# Machine learning 2 - Final Assignment

Frans de Boer
fransdeboer
5661439

July 1, 2022

The git repository for this assignment with the code and the report can be found at `https://github.com/Frans-db/ml2.final-assignment`. I will make this repository public after the Brightspace deadline has passed.

## 1 PAC Learning

### 1.a Extend the proof that was given in the slides for the PAC-learnability of hyper-rectangles: show that axis-aligned hyper-rectangles in n-dimensional feature spaces (n > 2) are PAC learnable.

We can extend the proof by taking the number of sides/faces of the hyper-rectangle into account.

- A hyper-rectangle in 2-dimensional space is simply a rectangle and has 4 sides.
- A hyper-rectangle in 3-dimensional space is a rectangular box and has 6 sides.

I will assume the number of sides on a n-dimensional hyper-rectangle to be $F(n)$.[1]

To start we can follow the steps from the lecture. We have the ground truth n-dimensional hyper-rectangle $R$, and the current tightest fit rectangle $R'$. The error $(R - R')$ can be split into $F(n)$ strips $T'$. On each of these strips we can now grow a new strip $T$ such that the probability mass of $T$ is $\frac{\varepsilon}{F(n)}$.

If $T$ covers $T'$ for all strips then the probability of an error is

$$P[error] \leq \sum_{i=0}^{F(n)-1} P[T_i] = F(n)\frac{\varepsilon}{F(n)} = \varepsilon \tag{1}$$

We can now estimate the probability that $T$ does not cover $T'$

[1] $F(n) = 2n$, `https://en.wikipedia.org/wiki/Hypercube#Faces`

$$P[\textit{random x hits T}] = \frac{\varepsilon}{F(n)}$$
$$P[\textit{random x misses T}] = 1 - \frac{\varepsilon}{F(n)}$$
$$P[\textit{m random x's misses T}] = (1 - \frac{\varepsilon}{F(n)})^m$$

Since we have $F(n)$ strips:

$$P[\textit{m random x's miss any Ts}] \leq F(n)(1 - \frac{\varepsilon}{F(n)})^m$$
$$P[\textit{R' has larger error than } \varepsilon] \leq F(n)(1 - \frac{\varepsilon}{F(n)})^m \ < \delta$$

Bounding the chance that our R' has an error larger than $\varepsilon$ by $\delta$:

$$F(n)(1 - \frac{\varepsilon}{F(n)})^m \ < \delta$$

Using $e^{-x} \geq (1-x)$

$$F(n)e^{-m\varepsilon/F(n)} \geq F(n)(1 - \frac{\varepsilon}{F(n)})^m$$

So instead we can use:

$$F(n)e^{-m\varepsilon/F(n)} < \delta$$
$$-m\varepsilon/F(n) < \log(\delta/F(n))$$
$$m\varepsilon/F(n) > \log(F(n)/\delta)$$
$$m > (F(n)/\varepsilon)\log(F(n)/\delta)$$

So any n-dimensional hyper-rectangle is learnable.

**1.b Assume we have a 2-dimensional feature space R2, and consider the set of concepts that are L1-balls: $c = \{(x,y) : |x| + |y| \leq r\}$ (basically, all L1-balls centered around the origin). Use a learner that fits the tightest ball. Show that this class is PAC-learnable from training data of size m.**

We can look at this as an axis-aligned hyper-rectangle (from Section 1.a) with $n = 1$, and rotated by 45° degrees. An example of this can be seen in Figure 1.

We can again create a strip $T$ representing the decision area on one side of the $L_1$ ball, and grow a region $T'$ to cover a probability mass of $\frac{\varepsilon}{4}$. From here the proof follows the steps taken in the lecture.

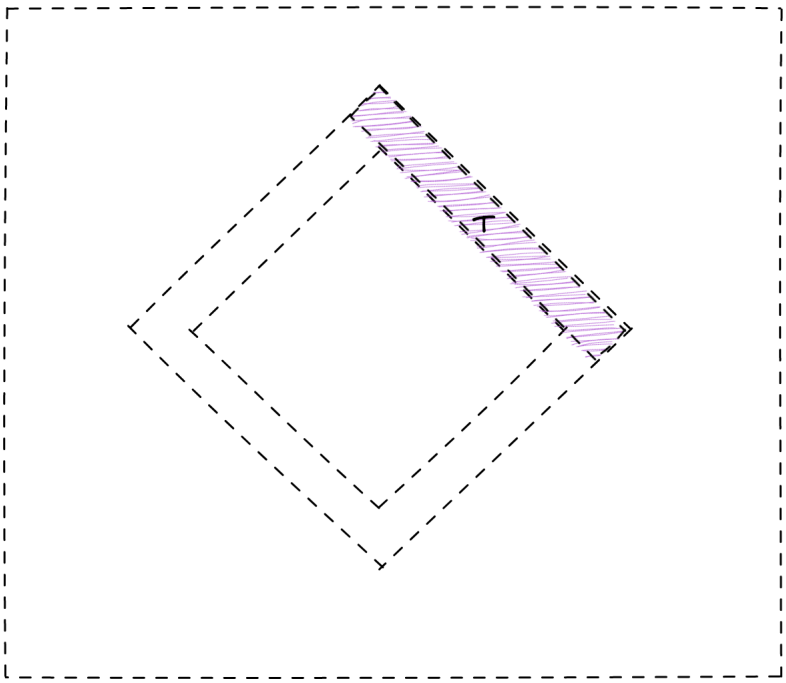
We can also take the result from Section 1.a which gave a general bound off

$$m > (F(n)/\varepsilon)\log(F(n)/\delta)$$

and replace $F(n)$ with 4 (or just use $F(n)$ with $n = 2$ which gives $F(2) = 4$). This gives a bound on the error: (same as a axis-aligned rectangle)

$$m > (4/\varepsilon)\log(4/\delta)$$

Figure 1: Error on a L1 Ball




**1.c Now we extend the previous class by allowing the varying center: $c = \{(x,y) : |x-x0| + |y-y0| \leq r\}$. Is this class PAC-learnable? Proof if it is, or otherwise disproof it.**

This is just a generalization of Section 1.b, but now with a varying center. Our hypothesis space is still a tightest fit ball around the sampled data, which will now also be shifted by $(x_0, y_0)$. Since we are still dealing with a consistent learner this class is also PAC-learnable.

The worked out proof for this is the same as for Section 1.b.

**1.d Generate data according to, what you would argue is, the 'best case' scenario, i.e., for which learning is easiest. Clearly specify how you generate this data and show the resulting learning curve.**

First I'll discuss the setup I'm using to run these tests. I'm generating $L_1$ balls with $r = 1$. I've set the size of the sample space to 2. This means that the sample space is a $2x2$ square with the center at $(0,0)$. The surface area of this square is 4, and the surface area of the $L_1$ ball is 2, exactly half of the sample space.

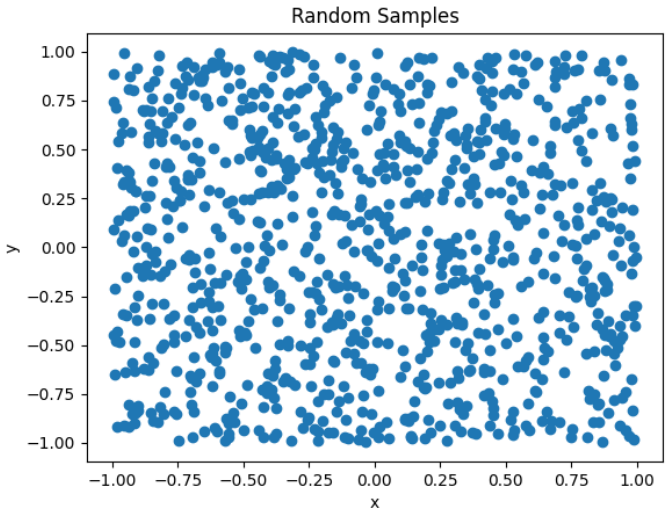
For each tests I'll be using a sample size of $m \in \{2,...,2000\}$, and I'll be taking the average of 100 trials. From a set of samples $m$ the surface area of the $L_1$ ball can be determined, and from here the error of the ground truth can be calculated. This error is

$$(2 - estimated\_area)/2$$

2 is the surface area of the ground truth $L_1$ ball. This is divided by 2 because only positive samples that fall into this area will be misclassified (negative samples are never misclassified).

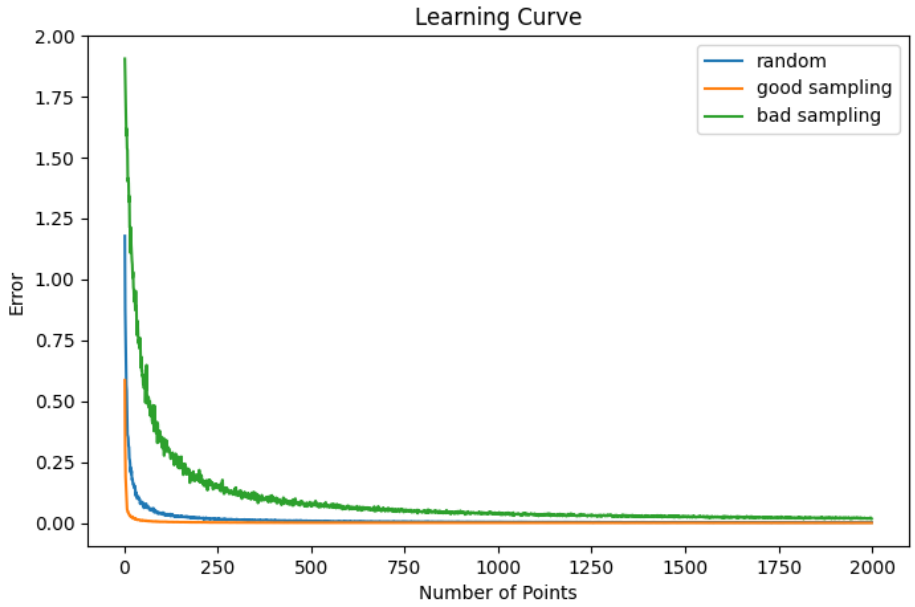
First I wanted to run a test using a basic uniform distribution (see Figure 2 for an example of generated samples) to confirm the code is working properly. The results from this test can be seen in Figure 3. The more samples get added, the lower the error will be.

Figure 2: 1000 randomly generated samples




The absolute best-case scenario would be a sampling method that generates the first sample to be $x \in \{(1,0),(0,1),(-1,0),(0,-1)\}$. All of these samples are on the outer corner of the $L_1$ ball, and the estimated $r$ will always be 1, so the estimated area will be 2, and error will be 0.

Figure 3: Learning Curve for a points sampled from a uniform distribution




This however does not seem like a very fair sampling method, so a fairer method might be one that generates samples close to the edge of the $L_1$ ball, so points where

$$|x| + |y| \approx 1$$

Some of these points will fall outside of the $L_1$ ball and cannot be learnt from, but some of them will fall inside and can be used to determine the tightest fit $L_1$ ball.

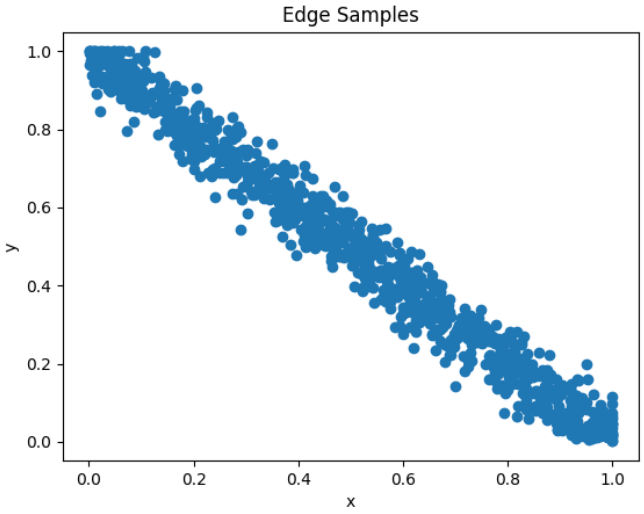
I generate these points in the following way. First I sample a radius $r \sim N(1, 0.05)$, then sample $x \sim U(0, r)$, and calculate $y = r - x$. both $x$ and $y$ are then clipped to be in $[-1, 1]$ (to be within the previously defined sample space) This generates samples around 1 edge of the $L_1$ ball. See Figure 4 for an example of how these samples are generated. The results can be seen in Figure 3. The method can be adapted to sample all 4 edges of the $L_1$ ball, but the results for this are the same since we are dealing with absolute values.

This sampling method goes to an error of zero quite quickly and stays there.

### 1.e Similarly, generate data according to, what you would argue is, the 'worst case' scenario. Clearly specify how you generate this data and show the resulting learning curve. How close is this curve to the theoretical PAC bound?

I think there is several cases that give very bad performace, they all have the same thing in common: Samples are generated far away from the edge of the ground-truth $L_1$ ball.

Figure 4: 1000 samples generated around the edge of the $L_1$ ball




An extreme version of this would be a method that always samples points outside of the ground truth $L_1$ ball. In this case the tightest fit $L_1$ ball would not exist. This however does not seem like a fair sampling method to compare to.

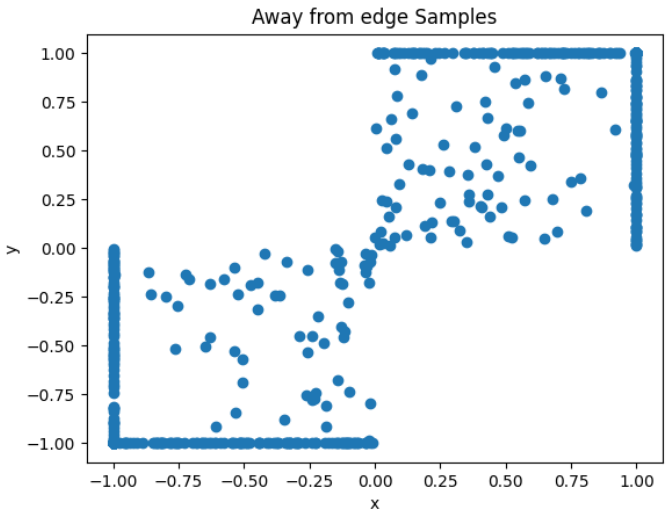
Instead I sampled using the same method as Section 1.d, but now $r \sim N(1, 7.5)$. An example of generated samples can be seen in Figure 5. Results can be seen in Figure 3.

This sampling method takes much longer to get close to an error of 0 than the 2 other sampling methods.

To test how close this method gets to the PAC-bound I made the following test: I chose $\varepsilon = 0.25$ and $\delta = 0.05$, according to the formula from Section 1.b this gives $m = 100$, i.e. for $m = 100$ we should get an error of 0.25 or lower 95% of the time. I than ran the same tests I used to create the graphs but now with 10000 trials.

However when running these tests it failed about 50% of the time. Perhaps this is because the sampling method used is unlikely to sample the entire sample space, and tends to clump around the edges due to the clamping (as can be seen in Figure 5).

Figure 5: 1000 samples generated away from the edge of the $L_1$ ball



## 2 VC Dimension

Let us assume we are dealing with a two-class classification problem in d-dimensions. Let us start off with refreshing our memories. Consider the class of linear hypotheses (i.e., all possible linear classifiers) in d-dimensional space.

### 2.a Given N data points, how many possible labelings does this data set have?

Each data point can have 2 labels, so $2^N$

### 2.b What is the dimensionality d we need, at the least, for our class of linear hypotheses to be able to find perfect solutions to all possible labelings of a data set of size N? Stated differently, what is the smallest d that allows us to shatter N points?

From the *Machine Learning 1* course (2021/22 edition, Week 4: Complexity and SVMs): A linear classifier with $d$ parameters (dimensions) can shatter $d+1$ points. In other words, the smallest $d$ that allows us to shatter $N$ points is $d = N-1$.

### 2.c Consider d = 1 and a data set of size N. At maximum, how many different labelings of such a data set can decision stumps solve perfectly, i.e., with zero training error?

This would be all cases where there is no overlap, i.e. all points with class 1 (or 2) are either all on the left or right side. See Figure 6 for an example with N=3. Note that the first 3 and last 3 examples are the same, just with inverted labelings.

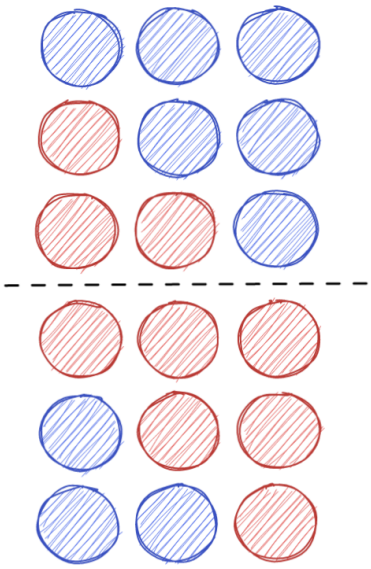
With a dataset of size N, there are N ways in which we can have class 1 be on the left side and have a decision stump be able to solve it perfectly. There can be $1, 2, ..., N$ points with class 1 in a row. Because each of these possible label assignments has a compliment (just invert the classes), we end up with a total of $2N$ possible labelings that a decision stump can solve perfectly.

### 2.d With the answer from 3.c, give an upper-bound on the maximum number of different labelings a data set of size N in d dimensions can get by means of decision stumps.

We can once again look at Figure 6, but now instead of true labels we can view the labels (red and blue) as labels predicted by the decision stump. The decision stump either assigns all points to the left of it as class 1 or as class 2. So in $d = 1$ we know the maximum number of different labelings a data set of size N can get is $2N$.

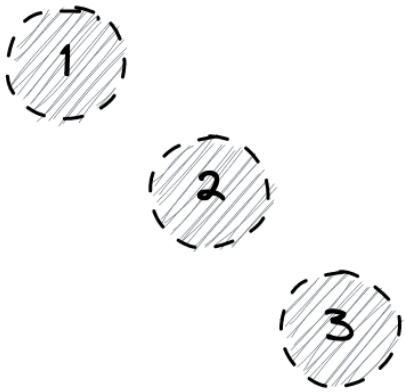
This can be viewed in another way, for every dimension the decision stump can make $2N$ different labelings. For example in 2D (Figure 7) the decision stump for the x dimension can make $2N$ labelings, and the decision stump for the y dimension can make $2N$ labelings.

Figure 6: Example of perfectly solvable labels with $N = 3$



This continues for each dimension added, so we end up with a total of $2Nd$ maximum number of different labelings a data set of size N in d dimension can get by means of decision stumps.

Figure 7: Example with $d = 2$ and $N = 3$




I'd like to add some more notes to this solution. This is not neccesary to read, but it sheds some light on some of the mistakes I made.

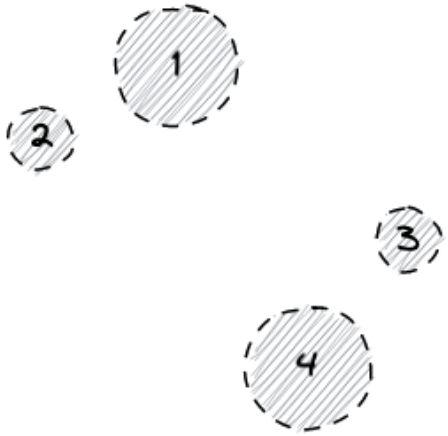
While trying to answer this question I pretty quickly came to the answer of $2Nd$, but then later thought it to be wrong because this formula does not take into account duplicate labelings. In Figure 7 for example the x and y decision stumps can create many duplicate label assignments.

I tried to find patterns in combinations of $N$ and $d$, for example I noticed that for $d = 2$ only $N \leq 3$ was solvable (see Figure 8), and for $d = 3$ at least $N \leq 4$ was solvable (see Figure 9, here smaller circles can be seen as being further away in the z dimension).

From here it seemed like there was a pattern where for any $N \leq (d+1)$ the problem was fully solvable (although I show in later sections that this is also not true), so the number of possible labelings the decision stump could make was $2^N$. I could however not find a pattern in the number of labelings a decision stump could make for problems with $N > (d+1)$.

Figure 8: Solvable example with $d = 2$ and $N = 3$


Figure 9: Solvable example with $d = 3$ and $N = 4$




It was not until a day of thinking and researching that I found these Slides[2] (Page 5) that confirmed my original answer.

[2] https://people.csail.mit.edu/alinush/6.867-fall-2013/2013.10.22.w8.tu-lecture-13-generalization-part-2.pdf

| Dimensionality | Upper-bound on VC-Dimension |
|---|---|
| 1 | 2 |
| 2 | 4 |
| 3 | 4 |
| 4 | 5 |
| 5 | 5 |
| 6 | 6 |
| 7 | 6 |
| 8 | 6 |
| 9 | 6 |
| 10 | 7 |
| 11 | 7 |
| 12 | 7 |
| 1024 | 14 |
| $2^{100}$ | 107 |

Table 1: Results for upper bounds on the VC Dimension

### 2.e Using the bound from Exercise 3.d, determine the smallest upper-bound on the VC-dimension for decision stumps for dimensionalities $d \in \{1,2,3,4,5,6,7,8,1024,2^{100}\}$

I used the definition from the book *Foundations of Machine Learning* [2] for the upper-bound on the VC-Dimension

**Smallest upper-bound on VC-dimension**: To give an upper bound, we need to prove that no set $S$ of cardinality $N$ can be shattered

Since we know the maximum number of different labelings a data set of size $N$ in $d$ dimensions can get by means of decision stumps to be $2Nd$ (Section 2.d), and the total number of possible different labelings of a data set of size $N$ can get is $2^N$ (Section 2.a), we have to find the first case where $2Nd < 2^N$. In other words, find the first $N$ for which the classifier cannot create every possible labeling.

Since I could not find an analytical solution to this I decided to solve it programmatically. The results can be seen in Table 1. I've also added results for $d \in \{9,10,11,12\}$ for Section 2.e.

To me one interesting thing about these results is that they did not seem to agree with some other results I had found online. This homework assignment[3] gives an upper-bound on the VC dimension of $2(\log_2(d)+1)$. I think they might have found an analytical solution to the upper-bound, but in doing so had to slightly increase it.

[3] `http://courses.cms.caltech.edu/cs253/hw/hw2.pdf`

**2.f Write a program that finds lower-bounds experimentally and that would even find the true VC-dimension given enough time (and a sufficient numerical precision). Provide a clear description for (possibly in terms of pseudo code) and reasoning behind the program. (Hint: given an arbitrary data set of of size N, you can of course always check if the class of decision stumps shatters it.)**

Once again the definition from the book *Foundations of Machine Learning* [2] for the lower-bound on the VC-Dimension is used

**Upper-bound on VC-dimension**: To give a lower bound, we need to show that a set $S$ of cardinality $N$ can be shattered

```python
def can_shatter(dataset):
    d = len(dataset[0])
    max_labelings = 2**len(dataset)
    # both are arrays of size 2
    min_values = np.min(dataset, axis=0)
    max_values = np.max(dataset, axis=0)

    found_labels = set()
    # iterate over dimension, lower_bound, upper_bound
    iterator = zip(range(d), min_values, max_values)
    for stump_dimension, start, end in iterator:
        for stump_boundary in range(start-1, end+1):
            # anything lower than the stump boundary gets True
            labeled = dataset[:, stump_dimension] <= stump_boundary
            found_labels.add(tuple(labeled))
            # inverse is always achievable
            found_labels.add(tuple(~labeled))
            # stop when we found the max number of possible labels:
            # dataset can be shattered
            if len(found_labels) == max_labelings:
                return True
    return False
```

Listing 1: Can Shatter Method

The code for this can be seen in Algorithm 1. I tried writing it out in pseudo code but this turned out to be almost the exact same as the Python code, so I think it's clearer to show the original code with comments.

One thing that's good to explain is how I choose the locations of the decision boundaries. In my code the samples are always on integer coordinates (more on my sampling method later), so by iterating my decision boundary over all possible integers I am sure to get all possible labelings. It is of course not needed to go over every possible integer, so instead for each dimension I choose the decision boundary to be in:

$$boundary \in \{lower\_bound - 1, lower\_bound, ..., upper\_bound\}$$

Here *lower_bound* and *upper_bound* are the lowest and highest value found for each dimension in the data set.

I tried multiple methods of sampling the possible search space. A first question I had was how big should the space be. For example if there is only 3 samples in 2 dimensions it seems unneccesary to have these samples be in a space of size $1000x1000$. In some earlier experiments I noticed that for $d \leq 4$ I could find solutions for a space where each dimension has a max size of $d$. So this was often the starting size of my sample space, which I then increased if I thought it was needed.

For the sampling method itself I tried 2 methods, *random* and *enumeration*. A problem I ran into with both methods was the sheer size of the possible number of data sets. If we have $d = 3$ and each dimension has a size of 3 this gives 27 possible points. Choosing 4 points from this (the upper-bound for $d = 3$ found in Section 2.e) gives a total of 17550 possible data sets. This is still easy to enumerate, but the size grows rapidly. For $d = 5$ and each dimension having a size of 5 there is 3125 possible points. Choosing 5 from this gives $2.47e15$ possible data sets. Because the number of possible data sets grows so rapidly I decided to stick with random sampling.

```
def check_dimensionality(config):
    d, upper_bound, max_attempts = config
    # check from highest possible VD dimension first
    # this was if it is found we can immediately stop
    iterator = list(range(1, upper_bound+1))[::-1]
    for N in iterator:
        attempts = 0
        for data in generate_random_data(d, N):
            # just need to find a single dataset that can be shattered
            if can_shatter(data):
                print(f'Highest N for dimension {d} is: {N}')
                return
            attempts += 1
            if attempts >= max_attempts:
                break

```

Listing 2: Dimensionality Check

Finally Algorithm 1 was used in Algorithm 2 to check multiple data set sizes to find the VC-dimension.

| Dimensionality | Lower-bound on VC-Dimension |
|---|---|
| 1 | **2** |
| 2 | 3 |
| 3 | **4** |
| 4 | 4 |
| 5 | **5** |
| 6 | 5 |
| 7 | 5 |
| 8 | 5 |
| 9 | 5 |
| 10 | 5 |
| 11 | 6 |
| 12 | 6 |

Table 2: Results for experimentally found lower bounds on the VC Dimension

**2.g Determine for every dimensionality $d \in \{1,2,3,...,10,11,12\}$ the true VC-dimension and proof that, indeed, it is the actual VC-dimension for decision stump in spaces of that dimensionality. (Note: please be advised that you are not expected to completely crack this open(?) problem, but do have a go at it and see how far you can get in a reasonable amount of time. Hint: you probably can get some inspiration from the lower-bounds your program from 3.f finds.)**

I used the method described in Section 2.f to experimentally find lower-bounds on the VC dimension. Results for this can be seen in Table 2. The boldfaced values are where the lower and upper bound match. To find these values I ran tests with the algorithm shown in Algorithm 2 using 1000000 attempts each time. I varied the size per dimension from $N$ to $16N$.

I already described some of my experiments with finding a tighter bounds and VC-dimensions at the end of section 2.d. While researching that I found the paper [1]. This paper states that the VC-dimension of a decision tree with dimensionality d is given by solving for the largest integer m that satisfies

$$2d \geq \binom{m}{\lfloor \frac{m}{2} \rfloor}$$

I ran code to solve this inequality, and the results for this can be seen in Table 3. Almost all VC-dimensions are the same as my experimentally found lower-bounds on the VC-dimension except for $d = 10$, where I found 5 but the inequality has a result of 6. After seeing this I tried to use Algorithm 2 to find the correct VC-dimension for $d = 10$ by only

| Dimensionality | VC-Dimension |
|---|---|
| 1 | 2 |
| 2 | 3 |
| 3 | 4 |
| 4 | 4 |
| 5 | 5 |
| 6 | 5 |
| 7 | 5 |
| 8 | 5 |
| 9 | 5 |
| 10 | **6** |
| 11 | 6 |
| 12 | 6 |

Table 3: VC-Dimension for decision stumps with dimensionality $d$

searching with $N = 6$ and my code was able to find a valid solution after about an hour of trying.

# References


[1] Jean-Samuel Leboeuf, Frédéric Leblanc, and Mario Marchand. Decision trees as partitioning machines to characterize their generalization properties. *CoRR*, abs/2010.07374, 2020.

[2] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of Machine Learning*. Adaptive Computation and Machine Learning. MIT Press, Cambridge, MA, 2 edition, 2018.