CS 4100: Artificial Intelligence (Fall 2023) Lawson L.S. Wong Northeastern University
Due November 13, 2023

Exercise 4: Reinforcement Learning and Supervised Machine Learning

Please remember the following policies:

ˆ Exercises are due at 11:59 PM Boston time (ET).

ˆ Submissions should be made electronically on Canvas, as a single .pdf file. You can make as many submissions as you wish, but only the latest one will be considered, and late days will be computed based on the latest submission.

ˆ Each exercise may be handed in up to one day late (24-hour period), penalized by 10%. Submissions later than this will not be accepted. There is no limit on the total number of late days used over the course of the semester.

ˆ Solutions may be handwritten or typeset. For the former, please ensure handwriting is legible. If you write your answers on paper and submit images of them, that is fine, but please put and order them correctly in a single .pdf file. One way to do this is putting them in a Word document and saving as a PDF file.

ˆ You are welcome to discuss these problems with other students in the class, but you must understand and write up the solution yourself, *and* indicate who you discussed with (if any). If you are collaborating with a large language model (LLM), acknowledge this and include your entire interaction with this system. We strongly encourage you to formulate your own answers first before consulting other students / LLMs.

ˆ Contact the teaching staff if there are *extenuating* circumstances.

1. 1 point. (Reinforcement Learning: An Introduction 6.2) *Temporal difference.*
Recall the commute-time prediction example discussed in Lecture 13. A generic model-free estimate (also referred to as the "Monte-Carlo method" below) makes updates based on the difference between the *final out come* and the current prediction, whereas a temporal-difference estimate makes updates based on the difference between the *next prediction* and the current prediction, as illustrated in the figure below.
Note: For full details of the example, refer to Example 6.1 of *Reinforcement Learning: An Introduction* (p. 122–123; also copied on the final page for reference).

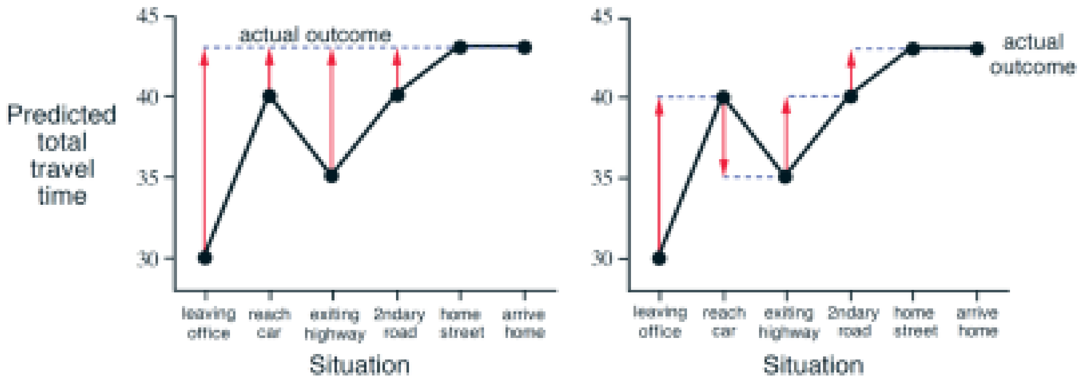


Figure 6.1: Changes recommended in the driving home example by Monte Carlo methods (left) and TD methods (right).

This is an exercise to help develop your intuition about why TD methods are often more efficient than Monte Carlo methods. Consider the driving home example and how it is addressed by TD and Monte-Carlo methods. Can you imagine a scenario in which a TD update would be better on average than a Monte-Carlo

update? Give an example scenario – a description of past experience and a current state – in which you would expect the TD update to be better.
Hint: Suppose you have lots of experience driving home from work. Then you move to a new building and a new parking lot (but you still enter the highway at the same place). Now you are starting to learn predictions for the new building. Can you see why TD updates are likely to be much better, at least initially, in this case? Might the same sort of thing happen in the original scenario?

Consider the game Minecraft, in which the general progression of the game is usually the same. First, get create wood tools, then cobblestone, iron, and finally diamond tools. Create a Nether portal. Beat the ender dragon -> Win!

In your past experience, you know that after you have made iron tools, it is estimated that it will take 4 more hours to win the game, after diamond tools 2 more hours, etc...

However, the seed of every game is different, and thus you start in a different environment every time. Using Monte-Carlo strategies, you would need to wait until you beat the game to update your predictions, a time consuming endeavor. TD methods, on the other hand, would update during the playthrough with live information(I.E. took you 5 hours to find iron instead of 1, therefore other predictions shift as well).

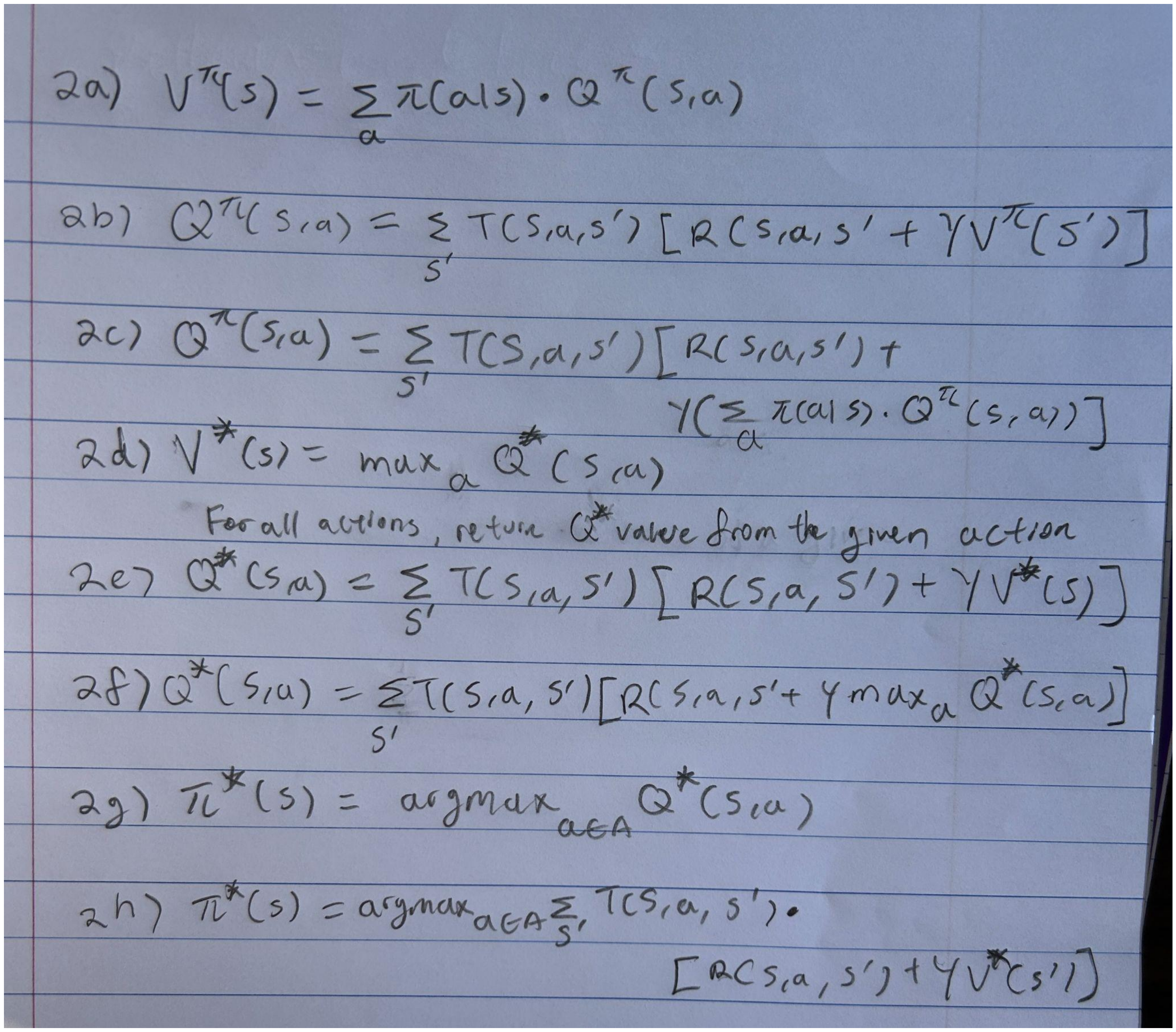
2. 3 points. (Reinforcement Learning: An Introduction 3.12, 3.13, 3.17, 3.25–3.29) *Fun with Bellman.* In Ex3 Q4, recall that we stated the Bellman equation for deterministic policies and state-only reward

and the more general version of the Bellman equation for stochastic policies and general reward functions

In this question, you will derive similar mathematical identities for Q^π, Q^*, and π^* for stochastic policies $\pi(a|s)$ and general reward functions $R(s, a, s')$.
Note: Not all identities are recursive / "Bellman-like".

(a) Give an equation for V^π in terms of Q^π and $\pi(a|s)$.
(b) Give an equation for Q^π in terms of V^π, $T(s, a, s')$, and $R(s, a, s')$.
(c) Derive the general Bellman equation for Q^π, in terms of $\pi(a'|s')$, $T(s, a, s')$, $R(s, a, s')$, and $Q^\pi(s', a')$.
(d) Give an equation for V^* in terms of Q^*.
(e) Give an equation for Q^* in terms of V^*, $T(s, a, s')$, and $R(s, a, s')$.
(f) Derive the general Bellman optimality equation for Q^*, in terms of $T(s, a, s')$, $R(s, a, s')$, and $Q^*(s', a')$.
(g) Give an equation for π^* in terms of Q^*.
(h) Give an equation for π^* in terms of V^*, $T(s, a, s')$, and $R(s, a, s')$.

All answers are found in the image below:



2a) $V^{\pi}(s) = \sum_{a} \pi(a|s) \cdot Q^{\pi}(s,a)$

2b) $Q^{\pi}(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s' + \gamma V^{\pi}(s')]$

2c) $Q^{\pi}(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma(\sum_{a} \pi(a|s) \cdot Q^{\pi}(s,a))]$

2d) $V^{*}(s) = \max_{a} Q^{*}(s,a)$

For all actions, return Q^{*} value from the given action

2e) $Q^{*}(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^{*}(s)]$

2f) $Q^{*}(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s' + \gamma \max_{a} Q^{*}(s,a)]$

2g) $\pi^{*}(s) = \operatorname{argmax}_{a \in A} Q^{*}(s,a)$

2h) $\pi^{*}(s) = \operatorname{argmax}_{a \in A} \sum_{s'} T(s,a,s') \cdot [R(s,a,s') + \gamma V^{*}(s'))$

3. 4 points. *Feature representation for approximate Q-learning.*
If you need an example on performing approximate Q-learning, see UC Berkeley CS 188 Section 5 Q1:
http://ai.berkeley.edu/sections/section_5_solutions_vVBDODDiXcVEWausVbSZ7eZgSpAUXL.pdf

We would like to use a *Q*-learning agent for Pacman (as you did in PA3), but the state space for a large grid is too massive to hold in memory. To solve this, we will switch to *feature-based* representation of the Pacman game state (similar to PA3 Q10), where we assume that *Q*(*s, a*) can be expressed as a *(weighted) linear combination*
of *p* state-action features:

(a) Suppose we design two features to represent the state (independent of actions): $f_g(s)$ is the number of ghosts within one step of Pacman, and $f_p(s)$ is the number of food pellets within one step of Pacman. Note that we do not have any features dependent on *a*. Why might this be a bad idea?

This is bad because $f_g(s)$ would produce the same value for all actions from that given state. Therefore, the action of running towards and running away from ghosts have the same value and the agent would behave incorrectly.

(b) The features defined above are inadequate because they do not depend on the action *a*. Describe two ways to fix this, one model-based (i.e., having access to $\hat{T}(s, a, s'))$, and one model-free. What are the advantages/disadvantages of each?

1. Model-based: by having access to T(hat), you have access to the probability of getting into the next state. Therefore, you can incorporate the transition dynamics into the feature presentation(I.E. create feature $f_{g'}$(s,a) representing the expected value of the distance to the nearest ghost after taking action a. You could calculate this using the transition model.
   a. Advantages:
      i. Incorporates more information about the consequence of actions
      ii. Allows for more "fine-grained" representation of the state-action space
   b. Disadvantages:
      i. Requires knowledge of the transition model, which may not always be available or may be hard to estimate accurately.
2. Model-free: the other approach would be the include action-dependent features that directly capture the impact of an action on the state. For example, $f_{g'}$(s,a) represents the change in the number of ghosts after taking action a. By making observations instead of predictions, the model can utilize temporal difference between successive states.
   a. Advantages:
      i. Doesn't require explicit knowledge of the transition model
   b. Disadvantages:
      i. May not always be accurate + would require many iterations of TD-learning to get it right

(c) Recall that the approximate *Q*-learning update rule is (there are *p* features in total):

Whereas the regular ("tabular") *Q*-learning update rule is:

These appear somewhat different, but we can show that *Q*-learning with linear function approximation is a strict generalization of the tabular case. In particular, come up with an appropriate set of features and weights such that $\hat{Q}(s, a; w)$ is an *exact* representation of *Q*(*s, a*) (i.e, equal for all $s \in S$, $a \in A$). Explain why your choice of features and weights gives an exact representation.

Additionally, using your features and weights, show that the two learning rules shown above are equivalent. *Hint*: Recall that information about values is stored in *Q*(*s, a*) in the tabular case, and in the weight vector *w* in the approximate case. *Q*(*s, a*) can be represented as a table of $|S| \times |A|$ numbers. How many elements should the weight vector *w* have to store the same amount of information?

Features: set of all state action pairs, in which f_j(s,a) returns 1 when index(s,a) = index j, and 0 otherwise
Weights: weight vector w has length |S| x |A| where each entry j is the Q value of the specific state action pair at index j.

Therefore Q(s, a) == w_j * f(s,a) = QValue * 1 = QValue

4. 7 points. *Regularized linear regression.*

Recall the problem of linear regression and the derivation of its various learning algorithms. We are given a dataset $\{(x^{(i)}, y^{(i)})\}^n_{i=1}$, where $x \in \mathbb{R}^p$(i.e., each data point has *p* features) and $y^{(i)} \in \mathbb{R}$. The hypothesis class we consider in linear regression is, for $\theta \in \mathbb{R}^{p+1}$:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \ldots + \theta_p x_p = \theta_0 + X^p\, j=1$$

$\theta_j x_j$

Consider the following alternative error function to *minimize* (while using squared-error loss):

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n} \left(y^{(i)} - h_\theta(x^{(i)})\right)^2 + \lambda\sum_{j=1}^{p} \theta_j^2$$

(a) Compared to standard linear regression, there is an extra *regularization* term: $\lambda\sum_{j=1}^{p} \theta_j^2$ From an optimization perspective, what is the role of this term, i.e, which weights (parameters) θ does it prefer, and which does it penalize, assuming $\lambda > 0$?

The role of the regularization term is the make sure that none of the individuals weights explode in significance. It prefers weights that are smaller by penalizing the value of weights^2, meaning that larger weights are penalized with polynomial significance.

(b) λ is known as a *regularization constant*; it is a *hyperparameter* that is chosen by the algorithm designer and fixed during learning.
What do you expect to happen to the optimal θ when $\lambda = 0$? $\lambda \to +\infty$? $\lambda \to -\infty$?

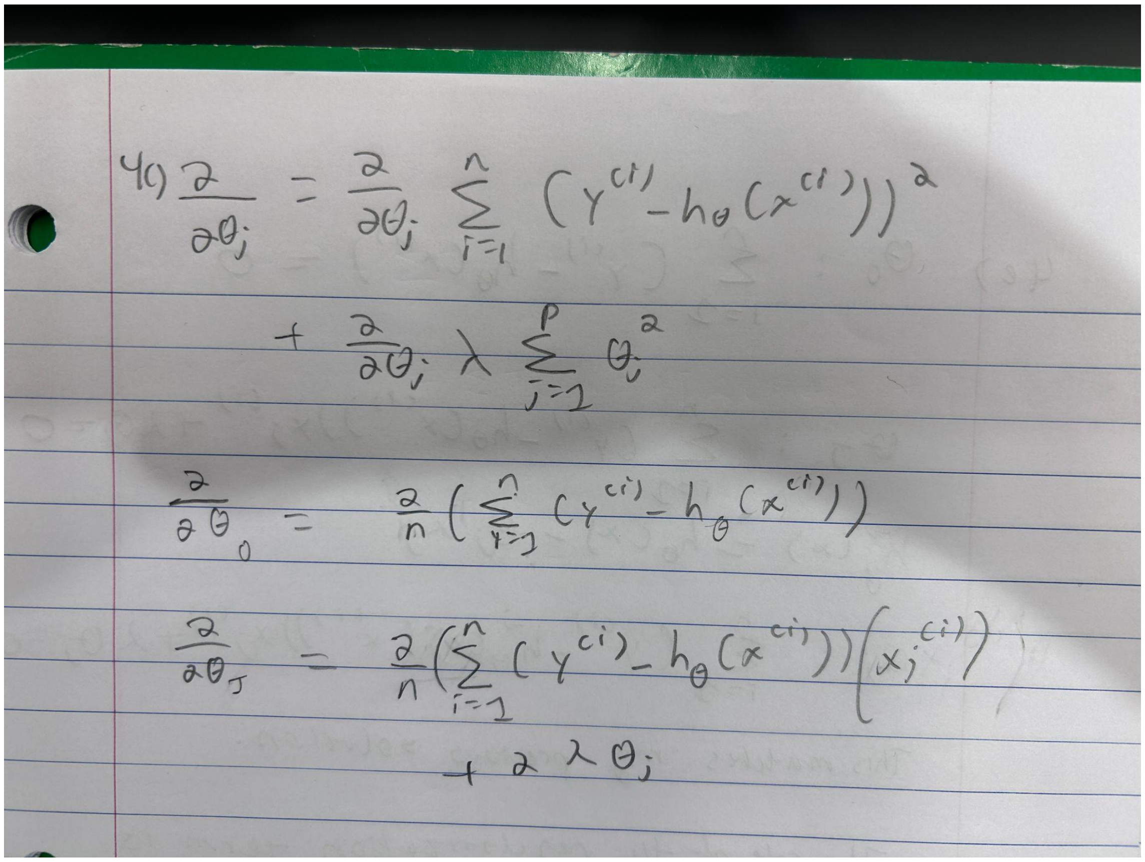
When λ=0, the regularization term has no effect, therefore the algorithm will optimize the standard squared error without any penalty for large weights. This could potentially lead to overfitting, especially if the number of features is large relative to the number of training samples.

When λ=+∞, the weights will approach a value of 0, as the regularization term will dominate the error function, potentially leading to an underfit model.

When λ=-∞: Typically, the error function is non-negative. However, because the regularization will dominate the error function with a negative term, the model will likely behave erratically. I would predict the weights to grow in value => +∞.

(c) Find the partial derivatives for weight$_0$ and weight$_j$

(These two cases are different; for the second one,
only derive once for an arbitrary index *j*.)



4c) $\frac{\partial}{\partial \theta_j} = \frac{\partial}{\partial \theta_j} \sum_{i=1}^{n} (y^{(i)} - h_\theta(x^{(i)}))^2$

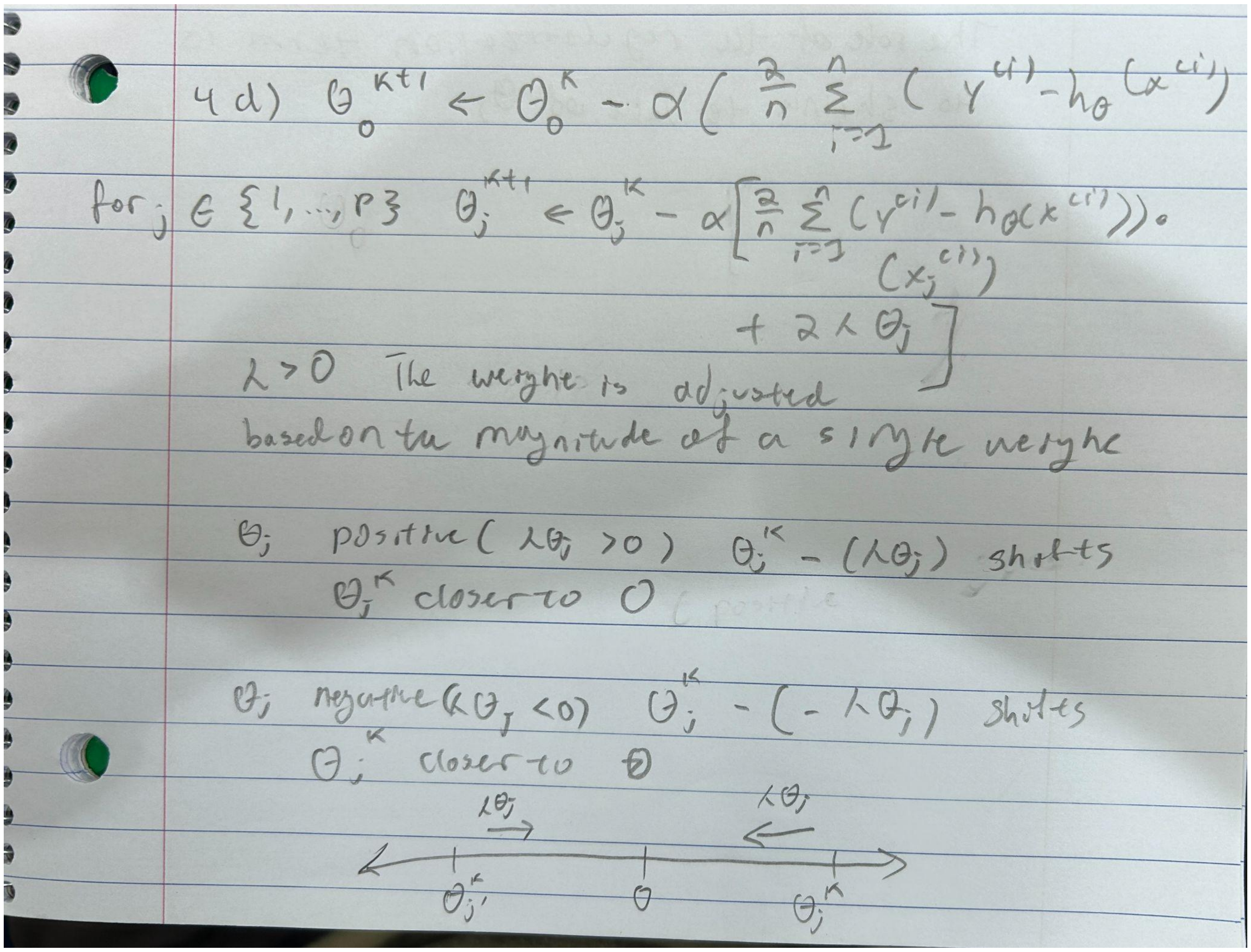
$+ \frac{\partial}{\partial \theta_j} \lambda \sum_{j=1}^{p} \theta_j^2$

$\frac{\partial}{\partial \theta_0} = \frac{2}{n} \left(\sum_{i=1}^{n} (y^{(i)} - h_\theta(x^{(i)})) \right)$

$\frac{\partial}{\partial \theta_j} = \frac{2}{n} \left(\sum_{i=1}^{n} (y^{(i)} - h_\theta(x^{(i)})) \right) (x_j^{(i)})$

$+ 2 \lambda \theta_j$

(d) Write out the update rule for gradient descent applied to the error function $J(\theta)$ above. Compare with the gradient descent update rule for standard linear regression; what is the difference? How does this difference affect gradient descent, assuming $\lambda > 0$?
Consider what happens both when θ_j is positive and when it is negative.



4 d) $\theta_0^{k+1} \leftarrow \theta_0^{k} - \alpha\left(\frac{2}{n}\sum_{i=1}^{n}\left(y^{(i)} - h_\theta(x^{(i)}\right)\right)$

for $j \in \{1, \ldots, p\}$ $\theta_j^{k+1} \leftarrow \theta_j^{k} - \alpha\left[\frac{2}{n}\sum_{i=1}^{n}\left(y^{(i)} - h_\theta(x^{(i)})\right)\right) \cdot (x_j^{(i)}) + 2\lambda\theta_j\right]$

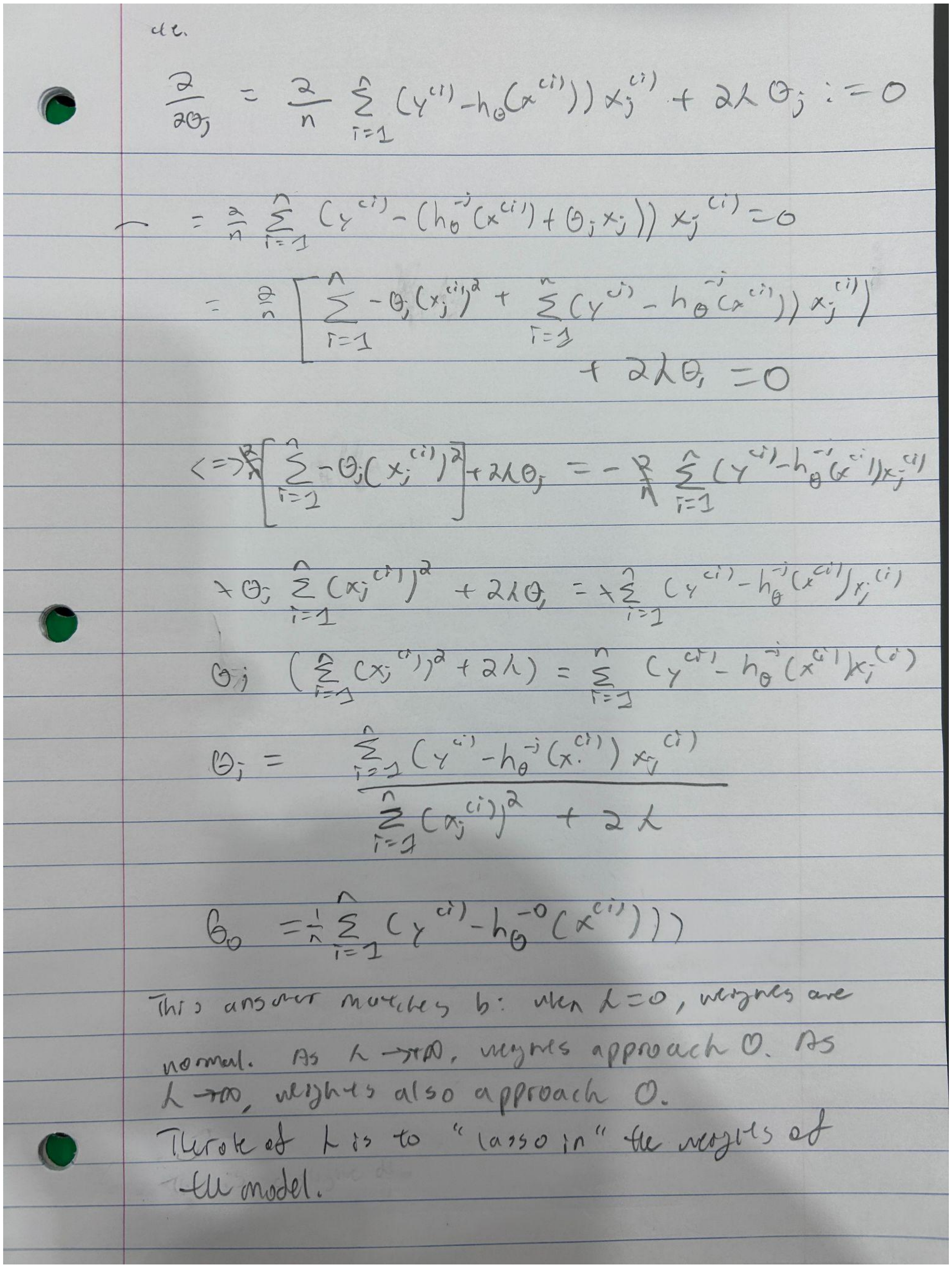
$\lambda > 0$ The weight is adjusted based on the magnitude of a single weight

θ_j positive ($\lambda\theta_j > 0$) $\theta_j^{k} - (\lambda\theta_j)$ shifts θ_j^{k} closer to 0

θ_j negative ($\lambda\theta_j < 0$) $\theta_j^{k} - (-\lambda\theta_j)$ shifts θ_j^{k} closer to θ

(e) Instead of applying gradient descent, solve for the optimal coordinate descent solution of θ_0 and θ_j.
Hint: Set the partial derivative to 0. You can use the following notation for convenience:

$$h^{-j}_{\theta}(x) \triangleq \theta_0 + \theta_1 x_1 + \ldots \theta_{j-1}x_{j-1} + \theta_{j+1}x_{j+1} + \ldots + \theta_p x_p = h_\theta(x) - \theta_j x_j$$



i.e.

$$\frac{\partial}{\partial\theta_j} = \frac{2}{n}\sum_{i=1}^{n}(y^{(i)}-h_\theta(x^{(i)}))x_j^{(i)} + 2\lambda\theta_j := 0$$

$$= \frac{2}{n}\sum_{i=1}^{n}(y^{(i)}-(h_\theta^{-j}(x^{(i)})+\theta_j x_j))x_j^{(i)} = 0$$

$$= \frac{2}{n}\left[\sum_{i=1}^{n}-\theta_j(x_j^{(i)})^2 + \sum_{i=1}^{n}(y^{(i)}-h_\theta^{-j}(x^{(i)}))x_j^{(i)}\right) + 2\lambda\theta_j = 0$$

$$\iff \cancel{\frac{2}{n}}\left[\sum_{i=1}^{n}-\theta_j(x_j^{(i)})^2\right]+2\lambda\theta_j = -\cancel{\frac{2}{n}}\sum_{i=1}^{n}(y^{(i)}-h_\theta^{-j}(x^{(i)}))x_j^{(i)}$$

$$\cancel{-}\theta_j\sum_{i=1}^{n}(x_j^{(i)})^2 + 2\lambda\theta_j = \cancel{-}\sum_{i=1}^{n}(y^{(i)}-h_\theta^{-j}(x^{(i)}))x_j^{(i)}$$

$$\theta_j\left(\sum_{i=1}^{n}(x_j^{(i)})^2 + 2\lambda\right) = \sum_{i=1}^{n}(y^{(i)}-h_\theta^{-j}(x^{(i)}))x_j^{(i)}$$

$$\theta_j = \frac{\sum_{i=1}^{n}(y^{(i)}-h_\theta^{-j}(x^{(i)}))x_j^{(i)}}{\sum_{i=1}^{n}(x_j^{(i)})^2 + 2\lambda}$$

$$\theta_0 = \frac{1}{n}\sum_{i=1}^{n}(y^{(i)}-h_\theta^{-0}(x^{(i)})))$$

This answer matches b: when $\lambda = 0$, weights are normal. As $\lambda \to +\infty$, weights approach 0. As $\lambda \to +\infty$, weights also approach 0.
The role of λ is to "lasso in" the weights of the model.

(f) Using the coordinate descent solution you found, explain the role of the regularization term, when $\lambda >$ 0. Does this match your answers in part (b) and (d)?

The role of the regularization term is to "lasso in" the weights of the model. As λ increases in magnitude, the size of the weights approaches zero(the denominator of the weights fraction approaches infinity).

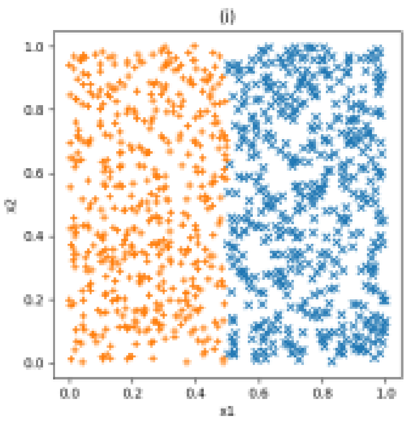
(g) The regularizer plays another role in ensuring the learning algorithm is stable. Consider the case when the feature values are very small – what happens to your analytical solution? How does regularization help (for $\lambda > 0$)?

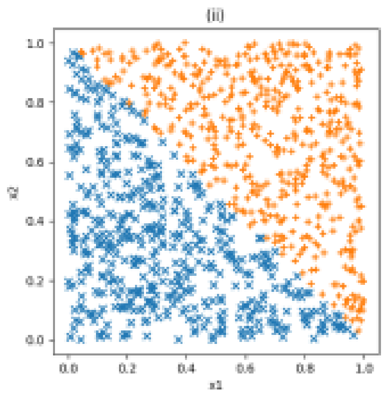
When the feature values are very small(I.E. less than 1), the regularization constant can help prevent the weights from imploding in value($1/(0.25)^2 = 16$. As feature value => 0, weight values would => infinity

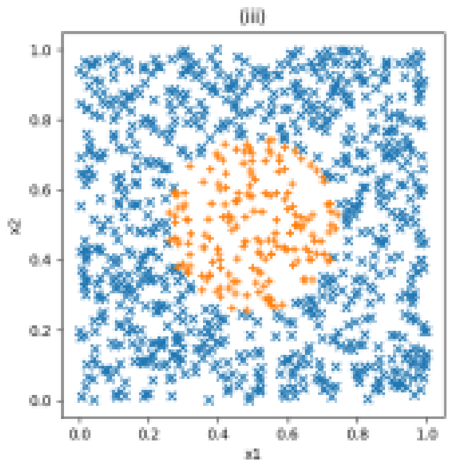
(h) In order for the problem to be well-defined, an appropriate regularization constant λ must be chosen for the given problem (dataset). How should the designer choose λ?

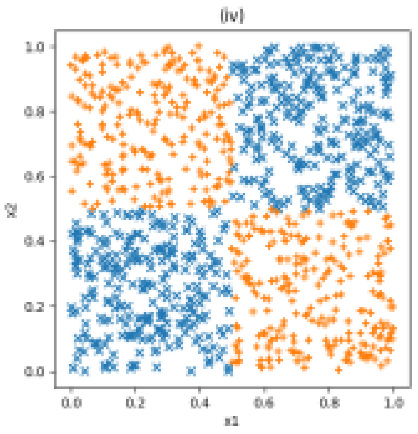
1. If the feature values are not very normalized(I.E. some features are quite small[$f_j < 1$] or quite large, a larger lambda constant would be useful). This way, all features would receive reasonably sized weights and no single feature would dominate the model decision making process.
2. However, if the feature characteristics are normal and well known, a lower λ value can help the model be more accurate(prevents under-fitting).

Note: This particular form of linear regression is known by many names – ridge regression, penalized least squares, weight decay, ℓ_2-regularization (because the regularizer is the ℓ_2-norm of the weight vector θ), Tikhonov regularization, and possibly more. As is common with concepts known by many names, this method was independently discovered by many people in many different contexts, which indicates its likely importance.









5. 3 points. *Logistic regression.*

(a) Consider the above datasets, with data $x = (x_1, x_2) \in \mathrm{R}^2$, and labels are '+' (orange) for the positive class and '×' (blue) for the negative class. Find features and parameters for each dataset such that a logistic regression classifier will classify all the data points correctly. Try to use as few features as possible.

Case 1: feature(x_1), weight(1)

Case 2: features(x_1, x_2), weights(1,1)

If $x_1 + x_2 > 1$, positive class(+)

Else negative class(x)

Case 3: features(x_1, x_2, $(x_1-0.5)^2$, $(x_2-0.5)^2$), weights(1, 1, 10, 10)

If $(x_1 - 0.5)^2 + (x_2 - 0.5)^2 > 0.125$, positive class(+)

If < 0.125, negative class(-)

Case 4: features(x_1, x_2, (x_1 && $x_2 > 0.5$), (x_1 && $x_2 > 0.5$))

If x_1 and x_2 are both < 0.5, or if x_1 and x_2 are both > 0.5, negative class(x)

Else, positive class(+)

(b) Suppose we had the following dataset consisting of 4 samples (listed as $(x_1, x_2) \to y$ pairs):
$(0, 0) \to 0$; $(0, 1) \to 0$; $(1, 0) \to 1$; $(1, 1) \to 1$
Run logistic regression on this dataset until the model classifies all samples correctly (this happens quickly). Start from parameters $(\theta_0, \theta_1, \theta_2) = (0, 0, 0)$, with learning rate $\alpha = 1$, using batch gradient ascent. After each iteration, show the parameters computed and draw the decision boundary.
If you find it too tedious to do by hand, you may write a program to do this for you (and submit your code); however, there may be some benefit in seeing the calculation unfolding in front of you.

5b.

$$\frac{\partial}{\partial \theta_j} = \sum_{i=1}^{4} (y^{(i)} - h_\theta(x^{(i)}))\, x_j^{(i)}$$

$K=0 \quad \vec{w} = \{\theta_0, \theta_1, \theta_2\} = \{0, 0, 0\}$

$$h_\theta(x) = \frac{1}{1+e^{-(\theta_0 + \theta_1 x_1 + \theta_2 x_2)}}$$

$$\frac{\partial}{\partial \theta_0} = 0 + 1 \times \sum_{i=1}^{4} (y^{(i)} - h_\theta(x^{(i)}))$$

$$\frac{\partial}{\partial \theta_1} = 0 + 1 \times \sum_{i=1}^{4} (y^{(i)} - h_\theta(x^{(i)}))\, x_1^{(i)}$$

$$\frac{\partial}{\partial \theta_2} = 0 + 1 \times \sum_{i=1}^{4} (y^{(i)} - h_\theta(x^{(i)}))\, x_2^{(i)}$$

$$\sum_{i=1}^{4} (y^{(i)} - h_\theta(x^{(i)}))$$

$h_\theta(x^1) = \text{sig}(0) = 0.5$
$h_\theta(x^2) = \text{sig}(0) = 0.5$
$h_\theta(x^3) = 0.5$
$h_\theta(x^4) = 0.5$

θ_0 : $i=1$ $(0-0.5)\cdot 1$ $(x_0) = 1$
$\sum_{i=1}^{4} (y^{(i)} - h_\theta(x^{(i)})) \rightarrow$
$i=2$ $(0-0.5)\cdot 1$
$i=3$ $(1-0.5)\cdot 1$
$i=4$ $(1-0.5)\cdot 1$

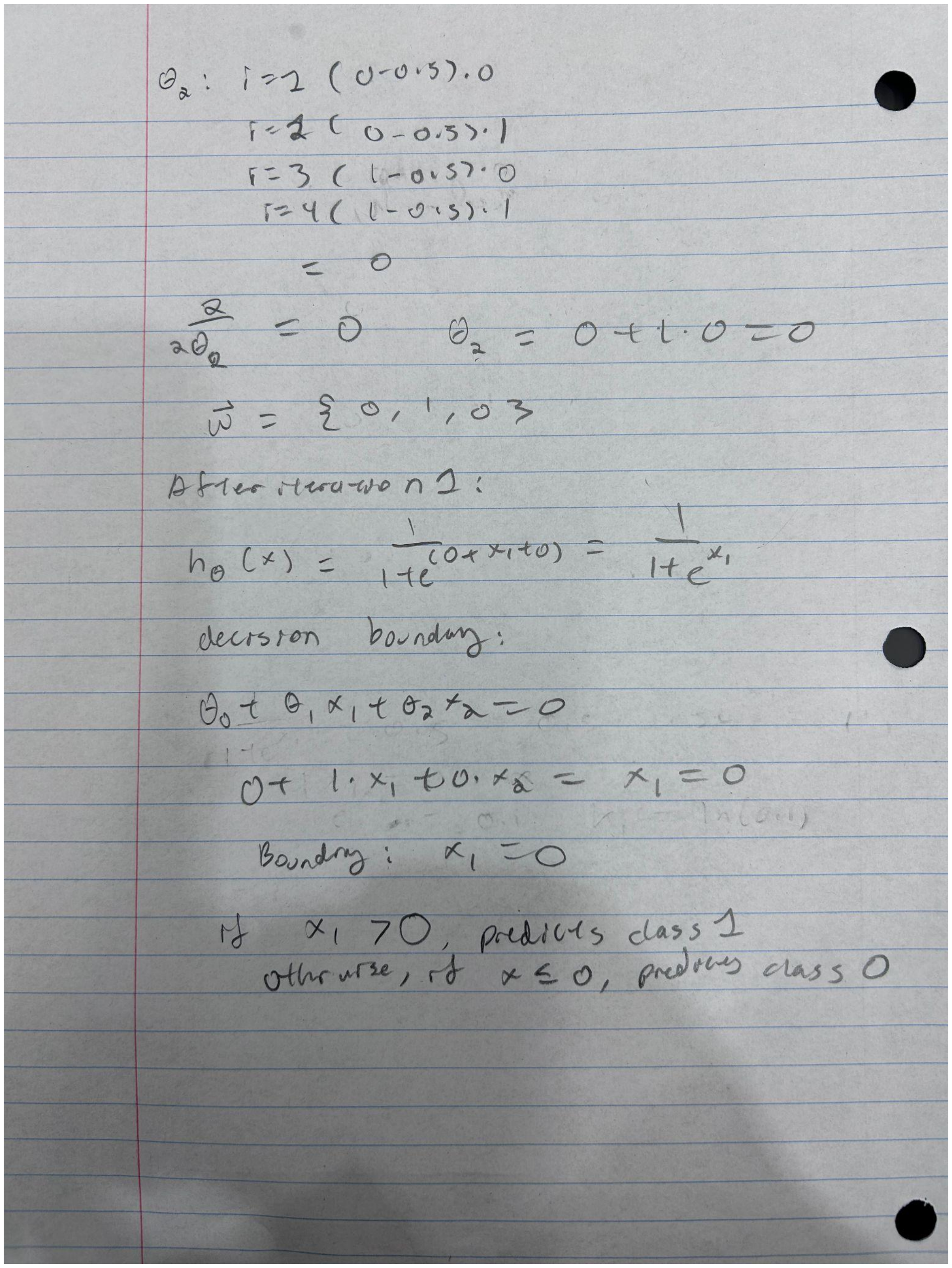
$= 0$

$\frac{\partial}{\partial \theta_0} = 0 \quad \boxed{\theta_0^{+} \leftarrow 0 + 1\cdot 0 = 0}$

θ_1 : $i=1$ $(0-0.5)\cdot 0$
$\sum_{i=1}^{4} (y^{(i)} - h_\theta(x^{(i)}))x_j^{(i)} \rightarrow$
$i=2$ $(0-0.5)\cdot 0$
$i=3$ $(1-0.5)\cdot 1$
$i=4$ $(1-0.5)\cdot 1$

$= 1$

$\frac{\partial}{\partial \theta_1} = 1 \quad \boxed{\theta_1^{+} \leftarrow 0 + 1\cdot 1 = 1}$



θ_2: $i=1\ (0-0.5)\cdot 0$
$i=2\ (0-0.5)\cdot 1$
$i=3\ (1-0.5)\cdot 0$
$i=4\ (1-0.5)\cdot 1$
$= 0$

$\frac{\partial}{\partial\theta_2} = 0 \quad \theta_2 = 0 + 1\cdot 0 = 0$

$\vec{w} = \{0, 1, 0\}$

After iteration 1:

$h_\theta(x) = \frac{1}{1+e^{(0+x_1+0)}} = \frac{1}{1+e^{x_1}}$

decision boundary:

$\theta_0 + \theta_1 x_1 + \theta_2 x_2 = 0$

$0 + 1\cdot x_1 + 0\cdot x_2 = x_1 = 0$

Boundary: $x_1 = 0$

if $x_1 > 0$, predicts class 1
otherwise, if $x \le 0$, predicts class 0

(c) Extra credit: 1 point. If we let logistic regression continue running on this example, eventually the parameters will keep on growing in magnitude to ±∞. Why does this happen, mathematically? What is happening to the model and the sigmoid function, informally?
Hint: Assuming that the model's parameters are such that all points are already classified correctly, consider what happens to the log-likelihood objective when the parameters are doubled.

Because the model is based on a sigmoid function, the probability it outputs will never be 1, but approach 1 as the parameters approach ±∞. Even if the all the points are classified correctly, the error function will still be non-zero(the sigmoid output is always non-zero and non-one, therefore |0 - h(x)| and |1-h(x)| is always non-zero. As a result, there will always be a "better" set of weights that decrease this error function. The parameters will continue growing in size and never converge.

6. 5 points. *Multinomial logistic regression.*

In lecture, we derived logistic regression for binary classification. In general, classification problems often involve more than two classes, e.g., digit recognition using the MNIST dataset. In this question, we will extend logistic regression to the *multi-class* setting.

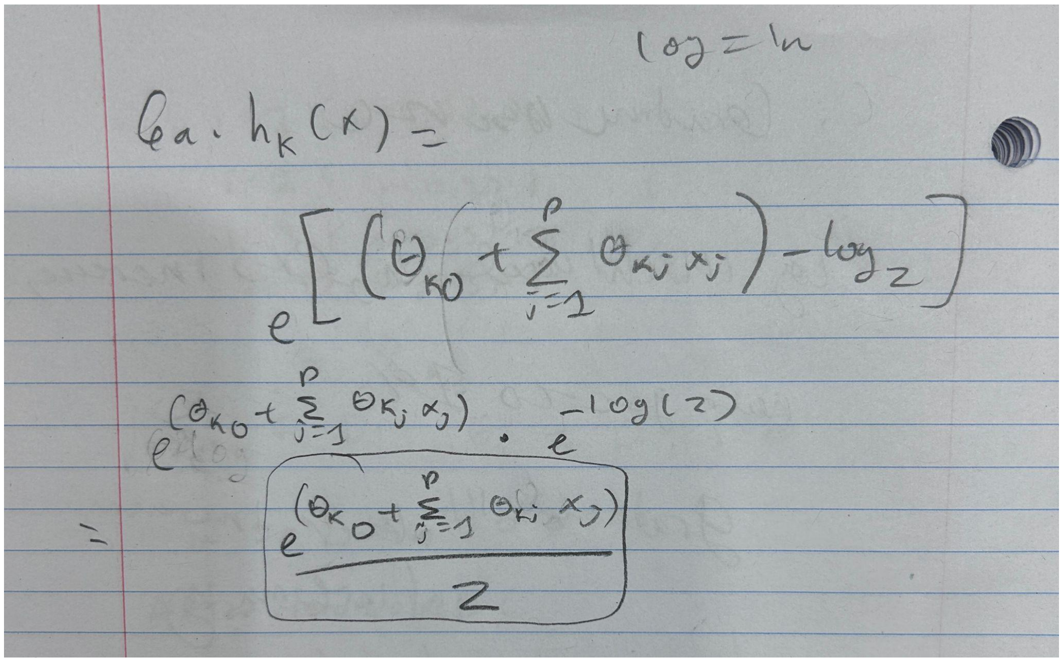
Recall that in the derivation of regular two-class logistic regression, we used the hypothesis to predict the *probability* that data point $x \in \mathbb{R}^p$ belongs to class 1:

The sigmoid form of the probability was derived by assuming a *log-linear model* (log = natural logarithm ln):

In the multi-class setting, we assume a slightly different log-linear model. Suppose there are K classes. Then we model the probability that x belongs to class k a

Each class k now has a *separate* parameter vector $\theta_k \in \mathbb{R}^{p+1}$, and θ_{kj} refers to the j'th element in θ_k. Z is known as the *partition function*, which normalizes $h_1(x), \ldots, h_k(x)$, since they should sum to 1.

(a) Using the multi-class log-linear model assumption, derive an expression for $h_k(x) = P(y = k|x; \theta_1, \ldots, \theta_K)$.



log = ln

(a). $h_K(x) =$

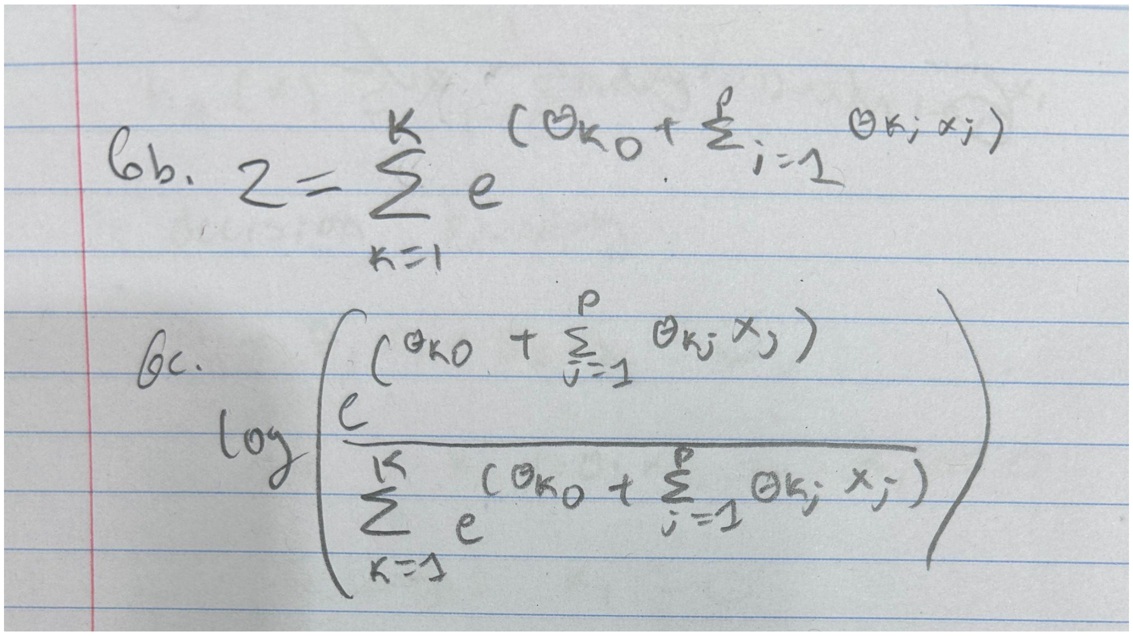
$$e^{\left[\left(\theta_{K0} + \sum_{j=1}^{p} \theta_{Kj} x_j\right) - \log Z\right]}$$

$$e^{\left(\theta_{K0} + \sum_{j=1}^{p} \theta_{Kj} x_j\right)} \cdot e^{-\log(Z)}$$

$$= \boxed{\frac{e^{\left(\theta_{K0} + \sum_{j=1}^{p} \theta_{Kj} x_j\right)}}{Z}}$$

(b) Using the fact that probabilities across all K classes must sum to 1, derive an expression for Z.

We want to learn parameter vectors $\{\theta_1, \ldots, \theta_K\}$ from a dataset $(x^{(i)}, y^{(i)})_{i=1}^{n}$, where each label $y^{(i)}$ is an integer from 1 to K. To do so, we will formulate an optimization problem according to the *maximum likelihood* principle, as we did for two-class logistic regression.



6b. $Z = \sum_{k=1}^{K} e^{(\theta_{k0} + \sum_{j=1} \theta_{kj} x_j)}$

6c. $\log\left(\dfrac{e^{(\theta_{k0} + \sum_{j=1}^{p} \theta_{kj} x_j)}}{\sum_{k=1}^{K} e^{(\theta_{k0} + \sum_{j=1}^{p} \theta_{kj} x_j)}} \right)$

(c) Derive an expression for the *log-likelihood* function $\ell(\theta_1, \ldots, \theta_K)$: $\theta_1, \ldots, \theta_K$

Assume that the data is *independent and identically distributed* (IID).

Hint: You should encounter a log $\mathrm{P}^{K}{}_{k=1}$ exp(. . .) quantity, which cannot be further simplified.

(d) To find a maximum-likelihood estimate of $\{\theta_1, \ldots, \theta_K\}$, we can do *gradient ascent* on the log-likelihood function $\ell(\theta_1, \ldots, \theta_K)$. Find the gradient $\hat{\partial}$

$$\partial\theta_{kj}\ell(\theta_1, \ldots, \theta_K).$$

For convenience, you may assume that $j \neq 0$ (i.e., ignore the case for the bias term), or assume that $x^{(i)}$ $_0 = 1$ for all data samples (i.e., treat the bias term as a "dummy" constant feature).

$$\partial\theta_{kj}\log \mathrm{P}^{K}k'_{=1}\exp\ \theta'_{k0} + \mathrm{P}^{p}j'_{=1}\theta'_{kj'}x_{j'} \text{ first.}$$

Hint: You may find it helpful to derive $\hat{\partial}$

Note the use of k' and j' in the summations to avoid overloading the indices.

The final gradient expression should be similar to the one derived in two-class logistic regression.

6d. $\frac{\partial}{\partial \theta_{kj}} \log \sum_{k'=1}^{K} e^{(\theta_{k'0} + \sum_{j'=1}^{p} \theta_{k'j'} x_{j'})}$

$$= \frac{1}{\sum_{k'=1} e^{(\theta_{k'0} + \sum_{j'=1}^{p} \theta_{k'j'} x_{j'})}} \cdot \frac{\partial}{\partial \theta_{kj}} \sum_{k'=1}^{K} e^{(\theta_{k'0} + \sum_{j'=1}^{p} \theta_{k'j'} x_{j'})}$$

$$\rightarrow \frac{\partial}{\partial \theta_{kj}} e^{(\theta_{k'0} + \sum_{j'=1}^{p} \theta_{k'j'} x_{j'})} =$$

$$e^{\theta_{k'0} + \sum_{j'=1}^{p} \theta_{k'j'} x_{j'}} \cdot \frac{\partial}{\partial \theta_{kj}} \left(\theta_{k'0} + \underline{\sum_{j'=1}^{p} \theta_{j'} x_{j'}} \right)$$

$$= e^{\theta_{k'0} + \sum_{j'=1}^{p} \theta_{k'j'} x_{j'}} \cdot x_{j'} = h_\theta(x) \cdot x_j$$

$$\text{log-likelihood} = \sum_{r=1}^{n} \left[\theta_{k0} + \sum_{j'=1}^{p} \theta_{kj} x_j - \log \sum_{k=1}^{K} e^{(\theta_{k0} + \sum_{j=1}^{p} \theta_{kj} x_{j'})} \right]$$

$$\frac{\partial}{\partial \theta_{kj}} = \sum_{r=1}^{n} \left[[y^{(n)} = y'] x_j^{(n)} - h_\theta(x)^{(n)} \cdot x_j^{(n)} \right]$$