ElectricDeel

A Scalable, Modular Product Recommendation System For Consumer Electronics
Dept. of CIS - Senior Design 2010-2011

Edward Siegel
edwards@seas.upenn.edu
Univ. of Pennsylvania
Philadelphia, PA

Vikas Shanbhogue
svikas@seas.upenn.edu
Univ. of Pennsylvania
Philadelphia, PA

Bennett Blazei
blazei@wharton.upenn.edu
Univ. of Pennsylvania
Philadelphia, PA

Zachary Ives
zives@cis.upenn.edu
Univ. of Pennsylvania
Philadelphia, PA

ABSTRACT

The consumer electronics market has a large number of novice shoppers who are not familiar with industry jargon and do not have the knowledge required to make informed decisions about which products to purchase. There is significant demand for a service that gathers data from consumers about their needs and preferences by asking simple English-language questions, and recommends products that match these needs. The current alternatives to this are physically traveling to a store or examining jargon-filled online reviews, both of which are tedious, untrustworthy, and generally unattractive options. Many existing recommendation systems do not provide users with adequate descriptions of how recommendations were computed. Others provide recommendations that are not sufficiently (a) trustworthy, (b) accurate, or (c) tailored to the individual customer.

ElectricDeel is a scalable, extensible platform for providing accurate product recommendations through a user interface that instills trust in the user. The system is also sufficiently modular to allow future updates, enhancements, and product additions. The web application asks the user a series of simple English questions and passes the answers to a recommendation system that uses these answers to compute a ranking for a set of relevant products. Each of the ranked products is presented along with a link to the online retailer that sells the product at the lowest price. Product data is collected using a distributed web crawler and processed in parallel using Hadoop MapReduce [20][6].

1. INTRODUCTION

For the average consumer, shopping for electronics can be a daunting and difficult task. In order to make an informed decision, consumers often need to be familiar with domain-specific knowledge and complex industry jargon. Currently, technology-ignorant consumers have two options for purchasing electronics:

1. They can go to a brick-and-mortar retail store where they will be assisted by sales associates that are incentivized to sell products that may not be the best for the consumer.

2. They can sift through jargon-heavy online product reviews and then scour the internet for the lowest price.

The fully informed online shopping experience is likely to consist of four steps, all of which use different resources:

1. 1. Researching the meaning of product specifications in order to determine the specifications that are most important to the shopper

2. 2. Researching what products exist that meet the desired specifications

3. 3. Finding the retailer that sells the desired product at the lowest price

4. 4. Purchasing the item

For instance, someone shopping for a TV must first understand the meaning of resolution, contrast ratio, refresh rate, and other terms. After conducting this research by reading articles online, the consumer may find that he/she wants a 47" LCD 1080p Television. The consumer must then compare all of the 47" LCD 1080p TVs on the market, along with their price, in order to determine which one he/she ultimately wants to purchase. This is typically done using online catalogs, review sites, or by browsing through the offerings of multiple retailers. Finally, the consumer must check multiple retailers in order to find which one sells the desired TV at the lowest price, possibly with the help of a tool such as Pricegrabber [12].

ElectricDeel drastically simplifies this process by shifting the burden of having to understand technical aspects of consumer electronics away from consumers. Not only are consumers freed from the responsibility of researching product specifications, but they are provided with a single resource with which to perform all four of the above steps.

2. RELATED WORK

The two products that most closely resemble ElectricDeel are Yahoo SmartSort and DiGiCam [14][16][23]. SmartSort was created in 2003 and provides customized electronics recommendations based on a user’s answers to certain questions (for example, “How important is Optical Zoom?”), although they do not openly discuss the method by which they arrive at recommendations. The products are also limited to those sold through Yahoo Shopping. The site is no longer available online and appears to have been abandoned.

DiGiCam provides digital camera recommendations using three techniques:

1. Collaborative filtering – Product recommendations based on the purchases of other shoppers

2. Utility-based recommendation – Product recommendations based on the maximization of a utility function
3. Knowledge-based recommendation – Product recommendations based on inferences regarding a shopper’s needs and preferences

While DiGiCam seemed promising, their implementation of knowledge-based recommendation resulted in low user satisfaction. It has also been shown that collaborative filtering is often ineffective because it can reflect the decision-making strategies of dissimilar customers [15]. For example, shoppers A and B might both purchase the same DVD online, and then person B may go on to buy a TV. If the two shoppers actually have very different decision-making strategies, it may not be appropriate to recommend that A buy the same TV as B, regardless of the fact that they both purchased the same DVD.

There has been a good deal of research regarding the kinds of recommendations that are most useful to consumers [28][27][15][18][34]. This research can be used in order to significantly outperform both SmartSort and DiGiCam. Neither SmartSort nor DiGiCam provide the user with an explanation for how the recommendation was computed. Research has shown that such explanations can help to build trust with the user [28]. It has also been shown that it is helpful to aid users in the consideration of trade-offs [27]. Further research has shown that consumers prefer recommendation engines that they believe think like them, which can be useful when combined with research about how consumers think [15][18][34].

There are also general recommendation systems such as Hunch, which makes recommendations based on a user’s answers to simple questions [7]. Hunch provides recommendations for all classes of problems, including shopping for electronics, but the results are user-generated (and may therefore be error-prone). While Hunch seems to perform well for many types of recommendations, its focus on creating a generalized recommendation system forces consumer electronics into a more generalized model. We believe that this results in a lot of room for improvement when it comes to a recommendation system that focuses solely on electronics.

Once a shopper has settled on a particular product, there are several comparison shopping sites including PriceGrabber, Bizrate and Milo that help find the cheapest retailer [12][2][9]. There are also sites such as CNET which provide product reviews to help inform purchase decisions [3].

One of the ways this product is differentiated from existing products is through focus on an aggressive data collection strategy. Since there is no centralized repository for all consumer electronics and pricing, most recommendation systems are limited to only recommending products from a single source (such as SmartSort only recommending products from Yahoo Shopping). To this end, ElectricDeel includes a distributed web crawler and data processing pipeline that extracts product information from online retailers in order to build a more comprehensive database of products and pricing.

Web crawling is a topic that has received a good deal of coverage in the literature, including the issues related to developing a crawler infrastructure and the tradeoff associated with different crawling algorithms [26]. There have also been several investigations into building large-scale, distributed crawlers [21][22]. Google arguably possesses one of the most intricate, successful and well known of such crawlers [30].

While there are a considerable number of well established, large scale web crawlers and crawling algorithms already in existence, ElectricDeel uses a unique crawler that leverages the properties of the specific problem domain. For instance, this crawler visits a large number of individual pages on a small number of domains, and its primary purpose is extracting information, not indexing pages.

3. SYSTEM MODEL

3.1 Challenges And Design Goals

Building such a system involves solving problems that are not solved by existing systems. This system faces novel challenges, as it must have significant improvements that enable it to be practically successful. Such novel challenges include:

- The system has to have an interface that users can trust. Since a user is allowing the system to make decisions for him/her, the user must completely trust both (a) the system’s ability to understand his/her needs and (b) the decision-making process employed by the system when arriving at recommendations. The recommendations must always reflect the needs of the user, and this reflection must always be made apparent to the user. In the case of existing systems such as DiGiCam and SmartSort, the user is left in the dark as to why or how recommendations are made, and not all predictions are necessarily relevant. In other words, one of the greatest challenges is to effectively bridge the gap between the expert and the layperson.

- It must be scalable. The system must not fail under increased load, and it must be able to handle a substantial increase in the number of supported products, crawl targets, and recommendation techniques. In addition, the system must be sufficiently modular so that updates and modifications can be made without disrupting the existing system.

- All relevant data must be extracted from websites accurately. All of the data must also be normalized – that is, it must be properly converted into a consistent internal representation. For instance, it is non-trivial to build a system that is capable of determining, with high accuracy, that the strings LG 42LD520 42-Inch 1080p 120Hz LCD HDTV and LG 42" Class / 1080p / 120Hz / LCD HDTV refer to the same product and that it has a higher resolution than the product referred to by the string VIZIO 32" Class 720p 60Hz LCD HDTV - Black (E320VL). Furthermore, since the sites being crawled are not perfect, all of the collected data must be aggregated in order to account for missing information, and outlier detection among other techniques must be employed in order to account for incorrect information.

3.2 System Overview

Figure 1 shows a block diagram of the system, which is comprised of four main components:

1. A web crawler
2. A data processing pipeline
3. A user-facing web application
4. A recommendation engine

The purpose of the web crawler is to collect product specifications, attributes, and prices from various retailers on the web.

The data processing pipeline aggregates, normalizes and transforms the data collected by the crawler into a format that is consumable by the recommendation engine.

The web application is the user-facing component of the system that takes user input and presents the user with product recommendations.

The recommendation engine uses the input that is collected by the web application to determine which products are best for the user.

3.3 The Crawler

The crawler is split up into two primary components: The master and the slave. While there is only one master, there may be many slaves running across many machines (there may also be multiple slaves on a single machine). The master’s job is to be a central point of coordination for the whole crawl process, while the slave’s job is to visit pages and extract data.

3.3.1 The Master

The master has 4 main responsibilities:

1. Determining which URLs should be crawled
2. Distributing URLs to slaves for crawling
3. Enqueuing new URLs to be crawled in the future
4. Restarting crawl jobs that have failed (i.e. due to slave failure)

Part of the master’s job is to ensure that the system as a whole doesn’t abuse any particular website. For instance, if the crawler has the capacity to crawl thousands of URLs at once, we want to ensure that the system is not requesting thousands of pages from Amazon.com. In order to provide some assurance that the crawler behaves in this way, the master splits up time into a series of one-minute windows. In each window, the master attempts to distribute as many URLs to slaves as possible subject to the following constraints:

1. Each slave may be sent at most $\text{max} \_\text{targets} \_\text{per} \_\text{slave}$ URLs per window
2. At most $\text{global} \_\text{max} \_\text{targets} \_\text{per} \_\text{domain}$ URLs from the same domain may be sent out per window
3. For each domain $x$, we may optionally have the additional constraint that no more than $\text{local} \_\text{max} \_\text{targets} \_\text{per} \_\text{domain} (x)$ URLs from $x$ may be sent out per window

3.3.2 The Slave

The slave’s job is to receive URLs from the master, and for each URL:

1. Visit the URL and download the page
2. Extract from the page: (a) Data and (b) Links (URLs to be crawled in the future)
3. Send the data to the distributed file system for future processing
4. Send the newly discovered URLs to the master so they can be crawled in the future
5. Notify the master that the URL has been successfully crawled

While the master splits time into windows, the slaves do not. This means that the constraints that limit the URLs sent out by the master are not actually guarantees, but should be good enough in practice. For instance, the master may only be willing to send out five URLs from bestbuy.com per window, but a slave may still have two bestbuy.com URLs from the previous window that it has not yet finished crawling.

3.3.3 Failure

One of the design goals of the crawler was fault tolerance. If a slave goes offline, the master will automatically detect that this has happened and stop distributing URLs to it. If the master had sent URLs to that slave and the slave went offline before notifying the master of its success, then the master will send the URLs to another slave for crawling. If the master goes offline, the slaves will simply continue listening for URLs, and will therefore be ready to continue crawling when a new master is started.

There are two failure scenarios that result in work being repeated:

1. A slave goes offline after successfully crawling a set of URLs, but before it is able to notify the master of its success

They should be good enough in practice since the slaves limit the time they’re willing to take crawling a particular URL, and URLs are crawled in parallel
2. The master goes offline after distributing a set of URLs, in which case the slaves will crawl them but will then be unable to report success to the master.

While these two scenarios can result in a URL being crawled more than once, there are no failure scenarios that can result in a URL being marked as crawled when data has not, in fact, been collected from it.

### 3.4 The Data Processing Pipeline

The purpose of the data processing pipeline is to take raw data collected by the crawler and convert it into a form that is consumable by the recommendation engine. The pipeline contains 4 primary steps:

1. Normalization
2. Aggregation
3. Attribute resolution
4. Attribute inference

The normalization step involves taking raw data stored as unformatted strings and converting it into a more meaningful representation. For example, the string "LG INFINIA 55" 1080p 240Hz LED-LCD HDTV 55LE8500" would be converted into multiple small units of data, one of which might look like:

- **ProductID**: TV-LG-55LE8500
- **DataType**: RESOLUTION
- **DataValue**: 1080p

The purpose of the aggregation step is to group all of the data produced by the normalization step by product. In the above example, all of the data for the LG Infinia 55LE8500 would be collected for processing together. This includes data collected from different sources (i.e. Best Buy, Amazon, etc.).

The attribute resolution step operates on all of the data for a single product and chooses a definitive value for each product attribute. For instance, if the crawler collected data on the LG Infinia 55LE8500 from multiple sources, these sources would be combined into a single value for each attribute. This includes detecting outliers. For example, if Best Buy and Newegg list the television as having a resolution of 1080p, but Amazon lists it as 1080i, the attribute resolution step would discard the outlier and choose 1080p as the value for the resolution attribute.

Once all of the attributes for a product are determined, the attribute inference step attempts to use the existing data to create new data. For example, a television’s resolution, brightness, response time and the presence of certain special features (such as the “gaming mode” featured by some Toshiba TVs) could be used to create a new attribute which represents how suitable the television is for playing video games.

### 3.5 The Web Application

In order to optimize the user experience and best meet user needs, the following usability principles were adhered to:

1. **Visibility** – Enhance the user’s experience by clearly displaying information that matters in a format that is meaningful.

2. **Feedback** – Information presented to the user must be informative and relevant to the inputs given. Accurate and honest feedback has been shown to increase user confidence and satisfaction [28].

3. **Constraints** – By limiting the complexity and options available to the user the entire experience is streamlined to fit his or her specific needs.

4. **Consistency** – Similar information must be conveyed using similar elements, layout, colors, etc.

Numerous market research studies have shown that adherence to these principles increases usability and contributes to high user satisfaction, likelihood of return, and frequency of use [25][17][32][29].

While there are many different aspects to the entire system, the web application is the only piece that the user sees and interacts with. Therefore, ease-of-use is not only a requirement but a top priority. The website was built with a simple and easy to follow layout in mind with end results only a few clicks away. In this manner, the user is able to get to his or her destination quickly and easily.

Upon arriving at the website, the user can immediately start answering questions and getting results. The questions are intentionally broad so that the system can extract more information without burdening the user with specifics. As each question is answered, all of the products’ cumulative weighting scores are recalculated, and the top four results are dynamically displayed at the bottom of the window. Additionally, based on the given responses, categories such as “Price” or “Size” are prioritized next to the questions. If the user disagrees on the extrapolated hierarchy, he or she can freely order the list and the corresponding scores change accordingly.

Pursuant to the transparency goal, the user can hover over any product result to see a complete breakdown of the different scores to understand why that particular product was recommended. Furthermore, upon selecting a product, a complete specification list along with pictures are provided as well as affiliate links to all of the suppliers.

### 3.5.1 Recommendation System

In order to ensure that questions are easy for all users to understand, the recommendation system avoids technological and jargon-heavy questions and focuses on how the user intends to use the product. This frees the user from having to understand terms such as resolution and refresh rate. For instance, questions like “Will you use this television to watch movies?” are preferred over questions like “Which is more important – resolution or refresh rate?”

Furthermore, all questions are optional, and users must be able to expect accurate recommendations when answering only a subset of questions.

To avoid overwhelming the user, the recommendation engine asks a small number of broad, information-rich questions. One way to achieve this goal is by combining multiple specific questions into a single broad question. For instance, one could combine “Will you use this television to watch sports?” and “Will you use this television to watch TV shows?” into “Check all of the following ways in which you intend to use this television”.

The recommendation system uses the user’s answers to infer which product features are most important to the user.
This relative importance is represented by assigning a numerical weight to each feature. Weights are values between -1 and 1, where 1 means that the user prefers a feature and -1 means that the user prefers not to have a feature. This not only captures which features are important to a user, but also how important the features are relative to one another.

In order for the system to be robust, the recommendation algorithm must have some notion of tradeoffs between features. This is particularly important when two features have equal weights. For example, consider two televisions $T_1$ and $T_2$ with two equally weighted features $F_1$ and $F_2$. Suppose $T_1$ has high $F_1$ and low $F_2$, and $T_2$ has low $F_1$ and high $F_2$. Without some notion of tradeoffs, there may be no way to choose between $T_1$ and $T_2$.

Questions are split into two different categories: reduction questions and weight questions. Reduction questions have answers that immediately reduce the size of the dataset. If a user indicates that he/she wants products cheaper than $500, this removes the need to consider any television that costs more than $500. After narrowing down the scope of potential products, weight questions are used to refine the attribute weights. Each potential answer to a weight question is associated with a predetermined set of weights. After answering a series of weight questions, the weights associated with the user’s answers are combined to produce a single set of weights for the user.

After a number of questions are answered and a set of weights is determined, a score is calculated for each product in the remaining dataset by multiplying each attribute by its associated weight. The sum of the weighted attributes is altered to account for tradeoffs in order to produce a final score. Products are then returned to the web application along with their scores and presented to the user in decreasing order of score.

4. SYSTEM IMPLEMENTATION

4.1 Languages and Tools

4.1.1 Ruby

Many of the system components (with the exception of the data processing pipeline) are written in Ruby [13]. The crawler uses Nokogiri, a Ruby library for parsing HTML, to aid in the extraction of data from web pages [10].

4.1.2 Thrift

Apache Thrift is a tool for defining cross-language remote procedures and serializable data types [31][1]. The crawler master and slaves communicate using a thrift interface, which allows the user to decouple their external interface from their implementation. This means that code written in languages other than Ruby can interface with the crawler (i.e. to query its status or enqueue new URLs). Data that is collected by the crawler is stored as serialized Thrift data types, which enables the data processing pipeline (which is written in Java) to easily deserialize and process the data.

4.1.3 Hadoop

Hadoop is an Apache project which contains two components that are used throughout the system – Hadoop MapReduce, and the Hadoop Distributed File System (HDFS) [5][19][6]. Hadoop MapReduce is a framework for performing large, distributed workloads in parallel, and is the primary tool used to create the data processing pipeline. HDFS is the accompanying file system which stores (a) data collected by the crawler and (b) intermediate data created by the data processing pipeline.

4.1.4 Cascading

Cascading is a Java library that simplifies process of building interconnected MapReduce jobs. The data processing pipeline uses cascading heavily.

4.1.5 PHP

The web application is written in PHP [11]. PHP was chosen because it is lightweight but is still able to interface with the recommendation engine (since it is one of Thrift’s target languages).

4.1.6 Python

The recommendation system is written in Python. Python provides an easy-to-use number handling data structures such as tuples. Additionally, the use of Python was motivated by the fact that it contains the “numpy” library, which can be used effectively if a machine learning component is to be added to the recommendation system.

4.2 The Crawler

![Figure 2: The Crawler](image)

Figure 2 shows a diagram of the crawler. Both the master and slave are written in Ruby and communicate using a Thrift interface. Crawl targets are described using a Thrift data structure with two fields:

1. **type** - An integer value representing the type of page represented by the target. For instance, the value 5 could represent a page on bestbuy.com for a single television
2. **url** - A string containing the URL of the target

4.2.1 The Master

The master exposes three primary remote procedures using Thrift:

1. **type** - An integer value representing the type of page represented by the target. For instance, the value 5 could represent a page on bestbuy.com for a single television
2. **url** - A string containing the URL of the target
1. `heartbeat()` - This method is called periodically by a slave to notify the master that it is running.

2. `report_success(crawl_target)` - This method is called by a slave to notify the master that `crawl_target` has been successfully crawled.

3. `enqueue(crawl_target)` - This method is called by a slave (or manually by a user) to enqueue `crawl_target` so it is crawled in the future.

The master stores all crawl targets in a MySQL database. Each crawl target has the following metadata:

1. `do_not_crawl_until` - The time at which the target will become eligible for crawling
2. `last_crawl_began_at` - The last time at which the master sent the target to a slave
3. `last_crawl_ended_at` - The last time at which a slave notified the master that the target was successfully crawled

When a slave calls `report_success()`, the crawl target’s `last_crawl_ended_at` field is updated and its `do_not_crawl_until` field is set to some time in the future. If a crawl target’s `last_crawl_began_at` becomes far enough in the past without `last_crawl_ended_at` being set, the master will consider the last crawl to be timed out (most likely due to slave failure) and will attempt to recrawl the target.

The master runs two threads concurrently. The first thread is responsible for responding to remote procedure calls. The second thread performs the following in a loop:

1. Fetch a large number of crawl targets from the database at random.
2. Group the targets by domain and
   (a) Limit the size of each group to `global_max_targets_per_domain`.
   (b) Further limit the size of each group according to `local_max_targets_per_domain` when applicable.
3. Limit the number of remaining targets to `max_targets_per_slave*n`, where `n` is the number of slaves.
4. Distribute the remaining targets in a round-robin fashion to each of the slaves
5. Sleep until the next window begins

### 4.2.2 Page Crawlers

For each type of page that the crawler is capable of crawling, there is a module called a page crawler that contains code for extracting data from that type of page. For instance, if we want to collect data on TVs listed on best-buy.com, we would need a page crawler that is capable of extracting product specifications, price, etc. from the HTML of a BestBuy TV page.

Page crawlers are written in Ruby and take parsed HTML as input. As output, they produce data extracted from the HTML along with a set of crawl targets containing other URLs from the page that should be crawled in the future.

### 4.2.3 The Slave

Like the master, the slave runs two threads concurrently, one of which is responsible for responding to remote procedure calls. The slave exposes one primary remote procedure using Thrift: `crawl(crawl_target)`. When this method is called, `crawl_target` is put on the end of an internal queue. The second thread performs the following in a loop:

1. Dequeue all targets from the internal queue
2. For each target, spawn a new thread that does the following:
   (a) Visit the target URL and download the HTML
   (b) Parse the HTML with Nokogiri
   (c) Provide the parsed HTML as input to the appropriate page crawler
   (d) Send the data output by the page crawler to Scribe
   (e) Send the links output by the page crawler to the master
   (f) Report success to the master
3. Send heartbeat to the master (if one hasn’t been sent recently)

### 4.2.4 The Collector

HDFS is optimized for storing large files – by default, a 1 kilobyte file requires the same amount of overhead as a 64 megabyte file. In addition, files in HDFS cannot be appended to after they are created. While HDFS was chosen primarily for use by the data processing pipeline, these two properties make it less than ideal for use by the crawler. Since the crawler rapidly produces data in small increments, the data needs to either be (a) stored in many small files or (b) appended to a large file. The collector was built to solve this problem.

The collector is a standalone application that is inspired by a similar application developed at BackType [33]. It is written in Java and exposes a Thrift interface which the slaves use to send data. Data is buffered locally in the collector and periodically dumped in large batches to HDFS. In order to avoid losing data that has been sent to the collector but not yet written to HDFS, the data is also written to a write-ahead log on the local filesystem as soon as it is received. In the event of a failure the log is replayed when the collector restarts and the data is written to HDFS.

In order to prevent the collector from becoming a point of contention, it is designed to run in a distributed fashion. The master keeps track of the set of active collectors, and the slaves periodically poll the master for an updated list. While crawling, the slaves spread the data evenly over all of the active collectors. If a particular collector is unreachable, the slaves simply send the data to a different collector.

### 4.3 Data Representation

Product data is split into atomic units called DataUnits (or DUs), which are stored as serialized Thrift objects. Describing DataUnits using Thrift makes it easy to exchange data between the crawler and collector, which are written in different languages.

This representation is similar to a graph database, and is loosely based on the graph-based schema described by Nathan Marz [24].
4.3.1 Raw Data

The crawler outputs a special type of DataUnit called a RawDataUnit (or RDU) which has the following fields:

1. identifier
2. product_type
3. data_type_or_type_hint (optional)
4. value

The identifier field is a string that is the same across all RDUs that are collected from a single page (and by the same page crawler). In practice, this is usually the raw string name that the target website gives to the product. It is worth noting that if two RDUs have the same identifier then they describe the same product, but if they do not have the same identifier then they may or may not describe the same product. For instance, an RDU that was produced while crawling Amazon is unlikely to have the same identifier as an RDU produced while crawling the same product on Newegg.

The product_type field is an integer that represents the type of data that the data unit describes, i.e., television, digital camera, etc. The data_type_or_type_hint field can contain either (a) an integer value representing exactly the type of data represented (i.e., the number 4 may indicate that this is a product’s price) or (b) a string containing a hint regarding the type of data that may be useful to the data processing pipeline. This is useful for product pages that contain a table of specifications where one column contains the name of the specification (i.e., “resolution”, “color”, etc.) and another column contains the value. In such cases, the first column can be provided as a type hint which the data processing pipeline can use to infer a type.

Lastly, the value field is a string containing the actual value of the data (i.e., “1080p”, “$700”, etc.).

4.3.2 Processed Data

The data processing pipeline takes RDUs as input and produces DUs as output. DUs have only two fields:

1. identifier
2. value

Unlike that of an RDU, the identifier field contains a normalized value that uniquely identifies a product. All DUs with the same identifier describe the same product, and all DUs that describe the same product have the same identifier. In practice, this value is usually comprised of the product’s type, manufacturer and model.

The value field is a struct that contains a normalized representation of the data itself. It roughly corresponds to a normalized combination of the data_type and value fields of an RDU.

4.3.3 Rationale

A coarser-grained data representation – such as storing data on the level of a single product rather than a single product attribute – may seem more natural than our fine-grained approach. However, this representation offers several advantages over a product-level representation:

Data can be added incrementally.

When the crawler discovers new data for an existing product, the data can be recorded by simply creating and storing a new RDU. If data were stored on a product level, this would instead require the crawler to search for the product in the existing datastore and update it with the new data.

Data can be processed efficiently.

Many of the individual components in the data processing pipeline only operate on a small subset of data. For instance, there is a single module that is responsible for normalizing the values of resolution data units. If data were stored on a per-product basis, this module would have to read in the entire raw datastore, update the “resolution” field on any products that have the field set, and then rewrite the entire store.

Schema updates do not affect existing data.

Collecting new types of data is as simple as writing new DataUnits. Existing data does not need to be updated with new fields, and existing data processing tools will not be impacted by the new data (since they only read in data that they know how to process, as described above).

Conflicting data is permitted at the schema level.

It is extremely useful to enforce consistency at the application level instead of the schema level. Although eventually the processed data should be consistent, the inconsistencies can be used in the unprocessed data to inform data processing decisions. For example, although every TV should ultimately only have one value for “resolution”, it is quite possible that a particular site has an incorrect resolution listed for the TV. In such a scenario it is desirable to store multiple RDUs (one from each data source) so that they can be considered when determining the actual resolution of the TV. If the crawler produces 4 RDUs that indicate a resolution of 1080p and a single RDU that indicates a resolution of 1080i, then it will be clear that 1080i is the outlier. If the crawler was only able to store a single value for each attribute, this data would be lost during crawling and would therefore not be available to the data processing pipeline.

4.4 The Data Processing Pipeline

The data processing pipeline consists of a number of Hadoop MapReduce jobs which are built with Cascading. The pipeline consists of the following steps:

Data Import

All of the data stored by the collector is copied to an intermediate location and is split up according to the type of data. RDUs are sorted and stored by product type, data type and type hint. This allows later steps to only read RDUs that they intend to process, as described above.

---

2The term consistency has many uses in computer science. In this instance we are referring to data that is not conflicting or contradictory. For instance, two “resolution” DUs for the same TV that contain different values would be considered inconsistent.
Type Inference
In this step, all RDUs with a type hint are examined and a normalized type is inferred. This is done primarily through a series of regular expressions. If a type cannot be determined, the RDU is discarded.\(^3\)

Raw Data Inference
This stage attempts to do some data inference at the level of a single data unit. For instance, consider an RDU whose type is \texttt{NEWEGG\_PRODUCT\_ID} and whose value is the string \texttt{LG 42LD520 42-Inch 1080p 120Hz LCD HDTV}. This RDU alone is enough to produce new RDUs with types \texttt{MANUFACTURER}, \texttt{MODEL}, \texttt{SIZE}, etc.

Normalization
Although all of the data units have proper types at this stage, they still contain raw string values. For each data type there is a module that processes these string values and produces a normalized value. For instance, the string \texttt{1080p} is transformed into the internal constant \texttt{RES\_1080\_P}.

Identifier Resolution
In this step, RDUs are grouped by their identifier and a normalized identifier is constructed. This is done by examining a subset of the RDUs in each group (such as manufacturer and model) and combining them in some canonical way. Although RDUs for the same product that are collected by different page crawlers will likely have different identifiers, this step will produce the same normalized identifier for each of them. This is therefore the first time in the pipeline when it is determined that different data sources are actually describing the same product. All of the RDUs have now been fully converted into DUs.

Consolidation
Now that data for a single product can be aggregated across data sources, it is likely that there are multiple DUs of the same type for each product (one from each data source). This step groups DUs by identifier and data type and decides on a single definitive value. For example, if there is a single TV with five different resolution DUs, the consolidation step will settle on a single resolution for the TV. This is usually as simple as choosing any DU and discarding the rest, but occasionally there is conflicting data which necessitates some basic outlier rejection.

Data Inference
At this point, each product has a definitive value for each attribute. Since the Raw Data Inference step only has access to RDU-level data and not aggregate product data, another round of data inference is necessary. For instance, it may be inferred that a TV is ideal for playing video games by examining its resolution, response time and brightness.

Data Export
Now that the data has been fully processed, it must be exported in a format that is consumable by the recommendation algorithm and web application. The DUs are aggregated by identifier and loaded into a MySQL database.

The data processing pipeline was designed with modularity and expansion in mind. Adding new types of data will not impact the existing pipeline. Furthermore, processing new types of data is accomplished by adding new modules to the Type Inference, Raw Data Inference, Normalization, and Data Inference steps (although it is usually sufficient to just add modules to the Type Inference and Normalization steps). New data sources can generally be added without any changes to the pipeline.

4.5 The Web Application

![Web Application WireFrame](image)

The web application was created using PHP [11]. A rich user experience, including animation and visual effects, was constructed using a combination of HTML, CSS, JavaScript and the jQuery JavaScript library [8]. Figure 3 shows a wireframe mockup of the user interface.

Because of the focus on sleek design and overall usability, dynamic pages and transitions were a necessity. jQuery was chosen because of its wide browser support and its ability to be run by users without the need to install additional software [4][8]. jQuery was used to create animated transitions between questions, dynamically updating the list of recommended products, and a drag-and-drop list of product attributes.

To provide adequate feedback, the interface includes a dynamic priority list that is populated according to the user’s answers. The list allows the user to understand how his or her answers determined the current product results and also allows the user to adjust the results by dragging and dropping. By using a small set of predetermined product questions, the web application is able to limit the amount of information required from the user to only the most relevant information.

As the user interacts with the various UI components, the results are sent asynchronously to the PHP application. The PHP application then communicates with the recommendation engine and returns updated recommendations. Product specifications are filled in using data stored in the MySQL database.

4.6 The Recommendation System

The recommendation system is written in Python and exposes a Thrift interface. The system has three primary tasks:
1. Receive the user's answers from the web application.

2. Generate weights based on the user's preferences and generate a ranked list of products.

3. Send the top 5 recommended products to the web application along with the weights.

After user input is received, the answered questions are divided into reduction questions and weight questions as described above. The reduction questions are used to construct a filter, and products in the database are only considered for recommendation when they are not filtered out.

Once the filter is constructed, the answers to the weight questions are examined to determine the corresponding weights. These predetermined weights play the role of the expert salesperson who knows about what makes a product good for a particular use. In order to produce an aggregate weight set, the system takes a vector-sum of the individual weight sets. This sum is then used to compute the score of each product using the following formula:

\[
\text{Score} = \sum_{f} \text{weight}_f \times \text{value}_f - \sum_{r} \log (|r_0 - r_1|)
\]

In the above formula, value\(_f\) refers to the deviation from the average of a particular feature \(f\), and weight\(_f\) refers to the weight for that feature. The purpose of the first term is to scale the quality of each feature by its importance to the user.

Initial versions of the recommendation formula only included the first term. This resulted in poor performance, even in unambiguous situations. Empirical tests indicated that the model was not properly capturing the idea of trade-offs. The second term was then introduced measure the tradeoff between certain pairs of features (\(r_0\) and \(r_1\) above) by measuring how far apart they are. This is meant to capture the following intuition: If a user says that feature \(F_1\) is extremely important and \(F_2\) is only slightly important, then we should reject products that have an extremely poor \(F_2\), even if it has extremely good \(F_1\). For instance, if a user that is shopping for a television says that resolution is much more important than refresh rate, we should reject TVs with extremely low refresh rates regardless of their resolution.

5. SYSTEM PERFORMANCE

Providing the crawler with a single newegg.com search results page as a seed URL resulted in the crawler discovering 119 new crawl targets, 112 products and a total of 5,143 RDUs (an average of just under 50 RDUs per product). Manual inspection of the data showed nearly all of the data to be accurate, complete, and useful for recommendations.

The Recommendation System performance was measured by running the algorithm on a modified database of televisions on certain input for which the output televisions are known. The accuracy of the initial algorithm was around 80%, and the accuracy of the final algorithm was around 94%. Recommendations were considered inaccurate if the top 5 recommended televisions differed from the known top 5.

6. ETHICS

ElectricDeel was designed with several ethical principles in mind:

1. **Recommendations must be trustworthy and unbiased.** The system is advertised as providing recommendations solely based on indicated user preferences, and maintaining the quality of recommendations is of utmost importance. Since the user places his/her trust in the recommendations, a recommendation based on anything other than user preferences – such as sponsored listings – constitute a violation of user trust.

2. **User data must only be used to compute recommendations.** Since a user is placing their trust in the system and providing personal information, any use of that information – such as selling data about consumer preferences – must not be done without the user's permission.

3. **The crawler must be a polite web citizen.** When collecting data from online retailers, we must ensure that we do not overwhelm or otherwise abuse any particular site. This was the primary motivation for building the master instead of a completely decentralized crawler.

7. FUTURE WORK

The following is a list of potential extensions to the work presented in this paper:

1. Expand to include many more product types and data sources.

2. Benchmark the system on a large cluster of machines.

3. Continue to iterate on the web application based on user feedback.

4. Continue to iterate on the recommendation engine based on user feedback.

5. Extend the recommendation engine to include a machine learning component.

8. REFERENCES


