University of Bucharest
Faculty of Mathematics and Computer Science
Department of Computer Science

Andrei Dumitriu
MSC in Computer Science
April 2020

Advanced Machine Learning

Assignment 2

1. a.

$\tau_H(m) = 2m,\ \forall\ m \geq 1$
$\tau_H(m) = 1,\ m\ =\ 0$
Proof:
VCdim = 2 (see 1st homework for justification) [1]

For $m\ =\ 0,\ we\ have\ C\ =\ \{\oslash\}\ => \tau_H(m) = 1$
For
$m\ =\ 1,\ we\ have\ C\ =\ \{c_1 |\ c_1\ \in\ R\ \}\ => H\ =\ \{\{0\},\ \{1\}\},\ |H_a|\ =\ 2^{|C|} = 2^1\ => \tau_H(m = 1) = 2$
For $m\ =\ 2,\ we\ have$
$C\ =\ \{c_1,\ c_2 |\ c_1 \leq c_2\}\ => H\ =\ \{\{0, 0\},\ \{0, 1\},\ \{1, 0\},\ \{1, 1\}\},\ |H_a|\ =\ 2^{|C|} = 2^2\ => \tau_H(m = 2) = 4$
For $m\ =\ 3,\ we\ have$
$C\ =\ \{c_1,\ c_2,\ c_3 |\ c_1 \leq c_2 \leq c_3\}\ => H\ =\ \{\{0, 0, 0\},\ \{0, 0, 1\},\ \{0, 1, 1\}, \{1, 0, 0\}, \{1, 1,\ 0\},\ \{1, 1, 1\}\}, \tau_H(m = 3) = 6$
There is no function to realize the vectors $\{1,\ 0,\ 1\}\ and\ \{0,\ 1,\ 0\}$. From these, we see a pattern for $\tau_H(m) = 2m$. H can label according to the following pattern:
000…000 => #1
100...000 => #2
110...000 => #3
…
111...100
111...110
111...111 => #m
011...111 => #m+1
001...111 => #m + 2
…
000...011 => #2m -1
000...001 => #2m
For m = 4 the vectors are:
0000
1000
1100

1110
1111
0111
0011
0001
$H = \{\{0,0,0,0\},\ \{0,0,0,1\},\ \{0,0,1,1\},\ \{0,1,1,1\},\ \{1,1,1,1\},\ \{1,1,1,0\},\ \{1,1,0,0\},\ \{1,0,0,0\}\}$
$=> \tau_H(m = 4) = 8$

We cannot have intermediate, “sandwiched”, values, such as 101 or 010 (even as the length increases)
Thus, all the possible combinations amount to **2m**.

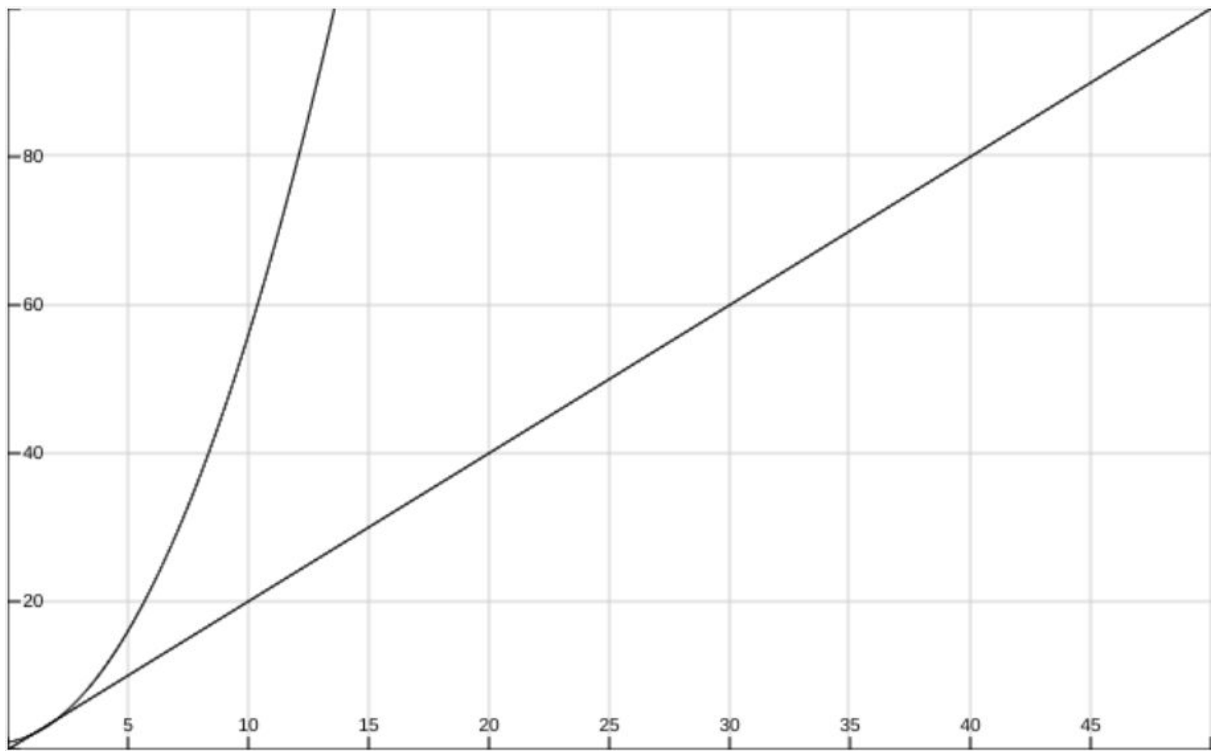
1. b.

General upper bound because VCdim = 2 [1]

$$\tau_H(m) \le \sum_{i=0}^{d} C_m^i = \frac{m!}{0!(m-0)!} + \frac{m!}{1!(m-1)!} + \frac{m!}{2!(m-2)!} = 1 + m + \frac{m(m-1)}{2}$$

Comparing $\tau_{H1}(m) = 1 + m + \frac{m(m-1)}{2}$ and $\tau_{H2}(m) = 2 * m$ we can see that $\tau_{H1}(m)$ grows exponentially while $\tau_{H2}(m)$ grows polynomially. The functions are equal in the points
$m = 0 => \tau_{H1} = \tau_{H2} = 1$

$m = 1 => \tau_{H1}(m) = \tau_{H2}(m) = 2$
$m = 2 => \tau_{H1}(m) = \tau_{H2}(m) = 4$



$Fig\ 1.1\ \tau_{H1(m)} vs\ \tau_{H2(m)}\ (\tau_{H1(m)}\ has\ exponential\ growth,\ \tau_{H2(m)}\ has\ polynomial\ growth)$

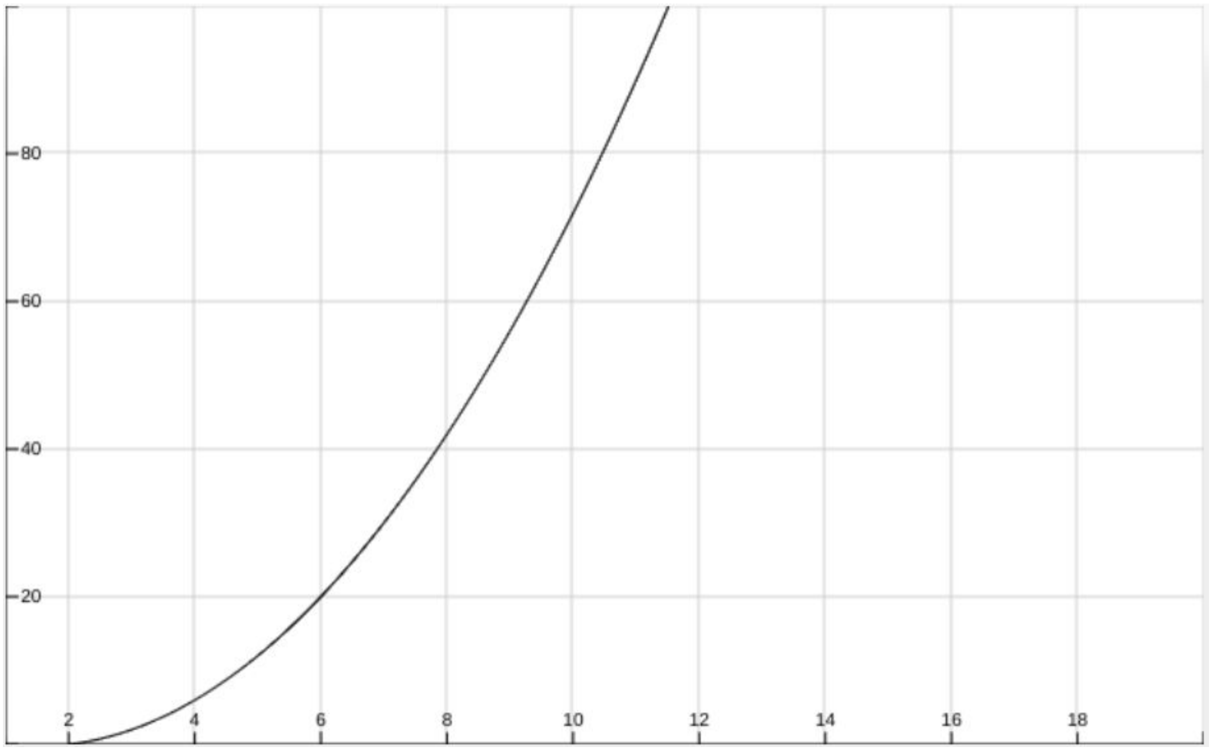
For $m \geq 1 => \tau_{H1} - \tau_{H2} = 1 + m + \frac{m(m-1)}{2} - 2m = 1 + \frac{m(m-1)}{2} - m = m^2 - 3m + 2$

We can see that for

$m = 1 => \tau_{H1} - \tau_{H2} = 0$

$m = 2 => \tau_{H1} - \tau_{H2} = 0$

$m \geq 3 => \tau_{H1} - \tau_{H2} > 0$



$Fig\ 1.2\ \tau_{H1}(m) - \tau_{H2}(m) = m^2 - 3m + 2,\ \forall\ m \geq 1,\ m \in R\ grows\ exponentially$

We conclude that τ_{H2} (from 1.a.) is always bounded by the general upper bound. When for $m \in \{0, 1, 2\}$ it has the same value with the general upper bound, and starting from $m \geq 3$ the difference grows exponentially.

1. c.

A possible H for which $\tau_H(m)$ is equal to the general upper bound over R and another domain X is $H_{thresholds}$

$H_{thresholds}$ is the set of threshold functions over the real line

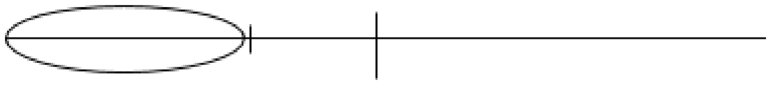
$H_{thresholds} = \{h_a -> \{0, 1\},\ h_a(x) = 1_{[x<a]},\ a \in R\},\ |H_{thresholds}| = infinite$

Vcdim($H_{thresholds}$)

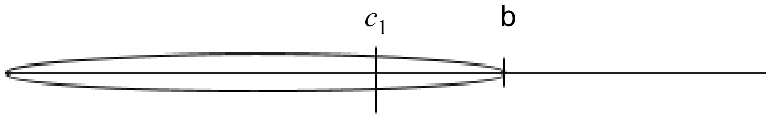
Consider C = {c_1 | c_1 ∈ R }. Then $H_C = \{h : C -> \{0, 1\} \mid h \in H\}$ has at most two elements $\{h_a, h_b\},\ a \leq c_1 and\ b \geq c_2$ so H shatters C. $H_c = \{(0), (1)\},\ |H_c| = 2^{|C|} = 2^1$

For $h_a\ s.t.\ a < c_1$, h_a generates the labeling (0)

a c_1



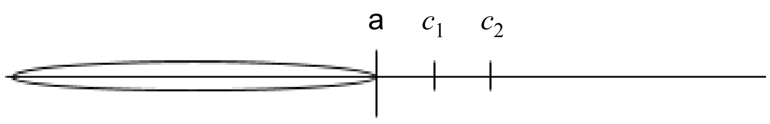
For h_b $s.t.$ $b \geq c_1$, h_a generates the labeling (1)



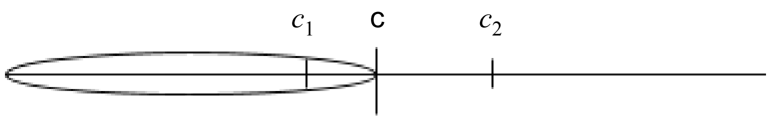
Thus, we have VCdim(H) $\geq$ 1

Consider C = $\{c_1, c_2 \mid c_1 \leq c_2\}$ Then $H_C = \{h : C \rightarrow \{0, 1\} \mid h \in H\}$ has at most three elements. There is no function that realizes the labeling $(0, 1)$ and so H does not shatter C.

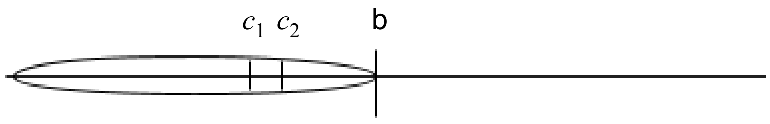
For h_a $s.t.$ $a < c_1 \leq c_2$, h_a generates the labeling (0, 0)



For h_c $s.t.$ $c_2 > c > c_1$, h_a generates the labeling (1, 0)



For h_b $s.t.$ $c_2 \geq c_1 < b$, h_a generates the labeling (1, 1)

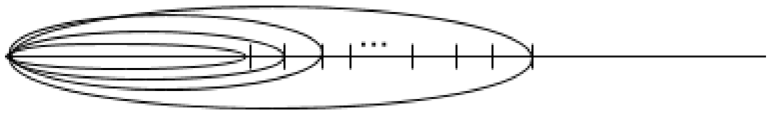


There is no function h_d in H_C $s.t.$ that realizes the vector $(0, 1)$
Thus, we have VCdim(H) $=$ 1
Now we compute the shattering coefficient

$$\tau_H(m) \leq \sum_{i=0}^{d} C_m^i = \frac{m!}{0!(m-0)!} + \frac{m!}{1!(m-1)!} = 1 + m$$

$$\tau_H(m = 1) \leq \sum_{i=0}^{d} C_m^i = C_1^0 + C_1^1 = 1 + 1 = 2$$

For C = $\{c_i \mid c_i \leq c_{i+1},\ i \in N\}$, $|C| = m$, we can label them in $m + 1$ ways: 1st label is (0,...,0) and we follow with (1, 0, 0,…,0), (1, 1, 0,…,0), (1, 1, 1,…,1), adding each point, resulting in $1 + m$ as the upper bound (no points + m steps, with 1 point added at each step).



2.a.

We have the formula $VCdim(H) \ \leq log_2|H|$

Our task is to find $|H|$ in terms of $\Sigma \ and \ m$

$|H| \ = \ \sum_{i=1}^{m} |\Sigma^i|, \ |\Sigma| = m => |H| = \sum_{i=1}^{m} m^i$

Therefore, an upper bound for $VCdim(H) \ \leq log_2|H| => \ VCdim(H) \ \leq log_2 \sum_{i=1}^{m} |\Sigma^i| = log_2 \sum_{i=1}^{m} m^i$

For example, for $m = 3,$ we have $log_2 \sum_{i=1}^{3} 3^i = log_2(3 + 9 + 27) \ = \ log_2(39)$

2.b.

The most direct way to do this (I propose a better way later on):

1. Split the first tuple with “1” into all possible substrings, do a hashtable (key is each substring and value is 1)
2. Go through the entire lists with tuples (excluding the 1st one) and do the following:
    a. Check, for each key in the initial hashtable, if it is present in the current tuple.
    b. If the tuple is 0 and the key is present, remove it from the initial hashtable.
    c. If the tuple is 1 and the key is not present, remove the key from the initial hashtable
    d. If the tuple is 1 and the key is present, do nothing
3. At the end, we are left only with the keys(substrings) that are present in each tuple with 1, but not present in each tuple with 0.

This has some inefficiencies:

1. You continuously check for all combinations. For example, if you have the substrings “a”, “ab”, “abc”, it is enough to realize that if “a” is not present, neither “ab” and “abc” cannot be present
2. It yields a $O(n^3)$ time complexity

I propose the following improvement:

1. Keep the initial hashtable in a TRIE data structure
2. Check the tuples with “1” first, as these are easier to remove (if “a” is not present, “abcd” is not present as well, so you can remove all the children of “a”. Removing the ones that are not present in 1 reduces the substrings we need to check for)

The steps are as follow:

1. Sort the entire vector of tuples in order to have the first ones with “1”
2. Take the first tuple element and split it into substrings. Add the substrings in a TRIE data structure. If there are no entries with (1), then we create a TRIE of the entire alphabet combinations
3. For all the tuples with 1, check for the lowest substrings first, doing a breadth-first searching in our initial trie hashtable. When a substring is not found, delete it **along with all its children**. This is because if your initial table tree has “a” -> “ab” -> “abc”,

if we do not find **a**, we know for sure that “ab” and “abc” will not be found as well. If there are no tuples with 1, this step is skipped.

4. For all tuples with 0, check if the current substring is present in the string of the tuple. If it is **delete only this node**, making its children point to the father of the current node. This is because if you our initial table tree has “a” -> “ab” -> “abc”, if we find “a” in a tree with 0, although we delete “a”, we can still return “ab” and “abc” as these combinations have not been found. **It is important to understand the difference in parsing between tuples with 1 and 0, as this is the main time complexity saving.**
5. Return tuples that are left, if any.

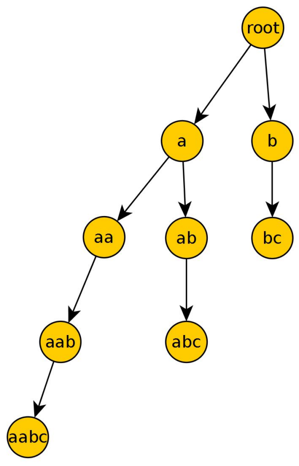
Important information:

1. It is enough to only split the first tuple with 1 in all substrings because a returned substring must be present **in all** substrings with 1. Therefore, any substring that is not present in the first tuple with 1 will not be returned. As we progress, we only check if the keys of our trie tree are present in the tuple string. We do not split the tuple string into substrings and we do not compare it with all substrings.
2. It is important to understand the difference when removing substrings **not found** in a tuple with 1 and removing substrings **found** in a tuple with 0. **This yields a considerable improvement in time complexity.**

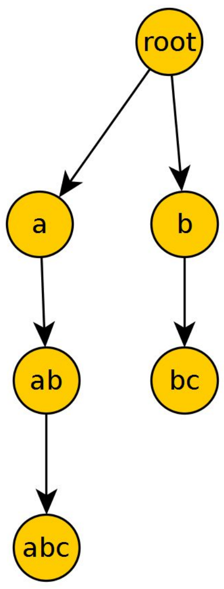
```
Pseudo-pseudocode
# breadth depth search with a tweak. It removes the ENTIRE SUBTREE of
matching elements
procedure bfs_for_s1(visited, graph, node, string):
   visited.append(node)
   queue.append(node)

   while queue:
       s = queue.pop(0)
       print (s, end = " ")

       for neighbour in graph[s]:
           if neighbour not in string:
               del trie[neighbour] # alternatively, trie[neighbour] =
{} # delete the entire subtree
       if neighbour not in visited:
           visited.append(neighbour)
           queue.append(neighbour)

# breadth depth search with a tweak. It removes ONLY THE MATCHING NODE
procedure bfs_for_s0(visited, graph, node, string):
   visited.append(node)
   queue.append(node)

   while queue:
       s = queue.pop(0)
       print (s, end = " ")

       for neighbour in graph[s]:
           if neighbour in string:
               tree[node].append(trie[neighbour]) # delete only this
node and make its children point to the parent
               del trie[neighbour] # alternatively, trie[neighbour] =
{}
       if neighbour not in visited:
           visited.append(neighbour)
           queue.append(neighbour)

procedure find_hypothesys(S,m): # S = {(aabc, 1), (baca, 0), (bcac, 0),
(abba, 1)}.
   S0 = []
   S1 = []

   for tpl in S: # split the tuples into 2 separate arrays, those with
1 and those with 0
       if tpl[1] == 0:
           S0.append(tpl[0])
       if tlp[1] == 1:
           S1.append(tpl[0]) # we only add the string as we know what
values they have
   if len(S1) == 0:
       trie = create_trie_of_the_entire_alphabet() # all the possible
combinations of the words
   else:
       trie = construct_trie(S1[0]) # return longest substring
   # trie is a hashtable of hashtables. For example, a -> ab -> abc is
trie['a'] = { "ab": {"abc":{}}
   # kills the memory, nails the time tho'

   # parse the array separately and apply a different strategy for them
   # parse first all the substrings with (1) value. Remove from trie
all the substrings that are NOT also in all the tuples
   for elem in s1[1:]: # we took the first one and made the trie out of
it
       for key, value in trie:
           bfs_for_s1(visited, trie, key, string)

   visited = []
   queue = []
   # parse all the arrays
   # remove from trie all the substrings that ARE IN ANY of the tuples
substrings
   for elem in s0:
       substrings = get_all_substrings(elem) # substrings is a
hashtable = {}
       for key, value in trie:
           bfs_for_s1(visited, trie, key, substrings)


return trie.any
```

Example how it goes for input S = {(aabc, 1), (baca, 0), (bcac, 0), (abba, 1)}

1. Split S into S1 and S0, resulting S0 = {baca, bcac} and S1 = {aabc, abba}
2. Built TRIE out of S1[1]

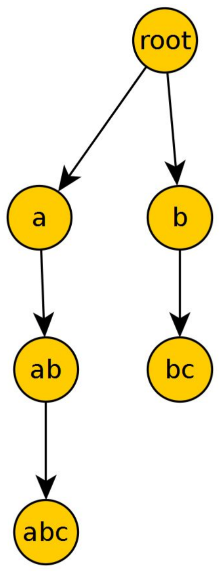


3. Check next in S1. For this example, we only have S[1] = "abba"
    a. "a" is found
    b. "b" is found
    c. "aa" is not found. We remove the entire subtree of "aa", including it. First example of the efficiency in action, removing 3 substrings(nodes) in O(1)

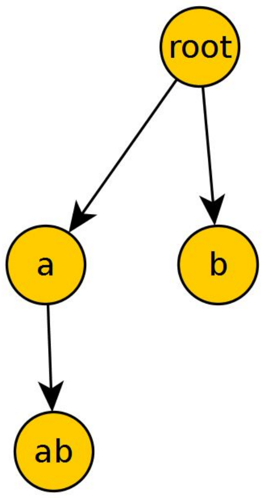


d. "ab" is found

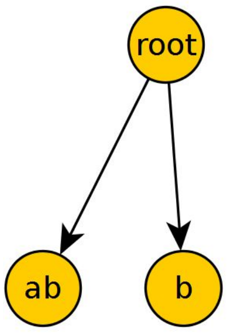
e. "bc" not found, we remove it and the entire subtree (in this case only this node)



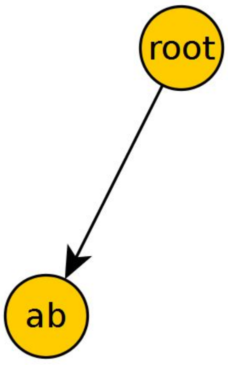
f. "abc" not found, we remove it and the entire subtree (in this case only this node)



4. Check the values present in S0 = {baca,bcac}. We remove the found ones (only the nodes)
    a. We check S0[0] = “baca”
        i. check if “a” is present. It is, so we remove the node and make its children point to the parent of “a” (“ab” in this case)



        ii. check if “b” is present. It is, so we remove the node and make its children point to the parent of “b” (none in this case)



        iii. check if “ab” is present in the tuple string. It is not, so we move on. Again, here is the advantage of not splitting the tuple string into substrings. If we did so and checked for all of them, we would have (m^2 substrings to check against. This way, we only have 1 substring to check against, which takes $m\,k$ time (k = 2 length of the substring) instead of $m^2 + mk$
    b. We check S0[1] = “bcac”

    i. Check if "ab" is present in the tuple string. It is not, and since it's the only node left, we are done.

5. Return $h_{trie.any}$, in our case h_{ab}

Complexity of our algorithm:

1. Split the first tuple with 1 into substrings: m^2
2. For each element, check if it is present in the string. Worst case: $(m^2 \ * \ mk)$
3. Worst case complexity: $(m^2 \ + \ m^2 \ * \ mk) = (m^3k), \ k \ = \ length\ of\ substring$
4. However, we can see that on average this will yield $(m^2 logm)$, as we check for the lowest substrings first (doing a breadth-depth search) and eliminating all its children.

Addendum: with minutes left to spare, I realized that I missed an important corner case. More specifically, when the entire input is 0. I rapidly modified the code to address this case, but it switched the worst-worst case to $O(2^n)$. However, I know how to fix this, but don't have the time to implement it. Basically, for this case, now I created the TRIE for the entire alphabet and remove the nodes that are present, returning at the end any that is left. However, a better case would be to create a trie out of all the substrings that we parse, with 0, keeping the depth of each branch and width of each level. After that, we know the depth that each branch should have and the width of each level. We check the level or the depth that is not equal to the maximum depth / maximum width of that specific branch / level and we return that in $O(n)$.

3.a.

We have 2 cases

1. When x_i is correctly classified and $h_t(x_i) \ = \ y_i$
    a. $h_t(x_i) \ = \ y_i = -1$
    b. $h_t(x_i) \ = \ y_i = 1$

Note that in both subcases, multiplying $h_t(x_i)\ and\ y_i$ yields the same sign (1)

2. When x_i is incorrectly classified and $h_t(x_i) \ \neq \ y_i$
    a. $h_t(x_i) \ = -1 \ \Rightarrow \ y_i = 1$
    b. $h_t(x_i) \ = 1 \ \Rightarrow \ y_i = -1$

Note that in both subcases, multiplying $h_t(x_i)\ and\ y_i$ yields the same sign (-1)

Scenario 1. The formula equivalence we have to prove is:

$$\frac{D^{(t)}(i) * \sqrt{\frac{1-\varepsilon_t}{\varepsilon_t}}}{Z_{t+1}} = \frac{w_{t,i} * \beta_t^{1-e_i}}{\sum_{j=1}^{n} w_{t+1,j}} \Leftrightarrow \frac{D^{(t)}(i) * \sqrt{\frac{1-\varepsilon_t}{\varepsilon_t}}}{\sum_{i=1}^{n} D^{(t)}(i) * e^{-w_t h_t(x_i) y_i}} = \frac{w_{t,i} * \beta_t^{1-e_i}}{\sum_{j=1}^{n} w_{t+1,j}}$$

Basically, in this case, $-w_t h_t(x_i) y_i = -w_t$

$$\frac{D^{(t)}(i)*\sqrt{\frac{1-\varepsilon_t}{\varepsilon_t}}}{\sum_{i=1}^{n} D^{(t)}(i)*e^{-w_t h_t(x_i) y_i}} = \frac{w_{t,i}*\beta_t^{1-e_i}}{\sum_{j=1}^{n} w_{t+1,j}} \Leftrightarrow \frac{D^{(t)}(i)*\sqrt{\frac{1-\varepsilon_t}{\varepsilon_t}}}{\sum_{\substack{i=1\\ h_t(x_i)=y_i}}^{n} D^{(t)}(i)*\sqrt{\frac{\varepsilon_t}{1-\varepsilon_t}}+\sum_{\substack{i=1\\ h_t(x_i)\neq y_i}}^{n} D^{(t)}(i)*\sqrt{\frac{1-\varepsilon_t}{\varepsilon_t}}} = \frac{w_{t,i}*\beta_t^{1-e_i}}{\sum_{j=1}^{n} w_{t+1,j}} \Leftrightarrow$$

$$\Leftrightarrow \frac{D^{(t)}(i)}{\sum_{\substack{i=1\\ h_t(x_i)=y_i}}^{n} D^{(t)}(i)*\frac{\varepsilon_t}{1-\varepsilon_t}+\sum_{\substack{i=1\\ h_t(x_i)\neq y_i}}^{n} D^{(t)}(i)} = \frac{w_{t,i}*\beta_t^{1-e_i}}{\sum_{j=1}^{n} w_{t+1,j}} \Leftrightarrow$$

$$\Leftrightarrow \frac{D^{(t)}(i)}{\sum_{\substack{i=1\\ h_t(x_i)=y_i}}^{n} D^{(t)}(i)*\frac{\varepsilon_t}{1-\varepsilon_t}+\sum_{\substack{i=1\\ h_t(x_i)\neq y_i}}^{n} D^{(t)}(i)} = \frac{w_{t,i}*\beta_t^{1-e_i}}{\sum_{\substack{j=1\\ h_t(x_j)\neq y_j}}^{n} w_{t,j}+\sum_{\substack{j=1\\ h_t(x_j)=y_j}}^{n} w_{t,j}\beta_t} \Leftrightarrow \quad // \# \beta_t = \frac{\varepsilon_t}{1-\varepsilon_t} and\ e_i = 1$$

$$\Leftrightarrow \frac{D^{(t)}(i)}{\sum_{\substack{i=1\\ h_t(x_i)=y_i}}^{n} D^{(t)}(i)*\frac{\varepsilon_t}{1-\varepsilon_t}+\sum_{\substack{i=1\\ h_t(x_i)\neq y_i}}^{n} D^{(t)}(i)} = \frac{w_{t,i}}{\sum_{\substack{j=1\\ h_t(x_j)\neq y_j}}^{n} w_{t,j}+\sum_{\substack{j=1\\ h_t(x_j)=y_j}}^{n} w_{t,j}\frac{\varepsilon_t}{1-\varepsilon_t}} \Leftrightarrow$$

$$\Rightarrow D^{(t)}(i) = w_{t,i}$$

Scenario 2. The formula equivalence we have to prove is:

$$\frac{D^{(t)}(i)*\sqrt{\frac{1-\varepsilon_t}{\varepsilon_t}}}{Z_{t+1}} = \frac{w_{t,i}*\beta_t^{1-e_i}}{\sum_{j=1}^{n} w_{t+1,j}} \Leftrightarrow \frac{D^{(t)}(i)*\sqrt{\frac{1-\varepsilon_t}{\varepsilon_t}}}{\sum_{i=1}^{n} D^{(t)}(i)*e^{-w_t h_t(x_i) y_i}} = \frac{w_{t,i}*\beta_t^{1-e_i}}{\sum_{j=1}^{n} w_{t+1,j}}$$

Basically, in this case, $-w_t h_t(x_i) y_i = w_t$

$$\frac{D^{(t)}(i)*\sqrt{\frac{\varepsilon_t}{1-\varepsilon_t}}}{\sum_{i=1}^{n} D^{(t)}(i)*e^{-w_t h_t(x_i) y_i}} = \frac{w_{t,i}*\beta_t^{1-e_i}}{\sum_{j=1}^{n} w_{t+1,j}} \Leftrightarrow \frac{D^{(t)}(i)*\sqrt{\frac{\varepsilon_t}{1-\varepsilon_t}}}{\sum_{\substack{i=1\\ h_t(x_i)=y_i}}^{n} D^{(t)}(i)*\sqrt{\frac{\varepsilon_t}{1-\varepsilon_t}}+\sum_{\substack{i=1\\ h_t(x_i)\neq y_i}}^{n} D^{(t)}(i)*\sqrt{\frac{1-\varepsilon_t}{\varepsilon_t}}} == \frac{w_{t,i}*\beta_t^{1-e_i}}{\sum_{j=1}^{n} w_{t+1,j}}$$

$$\Leftrightarrow \frac{D^{(t)}(i)*\frac{\varepsilon_t}{1-\varepsilon_t}}{\sum_{\substack{i=1\\ h_t(x_i)=y_i}}^{n} D^{(t)}(i)*\frac{\varepsilon_t}{1-\varepsilon_t}+\sum_{\substack{i=1\\ h_t(x_i)\neq y_i}}^{n} D^{(t)}(i)} = \frac{w_{t,i}*\beta_t^{1-e_i}}{\sum_{j=1}^{n} w_{t+1,j}} \Leftrightarrow \# \beta_t = \frac{\varepsilon_t}{1-\varepsilon_t} and\ e_i = 0$$

$$\Leftrightarrow \frac{D^{(t)}(i)*\frac{\varepsilon_t}{1-\varepsilon_t}}{\sum_{\substack{i=1\\ h_t(x_i)=y_i}}^{n} D^{(t)}(i)*\frac{\varepsilon_t}{1-\varepsilon_t}+\sum_{\substack{i=1\\ h_t(x_i)\neq y_i}}^{n} D^{(t)}(i)} = \frac{w_{t,i}*\frac{\varepsilon_t}{1-\varepsilon_t}}{\sum_{\substack{j=1\\ h_t(x_j)\neq y_j}}^{n} w_{t,j}+\sum_{\substack{j=1\\ h_t(x_j)=y_j}}^{n} w_{t,j}\frac{\varepsilon_t}{1-\varepsilon_t}}$$

$$\Rightarrow D^{(t)}(i) = w_{t,i}$$

We have proved that $D^{(t)}(i) = w_{t,i}$ in both cases.

3.b

We will write the Adaboost strong classifier with h^A and the course final classifier with h^C

$$h^A(x) = 1,\ if\ \sum_{t=1}^{n} \alpha_t h_t^A(x) \geq \tfrac{1}{2}\sum_{t=1}^{T}\alpha_t$$. We know that $\alpha_t = ln\frac{1}{\beta_t}$ and $\beta_t = \frac{\varepsilon_t}{1-\varepsilon_t}$

$$0,\ otherwise$$

resulting that

$$h^A(x) = 1,\ if\ \sum_{t=1}^{n} ln(\tfrac{1-\varepsilon_t}{\varepsilon_t})h_t^A(x) \geq \tfrac{1}{2}\sum_{t=1}^{T} ln(\tfrac{1-\varepsilon_t}{\varepsilon_t})$$

$$0,\ otherwise$$

We have

$h^C(x) = sign(\sum_{t=1}^{T} w_t h_t^A(x))$. We know that $w_t = \frac{1}{2}ln(\frac{1-\varepsilon_t}{\varepsilon_t})$

resulting that

$h^C(x) = sign(\sum_{t=1}^{T} \frac{1}{2}ln(\frac{1-\varepsilon_t}{\varepsilon_t})h_t^C(x))$. We know that $w_t = \frac{1}{2}ln(\frac{1-\varepsilon_t}{\varepsilon_t})$

So the equivalence is

$$\sum_{t=1}^{n} ln(\tfrac{1-\varepsilon_t}{\varepsilon_t})h_t^A(x) \geq \tfrac{1}{2}\sum_{t=1}^{T} ln(\tfrac{1-\varepsilon_t}{\varepsilon_t}) \Leftrightarrow sign(\sum_{t=1}^{T}\tfrac{1}{2}ln(\tfrac{1-\varepsilon_t}{\varepsilon_t})h_t^C(x)) = 1$$

Let’s get ‘em calculations rolling!

So

$$\sum_{t=1}^{n} ln(\tfrac{1-\varepsilon_t}{\varepsilon_t})h_t^A(x) \geq \tfrac{1}{2}\sum_{t=1}^{T} ln(\tfrac{1-\varepsilon_t}{\varepsilon_t}) \Leftrightarrow sign(\sum_{t=1}^{T}\tfrac{1}{2}ln(\tfrac{1-\varepsilon_t}{\varepsilon_t})h_t^C(x)) = 1$$

$$\sum_{t=1}^{n} ln(\tfrac{1-\varepsilon_t}{\varepsilon_t})h_t^A(x) - \tfrac{1}{2}\sum_{t=1}^{T} ln(\tfrac{1-\varepsilon_t}{\varepsilon_t}) \geq 0 \Leftrightarrow sign(\sum_{t=1}^{T}\tfrac{1}{2}ln(\tfrac{1-\varepsilon_t}{\varepsilon_t})h_t^C(x)) = 1$$

$$\sum_{t=1}^{n} ln(\tfrac{1-\varepsilon_t}{\varepsilon_t})(h_t^A(x) - \tfrac{1}{2}) \geq 0 \Leftrightarrow sign(\sum_{t=1}^{T}\tfrac{1}{2}ln(\tfrac{1-\varepsilon_t}{\varepsilon_t})h_t^C(x)) = 1$$

$$sign(\sum_{t=1}^{n} ln(\tfrac{1-\varepsilon_t}{\varepsilon_t})(h_t^A(x) - \tfrac{1}{2})) = 1 \Leftrightarrow sign(\sum_{t=1}^{T}\tfrac{1}{2}ln(\tfrac{1-\varepsilon_t}{\varepsilon_t})h_t^C(x)) = 1$$

$$sign(\sum_{t=1}^{n} (h_t^A(x) - \tfrac{1}{2})) = 1 \Leftrightarrow sign(\sum_{t=1}^{T} h_t^C(x)) = 1$$

$\Leftrightarrow$

$$h_t^A(x) - \tfrac{1}{2} > 0 \Leftrightarrow h_t^C(x) > 0$$

$$h_t^A(x) = 1 \Leftrightarrow h_t^C(x) = 1$$

At point **3a** we proved that each step of the algorithms’ distribution for each step is equivalent $(w_{t+1}\ and\ D^{t+1})$.

$h_t^A(x) = 0 \Leftrightarrow h_t^C(x) = -1\ and\ h_t^A(x) = 1 \Leftrightarrow h_t^C(x) = 1$

3.c.

We will assume that for iteration $t+1$ we have chosen the h_t classifier again

$$\varepsilon_{t+1}(h_t) = \sum_{i=1}^{n} w_{t+1,i} * |h_t(x_i) - y_i| = \sum_{i=1}^{n} \frac{w_{t,i} * \left(\frac{\varepsilon_t}{1-\varepsilon_t}\right)^{1-e_i}}{\sum_{j=1}^{n} w_{t,j}\left(\frac{\varepsilon_t}{1-\varepsilon_t}\right)^{1-e_j}} * |h_t(x_i) - y_i| = \frac{h_t(x_i) \neq y_i}{\sum_{j=1}^{n} w_{t,j} + \sum_{j=1}^{n} w_{t,j} * \frac{\varepsilon_t}{1-\varepsilon_t}} = \frac{\varepsilon_t}{\varepsilon_t + \frac{\varepsilon_t}{1-\varepsilon_t} \sum_{j=1}^{n} w_{t,j}}$$

$$\sum_{j=1}^{n} w_{t,j} = 1\,,\ \varepsilon_t = \sum_{i=1}^{n} w_{t,i} = \sum_{i=1}^{n} w_{t,i}(for\ h_t(x_i) = y_i + \sum_{i=1}^{n} w_{t,i}(for\ h_t(x_i) \neq y_i$$

$$\Rightarrow \varepsilon_{t+1} = \frac{\varepsilon_t}{\varepsilon_t + \frac{\varepsilon_t}{1-\varepsilon_t}(1-\varepsilon_t)} \Rightarrow \varepsilon_{t+1} = \frac{\varepsilon_t}{\varepsilon_t + \varepsilon_t} = \frac{1}{2}$$

We have reached a **contradiction.** For $\forall t - weak\ learner,\ \varepsilon_t < \frac{1}{2}$. Therefore, the probability that the classifier h_t (selected as the best weak learner at iteration t) will be selected again at iteration $t+1$ is 0.

References

[1] Andrei Dumitriu, Adjunct Professor at University of Bucharest, Advanced Machine Learning Assignment 1
[2] Matematică Online, Formule Algebră