CS 229, Summer 2023
Problem Set #2 Solutions

YOUR NAME HERE (`YOUR SUNET HERE`)

Due Friday, July 28th 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/41182/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 Friday, July 28th at 11:59 pm. If you submit after Friday, July 28th 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.

[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. **[20 points] Spam classification**

In this problem, we will use the naive Bayes algorithm to build a spam classifier.

In recent years, spam on electronic media has been a growing concern. Here, we'll build a classifier to distinguish between real messages, and spam messages. For this class, we will be building a classifier to detect SMS spam messages. We will be using an SMS spam dataset developed by Tiago A. Almedia and José María Gómez Hidalgo which is publicly available on `http://www.dt.fee.unicamp.br/~tiago/smsspamcollection` [3]

We have split this dataset into training and testing sets and have included them in this assignment as `src/spam/spam_train.tsv` and `src/spam/spam_test.tsv`. See `src/spam/spam_readme.txt` for more details about this dataset. Please refrain from redistributing these dataset files. The goal of this assignment is to build a classifier from scratch that can tell the difference the spam and non-spam messages using the text of the SMS message.

(a) [5 points] Implement code for processing the the spam messages into numpy arrays that can be fed into machine learning models. Do this by completing the `get_words`, `create_dictionary`, and `transform_text` functions within our provided `src/spam.py`. Do note the corresponding comments for each function for instructions on what specific processing is required.

The provided code will then run your functions and save the resulting dictionary into `spam_dictionary` and a sample of the resulting training matrix into `spam_sample_train_matrix`. In your writeup, report the vocabulary size after the preprocessing step. You do not need to include any other output for this subquestion. **Answer:**

Size of dictionary: 1722

(b) [10 points] In this question you are going to implement a naive Bayes classifier for spam classification with **multinomial event model** and Laplace smoothing.

Code your implementation by completing the `fit_naive_bayes_model` and `predict_from_naive_bayes_model` functions in `src/spam/spam.py`.

Now `src/spam/spam.py` should be able to train a Naive Bayes model, compute your prediction accuracy and then save your resulting predictions to `spam_naive_bayes_predictions`.

In your writeup, report the accuracy of the trained model on the **test set**.

Remark. If you implement naive Bayes the straightforward way, you will find that the computed $p(x|y) = \prod_i p(x_i|y)$ often equals zero. This is because $p(x|y)$, which is the product of many numbers less than one, is a very small number. The standard computer representation of real numbers cannot handle numbers that are too small, and instead rounds them off to zero. (This is called "underflow.") You'll have to find a way to compute Naive Bayes' predicted class labels without explicitly representing very small numbers such as $p(x|y)$. [**Hint:** Think about using logarithms.]

Answer:

Naive Bayes had an accuracy of 0.978494623655914 on the testing set.

In Naive-Bayes we can avoid the problem of "underflow" by considering $\log P(x|y) = \sum_i \log P(x_i|y_i)$ instead of $P(x|y) = \prod_i P(x_i|y_i)$. The *log* function makes the product of small numbers a sum, and the obtained values can be stored in a computer's memory optimally. In this case the model

[3] Almeida, T.A., Gómez Hidalgo, J.M., Yamakami, A. Contributions to the Study of SMS Spam Filtering: New Collection and Results. Proceedings of the 2011 ACM Symposium on Document Engineering (DOCENG'11), Mountain View, CA, USA, 2011.

is using the following expressions:

$$\log P(y=1|x) = \sum_{k=1}^{n} x_k \log(\phi_{k|y=1}) + \log(\phi_y) \tag{1}$$

$$\log P(y=0|x) = \sum_{k=1}^{n} x_k \log(\phi_{k|y=0}) + \log(1-\phi_y) \tag{2}$$

This approach overall allows Naive-Bayes to operate with high featured inputs.

(c) [5 points] Intuitively, some tokens may be particularly indicative of an SMS being in a particular class. We can try to get an informal sense of how indicative token i is for the SPAM class by looking at:

$$\log \frac{p(x_j = i \mid y = 1)}{p(x_j = i \mid y = 0)} = \log \left(\frac{P(\text{token } i \mid \text{email is SPAM})}{P(\text{token } i \mid \text{email is NOTSPAM})} \right).$$

Complete the `get_top_five_naive_bayes_words` function within the provided code using the above formula in order to obtain the 5 most indicative tokens. Report the top five words in your writeup.

Answer:

The top 5 indicative words for Naive Bayes are: ['claim', 'won', 'prize', 'tone', 'urgent!']

2. **[18 points] Constructing kernels**

In class, we saw that by choosing a kernel $K(x, z) = \phi(x)^T\phi(z)$, we can implicitly map data to a high dimensional space, and have a learning algorithm (e.g., SVM or logistic regression) work in that space. One way to generate kernels is to explicitly define the mapping ϕ to a higher dimensional space, and then work out the corresponding K.

However, in this question, we are interested in direct construction of kernels. I.e., suppose we have a function $K(x, z)$ that we think gives an appropriate similarity measure for our learning problem, and we are considering plugging K into the SVM as the kernel function. However, for $K(x, z)$ to be a valid kernel, it must correspond to an inner product in some higher dimensional space resulting from some feature mapping ϕ. Mercer's theorem tells us that $K(x, z)$ is a (Mercer) kernel if and only if for any finite set $\{x^{(1)}, \ldots, x^{(n)}\}$, the square matrix $K \in \mathbb{R}^{n\times n}$ whose entries are given by $K_{ij} = K(x^{(i)}, x^{(j)})$ is symmetric and positive semidefinite. You can find more details about Mercer's theorem in the notes, though the description above is sufficient for this problem. In this question we are interested to see which operations preserve the validity of kernels.

Let K_1, K_2 be kernels over $\mathbb{R}^d \times \mathbb{R}^d$, let $a \in \mathbb{R}^+$ be a positive real number, let $f : \mathbb{R}^d \mapsto \mathbb{R}$ be a real-valued function, let $\phi : \mathbb{R}^d \mapsto \mathbb{R}^p$ be a function mapping from $\mathbb{R}^d$ to $\mathbb{R}^p$, let K_3 be a kernel over $\mathbb{R}^p \times \mathbb{R}^p$, and let $p(x)$ a polynomial over x with *positive* coefficients.

For each of the functions K below, state whether it is necessarily a kernel. If you think it is, prove it; if you think it isn't, give a counter-example.

(a) [1 points] $K(x, z) = K_1(x, z) + K_2(x, z)$

(b) [1 points] $K(x, z) = K_1(x, z) - K_2(x, z)$

(c) [1 points] $K(x, z) = aK_1(x, z)$

(d) [1 points] $K(x, z) = -aK_1(x, z)$

(e) [5 points] $K(x, z) = K_1(x, z)K_2(x, z)$

(f) [3 points] $K(x, z) = f(x)f(z)$

(g) [3 points] $K(x, z) = K_3(\phi(x), \phi(z))$

(h) [3 points] $K(x, z) = p(K_1(x, z))$

[**Hint:** For part (e), the answer is that K *is* indeed a kernel. You still have to prove it, though. (This one may be harder than the rest.) This result may also be useful for another part of the problem.]

Answer:

(a) The first expression $K(x, z) = K_1(x, z) = K_2(x, z)$ is necessarily a valid kernel. Let's focus on the respective matrices K, K_1, K_2: the sum of two symmetric matrices is symmetric, the sum of two PSD matrices is still PSD.
Consider the following quadratic form $z^T Kz$, then

$$z^T Kz = z^T(K_1 + K_2)z = z^T K_1 z + z^T K_2 z \geq 0 \tag{3}$$

$$K = (K_1 + K_2) = (K_1 + K_2)^T = K^T \tag{4}$$

With K_1 and K_2 being valid kernels.

(b) The subtraction of two kernel matrices is not necessarily a valid kernel itself. Let's consider the case where the diagonal elements of K_2 are greater than the ones in K_1, than the resulting K

would no longer be PSD, hence not a valid kernel. As in the previous case, consider the following quadratic form $z^T K z$, then

$$z^T K z = z^T (K_1 + K_2) z = z^T K_1 z - z^T K_2 z \geq 0 \iff z^T K_1 z > z^T K_2 z \tag{5}$$
$$\tag{6}$$

With K_1 and K_2 being valid kernels.

(c) The resulting K matrix is necessarily a valid kernel since $aK_1 = aK_1^T$ and $aK_1 \geq 0$ (PSD) and, from hypothesis, we know that $a \in \mathbb{R}^+$. In fact

$$z^T K z = z^T (aK_1) z = a z^T K_1 z \geq 0 \tag{7}$$
$$K = (aK_1) = (aK_1) = K^T \tag{8}$$

(d) The resulting K matrix is not a valid kernel since $-aK_1 \leq 0$ (not PSD) and, from hypothesis, we know that $-a \in \mathbb{R}^-$. In fact

$$z^T K z = z^T (-aK_1) z = -a z^T K_1 z \ngeq 0 \tag{9}$$
$$\tag{10}$$

(e) We can expand the two valid kernels functions K_1 and K_2 as follows

$$K(x, z) = \langle \phi_1(x), \phi_1(z) \rangle \langle \phi_2(x), \phi_2(z) \rangle \tag{11}$$

And, in vector notation, we can express this as

$$\sum_{i=1}^{n} \sum_{j=1}^{n} \phi_1^{(i)}(x) \phi_1^{(i)}(z) \phi_2^{(j)}(x) \phi_2^{(j)}(z) \tag{12}$$

Let's define two new terms

$$\Omega_1^{i,j} = \phi_1^{(i)}(x) \phi_2^{(j)}(x) \tag{13}$$
$$\Omega_2^{i,j} = \phi_1^{(i)}(z) \phi_2^{(j)}(z) \tag{14}$$

And express the result in terms of a new feature map $\Omega(y) = \phi(y)\phi(y) \quad y \in \mathbb{R}^d$

$$K(x, z) = \sum_{i=1}^{n} \sum_{j=1}^{n} \Omega_1^{i,j}(x) \Omega_2^{i,j}(z) = \langle \Omega(x), \Omega(z) \rangle \tag{15}$$

We just showed that $K(x, z)$ is expressible in terms of an inner product, hence it is a valid kernel.

(f) Let's take in consideration the expression $K(x, z) = f(x) f(z)$. From hypothesis, $f : \mathbb{R}^d \mapsto \mathbb{R}$, so we can see that f maps the vectors in a sigle-dimensional feature space, and the inner product in $\mathbb{R}$ is exactly $f(x) f(z)$. We showed that K is mathematically equivalent to an inner product, hence $K(x, z)$ is necessarily a valid kernel function.

(g) The expression $K(x,z) = K_3(\phi(x)\phi(z))$ is necessarily a kernel, since K_3 is a kernel operating in a $\mathbb{R}^p \times \mathbb{R}^p$ space and ϕ is a feature map that transforms d-dimensional input vectors in p-dimensional outputs. $\phi(x)$, $\phi(z)$ are dimensionally compatible with the K_3 function in which they serve as parameters. K is a valid kernel.

(h) In the case where $p(x)$ is a polynomial over x with positive coefficients from hypothesis,the expression $K(x,z) = p(K_1(x,z))$ necessarily represents a valid kernel.

$$p(K_1(x,z)) = a + b[K_1(x,z)] + c[K_1(x,z)]^2 + ... + \xi[K_1(x,z)]^n \tag{16}$$

Recall points (a), (c) and (e). Each member of the polynomial is a valid kernel and the sum of them still is necessarily a kernel.

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:

i. Since $\theta^{(i)}$ is now a p-dimensional vector (it has the same dimension as the feature vectors $\phi(x)$) and p could be an extremely big value (even uncountably infinite), we cannot compute it right the way. We can represent it as the linear combination of a coefficient β (that depends on the learning grade α) and the feature vector $\phi(x)$ as follows

$$\theta^{(i)} = \sum_{j=1}^{n} \beta_j \phi(x^{(j)}) \tag{17}$$

The initial value $\theta^{(0)} = 0$ can be obtained by setting all the β_j's equal to zero.

ii. In order to make predictions about a new input $x^{(i+1)}$ we first need to compute the hypothesis class $h_{\theta^{(i)}}(x^{(i+1)}) = g\left({\theta^{(i)}}^T \phi(x^{(i+1)})\right)$. By using the expression obtain in the previous point we get

$$h_{\theta^{(i)}}(x^{(i+1)}) = g\left(\left[\sum_{j=1}^{n} \beta_j \phi(x^{(j)})\right]\phi(x^{(i+1)})\right) = g\left(\sum_{j=1}^{n} \beta_j K(x^{(j)}, x^{(i+1)})\right) \tag{18}$$

Where $g(z) = sign(z)$, this expression does not require the explicit calculation of $\phi(x)$

iii. Since we are utilizing a Kernel function, we don't need to calculate $\phi(x^{(i+1)})$ explicitly. All we need to do is to add an example $\{x^{(i+1)}, y^{(i+1)}\}$ to the list of the misclassified example if the prediction turns out to be incorrect. The update rule would look something like this

$$\theta^{(i+1)} = \sum_{j=1}^{n+1} \beta_j \phi(x^{(j)}) = \sum_{j=1}^{n} \beta_j \phi(x^{(j)}) + \beta_{j+1}\phi(x^{(j+1)}) \tag{19}$$

Where β_{j+1} corresponds to

$$\beta_{j+1} = \alpha(y^{(i+1)} - g\Big(\sum_{j=1}^{n} \beta_j K(x^{(j)}, x^{(i+1)})\Big))\phi(x^{(i+1)}) \tag{20}$$

The above notation corresponds to a close form expression of the generic updated parameter $\theta^{(i+1)}$ (which is also implicitly updated by updating the linear combination term β_j itself).

(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:

$$K(x, z) = x^\top z, \tag{21}$$

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

$$K(x, z) = \exp\left(-\frac{\|x - z\|_2^2}{2\sigma^2}\right), \tag{22}$$

and finally the following function:

$$K(x, z) = \begin{cases} -1 & x = z \\ 0 & x \neq z \end{cases} \tag{23}$$

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:

(c) [2 points] One of the choices in Q4b 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 linear kernel $K(x, z) = x^T z$ is a poor choice for this kind of classification problem, since it can only lead to a hyperplane (or approximatively) to separate the points. Hence, this is not optimal.

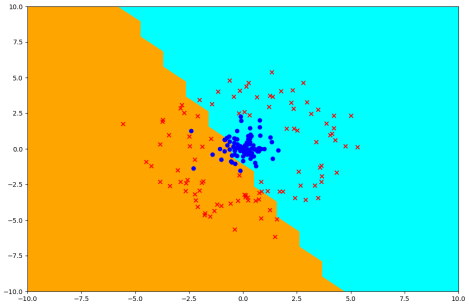

Figure 1: Linear kernel (dot)

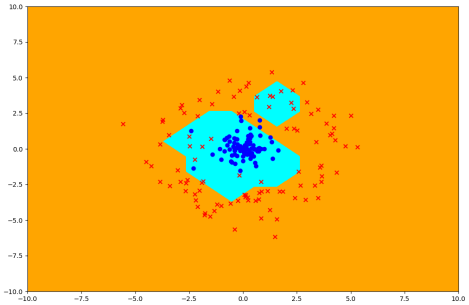

Figure 2: RBF kernel

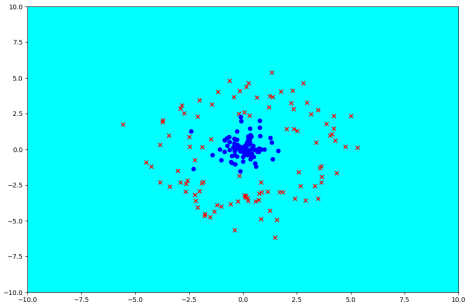

Figure 3: Non-PSD function

On the other hand, the RBF Kernel does really good on this data, it can build one or more circuilar (elliptic) decision boundaries to separate the two classes precisely.
The last kernel totally fails his job, since a kernel to be valid should satisfy Mercer's conditions, hence being PSD and symmetric. The classifier cannot even be built up (it cannot be geometrically represented).

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

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`

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. Use the sigmoid function as activation for the hidden layer, and softmax function for the output layer. Recall that for a single example (x, y), the cross entropy loss is:

$$CE(y, \hat{y}) = -\sum_{k=1}^{K} y_k \log \hat{y}_k,$$

where $\hat{y} \in \mathbb{R}^K$ is the vector of softmax outputs from the model for the training example x, and $y \in \mathbb{R}^K$ is the ground-truth vector for the training example x such that $y = [0, ..., 0, 1, 0, ..., 0]^\top$ contains a single 1 at the position of the correct class (also called a "one-hot" representation).

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 $y^{(i)} \in \mathbb{R}^K$ is a

one-hot vector as described above. Let h be the number of hidden units in the neural network, so that weight matrices $W^{[1]} \in \mathbb{R}^{d\times h}$ and $W^{[2]} \in \mathbb{R}^{h\times K}$. We also have biases $b^{[1]} \in \mathbb{R}^h$ and $b^{[2]} \in \mathbb{R}^K$. The forward propagation equations for a single input $x^{(i)}$ then are:

$$\begin{aligned} a^{(i)} &= \sigma\left({W^{[1]}}^\top x^{(i)} + b^{[1]}\right) \in \mathbb{R}^h \\ z^{(i)} &= {W^{[2]}}^\top a^{(i)} + b^{[2]} \in \mathbb{R}^K \\ \hat{y}^{(i)} &= \text{softmax}(z^{(i)}) \in \mathbb{R}^K \end{aligned}$$

where σ is the sigmoid function.

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

$$J(W^{[1]}, W^{[2]}, b^{[1]}, b^{[2]}) = \frac{1}{n}\sum_{i=1}^{n} CE(y^{(i)}, \hat{y}^{(i)}) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K} y_k^{(i)} \log \hat{y}_k^{(i)}.$$

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. In this case, the cost function is defined as follows:

$$J_{MB} = \frac{1}{B}\sum_{i=1}^{B} CE(y^{(i)}, \hat{y}^{(i)})$$

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

(a) **[5 points]**

For a single input example $x^{(i)}$ with one-hot label vector $y^{(i)}$, show that

$$\nabla_{z^{(i)}} \text{CE}(y^{(i)}, \hat{y}^{(i)}) = \hat{y}^{(i)} - y^{(i)} \in \mathbb{R}^K$$

where $z^{(i)} \in \mathbb{R}^K$ is the input to the softmax function, i.e.

$$\hat{y}^{(i)} = \text{softmax}(z^{(i)})$$

(Note: in deep learning, $z^{(i)}$ is sometimes referred to as the "logits".)

Hint: To simplify your answer, it might be convenient to denote the true label of $x^{(i)}$ as $l \in \{1, \ldots, K\}$. Hence l is the index such that that $y^{(i)} = [0, ..., 0, 1, 0, ..., 0]^\top$ contains a single 1 at the l-th position. You may also wish to compute $\dfrac{\partial \text{CE}(y^{(i)}, \hat{y}^{(i)})}{\partial z_j^{(i)}}$ for $j \neq l$ and $j = l$ separately.

Answer:

We can start from the definition of the softmax function

$$\hat{y}_j^{(i)} = \text{softmax}(z_j^{(i)}) = \frac{e^{z_j^{(i)}}}{\sum_{k=1}^{K} e^{z_j^{(i)}}} \tag{24}$$

And the Cross-Entropy loss of a single example can be expressed as

$$\text{CE}(y^{(i)}, \hat{y}^{(i)}) = -\sum_{k=1}^{K} y_k^{(i)} \log \hat{y}_k^{(i)} \tag{25}$$

We need to compute the gradient of the CE loss with respect to $z^{(i)}$

$$\nabla_{z^{(i)}} \mathrm{CE}(y^{(i)}, \hat{y}^{(i)}) \tag{26}$$

Let's consider the differenciated single i-th elements of the gradient. In the case where $k = j$, we have

$$\frac{\partial \hat{y}_k^{(i)}}{\partial z_k^{(i)}} = \frac{e^{z_j^{(i)}} \sum_{k=1}^{K} e^{z_k^{(i)}} - e^{z_j^{(i)}} e^{z_j^{(i)}}}{\left(\sum_{k=1}^{K} e^{z_k^{(i)}}\right)^2} = \hat{y}_k^{(i)}(1 - \hat{y}_k^{(i)}) \tag{27}$$

On the other side, if $k \neq j$, the derivative looks like this

$$\frac{\partial \hat{y}_k^{(i)}}{\partial z_j^{(i)}} = -\frac{e^{z_j^{(i)}} e^{z_k^{(i)}}}{\left(\sum_{k=1}^{K} e^{z_k^{(i)}}\right)^2} = -\hat{y}_k^{(i)} \hat{y}_j^{(i)} \tag{28}$$

By joining those two results together (chain rule of derivatives), we get

$$\frac{\partial \mathrm{CE}(y^{(i)}, \hat{y}^{(i)})}{\partial z_j^{(i)}} = \sum_{k=1}^{K} \frac{\partial \mathrm{CE}(y^{(i)}, \hat{y}^{(i)})}{\partial \hat{y}_k^{(i)}} \frac{\partial \hat{y}_k^{(i)}}{\partial z_j^{(i)}} \tag{29}$$

Since $\frac{\partial \mathrm{CE}(y^{(i)}, \hat{y}^{(i)})}{\partial \hat{y}_k^{(i)}} = -\frac{y_k^{(i)}}{\hat{y}_k^{(i)}}$ we can substitute and arrange the terms as follows

$$\frac{\partial \mathrm{CE}(y^{(i)}, \hat{y}^{(i)})}{\partial z_j^{(i)}} = -\sum_{k=1}^{K} y_k^{(i)} \left(\hat{y}_k^{(i)}\right)^{-1} \frac{\partial \hat{y}_k^{(i)}}{\partial z_j^{(i)}} = \hat{y}_j^{(i)} - y_j^{(i)} \tag{30}$$

We've shown that the gradient of the CE loss function with respect to the logit is

$$\nabla_{z^{(i)}} \mathrm{CE}(y^{(i)}, \hat{y}^{(i)}) = \hat{y}^{(i)} - y^{(i)} \tag{31}$$

Which is not other than the difference between the softmax output and the ground-truth label vector.

(b) **[15 points]**

Implement both forward-propagation and back-propagation for the above loss function $J_{MB} = \frac{1}{B} \sum_{i=1}^{B} CE(y^{(i)}, \hat{y}^{(i)})$. 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.

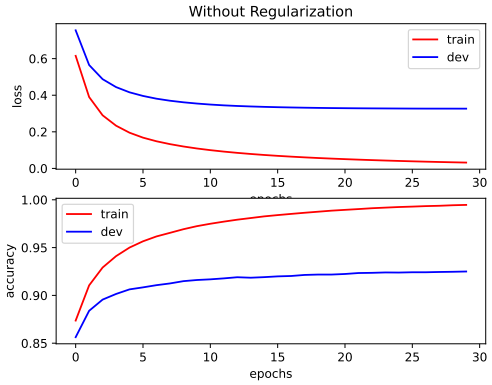

Figure 4: Unregularized

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.

Answer:

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

$$J_{MB} = \left(\frac{1}{B} \sum_{i=1}^{B} CE(y^{(i)}, \hat{y}^{(i)}) \right) + \lambda \left(||W^{[1]}||^2 + ||W^{[2]}||^2 \right)$$

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: Thanks to regularization, the loss function J_{MB} seems to stabilize and be asymptotic to a constant non-zero ordinate value. This implies the benefit of the regularization itself: the avoidance of overfitting while making the model much accurate on future predictions.

(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.

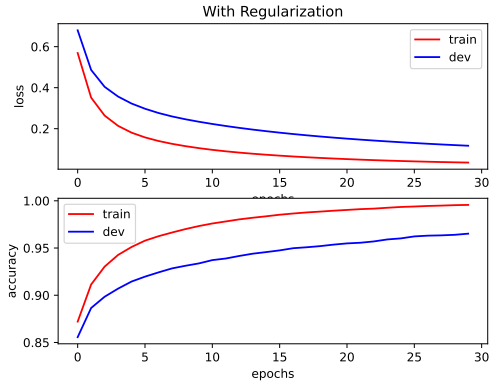

Figure 5: Regularized

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

Those values totally respect and reflect the point made up in the previous answer, the model is indeed being more accurate with regularization methods implemented.

5. **[20 points] Bayesian Interpretation of Regularization**

Background: In Bayesian statistics, almost every quantity is a random variable, which can either be observed or unobserved. For instance, parameters θ are generally unobserved random variables, and data x and y are observed random variables. The joint distribution of all the random variables is also called the *model* (*e.g.,* $p(x, y, \theta)$). Every unknown quantity can be estimated by conditioning the model on all the observed quantities. Such a conditional distribution over the unobserved random variables, conditioned on the observed random variables, is called the *posterior distribution*. For instance $p(\theta|x, y)$ is the posterior distribution in the machine learning context. A consequence of this approach is that we are required to endow our model parameters, *i.e.,* $p(\theta)$, with a *prior distribution*. The prior probabilities are to be assigned *before* we see the data—they capture our prior beliefs of what the model parameters might be before observing any evidence.

In the purest Bayesian interpretation, we are required to keep the entire posterior distribution over the parameters all the way until prediction, to come up with the *posterior predictive distribution*, and the final prediction will be the expected value of the posterior predictive distribution. However in most situations, this is computationally very expensive, and we settle for a compromise that is *less pure* (in the Bayesian sense).

The compromise is to estimate a point value of the parameters (instead of the full distribution) which is the mode of the posterior distribution. Estimating the mode of the posterior distribution is also called *maximum a posteriori estimation* (MAP). That is,

$$\theta_{\text{MAP}} = \arg\max_{\theta} p(\theta|x, y).$$

Compare this to the *maximum likelihood estimation* (MLE) we have seen previously:

$$\theta_{\text{MLE}} = \arg\max_{\theta} p(y|x, \theta).$$

In this problem, we explore the connection between MAP estimation, and common regularization techniques that are applied with MLE estimation. In particular, you will show how the choice of prior distribution over θ (*e.g.,* Gaussian or Laplace prior) is equivalent to different kinds of regularization (*e.g.,* L_2, or L_1 regularization). You will also explore how regularization strengths affect generalization in part (d).

(a) [3 points] Show that $\theta_{\text{MAP}} = \text{argmax}_\theta\, p(y|x, \theta)p(\theta)$ if we assume that $p(\theta) = p(\theta|x)$. The assumption that $p(\theta) = p(\theta|x)$ will be valid for models such as linear regression where the input x are not explicitly modeled by θ. (Note that this means x and θ are marginally independent, but not conditionally independent when y is given.)

Answer: Let's start from the statement made in the problem's hypothesis

$$\theta_{\text{MAP}} = \arg\ \max_\theta P(\theta|x, y) \tag{32}$$

By applying Bayes rule and the conditional probability to $P(\theta|x, y)$, we obtain

$$P(\theta|x, y) = \frac{P(y, x|\theta)P(\theta)}{P(y, x)} = \frac{P(\theta|x)P(y|\theta, x)}{P(y|x)} \tag{33}$$

In conclusion, we can express θ_{MAP} as follows

$$\theta_{\text{MAP}} = \arg\ \max_\theta P(\theta|x, y) = \arg\ \max_\theta P(\theta)P(y|x, \theta) \tag{34}$$

$P(\theta|x) = P(\theta)$ since in this case the input x are not explicitly modeled by θ and $P(y|x)$ is theta-independent, hence doesn't affect the arg max operation.

(b) [5 points] Recall that L_2 regularization penalizes the L_2 norm of the parameters while minimizing the loss (*i.e.,* negative log likelihood in case of probabilistic models). Now we will show that MAP estimation with a zero-mean Gaussian prior over θ, specifically $\theta \sim \mathcal{N}(0, \eta^2 I)$, is equivalent to applying L_2 regularization with MLE estimation. Specifically, show that for some scalar λ,

$$\theta_{\text{MAP}} = \arg\min_{\theta} -\log p(y|x,\theta) + \lambda||\theta||_2^2. \tag{35}$$

Also, what is the value of λ?

Answer: We can start by taking in consideration the expression we obtained in the previous point

$$\theta_{\text{MAP}} = \arg\max_{\theta} P(\theta)P(y|x,\theta) \tag{36}$$

By applying $-\log$ we can observe that minimizing the obtained expression is equivalent to maximizing the starting one

$$\theta_{\text{MAP}} = \arg\min_{\theta} -\log P(y|x,\theta) - \log P(\theta) \tag{37}$$

We know from hypothesis that $\theta \sim N(\vec{0}, \eta^2 I)$

$$-\log P(\theta) = -\log\left[\frac{1}{(2\pi)^{\frac{d}{2}}|\eta^2 I|^{\frac{1}{2}}} e^{-\frac{1}{2}\theta^T(\eta^2 I)^{-1}\theta}\right] = -\frac{d}{2}\log(2\pi\eta^2) - \frac{1}{2\eta^2}||\theta||_2^2 \tag{38}$$

Where the term $-\frac{d}{2}\log(2\pi\eta^2)$ is constant (doesn't affect the arg min operation), so it is safe to represent θ_{MAP} as follows

$$\theta_{\text{MAP}} = \arg\min_{\theta}(-\log P(y|x,\theta) + \frac{1}{2\eta^2}||\theta||_2^2) \tag{39}$$

We have successfully shown that the MAP estimation with a 0-mean gaussian prior distribution over θ is equivalent of applying L_2 norm regularization with MLE estimation. Furthermore the value of the hyperparameter lambda is

$$\lambda = \frac{1}{2\eta^2} \tag{40}$$

(c) [7 points] Now consider a specific instance, a linear regression model given by $y = \theta^T x + \epsilon$ where $\epsilon \sim \mathcal{N}(0, \sigma^2)$. Assume that the random noise $\epsilon^{(i)}$ is independent for every training example $x^{(i)}$. Like before, assume a Gaussian prior on this model such that $\theta \sim \mathcal{N}(0, \eta^2 I)$. For notation, let X be the design matrix of all the training example inputs where each row vector is one example input, and $\vec{y}$ be the column vector of all the example outputs.

Come up with a closed form expression for θ_{MAP}.

Answer: We can express the negative log likelihood from the gassian distribution as follows (ignoring the constant terms)

$$-\log P(y|x,\theta) = -\sum_{i=1}^{n}\log P(y^{(i)}|x^{(i)},\theta) = \frac{1}{2\sigma^2}\sum_{i=1}^{n}(y^{(i)} - \theta^T x^{(i)})^2 \tag{41}$$

$$-\log P(y|x,\theta) = \frac{1}{2\sigma^2}||\vec{y} - X\theta||_2^2 \tag{42}$$

Whereas the closed expression for the prior distribution was already obtained

$$-\log P(\theta) = \frac{1}{2\eta^2}||\theta||_2^2 \tag{43}$$

By joining them altogether we obtain a new function, the loss function $J(\theta)$. Recall that maximizing the likelihood equals to miniming the loss function, assuming the parameter distribution to be normal and zero-meaned.

$$J(\theta) = \frac{1}{2\sigma^2}||\vec{y} - X\theta||_2^2 + \frac{1}{2\eta^2}||\theta||_2^2 \tag{44}$$

In order to find the closed form of θ_{MAP} we need to take the gradient of the cost function and set it equal to zero

$$\nabla_\theta J(\theta) = 0 \tag{45}$$

$$\frac{1}{\sigma^2}X^T(X\theta - \vec{y}) + \frac{1}{\eta^2}\theta = 0 \tag{46}$$

Solving the equation for θ

$$\hat{\theta}_{\text{MAP}} = (X^TX + \frac{\sigma^2}{\eta^2}I)^{-1}X^T\vec{y} \tag{47}$$

Hence, the MAP estimation of θ in a linear regression model with gaussian noise and gaussian prior is given from the above expression.

(d) [5 points] Next, consider the Laplace distribution, whose density is given by

$$f_{\mathcal{L}}(z|\mu, b) = \frac{1}{2b}\exp\left(-\frac{|z-\mu|}{b}\right).$$

As before, consider a linear regression model given by $y = x^T\theta + \epsilon$ where $\epsilon \sim \mathcal{N}(0, \sigma^2)$. Assume a Laplace prior on this model, where each parameter θ_i is marginally independent, and is distributed as $\theta_i \sim \mathcal{L}(0, b)$.

Show that θ_{MAP} in this case is equivalent to the solution of linear regression with L_1 regularization, whose loss is specified as

$$J(\theta) = ||X\theta - \vec{y}||_2^2 + \gamma||\theta||_1$$

Also, what is the value of γ?

Note: A closed form solution for linear regression problem with L_1 regularization does not exist. To optimize this, we use gradient descent with a random initialization and solve it numerically.

Answer: This time the model's parameter is defined over a Laplace distribution $\theta \sim \mathcal{L}(\vec{0}, b)$. We can define the prior distribution over θ as follows

$$f_{\mathcal{L}}(\theta_i|\vec{0}, b) = \frac{1}{2b}e^{-\frac{\theta_i}{b}} \tag{48}$$

And by applying the -log we obtain

$$-\log f_{\mathcal{L}}(\theta_i|\vec{0}, b) = -\log\frac{1}{2b} + \frac{||\theta||_1}{b} \tag{49}$$

By considering the previously obtained closed form expression for the likelihood $||\vec{y} - X\theta||_2^2$, and linking it to the one we just deduced (ignoring the constant term $-\log \frac{1}{2b}$), we get the following $J(\theta)$ cost function

$$J(\theta) = ||X\theta - \vec{y}||_2^2 + \frac{1}{b}||\theta||_1 \tag{50}$$

Which shows how choosing a Laplace distribution over θ as a prior will lead to a L_1 norm regularization, where the hyperparameter γ is

$$\gamma = \frac{1}{b} \tag{51}$$

Remark: Linear regression with L_2 regularization is also commonly called *Ridge regression*, and when L_1 regularization is employed, is commonly called *Lasso regression*. These regularizations can be applied to any Generalized Linear models just as above (by replacing $\log p(y|x, \theta)$ with the appropriate family likelihood). Regularization techniques of the above type are also called *weight decay*, and *shrinkage*. The Gaussian and Laplace priors encourage the parameter values to be closer to their mean (*i.e.,* zero), which results in the shrinkage effect.

Remark: Lasso regression (*i.e.,* L_1 regularization) is known to result in sparse parameters, where most of the parameter values are zero, with only some of them non-zero.