Machine Learning Report - "*Titanic*"

Have you ever wondered whether you would survive the sinking of the Titanic?
In this report we will discuss the model that predicts this outcome.

The data that we' re going to use is stored in the file titanic-train.txt, for training, and titanic-test.txt for testing purposes.

Here is a sample of the dataset:

titanic-train.txt

```
 1 2 0 32 2 0 73.5 0
 2 2 1 4 1 1 23 1
 3 2 0 27 0 0 26 0
 4 2 0 34 1 0 21 0
 5 3 1 4 0 1 13.4167 1
 6 1 0 25 1 0 55.4417 1
 7 3 0 45 0 0 8.05 1
 8 3 0 22 1 0 7.75 0
 9 3 0 16 1 3 34.375 0
10 3 1 22 0 0 7.8792 1
```

Each row represents a passenger. Each row contains 6 features - excluding the first (index) and last (label) columns - indicating survived (1) or not (0) .
The features are informing us about:

- the ticket class (1st, 2nd, 3rd)
- sex (0 -> male, 1 -> female)
- age in years
- number of siblings and spouses aboard
- number of parents and children aboard
- the passenger fare

1. We start with loading the training data.

```python
import torch
import matplotlib.pyplot as plt


f = open("titanic-train.txt")
data = [float(x) for x in f.read().split()]
f.close()

data = torch.tensor(data).view(-1, 7)

X = data[:, :6]
Y = data[:, 6].long()

print(X.shape, X.dtype)
print(Y.shape, Y.dtype)
```

We use list comprehension in order to “unpack” the data. This one leaves us with array of numbers in a sequence, so passengers are mixed up, but we can reshape it with the .view(), when converting input to the tensor.
We pass two values to make it a matrix. Second value 7, tells us about how many columns do we want to have (6 features + 1 label), where the first argument just adjusts the number of rows, with respect to the other argument.

Then we divide it into two collections: X and Y -> X contains columns 0 to 5 (without 6 itself). Y gets 6th, because notice that we index from 0, so we have to shift.

2. Our model is based on logistic regression.
It is a simple formula, but works well with simple binary classification.

```
def logreg_inference(w, b, X):
    return torch.sigmoid(X @ w + b)
```

We will perform dot product of matrix and vector, which will produce a vector, and add b which is the scalar. But how is it even possible? Yes, because pytorch provide us with a function called broadcasting. So the value in b will be broadcasted to every single value located in the X@w.
We apply sigmoid in order to squash our output in range 0 and 1.

```
y_pred = logreg_inference(w, b, X)
y_pred = y_pred.clamp(0.001, 0.999)
```

Later on in the training loop, we will use clamp in order to limit prediction output. Minimum value is 0.001 and maximum one is 0.999.
It is done to prevent numerical instability in the loss function - Binary Cross Entropy. Which has formula:

$$BCE\ LOSS\ =\ \frac{1}{N}\sum_{i=0}^{N-1} -y_i\ \cdot\ log(\hat{y}_i)\ -\ (1\ -y_i)\cdot log(1\ -\ \hat{y}_i)$$

So as you see obtaining 0 or 1 for the prediction will give us fluctuations -> log(0) and log(1-1).

The optimizer is based on SGD - Stochastic Gradient Descent.
Our dataset is not that big so there is no point in dividing it into the batches - we will take into consideration one passenger at the time.

Formula for SGD on w example looks like this:

$$w_{i+1} = w - (LR \cdot \frac{dLOSS}{dw})$$

The same we will be calculating for b, with respect to b.

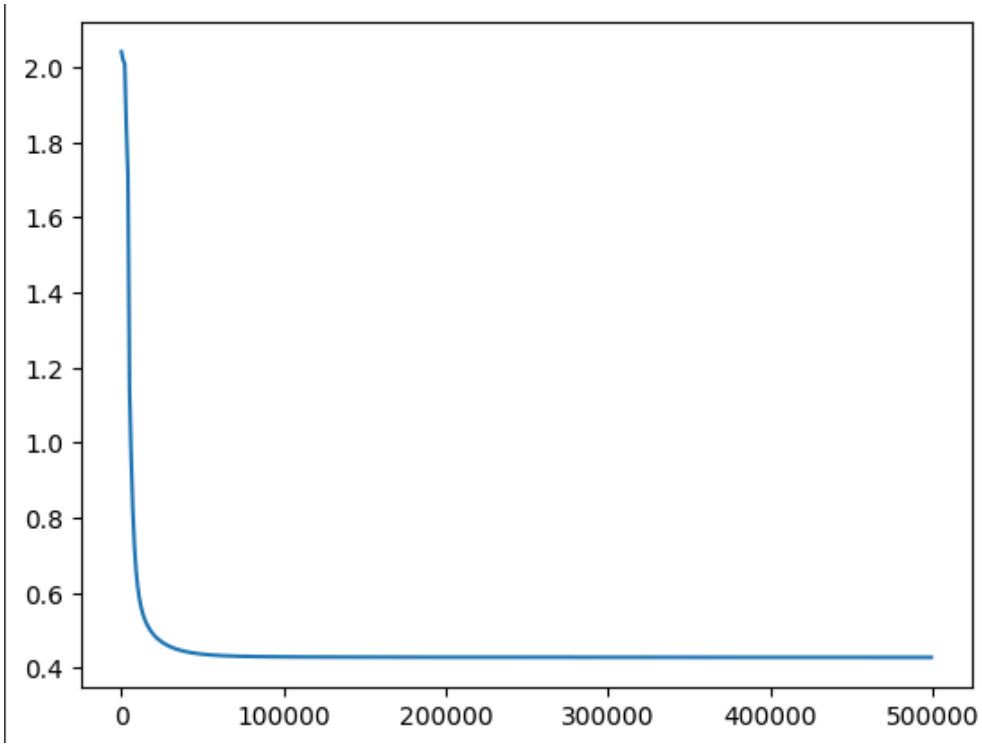
The training loop looks like this:

```
LR = 0.001
STEPS = 500000

w = torch.randn(6, requires_grad=True)
b = torch.randn(1,  requires_grad=True)
print(w)
print(b)
optimizer = torch.optim.SGD([w, b], lr=LR)
loss_fun = torch.nn.BCELoss()

losses = []
steps = []

for step in range(STEPS):
    optimizer.zero_grad()

    #prediction
    y_pred = logreg_inference(w, b, X)
    y_pred = y_pred.clamp(0.001, 0.999)

    loss = loss_fun(y_pred, Y.float())

    loss.backward()
    optimizer.step()
    accuracy = (y_pred > 0.5).long()
    accuracy = (accuracy == Y).float().mean()
    if step % 1000 == 0:
        print(step, loss.item(), (accuracy.item() * 100))
        losses.append(loss.item())
        steps.append(step)
```

Important thing is to zero the gradient in every single step, because we would have accumulated the value.
Another important thing is to set requires_grad = True in both - w and b, since we want to calculate the gradient of loss, with respect to those values.

With learning rate 0.001 after 500 000 steps, loss is set to around 0.428
and accuracy to 80.704%.
Plot presents that model could learn in first 20-30k steps, and later on not major changes on the loss are presented. The simplicity and linearity of the model may be the reason.

Plot of loss with respect to number of epochs.



3. So, would you survive on Titanic?
   To use the model we have to create a vector of values with the individual preferences. In my case this vector looks like this: vector_me = [1, 0, 21, 1, 0, 23].

   Reminder about the features:
   class ticket, gender, age, number of siblings/spouses aboard, number of parents/children, the passenger fare

   We only need to transition vector to tensor and specify the data type, and we can call a trained model on the input.
   For me model specified that the probability of survival is 53.2117%, so not big, but maybe I still have a chance since the model's accuracy is around 80% :).

```
vector_me = [1, 0, 21, 1, 0, 23] # class, gender, age, siblings&spouses, parents&children, fare
vector_me = torch.tensor(vector_me, dtype=torch.float32)
outcome = logreg_inference(w, b, vector_me)
print(f"Probability of survival: {outcome.item():.4f}")
```

But speaking of accuracy.. To calculate training accuracy, we call our trained model on X, passing adjusted w and b. The result is just and array of probabilities. We convert them to boolean values (those higher than 0.5 are True - the survivors, those below False). But we remember that initial labels, that we want to compare our predictions with, are 0's or 1's. That is why we transform boolean values with .long().

The final part is straightforward - we measure accuracy by comparing an array of predictions with actual labels. This comparison produces an array of booleans, so we convert it to floats, so we can perform mean on that - to perform means it is obligatory to change to float.

That leaves us with 80.4225% of accuracy. Not bad for the simple logistic regression model.

```
[21] y_pred = logreg_inference(w, b, X)
     predictions = (y_pred > 0.5).long()
     accuracy = (predictions == Y).float().mean()
     print(f"Training accuracy: {(accuracy * 100):.4f}%")

     Training accuracy: 80.4225%
```

But what do those learned weights actually tell us about the individual features?
Can we extract any information out of that?
The answer is yes we can! Let's see the w parameters first.

```
[33] for value in w:
       print(f"Weight w: {value.item():.4f}")


     print(f"Weight b: {b.item():.4f}")

     Weight w: -1.2412
     Weight w: 2.7938
     Weight w: -0.0447
     Weight w: -0.3291
     Weight w: -0.0902
     Weight w: 0.0012
     Weight b: 2.6097
```

So how do we look at it? It is simple - we check whether increasing the feature takes us towards one or zero. If it's one, the feature enhances your chances of survival.
In our case:

- lower class number, higher chance of survival
- female much bigger chance of survival (2.7938 rapidly increases survival probability)
- younger people more likely to survive
- fewer siblings/spouses the bigger the chances of survival
- fewer parents/children the bigger the chances of survival
- higher fare gives slightly bigger chances

From our training dataset, let's see who had the biggest chance of survival.
We are using .argmax() on our prediction list -> this will obtain the index of the tensor. That is why we can put it on the X dataset in order to find it.

In order to find the value, let's use this index on the prediction list.

```
[48] print(f"The highest chances of survival: {X[y_pred.argmax(0), :]}")

     print(f"{(y_pred[y_pred.argmax(0)].item() * 100):.4f}%")

The highest chances of survival: tensor([  1.0000,   1.0000,   2.0000,   1.0000,   2.0000, 151.5500])
97.7060%
```

So we can see that the biggest chances of surviving had 2 years old girl, having a 1st class ticket, having one sibling and two parents on the ship. And the fare cost her 151.5500. She had 97.7060% of surviving. Fair enough.

Let's do the opposite now and check who was the unluckiest.

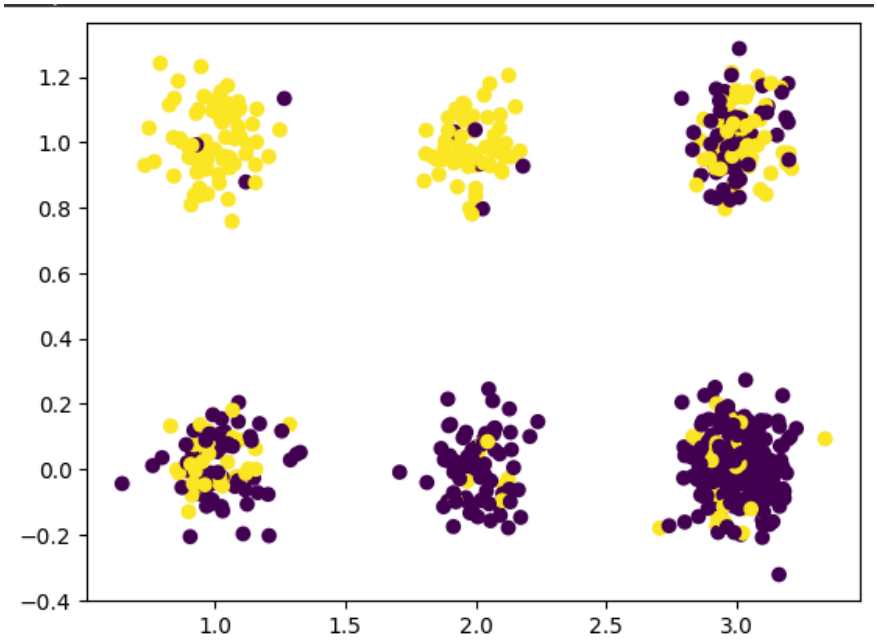
```
print(f"The lowest chances of survival: {X[y_pred.argmin(0), :]}")

print(f"{(y_pred[y_pred.argmin(0)].item() * 100):.4f}%")

The lowest chances of survival: tensor([ 3.0000,  0.0000, 20.0000,  8.0000,  2.0000, 69.5500])
0.8705%
```

So we can see that the unluckiest person on titanic would be:
20 years old man, that purchased 3rd class ticket, had a whole 8 siblings&spouses, two parents or children and have paid 68.5500. He had not even 1 percent of surviving - 0.8705%.

Lets see the correlation between two the most influential features now - gender and the class ticket. Before drawing the plot, we want to add a little bit of fluctuation,

so overlapping points will be more visible.

```
noisy_X = X + torch.randn_like(X) * 0.1
plt.scatter(noisy_X[:, 0], noisy_X[:, 1], c = Y)
```



Y axes represent the gender (1 female, 0 male), and X axes represents the class of the ticket. Yellow color represents survivors.
Like we would have thought female with the first class ticket contains a lot of survivors, and male passengers with 3rd class tickets have much more non - survivability.
In the second class ticket, both passengers have mixed outcomes, but man still much more non - survivability. It shows how strong a feature a gender is.

Let's evaluate the model on the test data right now. We follow the same steps as we did with a training data set, but on titanic-test.txt.
We are going to start with the test accuracy. We will follow the same steps as we did with training accuracy - the only difference is that this time we will run our model on test dataset.

```
[71] y_pred = logreg_inference(w, b, X_Test)
     predictions = (y_pred > 0.5).long()
     accuracy_test = (predictions == Y_Test).float().mean()
     print(f"Test accuracy: {(accuracy_test * 100):.4f}%")

     Test accuracy: 79.6610%
```

It leaves us with 79.6610% of accuracy. It is very similar to the one that we gained on the training dataset. The difference is only 0.7615% on the training advance. That is the sign that model is not overfitting and it generalizes well on unseen data.

To enhance that model we would think that maybe more steps we would have to acquire to make the training process longer. But as we saw from the plot model was not doing progress after 20-30k steps, and even if then the progress was unnoticeable. Adjusting hyperparameters may not be a solution that will change a lot.
We could try to add more features, which model could learn from, like cabin position or access to the lifeboat.

To enhance the performance even more we could try to use different models, that would break a linearity, like a feedforward network with hidden layers.

“I affirm that this report is the result of my own work and that I did not share any part of it with anyone else except the teacher."
Wiktor Andrzejewski