CS 229, Summer 2019
Problem Set #2 Solutions

ANDREW KIM (006405830)

Due Monday, July 29 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 Piazza forum, at `http://piazza.com/stanford/summer2019/cs229`. (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) To account for late days, the due date is Monday, July 29 at 11:59 pm. If you submit after Monday, July 29 at 11:59 pm, you will begin consuming your late days. If you wish to submit on time, submit before Monday, July 29 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 most notable difference is that dataset A converged in 30374 iterations while dataset B did not converge even after 1 million iterations. It seems that dataset B does not converge at all.

(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: I plotted the two plots for dataset A and dataset B. For the two plots, I used the converged theta value for dataset A, whereas I used the 100000th iterated value of theta = [-121.21097234 121.48709579 121.24190606] for dataset B (as theta does not converge for B).

```
==== Training model on data set B ====
Finished 10000 iterations
[-52.74109217  52.92982273  52.69691453]
Finished 20000 iterations
[-68.10040977  68.26496086  68.09888223]
Finished 30000 iterations
[-79.01759142  79.17745526  79.03755803]
Finished 40000 iterations
[-87.70771189  87.87276307  87.73897393]
Finished 50000 iterations
[-95.01838735  95.1948202   95.0551918 ]
Finished 60000 iterations
[-101.37921493  101.57119731  101.41805781]
Finished 70000 iterations
[-107.04156569  107.25200975  107.08020705]
Finished 80000 iterations
[-112.16638881  112.39737225  112.20335022]
Finished 90000 iterations
[-116.86340448  117.11642203  116.89769046]
Finished 100000 iterations
[-121.21097234  121.48709579  121.24190606]
>>>
```

Figure 1: 100,000th iteration

```
def logistic_regression(X, Y):
    """Train a logistic regression model."""
    theta = np.zeros(X.shape[1])
    learning_rate = 0.1

    i = 0
    while True:
        i += 1
        prev_theta = theta
        grad = calc_grad(X, Y, theta)
        theta = theta + learning_rate * grad
        if i % 10000 == 0:
            print('Finished %d iterations' % i)
            print(theta)

        if i>100000:
            break

        if np.linalg.norm(prev_theta - theta) < 1e-15:
            print('Converged in %d iterations' % i)
            break

    return theta


def main():
    print('==== Training model on data set A ====')
    Xa, Ya = util.load_csv('ds1_a.csv', add_intercept=True)
    theta = logistic_regression(Xa, Ya)
    util.plot(Xa, Ya, theta, "dsA")

    print('\n==== Training model on data set B ====')
    Xb, Yb = util.load_csv('ds1_b.csv', add_intercept=True)
    logistic_regression(Xb, Yb)
    thetaB = [-121.21097234, 121.48709579, 121.24190606]
    util.plot(Xb, Yb, thetaB, "dsB")
```

Figure 2: Code for plotting

Below are the two plots.

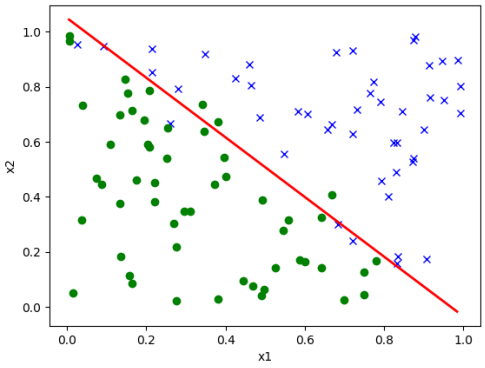

Figure 3: Dataset A

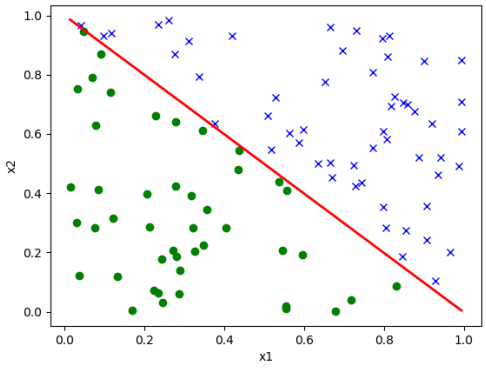

Figure 4: Dataset B

It can be seen that the data points in dataset B are perfectly and linearly separated while data points in dataset A are not perfectly separated. We can hypothesize that theta for dataset B does not converge because the data points are perfectly separated which means that the loss is always 0 (no error for logistic regression) which means that theta keeps increasing indefinitely.

To further justify this mathematically, assume that the limit goes to 0 for all data points in A and the limit goes to 1 for all data points in class B.

$$\begin{aligned} \textit{let } g(\theta, x) &= \frac{1}{1+e^{-\theta x}} \\ \lim_{\theta \to \infty} g(\theta, x) &= 0 \;\; \textit{or} \;\; 1 \end{aligned} \tag{1}$$

We know the log-likelihood function is:

$$L(\theta) = \sum_{i=1}^{n} y_i \log(g(\theta, x_i)) + (1 - y_i) \log(1 - g(\theta, x_i)) \tag{2}$$

For Class A ($y_i = 0$):

$$\begin{aligned} \lim_{\theta \to \infty} g(\theta, x_i) &= 0 \\ \lim_{\theta \to \infty} L(\theta) &= 0 + (1-0) \sum_{i=1}^{n} \log(1-0) = 0 \end{aligned} \tag{3}$$

For Class B ($y_i = 1$):

$$\begin{aligned} \lim_{\theta \to \infty} g(\theta, x_i) &= 1 \\ \lim_{\theta \to \infty} L(\theta) &= \sum_{i=1}^{n} \log(1) + 0 = 0 \end{aligned} \tag{4}$$

This shows that for y = 0 or 1, $L(\theta)$ goes to 0.
Therefore, likelihood $\ell(\theta) = exp\{L(\theta)\} = 1$ as $\theta \to \infty$

For perfectly and linearly separated dataset, the maximum likelihood is 1, therefore theta increases indefinitely and does not converge. This does not apply to dataset A, because A is not perfectly separated.

(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, because θ still goes to infinity to find maximum likelihood regardless of any learning rate value.
ii. No, because theta still goes to infinity regardless of any learning rate from i, so scaling the learning rate does not have any effect.
iii. No, because scaling the input features still makes the dataset perfectly separable which means that it does not change the maximum likelihood of $\ell(\theta) = 1$ when theta goes to infinity.
iv. Yes, because the regularization term penalizes θ of large value. Therefore, it would prevent θ from reaching infinity.
v. Yes, because adding noise to the training data or labels can make the data to be no longer perfectly separable which means that it will converge.

(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 do not try to maximize the likelihood and instead tries to maximize the geometric margin (which is not affected by theta) γ w.r.t. w, b:

$$\gamma = min_{i=1,\ldots,m} y^{(i)} \left(\left(\frac{w}{||w||} \right)^\intercal x^{(i)} + \frac{b}{||w||} \right) \tag{5}$$

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 Jos Mara Gmez 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:

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 (refer to class notes on Naive Bayes for details on Laplace smoothing in Section 2.3 of notes2.pdf).

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

[1] Almeida, T.A., Gmez 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.

(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:

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

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 words are: ['claim', 'won', 'prize', 'tone', 'urgent!']

(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 optimal SVM radius was 0.1
The SVM model had an accuracy of 0.9695340501792115 on the testing set

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.

Now here comes the question: 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:

let $K_{ij} = K(x^{(i)}, x^{(j)})$
$z^\intercal K z = \sum_{i=1}^{d} \sum_{j=1}^{d} z_i K_{ij} z_j$

(a) K is a kernel because:

$$\begin{aligned} z^\intercal K z &= \sum_{i=1}^{d}\sum_{j=1}^{d} z_i K_{ij} z_j \\ &= \sum_{i=1}^{d}\sum_{j=1}^{d} z_i (K_1(x^{(i)}, x^{(j)}) + K_2(x^{(i)}, x^{(j)})) z_j \\ &= \sum_{i=1}^{d}\sum_{j=1}^{d} z_i K_1(x^{(i)}, x^{(j)}) z_j + \sum_{i=1}^{d}\sum_{j=1}^{d} z_i K_2(x^{(i)}, x^{(j)}) z_j \\ &= z^\intercal K_1 z + z^\intercal K_2 z \end{aligned} \tag{6}$$

Since $z^\intercal K_1 z \geq 0,\ z^\intercal K_2 z \geq 0,\ z^\intercal K z \geq 0$, K is PSD
and since K_1, K_2 are symmetric, therefore $K = K_1 + K_2$ is also symmetric.

Therefore, K is a kernel.

(b) K is not necessarily a kernel. Counterexample:

$$\begin{aligned} z^\intercal K z &= \sum_{i=1}^{d}\sum_{j=1}^{d} z_i K_{ij} z_j \\ &= \sum_{i=1}^{d}\sum_{j=1}^{d} z_i (K_1(x^{(i)}, x^{(j)}) - K_2(x^{(i)}, x^{(j)})) z_j \\ &= \sum_{i=1}^{d}\sum_{j=1}^{d} z_i K_1(x^{(i)}, x^{(j)}) z_j - \sum_{i=1}^{d}\sum_{j=1}^{d} z_i K_2(x^{(i)}, x^{(j)}) z_j \\ &= z^\intercal K_1 z - z^\intercal K_2 z \end{aligned} \tag{7}$$

If $z^\intercal K_1 z < z^\intercal K_2 z$, then $z^\intercal K_1 z - z^\intercal K_2 z < 0$
which means that K is not PSD
Therefore, K is not necessarily a kernel.

(c) K is a kernel because:

$$\begin{aligned} z^\intercal K z &= \sum_{i=1}^{d}\sum_{j=1}^{d} z_i K_{ij} z_j \\ &= \sum_{i=1}^{d}\sum_{j=1}^{d} z_i a K_1(x^{(i)}, x^{(j)}) z_j \\ &= a \cdot \sum_{i=1}^{d}\sum_{j=1}^{d} z_i K_1(x^{(i)}, x^{(j)}) z_j \\ &= a \cdot z^\intercal K_1 z \end{aligned} \tag{8}$$

Since $z^\intercal K_1 z \geq 0$ and $a \in \mathbb{R}^+,\ a \cdot z^\intercal K z \geq 0$, K is PSD
and since K_1 is symmetric, $K = a \cdot K_1$ is also symmetric.
Therefore, K is a kernel.

(d) K is not necessarily a kernel. Counterexample:

If a is negative, from (c):
Since $z^\intercal K_1 z \geq 0$ and $a \in \mathbb{R}^-,\ a \cdot z^\intercal K z \leq 0$, K is not PSD.
Therefore, K is not necessarily a kernel.

(e) K is a kernel because:
Let $K_1(x,z) = \phi_1(x)^\intercal \phi_1(z) = \sum_i \phi_1^{(i)}(x)\phi_1^{(i)}(z)$
Let $K_2(x,z) = \phi_2(x)^\intercal \phi_2(z) = \sum_j \phi_2^{(i)}(x)\phi_2^{(i)}(z)$

$$\begin{aligned} K(x,z) &= K_1(x,z)K_2(x,z) \\ &= \phi_1(x)^\intercal \phi_1(z) \cdot \phi_2(x)^\intercal \phi_2(z) \\ &= \sum_i \phi_1^{(i)}(x)\phi_1^{(i)}(z) \cdot \sum_j \phi_2^{(j)}(x)\phi_2^{(j)}(z) \\ &= \sum_i \sum_j \phi_1^{(i)}(x)\phi_1^{(i)}(z)\phi_2^{(j)}(x)\phi_2^{(j)}(z) \\ &= \sum_i \sum_j (\phi_1^{(i)}(x)\phi_2^{(j)}(x)) \cdot (\phi_1^{(i)}(z)\phi_2^{(j)}(z)) \end{aligned} \tag{9}$$

If we let $\phi_3^{(i,j)}(x) = \phi_1^{(i)}(x)\phi_2^{(j)}(x)$ and $\phi_3^{(i,j)}(z) = \phi_1^{(i)}(z)\phi_2^{(j)}(z)$
Then we get:

$$\begin{aligned} K(x,z) &= \sum_i \sum_j \phi_3^{(i,j)}(x)\phi_3^{(i,j)}(z) \\ &= \phi_3(x)^\intercal \phi_3(z) \end{aligned} \tag{10}$$

This defines a kernel therefore, K is a kernel.

(f) Similar to how $K(x,z) = \phi(x)^\intercal \phi(z)$,
we can let $K(x,z) = K_{ij} = f(x)f(z) = f(x^i)f(x^j)$.

$$\begin{aligned} z^\intercal K z &= \sum_{i=1}^{d} \sum_{j=1}^{d} z_i f(x^i) f(x^j) z_j \\ &= \left(z_1 f(x^{(1)}) + z_2 f(x^{(2)}) + z_3 f(x^{(3)}) + \cdots + z_n f(x^{(n)})\right)^2 \\ &= (\sum_{i=1}^{n} z_i f(x^{(i)}))^2 \geq 0 \end{aligned} \tag{11}$$

Therefore, K is a kernel.

(g) K is a kernel because:

$$\begin{aligned} z^\intercal K z &= \sum_{i=1}^{d} \sum_{j=1}^{d} z_i K_{ij} z_j \\ &= \sum_{i=1}^{d} \sum_{j=1}^{d} z_i K_3(\phi(x^i), \phi(x^j)) z_j \geq 0 \end{aligned} \tag{12}$$

Since K_3 is a kernel, using transformed inputs as SVM still holds the fact that K_3 is a kernel. Therefore, K is a kernel.

(h) K is a kernel because:
p is a polynomial over x with positive coefficients as stated in the question
$p(x) = \sum_i a_i \cdot x^i$ where $a_i > 0$

$$\begin{aligned} z^\intercal K z &= \sum_{i=1}^{d}\sum_{j=1}^{d} z_i K_{ij} z_j \\ &= \sum_{i=1}^{d}\sum_{j=1}^{d} z_i p(K_1(x^{(i)}, x^{(j)})) z_j \\ &= \sum_{i=1}^{d}\sum_{j=1}^{d} z_i z_j \big(\sum_i a_i (K_1(x^{(i)}, x^{(j)}))^i\big) \end{aligned} \tag{13}$$

We have shown in previous questions a,c,e that a positive constant, addition of kernels and multiplications of kernels still create a kernel such that $z^\intercal K z \geq 0$
Therefore, K 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 Mercer 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. If we assume that $\theta^{(0)} = \vec{0}$, then θ is always a linear combination of $\phi(x^{(i)})$.

If we set $\theta^{(i)} = \sum_{k=1}^{i} \beta_k \phi(x^{(k)})$, then $\theta^{(i)}$ is a linear combination of β_k and $\phi(x^{(k)})$. Therefore, $\theta^{(i)}$ can be represented as a n by 1 vector $v^{(i)}$ where n is the no. of examples. Then, $v_j^{(i)}$ would be the β_j.

ii. Prediction can be made efficiently by using the given formula in the question.
Let $\theta^{(i)} = \sum_{k=1}^{i} \beta_k \phi(x^{(k)})$

$$\begin{aligned} h_{\theta^{(i)}}(x^{(i+1)}) &= g\Big(\sum_{k=1}^{i} \beta_k \phi(x^{(k)})^\intercal \phi(x^{(i+1)})\Big) \\ &= g\Big(\sum_{k=1}^{i} \beta_k K(x^{(k)}, x^{(i+1)})\Big) \end{aligned} \tag{14}$$

iii. We can update theta with the new training example by calculating using the above method in ii.

$$\alpha(y^{(i)} - g(\theta^{(i-1)\intercal}\phi(x^{(i)})))$$

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

We provide two kernels, a dot-product kernel and a radial basis function (RBF) kernel. 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 two plots (corresponding to each of the kernels) in your writeup, and indicate which plot belongs to which kernel.

Answer:

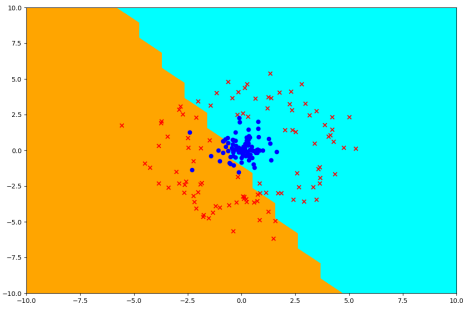

Figure 5: Dot-product kernel

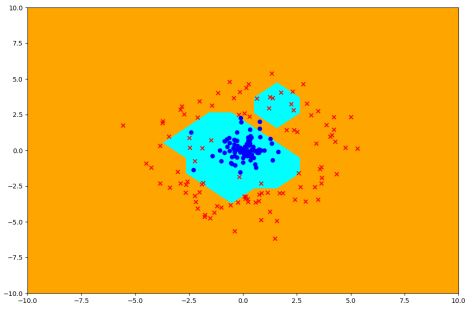

Figure 6: Radial basis function kernel

(c) [2 points]

One of the provided kernels performs extremely poorly in classifying the points. Which kernel performs badly and why does it fail?

Answer: Dot-product kernel performs extremely poorly because it performs linear operation on x, but the data is not linearly separable. (One class is clustered inside another ring of class)

5. **[25 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 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) **[15 points]**

Implement both forward-propagation and back-propagation for the above loss function. 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:

(Answer on next page)

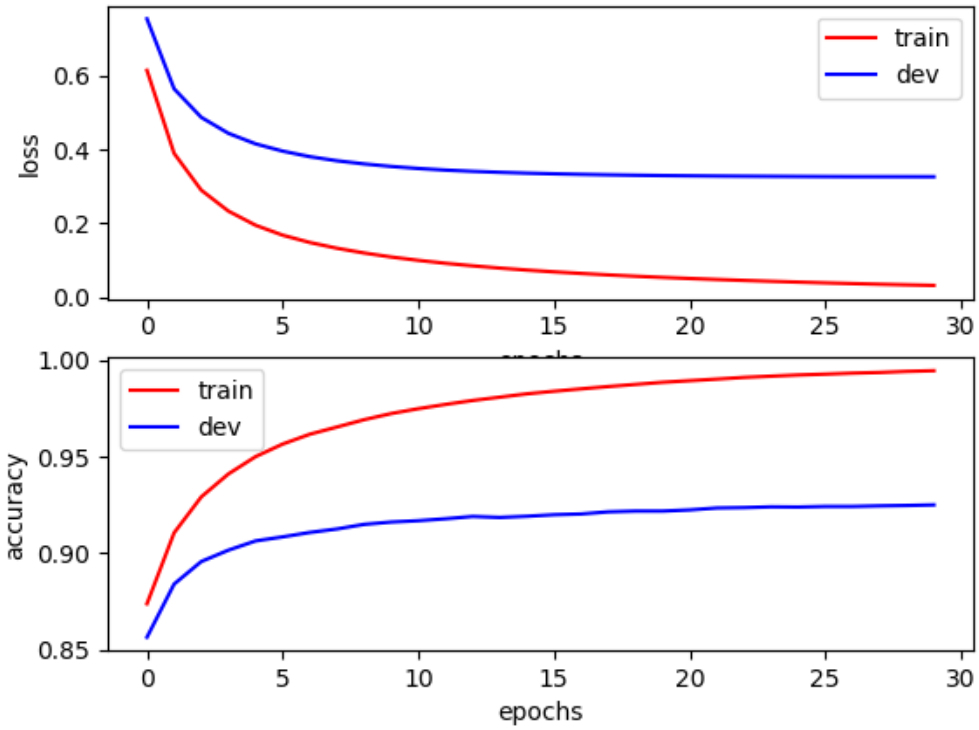

Figure 7: Baseline: loss vs epoch, accuracy vs epoch

(b) **[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:

(Answer on the next page)

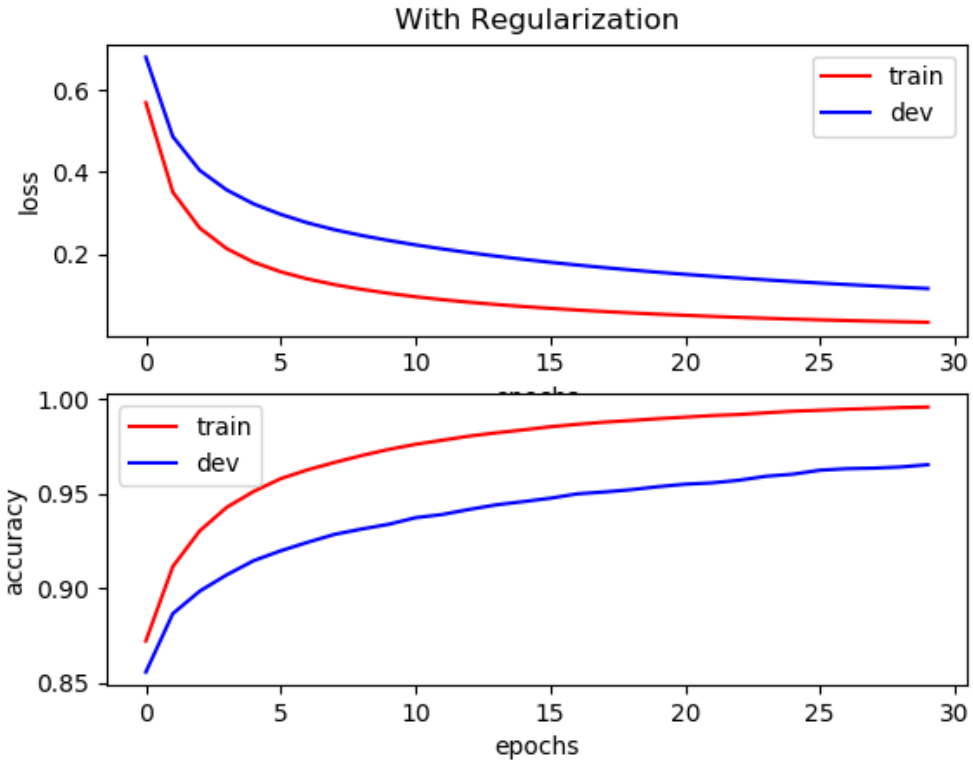

Figure 8: Regularized: loss vs epoch, accuracy vs epoch

Observations: It can be seen that the difference in both the accuracy and loss between training and validation is very big in the non-regularized plots. This could be due to high variance in the data which causes overfitting and therefore yielding much lower accuracy in the validation model. With the regularized model, it can be seen that the model performed much better as the performance of validation data is closer to the performance of training data. As regularization penalizes large values of theta, it reduces overfitting and hence explains the better accuracy from regularization.

(c) **[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.

Answer: For model baseline, got accuracy: 0.928700
For model regularized, got accuracy: 0.967600

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). To show this, we shall proceed step by step, showing intermediate steps.

(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:

From Bayes Rule,

$$p(\theta|x, y) = \frac{p(y|x, \theta)p(\theta|x)}{p(y|x)} \tag{15}$$

Assuming that $p(\theta) = p(\theta|x)$.

$$p(\theta|x, y) = \frac{p(y|x, \theta)p(\theta)}{p(y|x)} \tag{16}$$

$$arg\max_{\theta} p(\theta|x, y) = arg\max_{\theta} \frac{p(y|x, \theta)p(\theta)}{p(y|x)} \tag{17}$$

Since $p(y|x)$ is not a function of θ.

$$\therefore \theta_{MAP} = arg\max_{\theta} p(y|x,\theta)p(\theta) \tag{18}$$

(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

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

Also, what is the value of λ?

Answer:

$$\begin{aligned} \theta_{MAP} &= arg\max_{\theta} p(y|x,\theta)p(\theta) \\ &= arg\min_{\theta} -p(y|x,\theta)p(\theta) \end{aligned} \tag{19}$$

Since log is a monotonic function, taking the log does not affect the value of theta min.

$$\begin{aligned} \theta_{MAP} &= arg\min_{\theta} -\log\big[p(y|x,\theta)p(\theta)\big] \\ &= arg\min_{\theta} -\log p(y|x,\theta) - \log p(\theta) \end{aligned} \tag{20}$$

Given the normal distribution $\theta \sim N(0, \eta^2 I)$

$$\begin{aligned} p(\theta) &= \frac{1}{\sqrt{2\pi}|\eta^2 I|^{\frac{1}{2}}} exp\big(-\frac{1}{2}\theta^{\intercal}|\eta I|^{-1}\theta\big) \\ &= \frac{1}{\sqrt{2\pi}|\eta^2 I|^{\frac{1}{2}}} exp\big(-\frac{||\theta||_2^2 \cdot I}{2\eta^2}\big) \end{aligned} \tag{21}$$

$$\begin{aligned} \therefore \log p(\theta) &= \log\left(\frac{1}{\sqrt{2\pi}|\eta^2 I|^{\frac{1}{2}}} exp\big(-\frac{||\theta||_2^2 \cdot I}{2\eta^2}\big)\right) \\ &= -\log(\sqrt{2\pi}|\eta^2 I|^{\frac{1}{2}}) - \frac{||\theta||_2^2 \cdot I}{2\eta^2} \end{aligned} \tag{22}$$

$$\begin{aligned} \therefore \theta_{MAP} &= arg\min_{\theta} -\log p(y|x,\theta) + \log(\sqrt{2\pi}|\eta^2 I|^{\frac{1}{2}}) + \frac{||\theta||_2^2}{2\eta^2} I \\ &= arg\min_{\theta} -\log p(y|x,\theta) + \frac{||\theta||_2^2}{2\eta^2} I \end{aligned} \tag{23}$$

If we set $\lambda = \frac{1}{2\eta^2} I$:
$\theta_{MAP} = arg\min_{\theta} -\log p(y|x,\theta) + \lambda||\theta||_2^2$

(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: Given 3 Normal distributions

$\epsilon^{(i)} \sim N(0, \sigma^2), \quad y^{(i)} - \theta^\intercal x^{(i)} \sim N(0, \sigma^2), \quad y^{(i)} \sim N(\theta^\intercal x^{(i)}, \sigma^2), \quad \theta \sim N(0, \eta^2 I)$

Vectorizing the functions gives us:

$$\begin{aligned} p(y^{(i)}|x^{(i)}, \theta) &= \frac{1}{\sqrt{2\pi}\sigma} exp\big(-\frac{(y^{(i)} - \theta^\intercal x^{(i)})^2}{2\sigma^2}\big) \\ p(\vec{y}|X, \theta) &= \frac{1}{\sqrt{2\pi}\sigma} exp\big(-\frac{(X\theta - \vec{y})^\intercal (X\theta - \vec{y})}{2\sigma^2}\big) \end{aligned} \tag{24}$$

$$\begin{aligned} p(\theta) &= \frac{1}{\sqrt{2\pi}|\eta^2 I|^{\frac{1}{2}}} exp\big(-\frac{||\theta||_2^2 \cdot I}{2\eta^2}\big) \\ p(\theta) &= \frac{1}{\sqrt{2\pi}|\eta^2 I|^{\frac{1}{2}}} exp\big(-\frac{\theta^\intercal \theta}{2\eta^2} I\big) \end{aligned} \tag{25}$$

Using the previous equation:

$$\begin{aligned} \theta_{MAP} &= arg\max_\theta \log p(y|x, \theta) + \log p(\theta) \\ &= arg\max_\theta \log\left(\frac{1}{\sqrt{2\pi}\sigma} exp\big(-\frac{(X\theta - \vec{y})^\intercal (X\theta - \vec{y})}{2\sigma^2}\big)\right) + \log\left(\frac{1}{\sqrt{2\pi}|\eta^2 I|^{\frac{1}{2}}} exp\big(-\frac{\theta^\intercal \theta}{2\eta^2} I\big)\right) \\ &= arg\max_\theta -\frac{(X\theta - \vec{y})^\intercal (X\theta - \vec{y})}{2\sigma^2} - \frac{\theta^\intercal \theta}{2\eta^2} I \end{aligned} \tag{26}$$

Taking the derivative of w.r.t. θ and equating it to 0:

$$\begin{aligned} \frac{\partial}{\partial\theta}(-\frac{(X\theta - \vec{y})^\intercal (X\theta - \vec{y})}{2\sigma^2} - \frac{\theta^\intercal \theta}{2\eta^2} I) &= -\frac{1}{\sigma^2} X^\intercal (X\theta - \vec{y}) - \frac{\theta}{\eta^2} I = 0 \\ \therefore \theta_{MAP} &= (\frac{\sigma^2}{\eta^2} I + X^T X)^{-1} X^T \vec{y} \end{aligned} \tag{27}$$

(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:

$$\theta_{MAP} = \arg\max_{\theta} \log p(y|x,\theta)p(\theta) = \arg\min_{\theta} -\log p(y|x,\theta) - \log p(\theta) \tag{28}$$

Given $\theta_i \sim L(0,b)$,
The laplace distribution is

$$\frac{1}{2b}exp\big(-\frac{|\theta-0|}{b}\big) = \frac{1}{2b}exp\big(-\frac{||\theta||_1}{b}\big) \tag{29}$$

Since $\epsilon \sim N(0,\sigma^2)$, $y \sim N(x^\intercal\theta,\sigma^2)$:

$$p(y|x,\theta) = \frac{1}{\sqrt{2\pi}\sigma}exp\big(-\frac{1}{2\sigma^2}(y-x^\intercal\theta)^2\big) \tag{30}$$

Vector form of the equation gives us:

$$p(\vec{y}|X,\theta) = \frac{1}{\sqrt{2\pi}\sigma}exp\big(-\frac{1}{2\sigma^2}||X\theta-\vec{y}||_2^2\big) \tag{31}$$

$$\begin{aligned}\theta_{MAP} &= arg\min_{\theta} -\log\Big(\frac{1}{\sqrt{2\pi}\sigma}exp\big(-\frac{1}{2\sigma^2}||X\theta-\vec{y}||_2^2\big)\Big) - \log\Big(\frac{1}{2b}exp\big(-\frac{||\theta||_1}{b}\big)\Big)\\ &= arg\min_{\theta} \frac{1}{2\sigma^2}||X\theta-\vec{y}||_2^2 + \frac{1}{b}||\theta||_1\\ &= arg\min_{\theta} ||X\theta-\vec{y}||_2^2 + \frac{2\sigma^2}{b}||\theta||_1\end{aligned}$$

If we set $\gamma = \frac{2\sigma^2}{b}$:

$$\theta_{MAP} = arg\min_{\theta} ||X\theta-\vec{y}||_2^2 + \gamma||\theta||_1 = arg\min_{\theta} J(\theta) \tag{32}$$

Therefore, we have shown that theta map is equivalent to the loss function regularized where $\gamma = \frac{2\sigma^2}{b}$.

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.