CS 229, Summer 2022
Problem Set #2 Solutions

Jason Alexander Chan (`jchan7`)

Due Monday, July 25 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/23539/discussion/1600716`. (3) This quarter, Summer 2022, all homework assignments must be submitted individually. 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) To account for late days, the due date is Monday, July 25 at 11:59 pm. If you submit after Monday, July 25 at 11:59 pm, you will begin consuming your late days. If you wish to submit on time, submit before Monday, July 25 at 11:59 pm.

All students must submit an electronic PDF version of the written questions. 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.

1. **[15 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.

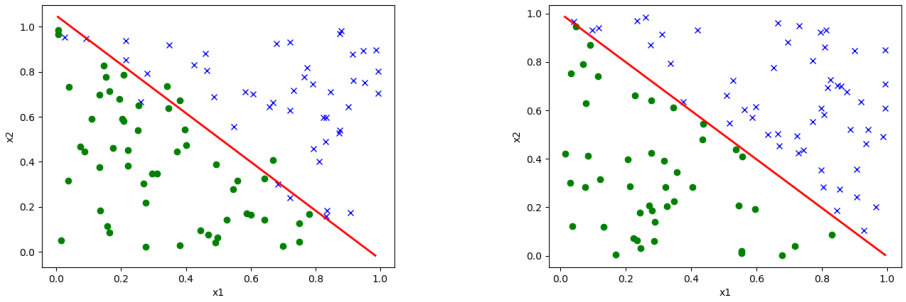
(a) [2 points] What is the most notable difference in training the logistic regression model on datasets A and B?

Answer: The model is stable when trained on ds1_a.csv but unstable when trained on ds1_b.csv. When trained on ds1_a.csv the parameter θ achieves the early stop condition in 30372 iterations but when trained on ds1_b.csv θ the model training doesn't stop even after 1 million iterations (the training error is never less than 1e-15).

(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: The model behaves unexpectedly on dataset B and not A because B has a dataset that is perfectly separable. Logistic regression assumes the dataset has non-perfect separation across a decision boundary. When perfect separation exists, the parameters don't converge because the loss function is unbounded. Examining the figure below, for dataset B, we can see that that the datapoints of each class lies perfectly on each side of the decision boundary. This is not the case for dataset A.



(a) ds1_a.csv after approx. 30k iterations (b) ds1_b.csv after approx. 200k iterations

Figure 1: ds1_a.csv vs. ds1_b.csv decision boundaries

(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. Different learning rate: no because the loss function for perfectly separable dataset is still unbounded.

ii. Decreasing learning rate over time: no because this has no effect on the perfectly separated dataset.

iii. Linearly scaling the input features: no because the relative separation between classes would still be preserved.

iv. Adding regularization to the loss function: Yes, L2 regularization can make ds1_b.csv converge. L2 regularisation adds an additional term to the objective function to penalise the 2-norm of θ. This controls the unbounded growth of θ with each iteration for a perfectly separated dataset. When $\lambda = 0.01$, ds1_b.csv converged in 5539 iterations.

v. Adding zero-mean Gaussian noise to training data or labels: Yes because this would disrupt the perfectly separated data.

(d) [3 points] Are support vector machines vulnerable to datasets like B? Why or why not? Give an informal justification.

Answer: SVMs are not vulnerable to datasets like B because SVMs find the optimal geometric margin for a given dataset. Specifically, a hard margin SVM will have a solution only if the dataset is linearly separable, which will be the case for a perfectly separated dataset.

2. **[22 points] Spam classification**

In this problem, we will use the naive Bayes algorithm and an SVM 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` [1]

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 vocabular size after the pre-processing step. You do not need to include any other output for this subquestion.

Answer: The vocab dictionary size is 1721.

(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: My naive bayes implementation achieved a 97.85% accuracy on the test data.

(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:

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

[1] 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.

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 spam tokens are claim, won, prize, tone, urgent! from highest to lowest.

(d) [2 points] Support vector machines (SVMs) are an alternative machine learning model that we discussed in class. We have provided you an SVM implementation (using a radial basis function (RBF) kernel) within `src/spam/svm.py` (You should not need to modify that code).

One important part of training an SVM parameterized by an RBF kernel (a.k.a Gaussian kernel) is choosing an appropriate kernel radius parameter.

Complete the `compute_best_svm_radius` by writing code to compute the best SVM radius which maximizes accuracy on the validation dataset. Report the best kernel radius you obtained in the writeup.

Answer: The best kernel radius is 0.1 which was found by training and then predicting against the validation dataset to maximise accuracy.

3. **[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 $\phi : \mathbb{R}^d \to \mathbb{R}^p$ 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: The goal of mercer's theorem is to show that the Kernel is Positive Semi-Definite (PSD). We want to show kernel operations preserve PSD. We are given that K_1 and K_2 are both in $R^{d\times d}$ so let's define $u \in R^d$. We wish to show that $u^T K(x, z)u$ is a PSD in all the cases below.

(a) Yes $K(x, z)$ is a kernel. Summation of two PSDs results in a PSD.

$$u^T(K_1 + K_2)u = u^T K_1 u + u^T K_2 u \geq 0 \tag{1}$$

(b) No, $K(x, z)$ is not a kernel. Subtraction of two PSDs doesn't guarantee a PSD. Suppose $K2 = 100K_1$.

$$u^T(K_1 - K_2)u = u^T K_1 u - 100u^T K_1 u < 0 \tag{2}$$

(c) Yes $K(x, z)$ is a kernel. Scalar multiple (if positive real number) of a PSD results in a PSD. We are given that $a \in \mathbb{R}^+$.

$$au^T K_1 u = a(u^T K_1 u) \geq 0 \tag{3}$$

(d) No, $K(x,z)$ is not a kernel. Suppose a = 50

$$-au^T K_1 u = -50(u^T K_1 u) \leq 0 \tag{4}$$

(e) Yes $K(x,z)$ is a kernel.

$$u^T K(x,z) u = u^T K_1 K_2 u \tag{5}$$

$$= u^T K_1 u u^{-1} (u^T)^{-1} u^T K_2 u \tag{6}$$

$$= (u^T K_1 u) u^{-1} (u^T)^{-1} (u^T K_2 u) \tag{7}$$

$$= (u^T K_1 u)(u^T u)^{-1} (u^T K_2 u) \tag{8}$$

Since $u^T u$ is an inner product of the same vector it is always greater or equal to zero. Inner products are scalars so the inverse of a positive scalar is also positive. Thus

$$(u^T u)^{-1} \geq 0 \tag{9}$$

We already know that a $u^T K_1 u \geq 0$ and $u^T K_2 u \geq 0$ since K_1 and K_2 are kernels and therefore are PSDs, thus $K(x,z)$ is a kernel because:

$$u^T K(x,z) u = (u^T K_1 u)(u^T u)^{-1} (u^T K_2 u) \geq 0 \tag{10}$$

(f) Yes, $K(x,z)$ is a kernel. We can consider $f(x)$ as a feature map to one dimension such that $\phi(x) = f(x)$ and $\phi(z) = f(z)$. Thus, $K(x,z)$ is a kernel because it satisfies the definition of a kernel as an inner product of a feature map.

$$K(x,z) = \phi(x)^T \phi(z) = f(x) f(z) \tag{11}$$

(g) Yes $K(x,z)$ is a kernel. K_3 is a kernel in $\mathbb{R}^{p \times p}$, which is defined as a function of the feature vectors of $\phi(x)$ and $\phi(z)$. $K(x,z)$ must also be a kernel since ϕ can be any arbitrary feature transformation function.

$$K_3(\phi(x), \phi(z)) = < \phi_3(\phi(x)), \phi_3(\phi(z)) > \tag{12}$$

(h) Yes $K(x,z)$ is a kernel. $p(x)$ is a polynomial over x with positive coefficients. Let's describe p(x) with arbitrary polynomial order k.

$$K(x,z) = p(K_1(x,z)) = a_0 + a_1 K_1(x,z) + ... + a_k K_1(x,z)^k \tag{13}$$

Since all coefficients are positive, and we are given that K_1 is a valid kernel and hence a PSD, we also know a PSD to any positive exponent is still a PSD, thus $K(x,z)$ is a kernel.

4. **[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. We implicitly represent the high-dimensional parameter vector as the below, where h is the index from 1 to i.

$$\theta^{(i)} = \sum_{h=1}^{i} \beta_h \phi(x^{(h)}) \tag{14}$$

We initialise the high-dimensional parameter vector $\theta^{(0)}$ as zeros. Where β is thus a vector of n number of zeros, where n corresponds to the number of training examples.

$$\theta^{(0)} = \beta\phi(x^{(0)}) \tag{15}$$

ii. We can efficiently make a prediction on a new input $x^{(i+1)}$ by introducing inner products to the perceptron algorithm, which we can then substitute for a kernel.

$$y^{(i+1)}_{predicted} = sign(g({\theta^{(i)}}^T \phi(x^{i+1})) \tag{16}$$

Substitute 14 in the above.

$$y^{(i+1)}_{predicted} = sign(\sum_{h=1}^{i} \beta_h \phi(x^{(h)})\phi(x^{(i+1)})) \tag{17}$$

Substitute a kernel rather than explicitly calculate the dot product in feature space

$$y^{(i+1)}_{predicted} = sign(\sum_{h=1}^{i} \beta_h K(x^{(h)}, x^{(i+1)})) \tag{18}$$

iii. We can modify the update rule to update on a new training example $(x^{(i+1)}, y^{(i+1)})$ from the below

$$\theta^{(i+1)} := \theta^{(i)} + \alpha(y^{(i+1)} - h_{\theta^{(i)}}(x^{(i+1)}))\phi(x^{(i+1)}) \tag{19}$$

To only needing to update β_i, which can then be used to update on new training examples as described in 18.

$$\beta_i = \alpha(y^{(i)} - y^{(i)}_{predicted}) \tag{20}$$

(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?

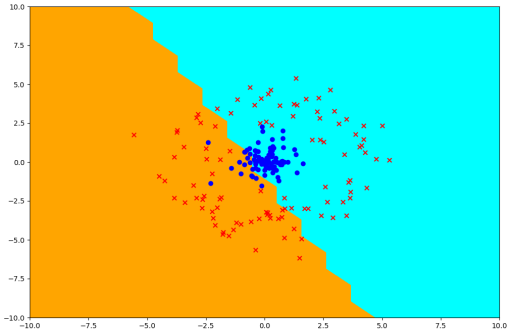


Figure 2: Dot product

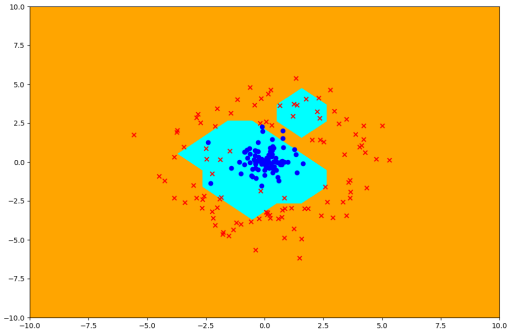


Figure 3: Radial Basis Function

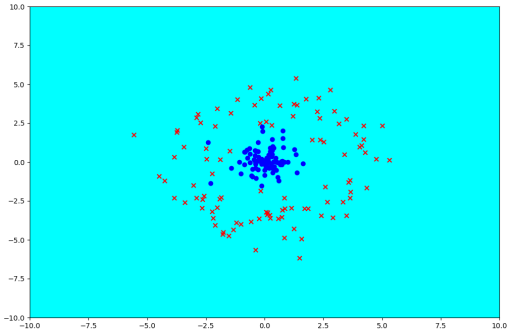


Figure 4: Invalid kernel

Answer:

- The dot-product kernel 'works a bit' but is still poor. It can't perform well on non-linearly separable data as in this case. Hence we see the apparent straight line between the two classes.
- Radial Basis Function (RBF) performed the best and 'works well'. The RBF is a kernel that represents a feature space of infinite dimensions. RBFs are thus very 'expressive' when attempting to classify highly non-linear datasets. With some parameter tuning RBF could work even better for this case.
- Unsurprisingly, invalid kernel 'completely fails' because an invalid kernel 'breaks' the perceptron algorithm. The kernel trick is a substitution technique which still relies upon the perceptron algorithm. Applying an invalid kernel results in a failure of the algorithm altogether.

5. **[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 example 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: For a single example

$$CE(y^{(i)}, \hat{y}^{(i)}) = -\sum_{k=1}^{K} y_k^{(i)} log(\hat{y}_k^{(i)}) \tag{24}$$

But since this is a softmax function we know that

$$\sum_{k=1}^{K} y_k^{(i)} = 1 \tag{25}$$

So the cross entropy loss becomes the below when we consider the class label l

$$CE(y^{(i)}, \hat{y}^{(i)}) = -log(\hat{y}^{(i)}) \tag{26}$$

$$CE(y^{(i)}, \hat{y}^{(i)}) = -log(\frac{exp(z_l^{(i)})}{\sum_{k=1}^{K} exp(z_k^{(i)})}) \tag{27}$$

$$CE(y^{(i)}, \hat{y}^{(i)}) = log(\frac{\sum_{k=1}^{K} exp(z_k^{(i)})}{exp(z_l^{(i)})}) \tag{28}$$

$$CE(y^{(i)}, \hat{y}^{(i)}) = log(\sum_{k=1}^{K} exp(z_k^{(i)})) - z_l^{(i)} \tag{29}$$

For the predicted label $\hat{y}$, the differentiation of $\sum_{k=1}^{K} exp(z_k^{(i)})$ with respect to $z^{(i)}$ for any element j in $1, ...K$ has two cases, when $j = l$ and when $j \neq l$

$$\frac{\partial}{\partial z_j^{(i)}} \sum_{k=1}^{K} exp(z_k^{(i)}) = \begin{cases} z_l^{(i)} & \text{if } j = l \\ z_j^{(i)} & otherwise \end{cases} \tag{30}$$

Thus

$$\frac{\partial}{\partial z_j^{(i)}} CE(y^{(i)}, \hat{y}^{(i)}) = \begin{cases} \frac{z_l^{(i)}}{\sum_{k=1}^{K} exp(z_k^{(i)})} - 1 & \text{if } j = l \\ \frac{z_j^{(i)}}{\sum_{k=1}^{K} exp(z_k^{(i)})} & otherwise \end{cases} \tag{31}$$

$$\frac{\partial}{\partial z_j^{(i)}} CE(y^{(i)}, \hat{y}^{(i)}) = \begin{cases} \hat{y_l}^{(i)} - 1 & \text{if } j = l \\ \hat{y_j}^{(i)} & otherwise \end{cases} \tag{32}$$

But we also know that $y_l^{(i)} = 1$ and $y_j^{(i)} = 1$ the 'truth' labels.

$$\frac{\partial}{\partial z_j^{(i)}} CE(y^{(i)}, \hat{y}^{(i)}) = \begin{cases} \hat{y_l}^{(i)} - y_l^{(i)} & \text{if } j = l \\ \hat{y_j}^{(i)} - y_j^{(i)} & otherwise \end{cases} \tag{33}$$

So we can conclude that

$$\nabla_{z^{(i)}} CE(y^{(i)}, \hat{y}^{(i)}) = \hat{y}^{(i)} - y^{(i)} \tag{34}$$

(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.

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:

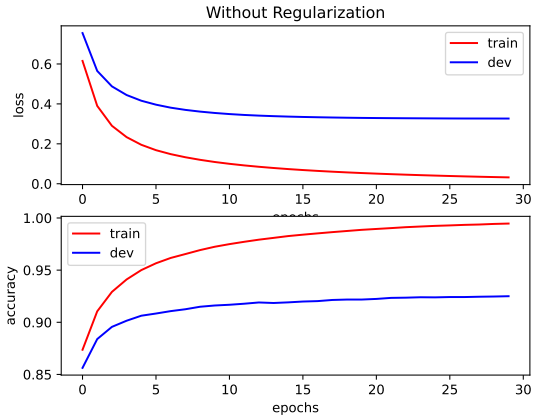


Figure 5: Baseline

(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: Both models achieved approximately the same accuracy for their training sets. However, the regularized model has higher dev accuracy vs. the baseline model. It's quite a stark improvement for adding a simple term to the loss function.

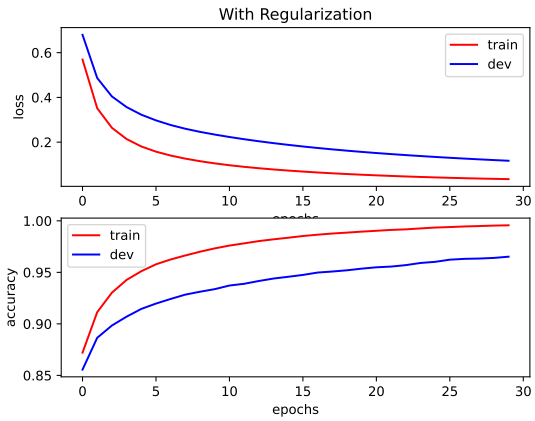


Figure 6: Regularized

In the baseline model, the loss vs. epoch plot has a large difference between its training vs. dev sets. The regularized model has a noticeably smaller difference between its loss vs. epoch plots for its training vs. dev sets.

(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: Test accuracy for regularized is 0.9676 vs. baseline which is 0.9287. This outcome makes sense because regularization reduces overfitting during training, which makes trained models more generalizable and lowers variance.

6. **[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:

$$\theta_{MAP} = \arg\max_{\theta} p(\theta|x, y) \tag{35}$$

Rearrange the conditional probability of three variables

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

We are told to assume that $p(\theta) = p(\theta|x)$ which is valid for inputs of x not modelled by θ

$$\theta_{MAP} = \arg\max_{\theta} \frac{p(y|\theta, x)p(\theta)}{p(x, y)} \tag{37}$$

We are finding the $\arg\max_\theta$ but the denominator is not a function of θ. Since it no effect on the final choice of θ, the denominator can be ignored.

$$\theta_{MAP} = \arg\max_\theta p(y|\theta, x)p(\theta) \tag{38}$$

(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,

$$\theta_{\text{MAP}} = \arg\min_\theta -\log p(y|x, \theta) + \lambda||\theta||_2^2. \tag{39}$$

Also, what is the value of λ?

Answer:

$$\theta_{MAP} = \arg\max_\theta p(y|\theta, x)p(\theta) \tag{40}$$

Apply negative log likelihood to our MAP equation.

$$= \arg\min_\theta -log(p(y|\theta, x)p(\theta)) \tag{41}$$

$$= \arg\min_\theta -logp(y|\theta, x) - log(p(\theta)) \tag{42}$$

Our class prior $p(\theta)$ is a Gaussian distribution $\theta \sim \mathcal{N}(0, \eta^2 I)$. Recall formula for multivariate Gaussian is

$$p(\theta; \mu, \Sigma) = \frac{1}{(2\pi)^{\frac{n}{2}}|\Sigma|^{\frac{1}{2}}} exp(-\frac{1}{2}(\theta - \mu)^T \Sigma^{-1}(\theta - \mu)) \tag{43}$$

Substitute our values for the mean and covariance matrix and simplify

$$= \frac{1}{(2\pi)^{\frac{n}{2}}|\eta^2 I|^{\frac{1}{2}}} exp(-\frac{1}{2}(\theta)^T (\eta^2 I)^{-1}(\theta)) \tag{44}$$

$$= \frac{1}{(2\pi)^{\frac{n}{2}}|\eta^2 I|^{\frac{1}{2}}} exp(-\frac{1}{2\eta^2}(\theta)^T I(\theta)) \tag{45}$$

$$= \frac{1}{(2\pi)^{\frac{n}{2}}|\eta^2 I|^{\frac{1}{2}}} exp(-\frac{1}{2\eta^2}\|\theta\|_2^2) \tag{46}$$

Substitute the above into our negative log likelihood MAP argmin equation:

$$\theta_{MAP} = \arg\min_\theta -logp(y|\theta, x) - log(\frac{1}{(2\pi)^{\frac{n}{2}}|\eta^2 I|^{\frac{1}{2}}} exp(-\frac{1}{2\eta^2}\|\theta\|_2^2) \tag{47}$$

$$= \arg\min_\theta -logp(y|\theta, x) - log(\frac{1}{(2\pi)^{\frac{n}{2}}|\eta^2 I|^{\frac{1}{2}}}) - log(exp(-\frac{1}{2\eta^2}\|\theta\|_2^2)) \tag{48}$$

Finally, we can drop the constant term since it is not a function of θ and has no effect on the minimisation

$$= \arg\min_\theta -logp(y|\theta, x) + \frac{1}{2\eta^2}\|\theta\|_2^2 \tag{49}$$

Where

$$\lambda = \frac{1}{2\eta^2} \tag{50}$$

(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 are given $y = \theta^T x + \epsilon$ and need to evaluate $P(y|\theta, x)$. We are given that θ and ϵ are Gaussian so the distribution of y and its conditional distributions are also Gaussian because we assumed independent random variables .

$$P(\vec{y}|x, \theta) = \mathcal{N}(\mu_{y|x,\theta}, \sigma_{y|x,\theta}) \tag{51}$$

$$P(\vec{y}|x, \theta) = \mathcal{N}(X\theta, \sigma^2 I) \tag{52}$$

We substitute the above into the equation we derived from the last question

$$\theta_{MAP} = \arg\min_\theta -log\Big[\frac{1}{(2\pi)^{\frac{n}{2}}|\sigma^2 I|^{\frac{1}{2}}} exp(-\frac{1}{2}(\vec{y} - X\theta)^T(\sigma^2 I)^{-1}(\vec{y} - X\theta))\Big] + \frac{1}{2\eta^2}\|\theta\|_2^2 \tag{53}$$

Using the same technique in the last question we can simplify as

$$\theta_{MAP} = \arg\min_\theta = \arg\min_\theta \frac{1}{2\sigma^2}\|\vec{y} - X\theta\|_2^2 + \frac{1}{2\eta^2}\|\theta\|_2^2 \tag{54}$$

$$= \arg\min_\theta \|\vec{y} - X\theta\|_2^2 + \frac{\sigma^2}{\eta^2}\|\theta\|_2^2 \tag{55}$$

To find the closed form solution for θ_{MAP} we take the gradient with respect to θ, set it to zero and solve for θ.

$$\nabla_\theta \|\vec{y} - X\theta\|_2^2 + \nabla_\theta \frac{\sigma^2}{\eta^2}\|\theta\|_2^2 = 0 \tag{56}$$

$$= -2X^T(\vec{y} - X\theta) + 2\frac{\sigma^2}{\eta^2}\theta \tag{57}$$

$$= -2X^T\vec{y} + 2X^T X\theta + 2\frac{\sigma^2}{\eta^2}\theta \tag{58}$$

$$= -X^T\vec{y} + (X^T X + \frac{\sigma^2}{\eta^2})\theta \tag{59}$$

Solving for θ, we finally find θ_{MAP}

$$\theta = (X^T X + \frac{\sigma^2}{\eta^2})^{-1} X^T \vec{y} \tag{60}$$

(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:

$$\theta_{MAP} = \arg\min_{\theta} -logp(y|\theta, x) - log(p(\theta)) \tag{61}$$

We can reuse our answer for $p(y|\theta, x)$ because $y = X\theta + \epsilon$ is still the same, and the means and variances for the gaussian distributions of θ and ϵ are the same. The prior, however, is now a laplace function parameterisd by b thus

$$\theta_{MAP} = \arg\min_{\theta} -\frac{1}{2\sigma^2}\|\vec{y} - X\theta\|_2^2 - log(\frac{1}{2b}exp(-\frac{|\theta|}{b}) \tag{62}$$

$$\theta_{MAP} = \arg\min_{\theta} \frac{1}{2\sigma^2}\|\vec{y} - X\theta\|_2^2 + \frac{|\theta|}{b} \tag{63}$$

$$\theta_{MAP} = \arg\min_{\theta} \|\vec{y} - X\theta\|_2^2 + \frac{2\sigma^2}{b}|\theta| \tag{64}$$

Thus

$$\gamma = \frac{2\sigma^2}{b} \tag{65}$$

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.