Partial Classification Forest

Jonas Faßbender

jonas@fc-web.de

Abstract

I. Introduction

Some datasets do not allow a classifier to generate a descision surface good enough to be able to predict unseen observations well. Well, in this case, refers to a context dependent threshold for any quality measurement of a classifier, for example the accuracy or an information loss metric.

But for some of those problems, it may still be valuable to predict only on partitions of the feature space, in which the dataset is 'clean' enough, meaning a classifier can be found within the subset of the dataset laying inside one of those partitions which equals or exceeds the threshold.

This paper proposes a Monte Carlo based ensemble method called Partial Classification Forest (PCF), which builds an ensemble of trees having a structure similar to k-d trees to partition the feature space of the dataset in order to find 'clean' partitions. In the following a tree generated by the PCF is spelled Tree with a capital T, rather than tree, which is used to denote the tree data structure.

It should be noted here, that this paper is rather a Proof of Concept of a very early version of the PCF and has several shortcomings in research and empirical tests, due to a lack of time and no complete, fast implementation. I will discuss these shortcomings further in Chapter VI, but the most important ones I will list right away:

- Tests with a more sofisticated γ (cmp. II-A)
- The PCF's performance on high dimensional data
- Untested possible optimizations and features
- A Tree's growing behaviour
- Benchmarks

In Section II I will lay out the structure and the operations of a Tree generated by the PCF before, in Section III, describing how PCF utilizes Tree instances. After that I will continue displaying test results using the PCF. In Section V I will discuss further optimizations and possible additional features before finishing with a conclusion.

II. The Tree structure

A Tree generated by the PCF is a binary search tree structure similar to k-d trees. Its purpose is to randomly generate disjoint partitions of a feature space.

A Tree has two types of nodes, non-leaf nodes, here denoted as Nodes and leaf nodes denoted as Leaves. It provides two operations: (i) FIT, initializing the Tree and (ii) PREDICT, returning a label for an oberservation.

The Node structure contains three properties: (i) a split value; (ii) a left and (iii) a right successor, both references to either another Node or a Leaf.

A Leaf on the other hand, is the structure representing a partition of the feature space, having the following properties: (i) *active*: a boolean value deciding whether the partition's quality, determined during the FIT operation, is equal or better than the defined threshold or not; (ii) optionally a classifier which is used to classify oberservations during the PREDICT operation.

Only if a Leaf's *active* property is true, a classifier must be provided. A Leaf also has two vectors with arbitrary length as properties: (iii) a vector containing the observations of the dataset used in FIT, which are laying inside the partition and (iv) their inherent labels.

During the FIT operation a Tree contains a third type of node, Nil. Nil is used to initialize Trees and the left and right successor of a Node. These nodes are transformed during FIT to either a Node or a Leaf, so after the FIT operation a Tree does not contain Nil nodes anymore. A Nil node does not have any properties.

A. The FIT operation

The FIT operation constructs a Tree, based on a dataset split in observations (X) and their labels (y). Algorithm 1 shows how FIT recursively builds a Tree, which is at the beginning a pointer to a Nil node.

The most important parameter passed to FIT is γ. γ is a function returning (i) a classifier and (ii) the loss of it. Otherwise γ is treated as a black box by the PCF, so what the classifier is and how its loss is calculated are not relevant to the PCF, as long as the classifier is callable[1] and returns an element from the label set when beeing called (Algorithm 2, line 9). The loss returned by γ gets compared to the quality threshold τ_l. Is the loss $\leq \tau_l$ the classifier is good enough and Θ is transformed to an active Leaf (Algorithm 1, lines 2, 3).

There are two other thresholds besides τ_l, $\tau_{|X|}$, τ_h. Both regulate the behaviour of a Tree's growth. $\tau_{|X|}$ defines a minimum amount of observations a Leaf must contain. One can easily imagine, without $\tau_{|X|}$ or $\tau_{|X|} = 0$ a Tree would never stop growing, since FIT would continue to split empty partitions, trying to find a smaller partition which would be predictable, even though no classifier could be generated without observations to train it on.

[1]There could be another interface for the classifier, for example a predict method similar to the scikit-learn library.[2]

τ_h further regulates the maximum path length of a Tree. It is necessary besides $\tau_{|X|}$, because of the following scenario: be $\tau_{|X|} = 2$ and there are two equal observations in the dataset, but both having a different label than the other one. Now γ, passed X containing only those two identical observations, returns a classifier with a loss $> \tau_l$. Since $|X|$ is still not smaller than $\tau_{|X|}$ FIT would continue trying to separate the two inseperable observations. To prevent such a szenario τ_h tells FIT to stop before the Tree's height, the amount of edges of the longest path, would exceed τ_h. The path length of the Tree's root to Θ is passed as a parameter h to FIT.

Now, if neither τ_l is exceeded nor $\tau_{|X|}$ or τ_h is violated, FIT performs a split and transforms Θ to a Node (Algorithm 1, lines 7ff). The dimension the split is performed on is chosen in a cyclic manner, a practise also applied to k-d trees (Algorithm 1, line 7). [1] But rather than chosing the splitting value at the median of the observations in the dimension, which is done in order to construct balanced k-d trees, the splitting value is random.[1]

In order to chose a proper splitting value β_X is passed as another parameter to FIT. β_X represents the boundries for every dimension of the feature space based on X. For each dimension β_X contains a tuple with the minimum and maximum value in the dimension of all observations in X.

β_X[dimension] is passed to a pseudo-random number generator[2] generating a random value so that $lower(\beta_X[\text{dimension}]) \leq$ random number $\leq upper(\beta_X[\text{dimension}])$ (Algorithm 1, line 8).

Afterwards X, y, β_X are splitted into two new disjoint partitions and FIT is recursively applied to the two new partitions (Algorithm 1, lines 10ff).

Since τ_h is defined, the maximum amount of nodes a Tree can have is $2^{\tau_h+1} - 1$ if the Tree would be perfectly balanced. [6, chapter 16.1] For each node FIT is called, so building a Tree

[2]The implementation uses Python 3.6's random library.[7, chapter 9.6]

has a worst case time complexity of $\mathcal{O}((2^{\tau_h+1}-1)*\mathcal{O}(\text{FIT}))$. $\mathcal{O}(\text{FIT})$ is determined by the size of X, since X has to be splitted and by $\mathcal{O}(\gamma)$. That said, a single FIT operation would have a worst case time complexity of $\mathcal{O}(|X|+\mathcal{O}(\gamma))$, which would mean the time complexity of the whole fitting process would be $\mathcal{O}((2^{\tau_h+1}-1)*(|X|+\mathcal{O}(\gamma)))$.

Algorithm 1 : FIT($\Theta, X, y, h, \beta_X, \gamma, \tau_l, \tau_{|X|}, \tau_h$)

A Tree's FIT operation.
Inputs:
- Θ – a pointer to a Nil node; initially pointing to the root node of an empty Tree,
- X – input data,
- y – labels of X,
- h – height of the Tree; initially $h = 0$,
- β_X – lower and upper boundries of every dimension of X,
- γ – function returning a classifier and its loss,
- τ_l – loss threshold,
- $\tau_{|X|}$ – threshold for the size of X,
- τ_h – height limit of the Tree

Output: void

```
1: classifier, loss ← γ(X, y)
2: if loss ≤ τ_l then
3:     Θ ← LEAF(true, classifier, X, y)
4: else if h > τ_h or |X| < τ_|X| or loss > τ_l then
5:     Θ ← LEAF(false, classifier, X, y)
6: else
7:     dimension ← h mod |X[0]|
8:     split ← RANDOM(β_X[dimension])
9:     Θ ← NODE(split, NIL, NIL)
10:    split X, y and β_X into X', X'', y', y'', β'_X, β''_X
11:    FIT(Θ.left, X', y', h + 1, β'_X, ...)
12:    FIT(Θ.right, X'', y'', h + 1, β''_X, ...)
13: end if
```

B. The PREDICT operation

The PREDICT operation traverses a Tree until it encounters a Leaf. If the Leaf is active a label to a provided observation x is returned by the classifier property of the Leaf, otherwise Λ is returned (Algorithm 2).

Λ must not be an element of the label set.

Since the PCF does not predict on its own but instead uses other classifier instances for the actual prediction the PCF is a Meta Classifier.[4, chapter 4.6]

PREDICT is fairly similar to the search operation of a binary tree, except for the type distinction and the prediction.[3, chapter 12.2] Therefore, PREDICT has a worst case time complexity of $\mathcal{O}(\tau_h + \mathcal{O}(\text{classifier}))$.

Algorithm 2 : PREDICT(Θ, x, h)

A Tree's PREDICT operation.
Inputs:
- Θ – a Tree node; initially pointing to the root of the Tree,
- x – an observation,
- h – height of the Tree; initially $h = 0$

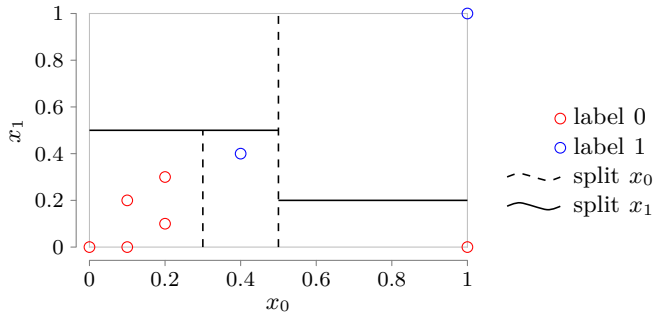
Output: the predicted label or Λ

```
1: if TYPE(Θ) is Node then
2:     dimension ← h mod |x|
3:     if x[dimension] ≤ Θ.split then
4:         PREDICT(Θ.left, x, h + 1)
5:     else
6:         PREDICT(Θ.right, x, h + 1)
7:     end if
8: else if Θ.active then
9:     return Θ.classifier(x)
10: else
11:     return Λ
12: end if
```

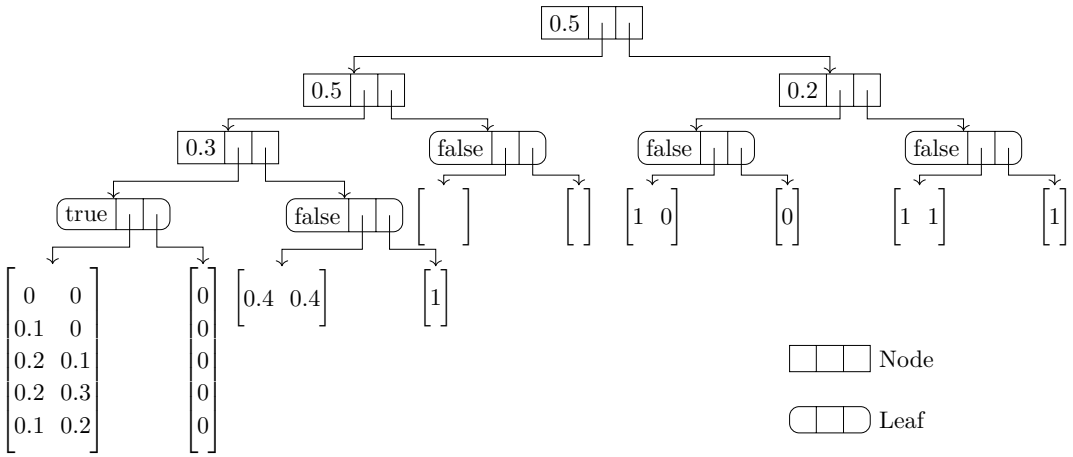
III. Partial Classification Forest

The PCF has two parameters, (i) N and (ii) an array with N pointers. N is the amount of Trees the PCF maintains. Initially the pointers inside the array are references to Nil nodes.

The PCF offers the same two operations a Tree has, FIT (Algorithm 3) and PREDICT (Algorithm 4), both abstractions to the equivalent Tree operations.



(a) Scatterplot showing the observations and the splits done by the FIT operation.



(b) The structure of the Tree generated by FIT.

Fig. 1: Example of FIT on a dataset seen in Figure 1a. γ simply computes the probability of each label in y and returns a function returning the label with the maximum probability and as loss 1 - the maximum probability. The thresholds are: $\tau_l = 1$, $\tau_{|X|} = 2$. τ_h can be any integer above 2.

Once FIT is executed, the pointers are references to the roots of fitted Tree instances.

Both FIT and PREDICT can be implemented as multithreaded operations as long as γ is threadsafe, since the Tree instances are independent of each other and the shared parameters X, y (FIT) and x (PREDICT) are read only, making synchronization unnecessary.

FIT first computes β_X which has a time complexity of $\mathcal{O}(|X[0]| * |X|)$, $|X[0]|$ denoting the amount of dimensions the feature space has.

After that the Tree's FIT operation is called N times, which means the PCF's FIT operation has a worst case time complexity of $\mathcal{O}(N * (2^{\tau_h+1} - 1) * (|X| + \mathcal{O}(\gamma)) + |X[0]| * |X|)$ (cmp. Section II-A).

The PCF's PREDICT operation first initializes an array with N elements (Algorithm 4, line 1). Each Tree instance fills one element of the array with its prediction. After that the PCF's PREDICT operation takes the label predicted most and returns it as its prediction for the observation x (Algorithm 4, lines 5, 6).

The worst case time complexity of the

Algorithm 3 : FIT($\Pi, X, y, \gamma, \tau_l, \tau_{|X|}, \tau_h$)

The PCF's FIT operation.
Inputs:
- Π – a PCF instance,
- X – input data,
- y – labels of X,
- γ – function returning a classifier and its loss,
- τ_l – loss threshold,
- $\tau_{|X|}$ – threshold for the size of X,
- τ_h – height limit of the Tree

Output: void

1: compute β_X
2: for all $\Theta \in \Pi$.trees do
3: FIT(Θ, X, y, 0, β_X, …)
4: end for

PCF's PREDICT operation is $\mathcal{O}(N * (\tau_h + \mathcal{O}(\text{classifier})) + N)$, since a Tree's PREDICT operation is executed N times, plus the most predicted label must be determined, which is $\mathcal{O}(N)$.

Algorithm 4 : PREDICT(Π, x)

The PCF's PREDICT operation.
Inputs:
- Π – a PCF instance,
- x – an observation,

Output: the predicted label or Λ

1: predictions $\leftarrow [\Lambda; N]$
2: for $i = 1$ to N do
3: predictions[i] = PREDICT(Π.trees[i], x, 0)
4: end for
5: determine l_{max}, the label predicted most
6: return l_{max}

IV. Tests

This Section will present two conducted tests. It should be noted here that, like described in the Introduction, these are not very comprehensive tests. This Section rather visualizes the use of the PCF predicting on partitions of otherwise unpredictable datasets.

The first test will visualize what the descision surface of the PCF looks like, the second will show how the PCF behaves with different amounts of Trees.

Both tests are performed on a randomly generated, normalized, two dimensional and binary labeled dataset. The dataset contains five thousand observations and was designed to be unpredictable, when predicted as a whole. The plane from which the observations are generated contains five partitions in which the observations all have the same label. Observations from those five partitions make up twenty percent of all observations and are the only ones which should be predicted, because all points not inside those partitions are labeled randomly. The optimal descision surface would be equal to the area of the five partitions.

All observations are generated by a pseudo-random number generator[7, chapter 9.6], therefore, every point on the plane has the same probability to be chosen as an observation for the dataset.

It is common practise in machine learning to split a dataset into a training and a test set in order to find the best model.[5, chapter 18] This approach is also used for the PCF. The training set is used as the parameters X and y of FIT (Algorithm 3) while PREDICT (Algorithm 4) is used on every observation of the test set.

Two metrics are used to describe the behaviour of the PCF, (i) *predicted* and (ii) *accuracy*. Both metrics are derived from comparing the label of an observation returned by PREDICT with its actual label from the dataset.

Predicted is the percentage of predicted observations, while *accuracy* is the percentage of correctly classified observartions from the test set.

For both tests γ, τ_l and τ_h are the same. γ and τ_l are equal to the values used in Figure 1 while $\tau_h = 32$. The dataset was splitted into a training and a test set such that ten percent of

the observations were used as the test set.[3] The observations were chosen randomly.

The first test will show the descision surface of the PCF with different N and $\tau_{|X|}$. For the test three different values, two, five and ten were used for both N and $\tau_{|X|}$ to show how those two parameters change the descision surface of the PCF. The test shows that for $\tau_{|X|} = 2$ the PCF overfitted the data, which means that *predicted* exceeded twenty percent, the amount of accurate observations while also failing to meet an *accuracy* equal to τ_l. This results in the chaotic descision surfaces shown in Figures ?? - ??. On the other hand for $\tau|X| = 10$ the descision surface is very small which means the PCF's *predicted* is less than twenty percent (Figures ?? - ??).

The second test shows the influence the amount of Trees N has on *predicted* and *accuracy*. In this test $\tau_{|X|}$ equals four.

Figure 3 shows that, for this dataset with the chosen thresholds, the PCF instances with $N < 30$ variate, both their *predicted* and *accuracy* values. Is $30 \leq N \leq 100$ *accuracy* is constant and equals τ_l. *Predicted* on the other hand, still rises in this interval. Is $N \geq 100$ *predicted* is higher than *predicted* with $N = 100$ and constant, but *accuracy* fails to meet τ_l, which means the PCF instances are overfitting.

Furthermore the second test relates *predicted* and *accuracy* of the observations from the test set to the values for the training set, showing that more training observations are predictable than test observations (cmp. Figure 3).

V. Further optimizations and additional features

There are some possible optimizations to the PCF which were not discussed in the previous Sections. Again it should be noted that these optimizations are yet untested (see Introduction).

The proposed optimizations are:

1) Weighing partitions.
2) Densifying a Tree after FIT.
3) Chosing the dimension a Node has randomly rather than cyclic (cmp. Section II-A).

The first optimization would be to weigh partitions. As of right now, the PCF's PREDICT operation determines l_{max} as the label predicted most (Algorithm 4, line 5).

This could be further refined with weighing each partition based on two properties: (i) the amount of observations a partition contains and (ii) its volume. This would result in a weight determined as:

$$\text{weight} = \frac{|\text{partition}.X|}{V(\text{partition})}. \tag{1}$$

A partition with a lot of observations and a small volume would have a higher weight than one with a small amount of observations and a high volume, further increasing the propability that the label predicted by the partition with the higher weight is determined as l_{max}. So instead of just counting each label predicted and returning the one predicted most, the weight of each prediction is summed and the label with the highest sum will be returned by PREDCIT as l_{max}.

The second optimization is densifying a Tree after FIT. If a Node has two unactive Leaves as children the Node is unnecessary since either way Λ is returned. The Node could be transformed into an unactive Leaf, reducing the path length of the branch by one and decreasing the amount of nodes by two, which decreases the size of the Tree and therefore the time complexity of PREDICT.

The last possible optimization would be to chose the splitting dimension, like the splitting value, at random rather than cyclic. A Node would have to store the dimension it is splitting.

In Section IV and in Figure 1 a very simple classifier is returned by γ. It only computes the probability for each label and returns the label with the highest probability (cmp. Figure 1).

A classifier like that does not need to know where the observations are in the partition which

[3]For splitting scikit-learn's model_selection.train_test_split was used during the tests.[2]

Fig. 2: The descision surfaces of differently configured PCFs. The threshold $\tau_{|X|}$ and the amount of Trees N of the PCF are different for each Figure, ?? - ??.

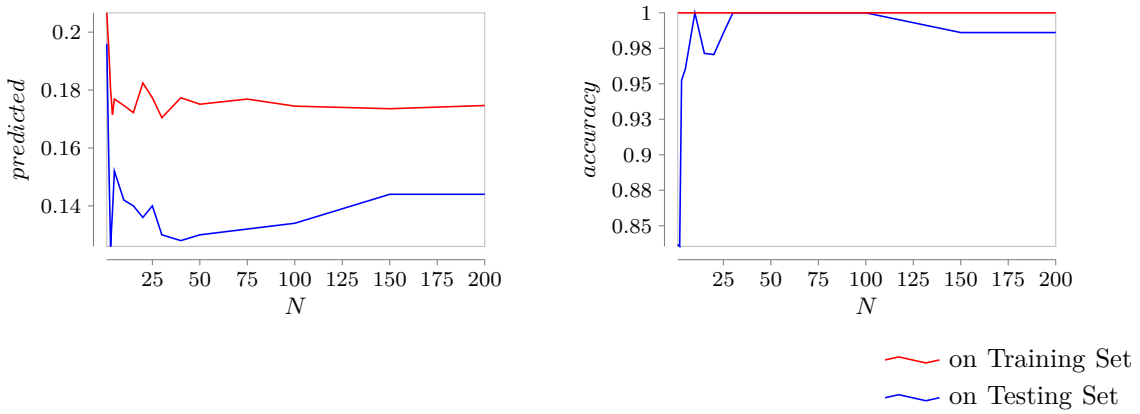


Fig. 3: The amount of Trees of the PCF N, in relation to *predicted* and *accuracy*.

makes it unnecessary to keep them in a Leaf node as X and y (cmp. Algorithm 1). Instead a dictionary with every label from the label space where each label is mapped to the amount of observations inside the partition having the particular label is enough.

A possible feature would be to provide a second variant of the PCF which passes this dictionary instead of the observations to γ, decreasing the complexity of the PCF for classifiers which do not need to know where each observations lays inside the partition.

VI. Conclusion

This paper provides an early Proof of Concept of the Partial Classification Forest. It describes the core structures and methods of the PCF. There is yet much to test, discover and first and foremost to implement.

References

[1] Brown, R. A. Building a balanced k-d tree in $o(kn \log n)$ time. Journal of Computer Graphics Techniques (JCGT) 4, 1 (March 2015), 50–68.

[2] Buitinck, L., Louppe, G., Blondel, M., Pedregosa, F., Mueller, A., Grisel, O., Niculae, V., Prettenhofer, P., Gramfort, A., Grobler, J., Layton, R., VanderPlas, J., Joly, A., Holt, B., and Varoquaux, G. API design for machine learning software: experiences from the scikit-learn project. In ECML PKDD Workshop: Languages for Data Mining and Machine Learning (2013).

[3] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. Introduction to Algorithms, Third Edition, 3rd ed. The MIT Press, 2009.

[4] Hanke, M., Halchenko, Y. O., and Oosterhof, N. N. PyMVPA Manual, 2.6.1st ed.

PyMVPA.org, 2017.

[5] Russel, S., and Norvig, P. Künstliche Intelligenz: Ein moderner Ansatz, 3 ed. Pearson, Higher Education, München, 2012.

[6] Teschl, G., and Teschl, S. Mathematik für Informatiker, Band 1, 4th ed. Springer Spektrum, 2013.

[7] van Rossum, G., and the Python development team. The Python Library Reference, 3.6th ed. Python Software Foundation, 2019.