An incremental linear-time learning algorithm for the Optimum-Path Forest classifier

Moacir Ponti, Mateus Riva

Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo – São Carlos, SP 13566-590 Brazil

Abstract

We present a classification method with incremental capabilities based on the Optimum-Path Forest classifier (OPF). The OPF considers instances as nodes of a fully-connected training graph, arc weights represent distances between two feature vectors. Our algorithm includes new instances in an OPF in linear-time, while keeping similar accuracies when compared with the original quadratic-time model.

1. Introduction

The optimum-path forest (OPF) classifier [1] a classification method that can be used to build simple, multiclass and parameter independent classifiers. One possible drawback of using the OPF classifier in learning scenarios in which there is need to constantly update the model, is its quadratic training time. Let a training set be composed of n examples, the OPF training algorithm runs in $O(n^2)$. Some efforts were made to mitigate such running time by using several OPF classifiers trained with ensembles of reduced training sets [2] and fusion using split sets using multi-threading [3]. Also, recent work developed strategies to speed-up the training algorithm by taking advantage of data structures such as [1, 4]. However, an OPF-based method with incremental capabilities is still to be investigated, since sub-quadratic algorithms are important in many scenarios [5].

Incremental learning is a machine learning paradigm in which the classifier changes and adapts itself to include new examples that emerged after the initial construction of the classifier [6]. As such, an incremental-capable classifier has to start with an incomplete *a priori* dataset and include successive new data without the need to rebuild itself. In [4] the authors propose

an alternative OPF algorithm which is more efficient to retrain the model, but their algorithm is not incremental. Also, the empirical evidence shows that the running time is still quadratic, although with a significantly smaller constant. In this paper we describe an algorithm that can include new examples individually (or in small batches) in an already build model, which is a different objective when compared to [4] and [1]. In fact, we already used the improvements proposed by [1]. Therefore our new algorithm **does not compete**, but rather can be used as a complement for those variants.

Because OPF is based on the Image Foresting Transform for which there is a differential algorithm available (DIFT) [7], it would be a natural algorithm to try. However, DIFT is an image processing algorithm and includes all new pixels/nodes as prototypes, which would progressively convert the model into a 1-Nearest Neighbour classifier. Therefore we propose an alternative solution that maintains the connectivity properties of optimum-path trees.

Our OPF-Incremental (OPFI) is inspired in graph theory methods to update minimum spanning trees [8] and minimal length paths [9] in order to maintain the graph structure and thus the learning model. We assume there is an initial model trained with the original OPF training, and then perform several inclusions of new examples appearing over time. This is an important feature since models should be updated in an efficient way in order to comply with realistic scenarios. Our method will be useful everywhere the original OPF is useful, along with fulfilling incremental learning requirements.

2. OPF Incremental (OPFI)

The optimum-path forest (OPF) classifier [1] interprets the instances (examples) as the nodes (vertices) of a graph. The edges connecting the vertices are defined by some adjacency relation between the examples, weighted by a distance function. It is expected that training examples from a given class will be connected by a path of nearby examples. Therefore the model that is learned by the algorithm is composed by several trees, each tree is a minimum spanning tree (MST) and the root of each tree is called prototype.

Our OPF incremental updates an initial model obtained by the original OPF training by using the minimum-spanning tree properties in the existing optimum-path forest. Provided this initial model, our algorithm is able to include a new instance in linear-time. Note that in incremental learning scenarios it is typical to start with an incomplete training set, often presenting a poor accuracy due to the lack of a sufficient sample.

Our solution works by first classifying the new example using the current model. Because the label of the classified example is known, it is possible to infer if it has been conquered by a tree of the same class (i.e. it was correctly classified) or not. We also know which node was responsible for the conquest, i.e. its **predecessor**. Using this knowledge, we the have three possible cases:

1. **Predecessor belongs to the same class and is not a prototype**: the new example is inserted in the predecessor's tree, maintaining the properties of a minimum spanning tree.

2. **Predecessor belongs to the same class and is a prototype**: we must discover if the new example will take over as prototype. If so, the new prototype must reconquer the tree; otherwise, it is inserted in the tree as in the first case.

3. **Predecessor belongs to another class**: the new example and its predecessor become prototypes of a new tree. The new example will be root of an new tree; while the predecessor will begin a reconquest of its own tree, splitting it in two.

The Figure 1 illustrates the three cases when an element of the 'triangle' class is inserted in the OPF.
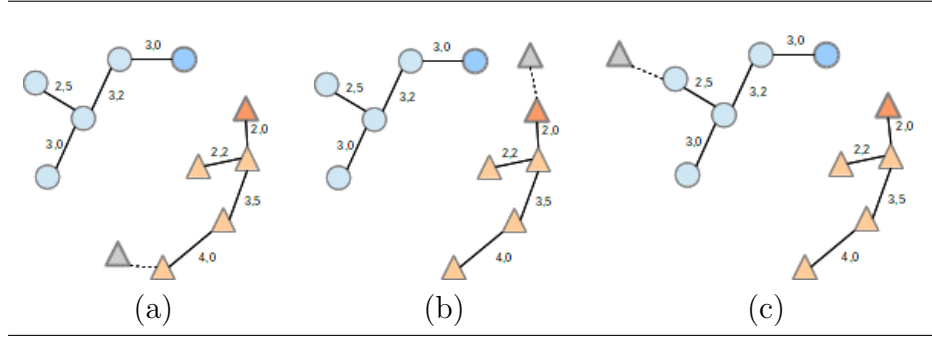

Figure 1: OPF-Incremental cases when adding a new example (a gray triangle): (a) conquered by a tree of the same class through a non-prototype, (b) conquered by a tree of the same class through a prototype, (c) conquered by a tree of a distinct class.

The classification and insertion of new elements is described on Algorithm 1 and shows the high-level solution described above.

Algorithm 1 OPF-Incremental insertion

Require: a previously trained OPF model T with n vertices; new instances to be included $Z[1...b]$.
1: OPF_Classify(Z, T) // as in [1]
2: **for** $i \leftarrow 1$ to b (each new example) **do**
3:   **if** $Z[i].label = Z[i].truelabel$ **then**
4:     **if** $Z[i].pred$ is *prototype* **then**
5:       recheckPrototype($Z[i]$,$Z[i].pred$,T) // Algorithm 3
6:     **else**
7:       insertIntoMST($Z[i]$,$Z[i].pred$,T) // Algorithm 2
8:     **end if**
9:   **else**
10:     $Z[i]$ becomes a prototype
11:     $Z[i].pred$ becomes a prototype
12:     reconquest($Z[i].pred\ Z[i].pred$, T) // Algorithm 4
13:   **end if**
14: **end for**
15: **return** T

Algorithm 2 OPF-Incremental MST insertion

Require: T is the graph; z is the new example; r is any vertex in the tree; t is a global variable and is the largest edge in the path between w to z, whereas m is the largest edge between r and z.
1: mark r "old"
2: $m \leftarrow (r, z)$
3: **for** each vertex w adjacent to r **do**
4:   **if** w is marked "new" **then**
5:     insertIntoMST(w, z, T) // recursive call
6:     $k \leftarrow$ the larger of the edges t and (w, r)
7:     $h \leftarrow$ the smaller of the edges t and (w, r)
8:     T gets the edge h
9:     **if** cost of $k <$ cost of m **then**
10:       $m \leftarrow k$
11:     **end if**
12:   **end if**
13: **end for**
14: $t \leftarrow m$
15: **return** T

The minimum spanning tree insertion function, described on Algorithm 2 is an adapted version of the minimum spanning tree updating algorithm proposed by [8]. The function for rechecking a prototype, described on Algorithm 3 takes the distance between the prototype and its pair (the corresponding prototype in the other tree, which edge was cut during the initial phase of classification), and between the new example and said pair. If the new example is closer to the pair, it takes over as prototype and reconquers the tree. Otherwise, it is inserted in the tree. The reconquest function was defined by [1], and described also here on Algorithm 4 for clarity.

Algorithm 3 OPF-Incremental recheck prototype

Require: an input node Z and its predecessor *pred*; a previously trained OPF model T; some distance function $dist(\cdot)$.
1: **if** $dist(Z, pred.pair) < dist(pred, pred.pair)$ **then**
2:    Z becomes a prototype
3:    reconquest(Z, Z, T) // Algorithm 4
4: **else**
5:    insertIntoMST(Z,*pred*,T) // Algorithm 2
6: **end if**
7: **return** T

Algorithm 4 OPF-Incremental reconquest

Require: an root node Z and its predecessor *pred*; a previously trained OPF model T; some distance function $dist(\cdot)$.
1: Z is marked "old"
   {In the first call the root is its own predecessor}
2: $newPathval \leftarrow dist(Z, pred)$
3: **if** $newPathval < Z.pathval$ **then**
4:    $Z.pred \leftarrow pred$
5:    $Z.pathval \leftarrow newPathval$
6:    **for** each adjacent w of Z **do**
7:       **if** w is not "old" **then**
8:          reconquest(w,Z,T) // recursive call
9:       **end if**
10:    **end for**
11: **end if**
12: **return** T

After the insertion is performed, an ordered list of nodes is updated as

in [1]. The new example is inserted in its proper position in linear time, thus allowing for the optimisation of the classification step.

When a new instance is inserted, we ensure that its classified label is equal to its true label, which is not always the case as in the original OPF algorithm because a given node can be conquered by a prototype with label different from the true label. Our method differs from the original OPF in this point, but we believe it is important to ensure the label of the new instance because the model is updated upon it and the label plays an important role for instance when a new class appears. Therefore, although our algorithm does not produces a model that is equal to the original OPF, it increments the model by maintaining the optimum path trees properties, rechecking prototypes and including new trees. Those are shown to be enough to achieve classification results that are similar to the original OPF.

3. Complexity Analysis

The complexity of inserting a new example into a model containing a total of n nodes is $\mathcal{O}(n)$, as we demonstrate for each case below.

(i) Predecessor is of another class. splitting a tree is $\mathcal{O}(1)$ since it only requires a given edge to be removed. The reconquest is $\mathcal{O}(n)$, since it goes through each node at most once as described by [1];

(ii) Predecessor is of same class and is a prototype. again, the complexity of the reconquest is $\mathcal{O}(n)$. Otherwise, it is an insertion, with complexity $\mathcal{O}(n)$ as described in the case (iii) below.

(iii) Predecessor is of same class and is not a prototype. the complexity of the operation is related to the inclusion of a new example on an existing tree. The complexity is $\mathcal{O}(n)$, or linear in terms of the number of examples, as proof below for Algorithm 2, showing that the function insertIntoMST() is able to update the MST in linear time.

Proof. Let z be the new example conquered by a vertex r on some tree. After executing line 5 of Algorithm 2, m and t are the largest edges in the paths from r to z, and from w (first vertex adjacent to r) to z respectively. Because the vertices are numbered in the order that they complete their calls to Algorithm 2 the linearity can be proven by induction.

6

Base step. The first node to complete its call, say w', must be a leaf node of the graph. Thus, lines 5 to 10 are skipped and t is assigned as (w', z) which is the only edge joining w' and z. If r' is a vertex incident to w', it is easy to see that $m = (r', z)$ both before and after the call insertIntoMST(w').

Induction steps. When executing line 5, i.e. insertIntoMST(w), if w is a leaf, again lines 5 and 9 are skipped and $t = (w, z)$. Otherwise, let x be the vertex which is incident to w and which is considered last in the call insertIntoMST(w). By induction hypothesis, m and t are the largest edges in the paths joining w and x to z, respectively, after executing insertIntoMST(x). It can be shown that in all cases t will be the largest edge in the path joining w to z. Similarly, m is the largest edge in the path joining r to z.

Also, in lines 6 to 9, the largest edge among m, (w, r) and t is deleted, and thus m and T (the MST) are updated. Since at most $n-1$ edges are deleted, each of which was the largest in a cycle, T will still remain a MST.

Because insertIntoMST(r) has $(n-1)$ recursive calls at line 5, the lines $1, 2$, 6–10 and 14 are executed n times. Lines 3 and 4 counts each tree edge twice (at most), and as this is proportional to the adjacency list, those are executed at most $2(n-1)$ times. Therefore, Algorithm 2 runs in $\mathcal{O}(n)$. $\quad\square$

4. Experiments

4.1. Data sets

The code and all datasets that are not publicly available can be found at `http://www.icmc.usp.br/~moacir/paper/16opfi.html`. A variety of synthetic and real datasets are used in the experiments.

Synthetic datasets: Base1/Base2/Base3: with size 10000 and data distributed in 4 regions. Base1 has 2 classes, Base2 has 4 classes and Base3 has 3 classes; Lithuanian, Circle vs Gaussian (C-vs-G), Cone-Torus and Saturn: are 2-d classes with different distributions.

Real datasets: CTG cardiotocography dataset, 2126 examples, 21 features, 3 classes; NTL (non-technical losses) energy profile dataset, 4952 examples, 8 features, 2 classes; Parkinsons dataset, 193 examples, 22 features, 2 classes; Produce image dataset, 1400 examples, 64 features, 14 classes; Skin segmentation dataset, 245057 examples, 3 features, 2 classes; Spam-Base email dataset, 4601 examples, 56 features, 2 classes; MPEG7-B shape dataset, 70 classes and 20 examples.

4.2. Experimental setup

Experiments were conduced to test the OPFI algorithm and comparing with the original OPF and DIFT. We aim to get similar accuracies with respect to them in linear time. Each experiment was conducted in 10-repeated hold-out sampling:

1. **Split data 50-50**: S for supervised training and T for testing, keeping the class distribution of the original dataset;

2. **Split** S: maintaining class proportions: BaseN, C-vs-G, Lithu, CTG, NTL, Parkinsons, Produce, SpamBase, Skin: into 100 subsets S_i, with $i = 0..99$. Cone-Torus, Saturn, MPEG7: into 10 subsets S_i, with $i = 0..9$ (fewer subsets because they have few examples/class).

3. **Initial training on** S_0 using original OPF as base to be incremented;

4. **Update model** including each S_i in sequence, starting with $i = 1$.

5. Results and Discussion

The balanced accuracy (takes into account the proportions of examples in each class) results are shown in Table 1. A plot with accuracy and running time results for 3 datasets are shown in Figure 2. By inspecting the average and standard deviations, it is possible to see that OPFI is able to keep accuracies similar to original OPF and DIFT. However, it runs faster then the OPF and does not degrade the graph structure as happens with DIFT. The optimum-path trees are preserved and therefore can be explored in scenarios when incremental learning is needed.

The running time curves shows the linear versus quadratic behaviour on all experiments: for n examples in the previous model, each new inclusion with original OPF would take $\mathcal{O}((n+1)^2)$, while with OPFI it is performed in $\mathcal{O}(n+1)$. When including examples in batches, our algorithm runs in $\mathcal{O}(n \cdot b)$, where b is the batch size, while OPF runs in $\mathcal{O}((n + b)^2)$. Therefore our method suits better several small inclusions than large batches. Neverthelles, OPFI's running time $(n \cdot b)$ is still $\mathcal{O}(n)$ and $o((n + b)^2)$.

We believe our contribution will allow the OPF method to be used more efficiently in future studies, such as data stream mining and active learning applications. Previous algorithms for decreasing the running time of the OPF training step can also be used within each batch of examples to be added to the OPFI algorithm to further speed-up the process.

(a) Lithuanian

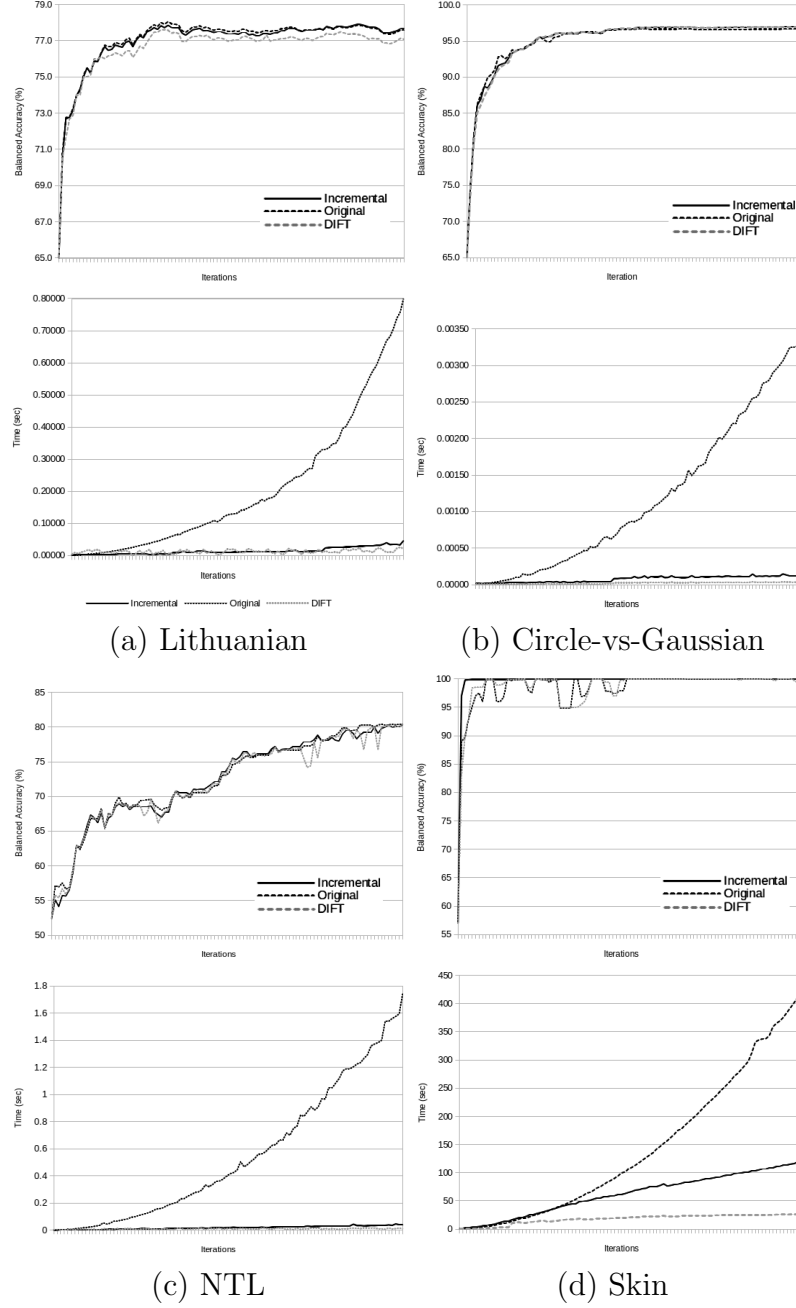(b) Circle-vs-Gaussian

(c) NTL

(d) Skin

Figure 2: Balanced accuracies (first and third rows) and running time (second and forth rows) for each iteration on the Lithuanian, Circle-vs-Gaussian, NTL and skin datasets

9

Table 1: Balanced accuracy results for the initial model, the first 3 increments, 50% and 100% of the increments

		S_0	1st	2nd	3rd	50%	100%
Base1	Incremental	85.3 ± 4.8	89.6 ± 1.8	91.8 ± 0.9	92.9 ± 0.8	98.0 ± 0.2	98.5 ± 0.1
	Original	85.3 ± 4.8	89.3 ± 2.5	91.1 ± 1.1	92.3 ± 0.8	98.0 ± 0.2	98.3 ± 0.2
	DIFT	85.3 ± 4.8	89.2 ± 1.8	91.5 ± 0.8	92.7 ± 0.7	97.7 ± 0.3	98.5 ± 0.1
Base2	Incremental	90.0 ± 2.2	93.4 ± 0.8	94.6 ± 0.6	95.0 ± 0.6	98.7 ± 0.5	99.1 ± 0.1
	Original	90.0 ± 2.2	92.8 ± 1.0	94.3 ± 0.9	95.1 ± 0.9	98.7 ± 0.7	99.1 ± 0.1
	DIFT	90.0 ± 2.2	93.2 ± 1.0	94.5 ± 0.7	94.9 ± 0.6	98.5 ± 0.4	99.1 ± 0.1
Base3	Incremental	88.7 ± 2.2	92.1 ± 1.6	93.8 ± 1.2	94.5 ± 0.8	98.5 ± 0.2	98.9 ± 0.1
	Original	88.7 ± 2.2	91.9 ± 1.6	93.6 ± 1.1	93.9 ± 0.8	98.3 ± 0.2	98.9 ± 0.1
	DIFT	88.7 ± 2.2	91.9 ± 1.4	93.8 ± 1.1	94.1 ± 0.8	98.5 ± 0.3	98.9 ± 0.1
C-vs-G	Incremental	64.9 ± 10.9	74.0 ± 7.0	81.2 ± 7.4	85.5 ± 5.4	96.8 ± 0.7	97.7 ± 0.8
	Original	64.9 ± 10.9	74.6 ± 8.1	81.9 ± 7.3	86.3 ± 5.5	96.8 ± 1.0	96.7 ± 1.0
	DIFT	64.9 ± 10.9	74.0 ± 8.2	81.0 ± 6.9	85.0 ± 4.8	96.9 ± 0.8	97.0 ± 0.8
Lithu	Incremental	65.0 ± 4.9	70.5 ± 4.4	72.8 ± 3.0	72.7 ± 2.9	77.4 ± 1.2	77.6 ± 0.6
	Original	65.0 ± 4.9	70.5 ± 4.0	71.7 ± 3.1	72.7 ± 3.4	77.1 ± 1.3	77.0 ± 1.0
	DIFT	65.0 ± 4.9	70.8 ± 4.3	72.7 ± 3.1	72.8 ± 2.9	77.1 ± 1.3	76.9 ± 0.7
Cone-Torus	Incremental	81.0 ± 2.6	83.7 ± 2.7	85.0 ± 1.7	85.8 ± 2.6	87.6 ± 1.1	87.9 ± 1.0
	Original	81.0 ± 2.6	82.9 ± 2.4	84.8 ± 2.1	85.0 ± 2.8	87.1 ± 0.8	87.3 ± 1.2
	DIFT	81.0 ± 2.6	83.6 ± 2.6	84.8 ± 1.6	85.5 ± 2.7	87.7 ± 1.1	87.1 ± 1.3
Saturn	Incremental	58.9 ± 11.6	68.5 ± 4.0	73.9 ± 5.3	78.3 ± 3.7	81.8 ± 3.8	88.2 ± 1.7
	Original	58.9 ± 11.6	69.0 ± 3.9	74.5 ± 5.5	78.4 ± 3.9	82.0 ± 4.0	88.0 ± 1.7
	DIFT	58.9 ± 11.6	68.4 ± 3.2	73.5 ± 5.4	78.1 ± 2.4	81.8 ± 3.7	87.6 ± 1.3
CTG	Incremental	72.2 ± 8.3	80.5 ± 2.7	81.0 ± 3.5	83.0 ± 2.9	92.8 ± 0.6	93.9 ± 0.8
	Original	72.2 ± 8.3	79.3 ± 3.0	80.8 ± 2.9	82.6 ± 2.7	92.5 ± 0.7	93.7 ± 1.2
	DIFT	72.2 ± 8.3	80.7 ± 2.9	81.1 ± 3.5	81.5 ± 2.9	92.4 ± 0.6	93.6 ± 1.1
NTL	Incremental	52.4 ± 3.5	54.1 ± 1.8	55.5 ± 2.1	56.9 ± 3.3	72.2 ± 0.9	82.0 ± 0.6
	Original	52.4 ± 3.5	53.6 ± 2.0	55.6 ± 2.0	57.0 ± 3.3	72.8 ± 1.0	82.0 ± 0.7
	DIFT	52.4 ± 3.5	54.1 ± 1.8	55.0 ± 3.0	55.6 ± 3.3	71.6 ± 0.9	80.4 ± 0.7
Parkinsons	Incremental	60.9 ± 11.8	62.0 ± 10.2	69.7 ± 12.2	72.6 ± 7.5	89.0 ± 3.8	89.4 ± 3.6
	Original	60.9 ± 11.8	61.4 ± 12.0	67.5 ± 11.9	72.1 ± 8.2	87.1 ± 5.6	88.1 ± 5.0
	DIFT	60.9 ± 11.8	62.8 ± 10.2	69.3 ± 12.3	72.2 ± 7.7	89.1 ± 3.7	89.5 ± 3.4
Produce	Incremental	63.6 ± 1.8	70.4 ± 2.1	74.4 ± 1.4	77.8 ± 0.8	95.2 ± 0.6	95.1 ± 0.6
	Original	63.6 ± 1.8	70.3 ± 2.1	74.3 ± 1.4	77.6 ± 0.7	95.2 ± 0.5	95.2 ± 0.7
	DIFT	63.6 ± 1.8	70.4 ± 2.1	74.4 ± 1.4	77.8 ± 0.8	94.5 ± 0.6	94.5 ± 0.6
SpamBase	Incremental	71.9 ± 3.1	76.0 ± 3.5	78.0 ± 2.4	78.7 ± 2.0	85.6 ± 0.7	87.6 ± 0.6
	Original	71.9 ± 3.1	75.8 ± 3.3	77.8 ± 2.2	78.5 ± 1.9	85.1 ± 0.7	87.0 ± 1.0
	DIFT	71.9 ± 3.1	76.0 ± 3.6	78.0 ± 2.4	78.6 ± 2.1	84.7 ± 0.7	85.6 ± 0.3
MPEG7-B	Incremental	72.9 ± 0.8	78.2 ± 0.9	81.2 ± 1.0	83.1 ± 0.9	85.4 ± 0.6	91.6 ± 0.4
	Original	72.9 ± 0.8	78.1 ± 0.9	81.1 ± 0.9	82.9 ± 0.8	85.3 ± 0.6	91.5 ± 0.4
	DIFT	72.9 ± 0.8	78.2 ± 0.9	81.2 ± 1.0	83.0 ± 0.8	85.5 ± 0.6	91.7 ± 0.5
Skin	Incremental	57.0 ± 1.8	97.0 ± 1.2	99.7 ± 0.2	99.8 ± 0.0	99.9 ± 0.0	99.9 ± 0.0
	Original	57.0 ± 1.8	89.0 ± 1.2	89.7 ± 0.4	93.5 ± 0.2	99.8 ± 0.1	99.9 ± 0.0
	DIFT	57.0 ± 1.8	83.0 ± 1.2	90.2 ± 1.0	93.1 ± 0.8	99.8 ± 0.2	99.9 ± 0.0

Acknowledgment

References

[1] J. P. Papa, A. Falcão, V. De Albuquerque, J. Tavares, Efficient supervised optimum-path forest classification for large datasets, Pattern Recognition 45 (1) (2012) 512–520.

[2] M. Ponti-Jr., I. Rossi, Ensembles of optimum-path forest classifiers using input data manipulation and undersampling., in: MCS 2013, Vol. 7872 of LNCS, 2013, pp. 236–246.

[3] M. Ponti-Jr, J. Papa, Improving accuracy and speed of optimum-path forest classifier using combination of disjoint training subsets, in: 10th Int. Work. on Multiple Classifier Systems (MCS 2011) LNCS 6713, Springer, Naples, Italy, 2011, pp. 237–248.

[4] A. S. Iwashita, J. P. Papa, A. Souza, A. X. Falcão, R. Lotufo, V. Oliveira, V. H. C. De Albuquerque, J. M. R. Tavares, A path-and label-cost propagation approach to speedup the training of the optimum-path forest classifier, Pattern Recognition Letters 40 (2014) 121–127.

[5] P. Berenbrink, B. Krayenhoff, F. Mallmann-Trenn, Estimating the number of connected components in sublinear time, Information Processing Letters 114 (11) (2014) 639–642.

[6] X. Geng, K. Smith-Miles, Incremental learning, Encyclopedia of Biometrics (2015) 912–917.

[7] A. X. Falcão, F. P. Bergo, Interactive volume segmentation with differential image foresting transforms, Medical Imaging, IEEE Transactions on 23 (9) (2004) 1100–1108.

[8] F. Chin, D. Houck, Algorithms for updating minimal spanning trees, Journal of Computer and System Sciences 16 (3) (1978) 333–344.

[9] G. Ausiello, G. F. Italiano, A. M. Spaccamela, U. Nanni, Incremental algorithms for minimal length paths, Journal of Algorithms 12 (4) (1991) 615–638.