AUTOMATED WORKFLOWS FOR DERIVING AND EXTRACTING METADATA FROM DISORGANIZED DATA SWAMPS

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

BY
TYLER J. SKLUZACEK

CHICAGO, ILLINOIS
MONDAY, NOVEMBER 19TH, 2018
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>iv</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>v</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>vi</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>vii</td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2 RELATED WORK</td>
<td>4</td>
</tr>
<tr>
<td>2.1 Data Lakes (and Spaces, Warehouses, and Marts)</td>
<td>4</td>
</tr>
<tr>
<td>2.2 Ingestion</td>
<td>5</td>
</tr>
<tr>
<td>2.3 Versioning</td>
<td>6</td>
</tr>
<tr>
<td>2.4 Schema Extraction</td>
<td>6</td>
</tr>
<tr>
<td>2.5 File Metadata Extraction</td>
<td>7</td>
</tr>
<tr>
<td>2.6 File Type Identification</td>
<td>10</td>
</tr>
<tr>
<td>3 PROBLEM DESCRIPTION</td>
<td>11</td>
</tr>
<tr>
<td>3.1 Metadata Extraction</td>
<td>11</td>
</tr>
<tr>
<td>3.2 File Types</td>
<td>12</td>
</tr>
<tr>
<td>4 SKLUMA REQUIREMENTS AND ARCHITECTURE</td>
<td>14</td>
</tr>
<tr>
<td>4.1 Requirements</td>
<td>14</td>
</tr>
<tr>
<td>4.2 Architecture</td>
<td>14</td>
</tr>
<tr>
<td>4.2.1 The Crawler</td>
<td>16</td>
</tr>
<tr>
<td>4.2.2 Extractors</td>
<td>16</td>
</tr>
<tr>
<td>4.2.3 The Orchestrator</td>
<td>18</td>
</tr>
<tr>
<td>4.2.4 File Updates</td>
<td>18</td>
</tr>
<tr>
<td>4.2.5 Output Metadata</td>
<td>19</td>
</tr>
<tr>
<td>5 EXTRACTORS</td>
<td>22</td>
</tr>
<tr>
<td>5.1 The sampler extractor</td>
<td>22</td>
</tr>
<tr>
<td>5.2 Content-Specific Extractors</td>
<td>23</td>
</tr>
<tr>
<td>5.2.1 Tabular</td>
<td>23</td>
</tr>
<tr>
<td>5.2.2 Structured</td>
<td>25</td>
</tr>
<tr>
<td>5.2.3 Unstructured</td>
<td>26</td>
</tr>
<tr>
<td>5.2.4 Images</td>
<td>27</td>
</tr>
<tr>
<td>6 EVALUATION</td>
<td>30</td>
</tr>
<tr>
<td>6.1 Evaluation</td>
<td>30</td>
</tr>
<tr>
<td>6.2 Extractor accuracy on CDIAC data</td>
<td>30</td>
</tr>
<tr>
<td>6.2.1 Accuracy of the sampler extractor</td>
<td>30</td>
</tr>
<tr>
<td>6.2.2 Accuracy of the topic extractor</td>
<td>31</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

1.1 Average file size vs. counts of common file extensions found in the CDIA data lake; 500,001 total files (excluding files with counts < 10) ........................................... 3

4.1 Skluma architecture showing the crawler used to traverse various file systems, the orchestrator for coordinating processing, an extraction pipeline composed of extractors for processing files, and the catalog containing extracted metadata. .... 15

5.1 Progression of metadata extraction tasks for example structured-unstructured hybrid file. ................................................................. 25

5.2 An example of location metadata extraction from maps. The red boxes delineate the extracted text; the white text is the tags generated for each region found in the map. ......................................................... 29

6.1 Throughput vs. file size on various container scalings: Local (top) and Cloud (bottom) ................................................................. 35

7.1 $F_1$ on Matrix M to format a vector as a raster. Black values are original, red are created on first sweep, and orange created on second. .................. 40

7.2 Work flow for Klimatic’s data integration and distribution. ................. 41
LIST OF TABLES

5.1 Skluma’s current extractor library. (Extensions listed are just examples.) . . . . 23

6.1 Distribution of file types, and median sizes per type, in the CDIAC repository. 30
6.2 Model accuracy and training and reading times, for 20,000 randomly selected
CDIAC files. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
6.3 Performance profiles for the crawler, extractors, and data download from S3 to
AWS compute, giving for each the number of CDIAC files processed and, for
those files, mean file size and execution time. For clarity, the decompression task
is separated from its parent extractor universal. . . . . . . . . . . . . . . . . . . . . . . 36
6.4 Time and cost to index all CDIAC files on AWS. . . . . . . . . . . . . . . . . . . . 36
ACKNOWLEDGMENTS

I foremost thank my advisors Dr. Ian Foster and Dr. Kyle Chard for guiding me through the process of preparing both the research and writing elements of this Master’s thesis. My grad school experience thus far has been jam-packed with support and opportunity, and I owe much of that to you both.

I thank Dr. Michael Franklin for serving on the committee for this thesis. You asked insightful questions and presented ideas that will certainly manifest in my future efforts.

I thank Dr. Ryan Chard for providing wisdom on Skluma’s system design, putting forth great effort on our recent e-Science paper, and for serving as my ‘third advisor’.

I acknowledge Dr. Aaron Elmore for his early insights on structured data processing, and for his ongoing support of my work.

I also acknowledge current and former students Rohan Kumar, Paul Beckman, Chaofeng Wu, and Emily Herron for their superb work in constructing metadata extractors and contributing to the larger body of this work. Thank you for teaching me a great deal about mentorship and collaborative research spaces.

Finally, I am grateful to my many colleagues in Globus Labs and the Computation Institute for the steady stream of insight, Whirlyball, and friendship.
ABSTRACT

In the face of high-velocity data expansion, institutions must provide rich, searchable metadata in order to maximize data utility. In practice this proves difficult as humans must wrangle these increasingly cumbersome data to extract latent metadata. To demonstrate how one can automatically develop a rich metadata index over large scientific repositories, this thesis presents Skluma – a system that automatically processes a target filesystem or repository, extracts content- and context-based metadata, and organizes extracted metadata for subsequent use. Skluma is able to extract diverse metadata, including aggregate values derived from embedded structured data; named entities and latent topics buried within free-text documents; and content encoded in images. Skluma implements an overarching probabilistic pipeline to extract increasingly specific metadata from files. It applies machine learning methods to determine file types, dynamically prioritizes and then executes a suite of metadata extractors, and explores contextual metadata based on relationships among files. The derived metadata, represented in JSON, describes probabilistic knowledge of each file that may be subsequently used for discovery or organization. Skluma’s architecture enables it to be deployed both locally and used as an on-demand, cloud-hosted service to create and execute dynamic extraction workflows on massive numbers of files. It is modular and extensible—allowing users to contribute their own specialized metadata extractors. Thus far Skluma has been tested on local filesystems, remote FTP-accessible servers, and publicly-accessible Globus endpoints. Applying Skluma to a scientific environmental data repository of more than 500,000 files demonstrated that it can effectively extract metadata from large scientific repositories. Finally, Skluma effectively extracted metadata from those files with modest cloud costs in a few hours.
CHAPTER 1
INTRODUCTION

Scientists have grown accustomed to instantly finding information, for example papers relevant to their research or scientific facts needed for their experiments. However, the same is not often true for scientific data. Irrespective of where data are stored (e.g., data repositories, file systems, cloud-based object stores) it is often remarkably difficult to discover and understand scientific data. This lack of data accessibility manifests itself in several ways: it results in unnecessary overheads on scientists as significant time is spent wrangling data [20]; it affects reproducibility as important data (and the methods by which they were obtained) cannot be found; and ultimately it impedes scientific discovery. In the rush to conduct experiments, generate datasets, and analyze results, it is easy to overlook the best practices that ensure that data retain their usefulness and value.

Many [18, 74, 28, 71, 55, 45] have developed systems that provide metadata catalogs to support the organization and discovery of research data. However, these approaches require upfront effort to enter metadata and ongoing maintenance effort by curators. Without significant upkeep, scientific data repositories and file systems often become data swamps—collections of data that cannot be usefully reused without significant manual effort [70]. Making sense of a data swamp requires that users crawl through vast amounts of data, parse cryptic file names, decompress archived file formats, identify file schemas, impute missing headers, demarcate encoded null values, trawl through text to identify locations or understand data, and examine images for particular content. As scientific repositories scale beyond human-manageable levels, increasingly exceeding several petabytes and billions of individual files, automated methods are needed to drain the data swamp: that is, to extract descriptive metadata, organize those metadata for human and machine accessibility, and provide interfaces via which data can be quickly discovered.

While automated methods exist for extracting and indexing metadata from personal and enterprise data [38, 31], such solutions do not exist for scientific data, perhaps due
to the complexity and specificity of that environment. For example, many of the myriad scientific data formats are used by only a small community of users—but are vital to those communities. Furthermore, even standard formats adopted by large communities are often used in proprietary and ad hoc manners.

In response, Skluma [63, 8] is a modular, scalable system for extracting metadata from scientific files and collating those metadata so that they can be used for discovery across repositories and file systems. Skluma applies a collection of specialized “metadata extractors” to files. It is able to crawl files in many commonly-used repository-types (e.g., object stores, local file systems) and accessible via various access protocols (e.g., FTP, HTTP, Globus). Skluma dynamically constructs a pipeline in which progressively more focused extractors are applied to different files and to different parts of each file. It automatically determines which metadata extractors are most likely to yield valuable metadata from a file and adapts the processing pipeline based on what is learned from each extractor.

Skluma’s capabilities are best evaluated using a disorganized, real-world scientific repository: the Carbon Dioxide Information Analysis Center (CDIAC) dataset. CDIAC is publicly available, containing 500,001 files ranging from tabular scientific data; photograph, map, and plot images; READMEs, papers, and abstracts; and a number of scientifically uninteresting files (e.g., Hadoop error logs, Windows installers, desktop shortcuts). CDIAC contains 152 file extensions, as shown in Figure 1.1. Further evaluation shows that Skluma can effectively extract metadata, with near-linear scalability.

The remainder of this thesis is as follows. Chapter 2 discusses related work in data lakes and metadata extraction. Chapter 3 formalizes metadata extraction and defines myriad scientific file types. Chapter 4 outlines Skluma’s overall architecture. Chapter 5 discusses Skluma’s built-in extractors. Chapter 6 evaluates Skluma from the perspectives of performance, correctness, and usability. Chapter 7 discusses a spatial data integration use case. Finally, Chapter 8 presents future work and concluding remarks.
Figure 1.1: Average file size vs. counts of common file extensions found in the CDIAC data lake; 500,001 total files (excluding files with counts < 10)
CHAPTER 2
RELATED WORK

Skluma is a dynamic pipeline system for extracting and deriving rich, searchable metadata for files in scientific data lakes. To this end, Skluma extends prior literature on data lakes, file ingestion, data versioning, schema extraction, file metadata extraction, and file type identification. Additionally, a number of systems index heterogeneous scientific data in myriad domain sciences.

2.1 Data Lakes (and Spaces, Warehouses, and Marts)

The data lake is a collection of raw data on which a schema is only enforced at query-time. Data lakes provide a flexible data storage alternative to competing organizational schemes (data warehouses, data marts, and databases) when users wish to store ‘everything’ to query later. The extract-transform-load (ETL) paradigm used by data warehouses [43] requires the system to transform data to fit a static schema. As schema-fitting is not always feasible with arbitrary scientific data, many data could remain uningested. By softening barriers to data ingestion (e.g., schema requirements and manual metadata curation), data lakes such as those built atop Skluma’s metadata indexes encourage insight by providing a low cost-of-entry means of turning disorganized scientific data repositories into rich, searchable data lakes.

Data lakes owe much of their current prominence to dataspaces [34, 57]. Dataspaces are an abstraction for providing a structured, relational view over unstructured, unintegrated data. Dataspaces apply a “pay-as-you-go” data integration model in which data integration tasks are performed only when needed, thereby reducing upfront data integration costs. Like Skluma, dataspaces do not assume complete control over the data, but merely integrate and coordinate the various methods of querying that data. Dataspace research builds upon related work in schema mapping, data integration, managing and querying uncertain data,
indexing, and incorporating human curation.

Data marts are (generally) smaller, domain-specific warehouses that partition data for individual subject areas into disparate stores. A primary goal of the data mart is to streamline the management and data mining tasks of each data partition [50, 37]. When a data mart is built atop a data lake, it proves imperative to extract descriptive file metadata in order to ensure that the system places data into the most relevant dimension (i.e., near data with similar themes or access patterns). EnsMart [44] is a biological data mart with partitions devoted to a number of species, and further sub-partitions devoted to genomes, disease expressions, and protein annotations.

\subsection*{2.2 Ingestion}

File ingestion represents a cornerstone of data lakes systems, and refers to the steps necessary for placing data into the system. Google Goods [38] is an enterprise data lake system that builds a centralized metadata index over data distributed across a number of Google’s data stores. Goods automatically extracts metadata from BigTable [17], Spanner [23], and a number of local file system to build a rich collection of data sets, logs, source code repositories, human-input, and the file system. IBM PAIRS [46] conversely opts to funnel files from multiple native host repositories to a single HDFS cluster. At the expense of larger storage and data movement requirements at ingestion-time, PAIRS’s architecture facilitates fast access to files downloadable by IBM’s consumers. DataONE [48] presents an interesting case where data are stored in multiple locations external to the metadata index. DataONE ingests data from private industry, academia, community, government, and non-profits while allowing its data to live in their natural habitat ‘member nodes.’ Each file’s metadata are relayed back to DataONE’s server via lightweight REST API calls. Such a setup theoretically minimizes file transfer and storage costs by the data lake’s administrators during each file’s metadata extraction.
2.3 Versioning

Data versioning allows the contents of a data lake to remain ‘fresh’ long after the initial metadata extraction. Data context [41] refers to a series of metadata elements that would ideally accompany each file, illustrated by the acronym ABC. (A)pplications refers to the core information as to how and why the data are to be used. (B)ehavior traces how the data have been used over time. Finally, (C)hange refers to both the alterations to data structure and values over time. A number of data versioning systems exist, but they generally only have a subsets of the ABCs (i.e., they are generally missing applications). DataHub [11] is a portal (similar to GitHub) constructed over a dataset version control system that captures the modifications to data sets over time to be communicated with the dataset’s stakeholder scientists. Applications and behaviors are not mandatory to the version control schema, and tracking the behavior should prove difficult when the dataset is forked. Moreover, this approach is designed only for structured datasets. Construction of an automated data versioning system that adheres by the ABCs is still an open research area.

2.4 Schema Extraction

Schema extraction refers to the process of discovering the structure of arbitrary data, including fields, types, null values, and delimiters. Current methods focus on well-structured data with greater uniformity than data sources like CDIAC. Tools such as RecordBreaker [21], PADS [31, 32], and Catamaran [35] represent solutions to the problem of identifying columns. Their models, while simplistic, are surprisingly accurate. The basic approach uses a set of possible delimiters and attempts to parse file rows using this set. The tools calculate the number of columns per row and overall variance across many rows. If the variance is below a predefined threshold the column proposition is accepted. For each column, the tools then analyze individual values to determine type. A histogram of values is created. If the values are homogeneous, a basic type can be easily inferred from the histogram. If the values are
heterogeneous, then the column is broken into a structure and the individual tokens with
the structure are analyzed in a recursive process.

Hybrid files add complexity to schema extraction, and are a primary motivator in Skluma,
as data in scientific repositories are often concatenations of various types. For instance,
a free text preamble followed by a row of header labels and data will be miscategorized
using the aforementioned models. Prior work exists in using multi-hypothesis delimiter-
separated-file parsing to extract a schema from untidy data when nulls and table structures
are unknown [27]. Predictive user interaction affords a system high-accuracy in identifying
multiple schemata, but does not allow full automation [40]. Google’s Web Tables [16]
harnesses advanced table interpretation at scale, but assumes a great deal of table structure,
including the existence of a header row.

Some have even gone as far as to reverse-engineer input formats with a rich information
set, including the record values, types, and constraints on the input [24]. Their approach (1)
uses generative code (e.g., scripts that generate data) to reverse-engineer field sequences and
(2) applies clustering to records into a small set of types based on the steps used to process
the record, and (3) infers constraints by tracking symbolic predicates from dynamic analysis
of data flow.

2.5 File Metadata Extraction

Skluma follows a long line of attempts to build simplistic models for gaining insight into
highly disorganized data.

Skluma is also not the first to provide a scalable solution to collect raw data sets and
extract metadata from them. Pioneering research on data lakes has developed methods for
extracting standard metadata from nonstandard file types and formats [70]. Recently the
data lake has been adapted to standardize and extract metadata from strictly-geospatial
data sets [62]. Normally data lakes have some sort of institution-specific target for which
they are optimized, whether they primarily input transactional, scientific, or networked data.
Skluma is optimized for data without any standardization guarantees, providing information on related, relevant data and their attributes.

While most related research performs metadata extraction to enable search, Skluma-like systematic sweeps across repositories can also be used for analysis. For example, the Big Data Quality Control (BDQC) framework [26] sweeps over large collections of biomedical data without regard to their meaning (domain-blind analysis) with the goal of identifying anomalies. BDQC employs a pipeline of extractors to derive properties of imaging, genomic, and clinical data.

Recent advances in image processing technology have enabled a wide range of advanced applications. Image extraction techniques support not only content-based feature extraction but also semantic feature extraction. For instance, Facebook identifies objects and people in photos, allowing for queries based on the objects within images (e.g., sheep or mountains) and for reverse image searches used to discover related images. Many researchers have investigated the use of context extracted from surrounding text to assign labels to images in articles [60]. These tasks are often referred to as Content-Based Image Retrieval (CBIR) systems. Libraries such as TensorFlow, scikit-learn, ImageJ, and MATLAB support feature extraction and image clustering. As such, image extraction techniques have been applied to a broad subset of science: identifying brain tumors [58], classifying materials microstructures [65], and mining satellite imagery [54].

Methods have also been developed for extracting data from web pages (e.g., again, Google’s Web Tables [16]), for example to mine real-estate pages to determine information about properties, such as type, size, number of bedrooms, and location. These tools commonly seek to identify structured components of the page (e.g., using HTML structure) and then apply rules to extract specific information. Systems for extracting information from scientific publications and electronic medical records work in similar ways, by locating structured information and then extracting features. A wide range of techniques have been explored for such tasks, including the use of templates, rules [15], schema mining [61], path
expressions [30], and ontologies [68]. Recently many of these techniques have been applied in machine learning approaches [67].

Human feedback improves machine learning models for metadata extraction by both supplying ‘ground truth’ input-labels and validating outputs [51]. Existing work in medical data lake curation requires users to manually tag (from a finite set of tags) all uploaded images in order to augment extracted metadata [73]. Such a system sacrifices scalability in order to tightly tether the metadata to a curator-defined ontology. Others have used text extraction to isolate polymer information from scientific literature via model training by a semi-expert crowd of students [69]. This method appears to provide greater scalability than manual metadata upload, as the students provide feedback on a small portion of the entire paper set.

Skluma supplements work on cleaning and labeling data via context features. Data Civilizer [25] accounts for proximate files in data warehouses by building and interpreting linkage graphs. Others have used context as a means to determine how certain metadata might be leveraged to optimize performance in a specific application or API [42]. Skluma collects and analyzes context metadata in order to allow research scientists to find, query, and download related datasets that aid their scientific efforts.

Finally, a number of full systems exist for automatically extracting metadata from repositories. ScienceSearch [56] uses machine learning techniques to extract metadata from a dataset served by the National Center for Electron Microscopy (NCEM). A majority of data in this use case are micrograph images, but additional contextual metadata are derived from file system data and freetext proposals and publications. Like Skluma, ScienceSearch provides a means for users to extensibly switch metadata extractors to suit a given data set. Brown Dog [52] also presents an extensible metadata extraction platform, providing metadata extraction services for a number of disciplines ranging from materials science to social science to humanities.
2.6 File Type Identification

There exists significant prior work on techniques for classifying file types or internal file structure within the anti-virus and digital forensics literature. This review stems from an extant review in this space [6]. Recent work has found that using k-nearest-neighbors (kNN) or Support Vector Machine (SVM) classification using n-grams can identify file types with about 90% accuracy [36]. The authors report this performance with 2-gram features.

Other work in this area has tried to identify methods to make file identification more efficient. Building a histogram of n-grams for a large file is not efficient, especially if it is possible to guess the file type from the first few bytes. Some have investigated feature selection techniques and found that sampling from the beginning of the file results in little accuracy loss for text files [5]. Further, while many have applied information theoretic measures to the problem, they have at most used entropy of a file as a feature [33, 10, 39].

Despite being a relatively intuitive metric, JS divergence has likely been overlooked because the FTI literature is generally focused on the identification of particular, known file-types. As a practical matter, it is unlikely that a forensic investigator will encounter a totally new file type. They are more likely to be interested in distinguishing a PDF from a Microsoft Word document specifically, than two arbitrary files taken from the universe of possible file formats.
CHAPTER 3

PROBLEM DESCRIPTION

As of now, existing research does not provide a tight, formalized language for metadata extraction workflows, including the inputs, processing steps, and outputs. This chapter presents this formalization as well as a crystallized ontology of the file types on which metadata extraction occurs.

3.1 Metadata Extraction

Skluma performs end-to-end metadata extraction using a series of encapsulated tasks called extractors. These extractors are applied to files within a repository to derive structured property sets of metadata. This is formalized as follows.

A repository $R$ is a collection of files, with each $f \in R$ comprising a file system path and a sequence of bytes, $f.p$ and $f.b$. A property set $M$ is a collection of metadata, with each $m \in M$ comprising a file system path, a metadata element, source extractor, and timestamp, $m.p$, $m.e$, $m.s$, and $m.t$, respectively. A property set $M$ is said to be valid for a repository $R$ iff for each $m \in M$, there is an $f \in F$ such that $m.p = f.p$. The metadata associated with a file $f \in F$ are then $M_f = m \in M : m.p = f.p$.

Skluma contains a set of extractors, $E$. An extractor $e \in E$ is a function $e(f, M_f)$ that returns a (potentially empty) set of new metadata elements, $N$, such that for each $m \in N$, $m.p = f.p$ and $m.s = e$.

The metadata extraction process for a file $f$ proceeds by: (i) calling a function $\text{next}(M_f, h_f)$, $h_f$ being the (initially empty) sequence of extractors applied to $f$ so far, to determine the extractor $e \in E$ to apply next; (ii) evaluating $N = e(f, M_f)$; and (iii) adding $N$ to $M$ and $e$ to $h_f$. These steps are repeated until $\text{next}(M_f, h_f) = \phi$.

Note that in this formulation of the metadata extraction process, each extractor $e$ is only provided with access to metadata, $M_f$, about the specific file $f$ that it is processing:
thus information learned about one file cannot directly inform extraction for another. This restriction is important for performance reasons. However, as discussed in the following, it proves helpful for both extractors and the next function to be able to access models created by analysis of metadata extracted from previously processed files.

3.2 File Types

Scientific repositories and file systems contain a wide range of file types, from free text through to images. The methods by which meaningful metadata can be extracted from file types differs based on the representation and encoding. Isolating a number of prevalent scientific file type classes has led to the construction of each type’s corresponding extractor(s). While recognizing that there are hundreds of well-known file formats [72], efforts thus far have focused on the following file type classes:

**Unstructured:** Files containing free text. Often human-readable natural language and encoded in ASCII or Unicode. Valuable metadata are generally related to the semantic meaning of the text, such as topics, keywords, places, etc. Unstructured files are often stored as .txt or README files.

**Structured:** Encoded or organized containers of data in a predefined file format. Often includes self-describing metadata that can be extracted through standard tools and interfaces. Common formats include JSON, XML, HDF and NetCDF files. Structured files may contain encoded null values.

**Tabular:** Files containing tabular data that are formatted in rows and columns and often include a header of column labels. Metadata can be derived from the header, rows, or columns. Aggregate column-level metadata (e.g., averages and maximums) often provide useful insights. Structured files are often stored as .csv and .tsv files.

**Images:** Files containing graphical images, such as plots, maps, and photographs. Common formats include .png, .jpg, and .tif.

**Compressed:** This class encompasses any file type that can be decompressed with
known decompression software to yield one or more new files. Example extensions are .zip and .gz. The new file(s) may or may not be of types listed here.

**Other:** Files that are not recognized as belonging to one of the types listed above. In the CDIAC repository, there exist Microsoft .docx and .xlsx files, for example.

**Hybrid:** Files containing multiple data types, such as tabular data with free text descriptions and images with text labels.
CHAPTER 4
SKLUMA REQUIREMENTS AND ARCHITECTURE

Skluma provides a system for scientists to automatically extract and derive latent metadata from a diverse file set. In practice, Skluma points at a scientific repository, iteratively extracts metadata from each stored file by dynamically assembling a pipeline of extractors specific to that file, and pushes the metadata to a search service. The remainder of this chapter outlines Skluma’s system requirements and provides a description of its architecture.

4.1 Requirements

Skluma is designed to address several requirements: extensibility and modularity to enable customization for different repositories and domains (e.g., via implementation of new crawlers or extractors), scalability and performance to support the processing of large repositories, smart and adaptive extraction pipelines that learn and guide extraction decisions, and comprehensibility to capture a diverse set of metadata with different levels of accuracy and value.

4.2 Architecture

Skluma is designed as a web service, with a JSON-based REST API that users can use to request metadata extraction from either an entire repository or a single file. An extraction request defines the repository or file to be processed, the crawler to use, optional configuration settings, and any custom extractors to be considered in the extraction process.

Skluma (see Figure 4.1) comprises three core components: a scalable crawler for processing target repositories, a set of extractors for deriving metadata from files, and an orchestrator that implements the API, invokes crawlers, manages metadata extraction for each file identified by a crawler, and manages the resources used by crawlers and extractors. The following contains details on each component.
The orchestrator, the crawler, and the different extractors are each packaged in separate Docker containers. This architecture makes scaling trivial. The crawler can run concurrently with extractors, and additional container instances can be created as needed to crawl multiple repositories at once and/or to extract metadata from multiple files at once. A well-defined extractor interface enables modularity and extensibility.

Skluma can be deployed both locally and in the cloud. In the first case, the entire Skluma stack is deployed locally and containers are executed on the host. In the cloud-based model, the Skluma service is hosted in one cloud instance, and crawler and extractor instances are executed in single-tenant cloud containers.

In both local and cloud deployments, users configure the maximum number of concurrent container instances to be run, currently $n \geq 3$ (1+ crawlers, 1 orchestrator, and 1+ extractors). In local deployments, Skluma manages container execution using Docker. In cloud deployments, Skluma uses the Amazon Web Services (AWS) Elastic Container Service
(ECS) to manage the allocation of compute resources and the mapping of container instances onto those compute resources.

4.2.1 The Crawler

Skluma can be applied to a wide range of data stores, from local file systems through to web-based data repositories. To this end, Skluma has a modular data access interface with functions for retrieving file system metadata, accessing file contents, and recursively listing folder contents. Implementations of this interface exist for different storage systems: so far, HTTP, FTP, Globus, and POSIX.

The orchestrator invokes the crawler in response to an extraction request. The crawler is launched within a container to process an entire repository. Starting from one or more root locations, it expands directories recursively and records each file that it encounters in a database as a (unique identifier, file system path) pair.

If multiple repositories need to be crawled, the orchestrator can launch multiple crawler tasks at once, each within its own container.

4.2.2 Extractors

An extractor is an application that, when applied to a file plus a set of metadata for that file, generates zero or more additional metadata items. Most extractors are type-specific. For example, they may be designed to extract values from a tabular file, content from an image, or semantic meaning from text. This section introduces two preparatory extractors, universal and sampler; the other seven extractors developed to date are described in Chapter 5.

The universal extractor extracts file system metadata—(e.g., name, path, size, extension)—and computes a file checksum that can be used to identify duplicate files. When the file extension suggests a known compression format (e.g., .gz, .zip), it attempts to decompress the file with the appropriate decompressor, and upon success adds the resulting file(s) to the
set of files to be processed.

The sampler extractor reads selected file contents and uses that information, plus file system metadata extracted by universal, to provide an initial guess at file type. For example, if extracted text are all ASCII and the filename prefix is .txt, sampler may infer that a file is a text file. An in-depth description of sampler appears in Section 5.1.

Subsequent extractors applied to a file vary according to both the file’s inferred type and extracted metadata. For example, if sampler infers a file type of “text,” then Skluma’s next function may next select a text extractor. If the file furthermore contains both lines representing tabular data (e.g., consistent delimiters and uniform inferred column types) and a free text header (e.g., describing the project or experiment), the file will also be processed by tabular and free text extractors. If a tabular extractor identifies a high probability of null values (e.g., repeated outliers or non-matching types), Skluma will apply a null value extractor to the file. Thus two files of the same type, as inferred by sampler, can have quite different extraction pipelines.

An extractor is implemented, either by itself or colocated with additional extractors, in a Docker container with a common interface. This organization provides for modularity, extensibility, enables trivial scaling, and simplifies execution in heterogeneous computing environments [47]. Extractors are registered with Skluma through a simple JSON configuration that defines the unique extractor name (i.e., container name used for invocation) and hints about the file types that it can be used to process. Extractors must implement a simple interface that accepts three parameters: a unique resource identifier (URI) to a file (e.g., the path to a file in a Globus endpoint) and that file’s current metadata (JSON). An extractor returns either a (perhaps empty) set of extracted metadata or an error.

Extracted metadata are added to set of metadata associated with the file, with, as noted in Chapter 3, the extractor name appended to indicate the metadata’s source. This approach allows for disambiguation when multiple extractors assert the same value.

To improve performance by enabling data reuse, extractors can be co-located within a
single container. This is especially beneficial in the case of extractors that are often applied to the same file. For example, co-locating the free text topic extractor with the tabular data extractor can prove useful in situations where tabular files include free text preambles.

4.2.3 The Orchestrator

The orchestrator manages the end-to-end extraction process. It implements the API by which users submit, and monitor the progress of, extraction tasks. For each such task, it is responsible for invoking the crawler on the repository named in the request, coordinating metadata extraction for each file identified by the crawler, and adding the extracted metadata to the metadata database. It is also responsible for managing the pool of resources used by crawlers and extractors.

The orchestrator processes each file identified by the crawler by repeatedly calling the next function (see Section 3) to determine the next extractor and then running that extractor on the file and its metadata, until the next function indicates that there are no more extractors to run. (The extraction process for a file currently terminates when the last extractor fails to identify future extraction steps for next. One could also consider other factors, such as a limit on the resources to be spent extracting metadata.) The sequence of extractors thus applied to a file constitutes the file’s extraction pipeline.

Skluma extractors can, in principle, be applied in any order. However, it will typically apply universal first, to extract file system metadata and to process compressed files, then sampler to infer file type, and then various increasingly specific extractors depending on that type.

4.2.4 File Updates

The constant churn of scientific file systems and repositories makes static indexing unfeasible. For a metadata catalog to be useful its contents must be up-to-date, reflecting changes to files as they are added, copied, or edited. Techniques to dynamically construct, update,
and curate metadata catalogs are required for them to provide the most use to researchers. To facilitate online metadata extraction and enable Skluma to update itself and its catalogs in real-time, it uses the Ripple [19] event detection service. Ripple enables both users and services to register for data triggers and define the actions to be performed in response. Ripple agents can monitor local file systems, Globus endpoints, and large, leadership-scale, storage devices. Using Ripple, Skluma is able to respond to file creation, modification, and deletion events, re-processing files as they are modified and created. This enables Skluma to automatically maintain real-time catalogs of repositories. When a file is edited or added to a file system, Ripple triggers a Globus transfer of the file to a cloud object store where a waiting Skluma cloud service can find the file, invoke an extraction pipeline, and update its metadata elements. When the file is initially added to the system, it gets an entirely new metadata document. When that same file is edited, the file is reprocessed entirely, swapping the old metadata for the new. Chapter 2 explores the challenges of partial metadata updates.

4.2.5 Output Metadata

The output of the pipeline is a metadata document for each processed file, as shown in Listing 4.1: a hierarchical JSON document that contains both file system metadata (ownership, modification time, and path) and the information generated by each extractor. Each extractor will generate extractor-specific metadata from a given file. For example, the tabular data extractor, commonly applied to comma-separated-value files, will create a list of header descriptions and aggregate, column-level information (e.g., type; min, max, average values; precision; and inferred null values).

One important use for extracted metadata is to implement a search index that allows flexible search across files in a repository. Thus, metadata are exported to Globus Search—a scalable cloud-hosted search service with fine grain access control and user managed indexes [7]. To do so, Skluma translates the metadata output into the GMeta format required by Globus Search and, via the Globus Search API, pushes the metadata to a user-defined
index.
Listing 4.1: Example Skluma metadata for a hybrid precipitation data file containing a free text header and structured data; processed on a local machine. Universal, sampler, tabular, and topic extractors operate on the file.

```json
"metadata":{
  "file":{
    "URI": "local::/home/skluzacek/data/precip08.csv",
    "updated": "03/08/2018-15:08:4537",
    "owner": "skluzacek"
  }
  "extractors":{
    "ex_universal":{
      "timestamp": "03/08/2018-15:25:1233",
      "name": "precip08",
      "ext": "csv",
      "size": "3091287"
    }
    "ex_sampler":{
      "timestamp": "03/08/2018-15:26:3163",
      "tabular-header"
    }
    "ex_tabular":{
      "timestamp": "03/08/2018-15:27:5390",
      "headers": ["id", "latx", "date"],
      "cols":{
        "id":{
          "type": "str"
        },
        "latx":{
          "type": "latstd-float",
          "prec": ".1",
          "null": "-999",
          "min": "0.0",
          "max": "180.0",
          "avg": "91.5"
        },
        "date":{
          "type": "datetime",
          "early": "02/04/2014",
          "late": "06/06/2014"
        }
      }
    }
    "ex_topic":{
      "timestamp": "03/08/2018-15:27:6340",
      "type": "preamble",
      "keywords": ["precipitation": 0.8],
      "topics": ["atmospheric science", "climate science", "oceanography", "meteorology", "weather"]
    }
  }
}
```
CHAPTER 5
EXTRACTORS

Given the tremendously wide range of scientific file formats, Skluma presents a modular
architecture that allows user-defined extractors to be plugged in easily. Section 4.2.2 in-
troduced the universal and sampler extractors; this chapter describes these extractors in
more detail, and also introduces extractors for tabular, text, and image formats.

5.1 The sampler extractor

Applying every Skluma extractor to every file would be both time consuming and un-
necessary: for example, there is little insight to be gained from applying an image extractor
to structured text. In order to identify a suitable first extractor(s) to apply to each file, the
sampler extractor samples and predicts each file’s content. It uses the metadata captured
by universal in combination with a (configurable) sample of $N$ bytes in a file. The sample
is then converted into features that are used by a machine learning model to predict a file’s
type. The file’s type is used to bootstrap the selection of which other extractors should be
applied to the file.

Any preprocessing done during the sampling process can be costly, for example to de-
termine fields, delimiters, or even lines. To avoid such preprocessing, sampler computes
byte $n$-grams from the sampled content and trains the set of selected models to determine
the file type [9]. Varying the size of $n$-grams showed that anything greater than $n = 1$ was
prohibitively slow with negligible improvements to accuracy. One could also sample the file
in multiple ways: head (all from top of the file) and randhead (half from top of file, half
from randomly throughout the rest of the file). Chapter 6 discusses the performance of these
approaches.

There are many machine learning models that perform well in supervised classification
problems including Support Vector Machines (SVMs), logistic regression, and random forests,
Table 5.1: Skluma’s current extractor library. (Extensions listed are just examples.)

<table>
<thead>
<tr>
<th>Name</th>
<th>File Type</th>
<th>Brief Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>universal</td>
<td>All</td>
<td>Compute file hash and record file system information (size, name, extension)</td>
</tr>
<tr>
<td>sampler</td>
<td>All</td>
<td>Generate hints to guide extractor selection</td>
</tr>
<tr>
<td>tabular</td>
<td>Column-structured CSV, TSV</td>
<td>Column-level metadata and aggregates, null and header analyses</td>
</tr>
<tr>
<td>null</td>
<td>Structured CSV, TSV</td>
<td>Compute single-class nulls for each column</td>
</tr>
<tr>
<td>structured</td>
<td>Hierarchical HDF, NetCDF</td>
<td>Extract headings and compute attribute-level metadata</td>
</tr>
<tr>
<td>topic</td>
<td>Unstructured Tst, Readme</td>
<td>Extract keywords and topic tags</td>
</tr>
<tr>
<td>image</td>
<td>Image PNG, TIF</td>
<td>SVM analysis to determine image type (map, plot, photograph, etc.)</td>
</tr>
<tr>
<td>plot</td>
<td>Image PNG, TIF</td>
<td>Extract plot titles, labels, and captions; assign axis-ranges and keywords</td>
</tr>
<tr>
<td>map</td>
<td>Image PNG, TIF</td>
<td>Extract geographical coordinates of maps; assign coordinates and area-codes</td>
</tr>
</tbody>
</table>

however all require “ground truth” labels for training. As all repositories are different and users may provide their own extractors, Skluma supports training on a subset of the repository. Skluma randomly selects a configurable percentage (e.g., \( k = 20\% \)) of files from a given repository. It will then run these files through the pool of extractors and determine for which extractors metadata are successfully extracted (i.e., when a given extractor does not throw an \texttt{ExtractionError}). Once \( k\% \) of the files are processed, Skluma trains a model on these results and the model is stored in the \texttt{sampler} container’s data volume for future prediction.

### 5.2 Content-Specific Extractors

Having identified the likely file type, the next stage of the pipeline focuses on content-specific extraction. Skluma provides an initial set of extractors designed for common scientific file types. Although these extractors target specific file types, they are not mutually exclusive. For example Skluma may apply a tabular extractor to one part of a file and also apply a topic extractor to another part of that file. The remainder of this section outlines several Skluma extractors. These extractors are summarized in Table 5.1.

#### 5.2.1 Tabular

Scientific data are often encoded in tabular, column-formatted forms, for example, in comma-separated-value files. Such data might appear trivial to parse and analyze. However, many scientific datasets do not conform strictly to such simple specifications. For example,
CDIAC contains many column-formatted data files with arbitrary free text descriptions of the data, multiple distinct tabular sections (with different header rows and number of columns), missing header rows, and inconsistent delimiters and null value definitions.

Skluma’s **tabular** data extractor is designed to identify and extract column-formatted data within text files. Its implementation applies various heuristics to account for the challenges described above. For example, if the first line of a file is not recognizable as a header row (e.g., it has a different number of columns to the subsequent lines), the extractor initiates a binary search over the file to identify the start of the delimited data. The extractor also works to identify columns (e.g., by identifying delimiters and line-break symbols) and to derive context (e.g., name, data type) and aggregate values (e.g., the mode, min, max, precision, and average for a numerical column; the bounding box for a geospatial column; and the $n$ most common strings in a text column).

Computing aggregate values is not without risk. For example, sometimes **tabular** finds ad hoc representations (e.g., empty string, “null,” or an unrealistic number such as $-999$) used to encode missing values. Without knowledge of such schemes, aggregate values can be meaningless. Furthermore, detecting their use can require domain knowledge: for example, $-99$ degrees is an unrealistic temperature for Chicago, but not for Neptune. To capture such patterns the **null** inference extractor employs a supervised learning model to determine null values so that they can be excluded from aggregate calculations. This is done via the $k$-nearest neighbor classification algorithm, using the average, the mode, the three largest values and their differences, and the three smallest values and their differences as features for the model. By taking a classification rather than regression-based approach, **null** selects from a preset list of null values, which avoids discounting real experimental outliers recorded in the data.
5.2.2 Structured

Structured files contain data that can be parsed into fields and their respective values. Extracting this data line-by-line proves ineffective due to each employing a specialized file schema. The \textit{structured} extractor decodes and retrieves metadata from two broad classes of structured files: data containers and documents. \textit{structured} first uses a simple set of rules to determine in which of the two classes the file belongs.

Containerized data formats (e.g., HDF, NetCDF) encode metadata in standard formats that are accessible via standard interfaces. Once a file is determined to be containerized, the \textit{structured} extractor first uses a second set of rules to determine the container type (e.g., HDF, NetCDF) and then calls appropriate container-specific libraries to retrieve header values and in-file self-describing metadata, and to compute a series of aggregate values for each dimension.

Structured document formats (e.g., XML, JSON, etc.) contain a number of key-value pairs. For scientific data, these keys can be considered “headers” and the values corresponding data. \textit{structured} automatically parses the keys from the file and computes aggregate values for each key containing numerical lists, while also storing the nonnumerical keys as a metadata attribute list.
Files in scientific repositories often contain free text (mostly, but certainly not always, in English). Text may be found, for example, in purely or primarily textual documents (e.g., READMEs, paper abstracts) or within certain fields of a structured document. Purely textual documents may be intended as documentation for other files, as sources of data in their own right, or for other purposes. For now the topic metadata extractor is designed to process free text within files. It takes a blob of free text as an input, and derives keyword and topic labels. topic performs three types of free text extraction: structured text extractors, latent keyword extractors, and topic models.

Structured extractors allow the topic extractor to collect domain-specific metadata. In the case of CDIAC, geographical attributes such as geological formations, landmarks, countries, and cities are potentially valuable metadata attributes, but are often hidden deep within files. The Geography [1] text-mining Python library is used to provide this capability. This library is representative of text extraction processes using a well-known ontology or vocabulary of attributes to be extracted. The output of the model is an unranked list of geographical attributes, including both man-made and natural landmarks, cities, states, and countries.

Many important metadata attributes do not map to a published ontology or vocabulary. In this case, more general latent keyword extraction can provide meaningful metadata. Here, a multi-stage approach is applied, first using a simple, non-probabilistic model to identify keywords and then an additional topic mixture model to find related topics.

While this keyword extraction method can extract words within the document, it often does not provide sufficient semantic information about a document (e.g., related to words not explicitly included in the text). Thus word embeddings prove useful in this context—specifically, pre-trained word embeddings from the GloVe project [53]—as the second stage [49]. Using the corresponding vectors of the top-$n$ keywords, an aggregate vector is computed to represent each document, weighted by the amount of information they provide (as above).
Since each document is now represented by a vector in the space of word embeddings, it can be used as a search facet when searching via topics, as one can easily retrieve the closest words to that vector representing a topic. More specifically, any query topic string could be converted to one or more vectors in this space, and the search could return those documents which are semantically closest to these query vectors. Another benefit of having document vectors in a space that preserves semantic relationships is that documents can be compared using their vectors to ascertain the similarity between their main topics.

**topic** also uses a topic modeling approach to extract contextual topics that provide even more information when combined with a text file’s keywords. For this purpose, the extractor leverages a topic mixture model based on Latent Dirichlet Allocation (LDA) [13]. The topic model trains on Web of Science (WoS) abstracts (as a proxy for general scientific text). It then uses the resulting model to generate topic distribution for free-text files: the finite mixture over an underlying set of topics derived from the model.

This LDA approach works well on longer text files. It works less well for the many small text files (e.g., READMEs) contained in the CDIAC repository. This most likely occurs because these documents are not long enough for LDA sampling to extract coherent topic models.

### 5.2.4 Images

Scientific repositories often contain large numbers of image files, such as graphs, charts, photographs, and maps. These files often contain useful structured metadata, such as color palette, size, and resolution, that can be extracted easily given knowledge of the file format. Other attributes important for discovery relate to the content represented by the image: for example, the locations in a map or the anatomical region displayed in an x-ray image. To extract those attributes, the first step should identify the type of the image (e.g., map) before applying more sophisticated extraction methods.

First, a specialized **image** extractor classifies image types. This extractor resizes an image
to standard dimensions, converts it to grayscale, and passes the result to a Support Vector
Machine (SVM) model that returns a most likely class: plot, map, photograph, etc. The
SVM model is trained offline with a set of labeled training images.

Having inferred a likely image type, Skluma then applies one or more specialized extrac-
tors: for example plot to extract titles from plots and map to extract locations for maps.

The plot extractor applies the Tesseract optical character recognition (OCR) engine [64]
to identify and extract text from the image. At present these text fragments are returned
without regard for where they appear in the image.

The map extractor attempts to derive meaningful geolocation metadata from images. It
first extracts text present in the image by applying a sequence of morphological filters to the
image, using a contour-finding algorithm to identify contours [12], and filtering contours to
identify text characters, which it then groups by location to form text boxes [66], as shown in
Figure 5.2. Each text box is then passed through Tesseract to extract text. Each valid piece
of text is parsed either as a latitude or longitude coordinate, or as the name of a location.
If at least two pairs of valid coordinates are found, a pixel-to-coordinates map is created for
later use.

The map extractor next applies a different set of morphological filters to the original image
for border extraction. Contours are found by the same algorithm above and are filtered to
isolate those that are large enough to be borders of geographical regions. Note that this
approach assumes that any geographical regions of significance will not be negligibly small
relative to the size of the image. The pixel-to-coordinates map transforms each contour into
a set of (latitude, longitude) pairs bordering a region. Each coordinate-border is searched
within an index [2] of regional shapefiles to find any geographical regions with which it
overlaps (see Figure 5.2). If the border extraction fails, it at least extracts the (latitude,
longitude) span of the map, which is useful in its own right, although less specific. At
this point in the pipeline, latitude and longitude spans of the map are extracted, and each
substantial region found in the map, with the geographical location(s) that it covers, is
Figure 5.2: An example of location metadata extraction from maps. The red boxes delineate the extracted text; the white text is the tags generated for each region found in the map.

tagged.

These contour-finding, text-isolation, and text-extraction techniques can also be extended to other sorts of images. For example, one could use contours to identify and characterize scientific plots within an image (e.g., histograms, pie-charts), and perhaps even to extract the data that they represent.
CHAPTER 6
EVALUATION

6.1 Evaluation

A number of metadata indexing tasks serve as the basis for evaluating Skluma’s accuracy and performance. The repository target is the 500,001 file (303 GB) CDIAC. This repository contains a large collection of minimally curated scientific data including an assortment of images, structured data files, hierarchical files, encoded files, and unstructured files: see Table 6.1. This chapter specifically shows evaluation metrics on the accuracy of the file sampler, Skluma’s overall correctness, and the performance of both cloud and local deployments of Skluma.

6.2 Extractor accuracy on CDIAC data

This section explores the accuracy of the sampler, topic, null, and map extractors.

6.2.1 Accuracy of the sampler extractor

This section explores the accuracy and training time when using two machine learning models (random forests: RandForest, or logistic regression: Logistic) and two sampling strategies (from the head of the file: Head, or half each from the head and random locations in the file: RandHead). The baseline model for analysis uses labels obtained by running the

<table>
<thead>
<tr>
<th>File type</th>
<th>Count</th>
<th>Size (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structured</td>
<td>22,127</td>
<td>11.90</td>
</tr>
<tr>
<td>Unstructured</td>
<td>53,184</td>
<td>13.57</td>
</tr>
<tr>
<td>Tabular</td>
<td>173,019</td>
<td>12.90</td>
</tr>
<tr>
<td>Image</td>
<td>32,127</td>
<td>118.20</td>
</tr>
<tr>
<td>Specialized</td>
<td>156,191</td>
<td>20.14</td>
</tr>
<tr>
<td>Other</td>
<td>63,353</td>
<td>15.32</td>
</tr>
<tr>
<td>Total</td>
<td>500,001</td>
<td>15.41</td>
</tr>
</tbody>
</table>
Table 6.2: Model accuracy and training and reading times, for 20,000 randomly selected CDIAC files.

<table>
<thead>
<tr>
<th>Model</th>
<th>Features</th>
<th>Accuracy (%)</th>
<th>Time (sec)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Train</td>
<td>Read</td>
<td></td>
</tr>
<tr>
<td>RandForest</td>
<td>Head</td>
<td>93.3</td>
<td>1.77</td>
<td>3.69</td>
</tr>
<tr>
<td></td>
<td>RandHead</td>
<td>90.6</td>
<td>3.76</td>
<td>356.40</td>
</tr>
<tr>
<td>Logistic</td>
<td>Head</td>
<td>92.4</td>
<td>597.00</td>
<td>2.97</td>
</tr>
<tr>
<td></td>
<td>Random</td>
<td>83.9</td>
<td>958.00</td>
<td>290.00</td>
</tr>
<tr>
<td>Baseline</td>
<td></td>
<td>71.4</td>
<td>0.00</td>
<td>9,097.00</td>
</tr>
</tbody>
</table>

tabular, structured, free text, and image extractors in sequence until either (1) one of the four successfully extracts metadata, or (2) all four raise exceptions.

First, 20,000 CDIAC files are selected uniformly at random from the repository. Five-fold cross validation randomly shuffles the files into five groups where each group serves as the training set for the other four (each withholding type labels) exactly once. Accuracy is measured as the percentage of predicted extractor-to-file pairings that result in non-empty metadata sets in each testing subset. Table 6.2 reports the accuracy and training time for the feature sets and models. Both models are significantly more accurate than the baseline approach. Random forests performs better than logistic regression, but only slightly. Sampling the head of the file, rather than using the head and a random sample throughout the file, is more accurate for both models, which is fortunate as the latter takes much longer due to the need to scan through the file. Of note, the time spent training this model is negligible (one can train a random forests model using the head of the file in just over three seconds) relative to the cost of running extractors (see average individual extractor times in Table 6.3).

### 6.2.2 Accuracy of the topic extractor

Human reviewers were used to validate extracted topics and keywords. Accuracy is measured as the percentage of files in which the extracted topics and keywords provide correct, useful information. Topics are considered correct if they match the intended topic of the text. Topics are considered useful if they subjectively reflect the content of a file.
There exist three mutually exclusive classes for determining accuracy: (1) correct and useful, (2) incorrect, (3) not useful. Incorrect metadata include incorrect terms: for example, a README about rainforest precipitation labeled as ‘concrete.’ Non-useful metadata are correct but too vague: for example, a PDF about jetstream currents labeled as ‘science.’ Human reviewers were tasked with assigning each file to one of the three classes. They examined 250 files with extracted topic metadata, and deemed those metadata to be correct and useful for 236, not useful for 14, and incorrect for none, for an accuracy of 94%.

6.2.3 Accuracy of the null extractor

The correctness of the null value extractor is measured by comparing the output of columns processed by the extractor with each column’s human-labeled null values. When trained and tested by cross-validation on a labeled test set of 4,682 columns from 335 unique CDIAC files, the model achieved accuracy 99%, precision 99%, and recall 96%, where both precision and recall are calculated by macro-averaging over all classifiers.

6.2.4 Accuracy of the map extractor

250 map images were selected at random from the 583 files classified as maps, and each was given associated metadata from map. Human reviewers classified the metadata into the three classes listed above: correct and useful, incorrect, or not useful. Reviewers found that the metadata were correct and useful for 231, incorrect for 7, and not useful for 12, for an accuracy of 92%. Two reasons for incorrect metadata were uncommon projections and/or fuzzy text. Thus, the map extractor should be extended to account for additional projections and to provide a confidence metric for extracted text.
6.3 Performance

Skluma’s performance is evaluated in four ways: first, by investing how quickly the crawler can perform a depth-first search of a repository; second, by investigating extractor performance with respect to input size and extractor concurrency; third, by analyzing the performance of each extractor as they are used to process the entire CDIAC repository; and finally, by reviewing the entire indexing pipeline.

6.3.1 Crawling

To evaluate the performance of the crawler, this section examines crawling latency on CDIAC stored first in an FTP repository, and again with CDIAC downloaded to local disk. Both crawlers run locally, using an Ubuntu 17.04 desktop machine with an Intel Core i7-3770 CPU (3.40GHz) and 16GB DDR3 RAM. The machine is configured with Docker 1.13.1, and the crawler is encapsulated within a single Docker container.

Crawling the FTP server took 34 hours to complete. The crawler was then reconfigured to run on the local copy. Running the crawler on the same machine as the data took approximately 11 minutes. This means that there is a significant cost for moving the data, amounting to approximately 4 seconds per file, on average.

6.3.2 Extractor Scalability

Exploring the performance of an individual extractor illustrates how the system performs when processing files of varying size using an increasing number of containers executing concurrently. These experiments employ multiple hybrid files, each of which start with a short, 10-line preamble followed by a column-header line, with structured data beneath. These files were determined to be a tabular+unstructured hybrid file by the sampler extractor, and thus this test uses each of the tabular, topic, and null extractors. Each file size class (5MB-5GB) is a truncated version of a larger 5.2GB data to avoid excessive statistical
variance between files.

Figure 6.1 shows the results when running locally and varying the number of concurrent containers. This figure shows that execution on the local deployment scales linearly as long as a free core is available, but does not scale well beyond this point (in this case, as Skluma upgrades from 4 to 8 concurrent containers).

Cloud deployments were hosted on an ECS cluster, using between 1 and 16 m4.xlarge instances. In this case, a live Skluma run was simulated by populating an S3 object store bucket with the dataset, and used a Simple Queue Service queue to provide a global FIFO file processing ordering. The cloud deployment results, shown in Figure 6.1, exhibit linear scalability with respect to number of nodes and file size in terms of both processing time and throughput.

6.3.3 Extractor Performance

Although the results above show Skluma is capable of near-linear scaling when processing hybrid, tabular+unstructured data, the throughput is not necessarily indicative of other extractors. Each Skluma extractor performs a specialized extraction task, where some, such as image extraction, perform much more compute-intensive tasks. Therefore, to forecast Skluma’s performance on a given repository, the performance profile of each extractor is necessary.

The average cloud performance of each Skluma extractor when applied to the entire CDIAC repository, shown in Table 6.3, shows vastly different resource requirements of extractors, ranging from 3.47 ms for null to almost 15s for map.

6.3.4 CDIAC Indexing

This section explores both the wall-time and cloud cost for indexing the entire CDIAC repository on AWS. March 2018 AWS spot price data are used to estimate the costs of running on 1–16-node ECS clusters, accounting for compute time on vanilla m4.xlarge instances in
Figure 6.1: Throughput vs. file size on various container scalings: Local (top) and Cloud (bottom)
Table 6.3: Performance profiles for the crawler, extractors, and data download from S3 to AWS compute, giving for each the number of CDIAC files processed and, for those files, mean file size and execution time. For clarity, the decompression task is separated from its parent extractor universal.

<table>
<thead>
<tr>
<th>Pipeline Step</th>
<th>File count</th>
<th>Mean values</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Pipeline Step</td>
<td>count</td>
<td>Size (KB)</td>
</tr>
<tr>
<td>Crawler (local)</td>
<td>500,001</td>
<td></td>
<td></td>
<td>4,734.00</td>
</tr>
<tr>
<td>Extractors</td>
<td></td>
<td>universal</td>
<td>500,001</td>
<td>4,734.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>compressed only</td>
<td>381</td>
<td>971.50</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sampler</td>
<td>173,019</td>
<td>180.96</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tabular</td>
<td>500,001</td>
<td>4,734.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>null</td>
<td>3,144</td>
<td>49.91</td>
</tr>
<tr>
<td></td>
<td></td>
<td>structured</td>
<td>22,127</td>
<td>28.37</td>
</tr>
<tr>
<td></td>
<td></td>
<td>topic</td>
<td>53,184</td>
<td>2,151.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>image</td>
<td>32,127</td>
<td>117.10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>map</td>
<td>583</td>
<td>201.50</td>
</tr>
<tr>
<td>Network download</td>
<td>500,001</td>
<td></td>
<td></td>
<td>4,734.00</td>
</tr>
</tbody>
</table>

Table 6.4: Time and cost to index all CDIAC files on AWS.

<table>
<thead>
<tr>
<th>Instances</th>
<th>Time (hours)</th>
<th>Cost ($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>70.30</td>
<td>16.61</td>
</tr>
<tr>
<td>2</td>
<td>35.10</td>
<td>15.35</td>
</tr>
<tr>
<td>4</td>
<td>17.60</td>
<td>14.72</td>
</tr>
<tr>
<td>8</td>
<td>8.80</td>
<td>14.40</td>
</tr>
<tr>
<td>16</td>
<td>4.49</td>
<td>14.25</td>
</tr>
</tbody>
</table>

the us-east-1 availability zone; RDS, S3, and SQS costs; and intra-network file transfers. Table 6.4 demonstrates that the entire repository can be processed in less than 5 hours, for just $14.25, when using 16 nodes. One should notice that cost monotonically decreases up to 16 nodes as the number of instances increases. This stems from pay-per-minute AWS services such as the Relational Database Service (RDS). One should expect the cost to increase at approximately 24 nodes under this configuration.
CHAPTER 7
KLIMATIC: INTEGRATION IN GEOSPATIAL DATA LAKES

Prior chapters illustrate how Skluma as a standalone system can create a searchable metadata index for repositories, but it can also serve as an effective tool for adding extraction and search capabilities to existing systems. This chapter illustrates how one can leverage Skluma to create Klimatic – a ‘virtual’ or distributed data lake system for crawling, indexing, integrating, and serving geospatial data to users.

Klimatic indexes large quantities of geospatial data stored in Globus-accessible shared file system endpoints. The virtual lake model allows for the local caching of raw data in a standardized format, making integration and distribution more efficient at query-time. A geospatial data lake should allow for the straightforward alignment of spatial and time-based variables, and be able to manage and integrate heterogeneous data formats. Given the huge quantity of geospatial data, Klimatic extends the data lake model to encompass a Skluma metadata index of all processed data and the use of the virtual lake as a cache for popular raw data. This approach allows for the tracking of less popular datasets without giving up valuable performance and space availability for oft-accessed data.

The remainder of this chapter is as follows. Section 7.1 discusses the domain challenges in constructing a geospatial data lake with integration support. Section 7.2 discusses the movement of the file and its accompanying metadata. Section 7.3 discusses the process of integrating and serving multiple datasets for user consumption.

7.1 Challenges

Klimatic strives to serve data to users such that the data are cleanly integrated (i.e., appropriately combined into the minimum number of files) while adhering to strict data integrity constraints. The remainder of this section outlines the challenges underlying data integration and integrity assurance.
Integration: The purpose of Klimatic is to create a flexible virtual data lake from which users may retrieve not only individual datasets but also integrated datasets defined by a specification such as “all temperature measurements for the region (30W to 32W, 80N to 82N) for the period of January, 2016.” It must process such requests efficiently, while also upholding the data’s integrity. Geospatial data are particularly complicated to integrate as heterogeneous collection methods result in different representations (e.g., raster vs. vector) and different granularities (e.g., spatial and temporal). Furthermore, the units used to represent common data, as stored in Skluma’s metadata index, may be different (or even missing). Thus, Klimatic must effectively manage misalignments between datasets to curate a new dataset with near-equal integrity to its ancestors.

Ensuring integrity: Given the integrative nature of Klimatic, a number of geospatial integrity rules must be followed when integrating multiple geo-spatial datasets into one. These constraints include topological, semantic, and user-defined integrity constraints [29, 14, 22]. Topological constraints require that data be divided into mutually exclusive regions with all space covered. Semantic constraints require that geological relationships are maintained, meaning, for example, that a road cannot exist in the same space as a building. Finally, user-defined constraints require that data are minimally affected following post-processing. Additionally, integrated data should include information that tracks data lineage. If a dataset cannot fit these constraints, the user is asked whether to reject the integrated dataset.

7.2 Extracting metadata and indexing

The process begins by pointing Skluma’s crawlers at a repository of geospatial data. All metadata are loaded as JSON objects into a standard PostgreSQL database and indexed via a PostgreSQL text-search (TS) vector, an alternative to checksums that creates a unique string out of a dataset’s metadata. The TS vector index also makes it easy for the integration module to check for the availability of certain data parameters, such as lat, long, variables, start date, end date, and the dataset’s publisher.
7.2.1 Data Storage

If a new dataset is not determined to be a duplicate, the Klimatic system next converts its contents to a relational format and loads them into a new PostgreSQL table, so as to accelerate subsequent retrieval and integration operations. The data are not otherwise modified, although future work could involve automatic transformation to reference grids, perhaps based on analysis of user query histories.

Given the virtually unlimited number of geospatial datasets, it is infeasible to retain the contents of every dataset. Thus, Klimatic operates a caching strategy. Metadata for every dataset located via crawling are stored in the index, but dataset contents are stored only if smaller than a predefined threshold and are subject to ejection via an LRU policy when the cache is full. Thus, larger and less popular datasets may need to be re-fetched when requested by a user. (One could also explore alternatives to discarding datasets, such as compression and transfer to slower, cheaper storage.)

7.3 Integration

Having loaded some number of datasets into the virtual data lake, Klimatic must next respond to queries. The query model is shown in Figure 7.2. The initial query interface is a simple web GUI using Flask and Python. With the goal of making the query interface as simple as possible, users may query using minimum and maximum latitudes and longitudes (i.e., a bounding box for their data); the variable(s) they would like included in their dataset; the begin and end dates; and (optionally) the data provider(s) from which data is wanted. Klimatic then estimates the amount of time required to conduct the join and deliver the dataset. Many queries require more than two minutes for the join, as many datasets in Globus shared endpoints have upward of 2 million cells.

The multiple possible encodings for climate data, most notably vector and raster, creates challenges when attempting to integrate multiple datasets into one. A vector is a data
Figure 7.1: $F_1$ on Matrix M to format a vector as a raster. Black values are original, red are created on first sweep, and orange created on second.

structure that represents many observations from a single point, but at different times (e.g., precipitation levels measured at a fixed weather station). A raster can be represented by a two dimensional grid, in which each cell is a certain area identifiable on a map. Each cell contains the value of some variable: for example, the percentage of pollen in the air. Thus, to enable users to retrieve integrated datasets Klimatic requires a method for integrating these two formats for cross-format data analysis: an integration that may involve a sparse set of vectors and a large raster database. (For example, $\sim$180,000 weather stations record precipitation in the U.S., each with a fixed latitude and longitude, while a complete radar mapping of the U.S. results in over 760,000 5 km$^2$ raster cells [3].)

Integration occurs via an interpolation from point values to a scalar field (a raster). The method presents itself as a series of sweeping focal operations for some raster $M$, where each point in $M$ represents a cell of a given region denoted by latitudinal and longitudinal boundaries. A focal operation is defined as the operation on a certain cell with regards to a small neighborhood around the cell [59]. The implementation of this algorithm begins with a focal neighborhood of 1, or the eight diagonal or adjacent cells of a selected empty cell. If there are at least two neighbors, the new cell becomes the non-weighted average of all cells in region $F_1$. The center of $F_1$ is moved from cell-to-cell until either all cells are full or there exist $F_1$s such that there are not at least two value-bearing cells inside. The algorithm

\[
M_1 = \begin{bmatrix}
. & . & . & 4 & . \\
. & 4 & . & . & 4 \\
6 & 2 & . & 8 \\
\end{bmatrix}
\]

\[
M_2 = \begin{bmatrix}
. & . & . & 4 & 4 \\
5 & 4 & 3.3 & 4.2 & 4 \\
6 & 4.1 & 2 & 4.3 & 8 \\
\end{bmatrix}
\]

\[
M_3 = \begin{bmatrix}
5 & 4.3 & 3.9 & 4 & 4 \\
5 & 4 & 3.3 & 4.2 & 4 \\
6 & 4.1 & 2 & 4.3 & 8 \\
\end{bmatrix}
\]
Figure 7.2: Work flow for Klimatic’s data integration and distribution.

then adds one more series of neighbors (i.e., neighbors of neighbors), which are called $F_2$, $F_3$ through $F_n$, where $F_n$ results in a complete matrix.

Figure 7.2 illustrates this process, where $M_1$ is the original sparse matrix and $M_2$ and $M_3$ are the second and third sweeps. As far as the data’s user-defined, post-processing integrity is concerned, Klimatic’s output header records the number of sweeps necessary to make the vector compatible with rasters. One may infer that a higher number of sweeps results in less ‘pure’ data. Klimatic’s interface will also prompt users with information regarding the data’s post-processing integrity as well as related data that could be selected to increase this integrity.

Klimatic currently supports the creation of integrated NetCDF and CSV files. NetCDF conventions simplify the creation of an integrated NetCDF dataset. NetCDF files can be conceptualized as having containers for multiple variables, while assuming that matching indices across the containers refers to a specific data point; index 0 in each container refers to the first data point, index 1 the second, and so on.

If a query response requires integration of both vector and raster data, Klimatic currently uses the grid dictated by the raster. Each vector always lies within a raster cell, so each cell containing one vector becomes the value of the vector at a given time. If multiple vectors fall within the same raster cell, the values are currently averaged. (Here and elsewhere, just one data conversion strategy is applied automatically in the prototype. Ultimately, one would want to allow the user to control such actions.) Once a standardized grid is achieved, the
addition of a variable only requires the addition of another variable container, as long as the spatial and temporal bounds align. If the resolutions and time-bounds are different (e.g., if one dataset is measured in months and the other in years), the cells are aggregated to the larger period (i.e., years). Future work could involve imputing values for missing areas and time periods, but this will require statistical distribution analysis.

7.4 Klimatic Summary

Klimatic effectively provides an accessible architecture for the collection and dissemination of large, distributed geospatial data. It is able to automatically crawl huge amounts of data distributed across various storage systems accessible via HTTP and Globus, call Skluma to extract metadata, integrate data sets, selectively transform and store popular data, and serve files to users.
CHAPTER 8
SUMMARY AND FUTURE WORK

This thesis explored the dynamic workflow system for metadata extraction processes called Skluma. Skluma uses both general and type-specific extractors, and with reasonable accuracy and performance, has orchestrated these extractors to index the large, disorganized CDIAC carbon dioxide data set. This thesis first formalized the metadata extraction problem, and then explored a number of trade-offs between accuracy, cost, and time in these metadata extraction processes. Moreover, Skluma can be used in the design of virtual data lake systems by providing a metadata index for files stored in disparate remote stores. Skluma should prove to serve as a solid starting point for deeper study of metadata utility.

A clear future work direction is the formalization and improvement of metadata utility derived from systems like Skluma. Specifically, one should explore a number of methods for quantifying trade-offs, namely those between extraction cost and (1) metadata utility and (2) required human effort. In regards to the latter case, one might explore the development of chatbot systems to confirm (and possibly supplement) machine-extracted metadata. Further, one could leverage active learning techniques to rank metadata documents such that the chatbot assigns a partial subset of files to humans, subject to the dual constraints of metadata utility maximization and human effort minimization.

Additionally, Skluma should leverage in its workflows the probabilistic inference of metadata yield from extractors. Not only would associating probabilities with extractors lead to efficient extractor-selection for the orchestrator, but would allow for more fine-grained parallelization of extractors on a given file. Moreover, as metrics are defined for formalizing metadata utility and cost, one could rank and only use the best extractors for bolstering utility, even if other extractors can produce non-empty metadata.

Next, one should consider ‘bigger’ big data than CDIAC. At the time of this writing, there exist many peta-scale stores such as Argonne’s Petrel [4] or the sum of Globus’ tens of thousands of connected endpoints. Exploring data at this scale would likely raise a number
of challenges unseen with Skluma on CDIAC, and could potentially require the adoption of HPC resources for Skluma’s extraction processes.

Finally, one can explore the creation of a unified language for developing extractors. This would ensure, while keeping the system extensible for scientists developing domain-specific extractors, that metadata are extracted with an emphasis on standardization, extraction efficiency, and query (or search) optimization.
REFERENCES


