop distributed computation project.
- **Finally, it's a Java library.**
  - It doesn't provide a user interface, a prepackaged server, or an installer.
  - It's a framework of tools intended to be used and adapted by developers.

Mahout History

- Hindi word that refers to an elephant driver, and to explain that one, here's a little history.
- Mahout began life in 2008 as a subproject of Apache's Lucene project, which provides the well-known open source search engine of the same name.
- Lucene provides advanced implementations of search, text mining, and information-retrieval techniques.
- In the universe of computer science, these concepts are adjacent to machine learning techniques like clustering and, to an extent, classification.
- As a result, some of the work of the Lucene committers that fell more into these machine learning areas was spun off into its own subproject.

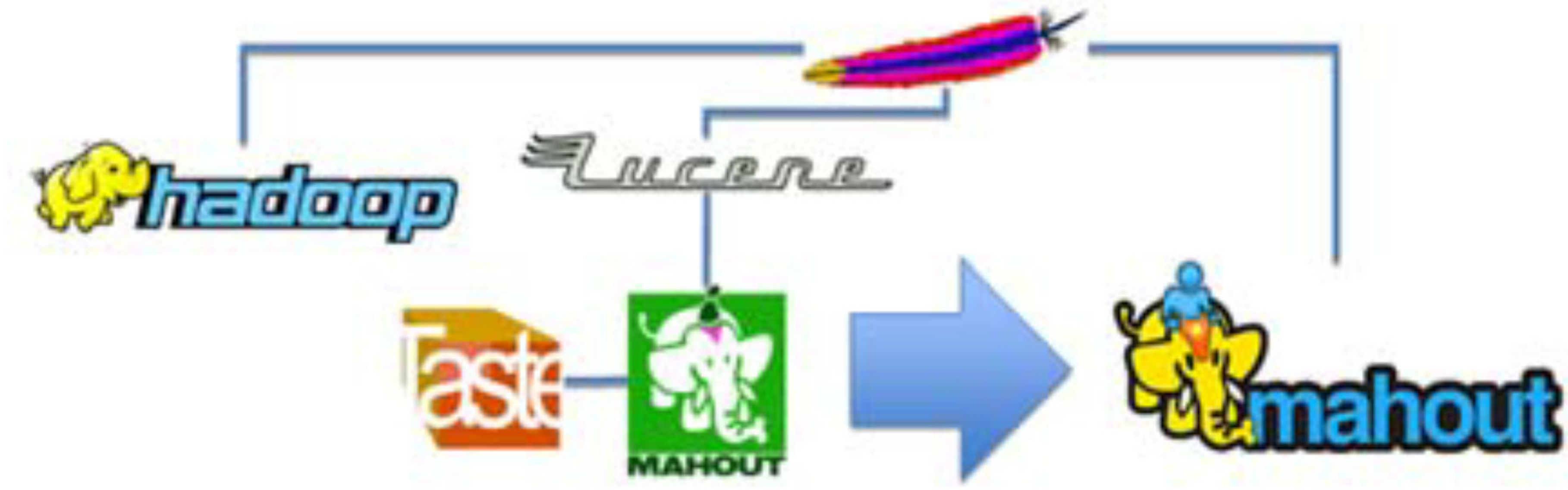
Mahout – Open Source

- As of April 2010, Mahout became a top-level Apache project in its own right, and got a brand-new elephant rider logo to boot.
- Much of Mahout's work has been not only implementing these algorithms conventionally, in an efficient and scalable way, but also converting some of these algorithms to work at scale on top of Hadoop.
- Hadoop's mascot is an elephant, which at last explains the project name!

Mahout Incubates a Number of Algorithms

- Mahout incubates a number of techniques and algorithms, many still in development or in an experimental phase (http://mahout.apache.org/users/basics/algorithms.html).
- At this early stage in the project's life, three core themes are evident:
  - recommender engines (collaborative filtering)
  - clustering
  - classification
- This is by no means all that exists within Mahout, but they are the most prominent and mature themes at the time of writing.

Mahout and its related projects within the Apache



Mahout’s machine learning themes

- Although Mahout is, in theory, a project open to implementations of all kinds of machine learning techniques, it’s in practice a project that focuses on three key areas of machine learning at the moment.

- Currently Mahout supports the following use cases:
  1. Recommendation mining takes users' behavior and from that tries to find items users might like.
  2. Clustering takes e.g. text documents and groups them into groups of topically related documents.
  3. Classification learns from exisiting categorized documents what documents of a specific category look like and is able to assign unlabelled documents to the (hopefully) correct category.
  4. Frequent itemset mining takes a set of item groups (terms in a query session, shopping cart content) and identifies, which individual items usually appear together.

Mahout currently has

- Collaborative Filtering
- User and Item based recommenders
- K-Means, Fuzzy K-Means clustering
- Mean Shift clustering
- Dirichlet process clustering
- Latent Dirichlet Allocation
- Singular value decomposition
- Parallel Frequent Pattern mining
- Complementary Naive Bayes classifier
- Random forest decision tree based classifier
- High performance java collections (previously colt collections)

Recommender Engines

- Recommender engines are the most immediately recognizable machine learning technique in use today.
- You probably have seen services or sites that attempt to recommend books or movies or articles based on your past actions.
- They try to infer tastes and preferences and identify unknown items that are of interest:
  - Amazon.com is perhaps the most famous e-commerce site to deploy recommendations.
  - Based on purchases and site activity, Amazon recommends books and other items likely to be of interest.
  - Netflix similarly recommends DVDs that may be of interest, and famously offered a $1,000,000 prize to researchers who could improve the quality of their recommendations.
  - Dating sites an even recommend people to people.
  - Social networking sites like Facebook use variants on recommender techniques to identify people most likely to be as-yet-unconnected friends.
- As Amazon and others have demonstrated, recommenders can have concrete commercial value by enabling smart cross-selling opportunities.
- One firm reports that recommending products to users can drive an 8-12% increase in sales

Clustering

- As its name implies, clustering techniques attempt to group a large number of things together into clusters that share some similarity.
- It’s a way to discover hierarchy and order in a large data set, and in that way reveal interesting patterns or make the data set easier to comprehend.

- Google News groups news articles by topic using clustering techniques, in order to present news grouped by logical story, rather than a raw listing of all articles.
- Search engines like Clusty group their search results for similar reasons.
- Consumers may be grouped into segments (clusters) using clustering techniques based on attributes like income, location, and buying habits.
- Clustering helps identify structure, and even hierarchy, among a large collection of things that may be otherwise difficult to make sense of.
- Enterprises might use this technique to discover hidden groupings among users, or to organize a large collection of documents sensibly, or to discover common usage patterns for a site based on logs.

Classification

- Classification techniques decide how much a thing is or isn't part of some type or category, or how much it does or doesn't have some attribute.
- Classification, like clustering, is common, but it's even more behind the scenes.
- Often these systems learn by reviewing many instances of items in the categories in order to deduce classification rules.

- This general idea has many applications:
  - Yahoo! Mail decides whether or not incoming messages are spam based on prior emails and spam reports from users, as well as on characteristics of the email itself.
  - Google's Picasa and other photo-management applications can decide when a region of an image contains a human face.
  - Optical character recognition software classifies small regions of scanned text into individual characters.
  - Apple's Genius feature in iTunes reportedly uses classification to classify songs into potential playlists for users.

Classification Uses

- Classification helps decide if a new input or thing matches a previously observed pattern or not, and it’s often used to classify behavior or patterns as unusual.
- It could be used to detect suspicious network activity or fraud.
- It might be used to figure out when a user’s message indicates frustration or satisfaction.

Input Data

- Each of these techniques works best when provided with a large amount of good input data.

- In some cases, these techniques must not only work on large amounts of input, but must produce results quickly, and these factors make scalability a major issue.

- And, as mentioned before, one of Mahout’s key reasons for being is to produce implementations of these techniques that do scale up to huge input.

Tackling large scale with Mahout and Hadoop

- Let’s consider the size of a few problems where you might deploy Mahout.
- Consider that Picasa may have hosted over half a billion photos even three years ago, according to some crude estimates.
- This implies millions of new photos per day that must be analyzed.
- The analysis of one photo by itself isn’t a large problem, even though it’s repeated millions of times.
- But the learning phase can require information from each of the billions of photos simultaneously—a computation on a scale that isn’t feasible for a single machine.

More Examples

- According to a similar analysis, Google News sees about 3.5 million new news articles per day.
- Although this does not seem like a large amount in absolute terms, consider that these articles must be clustered, along with other recent articles, in minutes in order to become available in a timely manner.
- The subset of rating data that Netflix published for the Netflix Prize contained 100 million ratings.
- Because this was just the data released for contest purposes, presumably the total amount of data that Netflix actually has and must process to create recommendations is many times larger!

Mahout makes scalability a top priority

- Machine learning techniques must be deployed in contexts like these, where the amount of input is large—so large that it isn’t feasible to process it all on one computer, even a powerful one.
- Without an implementation such as Mahout, these would be impossible tasks.
- Sophisticated machine learning techniques, applied at scale, were until recently only something that large, advanced technology companies could consider using.
- But today computing power is cheaper than ever and more accessible via open source frameworks like Apache’s Hadoop.
- Mahout attempts to complete the puzzle by providing quality, open source implementations capable of solving problems at this scale with Hadoop, and putting this into the hands of all technology organizations.

Setting up Mahout

- Mahout and its associated frameworks are Java-based and therefore platform independent, so you should be able to use it with any platform that can run a modern JVM.
- Note that Mahout requires Java 6.
- As with many Apache projects, Mahout's build and release system is built around Maven (http://maven.apache.org).
- Maven is a command-line tool that manages dependencies, compiles code, packages releases, generates documentation, and publishes formal releases.
- Although it has some superficial resemblance to the also popular Ant build tool, it isn't the same.
- Ant is a flexible, lower-level scripting language, and Maven is a higher-level tool more purpose-built for dependency and release management.
- Because Mahout uses Maven, you should install Maven yourself.

Installing Mahout

- Mahout is still in development.
- This release and others may be downloaded by following the instructions at https://cwiki.apache.org/confluence/display/MAHOUT/Downloads
  - the archive of the source code may be unpacked anywhere that's convenient on your computer.
- Because Mahout is changing frequently, and bug fixes and improvements are added regularly, it may be useful to use a later release than 0.5
  - or even the latest unreleased code from Subversion
  - See https://cwiki.apache.org/confluence/display/MAHOUT/Version+Control

- Once you've obtained the source, either from Subversion or from a release archive, create a new project for Mahout in your IDE.
- It will be easiest to use your IDE's Maven integration to import the Maven project from the pom.xml file in the root of the project source.

Adding Mahout to a Maven Project

- Mahout is also available via a maven repository under the group id org.apache.mahout.

- If you would like to import the latest release of mahout into a java project, add the following dependency in your pom.xml:

```
<dependency>
    <groupId>org.apache.mahout</groupId>
    <artifactId>mahout-core</artifactId>
    <version>0.9</version>
</dependency>
```

Collaborative filtering

- Producing recommendations based on knowledge of users' relationships to items.
- These techniques require no knowledge of the properties of the items themselves.
- This recommender framework doesn't care if the items are books, flowers, or even other people, because nothing about their attributes enters into any of the input.
- There are other approaches based on the attributes of items, and these are generally referred to as content-based recommendation techniques.

Collaborative Filtering Example

- For example, if a friend recommended a book to you because it’s a technical book, and the friend likes other technical books, then the friend is engaging in something more like content-based recommendation.
- The suggestion is based on an attribute of the books.
- Nothing is wrong with content-based techniques; on the contrary, they can work quite well.
- They’re necessarily domain-specific approaches, and they’d be hard to meaningfully codify into a framework.
- To build an effective content-based book recommender, one hase to decide which attributes of a book—page count, author, publisher, color, font—are meaningful, and to what degree.
- None of this knowledge translates into any other domain; recommending books this way doesn’t help in the area of recommending pizza toppings.

Running a first recommender engine

- Mahout contains a recommender engine—several types of them, in fact, beginning with conventional user-based and item-based recommenders.

- It includes implementations of several other algorithms as well, but for now we'll explore a simple user-based recommender.

Input

- The recommender requires input—data on which it will base its recommendations.
- This data takes the form of preferences in Mahout-speak.
- Because the recommender engines that are most familiar involve recommending items to users, it'll be most convenient to talk about preferences as associations from users to items—these users and items could be anything.
- A preference consists of a user ID and an item ID, and usually a number expressing the strength of the user's preference for the item.
- IDs in Mahout are always numbers—integers, in fact.
- The preference value could be anything, as long as larger values mean stronger positive preferences.
- For instance, these values might be ratings on a scale of 1 to 5, where 1 indicates items the user can't stand, and 5 indicates favorites.
- Create a text file containing data about users, who are cleverly named "1" to "5," and their preferences for seven books, simply called "101" through "107."
- In real life, these might be customer IDs and product IDs from a company database;
  - Mahout doesn't require that the users and items be named with numbers.
  - Write this data in a simple comma-separated value format.

Recommender Input file, Intro.csv

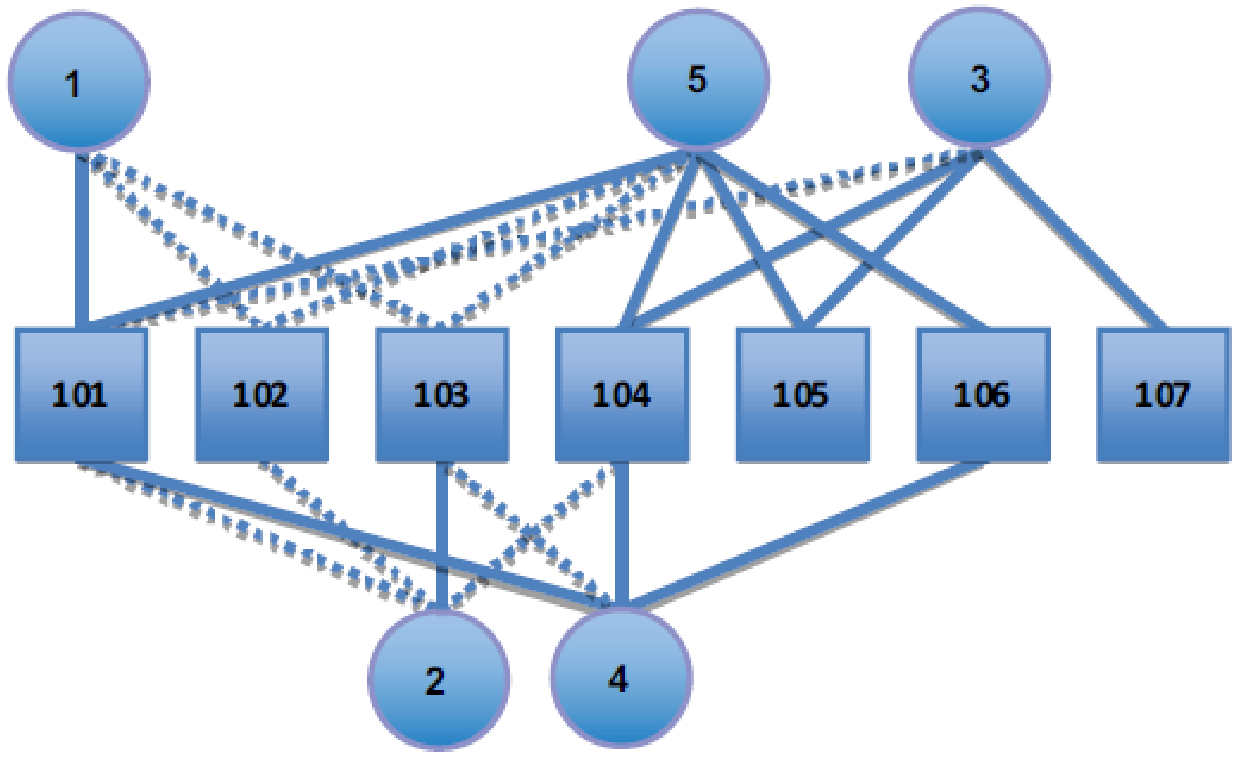
```
1,101,5.0
1,102,3.0
1,103,2.5

2,101,2.0
2,102,2.5
2,103,5.0
2,104,2.0

3,101,2.5
3,104,4.0
3,105,4.5
3,107,5.0

4,101,5.0
4,103,3.0
4,104,4.5
4,106,4.0

5,101,4.0
5,102,3.0
5,103,2.0
5,104,4.0
5,105,3.5
5,106,4.0
```

User 1 has preference 3.0 for item 102

User ID, item ID, preference value

With some study, trends appear

- Users 1 and 5 seem to have similar tastes.
- They both like book 101, like 102 a little less, and like 103 less still.
- The same goes for users 1 and 4, as they seem to like 101 and 103 identically
- (no word on how user 4 likes 102 though).
- On the other hand, users 1 and 2 have tastes that seem to run counter to each other—1 likes 101 whereas 2 doesn't, and 1 likes 103 but 2 is just the opposite.
- Users 1 & 3 don't overlap much-the only book both express a preference for is 101.
- Figure in the next slide illustrates the relationships, both positive and negative, between users and items.



Relationships between users 1 to 5 and items 101 to 107.

Dashed lines represent associations that seem negative.
The user doesn't seem to like the item much
but expresses a relationship to the item.

Creating a recommender

- So what book might you recommend to user 1?
- Not 101, 102, or 103—user 1 already knows about these books, apparently, and recommendation is about discovering new things.
- Intuition suggests that because users 4 and 5 seem similar to 1, it would be good to recommend something that user 4 or user 5 likes.
- That leaves books 104, 105, and 106 as possible recommendations.
- On the whole, 104 seems to be the most liked of these possibilities, judging by the preference values of 4.5 and 4.0 for item 104.

A simple user-based recommender program with Mahout

```
class RecommenderIntro {

  public static void main(String[] args) throws Exception {

    DataModel model =
      new FileDataModel (new File("intro.csv"));          <- Load data file

    UserSimilarity similarity =
      new PearsonCorrelationSimilarity (model);
    UserNeighborhood neighborhood =
      new NearestNUserNeighborhood (2, similarity, model);

    Recommender recommender = new GenericUserBasedRecommender (
        model, neighborhood, similarity);                 <- Create
                                                             recommender engine
    List<RecommendedItem> recommendations =
        recommender.recommend(1, 1);                      <- For user 1,
                                                             recommend
    for (RecommendedItem recommendation : recommendations) {  1 item
      System.out.println(recommendation);
    }

  }

}
```

Components

- DataModel implementation stores and provides access to all the preference, user, and item data needed in the computation.
- UserSimilarity implementation provides some notion of how similar two users are; this could be based on one of many possible metrics or calculations.
- UserNeighborhood implementation defines a notion of a group of users that are most similar to a given user.
- Recommender implementation pulls all these components together to recommend items to users.

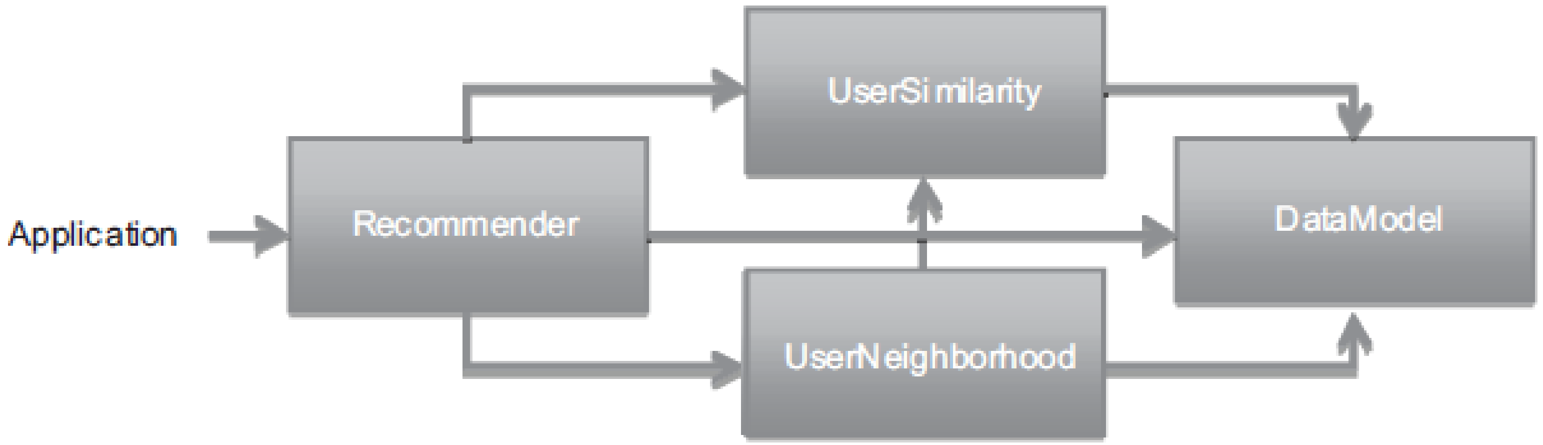
Analyzing the output

- When you run the code, the output in your terminal or IDE should be as follows:

```
RecommendedItem [item:104, value:4.257081]
```

- The request asked for one top recommendation, and it got one.
- The recommender engine recommended book 104 to user 1.
- Further, it said that the recommender engine did so because it estimated user 1's preference for book 104 to be about 4.3, and that was the highest among all the items eligible for recommendation.

That's not bad. Book 107 did not appear; it was also recommendable but was only associated with a user who had different tastes. It picked 104 over 106, and this makes



Simplified illustration of component interaction in a Mahout user-based recommender

sense when you note that 104 is a bit more highly rated overall. Further, the output contained a reasonable estimate of how much user 1 likes item 104—something between the 4.0 and 4.5 that users 4 and 5 expressed.

In real life, data sets are huge, and they’re noisy

- The right answer isn’t obvious from looking at the data, but the recommender engine made some decent sense of it and returned a defensible answer.
- For clear, small data sets, producing recommendations is as trivial as it appears in the preceding example.
- In real life, data sets are huge, and they’re noisy.
- For example, imagine a popular news site recommending news articles to readers.
- Preferences are inferred from article clicks, but many of these preferences may be bogus—maybe a reader clicked on an article but didn’t like it, or clicked on the wrong story.
- Perhaps many of the clicks occurred while not logged in, so they can’t be associated with a user.
- And imagine the size of the data set—perhaps billions of clicks in a month.
- Producing the right recommendations from this data and producing them quickly isn’t trivial.

Evaluating a recommender

- A recommender engine is a tool, a means to answer the question, “What are the best recommendations for a user?”
- Before investigating the answers, it’s best to investigate the question.
- What exactly is a good recommendation?
- And how does one know when a recommender is producing them?
- The best possible recommender would be a sort of psychic that could somehow know, before you do, exactly how much you’d like every possible item that you’ve not yet seen or expressed any preference for.
- A recommender that could predict all your preferences exactly would present all items ranked by your future preference and be done.
- These would be the best possible recommendations.
- And indeed, most recommender engines operate by trying to do just this, estimating ratings for some or all other items.
- As a result, one way of evaluating a recommender’s recommendations is to evaluate the quality of its estimated preference values—that is, by evaluating how closely the estimated preferences match the actual preferences.

Training data and scoring

- Those actual preferences don’t exist, though.
- Nobody knows for sure how you’ll like some new item in the future (including you).
- In the context of a recommender engine, this can be simulated by setting aside a small part of the real data set as test data.
- These test preferences aren’t present in the training data fed into a recommender engine under evaluation.
- Instead, the recommender is asked to estimate preferences for the missing test data, and estimates are compared to the actual values.
- From there, it’s fairly simple to produce a kind of score for the recommender.
- For example, it’s possible to compute the average difference between estimate and actual preference.
- With a score of this type, lower is better, because that would mean the estimates differed from the actual preference values by less.
- A score of 0.0 would mean perfect estimation—no difference at all between estimates and actual values.

Sometimes the root-mean-square of the differences is used: this is the square root of the average of the *squares* of the differences between actual and estimated preference values. See table 2.1 for an illustration of this. Again, lower is better.

Table 2.1 An illustration of the average difference and root-mean-square calculation

	Item 1	Item 2	Item 3
Actual	3.0	5.0	4.0
Estimate	3.5	2.0	5.0
Difference	0.5	3.0	1.0
Average difference	$= (0.5 + 3.0 + 1.0) / 3 = 1.5$		
Root-mean-square	$= \sqrt{((0.5^2 + 3.0^2 + 1.0^2) / 3)} = 1.8484$		

Table 2.1 shows the difference between a set of actual and estimated preferences, and how they're translated into scores. Root-mean-square more heavily penalizes estimates that are way off, as with item 2 here, and that's considered desirable by some. For example, an estimate that's off by two whole stars is probably more than twice as bad as one off by just one star. Because the simple average of differences is perhaps more intuitive and easier to understand

Running RecommenderEvaluator

Let's revisit the example code and evaluate the simple recommender on this simple data set, as shown in the following listing.

Listing 2.3 Configuring and running an evaluation of a recommender

```
    RandomUtils.useTestSeed();
    DataModel model = new FileDataModel (new File("intro.csv"));

    RecommenderEvaluator evaluator =
      new AverageAbsoluteDifferenceRecommenderEvaluator ();

    RecommenderBuilder builder = new RecommenderBuilder() {
      @Override
      public Recommender buildRecommender(DataModel model)
          throws TasteException {
    UserSimilarity similarity = new PearsonCorrelationSimilarity (model);
    UserNeighborhood neighborhood =
      new NearestNUserNeighborhood (2, similarity, model);
    return
      new GenericUserBasedRecommender (model, neighborhood, similarity);
  }
};

double score = evaluator.evaluate(
    builder, null, model, 0.7, 1.0);
System.out.println(score);
```

Generates repeatable results

Builds recommender in listing 2.2

Trains with 70% of data; tests with 30%

Most of the action happens in `evaluate()`: the `RecommenderEvaluator` splits the data into a training and test set, builds a new training `DataModel` and `Recommender` to test, and compares its estimated preferences to the actual test data.

Note that there's no `Recommender` passed to `evaluate()`. That's because, inside, the method will need to build a `Recommender` around a newly created training `Data-Model`. The caller must provide an object that can build a `Recommender` from a `Data-Model`—a `RecommenderBuilder`.

Evaluating precision and recall

We could also take a broader view of the recommender problem: it's not strictly *necessary* to estimate preference values in order to produce recommendations. In many cases, presenting an ordered list of recommendations, from best to worst, is sufficient, without including estimated preference values. In fact, in some cases the exact ordering of the list doesn't matter much—a set of a few good recommendations is fine.

Taking this more general view, we could also apply classic information retrieval metrics to evaluate recommenders: precision and recall. These terms are typically applied to things like search engines, which return a set of best results for a query out of many possible results.

A search engine shouldn't return irrelevant results in the top results, but it should strive to return as many relevant results as possible. *Precision* is the proportion of top results that are relevant, for some definition of relevant. *Precision at 10* would be this proportion judged from the top 10 results. *Recall* is the proportion of all relevant results included in the top results. See figure 2.3 for a visualization of these ideas.

These terms can easily be adapted to recommenders: *precision* is the proportion of top recommendations that are good recommendations, and *recall* is the proportion of good recommendations that appear in top recommendations.

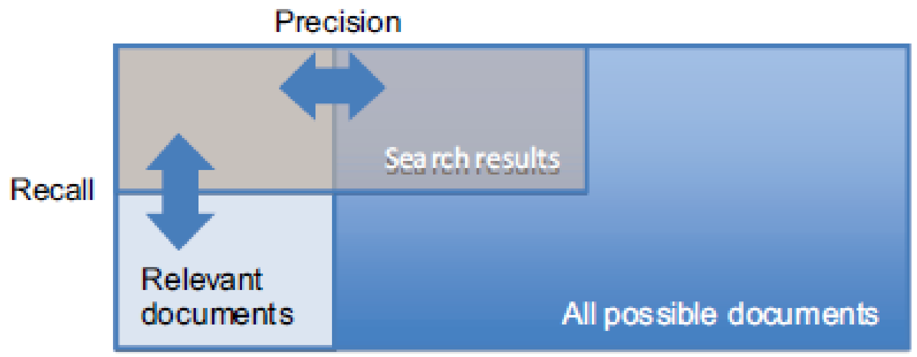


Figure 2.3 An illustration of precision and recall in the context of search results

Running RecommenderIRStatsEvaluator

Again, Mahout provides a fairly simple way to compute these values for a `Recommender`, as showing in the next listing.

Listing 2.4 Configuring and running a precision and recall evaluation

```
RandomUtils.useTestSeed();
DataModel model = new FileDataModel (new File("intro.csv"));

RecommenderIRStatsEvaluator evaluator =
  new GenericRecommenderIRStatsEvaluator ();
RecommenderBuilder recommenderBuilder = new RecommenderBuilder() {
  @Override
  public Recommender buildRecommender(DataModel model)
      throws TasteException {
    UserSimilarity similarity = new PearsonCorrelationSimilarity (model);
    UserNeighborhood neighborhood =
          new NearestNUserNeighborhood (2, similarity, model);
        return
          new GenericUserBasedRecommender (model, neighborhood, similarity);
      }
    };
    IRStatistics stats = evaluator.evaluate(
        recommenderBuilder, null, model, null, 2,
        GenericRecommenderIRStatsEvaluator.CHOOSE_THRESHOLD,
        1.0);

    System.out.println(stats.getPrecision());
    System.out.println(stats.getRecall());
```

Evaluate precision and recall at 2

Without the call to `RandomUtils.useTestSeed()`, the result you see would vary significantly due to the random selection of training data and test data, and because the data set is so small here. But with the call, the result ought to be

```
0.75
1.0
```

Precision at 2 is 0.75; on average about three-quarters of recommendations were good. Recall at 2 is 1.0; all good recommendations are among those recommended.

But what exactly is a good recommendation here? The framework was asked to decide—it didn't receive a definition. Intuitively, the most highly preferred items in the test set are the good recommendations, and the rest aren't.

Listing 2.5 User 5's preferences in the test data set

```
5,101,4.0
5,102,3.0
5,103,2.0
5,104,4.0
5,105,3.5
5,106,4.0
```

Look at user 5 in this simple data set again. Let's imagine the preferences for items 101, 102, and 103 were withheld as test data. The preference values for these are 4.0, 3.0, and 2.0. With these values missing from the training data, a recommender engine ought to recommend 101 before 102, and 102 before 103, because this is the order in which user 5 prefers these items. But would it be a good idea to recommend 103? It's last on the list; user 5 doesn't seem to like it much. Book 102 is just average. Book 101 looks reasonable, because its preference value is well above average. Maybe 101 is a good recommendation; 102 and 103 are *valid*, but not *good* recommendations.

Mahout Recommendation Example

- Mahout's recommenders expect interactions between users and items as input.
- The easiest way to supply such data to Mahout is in the form of a textfile, where every line has the format userID,itemID,value.
- Here userID and itemID refer to a particular user and a particular item, and value denotes the strength of the interaction (e.g. the rating given to a movie).
- In this example, we'll use some made up data for simplicity. Create a file called "dataset.csv" and copy the following example interactions into the file.
  - 1,10,1.0
  - 1,11,2.0
  - 1,12,5.0
  - …
  - 2,10,1.0
  - 2,11,2.0

Creating a user-based recommender

- Create a class called SampleRecommender with a main method.
- The first thing we have to do is load the data from the file.
- Mahout's recommenders use an interface called DataModel to handle interaction data.
- You can load our the interaction data like this:

```
DataModel model = new FileDataModel(new File("/path/to/dataset.csv"));
```

- In this example, we want to create a user-based recommender.
- The idea behind this approach is that when we want to compute recommendations for a particular users, we look for other users with a similar taste and pick the recommendations from their items.
- For finding similar users, we have to compare their interactions.
- There are several methods for doing this.
- One popular method is to compute the correlation coefficient between their interactions.
- In Mahout, you use this method as follows:

```
UserSimilarity similarity = new PearsonCorrelationSimilarity(model);
```

Defining which similar users to leverage for the recommender

- The next thing we have to do is to define which similar users we want to leverage for the recommender.
- For the sake of simplicity, we'll use all that have a similarity greater than 0.1.
- This is implemented via a ThresholdUserNeighborhood:

```
UserNeighborhood neighborhood = new ThresholdUserNeighborhood(0.1, similarity, model);
```

- Now we have all the pieces to create our recommender:

```
UserBasedRecommender recommender =
                        new GenericUserBasedRecommender(model, neighborhood, similarity);
```

- We can easily ask the recommender for recommendations now.

- If we wanted to get three items recommended for the user with userID 2, we would do it like this:

```
List recommendations = recommender.recommend(2, 3);
for (RecommendedItem recommendation : recommendations) {
  System.out.println(recommendation);
}
```

Evaluation

- You might ask how to make sure that your recommender returns good results.
- Unfortunately, the only way to be really sure about the quality is by doing an A/B test with real users in a live system.
- We can however try to get a feel of the quality, by statistical offline evaluation.
- Just keep in mind that this does not replace a test with real users!
- One way to check whether the recommender returns good results is by doing a hold-out test.
- We partition our dataset into two sets:
  - a trainingset consisting of 90% of the data
  - a testset consisting of 10%.

Training

- Then we train our recommender using the training set and look how well it predicts the unknown interactions in the testset.

- To test our recommender, we create a class called EvaluateRecommender with a main method and add an inner class called MyRecommenderBuilder that implements the RecommenderBuilder interface.

- We implement the buildRecommender method and make it setup our user-based recommender:

```
UserSimilarity similarity = new PearsonCorrelationSimilarity(dataModel);
UserNeighborhood neighborhood = new ThresholdUserNeighborhood(0.1, similarity, dataModel);
return new GenericUserBasedRecommender(dataModel, neighborhood, similarity);
```

Testing

- Now we have to create the code for the test.
- We'll check how much the recommender misses the real interaction strength on average.
- We employ an AverageAbsoluteDifferenceRecommenderEvaluator for this.
- The following code shows how to put the pieces together and run a hold-out test:

```
DataModel model = new FileDataModel(new File("/path/to/dataset.csv"));
RecommenderEvaluator evaluator = new AverageAbsoluteDifferenceRecommenderEvaluator();
RecommenderBuilder builder = new MyRecommenderBuilder();
double result = evaluator.evaluate(builder, null, model, 0.9, 1.0);
System.out.println(result);
```

- Note: if you run this test multiple times, you will get different results, because the splitting into training set and test set is done randomly.