CS 229, Spring 2023
Problem Set #2 Solutions

YOUR NAME HERE (`YOUR SUNET HERE`)

Due Wednesday, May 10 at 11:59 pm on Gradescope.

Notes: (1) These questions require thought, but do not require long answers. Please be as concise as possible.

(2) If you have a question about this homework, we encourage you to post your question on our Ed forum, at `https://edstem.org/us/courses/37893/discussion/`.

(3) If you missed the first lecture or are unfamiliar with the collaboration or honor code policy, please read the policy on the course website before starting work.

(4) For the coding problems, you may not use any libraries except those defined in the provided `environment.yml` file. In particular, ML-specific libraries such as scikit-learn are not permitted.

(5) The due date is Wednesday, May 10 at 11:59 pm. If you submit after Wednesday, May 10 at 11:59 pm, you will begin consuming your late days. The late day policy can be found in the course website: Course Logistics and FAQ.

All students must submit an electronic PDF version of the written question including plots generated from the codes. We highly recommend typesetting your solutions via LaTeX. All students must also submit a zip file of their source code to Gradescope, which should be created using the `make_zip.py` script. You should make sure to (1) restrict yourself to only using libraries included in the `environment.yml` file, and (2) make sure your code runs without errors. Your submission may be evaluated by the auto-grader using a private test set, or used for verifying the outputs reported in the writeup. Please make sure that your PDF file and zip file are submitted to the corresponding Gradescope assignments respectively. We reserve the right to not give any points to the written solutions if the associated code is not submitted.

Honor code: We strongly encourage students to form study groups. Students may discuss and work on homework problems in groups. However, each student must write down the solution independently, and without referring to written notes from the joint session. Each student must understand the solution well enough in order to reconstruct it by him/herself. It is an honor code violation to copy, refer to, or look at written or code solutions from a previous year, including but not limited to: official solutions from a previous year, solutions posted online, and solutions you or someone else may have written up in a previous year. Furthermore, it is an honor code violation to post your assignment solutions online, such as on a public git repo. We run plagiarism-detection software on your code against past solutions as well as student submissions from previous years. Please take the time to familiarize yourself with the Stanford Honor Code[1] and the Stanford Honor Code[2] as it pertains to CS courses.

Due to size limit, the starter code is provided in canvas `PSet2` folder.

[1]https://communitystandards.stanford.edu/policies-and-guidance/honor-code
[2]https://web.stanford.edu/class/archive/cs/cs106b/cs106b.1164/handouts/honor-code.pdf

1. **[35 points] Linear Classifiers (logistic regression and GDA)**

In this problem, we cover two probabilistic linear classifiers we have covered in class so far. First, a discriminative linear classifier: logistic regression. Second, a generative linear classifier: Gaussian discriminant analysis (GDA). Both of the algorithms find a linear decision boundary that separates the data into two classes, but make different assumptions. Our goal in this problem is to get a deeper understanding of the similarities and differences (and, strengths and weaknesses) of these two algorithms.

For this problem, we will consider two datasets, along with starter codes provided in the following files:

- `src/linearclass/ds1_{train,valid}.csv`
- `src/linearclass/ds2_{train,valid}.csv`
- `src/linearclass/logreg.py`
- `src/linearclass/gda.py`

Each file contains n examples, one example $(x^{(i)}, y^{(i)})$ per row. In particular, the i-th row contains columns $x_1^{(i)} \in \mathbb{R}$, $x_2^{(i)} \in \mathbb{R}$, and $y^{(i)} \in \{0, 1\}$. In the subproblems that follow, we will investigate using logistic regression and Gaussian discriminant analysis (GDA) to perform binary classification on these two datasets.

Typically, a trained model is evaluated by its performance on the validation dataset. The validation dataset is a set of examples drawn from the same (or a similar) distribution as the training data. Intuitively, this is because we need the trained model to correctly predict the label for not only the training data, but also new samples from the same distribution.

(a) [10 points]

In lecture we saw the average empirical loss for logistic regression:

$$J(\theta) = -\frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)}\log(h_\theta(x^{(i)})) + (1-y^{(i)})\log(1-h_\theta(x^{(i)}))\right),$$

where $y^{(i)} \in \{0,1\}$, $h_\theta(x) = g(\theta^T x)$ and $g(z) = 1/(1+e^{-z})$.

Find the Hessian H of this function, and show that for any vector z, it holds true that

$$z^T H z \geq 0.$$

Hint: You may want to start by showing that $\sum_i \sum_j z_i x_i x_j z_j = (x^T z)^2 \geq 0$. Recall also that $g'(z) = g(z)(1-g(z))$.

Remark: This is one of the standard ways of showing that the matrix H is positive semi-definite, written "$H \succeq 0$." This implies that J is convex, and has no local minima other than the global one. If you have some other way of showing $H \succeq 0$, you're also welcome to use your method instead of the one above.

Answer: Since we have $h(x) = g(\theta^T x)$, we have $\partial h(x)/\partial\theta_j = h(x)(1-h(x))x_j$. Thus,

$$\frac{\partial \log h_\theta(x^{(i)})}{\partial\theta_j} = (1-h_\theta(x^{(i)}))x_j^{(i)}$$

and

$$\frac{\partial \log(1 - h_\theta(x^{(i)}))}{\partial \theta_j} = -h_\theta(x^{(i)})x_j^{(i)}$$

Thus, we have

$$\begin{aligned}\frac{\partial J(\theta)}{\partial \theta_j} &= -\frac{1}{n}\sum_{i=1}^{n} y^{(i)}(1 - h_\theta(x^{(i)}))x_j^{(i)} - (1 - y^{(i)})h_\theta(x^{(i)})x_j^{(i)} \\ &= -\frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - h_\theta(x^{(i)}))x_j^{(i)}\end{aligned}$$

Therefore, the (j, k)th entry of the Hessian is

$$\begin{aligned}H_{jk} &= \frac{\partial^2 J(\theta)}{\partial \theta_j \partial \theta_k} \\ &= \frac{1}{n}\sum_{i=1}^{n} h_\theta(x^{(i)})(1 - h_\theta(x^{(i)}))x_j^{(i)}x_k^{(i)}\end{aligned}$$

Since $X = xx^T$ if and only if $X_{jk} = x_j x_k$, we have

$$H = \frac{1}{n}\sum_{i=1}^{n} h_\theta(x^{(i)})(1 - h_\theta(x^{(i)}))x^{(i)}(x^{(i)})^T$$

We need to show that $z^T H z \geq 0$ for any vector z.

$$\begin{aligned}z^T H z &= \frac{1}{n}\sum_{i=1}^{n} z^T h_\theta(x^{(i)})(1 - h_\theta(x^{(i)}))x^{(i)}(x^{(i)})^T z \\ &= \frac{1}{n}\sum_{i=1}^{n} h_\theta(x^{(i)})(1 - h_\theta(x^{(i)}))((x^{(i)})^T z)(z^T x^{(i)}) \\ &= \frac{1}{n}\sum_{i=1}^{n} h_\theta(x^{(i)})(1 - h_\theta(x^{(i)}))((x^{(i)})^T z)^2 \\ &\geq 0\end{aligned}$$

At the last step, we used the fact that $h_\theta(x^{(i)}) \in [0, 1]$.

(b) [0 points] **Coding problem.** In `src/linearclass/logreg.py`, we provide the code to train a logistic regression classifier using Newton's Method. Starting with $\theta = \vec{0}$, run Newton's Method until the updates to θ are small: Specifically, train until the first iteration k such that $\|\theta_k - \theta_{k-1}\|_1 < \epsilon$, where $\epsilon = 1 \times 10^{-5}$. Make sure to write your model's predicted probabilities on the validation set to the file specified in the code.

Include a plot of the **validation data** with x_1 on the horizontal axis and x_2 on the vertical axis. To visualize the two classes, use a different symbol for examples $x^{(i)}$ with $y^{(i)} = 0$ than for those with $y^{(i)} = 1$. On the same figure, plot the decision boundary found by logistic regression (i.e., line corresponding to $p(y|x) = 0.5$).

Note: If you want to print the loss during training, you may encounter some numerical instability issues. Recall that the loss function on an example (x, y) is defined as

$$y\log(h_\theta(x)) + (1 - y)\log(1 - h_\theta(x)),$$

where $h_\theta(x) = (1 + \exp(-x^\top \theta))^{-1}$. Technically speaking, $h_\theta(x) \in (0, 1)$ for any $\theta, x \in \mathbb{R}^d$. However, in Python a real number only has finite precision. So it is possible that in your implementation, $h_\theta(x) = 0$ or $h_\theta(x) = 1$, which makes the loss function ill-defined. A typical solution to the numerical instability issue is to add a small perturbation. In this case, you can compute the loss function using

$$y \log(h_\theta(x) + \epsilon) + (1 - y) \log(1 - h_\theta(x) + \epsilon),$$

instead, where ϵ is a very small perturbation (for example, $\epsilon = 10^{-5}$). **Answer:**

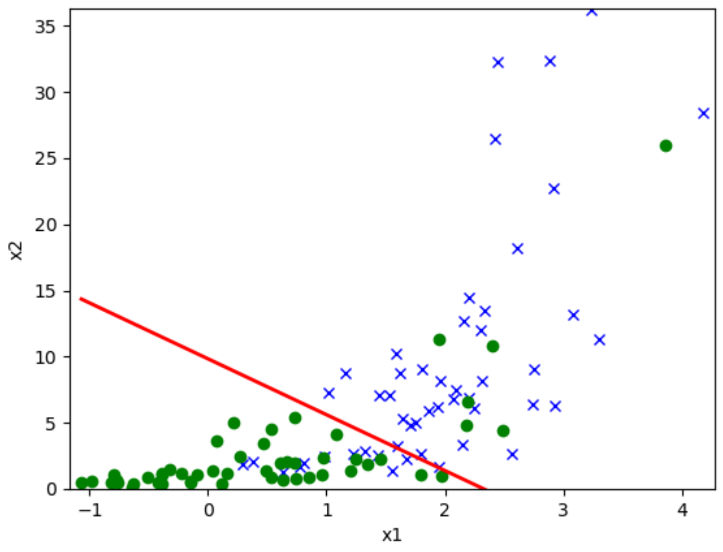

Figure 1: Dataset 1

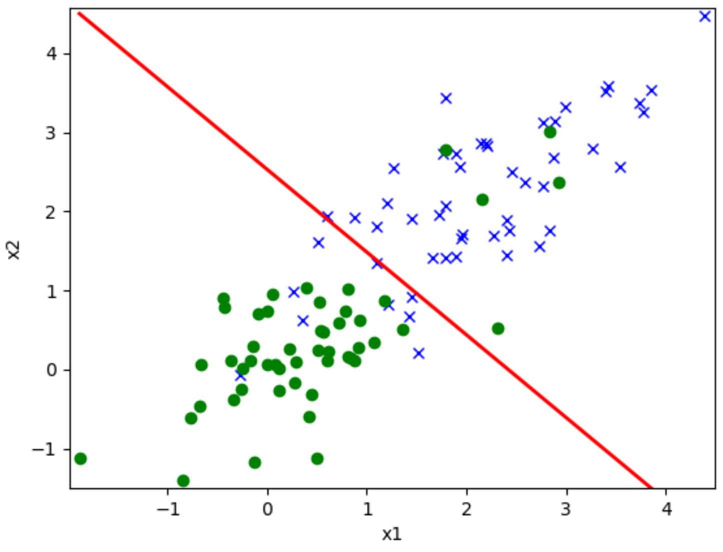

Figure 2: Dataset 2

(c) [5 points] Recall that in GDA we model the joint distribution of (x, y) by the following equations:

$$p(y) = \begin{cases} \phi & \text{if } y = 1 \\ 1 - \phi & \text{if } y = 0 \end{cases} \tag{1}$$

$$p(x|y = 0) = \frac{1}{(2\pi)^{d/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x - \mu_0)^T \Sigma^{-1}(x - \mu_0)\right) \tag{2}$$

$$p(x|y = 1) = \frac{1}{(2\pi)^{d/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x - \mu_1)^T \Sigma^{-1}(x - \mu_1)\right),$$

where ϕ, μ_0, μ_1, and Σ are the parameters of our model.
Suppose we have already fit ϕ, μ_0, μ_1, and Σ, and now want to predict y given a new point x. To show that GDA results in a classifier that has a linear decision boundary, show the posterior distribution can be written as

$$p(y = 1 \mid x; \phi, \mu_0, \mu_1, \Sigma) = \frac{1}{1 + \exp(-(\theta^T x + \theta_0))},$$

where $\theta \in \mathbb{R}^d$ and $\theta_0 \in \mathbb{R}$ are appropriate functions of ϕ, Σ, μ_0, and μ_1. State the value of θ and θ_0 as a function of $\phi, \mu_0, \mu_1, \Sigma$ explicitly.

Answer: Since we have

$$\begin{aligned} p(y = 1|x; \phi, \mu_0, \mu_1, \Sigma) &= \frac{p(y = 1)p(x|y = 1)}{p(y = 0)p(x|y = 0) + p(y = 1)p(x|y = 1)} \\ &= \frac{1}{1 + \frac{p(y=0)p(x|y=0)}{p(y=1)p(x|y=1)}} \end{aligned}$$

We just need to show that the term in the denominator is equal to the exponential term in the problem description. Put in the equations for $p(x|y=0)$ and $p(x|y=1)$, we have

$$\begin{aligned}\frac{p(y=0)p(x|y=0)}{p(y=1)p(x|y=1)} &= \frac{(1-\phi)\frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}}\exp\left(-\frac{1}{2}(x-\mu_0)^T\Sigma^{-1}(x-\mu_0)\right)}{\phi\frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}}\exp\left(-\frac{1}{2}(x-\mu_1)^T\Sigma^{-1}(x-\mu_1)\right)} \\ &= \frac{1-\phi}{\phi}\exp\left(-\frac{1}{2}(x-\mu_0)^T\Sigma^{-1}(x-\mu_0)+\frac{1}{2}(x-\mu_1)^T\Sigma^{-1}(x-\mu_1)\right)\end{aligned}$$

After rearranging the terms, and setting $\theta = -\Sigma^{-1}(\mu_1-\mu_0)$, $\theta_0 = \frac{1}{2}(\mu_0^T\Sigma^{-1}\mu_0 - \mu_1^T\Sigma^{-1}\mu_1) - \log\frac{1-\phi}{\phi}$, we have

$$\frac{p(y=0)p(x|y=0)}{p(y=1)p(x|y=1)} = \exp(-(\theta^Tx+\theta_0))$$

which is exactly the exponential term in the problem description.

(d) [7 points] Given the dataset, we claim that the maximum likelihood estimates of the parameters are given by

$$\phi = \frac{1}{n}\sum_{i=1}^n 1\{y^{(i)}=1\} \tag{3}$$

$$\mu_0 = \frac{\sum_{i=1}^n 1\{y^{(i)}=0\}x^{(i)}}{\sum_{i=1}^n 1\{y^{(i)}=0\}} \tag{4}$$

$$\mu_1 = \frac{\sum_{i=1}^n 1\{y^{(i)}=1\}x^{(i)}}{\sum_{i=1}^n 1\{y^{(i)}=1\}} \tag{5}$$

$$\Sigma = \frac{1}{n}\sum_{i=1}^n (x^{(i)}-\mu_{y^{(i)}})(x^{(i)}-\mu_{y^{(i)}})^T$$

The log-likelihood of the data is

$$\begin{aligned}\ell(\phi,\mu_0,\mu_1,\Sigma) &= \log\prod_{i=1}^n p(x^{(i)},y^{(i)};\phi,\mu_0,\mu_1,\Sigma) \\ &= \log\prod_{i=1}^n p(x^{(i)}|y^{(i)};\mu_0,\mu_1,\Sigma)p(y^{(i)};\phi).\end{aligned} \tag{6}$$

By maximizing ℓ with respect to the four parameters, prove that the maximum likelihood estimates of ϕ, μ_0, μ_1, and Σ are indeed as given in the formulas above. (You may assume that there is at least one positive and one negative example, so that the denominators in the definitions of μ_0 and μ_1 above are non-zero.)

Answer: The log likelihood is

$$\begin{aligned}
&\ell(\phi,\mu_0,\mu_1,\Sigma)\\
=&\log\prod_{i=1}^n p(x^{(i)}|y^{(i)};\phi,\mu_0,\mu_1,\Sigma)p(y^{(i)};\phi)\\
=&\sum_{i=1}^n \log p(x^{(i)}|y^{(i)};\phi,\mu_0,\mu_1,\Sigma)+\sum_{i=1}^n \log p(y^{(i)};\phi)\\
\to&\sum_{i=1}^n\left(\log\frac{1}{|\Sigma|}-(x^{(i)}-\mu_{y^{(i)}})^T\Sigma^{-1}(x^{(i)}-\mu_{y^{(i)}})\right)+\sum_{i=1}^n\left(y^{(i)}\log\phi+(1-y^{(i)})\log(1-\phi)\right)
\end{aligned}$$

Where I did some scaling to get rid of some constants. We now set the derivative of the log likelihood to zero to find the parameters. For ϕ, we have

$$\begin{aligned}
\frac{\partial\ell}{\partial\phi}&=\sum_{i=1}^n\left(\frac{y^{(i)}}{\phi}-\frac{1-y^{(i)}}{1-\phi}\right)\\
&=\frac{\sum_{i=1}^n 1\{y^{(i)}=1\}}{\phi}-\frac{n-\sum_{i=1}^n 1\{y^{(i)}=0\}}{1-\phi}
\end{aligned}$$

Setting this to zero, we have

$$\phi=\frac{1}{n}\sum_{i=1}^n 1\{y^{(i)}=1\}$$

For μ_0, we have

$$\begin{aligned}
\frac{\partial\ell}{\partial\mu_0}&=\sum_{i=1}^n\left(-\Sigma^{-1}(x^{(i)}-\mu_0)\right)\\
&=-\Sigma^{-1}\sum_{i=1}^n 1\{y^{(i)}=0\}(x^{(i)}-\mu_0)
\end{aligned}$$

Setting this to zero, we have

$$\mu_0=\frac{\sum_{i=1}^n 1\{y^{(i)}=0\}x^{(i)}}{\sum_{i=1}^n 1\{y^{(i)}=0\}}$$

Similarly, we have

$$\mu_1=\frac{\sum_{i=1}^n 1\{y^{(i)}=1\}x^{(i)}}{\sum_{i=1}^n 1\{y^{(i)}=1\}}$$

For Σ, we have

$$\begin{aligned}
\frac{\partial\ell}{\partial\Sigma}&=\sum_{i=1}^n\left(\frac{1}{2}\Sigma^{-1}-\frac{1}{2}\Sigma^{-1}(x^{(i)}-\mu_{y^{(i)}})(x^{(i)}-\mu_{y^{(i)}})^T\Sigma^{-1}\right)\\
&=\frac{n}{2}\Sigma^{-1}-\frac{1}{2}\Sigma^{-1}\sum_{i=1}^n(x^{(i)}-\mu_{y^{(i)}})(x^{(i)}-\mu_{y^{(i)}})^T\Sigma^{-1}
\end{aligned}$$

Setting this to zero, we have

$$\Sigma = \frac{1}{n} \sum_{i=1}^{n} (x^{(i)} - \mu_{y^{(i)}})(x^{(i)} - \mu_{y^{(i)}})^T$$

(e) [5 points] **Coding problem.** In `src/linearclass/gda.py`, fill in the code to calculate ϕ, μ_0, μ_1, and Σ, use these parameters to derive θ, and use the resulting GDA model to make predictions on the validation set. Make sure to write your model's predictions on the validation set to the file specified in the code.

Include a plot of the **validation data** with x_1 on the horizontal axis and x_2 on the vertical axis. To visualize the two classes, use a different symbol for examples $x^{(i)}$ with $y^{(i)} = 0$ than for those with $y^{(i)} = 1$. On the same figure, plot the decision boundary found by GDA (i.e, line corresponding to $p(y|x) = 0.5$).

Answer:

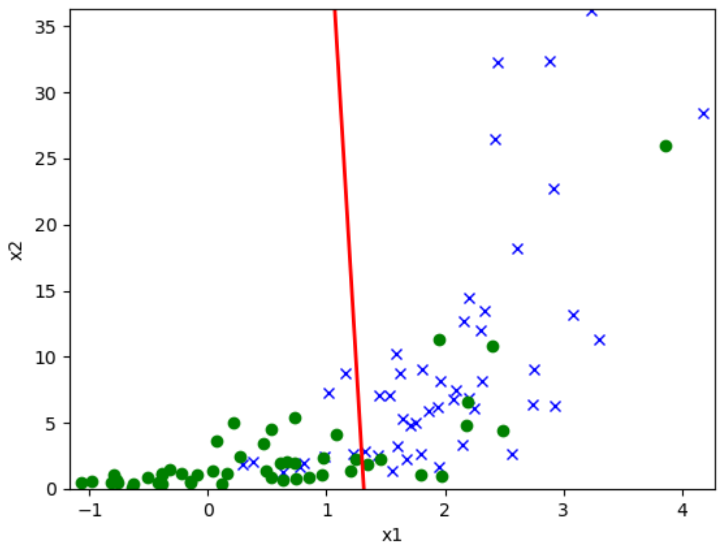

Figure 3: GDA prediction on dataset 1

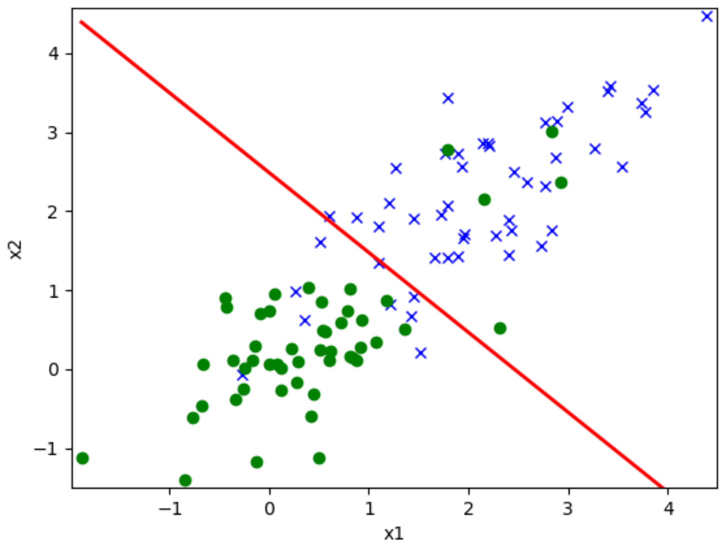

Figure 4: GDA prediction on dataset 2

(f) [2 points] For Dataset 1, compare the validation set plots obtained in part (b) and part (e) from logistic regression and GDA respectively, and briefly comment on your observation in a couple of lines.

Answer: Logistic regression prediction on dataset 1 looks more reasonable and more accurate than GDA prediction on dataset 1. The boundary captures the shape of the data better.

(g) [5 points] Repeat the steps in part (b) and part (e) for Dataset 2. Create similar plots on the **validation set** of Dataset 2 and include those plots in your writeup.

On which dataset does GDA seem to perform worse than logistic regression? Why might this be the case?

Answer: As the plots included in parts (b) and (e) shown, the two predictions look similar for dataset 2.

GDA seem to perform worse than logistic regression on dataset 1. The reason may be that the data is not Guassian distributed. Logistic regression performs better when the data is not Guassian distributed.

(h) [**1 points**] For the dataset where GDA performed worse in parts (f) and (g), can you find a transformation of the $x^{(i)}$'s such that GDA performs significantly better? What might this transformation be?

Answer: Transforming the data with a log function might make the data more like Guassian distributed, which could benefit GDA.

2. **[12 points] Logistic Regression: Training stability**

In this problem, we will be delving deeper into the workings of logistic regression. The goal of this problem is to help you develop your skills debugging machine learning algorithms (which can be very different from debugging software in general).

We have provided an implementation of logistic regression in `src/stability/stability.py`, and two labeled datasets A and B in `src/stability/ds1_a.csv` and `src/stability/ds1_b.csv`.

Please do not modify the code for the logistic regression training algorithm for this problem. First, run the given logistic regression code to train two different models on A and B. You can run the code by simply executing `python stability.py` in the `src/stability` directory.

(a) [2 points] What is the most notable difference in training the logistic regression model on datasets A and B?

Answer: Training on A is pretty fast while the training on B is running forever.

This may be model on dataset B is not converging.

(b) [5 points] Investigate why the training procedure behaves unexpectedly on dataset B, but not on A. Provide hard evidence (in the form of math, code, plots, etc.) to corroborate your hypothesis for the misbehavior. Remember, you should address why your explanation does *not* apply to A.

Hint: The issue is not a numerical rounding or over/underflow error.

Answer:

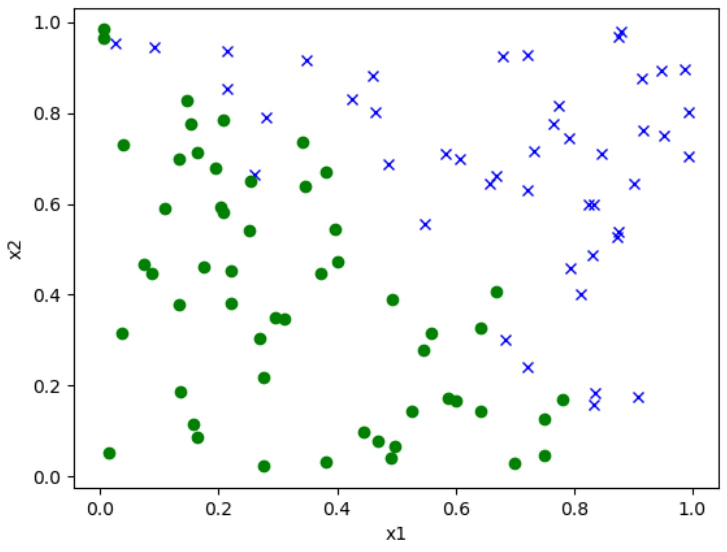

Figure 5: Dataset A

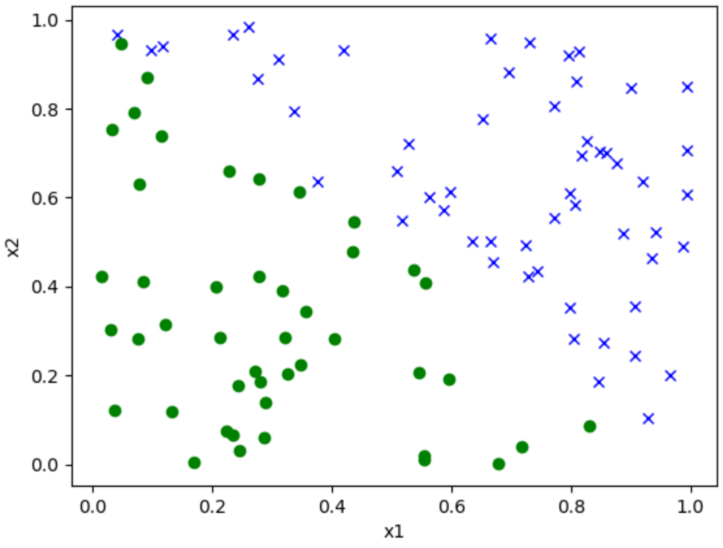

Figure 6: Dataset B

Plotting the two datasets, we can see that dataset A have interleaving points between two classes while B seem to have a clear boundary between two classes. It may be that the gradient of loss to theta is never going to be zero for dataset B, so the model is not converging. We can always increase theta to make the loss smaller, but it will never reach the minimum.

(c) [5 points] For each of these possible modifications, state whether or not it would lead to the provided training algorithm converging on datasets such as B. Justify your answers.

i. Using a different constant learning rate.

ii. Decreasing the learning rate over time (e.g. scaling the initial learning rate by $1/t^2$, where t is the number of gradient descent iterations thus far).

iii. Linear scaling of the input features.

iv. Adding a regularization term $\|\theta\|_2^2$ to the loss function.

v. Adding zero-mean Gaussian noise to the training data or labels.

Answer:

i. No. The constant learning rate does not help converge.

ii. Yes. The learning rate is decreasing, so the model might converge when the update is smaller than the threshold.

iii. No. This is the same as scaling the theta, which does not help.

iv. Yes. This will make theta converge without being unexpectedly large, which helps stops the infinite loop.

v. Probably. If the added noise makes the data not linearly separable, then the model will converge like A.

3. **[15 points] Kernelizing the Perceptron**

Let there be a binary classification problem with $y \in \{0, 1\}$. The perceptron uses hypotheses of the form $h_\theta(x) = g(\theta^T x)$, where $g(z) = \text{sign}(z) = 1$ if $z \geq 0$, 0 otherwise. In this problem we will consider a stochastic gradient descent-like implementation of the perceptron algorithm where each update to the parameters θ is made using only one training example. However, unlike stochastic gradient descent, the perceptron algorithm will only make one pass through the entire training set. The update rule for this version of the perceptron algorithm is given by

$$\theta^{(i+1)} := \theta^{(i)} + \alpha(y^{(i+1)} - h_{\theta^{(i)}}(x^{(i+1)}))x^{(i+1)}$$

where $\theta^{(i)}$ is the value of the parameters after the algorithm has seen the first i training examples. Prior to seeing any training examples, $\theta^{(0)}$ is initialized to $\vec{0}$.

(a) [3 points] Let K be a kernel corresponding to some very high-dimensional feature mapping ϕ. Suppose ϕ is so high-dimensional (say, ∞-dimensional) that it's infeasible to ever represent $\phi(x)$ explicitly. Describe how you would apply the "kernel trick" to the perceptron to make it work in the high-dimensional feature space ϕ, but without ever explicitly computing $\phi(x)$. [**Note:** You don't have to worry about the intercept term. If you like, think of ϕ as having the property that $\phi_0(x) = 1$ so that this is taken care of.] Your description should specify:

i. [1 points] How you will (implicitly) represent the high-dimensional parameter vector $\theta^{(i)}$, including how the initial value $\theta^{(0)} = 0$ is represented (note that $\theta^{(i)}$ is now a vector whose dimension is the same as the feature vectors $\phi(x)$);

ii. [1 points] How you will efficiently make a prediction on a new input $x^{(i+1)}$. I.e., how you will compute $h_{\theta^{(i)}}(x^{(i+1)}) = g({\theta^{(i)}}^T \phi(x^{(i+1)}))$, using your representation of $\theta^{(i)}$; and

iii. [1 points] How you will modify the update rule given above to perform an update to θ on a new training example $(x^{(i+1)}, y^{(i+1)})$; *i.e.*, using the update rule corresponding to the feature mapping ϕ:

$$\theta^{(i+1)} := \theta^{(i)} + \alpha(y^{(i+1)} - h_{\theta^{(i)}}(x^{(i+1)}))\phi(x^{(i+1)})$$

Answer: From the lecture, we know that this update rule implies that the parameter θ can be expressed as a linear combination of of the feature mapped vectors $\phi(x^{(i)})$. Therefore, we only need to store and operate on the coefficients of the linear combination instead of the high dimensional parameter vector. At any iteration i, there are only i coefficients to store and update.

From the kernel trick, we know that to compute $h_{\theta^{(i)}}(x^{(i+1)}) = g({\theta^{(i)}}^T \phi(x^{(i+1)}))$, we can use the kernel function $K(x^{(j)}, x^{(i+1)}) = \phi(x^{(j)})^T \phi(x^{(i+1)})$ which is precomputed and stored in a matrix K. Then, we can compute the prediction as

$$h_{\theta^{(i)}}(x^{(i+1)}) = g({\theta^{(i)}}^T \phi(x^{(i+1)})) = g(\sum_{j=1}^{i} \theta_j^{(i)} \phi(x^{(j)})^T \phi(x^{(i+1)})) = g(\sum_{j=1}^{i} \theta_j^{(i)} K(x^{(j)}, x^{(i+1)}))$$

where $\theta_j^{(i)}$ is the j-th coefficient of $\theta^{(i)}$.

Finally, to update $\theta^{(i)}$ on a new training example $(x^{(i+1)}, y^{(i+1)})$, we can append a new coefficient $\theta_{i+1}^{(i+1)}$ which is equal to $\alpha(y^{(i+1)} - h_{\theta^{(i)}}(x^{(i+1)}))$. Which can be computed as described above.

(b) [10 points] Implement your approach by completing the `initial_state`, `predict`, and `update_state` methods of `src/perceptron/perceptron.py`.

We provide three functions to be used as kernel, a dot-product kernel defined as:

$$K(x, z) = x^\top z, \tag{7}$$

a radial basis function (RBF) kernel, defined as:

$$K(x, z) = \exp\left(-\frac{\|x - z\|_2^2}{2\sigma^2}\right), \tag{8}$$

and finally the following function:

$$K(x, z) = \begin{cases} -1 & x = z \\ 0 & x \neq z \end{cases} \tag{9}$$

Note that the last function is not a kernel function (since its corresponding matrix is not a PSD matrix). However, we are still interested to see what happens when the kernel is invalid. Run `src/perceptron/perceptron.py` to train kernelized perceptrons on `src/perceptron/train.csv`. The code will then test the perceptron on `src/perceptron/test.csv` and save the resulting predictions in the `src/perceptron/` folder. Plots will also be saved in `src/perceptron/`.

Include the three plots (corresponding to each of the kernels) in your writeup, and indicate which plot belongs to which function.

Answer:

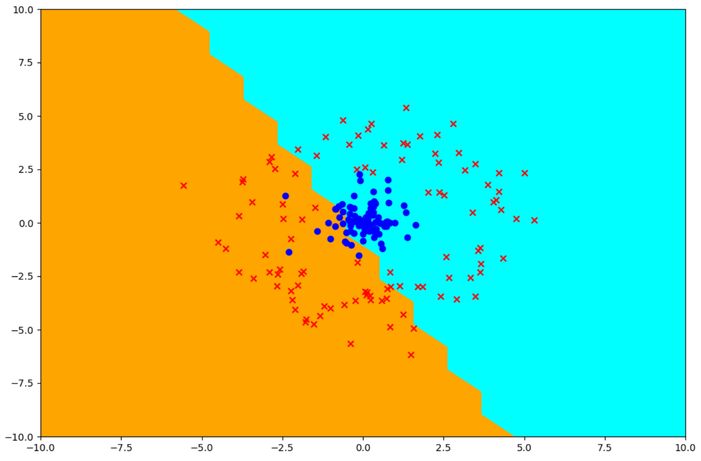

Figure 7: Dot-product kernel

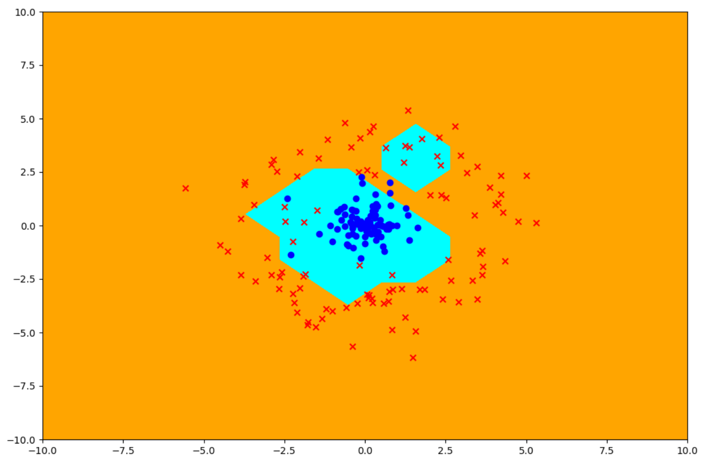

Figure 8: RBF kernel

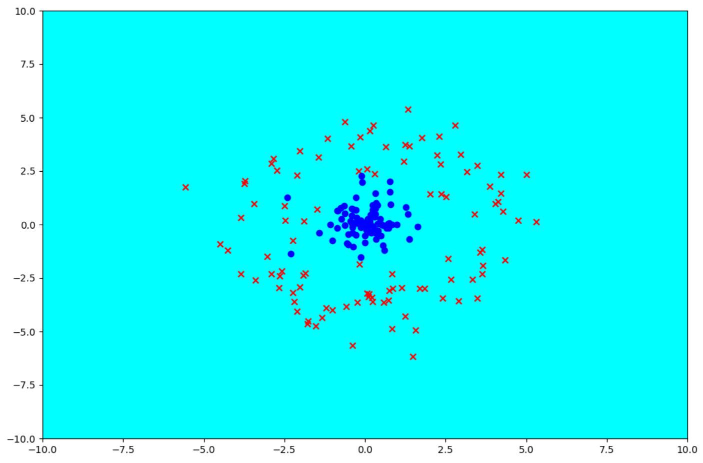

Figure 9: Non PSD kernel

(c) [2 points] One of the choices in part b completely fails, one works a bit, and one works well in classifying the points. Discuss the performance of different choices and why do they fail or perform well?

Answer: The non PSD kernel completely fails because it is not a valid kernel. The dot-product kernel works poorly because the data is not linearly separable. The RBF kernel works well because it is able to do radial boundaries.

4. **[30 points] Neural Networks: MNIST image classification**

Note: This question may requires knowledge on backpropagation covered on Monday of Week 5.

In this problem, you will implement a simple neural network to classify grayscale images of handwritten digits (0 - 9) from the MNIST dataset. The dataset contains 60,000 training images and 10,000 testing images of handwritten digits, 0 - 9. Each image is 28×28 pixels in size, and is generally represented as a flat vector of 784 numbers. It also includes labels for each example, a number indicating the actual digit (0 - 9) handwritten in that image. A sample of a few such images are shown below.

The data and starter code for this problem can be found in

- `src/mnist/nn.py`
- `src/mnist/images_train.csv`
- `src/mnist/labels_train.csv`
- `src/mnist/images_test.csv`
- `src/mnist/labels_test.csv`

Due to the limit of Ed, `images_train.csv` **and** `labels_train.csv` **are uploaded to canvas** `PSet2` **folder and not included in the zip file. The zip file with** `images_train.csv` **and** `labels_train.csv` **is also provided in canvas.**

The starter code splits the set of 60,000 training images and labels into a set of 50,000 examples as the training set, and 10,000 examples for dev set.

To start, you will implement a neural network with a single hidden layer and cross entropy loss, and train it with the provided data set. You will use the sigmoid function as activation for the hidden layer and use the cross-entropy loss for multi-class classification. Recall that for a single example (x, y), the cross entropy loss is:

$$\ell_{\mathrm{CE}}(\bar{h}_\theta(x), y) = -\log\left(\frac{\exp(\bar{h}_\theta(x)_y)}{\sum_{s=1}^{k}\exp(\bar{h}_\theta(x)_s)}\right),$$

where $\bar{h}_\theta(x) \in \mathbb{R}^k$ is the logits, i.e., the output of the the model on a training example x, $\bar{h}_\theta(x)_y$ is the y-th coordinate of the vector $\bar{h}_\theta(x)$ (recall that $y \in \{1, \dots, k\}$ and thus can serve as an index.)

For clarity, we provide the forward propagation equations below for the neural network with a single hidden layer. We have labeled data $(x^{(i)}, y^{(i)})_{i=1}^n$, where $x^{(i)} \in \mathbb{R}^d$, and $y^{(i)} \in \{1, \dots, k\}$ is ground truth label. Let m be the number of hidden units in the neural network, so that weight matrices $W^{[1]} \in \mathbb{R}^{d\times m}$ and $W^{[2]} \in \mathbb{R}^{m\times k}$.[3] We also have biases $b^{[1]} \in \mathbb{R}^m$ and $b^{[2]} \in \mathbb{R}^k$. The parameters of the model θ is $(W^{[1]}, W^{[2]}, b^{[1]}, b^{[2]})$. The forward propagation equations for a single input $x^{(i)}$ then are:

$$\begin{aligned} a^{(i)} &= \sigma\left({W^{[1]}}^\top x^{(i)} + b^{[1]}\right) \in \mathbb{R}^m \\ \bar{h}_\theta(x^{(i)}) &= {W^{[2]}}^\top a^{(i)} + b^{[2]} \in \mathbb{R}^k \\ h_\theta(x^{(i)}) &= \text{softmax}(\bar{h}_\theta(x^{(i)})) \in \mathbb{R}^k \end{aligned}$$

where σ is the sigmoid function.

For n training examples, we average the cross entropy loss over the n examples.

$$J(W^{[1]}, W^{[2]}, b^{[1]}, b^{[2]}) = \frac{1}{n}\sum_{i=1}^n \ell_{\text{CE}}(\bar{h}_\theta(x^{(i)}), y^{(i)}) = -\frac{1}{n}\sum_{i=1}^n \log\left(\frac{\exp(\bar{h}_\theta(x^{(i)})_{y^{(i)}})}{\sum_{s=1}^k \exp(\bar{h}_\theta(x^{(i)})_s)}\right).$$

Suppose $e_y \in \mathbb{R}^k$ is the one-hot embedding/representation of the discrete label y, where the y-th entry is 1 and all other entries are zeros. We can also write the loss function in the following way:

$$J(W^{[1]}, W^{[2]}, b^{[1]}, b^{[2]}) = -\frac{1}{n}\sum_{i=1}^n e_{y^{(i)}}^\top \log\left(h_\theta(x^{(i)})\right).$$

Here $\log(\cdot)$ is applied entry-wise to the vector $h_\theta(x^{(i)})$. The starter code already converts labels into one-hot representations for you.

Instead of batch gradient descent or stochastic gradient descent, the common practice is to use mini-batch gradient descent for deep learning tasks. Concretely, we randomly sample B examples $(x^{(i_b)}, y^{(i_b)})_{b=1}^B$ from $(x^{(i)}, y^{(i)})_{i=1}^n$. In this case, the mini-batch cost function with batch-size B is defined as follows:

$$J_{MB} = \frac{1}{B}\sum_{b=1}^B \ell_{\text{CE}}(\bar{h}_\theta(x^{(i_b)}), y^{(i_b)})$$

where B is the batch size, i.e., the number of training examples in each mini-batch.

(a) **[5 points]**

Let $t \in \mathbb{R}^k, y \in \{1, \dots, k\}$ and $p = \text{softmax}(t)$. Prove that

$$\frac{\partial \ell_{\text{CE}}(t, y)}{\partial t} = p - e_y \in \mathbb{R}^k, \tag{10}$$

[3]Please note that the dimension of the weight matrices is different from those in the lecture notes, but we also multiply ${W^{[1]}}^\top$ instead of $W^{[1]}$ in the matrix multiplication layer. Such a change of notation is mostly for some consistence with the convention in the code.

where $e_y \in \mathbb{R}^k$ is the one-hot embedding of y, (where the y-th entry is 1 and all other entries are zeros.) As a direct consequence,

$$\frac{\partial \ell_{\text{CE}}(\bar{h}_\theta(x^{(i)}), y^{(i)})}{\partial \bar{h}_\theta(x^{(i)})} = \text{softmax}(\bar{h}_\theta(x^{(i)})) - e_{y^{(i)}} = h_\theta(x^{(i)}) - e_{y^{(i)}} \in \mathbb{R}^k \tag{11}$$

where $\bar{h}_\theta(x^{(i)}) \in \mathbb{R}^k$ is the input to the softmax function, i.e.

$$h_\theta(x^{(i)}) = \text{softmax}(\bar{h}_\theta(x^{(i)}))$$

(Note: in deep learning, $\bar{h}_\theta(x^{(i)})$ is sometimes referred to as the"logits".)

Answer: We have

$$\begin{aligned}\ell_{\mathsf{CE}}(t, y) &= -\log\left(\frac{\exp(t_y)}{\sum_{s=1}^k \exp(t_s)}\right)\\ &= -t_y + \log\left(\sum_{s=1}^k \exp(t_s)\right)\end{aligned}$$

and take the derivative with respect to t:

$$\begin{aligned}\frac{\partial \ell_{\mathsf{CE}}(t, y)}{\partial t} &= -\frac{\partial t_y}{\partial t} + \frac{\partial \log\left(\sum_{s=1}^k \exp(t_s)\right)}{\partial t}\\ &= -e_y + \frac{1}{\sum_{s=1}^k \exp(t_s)} \frac{\partial \sum_{s=1}^k \exp(t_s)}{\partial t}\\ &= -e_y + \frac{1}{\sum_{s=1}^k \exp(t_s)} \sum_{s=1}^k \frac{\partial \exp(t_s)}{\partial t}\\ &= -e_y + \frac{1}{\sum_{s=1}^k \exp(t_s)} \sum_{s=1}^k \exp(t_s) e_s\\ &= -e_y + \sum_{s=1}^k \frac{\exp(t_s)}{\sum_{s=1}^k \exp(t_s)} e_s\\ &= -e_y + \sum_{s=1}^k p_s e_s\\ &= -e_y + p\\ &= p - e_y\end{aligned}$$

There we have proved that $\frac{\partial \ell_{\mathsf{CE}}(t,y)}{\partial t} = p - e_y$.

(b) **[15 points]**

Implement both forward-propagation and back-propagation for the above loss function $J_{MB} = \frac{1}{B}\sum_{b=1}^{B} \ell_{\text{CE}}(\bar{h}_\theta(x^{(i_b)}), y^{(i_b)})$ Initialize the weights of the network by sampling values from a standard normal distribution. Initialize the bias/intercept term to 0. Set the number of hidden units to be 300, and learning rate to be 5. Set $B = 1,000$ (mini batch size). This means that we train with 1,000 examples in each iteration. Therefore, for each epoch, we need 50 iterations to cover the entire training data. The images are pre-shuffled. So you don't need to randomly sample the data, and can just create mini-batches sequentially.

Train the model with mini-batch gradient descent as described above. Run the training for 30 epochs. At the end of each epoch, calculate the value of loss function averaged over the entire training set, and plot it (y-axis) against the number of epochs (x-axis). In the same image, plot the value of the loss function averaged over the dev set, and plot it against the number of epochs.

Similarly, in a new image, plot the accuracy (on y-axis) over the training set, measured as the fraction of correctly classified examples, versus the number of epochs (x-axis). In the same image, also plot the accuracy over the dev set versus number of epochs.

Submit the two plots (one for loss vs epoch, another for accuracy vs epoch) in your writeup.

Also, at the end of 30 epochs, save the learnt parameters (i.e., all the weights and biases) into a file, so that next time you can directly initialize the parameters with these values from the file, rather than re-training all over. You do NOT need to submit these parameters.

Hint: Be sure to vectorize your code as much as possible! Training can be very slow otherwise. For better vectorization, use one-hot label encodings in the code (e_y in part (a)).

Answer:

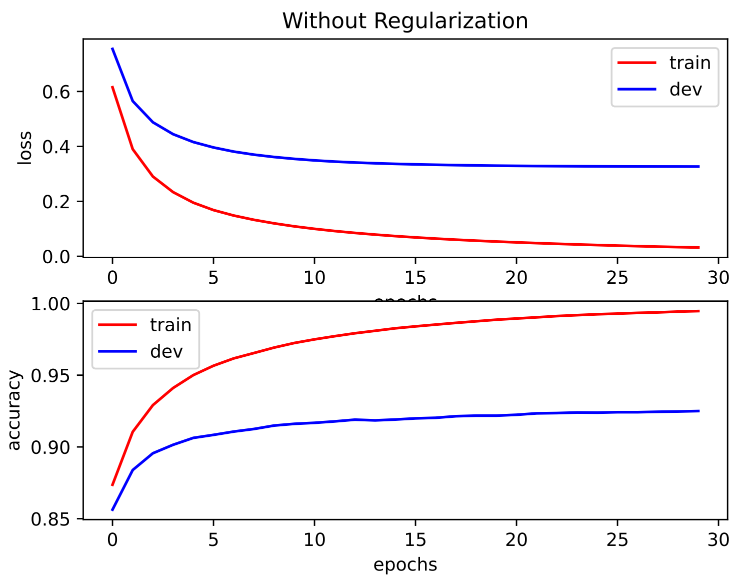

Figure 10: Baseline model

(c) **[7 points]** Now add a regularization term to your cross entropy loss. The loss function will become

$$J_{MB} = \left(\frac{1}{B} \sum_{b=1}^{B} \ell_{\text{CE}}(\bar{h}_\theta(x^{(i_b)}), y^{(i_b)}) \right) + \lambda \left(||W^{[1]}||^2 + ||W^{[2]}||^2 \right)$$

Be careful not to regularize the bias/intercept term. Set λ to be 0.0001. Implement the regularized version and plot the same figures as part (a). Be careful NOT to include the regularization term to measure the loss value for plotting (i.e., regularization should only be used for gradient calculation for the purpose of training).

Submit the two new plots obtained with regularized training (i.e loss (without regularization term) vs epoch, and accuracy vs epoch) in your writeup.

Compare the plots obtained from the regularized model with the plots obtained from the non-regularized model, and summarize your observations in a couple of sentences.

As in the previous part, save the learnt parameters (weights and biases) into a different file so that we can initialize from them next time.

Answer:

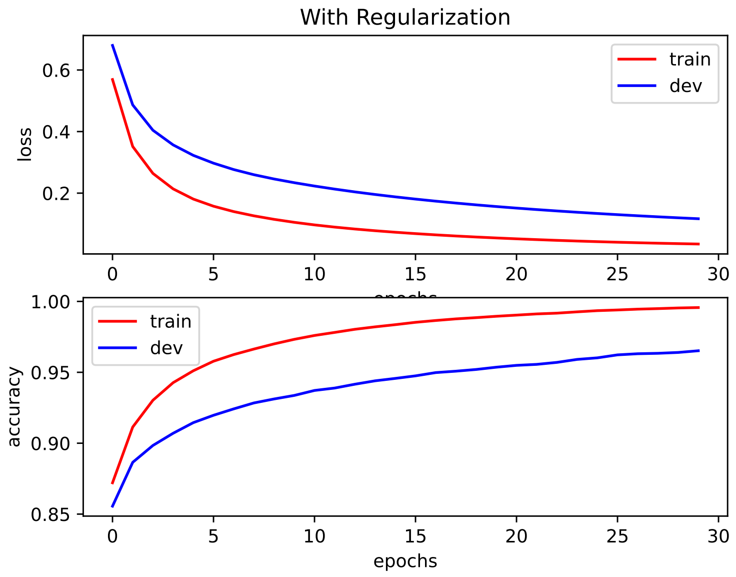

Figure 11: Regularized model

The regularized model has higher accuracy on the test set than the baseline model, which means regularization is effective in preventing overfitting.

(d) **[3 points]** All this while you should have stayed away from the test data completely. Now that you have convinced yourself that the model is working as expected (i.e., the observations you made in the previous part matches what you learnt in class about regularization), it is finally time to measure the model performance on the test set. Once we measure the test set performance, we report it (whatever value it may be), and NOT go back and refine the model any further.

Initialize your model from the parameters saved in part (a) (i.e., the non-regularized model), and evaluate the model performance on the test data. Repeat this using the parameters saved in part (b) (i.e., the regularized model).

Report your test accuracy for both regularized model and non-regularized model. Briefly (in one sentence) explain why this outcome makes sense. You should have accuracy close to 0.92870 without regularization, and 0.96760 with regularization. Note: these accuracies assume you implement the code with the matrix dimensions as specified in the comments, which is not the same way as specified in your code. Even if you do not precisely these numbers, you should observe good accuracy and better test accuracy with regularization.

Answer: For model baseline, got accuracy: 0.928700

For model regularized, got accuracy: 0.967600

Regularization prevents overfitting and therefore improves the accuracy on the test set.

5. **[12 points] Double Descent on Linear Models**

Note: This question may require knowledge on double descent that is covered on Wed of Week 5.

Background: In this question, you will empirically observe the sample-wise double descent phenomenon. That is, the validation losses of some learning algorithms or estimators do not monotonically decrease as we have more training examples, but instead have a curve with two U-shaped parts. The double descent phenomenon can be observed even for simple linear models. In this question, we consider the following setup. Let $\{(x^{(i)}, y^{(i)})\}_{i=1}^n$ be the training dataset. Let $X \in \mathbb{R}^{n\times d}$ be the matrix representing the inputs (i.e., the i-th row of X corresponds to $x^{(i)}$)), and $\vec{y} \in \mathbb{R}^n$ the vector representing the labels (i.e., the i-th row of $\vec{y}$ corresponds to $y^{(i)}$)):

$$X = \begin{bmatrix} - & x^{(1)} & - \\ - & x^{(2)} & - \\ \vdots & \vdots & \vdots \\ - & x^{(n)} & - \end{bmatrix}, \qquad \vec{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(n)} \end{bmatrix}.$$

Similarly, we use $X_v \in \mathbb{R}^{m\times d}, \vec{y}_v \in \mathbb{R}^m$ to represent the validation dataset, where m is the size of the validation dataset. We assume that the data are generated with $d = 500$.

In this question, we consider *regularized* linear regression. For a regularization level $\lambda \geq 0$, define the regularized cost function

$$J_\lambda(\beta) = \frac{1}{2}\|X\beta - \vec{y}\|_2^2 + \frac{\lambda}{2}\|\beta\|_2^2,$$

and its minimizer $\hat{\beta}_\lambda = \arg\min_{\beta\in\mathbb{R}^d} J_\lambda(\beta)$.

(a) [2 points] In this sub-question, we derive the closed-form solution of $\hat{\beta}_\lambda$. **Prove** that when $\lambda > 0$,

$$\hat{\beta}_\lambda = (X^\top X + \lambda I_{d\times d})^{-1} X^\top \vec{y} \tag{12}$$

(recall that $I_{d\times d} \in \mathbb{R}^{d\times d}$ is the identity matrix.)

Note: $\lambda = 0$ is a special case here. When $\lambda = 0$, $(X^\top X + \lambda I_{d\times d})$ could be singular. Therefore, there might be more than one solutions that minimize $J_0(\beta)$. In this case, we define $\hat{\beta}_0$ in the following way:

$$\hat{\beta}_0 = (X^\top X)^+ X^\top \vec{y}. \tag{13}$$

where $(X^\top X)^+$ denotes the Moore-Penrose pseudo-inverse of $X^\top X$. You don't need to prove the case when $\lambda = 0$, but this definition is useful in the following sub-questions.

Answer: To minimize the cost function $J_\lambda(\beta) = \frac{1}{2}\|X\beta - \vec{y}\|_2^2 + \frac{\lambda}{2}\|\beta\|_2^2$, we take the derivative with respect to β and set it to zero:

$$\begin{aligned} \nabla_\beta J_\lambda(\beta) &= X^T(X\beta - \vec{y}) + \lambda\beta \\ &= X^T X\beta - X^T\vec{y} + \lambda\beta \\ &= (X^T X + \lambda I)\beta - X^T\vec{y} = 0 \\ \implies \hat{\beta}_\lambda &= (X^T X + \lambda I)^{-1} X^T \vec{y} \end{aligned}$$

Proved.

(b) [5 points] **Coding question: the double descent phenomenon for unregularized models**

In this sub-question, you will empirically observe the double descent phenomenon.You are given 13 training datasets of sample sizes $n = 200, 250, \ldots, 750$, and 800, and a validation dataset, located at

- `src/doubledescent/train200.csv`, `train250.csv`, etc.
- `src/doubledescent/validation.csv`

For each training dataset $(X, \vec{y})$, compute the corresponding $\hat{\beta}_0$, and evaluate the mean squared error (MSE) of $\hat{\beta}_0$ on the validation dataset. The MSE for your estimators $\hat{\beta}$ on a validation dataset $(X_v, \vec{y}_v)$ of size m is defined as:

$$\text{MSE}(\hat{\beta}) = \frac{1}{2m} \|X_v \hat{\beta} - \vec{y}_v\|_2^2.$$

Complete the `regression` method of `src/doubledescent/doubledescent.py` which takes in a training file and a validation file, and computes $\hat{\beta}_0$. You can use `numpy.linalg.pinv` to compute the pseudo-inverse.

In your writeup, include a line plot of the validation losses. The x-axis is the size of the training dataset (from 200 to 800); the y-axis is the MSE on the validation dataset. You should observe that the validation error increases and then decreases as we increase the sample size.

Note: When $n \approx d$, the test MSE could be very large. For better visualization, it is okay if the test MSE goes out of scope in the plot for some points. **Answer:**

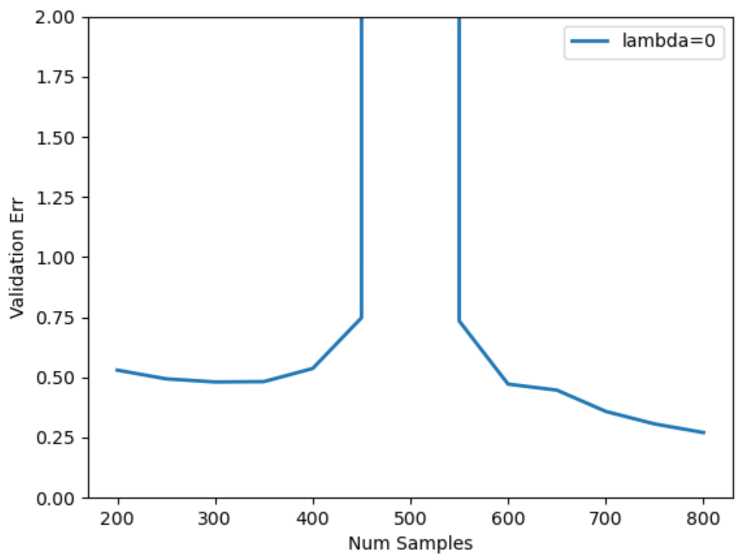

Figure 12: Unregularized model. $\lambda = 0$

(c) [5 points] **Coding question: double descent phenomenon and the effect of regularization.**

In this sub-question, we will show that regularization mitigates the double descent phenomenon for linear regression. We will use the same datasets as specified in sub-question (b). Now consider using various regularization strengths. For $\lambda \in \{0, 1, 5, 10, 50, 250, 500, 1000\}$, you will compute the minimizer of $J_\lambda(\beta)$.

Complete the `ridge_regression` method of `src/doubledescent/doubledescent.py` which takes in a training file and a validation file, computes the $\hat{\beta}_\lambda$ that minimizes the training objective under different regularization strengths, and returns a list of validation errors (one for each choice of λ).

In your writeup, include a plot of the validation losses of these models. The x-axis is the size of the training dataset (from 200 to 800); the y-axis is the MSE on the validation dataset. Draw one line for each choice of λ connecting the validation errors across different training dataset sizes. Therefore, the plot should contain 8×13 points and 8 lines connecting them.

You should observe that for some small λ's, the validation error may increase and then decrease as we increase the sample size. However, double descent does not occur for a relatively large λ.

Remark: If you want to learn more about the double descent phenomenon and the effect of regularization, you can start with this paper Nakkiran, et al. 2020. **Answer:**

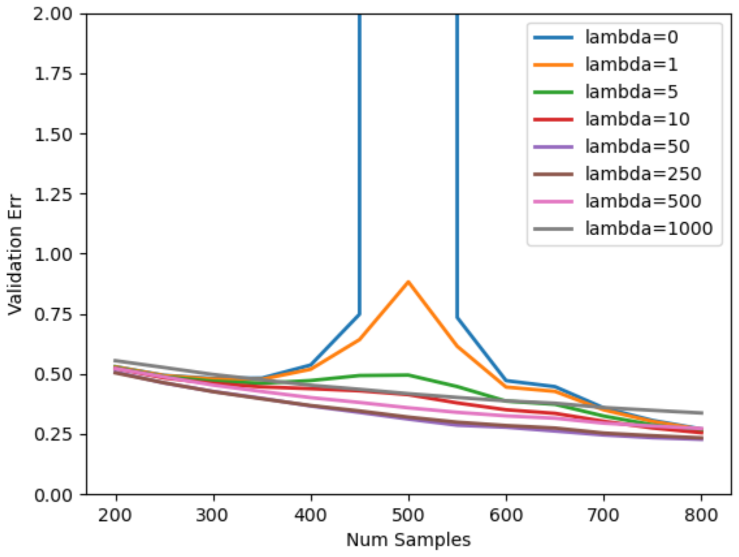

Figure 13: Models with various regularization strengths.