Machine Learning for Image Classification and Clustering Using a Universal Distance Measure

Uzi Chester and Joel Ratsaby*

Electrical and Electronics Engineering Department,
Ariel University of Samaria, ARIEL 40700
ratsaby@ariel.ac.il
http://www.ariel.ac.il/sites/ratsaby/

Abstract. We present a new method for image feature-extraction which is based on representing an image by a finite-dimensional vector of distances that measure how different the image is from a set of image prototypes. We use the recently introduced Universal Image Distance (UID) [1] to compare the similarity between an image and a prototype image. The advantage in using the UID is the fact that no domain knowledge nor any image analysis need to be done. Each image is represented by a finite dimensional feature vector whose components are the UID values between the image and a finite set of image prototypes from each of the feature categories. The method is automatic since once the user selects the prototype images, the feature vectors are automatically calculated without the need to do any image analysis. The prototype images can be of different size, in particular, different than the image size. Based on a collection of such cases any supervised or unsupervised learning algorithm can be used to train and produce an image classifier or image cluster analysis. In this paper we present the image feature-extraction method and use it on several supervised and unsupervised learning experiments for satellite image data. The feature-extraction method is scalable and is easily implementable on multi-core computing resources.

1 Introduction

Image classification research aims at finding representations of images that can be automatically used to categorize images into a finite set of classes. Typically, algorithms that classify images require some form of pre-processing of an image prior to classification. This process may involve extracting relevant features and segmenting images into sub-components based on some prior knowledge about their context [2,3].

In [1] we introduced a new distance function, called Universal Image Distance (UID), for measuring the distance between two images. The UID first transforms each of the two images into a string of characters from a finite alphabet and then uses the string distance of [4] to give the distance value between the images. According to [4] the distance between two strings x and y is a normalized

* Corresponding author.

N. Brisaboa, O. Pedreira, and P. Zezula (Eds.): SISAP 2013, LNCS 8199, pp. 59–72, 2013.

difference between the complexity of the concatenation xy of the strings and the minimal complexity of each of x and y. By complexity of a string x we mean the Lempel-Ziv complexity [5].

In the current paper we use the UID to create a finite-dimensional representation of an image. The i^{th} component of this vector is a feature that measures how different the image is from the i^{th} feature catgory. One of the advantages of the UID is that it can compare the distance between two images of different sizes and thus the prototypes which are representative of the different feature categories may be relatively small. For instance, the prototypes of an *urban* category can be small images of size 45×70 pixels of various parts of cities.

In this paper we introduce a process to convert the image into a labeled case (feature vector). Doing this systematically for a set of images each labeled by its class yields a data set which can be used for training any supervised and unsupervised learning algorithms. After describing our method in details we report on the accuracy results of several classification-learning algorithms on such data. As an example, we apply out method to satellite image classification and clustering.

We note that our process for converting an image into a finite dimensional feature vector is very straightforward and does not involve any domain knowledge about the images. In contrast to other image classification algorithms that extract features based on sophisticated mathematical analysis, such as, analyzing the texture, the special properties of an image, doing edge-detection, or any of the many other methods employed in the immense research-literature on image processing, our approach is very basic and universal. It is based on the complexity of the 'raw' string-representation of an image. Our method extracts features automatically just by computing distances from a set of prototypes. It is therefore scalable and can be implemented using parallel processing techniques, such as on system-on-chip and FPGA hardware implementation [6,7,8].

Our method extracts image features that are unbiased in the sense that they do not employ any heuristics in contrast to other common image-processing techniques [2]. The features that we extract are based on information implicit in the image and obtained via a complexity-based UID distance which is an information-theoretic measure. In our method, the feature vector representation of an image is based on the distance of the image from some fixed set of representative class-prototypes that are initially and only once picked by a human user running the learning algorithm.

Let us now summarize the organization of the paper: in section 2 we review the definitions of LZ-complexity and a few string distances. In section 3 we define the UID distance. In section 4 we describe the algorithm for selecting class prototypes. In section 5 we describe the algorithm that generates a feature-vector representation of an image. In section 6 we discuss the classification learning method and in section we conclude by reporting on the classification accuracy results.

2 LZ-Complexity and String Distances

The UID distance function [1] is based on the LZ- complexity of a string. The definition of this complexity follows [4,5]: let S,Q and R be strings of characters that are defined over the alphabet $\mathcal{A}$. Denote by $l(S)$ the length of S, and $S(i)$ denotes the i^{th} element of S. We denote by $S(i,j)$ the substring of S which consists of characters of S between position i and j (inclusive). An extension $R = SQ$ of S is reproducible from S (denoted as $S \to R$) if there exists an integer $p \leq l(S)$ such that $Q(k) = R(p+k-1)$ for $k = 1, \ldots, l(Q)$. For example, $aacgt \to aacgtcgtcg$ with $p = 3$ and $aacgt \to aacgtac$ with $p = 2$. R is obtained from S (the seed) by first copying all of S and then copying in a sequential manner $l(Q)$ elements starting at the p^{th} location of S in order to obtain the Q part of R.

A string S is *producible* from its prefix $S(1,j)$ (denoted $S(1,j) \Rightarrow R$), if $S(1,j) \to S(1,l(S)-1)$. For example, $aacgt \to aacgtac$ and $aacgt \to aacgtacc$ both with pointers $p = 2$. The production adds an extra 'different' character at the end of the copying process which is not permitted in a reproduction.

Any string S can be built using a *production process* where at its i^{th} step we have the production $S(1,h_{i-1}) \Rightarrow S(1,h_i)$ where h_i is the location of a character at the i^{th} step. (Note that $S(1,0) \Rightarrow S(1,1)$).

An m-step production process of S results in parsing of S in which $H(S) = S(1,h_1) \cdot S(h_1+1,h_2) \cdots S(h_{m-1}+1,h_m)$ is called the *history* of S and $H_i(S) = S(h_{i-1}+1,h_i)$ is called the i^{th} component of $H(S)$. For example for $S = aacgtacc$ we have $H(S) = a \cdot ac \cdot g \cdot t \cdot acc$ as the history of S.

If $S(1,h_i)$ is not reproducible from $S(1,h_{i-1})$ then the component $H_i(S)$ is called *exhaustive* meaning that the copying process cannot be continued and the component should be halted with a single character *innovation*. Every string S has a unique exhaustive history [5].

Let us denote by $c_H(S)$ the number of components in a history of S. The LZ complexity of S is $c(S) = \min\{c_H(S)\}$ where the minimum is over all histories of S. It can be shown that $c(S) = c_E(S)$ where $c_E(S)$ is the number of components in the exhaustive history of S.

A distance for strings based on the LZ-complexity was introduced in [4] and is defined as follows: given two strings X and Y, denote by XY their concatenation then define

$$d(X,Y) := \max\{c(XY) - c(X), c(YX) - c(Y)\}.$$

In [1] we have found that the following normalized distance

$$\mathtt{d}(X,Y) := \frac{c(XY) - \min\{c(X), c(Y)\}}{\max\{c(X), c(Y)\}}. \tag{1}$$

performs well in classification and clustering of images.

We note in passing that (1) resembles the normalized compression distance of [9] except that here we do not use a compressor but instead resort to the LZ-complexity c of a string. Note that $\mathtt{d}$ is not a metric since it does not satisfy

the triangle inequality and a distance of 0 implies that the two strings are close but not necessarily identical. We refer to $\mathtt{d}$ as a universal distance because it is not dependent on some specific representation of a string nor on heuristics common to other string distances such as edit-distances [10]. D only depends on the LZ-complexity of each of the two strings and their concatenation and this is a pure information-quantity which depends on the string's context but not its representation.

3 Universal Image Distance

Based on $\mathtt{d}$ we now define a distance between images. The idea is to convert each of two images I and J into strings $X^{(I)}$ and $X^{(J)}$ of characters from a finite alphabet of symbols. Once in string format, we use $\mathtt{d}(X^{(I)}, X^{(J)})$ as the distance between I and J. The details of this process are described in Algorithm 1 below.

Algorithm 1. UID distance measure

1. **Input**: two color images I, J in jpeg format (RGB representation)
2. Transform the RGB matrices into gray-scale by forming a weighted sum of the R, G, and B components according to the following formula: $grayscaleValue := 0.2989R + 0.5870G + 0.1140B$, (used in Matlab©). Each pixel is now a single numeric value in the range of 0 to 255 . We refer to this set of values as the alphabet and denote it by $\mathcal{A}$.
3. Scan each of the grayscale images from top left to bottom right and form a string of symbols from $\mathcal{A}$. Denote the two strings by $X^{(I)}$ and $X^{(J)}$.
4. Compute the LZ-complexities: $c\left(X^{(I)}\right)$, $c\left(X^{(J)}\right)$ and the complexity of their concatenation $c\left(X^{(I)}X^{(J)}\right)$
5. **Output**: $UID(I,J) := \mathtt{d}\left(X^{(I)}, X^{(J)}\right)$.

Remark 1. The transformation into gray-scale is a matter of representational convenience. To deal with color images without this transformation one can create a 3D alphabet whereby each 'letter' in this alphabet corresponds to an RGB triple with each component in the range 0 to 255. This way the image color information remains in the string representation.

4 Prototype Selection

In this section we describe the algorithm for selecting image prototypes from each of the feature categories . This process runs only once before the stage of converting the images into finite dimensional vectors, that is, it does not run

once per image but once for all images. For an image I we denote by $P \subset I$ a sub-image P of I where P can be any rectangular-image obtained by placing a window over the image I where the window is totally enclosed by I. The size of the window depends on how much information a single prototype will convey about the associated feature-category.

In the following algorithm we use clustering as a simple means of validation that the prototypes selected maintain the inherent differences between the feature-categories (the clustering algorithm is not given the feature-category information but only the inter-prototype distance information).

Algorithm 2. Prototypes selection

1. **Input**: M image feature categories, and a corpus $\mathcal{C}_N$ of N unlabeled colored images $\{I_j\}_{j=1}^{N}$.
2. **for** $(i := 1$ **to** $M)$ **do**
   (a) Based on *any* of the images I_j in $\mathcal{C}_N$, let the user **select** L_i prototype images $\left\{P_k^{(i)}\right\}_{k=1}^{L_i}$ and set them as feature category i. Each prototype is contained by some image, $P_k^{(i)} \subset I_j$, and the size of $P_k^{(i)}$ can vary, in particular it can be much smaller than the size of the images I_j, $1 \leq j \leq N$.
   (b) **end for;**
3. **Enumerate** all the prototypes into a single *unlabeled* set $\{P_k\}_{k=1}^{L}$, where $L = \sum_{i=1}^{M} L_i$ and calculate the distance matrix $H = \left[UID\left(X^{(P_k)}, X^{(P_l)}\right)\right]_{k=1,l=1}^{L}$ where the (k,l) component of H is the UID distance between the unlabeled prototypes P_k and P_l.
4. **Run** hierarchical clustering on H and obtain the associated dendrogram (note: H does not contain any 'labeled' information about feature-categories, as it is based on the unlabeled set).
5. **If** there are M clusters with the i^{th} cluster consisting of the prototypes $\left\{P_k^{(i)}\right\}_{k=1}^{L_i}$ **then** terminate and **go to** step 7.
6. **Else go to** step 2**.**
7. **Output**: the set of labeled prototypes $\mathcal{P}_L := \left\{\left\{P_k^{(i)}\right\}_{k=1}^{L_i}\right\}_{i=1}^{M}$ where L is the number of prototypes.

From the theory of learning pattern recognition, it is known that the dimensionality M of a feature-vector is usually taken to be small compared to the data size N. A large L will obtain better feature representation accuracy of the image, but it will increase the time for running Algorithm 3 (described below).

Algorithm 2 convergence is based on the user's ability to select good prototype images. We note that from our experiments this is easily achieved primarily because the UID permits to select prototypes $P_k^{(i)}$ which are considerably *smaller* than the size of the full images I_j. For instance, in our experiments we used 45×70

pixels prototype size for all feature categories. This fact makes it easy for a user to quickly choose typical representative prototypes from every feature-category. This way it is easy to find informative prototypes, that is, prototypes that are distant when they are from different feature-categories and close when they are from the same feature category. Thus Algorithm 2 typically converges rapidly.

As an example, Figure 1 displays 12 prototypes selected by a user from a corpus of satellite images. The user labeled prototypes $1, \ldots, 3$ as representative of the feature category *urban,* prototypes $4, \ldots, 6$ as representatives of class *sea,* prototypes $7, \ldots, 9$ as representative of feature *roads* and prototypes $10, \ldots, 12$ as representative of feature *arid.* The user easily found these representative prototypes as it is easy to fit in a single picture of size 45×70 pixels a typical image. The dendrogram produced in step 4 of Algorithm 2 for these set of 12 prototypes is displayed in Figure 2. It is seen that the following four clusters were found $\{10, 12, 11\}, \{1, 2, 3\}, \{7, 8, 9\}, \{4, 6, 5\}$ which indicates that the prototypes selected in Algorithm 2 are good.

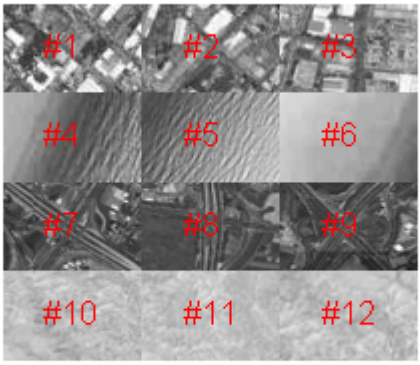

Fig. 1. Labeled *p*rototypes of feature-categories *urban*, *sea , roads*, and *arid* (each feature has three prototypes, starting from top left and moving right in sequence)

5 Image Feature-Representation

In the previous section we described Algorithm 2 by which the prototypes are manually selected. This algorithm is now used to create a feature-vector representation of an image. It is described as Algorithm 3 below (in [1] we used a similar algorithm UIC to soft-classify an image whilst here we use it to only produce a feature vector representation of an image which later serves as a single labeled case for training any supervised learning algorithm or a single unlabeled case for training an unsupervised algorithm).

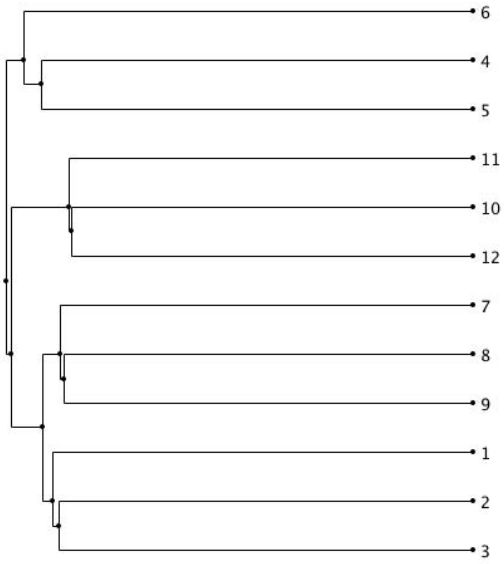

Fig. 2. Dendrogram of prototypes of Figure 1

Algorithm 3. Feature-vector generation

1. **Input**: an image I to be represented on the following feature categories $1 \leq i \leq M$, and given a set $\mathcal{P}_L := \left\{ \left\{ P_k^{(i)} \right\}_{k=1}^{L_i} \right\}_{i=1}^{M}$ of labeled prototype images (obtained from Algorithm 2).
2. **Initialize** the count variables $c_i := 0$, $1 \leq i \leq M$
3. Let W be a rectangle of size equal to the maximum prototype size. (See remark below)
4. Scan a window W across I from top-left to bottom-right in a non-overlapping way, and let the sequence of obtained sub-images of I be denoted as $\{I_j\}_{j=1}^{m}$.
5. **for** ($j := 1$ to m) **do**
   (a) **for** ($i := 1$ to M) **do**
      i. $temp := 0$
      ii. **for** ($k := 1$ to L_i) **do**
         A. $temp := temp + \left(UID(I_j, P_k^{(i)}) \right)^2$
         B. **end for;**
      iii. $r_i := \sqrt{temp}$
      iv. **end for;**
   (b) Let $i^*(j) := \text{argmin}_{1 \leq i \leq M} r_i$, this is the decided feature category for sub-image I_j.
   (c) **Increment** the count, $c_{i^*(j)} := c_{i^*(j)} + 1$
   (d) **end for;**
6. **Normalize** the counts, $v_i := \frac{c_i}{\sum_{l=1}^{M} c_l}$, $1 \leq i \leq M$
7. **Output**: the normalized vector $v(I) = [v_1, \ldots v_M]$ as the feature-vector representation for image I

Remark 2. In Step 3, we choose the size of W to be the maximal size of a prototype but this is not crucial since D can measure the distance between two images of different sizes. From our experiments, the size of W needs to be large enough such that the amount of image information in W is not smaller than that captured in any of the prototypes.

6 Supervised and Unsupervised Learning on Images

Given a corpus $\mathcal{C}$ of images and a set $\mathcal{P}_L$ of labeled prototypes we use Algorithm 3 to generate the feature-vectors $v(I)$ corresponding to each image I in $\mathcal{C}$. At this point we have a database $\mathcal{D}$ of size equal to $|\mathcal{C}|$ which consists of feature vectors of all the images in $\mathcal{C}$. This database can be used for *unsupervised* learning, for instance, discover interesting clusters of images. It can also be used for *supervised* learning provided that each of the cases can be labeled according to a value of some target class variable which in general may be different from the feature categories. Let us denote by T the class target variable and the database $\mathcal{D}_T$ which consists of the feature vectors of $\mathcal{D}$ with the corresponding target class values. The following algorithm describes the process of learning classification of images.

Algorithm 4. Image classification learning

1. **Input**: (1) a target class variable T taking values in a finite set $\mathcal{T}$ of class categories, (2) a database $\mathcal{D}_T$ which is based on the M-dimensional feature-vectors database $\mathcal{D}$ labeled with values in $\mathcal{T}$ (3) any supervised learning algorithm $\mathcal{A}$
2. Partition $\mathcal{D}_T$ using n-fold cross validation into Training and Testing sets of cases
3. Train and test algorithm $\mathcal{A}$ and produce a classifier C which maps the feature space $[0,1]^M$ into $\mathcal{T}$
4. Define Image classifier as follows: given any image I the classification is $F(I) := C(v(I))$, where $v(I)$ is the M-dimensional feature vector of I
5. **Output**: classifier F

7 Computational Time

Given an image I let us denote by $\tau(I)$ the total time that it takes Algorithm 3 to generate the case (vector-representation) $v(I)$ of I that can be used as a training case or as an input to the classifier F in order to classify the image I.

As mentioned above, Algorithm 2 involves a one-time manual selection of prototypes and the speed is dictated by the user (not the computer). Algorithms 3 is where the computational time is relevant. Step 5 of Algorithm 3 is the time-critical section which governs the overall computational time of the algorithm. This step iterates over all subimages I_j, $1 \leq j \leq m$, of the input image I, and for each subimage it computes the values r_i, $1 \leq i \leq M$, one for each feature-category. In order to compute r_i it computes the UID distance between I_j and

prototype $P_k^{(i)}$, $1 \leq k \leq L_i$. To compute $UID(I, J)$ requires building the exhaustive history of both strings $X^{(I)}$, $X^{(J)}$ and of their concatenation $X^{(I)}X^{(J)}$. So the time to compute $UID(I, J)$ is $O(c(X^{(I)}X^{(J)}))$ where $c(X^{(I)}X^{(J)})$ is the length of the exhaustive history of their concatenation. Denoting by $\tau(I, J)$ the time to compute $UID(I, J)$ then it is clear that $\tau(I, J)$ depends on the images I, J and not just on their sizes. That is, $\tau(I, J)$ depends on the LZ-complexity of the two images and on their similarity–the more similar the two, the less complex the concatenation string $X^{(I)}X^{(J)}$ and the smaller $\tau(I, J)$ is.

The time to compute the decided feature-category for subimage I_j of I is big-O the time that it takes to perform the j^{th} iteration of the outer for-loop of step 5. We refer to this as *subimage-time* and denote it by $\tau_j(I)$. We have

$$\tau_j(I) : = O\left(\sum_{i=1}^{M}\sum_{k=1}^{L_i}\tau(I_j, P_k^{(i)})\right)$$

where M is the number of categories, and L_i is the number of prototypes for category i.

Hence if we run on a single processor (single core) the case-generation time $\tau(I)$ of an image I is

$$\tau(I) = \sum_{j=1}^{m}\tau_j(I) \tag{2}$$

where m is the number of subimages in a single image I. It is clear from this formula that parallel computations (in particular, stream processing where the same function is applied to different data) can be very advantegeous for reducing the case-generation time $\tau(I)$.

For instance, on a processor with n cores, where $n \geq m$, each of the cores can compute in parallel a different subimage. This yields a total time

$$\tau(I) = O(\max_{1\leq j\leq m}\tau_j(I)).$$

If the number of cores n satisfies $m > n \geq M$ then we can let each core compute a different category-sum. This takes a single *sub-image-category* time

$$\tau_j^{(i)}(I) : = O\left(\sum_{k=1}^{L_i}\tau\left(I_j, P_k^{(i)}\right)\right)$$

and in this case the total time is

$$\tau(I) = \sum_{j=1}^{m}\max_{1\leq i\leq M}\tau_j^{(i)}(I). \tag{3}$$

If the number of cores $n \geq m \cdot \sum_{i=1}^{M} L_i$ then the total time to generate $v(I)$ from I is

$$\tau(I) = O\left(\max_{\substack{1\leq j\leq m\\ 1\leq i\leq M\\ 1\leq k\leq L_i}}\tau\left(I_j, P_k^{(i)}\right)\right). \tag{4}$$

8 Experimental Setup and Results

We created a corpus $\mathcal{C}$ of 60 images of size 670×1364 pixels from GoogleEarth©of various types of areas. Figure 3 displays a few scaled-down examples of such images. From these images we let a user define four feature-categories: *sea*, *urban*, *arid*, *roads* and choose three relatively-small image-prototype of size 45×70 pixels from each feature-category, that is, we ran Algorithm 2 with $M = 4$ and $L_i = 3$ for all $1 \leq i \leq M$. We then ran Algorithm 3 to generate the feature-vectors for each image in the corpus and obtained a database $\mathcal{D}$.

We then let the user label the images by a target variable *Humidity* with possible values 0 or 1. An image is labeled 0 if the area is of low humidity and labeled 1 if it is of higher humidity. We note that an image of a low humidity region may be in an arid (dry) area or also in the higher-elevation areas which are not necessarily arid. Since elevation information is not available in the feature-categories that the user has chosen then the classification problem is hard since the learning algorithm needs to discover the dependency between humid regions and areas characterized only by the above four feature categories.

With this labeling information at hand we produced the labeled database $\mathcal{D}_{Humidity}$. We used Algorithm 4 to learn an image classifier with target *Humidity.* As the learning algorithm $\mathcal{A}$ we used the following standard supervised algorithms: $J48$, $CART$, which learn decision trees, *NaiveBayes* and *Multi-Layer Perceptrons* (backpropagation) all of which are available in the WEKA©toolkit.

We performed 10-fold cross validation and compared their accuracies to a baseline classifier (denoted as ZeroR) which has a single decision that corresponds to the class value with the highest *prior* empirical probability. As seen in Table 1 (generated by WEKA©) $J48$, $CART$, NaiveBayes and Backpropagation performed with an accuracy of 86.5%, 81.5%, 89.25%, and 87.25%, respectively, compared to 50% achieved by the baseline $ZeroR$ classifier. The comparison concludes that all three learning algorithms are significantly better than the baseline classifier, based on a T-test with a significance level of 0.05.

Next, we performed clustering on the unlabeled database $\mathcal{D}$. Using the k-means algorithm, we obtained 3 significant clusters, shown in Table 2.

One can read the information directly from Table 2 and see that the first cluster captures images of *highly urban* areas that are next to concentration of roads, highways and interchanges. The second cluster contains *less populated* (urban) areas in arid locations (absolutely no sea feature seen) with very low concentration of roads. The third cluster captures the *coastal areas* and here we can see that there can be a mixture of urban (but less populated than images of the first cluster) with roads and extremely low percentage of arid land.

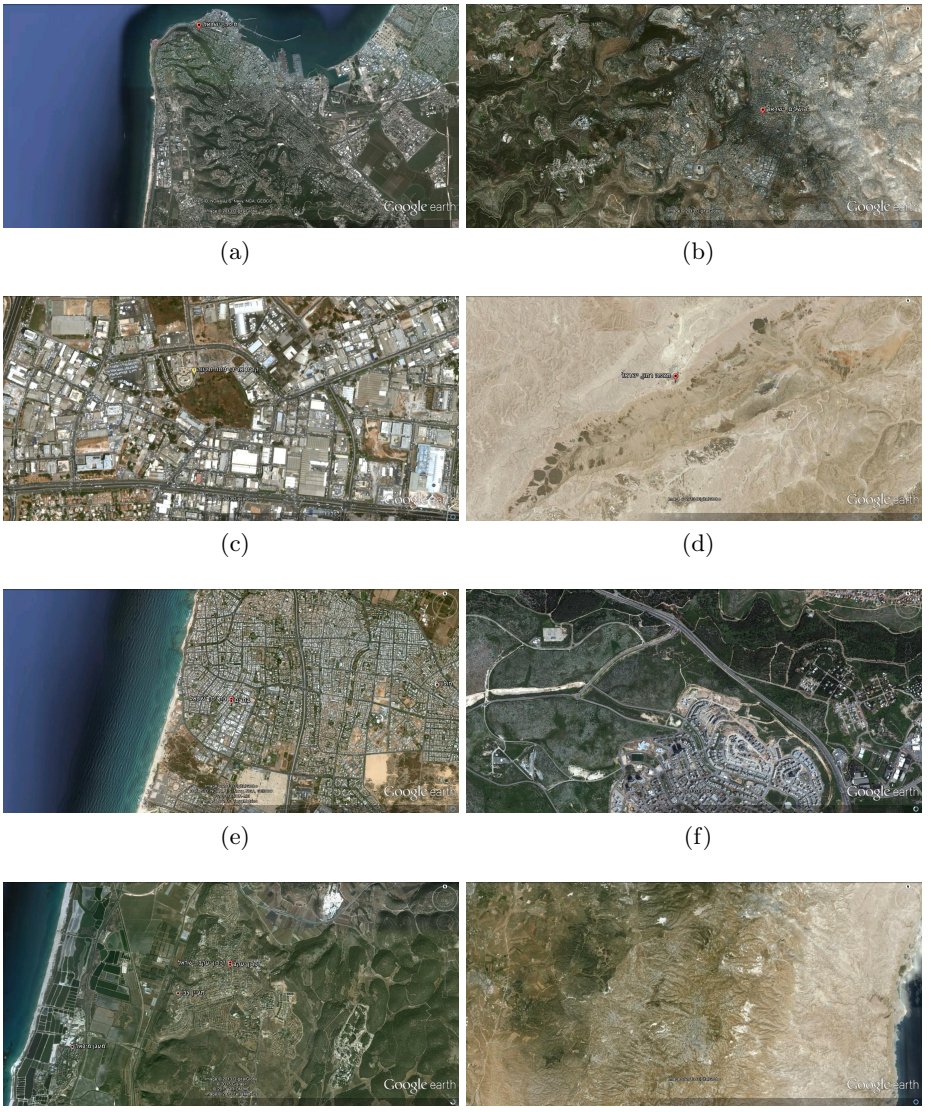

Fig. 3. Examples of images in the corpus

Table 1. Percent correct results for classifying *Humidity*

Dataset $\mathcal{D}_{Humidity}$	(1)	(2)	(3)	(4)	(5)
Classify Image into Humidity:	50.00	86.50 ∘	81.50 ∘	89.25 ∘	87.25 ∘

∘, • statistically significant improvement or degradation

(1) rules.ZeroR ” 48055541465867954
(2) trees.J48 ’-C 0.25 -M 2’ -217733168393644444
(3) trees.SimpleCart ’-S 1 -M 2.0 -N 5 -C 1.0’ 4154189200352566053
(4) bayes.NaiveBayes ” 5995231201785697655
(5) functions.MultilayerPerceptron ’-L 0.3 -M 0.2 -N 500 -V 0 -S 0 -E 20 -H a’ -5990607817048210779

Table 2. k-means clusters found on unsupervised database $\mathcal{D}$

Feature	Full data	Cluster#1	Cluster#2	Cluster#3
urban	0.3682	0.6219	0.1507	0.2407
sea	0.049	0.0085	0	0.1012
road	0.4074	0.2873	0.0164	0.655
arid	0.1754	0.0824	0.8329	0.003

The fact that such interesting knowledge can be extracted from raw images using our feature-extraction method is very significant since as mentioned above our method is fully automatic and requires no image or mathematical analysis or any sophisticated preprocessing stages that are common in image pattern analysis.

Let us report on some computational time statistics. The hardware we used is a 2.8Ghz AMD Phenom©II X6 1055T Processor with number of cores $n = 6$. The operating system is Ubuntu Linux 2.6.38-11-generic. The corpus consists of images of size 670×1364 pixels, with a sub-image and prototype sizes of 45×70 pixels, the number of subimages per image is $m = 266$. For this test we chose the number of feature-categories $M = 3$ with a total number of prototypes $\sum_{i=1}^{M} L_i = 11$. We implemented the algorithms in Matlab©in standard code style, that is, with no time-efficiency optimization, except in step 5.a. where the for statement is implemented using a `parfor` statement of Matlab. Note that in this case $m > n > M$ hence the total time to compute one image I is as stated in (3).

We obtained the set of average computational times

$$Timing\_Data = \left\{ \overline{\tau}_j^{(i)} : 1 \leq j \leq m, 1 \leq i \leq M \right\}$$

where

$$\overline{\tau}_j^{(i)} := \frac{1}{L_i} \sum_{k=1}^{L_i} \tau \left(I_j, P_k^{(i)} \right).$$

Figure 4 shows the histogram for this $Timing\_Data$, where the horizontal axis is time in units of seconds. The mean time is 0.851 sec. and the standard deviation

is 0.264 sec. Some of the state-of-the-art Graphics Processor Unit (GPU) accelerators have thousands of execution cores (see for instance, NVIDIA Tesla© K20 which has 2,496 cores) and are offered at current prices of approximately $2,000. On the NVIDIA Tesla© K10 the number of execution cores is $n = 3072$ and is greater than $m \sum_{i=1}^{M} L_i = 2,926$ so the total computation time $\tau(I)$ to process a single image I in the corpus will be as in (4), which for this example is approximately 0.851 sec. using Matlab with no optimization. We have not yet done so but we expect that transforming the code from Matlab to C and rewriting it with parallel processing code optimization can yield an average $\tau(I)$ which is lower by several orders of magnitude.

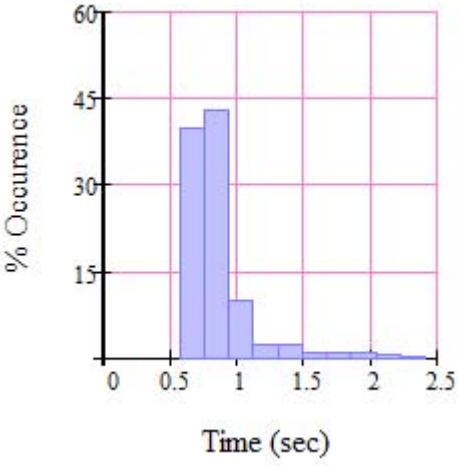

Fig. 4. Histogram of the computational times $\overline{\tau}_j^{(i)}$, $1 \leq j \leq m$, $1 \leq i \leq M$, $m = 266$, $M = 3$. The mean is 0.851.

9 Conclusion

We introduced a method for automatically defining and measuring features of colored images.The method is based on a universal image distance that is measured by computing the complexity of the string-representation of the two images and their concatenation. An image is represented by a feature-vector which consists of the distances from the image to a fixed set of small image prototypes, defined once by a user. There is no need for any sophisticated mathematical-based image analysis or pre-processing since the universal image distance regards the image as a string of symbols which contains all the relevant information of the image. The simplicity of our method makes it very attractive for fast and scalable implementation, for instance on a specific-purpose hardware acceleration chip. We applied our method to supervised and unsupervised machine learning on satellite images. The results show that standard machine learning algorithms perform well based on our feature-vector representation of the images.

Acknowledgements. We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Tesla K20 GPU used for this research.

References

1. Chester, U., Ratsaby, J.: Universal distance measure for images. In: 2012 IEEE 27th Convention of Electrical Electronics Engineers in Israel (IEEEI), pp. 1–4 (November 2012)
2. Canty, M.J.: Image Analysis, Classification and Change Detection in Remote Sensing: With Algorithms for Envi/Idl. CRC/Taylor & Francis (2007)
3. Lillesand, T.M., Kiefer, R.W., Chipman, J.W.: Remote sensing and image interpretation. John Wiley & Sons (2008)
4. Sayood, K., Otu, H.H.: A new sequence distance measure for phylogenetic tree construction. Bioinformatics 19(16), 2122–2130 (2003)
5. Ziv, J., Lempel, A.: On the complexity of finite sequences. IEEE Transactions on Information Theory 22(3), 75–81 (1976)
6. Ratsaby, J., Sirota, V.: Fpga-based data compressor based on prediction by partial matching. In: 2012 IEEE 27th Convention of Electrical Electronics Engineers in Israel (IEEEI), pp. 1–5 (November 2012)
7. Ratsaby, J., Zavielov, D.: An fpga-based pattern classifier using data compression. In: Proc. of 26th IEEE Convention of Electrical and Electronics Engineers, Israel, Eilat, November 17-20, pp. 320–324 (2010)
8. Kaspi, G., Ratsaby, J.: Parallel processing algorithm for Bayesian network inference. In: 2012 IEEE 27th Convention of Electrical Electronics Engineers in Israel (IEEEI), pp. 1–5 (November 2012)
9. Cilibrasi, R., Vitanyi, P.: Clustering by compression. IEEE Transactions on Information Theory 51(4), 1523–1545 (2005)
10. Deza, M., Deza, E.: Encyclopedia of Distances. Series in Computer Science, vol. 15. Springer (2009)