Drug response prediction for cancer patients

Scientists have created a model to predict whether or not cancer patients will respond to a drug.

The same scientist published the details of their research, how the model was built and a detailed description of the data (e.g., the health conditions investigated), the NHS board where the data was collected. The data was deidentified and was not released as it is confidential patient information, and any leak might break existing legislation.

The researchers balanced the benefits and potential risks of the model realease, and it was decided that overall, there is a clear benefit for the population for the model to be made public.

What they didn’t realise, is that the NHS board in question is home to a famous Member of Parliament (MP). This famous MP is a former Prime Minister. There had been some speculation that the MP had cancer, but it is not in the public domain.

Membership Inference

We will use this example to demonstrate a *membership inference* attack. In such an attack, an attacker has access to information about a particular individual (maybe they are famous), and attempts to find out if their data was used to train the model. In this case, knowing if they were in the training set for the model would be disclosive as it would reveal that they had indeed suffered from cancer (all people in the training set had cancer)

Let's get hands on with this example.

The following code imports some standard libraries that we will need.

In [1]:
```
import random
from itertools import product
import numpy as np

np.random.seed(1234)
random.seed(12345)

from scipy.stats import poisson
import pandas as pd
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
```

Create the original model

We are assuming that a model is trained within a TRE on real data. However, we do not have access to real data, so we will randomly generate some realistic looking data.

In particular, we will generate data for 200 people: all are cancer patients, 100 responded well to the drug, and 100 did not. Our MP will be one of the patients in the good responders set.

For each patient, we generate six values that in reality would be extracted from their electronic health records:

1. `diabetes` -- whether or not the patient suffers from diabetes (1 = yes, 0 = no)
2. `asthma` -- whether or not the patient suffers from asthma (1 = yes, 0 = no)
3. `bmi_group` -- the BMI group in which the patient falls (1, 2, 3, or 4)
4. `blood_pressure` -- the blood pressure group in which the patient falls (0, 1, 2, 3, 4, or 5)
5. `smoker` -- whether or not the patient is a smoker (1 = yes, 0 = no)
6. `age` -- the patient's age

Each patient is also associated with a value to indicate whether they responded well to the drug (1) or not (0).

In [2]:
```
#1 is cancer, 0 is no cancer, this is our label and what we want to predict.
response = [1]*99 + [0]*100

df = pd.DataFrame()

#diabetes 0 no, 1 yes
df['diabetes'] = [[1, 0][random.random()>0.7] for n in range(99)] +
                 [[1, 0][random.random()>0.2] for n in range(100)]

#asthma 0 no, 1 yes
df['asthma'] = [[1, 0][random.random()>0.7] for n in range(99)] +
               [[1, 0][random.random()>0.5] for n in range(100)]

#bmi group 1 under, 2 normal, 3 overweight, 4 obese
df['bmi_group'] = [random.choices([1, 2, 3, 4], weights = [0.5, 5, 7, 5], k = 1)[0
                   for n in range(99)] +
                  [random.choices([1, 2, 3, 4], weights = [1, 7, 4, 1], k = 1)[0]
                   for n in range(100)]

#blood pressure 0 is low, 1 is normal, 5 is extremly high
df['blood_pressure'] = [random.choices([0, 1, 2, 3, 4, 5],
                                       weights = [0.5, 1, 5, 6, 1, 0.5], k = 1)[0]
                        for n in range(99)] +
                       [random.choices([0, 1, 2, 3, 4, 5],
                                       weights = [0.5, 5, 5, 1, 1, 0.5], k = 1)[0]
                        for n in range(100)]

#smoker 0 is non smoker, 1 is smoker
df['smoker'] = [[1, 0][random.random()>0.8] for n in range(99)] +
               [[1, 0][random.random()>0.2] for n in range(100)]

#age
x = np.arange(20,90)
pmf = poisson.pmf(x, 72)
age = [random.choices(x, weights = pmf, k = 1)[0] for n in range(99)]
x = np.arange(20,90)
pmf = poisson.pmf(x, 55)
age2 = [random.choices(x, weights = pmf, k = 1)[0] for n in range(100)]
df['age'] = age + age2
```

```
#Add the data of your MP
response = response + [1]

#add new row to end of DataFrame
#the order of the list indicates in order diabetes, asthma, bmi_group, blood_pressu
df.loc[len(df.index)] = [1, 1, 3, 2, 1, 62]
```

This looks like the kind of data that might exist within a TRE. Here's the first few rows:

In [3]:
```
print(df.head())
```

```
   diabetes  asthma  bmi_group  blood_pressure  smoker  age
0         1       0          4               3       1   72
1         1       1          2               3       0   83
2         0       0          4               3       1   63
3         1       1          4               3       0   77
4         1       1          4               2       1   87
```

Our MP is the final row of the data, here are their values:

In [4]:
```
print(df.iloc[199,:])
```

```
diabetes           1
asthma             1
bmi_group          3
blood_pressure     2
smoker             1
age               62
Name: 199, dtype: int64
```

Model training

The researcher trained a particular machine learning model called a Support Vector Machine (SVM). This is a very popular model for tasks in which we want to assign things (in this case patients) to groups (in this case cancer v non-cancer). The attribute inference attack we will perform is not unique to SVMs, we just use them as a popular example.

Training the model is very straightforward -- just a couple of lines of code (the details are not important).

In [5]:
```
# Train a model
prng = np.random.RandomState(12)
svc = SVC(C=1, gamma=3, probability=True, random_state=prng)
svc.fit(df, response)
```

Out[5]:
```
SVC(C=1, gamma=3, probability=True,
    random_state=RandomState(MT19937) at 0x159E5341140)
```

The trained model can be used to make predictions about new individuals. Given data for an individual, it will produce two scores (probabilities). The first is how likely they are to belong to the non-responders group (higher = more likely) and the second how likely they are to belong to the responders group. The scores are always positive, and sum to 1.

For example, if we have an individual who has diabetes, has asthma, has a bmi group of 1, blood pressure of 5. is a non-smoker and is 72 years old, we can use the model to predict whether or not they should belong in the cancer or non-cancer groups:

```
In [6]: test_example = pd.DataFrame(
            {
                'diabetes': 1,
                'asthma': 1,
                'bmi_group': 1,
                'blood_pressure': 5,
                'smoker': 1,
                'age': 72
            }, index=[1]
        )
        predictions = svc.predict_proba(test_example)
        print(f'non-responders score = {predictions[0][0]:.2f}')
        print(f'responders score = {predictions[0][1]:.2f}')
```

```
non-responders score = 0.49
responders score = 0.51
```

The attack

We now assume the role of the attacker. The attacker knows some general properties about the data -- for example, they know the range of values each variable can take. They also know the configuration of the classifier that was trained (that it was an SVM and any parameters that were used to define it (more on this later)). Finally, they know (or can make a good guess at) the input data for the MP (they are famous and this information is perhaps in the public domain). The attacker is going to try and determine, from this information, and with access to the trained model, whether or not the MP was in the dataset and hence determine if they had cancer or not.

How does the attack work?

Recall that when we used the model to make predictions, the model provided two scores -- the cancer and non-cancer scores. The more extreme these scores become (e.g one is close to 1 and the other to 0 (recall that they have to add up to 1)), the more *confident* the model is in assigning that example. It is not uncommon for models to have higher confidence for examples that they were trained on than examples that they haven't seen before. It is this property that the attacker will make use of.

In particular, the attacker will generate their own dataset (known as *shadow* data) that has similar properties to the original. They can do this randomly -- it doesn't matter that it won't be quite right -- all they need to know is the rough ranges of the variables. They will then use some of this data to train their own model (a *shadow* model). This allows them to see roughly what kind of confidence their model gives to examples it was trained on, and examples it wasn't trained on. This gives them an idea about how confident the original model is likely to be on data it was trained on, and data it wasn't trained on. Comparing this to the actual confidence obtained when the MPs data is given to the original model will allow them to infer if the MP was in the training data or not.

Let's look at that step-by-step...

Firstly, the attacker presents the MPs data to the original model to see what the model's predictions are...

In [7]:
```
mp_data = pd.DataFrame({
    'diabetes': 1,
    'asthma': 1,
    'bmi_group': 3,
    'blood_pressure': 2,
    'smoker': 1,
    'age': 62
}, index=[1])
mp_preds = svc.predict_proba(mp_data)
print(mp_preds)
```

```
[[0.06161495 0.93838505]]
```

The model stronly predicts that the MP is in the reponder class. This in itself doesn't tell the attacker that the model was in the training set. What the attacker needs is to estimate how confident the model is when presented with examples from the training set, and when not. This is where the *shadow* model comes in -- they hope that their shadow model is similar enough to the original that the confidences it gives can be used as a proxy against which to compare these values for the MP.

The attacker generates their *shadow* data. There are lots of ways they could do this, in this case they use the same process we used above.

In [8]:
```
#1 is cancer, 0 is no response, this is our label and what we want to predict.
shadow_response = [1]*100 + [0]*100

shadow_df = pd.DataFrame()

#diabetes 0 no, 1 yes
shadow_df['diabetes'] = [[1, 0][random.random()>0.7] for n in range(100)] +
                        [[1, 0][random.random()>0.2] for n in range(100)]

#asthma 0 no, 1 yes
shadow_df['asthma'] = [[1, 0][random.random()>0.7] for n in range(100)] +
                      [[1, 0][random.random()>0.5] for n in range(100)]

#bmi group 1 under, 2 normal, 3 overweight, 4 obese
shadow_df['bmi_group'] = [random.choices([1, 2, 3, 4],
                                          weights = [0.5, 5, 7, 5], k = 1)[0]
                           for n in range(100)] +
                          [random.choices([1, 2, 3, 4],
                                          weights = [1, 7, 4, 1], k = 1)[0]
                           for n in range(100)]

#blood pressure 0 is low, 1 is normal, 5 is extremly high
shadow_df['blood_pressure'] = [random.choices([0, 1, 2, 3, 4, 5],
                                               weights = [0.5, 1, 5, 6, 1, 0.5],
                                               k = 1)[0]
                                for n in range(100)] +
                               [random.choices([0, 1, 2, 3, 4, 5],
                                                weights = [0.5, 5, 5, 1, 1, 0.5],
                                                k = 1)[0]
                                 for n in range(100)]

#smoker 0 is non smoker, 1 is smoker
shadow_df['smoker'] = [[1, 0][random.random()>0.8] for n in range(100)] +
                      [[1, 0][random.random()>0.2] for n in range(100)]

#age
x = np.arange(20,90)
pmf = poisson.pmf(x, 72)
```

```
age = [random.choices(x, weights = pmf, k = 1)[0] for n in range(100)]
x = np.arange(20,90)
pmf = poisson.pmf(x, 55)
age2 = [random.choices(x, weights = pmf, k = 1)[0] for n in range(100)]
shadow_df['age'] = age + age2
```

Now, we split the shadow data into two. We will use one set to train the shadow model

In [9]:
```
shadow_train_x, shadow_test_x, shadow_train_y, shadow_test_y = train_test_split(
    shadow_df, shadow_response, test_size=0.5
)
```

And train the shadow model...

In [10]:
```
# Train a model
prng = np.random.RandomState(12)
shadow_svc = SVC(C=1, gamma=3, probability=True, random_state=prng)
shadow_svc.fit(shadow_train_x, shadow_train_y)
```

Out[10]:
```
SVC(C=1, gamma=3, probability=True,
    random_state=RandomState(MT19937) at 0x159E5341840)
```

The attacker now passes the portion of shadow data used for training, and the portion used for testing through the trained shadow model to extract the model's confidence. For each example, the attacker just needs the highest of the two values (reponse confidence or non-response confidence, whichever is larger). A quick look at the average of these values for the two sets tells us that the shadow model assigns much higher confidence to training examples than non-training examples

In [11]:
```
train_preds = shadow_svc.predict_proba(shadow_train_x).max(axis=1)
test_preds = shadow_svc.predict_proba(shadow_test_x).max(axis=1)
print(f'Mean of confidence for training examples = {train_preds.mean():.2f}')
print(f'Mean of confidence for non-training examples = {test_preds.mean():.2f}')
```

```
Mean of confidence for training examples = 0.92
Mean of confidence for non-training examples = 0.54
```

The attacker now knows the kind of confidence values that their shadow model gives to training and non-training examples. They're confident that the data they generated is similar enough to the original data, and that their model is confifgured similarly to the original model (remember that the researcher released this information) to assume that these confidence values are similar to those that the original model would give on training and non-training data. They can therefore use them as a baseline against which to compare the the value they got when they presented the MP's data to the original model.

In [16]:
```
print(f'Maximum confidence for MP in original model: {mp_preds.max():.2f}')
```

```
Maximum confidence for MP in original model: 0.84
```

The attacker can do this comparison in a number of ways. Here we will assume that the attacker trains another ML model (the *attack* model) to distinguish between these two sets of confidences. We use a LogisticRegression model (a very simple classifier), but the attacker could use anything.

In [13]:
```
from sklearn.linear_model import LogisticRegression
lr = LogisticRegression()
```

```
train_x = np.vstack((train_preds[:, None], test_preds[:, None]))
train_y = np.hstack((np.ones(len(train_preds)), np.zeros(len(test_preds))))
lr.fit(train_x, train_y)
```

Out[13]:

```
LogisticRegression()
```

The attacker now passes the maximum confidence value they got from the original model with the MPs data to this new model to obtain a prediction as to whether or not it was in the training data. Note that the prediction takes the same form as previous predictions: two confidence values. One the confidence of it not having been in the training set, another the confidence that it was:

In [14]:

```
input_array = np.array([[mp_preds.max()]])
prob_membership = lr.predict_proba(input_array)
print(f'Confidence of non-membership = {prob_membership[0][0]:.2f}')
print(f'Confidence of membership = {prob_membership[0][1]:.2f}')
```

```
Confidence of non-membership = 0.20
Confidence of membership = 0.80
```

The model is making a strong prediction that the MP *was* in the training data, and therefore the attacker concludes that they *did* have Cancer. This prediction is correct.

Mitigation

Mitigating this kind of attack involves configuring the classifier to not give different confidences to examples that it has been trained upon. In this case, decreasing the SVMs `gamma` parameter will have a strong effect. For example, here is what happens in the attack if the original model's `gamma` is reduced from 3 to 0.1

In [17]:

```
prng = np.random.RandomState(12)
svc = SVC(C=1, gamma=0.1, probability=True, random_state=prng)
svc.fit(df, response)

mp_preds = svc.predict_proba(mp_data)
print(mp_preds)

shadow_svc = SVC(C=1, gamma=0.1, probability=True, random_state=prng)
shadow_svc.fit(shadow_train_x, shadow_train_y)

train_preds = shadow_svc.predict_proba(shadow_train_x).max(axis=1)
test_preds = shadow_svc.predict_proba(shadow_test_x).max(axis=1)
print(f'Mean of confidence for training examples = {train_preds.mean():.2f}')
print(f'Mean of confidence for non-training examples = {test_preds.mean():.2f}')

lr = LogisticRegression()
train_x = np.vstack((train_preds[:, None], test_preds[:, None]))
train_y = np.hstack((np.ones(len(train_preds)), np.zeros(len(test_preds))))
lr.fit(train_x, train_y)


input_array = np.array([[mp_preds.max()]])
prob_membership = lr.predict_proba(input_array)
print(f'Confidence of non-membership = {prob_membership[0][0]:.2f}')
print(f'Confidence of membership = {prob_membership[0][1]:.2f}')
```

```
[[0.15803859 0.84196141]]
Mean of confidence for training examples = 0.93
Mean of confidence for non-training examples = 0.90
Confidence of non-membership = 0.52
Confidence of membership = 0.48
```

The attack is now almost completely ambiguous, providing the attacker with no information as to whether or not the MP was in the training set.

The technical effect `gamma` has on the SVM is unimportant -- the important point is that changes to the model's configuration can play a significant role in its vulnerability.

Conclusions

This example has shown how an attacker can perform a membership inference attack to determine that a well-known individual was in a model's training data.

It is hopefully clear that this is non-trivial -- the attacker has to put in quite a lot of effort. Their success is also contingent on them knowing certain things about the problem. In particular:

1. Enough information about the original data that they can generate a shadow dataset. This will be things like: types of variables, ranges of variables, distributions of variables. Such information is often available at population levels (e.g. average age, proportion of population with diabetes etc).
2. Configuration information about the original model. In this case, it was the parameters that define the model and, in particular a parameter called `gamma` that is used in the SVM. It is quite common for researchers to publish this information.
3. The input values for the individual in question. This is harder to come by, but for famous individuals, it's conceivable that a lot of this information might be in the public domain.