Examples for Machine Learning

December 25, 2023

"If I had more time, I would have written a shorter letter."

1 Introduction

1.1 Power set

The power set is the set of all subset. For example, let $A = \{1, 2, 3\}$

$$2^A = \{\{\}, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$$

In this example, there are $8 = 2^3 = 2^{|A|}$ elements in the power set. In general $|2^A| = 2^{|A|}$, hence the notation.

1.2 m-elementary subsets

The set of all m-elementary subsets is denoted by $\binom{A}{m}$. For example, let $A = \{1, 2, 3\}$.

$$\binom{A}{2} = \{\{1, 2\}, \{1, 3\}, \{2, 3\}\}.$$

In this example, there are $3 = \binom{3}{2} = \binom{|A|}{2}$ elements in the set (binomial coefficient). In general $|\binom{A}{2}| = \binom{|A|}{2}$, hence the notation.

1.3 set of all maps from A to B

The set of all maps from A to B is denoted by B^A. For example let $A = \{x, y\}$ and $B = \{1, 2, 3\}$.

$$\begin{aligned} B^A = \{&(x \mapsto 1, y \mapsto 1),\\ &(x \mapsto 1, y \mapsto 2),\\ &(x \mapsto 1, y \mapsto 3),\\ &(x \mapsto 2, y \mapsto 1),\\ &(x \mapsto 2, y \mapsto 2),\\ &(x \mapsto 2, y \mapsto 3),\\ &(x \mapsto 3, y \mapsto 1),\\ &(x \mapsto 3, y \mapsto 2),\\ &(x \mapsto 3, y \mapsto 3)\} \end{aligned}$$

In this example, there are $9 = 3^2 = |B|^{|A|}$ elements in the set. In general $|B^A| = |B|^{|A|}$, hence the notation.

2 Supervised learning

The general idea will be presented, once we had an example.

3 Learning of DNFs

3.1 Disjunctive Normal Forms

DNFs always look a bit like this:

$$(x_1 \wedge x_2 \wedge x_4) \vee (x_1 \wedge \neg x_2) \vee (x_2 \wedge \neg x_3)$$

It's some and-clauses (conjunctions) joined in one big or-clause (disjunction), hence the name. If we are using 0 and 1 instead of booleans, we can represent an and by multiplying the variables ($x_1 \wedge x_2 = x_1 * x_2$) and negation as 1 - x ($\neg x_1 = (1 - x_1)$). I will stick with the boolean notation as I find it more intuitive, even though in the lecture the product notation is chosen to be more consistent with the labelling using 0 and 1. Just know that 0, 1 and false, true will be used somewhat interchangeably here.

3.2 Example

Let's start with an example. Our function / program will take some input about attributes of a city and should return, whether the city is a good choice to live in. The attributes for each city will be

1. Are the housing prices high? (yes/no)
2. Does the city have a high population? (yes/no)
3. Is the public transport system good? (yes/no)

And the decision will be

- Is the city livable? (yes/no)

The inputs and outputs are binary decisions (yes/no) which can be represented by booleans (true/false) or binary labels (1/0). Our function should take in the attributes of a city and use a DNF (disjunctive normal form) to output a decision for the city. Our DNF for deciding, whether a city is livable could look something like this:

$$(\neg x_{costly} \wedge x_{transport})$$

Notice, this is a DNF with just one and-clause, which is why there is no $\vee$ required. It also doesn't make use of all attributes (it skips population). This DNF would label all cities with low costs and good transport as a livable city. All other cities would not be considered livable. This sounds reasonable, but we are not just interested in finding something that sounds reasonable. We are interested in finding something that fits some labeled data we are given.

3.2.1 Labeled Data

The data we are given could look something like this.

- Dresden
  - $x_{costly} = 0$ (low housing prices)
  - $x_{populated} = 1$ (high population (for German standards))
  - $x_{transport} = 1$ (good public transport)
  - $y = 1$ (It's a good city to live in)
- Heidelberg
  - $x_{costly} = 1$
  - $x_{populated} = 0$
  - $x_{transport} = 1$

- $y = 1$

- Stuttgart
  - $x_{costly} = 1$
  - $x_{populated} = 1$
  - $x_{transport} = 1$
  - $y = 0$ (I just needed something with a zero, Stuttgart isn't that bad)

It could also be written in a table like this:

	costly	populated	transport	livable
Dresden	0	1	1	1
Heidelberg	1	0	1	1
Stuttgart	1	1	1	0

The formal notation will be presented in the following paragraphs.

3.2.2 S: Set of Samples

S is our set of samples:

$$S = \{Dresden, Heidelberg, Stuttgart\}$$

3.2.3 X: Set of all possible attribute assignments

Our attributes are denoted by V.

$$V = \{costly, populated, transport\}$$

In the lecture our attributes are often defined more generally with just $V = \{1, 2, 3\}$. Every attribute can take on the value 0 or 1. Thus, our set of all possible attribute assignments, denoted by X, will look like this:

$$\begin{aligned} X &= \{0,1\}^V \\ &= \{(costly \mapsto 0, populated \mapsto 0, transport \mapsto 0), \\ &\quad (costly \mapsto 0, populated \mapsto 0, transport \mapsto 1), \\ &\quad (costly \mapsto 0, populated \mapsto 1, transport \mapsto 0), \\ &\quad (costly \mapsto 0, populated \mapsto 1, transport \mapsto 1), \\ &\quad (costly \mapsto 1, populated \mapsto 0, transport \mapsto 0), \\ &\quad (costly \mapsto 1, populated \mapsto 0, transport \mapsto 1), \\ &\quad (costly \mapsto 1, populated \mapsto 1, transport \mapsto 0), \\ &\quad (costly \mapsto 1, populated \mapsto 1, transport \mapsto 1)\} \end{aligned}$$

3.2.4 x: Matching attribute assignments to cities

Assigning cities one of the attribute assignments will be done with a map x from the samples to some elements of X.
$(x : S \to X)$:

$$\begin{aligned} x = \{&Dresden \mapsto (costly \mapsto 0, populated \mapsto 1, transport \mapsto 1), \\ &Heidelberg \mapsto (costly \mapsto 1, populated \mapsto 0, transport \mapsto 1), \\ &Stuttgart \mapsto (costly \mapsto 1, populated \mapsto 1, transport \mapsto 1)\} \end{aligned}$$

3.2.5 y: Labels for cities

The labels for the cities (whether they are livable or not) are defined in a map y from S to 0 or 1.
$(y : S \to \{0, 1\})$:

$$y = (Dresden \mapsto 1, Heidelberg \mapsto 1, Stuttgart \mapsto 0)$$

3.3 Deciding

We now try to find the best DNF that takes in the attributes of a city and returns the correct label for this city. The DNF defined earlier, which was just a guess, looked like this:

$$(\neg x_{costly} \wedge x_{transport})$$

It would map Dresden as follows:

$$Dresden : (\neg x_{costly} \wedge x_{transport}) = \neg false \wedge true = true$$

Or with $\{0, 1\}$:

$$Dresden : (1 - x_{costly}) * x_{transport} = (1 - 0) * 1 = 1$$

It would correctly assess that Dresden is a livable city. However, it would incorrectly label Heidelberg as a non-livable city. So our goal now is to find among all possible DNFs (Θ) one that correctly maps all cities to their respective label.

3.3.1 Formal Notation of a DNF

A DNF like this

$$(\neg x_{costly} \wedge \neg x_{populated} \wedge x_{transport}) \vee (x_{costly} \wedge x_{populated})$$

will be represented as a set. Each element in the set corresponds to one and-clause. Each of these and-clauses is represented by a tuple (V_0, V_1), where V_0 defines the variables that occur in the negated form in the and-clause and V_1 defines the variables that occur in the non-negated form in the and-clause. The DNF above would become

$$\{(\{x_{costly}, x_{populated}\}, \{x_{transport}\}), (\{\}, \{x_{costly}, x_{population}\}\}$$

The set of all possible and-clauses can then be defined as $\Gamma = \{(V_0, V_1) \in 2^V \times 2^V | V_0 \cap V_1 = \emptyset\}$. The restriction $V_0 \cap V_1 = \emptyset$ states that a variable cannot occur in it's negated and non-negated form at the same time. The set of all DNFs Θ then is all disjunctions that can be built by using elements of the set of all conjunctions Γ. So $\Theta = 2^\Gamma$.

3.3.2 Complexity of DNFs

Two way of judging the complexity of DNFs are depth and length. The regularizer R_l for length counts the total number of variables that occur in the DNF. The regularizer R_d for depth counts the number of variables in the longest and-clause withing the DNF. The following DNF has a length of 5 and a depth of 3.

$$(\neg x_{costly} \wedge \neg x_{populated} \wedge x_{transport}) \vee (x_{costly} \wedge x_{populated})$$

3.3.3 Learning Problem

The supervised learning problem asks us to find a DNF that gets the labelling correct, while keeping complexity low. How relevant correctness is compared to complexity is determined by a parameter λ. We want to find a θ to minimize

$$\lambda * R(\theta) + (1 - \lambda) * averageLoss(\theta)$$

Let's look at our example. The first DNF we came up with

$$(\neg x_{costly} \wedge x_{transport})$$

mapped Dresden and Stuttgart correctly, but failed for Heidelberg. The loss for a sample is 0, if the labelling was correct and 1 if the labelling was incorrect. The total loss is therefore:

$$\sum_{s \in S} L(f_\theta(x_s), y_s) = 0 + 0 + 1 = 1$$

The average loss then is $\frac{1}{3}$. The length is 2 and depth is also 2.

By looking at the labeled data that is mapped to 1 we can also define a DNF that maps everything correctly:

$$(\neg x_{costly} \wedge x_{populated} \wedge x_{transport}) \vee (x_{costly} \wedge \neg x_{populated} \wedge x_{transport})$$

Since this DNF gets every label right, the total and average loss are 0. However, the length is 6 and depth is 3. So depending on how we weigh correctness vs complexity (and whether we look at depth or length), both of the two DNFs could be more suitable for some given requirements than the other.

3.4 Generalizing learning problems

Our goal was to find a good DNF, that can correctly decide for some input data, whether a city is livable or not. To define what a good DNF is we looked at two aspects:

1. Correctness: We are given some labeled data. The DNF (θ) should be chosen/learned in such a way, that the function gets this training data right. To measure this, we use the Loss function L.
2. Complexity: The function should not be too complicated and have small length or depth. To measure this, we use the regularizer R.

In general, we are interested in finding a good θ, which has little loss and small complexity. θ could describe the structure of a DNF, but also the structure of a Binary Decision Tree or parameters in a neural network.

3.5 Bounded Depth DNF Problem

A different learning problem is the bounded depth DNF problem. It asks us to get every label right, while restricting the depth (longest and-clause) to some limit m. Both of the examples above fail to be a solution of a DEPTH-2-DNF. One doesn't get everything correct, the other one has a depth over 2.

A suitable solution would be:

$$(\neg x_{costly} \wedge x_{populated}) \vee (x_{costly} \wedge \neg x_{populated})$$

3.5.1 Hardness of finding solutions to depth-m-DNF

Finding a solution to the DEPTH-M-DNF problem for large inputs is hard. For a growing number of attributes and labeled data, the time to find a solution grows at least exponentially in our best known algorithms. In fact, we can show that finding a solution would also allow us to find solutions to problems, for which there is believed to be no algorithm better than exponential time growth. One such problems is SET-COVER.

set-cover We are given some sets $\Sigma = \{\{a, b\}, \{b\}, \{c, d\}\}$ and another set $S' = \{a, b, c, d\}$ which we are supposed to cover using only m (for example m=2) sets from Σ. As suitable cover for m=2 would be $\sigma_1 = \{a, b\}$ and $\sigma_3 = \{c, d\}$. This problem is known to be hard to solve as the inputs grow larger.

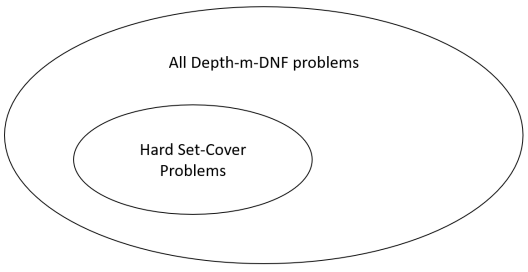

Figure 1: DEPTH-M-DNF is a more general, possibly harder problem than SET-COVER

Purpose of a reduction We can show that solving SET-COVER is basically the same problem as some DEPTH-M-DNF problems. This would show that some instances of DEPTH-M-DNF are just as hard to solve as the hard SET-COVER problem. A program that solves all DEPTH-M-DNF problems would therefore also have to solve the hard SET-COVER problems making DEPTH-M-DNF at least as difficult as SET-COVER. Showing that one problem (SET-COVER) is a 'sub-problem' of another problem (DEPTH-M-DNF) is called a reduction (from SET-COVER to DEPTH-M-DNF).

The reduction Given any SET-COVER problem we can translate it into a DEPTH-M-DNF problem. We can then also translate it back, thus showing that the problems are basically the same. The SET-COVER problem can be written in a table, where every cell denotes, whether the set at the top covers the letter on the left, like this:

	$\{a,b\}$ σ_1	$\{b\}$ σ_2	$\{c,d\}$ σ_3	$\{a,b,c,d\}$ S'
a	covered	not covered	not covered	→ covered
b	covered	covered	not covered	→ covered
c	not covered	not covered	covered	→ covered
d	not covered	not covered	covered	→ covered

We will try to choose some σ, corresponding to a column, such that each row contains 'covered' at least once. Since the rows correspond to each element of S', we have then found a cover for S'. A cover for m=2 is σ_1 and σ_3 as established earlier.

	$\{a,b\}$ σ_1	$\{b\}$ σ_2	$\{c,d\}$ σ_3	$\{a,b,c,d\}$ S'
a	covered	not covered	not covered	→ covered
b	covered	covered	not covered	→ covered
c	not covered	not covered	covered	→ covered
d	not covered	not covered	covered	→ covered

The equivalent DEPTH-M-DNF problem would look like this:

	σ_1	σ_2	σ_3	y
a	0	1	1	0
b	0	0	1	0
c	1	1	0	0
d	1	1	0	0

'Covered' has been replaced by 0 and 'not covered' by 1.[1]

As we can see, the solution of choosing σ_1 and σ_3 would also be a solution to our DNF problem: $\sigma_1 \wedge \sigma_3$ would be a (single-and-clause-) DNF, that calculates each y correctly.

However, there is one small problem. This showed that a solution of the SET-COVER can be translated to the DNF problem, but not the other way around. In fact, $\sigma_1 \wedge \neg\sigma_2$ and even $\bot$ would be valid solution to the DNF problem, but not to the SET-COVER problem.[2]

To translate back, we need one more line in our table, that ensures we don't use negations. The equivalent DNF problem becomes:

	σ_1	σ_2	σ_3	y
a	0	1	1	0
b	0	0	1	0
c	1	1	0	0
d	1	1	0	0
1	1	1	1	1

Now solutions, such as $\sigma_1 \wedge \neg\sigma_2$, are no longer possible and we will also arrive at a DNF like $\sigma_1 \wedge \sigma_3$, which can be translated back to SET-COVER. Since length and depth of any single-and-clause-DNF are the same, we have also shown that LENGTH-M-DNF is hard to solve.[3]

[1] That's a bit counter-intuitive, as one might expect it the other way around, but it is caused by the structure of DNFs. The other way around would not work that well. You can try and see what would come out of it.

[2] The corresponding solutions to the set cover problem would be to choose σ_1 and $S' \setminus \sigma_2$ or $S' \setminus \{\}$, which do build proper set covers, but the chosen elements aren't in the set Σ

[3] For this last part we assumed that the DNF can always be written as a single-and-clause-DNF, which is not at all obvious and only works, due to our 0 and 1 choices. Feel free to pause and ponder why, if we find a solution to the DNF problem, it can be reduced to a single-and-clause-DNF.