Boolean Learning Example: Let's consider a concrete example with a finite number of the hypothesis set to further elaborate the main concept on the size of the hypothesis set, the probability structure of the data, and the influence of the size of the training set.

Let us consider a Boolean target function (i.e., $\mathcal{Y} = \{0, 1\}$) over a four-bit vector representation of input space $\{0000, 0001, \dots, 0111, 1000, 1001, \dots, 1111\}$.

Q: For this example, what is the dimension of the input space $\mathcal{X}$?

A: 4

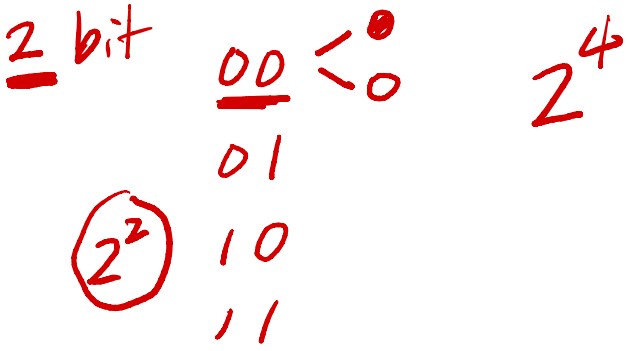
Q: For this example, how big is the entire input space $\mathcal{X}$?

A: $2^4 = 16$

Q: For this example, how big is the entire Boolean hypothesis set $\mathcal{H}$?

A: $2^{16} = 65{,}536$

How to compute these numbers?



In general, the target function f is **unknown**. However, to illustrate the concept and identify the learning performance, let us *define* the target function f which just checks if the number of zeros in the binary representation exceeds the number of ones. If so, then the function outputs 1 and 0 otherwise

```
import pandas as pd
import numpy as np
from pandas import DataFrame
df=DataFrame(index=pd.Index(['{0:04b}'.format(i) for i in range(2**4)],
                            dtype='str',
                            name='x'),columns=['f'])
```

Programming Tip: The string specification above uses Python's advanced string formatting mini-language. In this case, the specification says to convert the integer into a fixed-width, four-character (04b) binary representation.

```
df.f=np.array(df.index.map(lambda i:i.count('0'))
              > df.index.map(lambda i:i.count('1')),dtype=int)
df # show all the input vectors and target values
```

x	f
0000	1
0001	1
0010	1
0011	0
0100	1
0101	0
0110	0
0111	0
1000	1
1001	0
1010	0
1011	0
1100	0
1101	0
1110	0
1111	0

target fun

Learning is only feasible in a *probabilistic* sense. To this end, we **assume** that the training set represents a random sampling (**in-sample** data) from a greater population (**out-of-sample** data) that would be consistent with the population that g would ultimately predict upon.

Remark: In other words, we are assuming a stable probability structure for both the in-sample and

out-of-sample data. **This is a major assumption and makes learning feasible!**

Now, presented with a training set consisting of some input/output pairs, our goal is to find g that best matches f on the **training** samples in the dataset $\mathcal{D}$. In other words, minimize errors over the training set ($E_{\text{in}}(g)$) where the subscript indicates in-sample data.

We will now show that we can indeed **predict** something useful outside $\mathcal{D}$ using only $\mathcal{D}$.

Let us assume a very simple probabilistic distribution in which we have the first eight elements from the input space $\mathcal{X}$ are twice as likely as the last eight. The following code is a function that generates elements from $\mathcal{X}$ according to this distribution.

```
def get_sample(n=1):
   if n==1:
      return
'{0:04b}'.format(np.random.choice(list(range(8))*2+list(range(8,16))))
   else:
      return [get_sample(1) for _ in range(n)]
```

Programming Tip: The function that returns random samples uses the `np.random.choice` function from `Numpy` which takes samples (with replacement) from the given iterable. Because we want the first eight numbers to be twice as frequent as the rest, we simply repeat them in the iterable using `range(8)*2`. Recall that multiplying a Python list by an integer duplicates the entire list by that integer. It does not do element-wise multiplication as with `Numpy` arrays.

```
list(range(8))*2+list(range(8,16))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

If we wanted the first eight to be 10 times more frequent, then we would use `range(8)*10`, for example. This is a simple but powerful technique that requires very little code. Note that `np.random.choice` also permits an explicit way to specify more complicated distributions.

The next block applies the function definition f to the sampled data to generate the training set consisting of 5 elements.

```
np.random.seed(12) # for reproduction
train=df.loc[get_sample(5),'f'] # 5-element training set
train.index.unique().shape    # how many unique elements?
```

```
(4,)
```

Notice that even though there are 5 elements, there is redundancy because these are drawn according

to an underlying probability.

```
df['g']=df.loc[train.index.unique(),'f']
df.g
```

```
x
0000    NaN
0001    NaN
0010    1.0
0011    0.0
0100    NaN
0101    NaN
0110    0.0
0111    NaN
1000    NaN
1001    0.0
1010    NaN
1011    NaN
1100    NaN
1101    NaN
1110    NaN
1111    NaN
Name: g, dtype: float64
```

Note that there are `NaN` symbols where the training set had no values. We now do a *bold guess*: we will fill `NaN` with 0.0 to define our final hypothesis g.

```
df.g.fillna(0,inplace=True) #final specification of g
```

We can now address the performance of g with 5 elements of training data. Let's pretend we have deployed this and generate 150 test data.

```
np.random.seed(30) # for reproduction
test= df.loc[get_sample(150),'f']
(df.loc[test.index,'g'] != test).mean()
```

```
0.28
```

The result shows the error rate of 0.28, given the probability mechanism that is generating the data. The following `Pandas` compares the overlap between the training set and the test set in the context of all possible data. The `NaN` values show the rows where the test data had items absent in the training data. Recall that we did a bold guess to set zero for these items. As shown, sometimes this works in its favor, and sometimes not.

```
pd.concat([test.groupby(level=0).mean(),
           train.groupby(level=0).mean()],
           axis=1,
           keys=['test','train'])
```

Programming Tip: The `pd.concat` function concatenates the two `Series` objects in the list. The

`axis=1` means join the two objects along the columns where each newly created column is named according to the given keys. The `level=0` in the `groupby` for each of the `Series` objects means group along the index. Because the index corresponds to the 4-bit elements, this accounts for repetition in the elements. The `mean` aggregation function computes the values of the function for each 4-bit element. Because all functions in each respective group have the same value, the `mean` just picks out that value because the average of a list of constants is that constant.

	test	train
0000	1	NaN
0001	1	NaN
0010	1	1.0
0011	0	0.0
0100	1	NaN
0101	0	NaN
0110	0	0.0
0111	0	NaN
1000	1	NaN
1001	0	0.0
1010	0	NaN
1011	0	NaN
1100	0	NaN
1101	0	NaN
1110	0	NaN
1111	0	NaN

Influence of Size of Training Data

We can now play around to see the influence of the size of training data. For example, if we keep everything the same and increase the size of training data to 12, we will have another final hypothesis g with a better prediction.

```
np.random.seed(12) # for reproduction
train=df.loc[get_sample(12),'f']
del df['g']
df['g']=df.loc[train.index.unique(),'f']
df.g.fillna(0,inplace=True) #final specification of g
np.random.seed(12) # for reproduction
test= df.loc[get_sample(50),'f']
(df.loc[test.index,'g'] != df.loc[test.index,'f']).mean() # error rate
```

```
0.12666666666666668
```

And Pandas

```
pd.concat([test.groupby(level=0).mean(),
           train.groupby(level=0).mean()],
          axis=1,
          keys=['test','train'])
```

	test	train
0000	1	NaN
0001	1	NaN
0010	1	1.0
0011	0	0.0
0100	1	1.0
0101	0	0.0
0110	0	0.0
0111	0	NaN
1000	1	1.0
1001	0	0.0
1010	0	NaN
1011	0	NaN
1100	0	0.0
1101	0	NaN
1110	0	0.0
1111	0	NaN

The above Python code `Boolean_Learning.ipynb` can be downloaded from Google Colab https://colab.research.google.com/drive/1BMmLiTWpkkQjX24uLJJBF2vkcT1OE7UQ?usp=sharing

Important Takeaways from the Learning Example

- Learning is only feasible in a *probabilistic* way. We don't insist on using any particular probability distribution, or even on knowing what distribution is used. However, whatever distribution we use for generating the samples, we must also use when we evaluate how well g approximates the *unknown* target function f.
- The hypothesis g is not fixed ahead of time before generating the data, because which hypothesis is selected to be g depends on the data.
- We can **predict** something useful outside the training set $\mathcal{D}$ using only $\mathcal{D}$. The bigger the training set, the less likely that there will be real-world data that fall outside of it and the better g will perform.