Computers & Graphics 37 (2013) 484–495

Contents lists available at SciVerse ScienceDirect

Computers & Graphics

journal homepage: www.elsevier.com/locate/cag

Special Section on 3D Object Retrieval

A flexible and extensible approach to automated CAD/CAM format classification ☆, ☆☆

Vincent A. Cicirello [a,*], William C. Regli [b]

[a] Computer Science and Information Systems, School of Business, 101 Vera King Farris Drive, Richard Stockton College, Galloway, NJ 08205, United States
[b] College of Information Science and Technology, Drexel University, Philadelphia, PA 19104, United States

ARTICLE INFO

Article history:
Received 25 October 2012
Received in revised form
31 January 2013
Accepted 7 March 2013
Available online 1 May 2013

Keywords:
CAD data management
Compression-based distance
Compression-based similarity
Digital preservation
Engineering digital libraries
Format classification
Nearest-neighbor classifier

ABSTRACT

There are hundreds of distinct 3D, CAD and engineering file formats. As engineering design and analysis has become increasingly digital, the proliferation of file formats has created many problems for data preservation, data exchange, and interoperability. In some situations, physical file objects exist on legacy media and must be identified and interpreted for reuse. In other cases, file objects may have varying representational expressiveness.

We introduce the problem of automated file recognition and classification in emerging digital engineering environments, where all design, manufacturing and production activities are "born digital." The result is that massive quantities and varieties of data objects are created during the product lifecycle.

This paper presents an approach to automated identification of engineering file formats. This work operates independent of any modeling tools and can identify families of related file objects as well as variations in versions. This problem is challenging as it cannot assume any a priori knowledge about the nature of the physical file object. Applications for these methods include support for a number of emerging applications in areas such as forensic analysis, data translation, as well as digital curation and long-term data management.

1. Introduction

There are hundreds of distinct 3D, CAD and engineering file formats. As engineering design and analysis has become increasingly digital, the proliferation of file formats has created many problems for data preservation, data exchange, and interoperability [1–3]. In some situations, physical file objects exist on legacy media and must be identified and interpreted for reuse [1]. In other cases, file objects may have varying representational expressiveness.

We introduce the problem of automated file recognition and classification in emerging digital engineering environments. In emerging digital engineering environments, all design, manufacturing and production activities are "born digital." The result is that massive quantities and varieties of data objects are created during the product lifecycle [1]. For historic, paper-based processes, the activity of record keeping and management would be the task for a team of human engineers and archivists. In the new digital processes, the overwhelming amount of information requires the development of tools to enable assimilation of massively more diverse and complex data.

Managing these new processes and workflows requires automated tools to make the task of the engineer, archivist, or records manager tractable. Consider the scenario common in the aerospace industry: compliance. Records for aircraft must be retained for decades past the end of the production run [4]. Hence, for new aircraft (such as the 787 or A380), one can reasonably expect airplanes designed (and even manufactured!) in the past decade to be flying at the turn of the next century. Records for these artifacts are predominantly digital, with CAD models, configuration models, surrogate models, simulation objects, etc, proliferating and even unique to each and every aircraft. Making sense of this typhoon of data, organizing and harvesting it for its long-term use, requires new kinds of automated tools for sifting CAD data.

This paper presents an approach to automated identification of engineering file formats. This work operates independent of any modeling tools and can identify families of related file objects as well as variations in versions. This problem is challenging as it cannot assume any a priori knowledge about the nature of the physical file object. Hence, the file object could be data from a legacy system, or a one-off format created for some ad hoc analysis. In these scenarios, one cannot assume there is even

☆The associated collage experiments are released via doi: 10.0000/1359042156382. The same code can be used to recreate the full-scale experiments using the full Drexel Repository or with other sets of testcases of the readers' choosing.

☆☆To comment on this article, please join the discussion on the Collage Authoring Environment Google Group https://groups.google.com/group/collage-authoring-environment.

* Corresponding author. Tel.: +1 609 626 3526.
E-mail addresses: cicirelv@stockton.edu (V.A. Cicirello), regli@drexel.edu (W.C. Regli).

0097-8493
http://dx.doi.org/10.1016/j.cag.2013.03.007

a software system capable of reading the object and rendering the digital model into a form that can be operated on. These are just some of the daunting requirements to support the emerging applications in areas of digital curation and long-term data management for engineering.

This problem may appear, on its surface, to be trival. Why cannot we identify files based on file extension or other means? What makes this problem unique, and difficult, is the nature of use case for those engaged in curation of digital objects. For classes of engineering objects such as airframes or ships, the expected artifact lifecycle is measured in decades [4]. It is often the case today that engineering data objects on tape, or pre-digital media, are critical to understanding the performance and maintenance of existing structure. In this setting, those who must manage and curate engineering data are constantly faced with files of unknown origin or provenance, for which the task of understanding their contents is more of matter of forensics rather than simply reading the file into a browser or CAD system. Further complex is the case of contemporary processes today where file formats proliferate due to the analyses, measurements and simulations done on a particular artifact. In case studies by the authors, the number of files associated with the analysis of an *individual* part can number in the thousands [1]. Managing this overwhelming mass of new digital objects requires new tools for understanding the content and context of physical file objects. We believe this paper offers an initial tool for those working on engineering data.

This paper is organized as follows. We begin, in Section 2, with a review of related work. Next, we detail our technical approach in Section 3 which is motivated by the information theory concept of Kolmogorov complexity and is an application of compression-based classification. In Section 4, we present comprehensive experiments using the Drexel University Design Repository. The Drexel Design Repository originated at NIST in 1993 and contains tens of thousands of CAD, CAM, and other related engineering data in hundreds of formats [5–7]. The Drexel Repository spans two decades and includes formats that date back to the 1980s (e.g., from the early CAD system Romulus) and multiple versions of formats (e.g., models from ACIS 1.1, multiple versions of Pro/E dating back to the early 1990s). Our experiments specifically use the 54 most frequently occurring filetypes from the Drexel Design Repository which amounts to over 40,000 files in our dataset. The formats primarily include 3D CAD model formats, CAM system formats, 3D simulation formats, formats used by 3D object similarity algorithms, various non-model formats associated with CAD/CAM systems, as well as 2D graphics formats, among others. We conclude in Section 5.

2. Related work

2.1. Digital object preservation and curation

Perhaps the most prominent tool related to digital preservation is the JSTOR/Harvard Object Validation Environment (JHOVE) [8,9]. JHOVE supports automated format identification, as well as format validation. JHOVE is extensible in that it enables easy integration of additional format identifiers and validators. JHOVE currently supports a variety of image, audio, and text formats, as well as pdf through what the JHOVE developers call modules. Identification of a format in JHOVE relies on a parser for the format. To extend JHOVE for a new format, one implements a corresponding JHOVE module, which involves implementing a parser for the format, among other things. The advantages of JHOVE are that, in addition to identification, JHOVE is capable of extracting various meta-data from the file since JHOVE is aware of the format's details. JHOVE can also validate a file for conformance to the format's specification. JHOVE has been extended to support preservation of newspaper data as part of the National Digital Newspaper Program [10].

Recently, development of a successor of JHOVE has begun, known as JHOVE2 [11]. The developers of JHOVE2 have significantly redesigned JHOVE to achieve improved performance, among other things. However, format identification still requires file format knowledge. In JHOVE2, identification has been decoupled from validation, unlike JHOVE where identification relied on the validators for the known types. JHOVE2 utilizes the PRONOM signature model of the DROID format identification algorithm [12]. PRONOM includes what its authors refer to as “external” and “internal” signatures. A so-called “external” signature is something external to the object's data file, such as a file extension; whereas an “internal” signature is a pattern sequence within the data file that identifies its type. Thus, this approach still relies on a priori known knowledge of the format—although it is simpler knowledge.

However, the source of JHOVE and JHOVE2's power is also the source of their limitations. If one is dealing with legacy data or data stored in a proprietary format whose specification was never made public, then you will not have access to a parser nor the specification needed to implement one; and no known internal signatures may be available. If you could only identify the format, then perhaps you could identify a legacy CAD system capable of interpreting the data. Therefore, in our work, we focus on developing a format identification approach that does not rely on a priori format specifications including format signatures. Instead, we develop an approach capable of learning by example to identify a file's format. Essentially, our approach is to automate the discovery of internal signatures.

Specifically within the realm of 3D object formats, the objectives of the NCSA Polyglot project are to develop a universal file format converter capable of converting among the myriad of 3D object formats [13]. In contrast to the work of the JHOVE, JHOVE2, and related projects, the developers of Polyglot do not rely on well-documented format specifications for identification and conversion. Instead, they use a scripting language to create wrappers for existing CAD systems relying on the native CAD systems' ability to recognize their own formats and to export to certain other formats.

2.2. Compression-based similarity detection

There are several existing approaches in the literature on the application of compression algorithms in the detection of similarities across objects, strings, files, etc [14–19]. These approaches vary in terms of underlying theory and motivation; and span an interesting array of applications including classifying protein sequences [18], file clustering [17], classifying genomes [14], classifying textures [20], among others [16,15]. Our technical approach to the problem of automated format classification is a nearest neighbors classifier, and as such, requires a measure of distance in order to determine the nearest neighbors of the query example we are classifying.

One such distance measure is known as Normalized Compression Distance (NCD) and is motivated by conditional Kolmogorov complexity [14]. Conditional Kolmogorov complexity of a pair of strings (likewise, files, objects, etc) is the length of the shortest program capable of generating one of the files if the other is given as input. It is a noncomputable function, however, Li et al. [14] showed that one can define a reasonable approximation relying on the compressed length of the concatenation of the pair of objects and the compressed lengths of the individual objects. NCD has previously been applied to automated language tree computation and to inferring the evolutionary history of genomes [14]. NCD has

also been applied more generally within clustering problems with a diverse set of applications from the sciences, character recognition, art, music, and some others [15]. Another distance measure that is closely related to NCD is the Compression-Based Dissimilarity Measure (CDM) [16]. CDM has been applied to classification and clustering problems using DNA data, text-based data, and video data. In their work on texture classification, Campana and Keogh [20] apply video compression algorithms in defining their CK-1 distance measure. More recently, the Length Delimiting Dictionary Distance (LD^3) of Li et al. [19] improves upon the computational cost of NCD by not actually performing the compression.

We discuss both NCD and CDM in more detail later in Section 3.1 as we specifically apply these within our experimental work. In our application, we restrict the use of these distance measures to a portion of the objects under comparison, rather than to the entire objects. This both provides a more efficient implementation as distance computation is dependent upon the length of the objects under comparison, as well as a more accurate automated identification of the key underlying commonalities between the pair of objects that make them similar. The latter is due to a property of most practical compression algorithm implementations which are able only to identify compressible patterns within a fixed window width.

The approach of NCD and CDM, which rely on using compression as an approximation of Kolmogorov complexity is not the only compression-based approach to similarity detection and classification. Another general type of approach relies on the compressibility of the object. Compressibility refers to the ratio of the compressed size of the object and its uncompressed size. It can be used as an approximation of information entropy. One simple approach in this category that lacks robustness relies on the compressibility of the entire object, comparing this to the compressibility of representative examples of the possible object types. This approach suffers from the inability to detect variations in the compressibility of different parts of the object. In their attempt to circumvent this problem, Hall and Davis [17] developed an approach based on what they term "sliding windows." The sliding window approach computes the compressibility of a substring of the file's contents with a fixed window size. It repeats this process sliding the window down one byte at a time to produce a profile of the file's compressibility throughout its length. This profile is then compared to profiles corresponding to the different available file classes. There are disadvantages to this approach. First, there is a trade-off between computational cost and quality of the compressibility profiles. Too small a window size leads to higher cost (since more windows need to be compressed) but better coverage of the file. Second, and most importantly, compressibility alone cannot capture common patterns between files. An important identifying element of the format can coincidentally be similarly compressible to a completely unrelated byte sequence.

3. Technical approach

In our technical approach to the problem, we consider two variations of nearest neighbors classifiers: (a) *k*-nearest neighbors; and (b) distance weighted *k*-nearest neighbors. In a nearest neighbors classifier, the class of a given query object is predicted based on its "distance" to a set of example objects whose classes are a priori known [21, Chapter 8].

In *k*-nearest neighbors, the *k* examples closest to the query object are used to predict its class via a majority vote. In the distance weighted variation, the votes of examples are weighted based on their distance to the query—the nearer to the query object, the more weight is applied to that example's vote.

We now proceed to provide the details of our nearest neighbors classifier. Section 3.1 summarizes two existing alternative distance measures, both motivated by Kolmogorov complexity [22] that rely on compression as an approximation of Kolmogorov complexity. Section 3.2 presents our distance measure framework which provides a practical algorithmic approach to applying the distance measures from Section 3.1. Section 3.3 presents further details of the classifiers, including the approach to weighting. And Section 3.4 explains our procedure for selecting a value for *k*, the number of neighbors.

3.1. Distance measures

In our approach, we apply two existing distance measures that are both motivated by the concept of Kolmogorov complexity. The Kolomogorov complexity of a string x is the length of the shortest program that is capable of generating x on a universal Turing machine [22]. Kolomogorov complexity, usually denoted by $K(x)$, is noncomputable in the general case. The conditional Kolmogorov complexity $K(x|y)$ is the length of the shortest program capable of generating x if y is given as an additional input. It is likewise noncomputable. Li et al. [14] also define $K(x,y)$ as the length of the shortest program capable of generating both x and y and capable of telling the difference; and $K(xy)$ as the Kolomogorov complexity of the concatenation of x and y.

3.1.1. Normalized compression distance (NCD)

The first of the two distance measures that we consider is known as Normalized Compression Distance (NCD) [14]. In the work of Li et al. [14], a distance metric was first developed known as Normalized Information Distance (NID), however, NID is noncomputable as it requires computing Kolomogorov complexity. Li et al. [14] then presented a more practical measure of distance, the NCD, which relies on using a compression algorithm to approximate Kolomogorov complexity. Due to the use of approximations, though NID is a metric, NCD is not guaranteed to satisfy all of the requirements of a metric. It is, however, a practical approximation. NCD is defined as

$$\mathrm{NCD}(x,y) = \frac{C(xy) - \min(C(x), C(y))}{\max(C(x), C(y))}, \tag{1}$$

where $C(.)$ is any real world compressor, and $C(x)$, $C(xy)$ refers to the compressed lengths of x and the concatenation of x and y, respectively. Li et al. [14] showed that $K(x|y) = K(x,y) - K(y)$ to an additive constant and $K(x,y) = K(xy)$ to an additive logarithmic precision. Thus, the numerator in Eq. (1) is an approximation of conditional Kolomogorov complexity. A compressor, $C(.)$, can be used to compute an upper bound on $K(.)$. The better the compressor, the tighter the upper bound on $K(.)$ it provides. If $C(.)$ was a prefect compressor (i.e., $C(.) = K(.)$), then $0 \le \mathrm{NCD}(x,y) \le 1$ for all x and y. In their experiments, Li et al. [14] only occasionally found NCD values greater than 1. The lower the value of $\mathrm{NCD}(x,y)$, the more similar are x and y. Elsewhere, Cilibrasi and Vitányi [15] presented the requirements of a compressor needed to ensure that NCD is a metric, and explained how real-world compressors typically meet these requirements. They argue that the one property that stream-based compressors possibly lack is symmetry (i.e., that $C(xy) = C(yx)$) due to the possibility of the compressor adapting to regularities in x prior to transitioning into the compression of y. We later modify NCD to retain its status as a metric even if the compressor used is not strictly symmetric (see Eq. (3)).

3.1.2. Compression-based dissimilarity measure (CDM)

A second distance measure that we consider is the Compression-Based Dissimilarity Measure (CDM) [16]. The objectives of Keogh

et al. [16] in their development of CDM was to produce a practical approach to measuring the similarity of strings, so although CDM uses a compressor $C(.)$ as an approximation of Kolmogorov complexity, their approach is less strongly tied to the theory of Kolmogorov complexity as compared to that of Li et al. [14]. CDM is defined as

$$\mathrm{CDM}(x,y)=\frac{C(xy)}{C(x)+C(y)}. \quad (2)$$

Just like NCD, the lower the value of $\mathrm{CDM}(x,y)$, the more similar are x and y. However, unlike NCD, the minimum value of CDM is not 0, but rather 0.5. Specifically, we have $0.5 \le \mathrm{CDM}(x,y) \le 1$ for all x and y.

3.2. Distance measure framework

Unlike other applications of the NCD and CDM distance measures, we do not apply the distance measure directly to the entire files. We have also made additional modifications to improve the quality of the approximation of Kolomogorov complexity. As motivation for our distance measure framework, we begin with some background information on the compression algorithm that we employ.

In their work with NCD, Li et al. [14] performed experiments with a variety of different compression algorithms, and demonstrated that the approach works under a variety of compressors. Likewise, Keogh et al. [16] considered multiple compressors in their work. However, their approach differed in that instead of using a single compressor, Keogh et al. [16] compute $C(xy)$, $C(x)$, and $C(y)$ with multiple compressors and use whichever provides the best compression for the given data. The motivation is that, since $C(.)$ is an upper bound for $K(.)$, better compression means a tighter bound on the true value of $K(.)$.

We take a similar approach, though rather than using multiple compression algorithms, we use a single compressor with different compression parameters. Specifically, our approach uses the Deflate Compression Algorithm [23]. Our implementation uses Java's implementation of Deflate via Java's Deflater class [24]. This implementation of Deflate provides 3 alternative compression strategies: (1) the default which is a combination of dictionary lookup and Huffman compression; (2) "filtered" which provides less dictionary lookup as compared to the default with more reliance on Huffman compression; and (3) "Huffman Only" which applies Huffman compression without dictionary lookup. In computing $C(.)$, we compress the data with each of the 3 Deflate strategies. We use the strategy that provides the best level of compression for each specific computation of $C(.)$ (see last 4 lines of Algorithms 1 and 2).

Algorithm 1. Computes $C(xy)$ using either first or last 16 KB of each file, and using the "best" compression strategy (from the 3 strategies available in Deflate).

Input: file1, file2, filePart ▷ filePart ∈{Headers, Trailers}
Output: $C(xy)$
  $N_1 = \min(16\ \mathrm{KB}, \mathrm{sizeOf}(\mathrm{file1}))$ ▷ in bytes
  $N_2 = \min(16\ \mathrm{KB}, \mathrm{sizeOf}(\mathrm{file2}))$ in bytes
  if filePart = Headers **then**
    $x \Leftarrow$ {First N_1 bytes of file1}
    $y \Leftarrow$ {First N_2 bytes of file2}
  else ▷ must be trailers if if's not headers
    $x \Leftarrow$ {Last N_1 bytes of file1}
    $y \Leftarrow$ {Last N_2 bytes of file2}
  end if
  $C(xy) \Leftarrow \infty$
  $C(xy) \Leftarrow \min(C(xy), \mathrm{Deflate}(xy, \mathrm{default}))$
  $C(xy) \Leftarrow \min(C(xy), \mathrm{Deflate}(xy, \mathrm{filtered}))$
  $C(xy) \Leftarrow \min(C(xy), \mathrm{Deflate}(xy, \mathrm{huffman}))$

Algorithm 2. Computes $C(x)$ using either first or last 16 KB of file, and using the "best" compression strategy (from the 3 strategies available in Deflate).

Input: file, filePart ▷ *filePart*∈{Headers, Trailers}
Output: $C(x)$
  $N = \min(16\ \mathrm{KB}, \mathrm{sizeOf}(\mathrm{file}))$ ▷ in bytes
  if filePart = Headers **then**
    $x \Leftarrow$ {First N bytes of file}
  else ▷ must be trailers if if's not headers
    $x \Leftarrow$ {Last N bytes of file}
  end if
  $C(x) \Leftarrow \infty$
  $C(x) \Leftarrow \min(C(x), \mathrm{Deflate}(x, \mathrm{default}))$
  $C(x) \Leftarrow \min(C(x), \mathrm{Deflate}(x, \mathrm{filtered}))$
  $C(x) \Leftarrow \min(C(x), \mathrm{Deflate}(x, \mathrm{huffman}))$

Additionally, we make a second modification in our application of NCD and CDM. The Deflate Compression algorithm, as well as all common compression algorithms, rely on a "window size". The window size indicates how far back the algorithm will look for patterns that match the next part of the data to be compressed. Since $C(xy)$, the compression of the concatenation of the files, is being used to estimate $K(x|y)$, the window size has a significant effect on the efficacy of using a compressor as an estimate of conditional Kolmogorov complexities. $K(x|y)$ is the length of the shortest program capable of generating x if we are given y as an additional input. We can also look at that as how well we can compress x if the dictionary from having compressed y is available to us. If the concatenation of x and y is longer than the window size of the compressor, then $C(xy)$ is not really capturing what is needed to effectively estimate $K(x|y)$, since by the time the compressor is looking at y, portions of x (or possibly all of x depending upon the file sizes) are outside the window. The Deflate algorithm, limits the window size to no higher than 32 KB. If we are to ensure that $C(xy)$ will lead to a reasonable estimate of $K(x|y)$, then we can use no more than 16 KB of each of x and y in the event that they are larger. We consider two alternatives:

- Headers: Using first 16 KB of each of x and y.
- Trailers: Using last 16 KB of each of x and y.

Algorithm 1 formalizes our approach to computing $C(xy)$ using no more than 16 KB of each file, and using the "best" compression strategy (from the 3 strategies available in Deflate). Algorithm 2 shows its counterpart for computing $C(x)$ (and likewise $C(y)$).

One final modification is that in our versions of NCD and CDM, we replace $C(xy)$ with $\min(C(xy), C(yx))$, resulting in

$$\mathrm{NCD}'(x,y)=\frac{\min(C(xy),C(yx))-\min(C(x),C(y))}{\max(C(x),C(y))}, \quad (3)$$

and

$$\mathrm{CDM}'(x,y)=\frac{\min(C(xy),C(yx))}{C(x)+C(y)}. \quad (4)$$

The use of $\min(C(xy), C(yx))$ instead of $C(xy)$ ensures that if our compressor $C(.)$ lacks symmetry that $\mathrm{NCD}'(x,y)$ is still a metric.

3.3. Nearest neighbor classifiers

We consider two types of Nearest Neighbors Classifier: (a) k-nearest neighbors; and (b) distance weighted k-nearest neighbors.

3.3.1. k-nearest neighbors

Our approach to k-Nearest Neighbors is the basic approach. Our classifier requires that it is provided with a set of example files

with known filetypes. When presented with a classification task, our classifier computes the distance between the query file and each of the example files. The k files that are closest to the query are then used to predict the type of the query via a majority vote. To compute the distance between the query file of unknown type and an example of known type, our classifier uses one of our versions of NCD or CDM (Eqs. (3) and (4), respectively) and our approach of limiting its computation to either the beginning or end of the files, as described by Algorithms 1 and 2.

3.3.2. Distance weighted k-nearest neighbors

Our distance weighted version, uses the k closest example files to predict the query's filetype. But instead of a simple majority vote, the vote of each example e is weighted by the inverse of its distance to the query file q according to either $1/\mathrm{NCD}'(e,q)$ or $1/\mathrm{CDM}'(e,q)$. In this way, examples that are closer to the query have a stronger influence on the predicted filetype than those that are further away.

3.4. Selecting K

To determine an appropriate value for k, we use leave-one-out cross validation. Specifically, we begin with a set of example files, T, of known type. T is the set of files that our classifier will use to predict the filetype of queries. For each value of $k = 1 \ldots |T|-1$, and for each training example $t \in T$, we predict the type of t using the other training examples in the set $T-t$. Let f_i^k be the predicted type of the i-th training example $t_i \in T$ for a given value of k; and let f_i^* be the true type of training example t_i. Our training procedure chooses the value of k that maximizes the number of correctly predicted filetypes, such that:

$$k^* = \underset{k=1}{\overset{|T|-1}{\operatorname{argmax}}} \sum_{i=1}^{|T|} c_i^k, \tag{5}$$

where c_i^k is:

$$c_i^k = \begin{cases} 1 & \text{if } f_i^k = f_i^* \\ 0 & \text{otherwise} \end{cases}. \tag{6}$$

In the event that more than one value of k maximizes Eq. (5), then the lowest value of k that maximizes Eq. (5) is chosen.

4. Experiments

4.1. The dataset: Drexel's design repository

We have empirically validated our approach using Drexel University's Design Repository [5]. Initially developed at National Institutes of Standards and Technology (NIST) in 1993 with continued development at Drexel University, the Design Repository contains several tens of thousands of CAD, CAM, and related files in over a hundred formats [6,7]. The vast majority of data in the design repository are 3D models from several different CAD systems. The design repository also contains other related CAD/CAM data such as related to the manufacturing of objects, as well as data related to various 3D object comparison algorithms developed over the years by the Drexel group (e.g., graph formats of an object's features), as well as 2D image files for the 3D objects in a variety of formats. The complete repository includes 19 GB of CAD/CAM data with an average file size of 432 KB. Some formats date back to the early 1990s (e.g., ACIS version 1.1 and early versions of Pro/E) or even earlier (e.g., files from the early CAD system Romulus from the 1980s). In our experiments, we use the 54 most common file formats from the Drexel Design Repository (40,277 files). These 54 filetypes are all of the types for which there are at least 50 examples in the repository, which include the following:

- CAD Model Formats (Parts/Assemblies/Etc)
  - 3D-Design-Format-For-Data-Transfer-STL
  - ACIS-Solid-Modeler-format
  - Autodesk-AutoCAD-Native-Format
  - Autodesk-Drawing-Exchange-Format
  - Bentley-Systems-Microstation
  - Initial-Graphics-Exchange-Specification
  - Parasolid-Text
  - ProE-Assembly-Format
  - ProE-Drawing-File
  - ProE-Neutral-file
  - ProE-Part-Format
  - SDRC-I-DEAS-Model-file-1
  - SDRC-I-DEAS-Model-file-2[1]
  - Simple-Model-Format
  - STEP-Application
- CAM System Related Formats
  - ProE-Cutter-Location-Data-File
  - ProE-Manufacturing-File
  - ProE-Mechanical-Database
  - ProE-Tool-Parameters-File
  - ProE-Tool-Path-File
  - VWS-Vertical-Workstation-Part-Design-file [25]
- 3D Simulation Related Formats
  - CMU-IAMS-Facet-File [26]
  - CMU-IAMS-Part-file [26]
- Non-model files of CAD/CAM and other 3D systems
  - ACIS-Test-Harness-Monitor-file
  - ACIS-Test-Harness-Script
  - GeMS-Generic-Memory-Structures (Honeywell)
  - GeMS-to-ACIS-Map-file
  - ProE-Configuration-file
  - ProE-Format-File
  - SDRC-I-DEAS-Archive
  - SDRC-I-DEAS-Information-Processing-Report
  - SDRC-I-DEAS-Program-file
- Other 3D Data Formats
  - VRML
- File types used by 3D Search Algorithms
  - The following formats are those used by various 3D CAD Search algorithms of the Geometric and Intelligent Computing Laboratory (GICL) [27–31]. This set of formats pose an interesting challenge to the classifiers in that all 4 are either LEDA graphs [32] or a variation of a LEDA graph, but with minor differences in node and edge attributes.
    - LEDA-Graph-GICL-1: A LEDA graph.
    - LEDA-Graph-GICL-2: LEDA-Graph-GICL-1 with additional node and edge attributes.
    - LEDA-Graph-GICL-3: A non-standard LEDA graph, that consists of LEDA-Graph-GICL-2 with extra parameters at end of file.
    - LEDA-Graph-GICL-4: A LEDA graph, unrelated to the other GICL LEDA graph types.
- 2D Graphics Formats
  - Graphics-Interchange-Format
  - JPEG-File
  - X-Bitmap

[1] A model in this format is stored in a pair of files, usually denoted by mf1 and mf2.

- Languages
  - C-Object-File
  - C-Source-Code
  - Encapsulated-Postscript
  - Header-File-C
  - Java-Class-File
  - Java-Source-Code
  - Postscript
- Archive & Compression Formats
  - GZIP-Compressed
  - Unix-Compressed-Archive
  - ZIP-Compressed
- Document Formats
  - Adobe-Portable-Document-Format
- Other Text-Based Formats
  - HTML-File
  - Other Plain-Text
  - Various-Log-Formats in Plaintext

4.2. Training the classifiers

Our training set consists of 364 files selected randomly from the complete set. Specifically, for filetypes where we have over 1000 files available, we randomly selected 0.5% of the files. For filetypes where we have 1000 or less files available, we selected 5 at random. We used leave-one-out cross validation to train our classifiers (i.e., to select the value for k) as described in Section 3.4. When we use the NCD distance measure and Trailers (i.e., the last half, up to 16 KB of the files), the training process found $k=3$ for both the distance weighted and non-weighted classifiers. In all other cases, the training process found a value for k equal to 1. For both k-nearest neighbors as well as distance weighted k-nearest neighbors, this is simply equivalent to a nearest-neighbor approach which uses the closest example for prediction. This is true even for the distance weighted version since with $k=1$ the nearest neighbor's vote is not being weighted against any alternatives. Due to this, in our experiments, we consider five classifiers, the 4 non-weighted options and distance weighted using NCD with Trailers.

4.3. Evaluating classification performance

In Table 1, we compare the classification accuracy of our 5 classifiers on the remaining 39,913 files after excluding those from the training set.

Additionally, in Fig. 1, we show Precision–Recall graphs for the 4 combinations of distance measure (NCD or CDM) and file part (Headers or Trailers). In generating the Precision–Recall graphs, for each of the 39,913 testcases, the classifier's training cases were sorted by their distance to the testcase—essentially using the distance measure to rank the "relevance" of the classifier's training cases to a testcase query. From this, Precision–Recall curves were computed for each testcase, and averaged over the 39,913 testcases to obtain the curves in Fig. 1. The Precision–Recall curves can be used to characterize the performance of our approach on a task where we have a model file of unknown format or unknown origin, and we need to determine what we can do with it. For example, what CAD systems is it compatible with? What other formats can we convert it to? What 3D object reasoners support its format? A query that can obtain other models or relevant data of the same or like format can enable answering these questions.

We also consider (in Figs. 2 and 3) two smaller scale experiments designed to explore the efficacy of the different combinations of the distance measures with file portions at distinguishing among the available formats. Specifically, in the small scale experiments, 100 files were selected uniformly at random from the complete set of 39,913 to approximate the overall distribution of formats from the Repository, and another 100 files were selected randomly from among the 4 LEDA Graph formats in the Repository. From these, Precision–Recall curves were computed, and averaged over 100 cases of a variety of formats (Fig. 2) and averaged over 100 cases of the different LEDA graph types (Fig. 3).

The small scale experiments can be interactively recreated by the reader via Elsevier's Collage environment. Experiment Code Item 1 and Experiment Code Item 2 with Experiment Data Item 1 (list of testcases) generates the Precision–Recall graphs (Experiment Data Item 3 and Experiment Data Item 4) of Fig. 2. Experiment Data Item 2 provides a complete list of the testcases available within the Collage system to enable the reader to choose their own testcases (alternatively, testcases can be added to Experiment Data Item 1 from other sources via http). Likewise, Experiment Code Item 3 and Experiment Code Item 4 generates the Precision–Recall graphs (Experiment Data Item 6 and Experiment Data Item 7) of Fig. 3 from the input list of LEDA graph formatted testcase (Experiment Data Item 5). Also within Collage, Experiment Code Item 5 enables executing the classifier on a set of testcases of the reader's choosing (via input Experiment Data Item 9). The distance measure and file part can be selected via input Experiment Data Item 8.

We discuss the results in depth in the following subsection, including discussing the effects of using Headers vs. Trailers (Section 4.3.1), the NCD distance vs. the CDM distance (Section 4.3.2), the power of the approach at distinguishing among very similar formats (Section 4.3.3), especially challenging scenarios for the approach (Section 4.3.4), and considerations for distance weightings (Section 4.3.5).

4.3.1. Headers vs. trailers

For both distance measures, NCD and CDM, overall classification accuracy over the entire dataset (Table 1) is higher when we use Headers (the first 16 KB of the files) rather than Trailers (the last 16 KB of the files). When using Headers, we correctly classify 89.81% and 88.13% of the repository's files (with NCD and CDM respectively) as compared to only 81.41% and 77.41% when using Trailers for k nearest neighbors, and only 82.82% when using Trailers, NCD, and distance-weightings. In particular, note in Fig. 1 that for all levels of recall, precision is highest when Headers are used. The Precision–Recall curves associated with the use of Headers very clearly dominate those when Trailers are used. The results of the small-scale experiment are consistent with this as can be seen in Fig. 2.

There are a couple rather notable exceptions to this, however, where the use of trailers leads to a more accurate classifier. Consider the types LEDA-Graph-GICL-3 and LEDA-Graph-GICL-4. These are LEDA graph formats used by Drexel University's Geometric and Intelligent Computing Laboratory for various 3D CAD search algorithms. For files of type LEDA-Graph-GICL-3, classification accuracy is at 92.85% when using Trailers and the NCD distance measure, compared to only 85.82% when Headers are used. This is particularly interesting since LEDA-Graph-GICL-3 is a format that is almost identical to LEDA-Graph-GICL-2 except that LEDA-Graph-GICL-3 has additional parameters stored at the end of the file after the LEDA graph data. More startling is the difference in classification accuracy when we look at LEDA-Graph-GICL-4. When Trailers and NCD are used, we correctly classify 99.55% of files of type LEDA-Graph-GICL-4 (nearly all of them), but no more than around 72% if we use Headers or if we use CDM with Headers or Trailers. LEDA-Graph-GICL-4 is a LEDA graph format, but rather different than the other 3 GICL LEDA graph types.

As a counterpart to our overall small scale experiment, we also conducted a smaller scale experiment, specifically with files of the 4 LEDA graph formats. This experiment may correspond to a case where a collection of legacy data files may need to be classified,

Table 1
Classification accuracy for entire dataset and organized by filetype. *N* is the number of files. NCD or CDM refers to the distance measure used by the nearest neighbors classifier. Headers or Trailers refer to whether the beginning or the end of large files were used for classification.

	N	*k*-nearest neighbors				Distance weighted nearest neighbors Trailers
		Headers		Trailers		
		NCD	CDM	NCD	CDM	NCD
ALL FILES	39,913	89.81	88.13	81.41	77.41	82.82
ACIS-Solid-Modeler-format	7859	99.36	99.57	98.97	99.43	98.97
Graphics-Interchange-Format	4491	85.79	77.40	68.69	23.96	68.69
VRML	3051	98.20	96.33	88.69	90.95	88.79
Initial-Graphics-Exchange-Specification	2408	97.84	96.89	91.90	90.91	91.90
LEDA-Graph-GICL-3	1805	85.82	82.55	92.85	82.55	92.63
LEDA-Graph-GICL-2	1695	90.56	90.91	72.86	91.62	72.74
Postscript	1599	89.37	89.18	87.80	87.43	87.80
GeMS-to-ACIS-Map-file	1599	99.44	99.56	99.25	99.56	99.25
ProE-Part-Format	1575	69.84	67.94	65.46	53.78	65.46
GeMS-Generic-Memory-Structures-file	1574	100.00	100.00	99.36	99.36	99.36
LEDA-Graph-GICL-4	1564	72.51	72.51	99.55	71.80	99.55
LEDA-Graph-GICL-1	1188	98.23	98.23	90.07	97.47	94.61
STEP-Application	847	99.65	99.65	77.69	80.40	77.69
GZIP-Compressed	843	19.57	18.62	18.15	15.78	18.15
Unix-Compressed-Archive	714	80.53	80.67	23.39	29.13	23.39
ACIS-Test-Harness-Script	702	84.76	71.94	76.92	71.94	76.92
HTML-File	608	93.09	91.45	90.79	91.45	90.95
Plain-Text	597	71.52	73.37	19.60	73.53	65.83
ProE-Assembly-Format	388	84.54	81.96	42.53	67.53	42.78
Parasolid-Text	343	100.00	100.00	45.77	45.77	45.77
CMU-IAMS-Part-file	323	100.00	100.00	55.42	100.00	100.00
CMU-IAMS-Facet-File	297	96.63	96.63	93.27	93.27	93.27
Java-Source-Code	293	90.78	89.08	83.28	89.08	83.28
C-Source-Code	250	54.80	26.00	43.60	38.00	44.40
JPEG-File	243	95.47	95.47	2.06	3.70	2.06
Bentley-Systems-Microstation	198	100.00	100.00	87.88	87.88	87.88
Encapsulated-Postscript	209	84.69	84.21	35.41	40.67	44.50
Autodesk-Drawing-Exchange-Format	175	78.86	78.86	76.00	77.14	76.00
ACIS-Test-Harness-Monitor-file	173	99.42	99.42	99.42	99.42	99.42
Java-Class-File	164	100.00	100.00	92.68	93.90	92.68
VWS-Vertical-Workstation-Part-Design-file [25]	151	100.00	100.00	100.00	100.00	100.00
ProE-Drawing-File	145	54.48	53.79	49.66	49.66	49.66
ProE-Tool-Parameters-File	135	100.00	100.00	100.00	100.00	100.00
SDRC-I-DEAS-Model-file-1	131	93.89	91.60	43.51	78.63	61.07
SDRC-I-DEAS-Model-file-2	129	72.09	78.29	76.74	80.62	79.07
ProE-Cutter-Location-Data-File	126	84.13	88.10	60.32	60.32	60.32
ProE-Manufacturing-File	107	95.33	95.33	100.00	95.33	100.00
ProE-Neutral-file	104	88.46	88.46	78.85	77.88	78.85
3D-Design-Format-For-Data-Transfer-STL	104	95.19	96.15	98.08	98.08	98.08
Simple-Model-Format	100	96.00	96.00	39.00	41.00	40.00
C-Object-File	93	89.25	90.32	88.17	91.40	88.17
ProE-Tool-Path-File	87	100.00	100.00	36.78	88.51	72.41
SDRC-I-DEAS-Information-Processing-Report	79	98.73	98.73	98.73	98.73	98.73
Adobe-Portable-Document-Format	74	94.59	93.24	71.62	64.86	71.62
ProE-Configuration-file	71	49.30	71.83	43.66	71.83	43.66
X-Bitmap	69	97.10	98.55	97.10	97.10	97.10
Various-Log-Formats	64	53.13	57.81	40.63	57.81	51.56
SDRC-I-DEAS-Program-file	60	93.33	100.00	96.67	100.00	96.67
SDRC-I-DEAS-Archive	57	100.00	100.00	84.21	87.72	84.21
ZIP-Compressed	56	7.14	1.79	1.79	0.00	1.79
Header-File-C	54	74.07	90.74	85.19	90.74	88.89
ProE-Mechanical-Database	52	88.46	88.46	82.69	90.38	82.69
Autodesk-AutoCAD-Native-Format	45	95.56	95.56	84.44	80.00	84.44
ProE-Format-File	45	88.89	88.89	84.44	84.44	84.44

where the formats are all very similar (perhaps different versions of a single format), complicating the identification process. A set of 100 LEDA graphs were selected at random from the complete set from the Drexel Repository. All 4 LEDA graph formats are relied upon by various 3D CAD search algorithms. Most striking in the results is that when we use NCD with trailers, the classifier correctly classifies 87% of LEDA graphs (89% if the distance weighted classifier is used) into the correct class from among LEDA-Graph-GICL-1, LEDA-Graph-GICL-2, LEDA-Graph-GICL-3, and LEDA-Graph-GICL-4. This is compared to between 80% and 82% depending on which of the alternative classifiers are used. Note that we are using the same classifier as previously with all 54 formats available as class predictions. In Fig. 3, one should also note that precision remains very high (near or above 0.9) up to a recall level of 0.4 for all variations of the classifier; and continues to remain high up to a recall level around 0.7 when headers are used in particular.

Given that our experiments demonstrate that using the initial 16 KB of a file for format classification leads to higher accuracy levels as compared to using the final 16 KB, one may speculate that

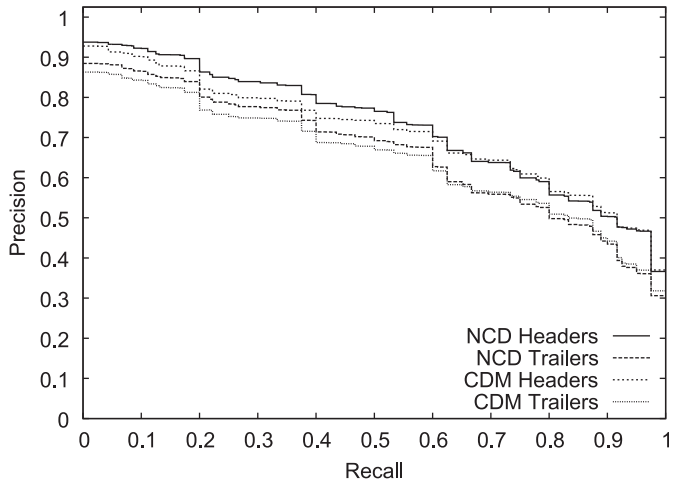

Fig. 1. Precision–Recall curves comparing the 4 different combinations of distance (NCD or CDM) and file part (Headers or Trailers) for the complete testset of 39,913 cases.

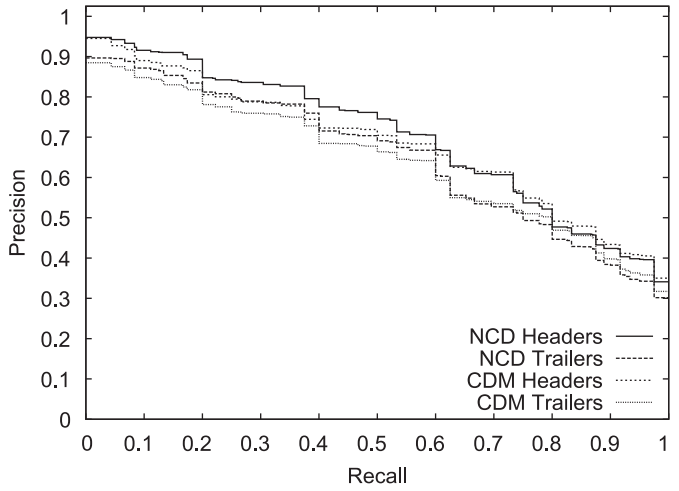

Fig. 2. Precision–Recall curves comparing the 4 different combinations of distance (NCD or CDM) and file part (Headers or Trailers) for a set of 100 files selected uniformly at random from the complete testset.

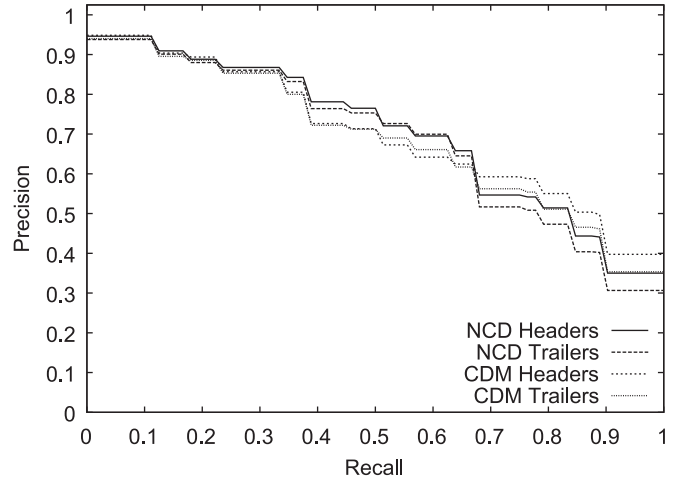

Fig. 3. Precision–Recall curves comparing the 4 different combinations of distance (NCD or CDM) and file part (Headers or Trailers) for a set of 100 files selected uniformly at random from among 4 different LEDA graph formats.

classification performance is largely due to recognizing the so-called "magic number" that many common file formats employ—i.e., the first couple bytes of the file (e.g., "GIF87a", "GIF89a", "#VRML", etc), and thus may speculate that using even less of the file will improve performance. However, there are several reasons to use as much of the file's contents as possible during the classification process. First, several of the CAD/CAM and related formats contained in the Drexel Repository do not appear to use such a "magic number" or other similar identifying mark at the start of the file. Second, there exist sets of closely related formats that may begin with the same identifying information. For example, all 4 LEDA Graph types contained in the Drexel Repository begin with a line "LEDA.GRAPH" and otherwise contain no explicit format identification mark. The differences among these 4 graph file formats is in what data is stored in the nodes of a graph and how edges are annotated, and in the case of LEDA-Graph-GICL-4 there is additional global information about the graph toward the end of the file. The first few bytes are insufficient to identify any of these formats. By using as much of the files as allowed by our approach (16 KB), we better enable the classifier to identify general syntactic patterns throughout the body of the file; and thus, can provide a much more robust classification. Finally, in the event of data corruption, by using a larger sample of the file's contents, our approach can be more robust to small modifications (e.g., a missing or altered "magic number").

4.3.2. NCD vs. CDM

Regardless of which part of the files are used for classification, Headers or Trailers, overall classification accuracy over the entire dataset is higher when we use NCD (Table 1). The difference is most clearly evident when trailers are used for classification–NCD in that case correctly classifies 81.41% of the files compared with 77.41% of the files when CDM is used. NCD is still superior to CDM when headers are used, but it is not as large a difference (89.81% vs. 88.13%)—although due to the size of the dataset, this amounts to around 670 more files correctly classified when NCD is used than when CDM is used. The Precision–Recall curves of Fig. 1 further demonstrate the performance difference between NCD and CDM. For recall levels from 0.0 up to about 0.6, the precision of NCD with Headers very clearly dominates the precision of CDM with Headers (top 2 curves), and the precision of NCD with Trailers very clearly dominates the precision of CDM with Trailers (bottom 2 curves).

There are a few formats that are exceptions to this general observation, but most of these exceptions are rather small differences between CDM and NCD. The more significant exceptions are mainly formats where we have fewer examples. For example, ProE Configuration Files are correctly classified more than 70% of the time when we use CDM (regardless of the use of headers or trailers) while less than 50% of the time when NCD is used. However, we have only 71 files of this type in the test set, and the classifier itself only has an additional 5 examples of this type used for classification. Another example is C language header files, where with CDM we can correctly classify over 90%, while we correctly classify only 74% of these files with NCD if headers are used and around 85% if trailers are used.

4.3.3. Distinguishing among very similar formats

Among our most exciting results relate to classification accuracy for groups of formats that are very similar in structure. There are several examples that can be found among the results. We highlight two of the most interesting cases here.

First, consider Postscript (PS) and Encapsulated Postscript (EPS). An EPS file is extremely similar in structure to a PS file. There are only a couple rather subtle differences (e.g., the first few bytes of

the formats differ slightly and EPS files include bounding box data). Our classifiers are able to detect these differences quite effectively. If we specifically look at our overall best performing classifier (using Headers and NCD), we find that 89.37% of the PS files in our dataset are correctly classified and 84.69% of our EPS files are correctly classified. Our dataset includes a relatively large number of PS files, 1599, and a more modest 209 EPS files, and yet our classifier is capable of distinguishing between these two types.

Our second, and most exciting example of the power of our classifier to distinguish among very similar types concerns 4 file formats developed within the Geometric and Intelligent Computing Laboratory (GICL) at Drexel University. We have been developing approaches to 3D CAD search since 1998, and the Design Repository includes file formats associated with the various CAD search related algorithms [27–31]. This set of formats pose an interesting challenge to the classifiers in that all 4 are either LEDA graphs [32] or a variation of a LEDA graph, but with minor differences in node and edge attributes. LEDA-Graph-GICL-1 is a LEDA graph where nodes are the faces from a boundary representation and the edges of the graph are the edges between the B-Rep faces. LEDA-Graph-GICL-2 is similar to a LEDA-Graph-GICL-1, but with node and edge attributes describing various properties of the B-rep faces and the interaction between adjoining faces. LEDA-Graph-GICL-3 is a LEDA-Graph-GICL-2 with extra parameters describing global properties of the 3D object tacked on at the end of the file, making LEDA-Graph-GICL-3 a non-standard LEDA graph. LEDA-Graph-GICL-4 is a LEDA graph where nodes are machining features extracted using Allied Signal's FBMach Machining Feature Recognition Husk [33,34] and edges between nodes correspond to non-empty feature intersections.

If you consider these 4 types, you will discover the following (unless otherwise noted, we are referring to our best overall performing classifier (using NCD and Headers):

- For the LEDA-Graph-GICL-1 format, we achieve a classification accuracy over 98%.
- For the LEDA-Graph-GICL-2 format, we achieve a classification accuracy over 90%.
- For the LEDA-Graph-GICL-3 format, we achieve a classification accuracy nearly 86%. It goes up to nearly 93% if trailers are used, which seem to help detect the additional global model parameters that follow the LEDA graph in this format.
- For the LEDA-Graph-GICL-4 format, we achieve a classification accuracy over 72%. This is an interesting result in that of the 4 types, this one is the most different. As noted, previously however, if we use trailers, we correctly classify 99.55% of these files.
- For all 4 LEDA graph types combined, we achieve a classification accuracy of 86.1% over the entire collection of LEDA graphs (6234 examples).
- For the small scale experiment consisting of 100 representative LEDA graphs from the repository, we achieve a classification accuracy of 82%.

Performance on these 4 types demonstrates just how powerful the approach can be at detecting the subtle format features necessary to distinguish among very similar formats. It should be further noted that our dataset has a very large number of each of these 4 types, as many as 1805 files of type LEDA-Graph-GICL-3 and only as few as 1188 for type LEDA-Graph-GICL-1 (see column N in Table 1).

4.3.4. Classification challenges

Although overall classification accuracy is very high, and although for the majority of formats in the dataset, accuracy is likewise high, there are formats that pose challenges to our classifier.

The biggest challenge faced by our approach are highly compressed formats. For example, only 19.57% of GZIP Compressed files are correctly classified, and only 7% of Zip Files. Although, note that we only have 56 Zip files in our dataset, while we have many more GZIP files (843). However, we are able to correctly classify over 80% of Unix compressed files and we have a very significant number of them in our dataset (714).

The challenge posed by highly compressed formats relates to our approach's reliance on compression to detect commonalities between pairs of files. As the files are already highly compressed, there is little more that can be done. Any occurrences of our classifier discovering elements of such a file that are in common with elements of another could be just as likely chance flukes as they can be true distinguishing characteristics.

In some cases, the classifier was able to overcome the challenge of compressed data. For example, it correctly classified nearly 86% of GIF images and over 95% of JPEG images. Any format specific identifying data in the headers of these formats is sufficiently detected by our classifier, despite the remainder of the file appearing in a compressed state.

4.3.5. On distance weightings

As previously indicated, during the training phase, cross validation demonstrated that k should be set to 1 in all cases except for when we use NCD and Trailers, where for both the basic version of nearest neighbors and its distance weighted variant, k should be set to 3. When $k=1$, both variations are the same and are simply a nearest neighbor classifier.

In the one case (NCD and Trailers) where cross validation produced a $k>1$, distance weighted k nearest neighbors outperformed its non-weighted counterpart (82.82% accurate vs. 81.41% accurate). This is still significantly lower than our best performing classifier, non-weighted nearest neighbors using the NCD distance measure and headers (the first 16 KB of the file).

4.4. Frequency of correct class among top predictions

Next, we examine how often the correct class of a file is among the top N predicted types for our various classifiers (for $N=1\ldots5$). Fig. 4 shows the percentage of the entire dataset (39,913 files) where the correct class appears among the first N predictions. The results are exciting and place additional emphasis on which classifiers are the best performers. Specifically, using the first half of the file, what we have called headers, clearly dominates over

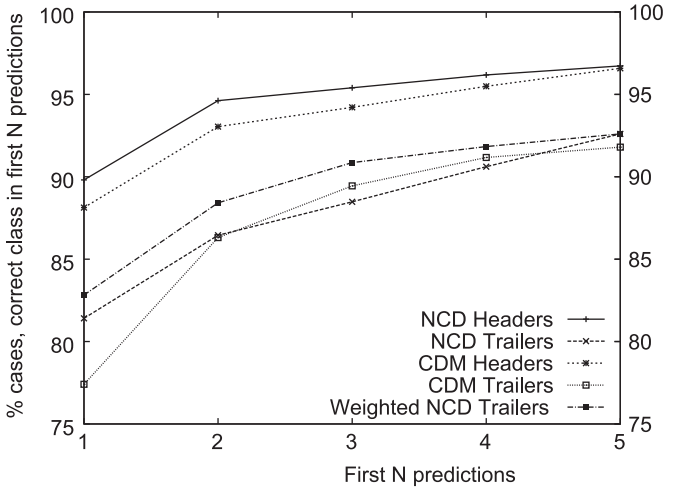

Fig. 4. Percentage of the complete testset (39,913 files) for which the correct class is found among the first N predictions (for $N\in\{1,2,3,4,5\}$) for the variations of our classifier.

using the second half of the file for classification, and that the NCD distance measure appears the more powerful distance measure.

First, consider the use of NCD and Headers. The correct classification is among the top 2 predicted classes for 94.62% of the files in the dataset (top curve in Fig. 4). This is compared to 93.04% when CDM is used with Headers. For contrast, consider the use of either NCD or CDM with Trailers or distance weighted nearest neighbors with NCD and Trailers (bottom 3 curves of Fig. 4). In those cases, even if we consider the top 5 predicted classes instead of just the top 2, we only find the correct classification 92.6%, 91.8%, and 92.6% of the time, respectively. For NCD and Headers, the correct classification is among the top 5 predicted classes for 96.73% of the files in the dataset (and 96.58% for CDM).

This shows that when our classifier mis-classifies, it is usually close. Our approach can greatly zero in on a small set of formats to be examined more closely by other means, perhaps manual. Furthermore, this is true even for formats that are otherwise problematic, such as highly compressed formats as can be seen in Fig. 5. For example, even though our classifier correctly predicts a Zip file's format only 7% of the time, the correct classification is among the top 5 predicted types 50% of the time (Fig. 5(a)); and for GZIP files we go from a classification accuracy of 19.57% to finding the correct class among the top 5 over 64% of the time (Fig. 5(b)).

For perhaps what is the most impressive example of the discriminating power of our classifier, consider the 4 LEDA graph

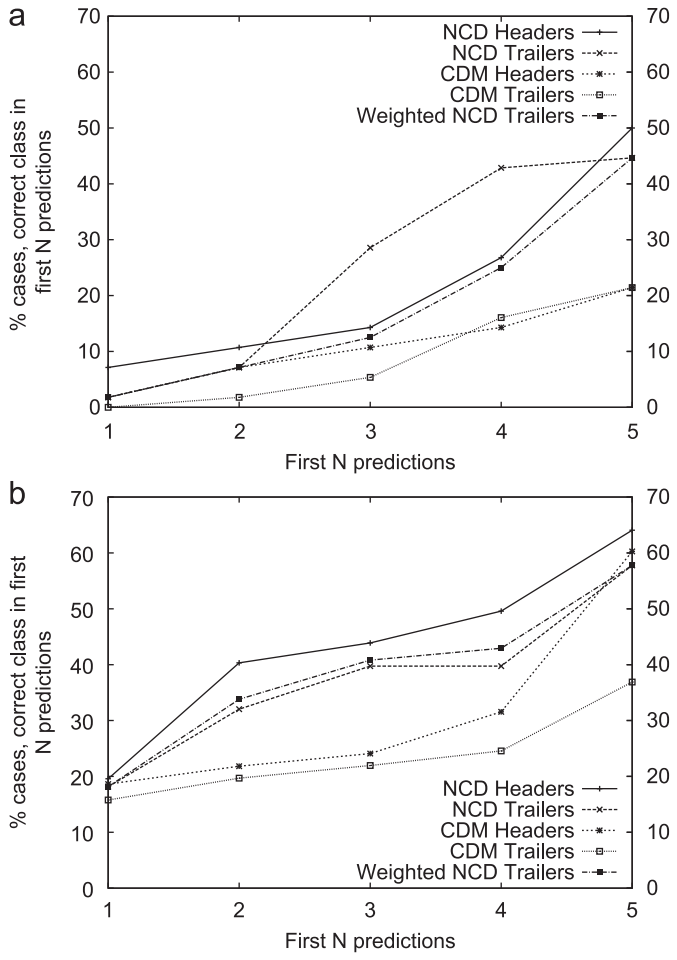

Fig. 5. Percentage of ZIP-Compressed files (a) and GZIP-Compressed files (b) for which the correct class is found among the first N predictions (for $N \in \{1,2,3,4,5\}$) for the variations of our classifier: (a) ZIP-Compressed (56 files) and (b) GZIP-Compressed (843 files).

formats from the Drexel Repository. Earlier in Section 4.3.3, we saw that our classifier is capable of accurately predicting the format of each of these 4 different LEDA graph types despite extreme similarities in their structure. Fig. 6(a–d) explores this further. For all 5 variations of our classifier, the correct class of any of the 4 LEDA graph types is within the first 4 predicted formats either 100% of the time or nearly 100% of the time. A user of the system who needs to identify a collection of such similar data formats can significantly narrow down on the possible formats of a file.

4.5. Execution time

In Table 2 we provide timing results for our entire dataset of 39,913 testcases. Our experiments were executed on an Ubuntu 10.10 Server with two AMD Opteron 244 Processors (1.8 GHz) and 4 GB of memory using the OpenJDK 64-bit Server VM, Java version 1.6. Using Trailers requires more CPU time to perform a classification than it does when Headers are used. When Headers are used, file access is limited to reading at most the first 16 KB of the files. Although with Trailers we are also limiting the amount of the file used for classification to 16 KB, the classifier must first determine where the final 16 KB begins and then must skip to that point. Therefore, in addition to the better classification performance, using Headers requires less computational resources.

For comparison, we turn to a benchmarking of several existing format identification tools that rely on format signatures. Specifically, van der Knijff et al. [35] benchmarked several identification tools, including JHOVE2 and DROID (version 6.0) using a set of 11,892 files from various scientific journal publishers. The total combined size of their dataset is 1.15 GB (approximately 104 KB per file). Their test machine was equipped with Windows XP, a 2.99 GHz processor, and 1 GB memory, and used Java 1.6 with the Java Hotspot Client VM. In this environment, JHOVE2 used an average of 7.6 s per file for format identification, and DROID used an average of 11.9 s per file.

There are several reasons why we cannot directly compare our classifier's average of 4.0 s per file format identification to these results (e.g., different test machine platform, and different dataset). However, the files in the Drexel Repository are on average larger (432 KB vs. 104 KB) and our test machine has a slower processor (1.8 GHz vs. 2.99 GHz). Additionally, the Drexel Repository contains a larger variety of file formats. Therefore, our compression-based approach to format classification does not appear to noticeably suffer in performance as compared to an approach that directly searches for known internal format signatures; and in fact, our approach may be faster.

We should expect the execution time of both our approach as well as the approach taken by JHOVE2 and DROID to increase proportionately with the number of supported formats. In our case, each new format means more training examples for the classifier to compare the query object. In the case of JHOVE2 and DROID, each new format means an additional format specification to run the object through. In our case, however, adding support for a new format is as simple as adding additional example cases to the classifier; while in the case of JHOVE2 or DROID one would need to implement an identification procedure for the relevant signature.

5. Conclusions

The digital revolution has produced overwhelming quantities and varieties of “born digital” engineering data. As systems evolve over decades, their digital records are essential to tasks such as intelligent maintenance, disassembly and demanufacture, redesign and reuse, compliance investigations and failure studies. Today's

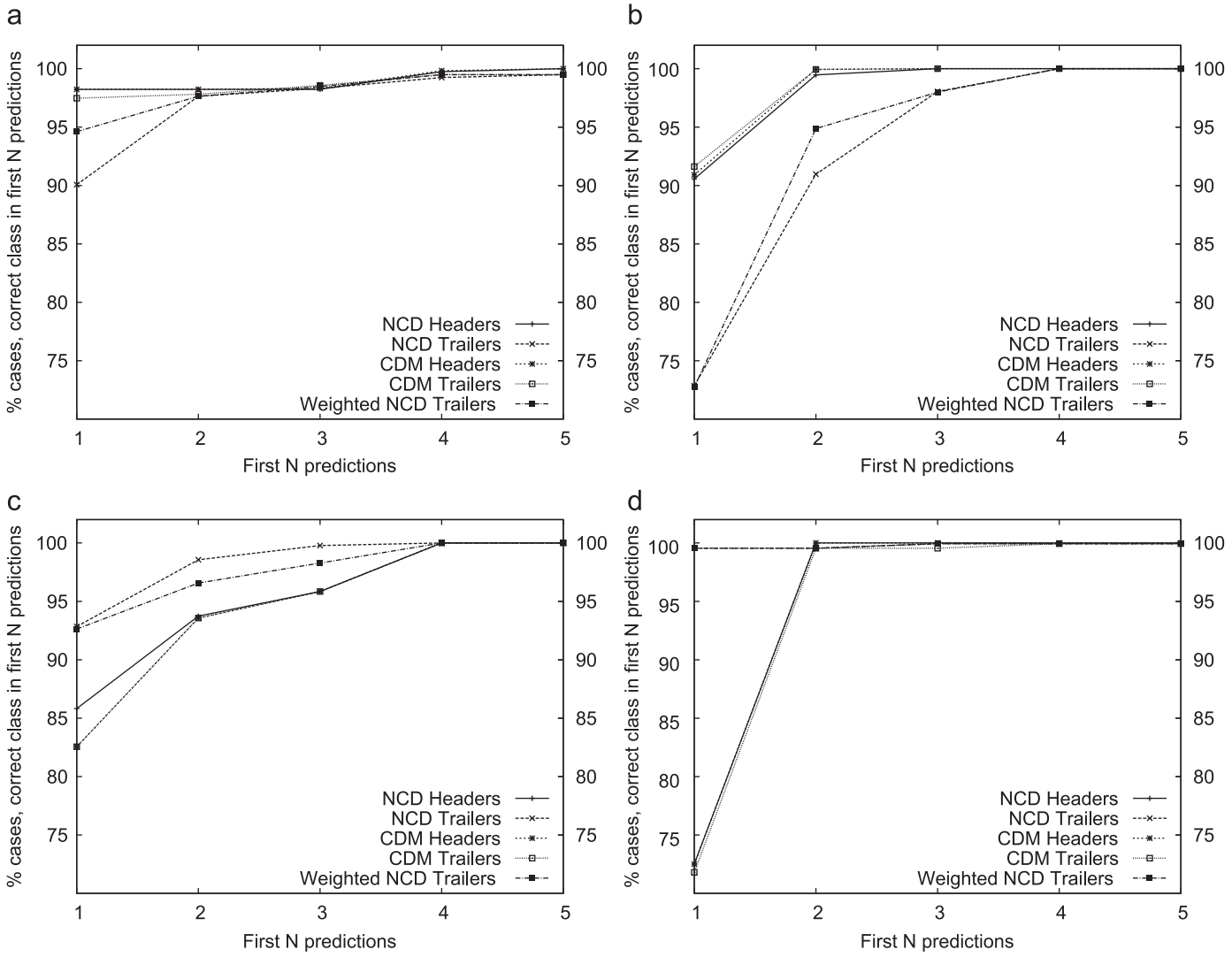

Fig. 6. Percentages of the 4 LEDA Graph types from the dataset for which the correct class is found among the first N predictions (for $N \in \{1, 2, 3, 4, 5\}$) for the variations of our classifier: (a) LEDA-Graph-GICL-1 (1188 files); (b) LEDA-Graph-GICL-2 (1695 files); (c) LEDA-Graph-GICL-3 (1805 files); and (d) LEDA-Graph-GICL-4 (1564 files).

Table 2
CPU timing statistics across entire dataset of 39,913 testcases.

Classifier	CPU time in seconds		
	Average	St. Dev.	Median
NCD Headers	4.030	2.340	3.73
NCD Trailers	4.172	2.480	3.91
CDM Headers	3.994	2.336	3.68
CDM Trailers	4.178	2.478	3.92

digital engineering tools consume and produce a wide variety of models in the course of a given project: preliminary CAD models lead to detailed CAD models, which in turn are used to create surrogate models for simulation, generate tooling models, etc.

In light of these emerging problems, this paper presented an approach to automated identification of engineering objects. As engineering activities become increasingly all digital, the proliferation of different physical file formats makes the sustainment and curation of engineering data very difficult. The tools and techniques presented in this paper work without any modeler or a priori knowledge about physical file structure and content. Using this approach, one can identify engineering data objects, their versions and provenance. This information is essential for those charged with engineering integration, data stewardship and records keeping—enabling them to automate the identification of file migration strategies, use of appropriate translators, and manage format diversity.

Acknowledgments

This work was supported by the National Science Foundation, Office of Cyber-Infrastructure, under the DataNet Federation Consortium Initiative under cooperative agreement #NSF/ OCI-0940841. Vincent Cicirello was supported through a sabbatical from the Richard Stockton College of New Jersey.

Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or the other supporting government and corporate organizations.

Appendix. Supporting information

Supplementary data associated with this article can be found in the online version at http://dx.doi.org/10.1016/j.cag.2013.03.007.

Note from publisher: this material was originally submitted as part of the Collage Executable Paper pilot, please visit http://www.elsevier.com/executablepaper for more information.

References

[1] Regli WC, Kopena JB, Grauer M. On the long-term retention of geometry-centric digital engineering artifacts. Comput-Aided Des 2011;43(7):820–37 Special Issue from the 2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling..

[2] Arms C, Fleischhauer C. Digital formats: factors for sustainability, functionality, and quality. In: Archiving 2005 conference proceedings. The Society for Imaging Science and Technology; 2005. p. 222–7.

[3] Library of Congress. Sustainability of digital formats: planning for library of congress collections. On the WWW. URL ⟨http://www.digitalpreservation.gov/formats⟩.

[4] Thilmany J. Ephemeral warehouse. Mech Eng 2005;127(9):1–10.

[5] Geometric and intelligent computing laboratory. The drexel university design repository, 2012. ⟨http://www.designrepository.org⟩.

[6] Regli WC, Gaines DM. An overview of the NIST repository for design, process planning, and assembly. Comput-Aided Des 1997;29(12):895–905.

[7] Regli WC, Cicirello VA. Managing digital libraries for computer-aided design. Comput-Aided Des 2000;32(2):119–32.

[8] JSTOR, Harvard University. Jhove–Jstor/Harvard object validation environment; 2012. ⟨http://jhove.sourceforge.net/⟩.

[9] Abrams S. Digital object format validation. In: Digital library federation: fall forum 2003; 2003. Available at: ⟨http://old.diglib.org/forums/fall2003/abrams/DLF-formats-Abrams_files/v3_document.htm⟩.

[10] Littman J. A technical approach and distributed model for validation of digital objects. D-Lib Mag 2006;12(5).

[11] Abrams S. What? so what: the next-generation jhove2 architecture for format-aware characterization. Int J Digital Curation 2009;4(3):123–36.

[12] Brown A. Automatic format identification using PRONOM and DROID. Digital preservation technical paper; The National Archives of the UK; 2006. Available at: ⟨http://www.nationalarchives.gov.uk/aboutapps/fileformat/pdf/automatic_format_identification.pdf⟩.

[13] McHenry K, Kooper R, Bajcsy P. Towards a universal, quantifiable, and scalable file format converter. In: Proceedings of the 2009 fifth ieee international conference on e-Science. IEEE Computer Society; 2009. p. 140–7.

[14] Li M, Chen X, Li X, Ma B, Vitányi PMB. The similarity metric. IEEE Trans Inf Theory 2004;50(12):3250–64.

[15] Cilibrasi R, Vitányi PMB. Clustering by compression. IEEE Trans Inf Theory 2005;51(4):1523–45.

[16] Keogh E, Lonardi S, Ratanamahatana CA. Towards parameter-free data mining. In: KDD-2004: Proceedings of the tenth ACM SIGKDD international conference on knowledge discovery and data mining. ACM; 2004. p. 206–15.

[17] Hall GA, Davis WP. Sliding window measurement for file type identification. Technical Report, ManTech Security and Mission Assurance; 2006. Available at: ⟨http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.113.8439⟩.

[18] Kocsor A, Kertész-Farkas A, Kaján L, Pongor S. Application of compression-based distance measures to protein sequence classification: a methodological study. Bioinformatics 2006;22(4):407–12.

[19] Burkovski A, Klenk S, Heidemann G. Similarity calculation with length delimiting dictionary distance. In: Proceedings of the 2011 IEEE 23rd international conference on tools with artificial intelligence. ICTAI '11; 2011. p. 856–64.

[20] Campana BJL, Keogh EJ. A compression-based distance measure for texture. Stat Anal Data Min 2010;3(6):381–98.

[21] Mitchell TM. Machine learning. McGraw-Hill; 1997.

[22] Li M, Vitányi P. An introduction to Kolmogorov complexity and its applications. Texts in computer science. 3rd ed.Springer; 2008.

[23] Deutsch P. DEFLATE compressed data format specification version 1.3. RFC 1951. The Internet Engineering Task Force (IETF); 1996. Available from ⟨http://www.ietf.org/rfc/rfc1951.txt⟩.

[24] Java Platform, Standard Edition 7 API Specification: Package java.util.zip. Oracle Corporation; 2012. Available from ⟨http://docs.oracle.com/javase/7/docs/api/java/util/zip/package-summary.html⟩.

[25] Kramer TR. A parser that converts a boundary representation into a features representation. Int J Comput Integrated Manuf 1989;2(3):154–63.

[26] Gupta SK, Paredis CJ, Sinha R, Brown PF. Intelligent assembly modeling and simulation. Assem Autom 2001;21(3):215–35.

[27] McWherter D, Peabody M, Regli WC, Shokoufandeh A. Solid model databases: techniques and empirical results. J Comput Inf Sci Eng 2001:300–10.

[28] Transformation Invariant Shape Similarity Comparison of Solid Models; 2001.

[29] Techniques for Indexing and Clustering of Solid Models; 2001.

[30] Ip CY, Lapadat D, Sieger L, Regli WC. Using shape distributions to compare solid models. In: Proceedings of the seventh ACM symposium on solid modeling and applications. SMA'02. ACM; 2002. p. 273–80.

[31] Cicirello VA, Regli WC. Machining feature-based comparison of mechanical parts. In: Proceedings of the international conference on shape modeling and applications. IEEE Computer Society Press; 2001. p. 176–85.

[32] Mehlhorn K, Naher S. LEDA: a platform for combinatorial and geometric computing. Cambridge University Press; 1999.

[33] Brooks SL, Greenway RB. Using STEP to integrate design features with manufacturing features. In: ASME computers in engineering conference. ASME; 1995. p. 579–86.

[34] Han J, Regli WC, Brooks S. Hint-based feature-recognition. In: ASME computers in engineering conference. ASME; 1997.

[35] van der Knijff J, Wilson C. Scape: scalable preservation environments: evaluation of characterisation tools. Technical Report. National Library of the Netherlands; 2011. Available at: ⟨http://www.openplanetsfoundation.org/system/files/SCAPE_PC_WP1_identification21092011.pdf⟩.