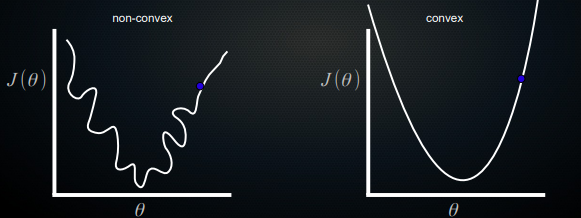
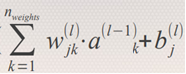
**Practice questions answers ML Basics resit**

Q1.



*In gradient descent, we use gradients (specifically, partial derivatives of the cost function that tells us how wrong we are) to find the current step we can take that best minimises the cost function. By making many of these small steps, we hope to reach a global minimum. Remember that in reality, unlike the cartoons I just drew, we have no idea how the complex multi-dimensional cost function really looks! We simply start somewhere and start minimising. If the cost function is non-convex, you can imagine that as troughs or valleys which you do not cross, local optima that you get stuck in. This is not good!*

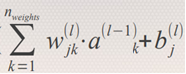
Q2.



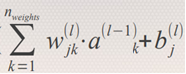
*(it’s fine if you didn’t have the j, that’s because there may be multiple neurons in a layer; I would like the k (or any letter you used for indexing) because every activation from the previous layer needs its own weight. You’d get ~40% of points if you didn’t use any indices).*

*You can tweak the bias, the weights, and the activations of the previous layer (save for in the first layer after the input layer).*

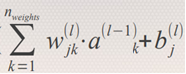
*Partial derivatives:*

∂  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ = 1

∂b

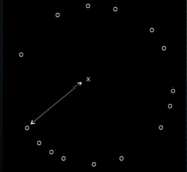
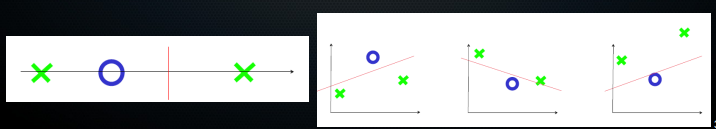
∂  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ = w(l)jk (so how the weighted sum changes with the activations of the

∂a previous layer is the weights, which makes sense)

∂  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ = a(l-1)

∂w

Q3.

1. *As the number of dimensions rises, basically all points lie extremely far from each other. An intuitive explanation is that if you just add random normally distributed features, the more features you add, the more chance that you have an extreme value in one of these, and a single extreme value already puts you far away (on one axis) from the center of the multidimensional space.   
   You could also say that, as the number of dimensions rises, the huge amounts of noise (squared distances in dimensions where points don’t really differ but have a slightly different value by chance) overrules any real signal (dimensions where points really have marked differences).   
   In the lectures, we also saw an explanation showing how higher-dimensional spheres occupy less and less of the volume of a hypercube, meaning that the space ‘average’ data points occupy relative to the volume there is for extreme points becomes extremely small. And if everything is extremely far away from each other, then it doesn’t really makes sense to talk about how far you are away from other points anymore: you’re not close to anyone at all in a high-dimensional space. An illustration in 2D is this circle of points:  Yes,one is infinitesimally closer to the middle point. But is that really a useful determinant of closeness? No.  
   (****Note:*** *I expect either of these 3 possible answers, or some combination. Ideally you have some elements of 1 and 2, and some elements of 3, since they are different ways of explaining it).*
2. *As you increase dimensionality, the amount of possible feature combinations rises exponentially. You run into huge data sparsity issues. If you have one feature, you need to sample 20% of the possible feature values to, well, cover 20% of the possible feature space. But for 2 dimensions, you need 45% of possible feature 1 values, and 45% of possible feature 2 values, to cover 20% of the total possibilities, since 0.45^2 = 0.2 (20%). This problem only gets worse. ML methods rely on statistics: if you cover less and less of the possibilities, you can much more easily make spurious decision boundaries that work perfectly on your training data. Not because you learn a meaningful function, but because by chance undersampling that decision boundary works perfectly! This is illustrated here, where adding a completely random second feature immediately gives perfect opportunity to separate 3 samples (but would break down for more samples since the 2nd feature is totally random!):  
     
   (****Note:*** *this answer needs at least the notion that with equal training data your data becomes so sparse that you can easily fit perfect functions that nevertheless are useless in reality because you are hugely overfitting to (statistical noise in) the training data.****NOTE:*** *What I wrote here is an* ***extremely complete*** *example, you can do with about half of the detail in the test and still get full points, I think).*

Q4.   
*1. As we have seen in the lectures, if there’s, say, 20 negative samples and 80 positive samples, you can get an accuraccy of 80% by simply classifying everything as a positive sample. The classifier has then learned basically nothing, and moreover, if we care at all about knowing which cases are negative our classifier’s performance in that respect is 0. Instead, we can use something like sensitivity and specificity, i.e. metrics involving true positives, false positives, true negatives and false negatives to get much better insight into how our classifier is doing. An example of a metric that incorporates both into 1 value is the F1 score (****Note:*** *this final sentence is really optional, you don’t need to know that metric by heart!)*

*2. The key word here is thresholds. Classifiers never give 0 or 1 answers, instead they give values between 0 and 1 and we use a decision threshold to determine where to make the cut-off between what we interpret as 0 or as 1. Your classifier’s performance is thus based on this threshold. This threshold is something you choose based on the application you want for your classifier, e.g. do you want to only count as positive cases ones that you are very sure about (like for chemo treatment classification)? Or are you more permissive? By using metrics like ROC AUC, you can assess classifiers independent of these thresholds, simply by averaging performance over all (or at least many) thresholds. Otherwise, comparing classifiers that use different thresholds would be like comparing apples to oranges.*

Q5. Sigmoid(

|  |  |  |
| --- | --- | --- |
| 1 | 12 | 38 |
| 1 | 3 | 49 |
| 1 | 9 | 67 |

@

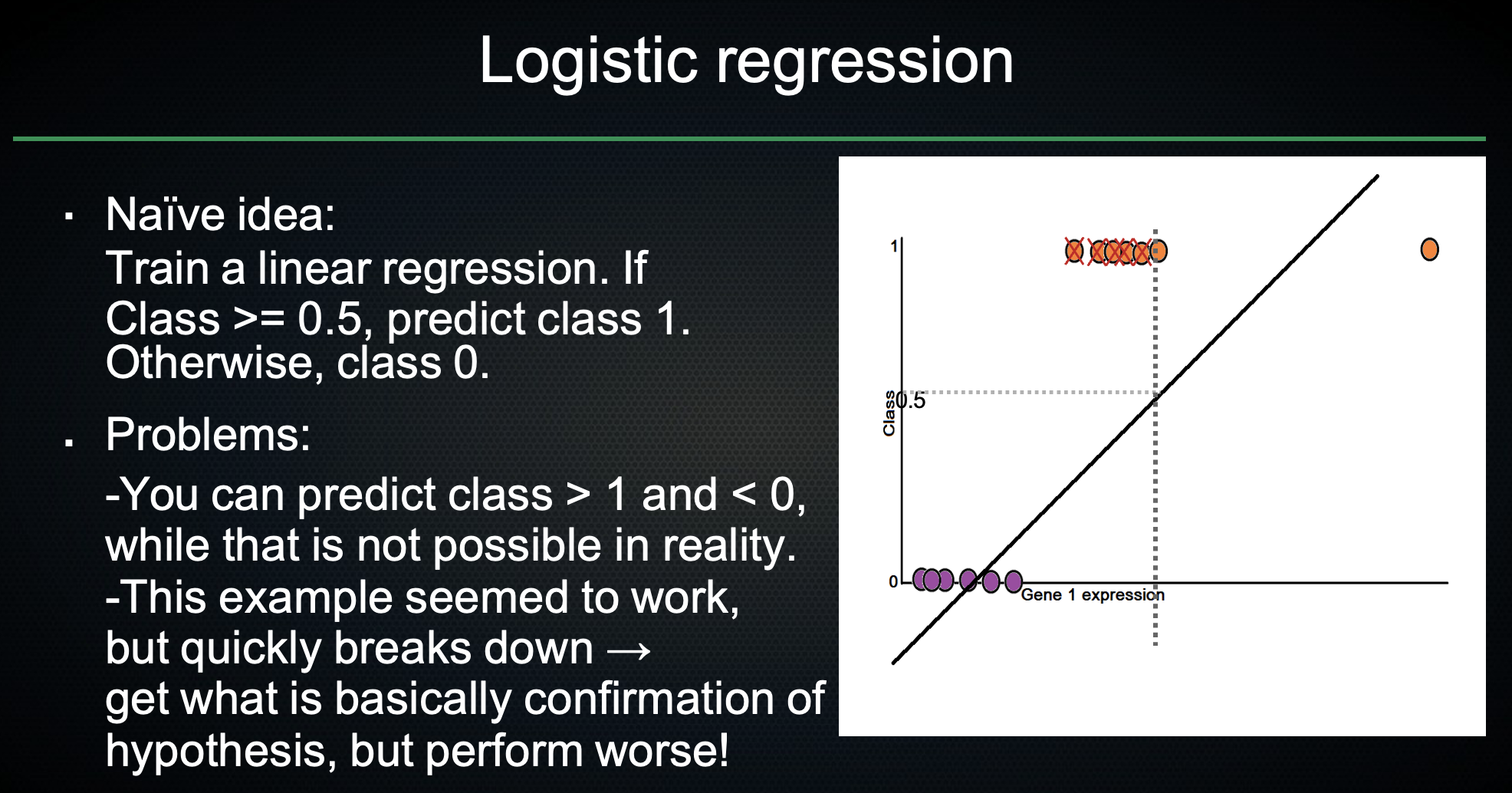
|  |
| --- |
| Theta\_0 |
| Theta\_1 |
| Theta\_2 |

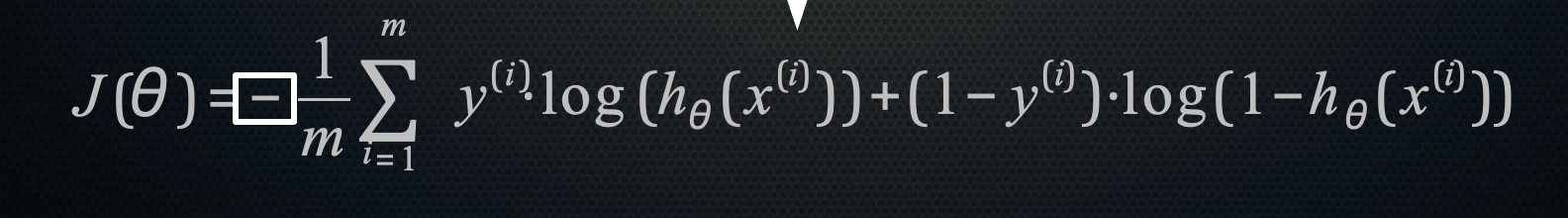
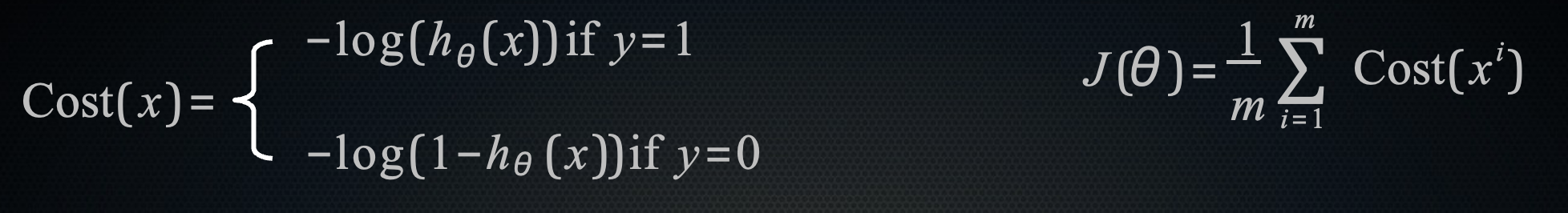
)

=

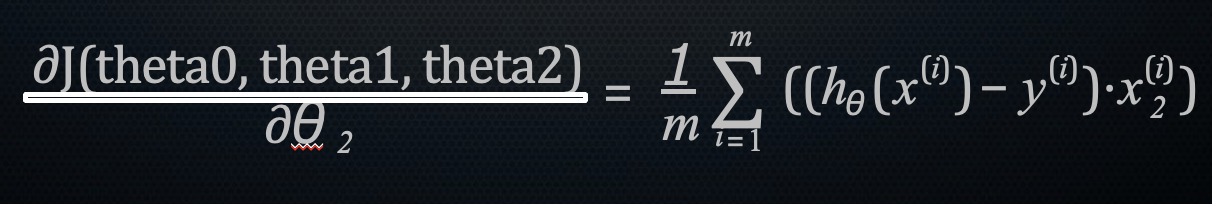
(3 by 3 \* 3 by 1 = 3 by 1)

|  |
| --- |
| Hypothesis sample1 |
| Hypothesis sample 2 |
| Hypothesis sample 3 |

2. *If we do that we can predict values that are less than 0 or more than 1, which doesn’t make sense for classification. Additionally, while you might think you could fit a linear regression anyway and just put a threshold, we saw that this line starts misclassifying if you get an observation that confirms your earlier ‘classification’*  


3.   
  
Or:  


*The partial derivative is, as we have learned, of the exact same form as those for linear regression. So it will look like:*

  
(Note: I would consider this a difficult question)

*4. The area under the curve of the precision-recall curve.*

Q6.

*We assess the expected generalisation performance of our classifier (that is, how well we think it will do on unseen data when we actually deploy or use it) by cross-validation. If we simply split our data into k folds, we are assuming that the data we will see in the wild is exactly like our training data in this way. But: clinical practice might change, or the patient population or samples that your classifier is asked to work on changes: maybe it is deployed in a different country. Ideally, we want to already test robustness to these types of shifts in the data. One way to do this is to go to leave-one-study-out cross-validation, where you train on data from k-1 studies (say different microarray experiments for gene expression, or patient populations you follow in different hospitals/countries). Then you’re not satisfied unless your classifier does sort of well on these disparate populations that have different data biases. You are using a more realistic situation for your expected generalisation performance. (In the end, of course, you would train your final classifier on all the data).*

Q7.

***(If you want to do this with a loop, that is fine by me). Note: gradient descent is a quite difficult one, I probably won’t ask exactly this on the exam. But some pseudocode question will be asked. This is not the most neat answer but is good enough I think. You can do it with more looping or more linear algebra or whatever you want. Again, the focus is on showing me the logical steps of calculating predictions, getting errors, calculating the partial derivatives, and making a step in the parameters. I don’t care about correct Python code etc.***

*You will need X, y, thetas, and alpha*

*#I assume these things are numpy arrays*

*Theta\_0, Theta\_1 = thetas[0], thetas[1]*

*N\_samples = length of X*

*Predicted = X\_with\_a\_column\_of\_ones\_in\_front @ thetas*

*Error\_for\_part\_deriv = pred-y*

*#now I have a column of errors*

*#partial derivatives are:*

*#Theta\_0 = 1/m \* sum(1 to m) (pred – y) \* 1  
#Theta\_1 = 1/m \* sum(1 to m) (pred – y) \* x*

*Theta\_0\_part\_deriv = 1/N\_samples \* sum(Error\_for\_part\_deriv \* 1)*

*Theta\_1\_part\_deriv = 1/N\_samples \* sum(Error\_for\_part\_deriv \* x)*

*Theta\_0\_new = Theta\_0 – alpha \* Theta\_0\_part\_deriv*

*Theta\_1\_part\_deriv = Theta\_1 – alpha \* Theta\_1\_part\_deriv*

*Done.*