CS229 Problem Set 2

anjay.h.friedman

August 2020

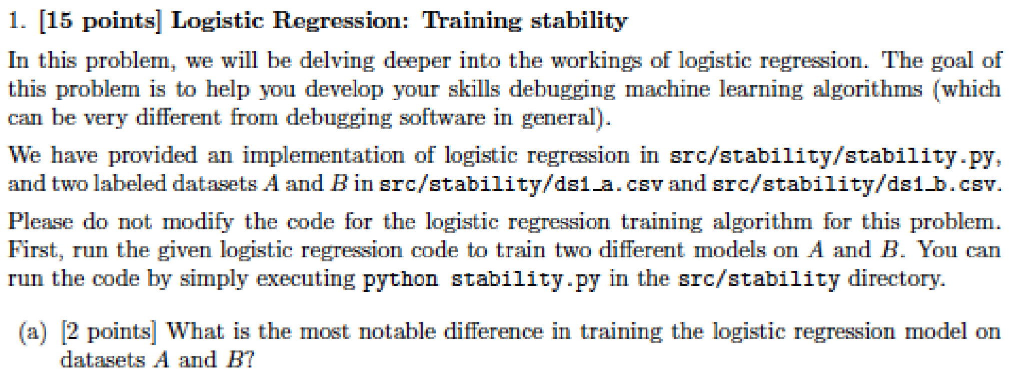
1 Introduction

This document represents my solutions to the 2019 CS229 course's <u>second</u> problem set. I appreciate that the materials for the course are available from Stanford's and other websites and hope that I am not infringing on any rights. It took me 25 hours to complete.

Code accompanying the problems can be found here

2 Problems

2.1 1. [15 points] Logistic regression: Training Stability


1. **[15 points] Logistic Regression: Training stability**

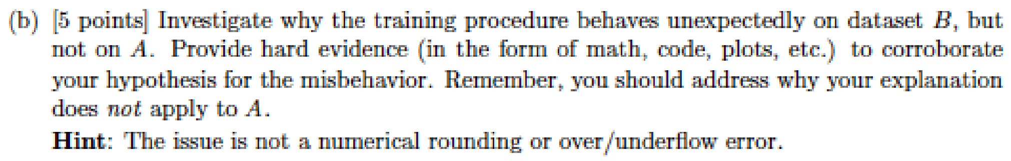
In this problem, we will be delving deeper into the workings of logistic regression. The goal of this problem is to help you develop your skills debugging machine learning algorithms (which can be very different from debugging software in general).

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.

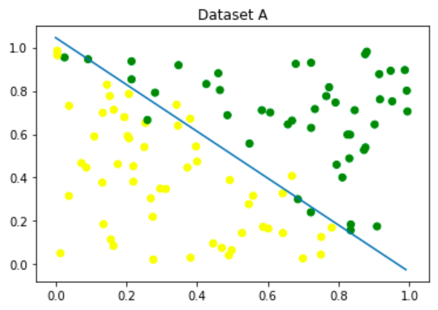
(a) [2 points] What is the most notable difference in training the logistic regression model on datasets A and B?

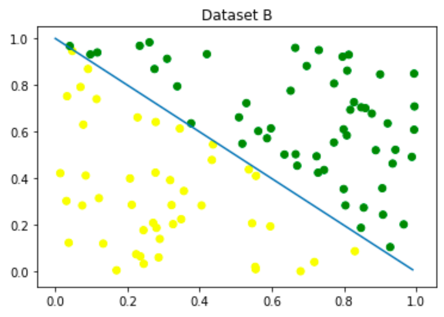
Solution
Logistic regression on data set A converges in about 30,000 iterations while it does not converge with data set B in any reasonable amount of time.


(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.

Solution





From the above plots, it can be seen that data from dataset A is not linearly separable whereas dataset B is just barely linearly separable.

Additionally, examining the norm with each iteration finds that while the norm for B does not become smaller than EPSILON = 1E-15 in a reasonable amount of time, it does become less than 0.001, demonstrating that the algorithm is working and the problem has to do with convergence.

Mathematically, the reason that dataset B does not converge using gradient ascent is that θ can be increased without affecting the correctness of the classifications due to the linear separability. Thus, there is no real cost to increasing θ. Further, as θ increases each iteration by c, the likelihood increases marginally since $h_{\theta_{i+1}} = g(c\theta_i^T x) > g(\theta_i^T x) = h_{\theta_i}$ for positively classified examples. The reverse is true for negatively classified examples, which is captured by the likelihood function as an increase in probability.

Alternatively, with datasets like A that are not linearly separable, the eventual cost to increasing θ coming from the degree of misclassification of certain examples increasing, outweights the increase in the likelihood function from correctly classifying examples by a greater margin.

(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.

Solution

1. A different constant learning rate will not change the fact that there is no regularization for the size of θ and that increasing it will increase the likelihood to no end.

2. This is unlikely to work for most ways to decrease the learning rate over time as θ can still be scaled by a large enough amount for the likelihood to not converge. Still, for a rapid enough decrease in learning rate, it is plausible that the rate of decrease in learning rate outweights the rate of increase in θ.

3. This won't change the fact that data in datasets like B are linearly separable and thus, that the likelihood function can be increased endlessly by scaling θ

4. By definition, this will work as the core reason that datasets like B do not converge is the lack of a penalty for increasing θ endlessly.

5. This could also work if it alters the data to the extent of it not being linearly separable, such that the data is more like dataset A.

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

Solution
Because of the $||W|| = 1$ constraint, the parameters can't endlessly increase to increase the likelihood. Thus, the parameters that maximize the margin should converge within a reasonable amount of time.

2.2 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.

(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.]

Size of dictionary: 1721
Test set accuracy: 0.9785

(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.

The top 5 indicative 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.

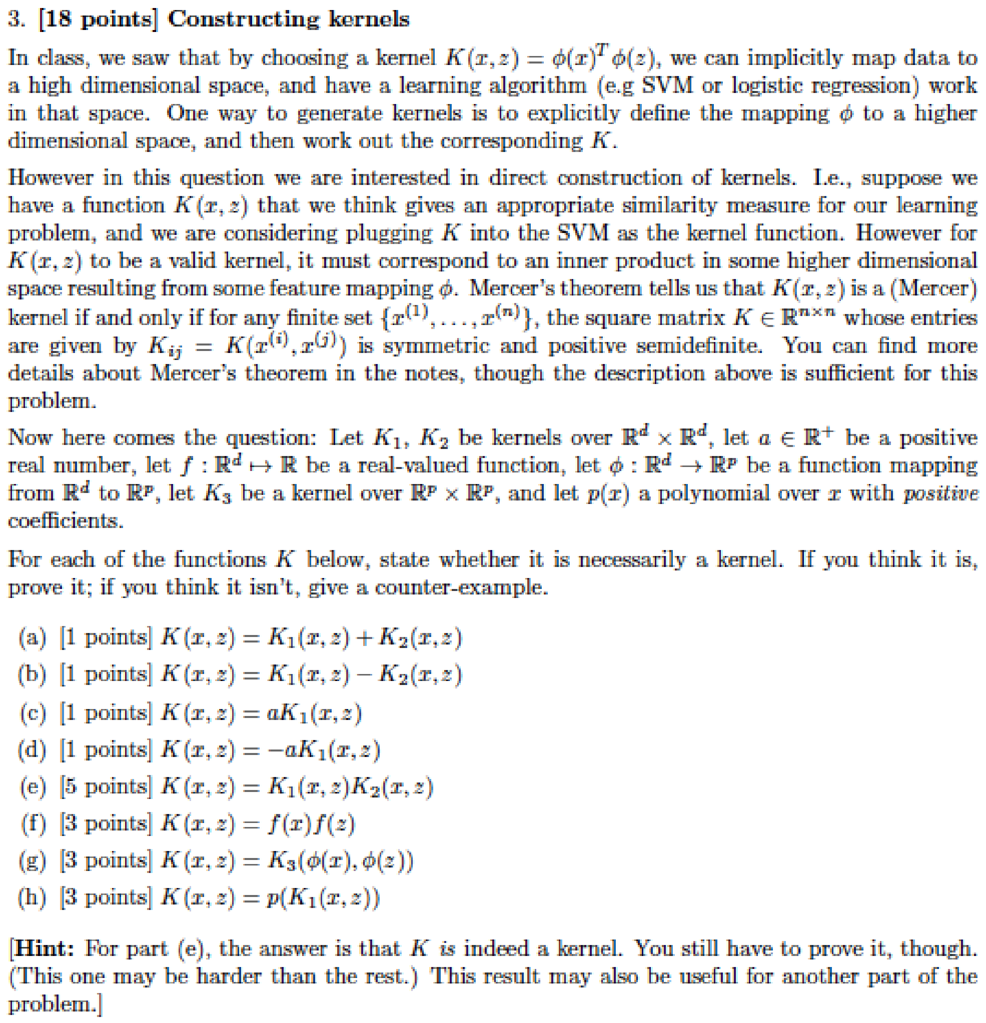
Complete the `compute_best_svm_radius` by writing code to compute the best SVM radius which maximizes accuracy on the validation dataset. Report the best kernel radius you obtained in the writeup.

Solution

The optimal SVM radius was 10

The SVM model had an accuracy of 0.8799 on the testing set

2.3 3. [18 points] Constructing Kernels



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.]

Note that if $K(x,y)$ is a valid kernel, it's Kernel matrix, K, is both symmetric and positive semi-definite (PSD) meaning $K_{ij} = K_{ji}$ and $z^TKz \geq 0 \forall z$

Solution

(a) $K(x,z) = K_1(x,z) + K_2(x,z)$ is a valid kernel

Proof

Note that K_1 and K_2 are valid kernels and thus are both symmetric and positive semi-definite (PSD)

$K(x,z) = K_1(x,z) + K_2(x,z) = K_1(z,x) + K_2(z,x) = K(z,x) \longrightarrow$ symmetric

$a^TKa = a^T(K_1+K_2)a = a^TK_1a + a^TK_2a \geq 0$ (since both K_1 and K_2 are PSD)

$\longrightarrow$ K is PSD $\longrightarrow$ K is a valid kernel since both symmetric and PSD $\diamond$

(b) $K(x,z) = K_1(x,z) - K_2(x,z)$ is not a valid kernel

Counter-example

Consider $K_1(x,z) = x^T z$ and $K_2(x,z) = 2x^T z$ (both valid...check yourself)
$a^T K a = a^T(K_1 - K_2)a = a^T(K_1 - 2K_1)a = a^T(-1K_1)a \leq 0$ (since K_1 is PSD...
Thus, K is not PSD and thus, not a valid kernel

(c) $K(x,z) = aK_1(x,z)$ is a valid kernel

Proof

$K(x,z) = aK_1(x,z) = aK_1(z,x) = K(z,x) \longrightarrow$ symmetric
$z^T K z = z^T(aK_1)z = a(z^T K_1 z) \geq 0$ (since K_1 is PSD and a is positive)
$\longrightarrow$ K is PSD $\longrightarrow$ K is a valid kernel since both symmetric and PSD $\diamond$

(d) $K(x,z) = -aK_1(x,z)$ is not a valid kernel

Counter-example

Consider $K_1(x,z) = x^T z$ and $a = 1$
$z^T K z = z^T(-aK_1)z = z^T(-K_1)z = -z^T(K_1)z \leq 0$ (since K_1 is PSD... Thus, K is not PSD and thus, not a valid kernel

(e) $K(x,z) = K_1(x,z)K_2(x,z)$ is a valid kernel

Proof

$K(x,z) = K_1(x,z)K_2(x,z) = K_1(x,z)K_2(z,x) = K(z,x) \longrightarrow$ symmetric
PSD:

$$z^T K z = \sum_{ij=1}^{n} z_i K_{ij} z_j = \sum_{ij=1}^{n} z_i K_{1_{ij}} K_{2_{ij}} z_j$$

$$= \sum_{ij=1}^{n} z_i(\phi_1(x^i)^T \phi_1(x^j))(\phi_2(x^i)^T \phi_2(x^j)) z_j \quad (1)$$

$$= \sum_{ij=1}^{n} z_i(\sum_{k=1}^{n} \phi_1(x^i)_k \phi_1(x^j)_k)(\sum_{l=1}^{n} \phi_2(x^i)_l \phi_2(x^j)_l) z_j \quad (2)$$

$$= \sum_{ij=1}^{n} \sum_{kl=1}^{n} z_i \phi_1(x^i)_k \phi_1(x^j)_k) \phi_2(x^i)_l \phi_2(x^j)_l z_j \quad (3)$$

$$= \sum_{kl=1}^{n} \sum_{ij=1}^{n} (z_i \phi_1(x^i)_k \phi_2(x^i)_l)(z_j \phi_1(x^j)_k \phi_2(x^j)_l) \quad (4)$$

$$= \sum_{kl=1}^{n} (\sum_{i=1}^{n} z_i \phi_1(x^i)_k \phi_2(x^i)_l)^2 \geq 0 \quad (5)$$

$\longrightarrow$ K is PSD $\longrightarrow$ K is a valid kernel since both symmetric and PSD $\diamond$

(1) Uses the fact that K_1 and K_2 are valid kernels and thus have corresponding feature maps ϕ_1 and ϕ_2 that they perform the inner product of (x,z) over

(2) Expands the inner products of ϕ_1 and ϕ_2 (3) Simplifies, placing them all into the same sum (4) Re-orders the sums and factorizes to separate i and j (5) Uses the fact that $\sum_{ij} a_i a_j = (\sum_i a_i)^2$

(f) $K(x,z) = f(x)f(z)$ is a valid kernel

<u>Proof</u>

Simply note that if K is a valid kernel, there must be some feature map ϕ_k such that $K(x,z) = \phi_k(x)^T\phi_k(z)$

In this case, $\phi_k = f$ and $\in R^1 \longrightarrow$ K is a valid kernel $\diamond$

(g) $K(x,z) = K_3(\phi(x), \phi(z))$ is a valid kernel

<u>Proof</u>

$K(x,z) = K_3(\phi(x), \phi(z)) = K_3(\phi(z), \phi(x)) = K(z,x) \longrightarrow$ symmetric

$z^T K z = \sum_{ij=1}^n z_i K_{ij} z_j = \sum_{ij=1}^n z_i K_3(\phi(x^i), \phi(x^j)) z_j \geq 0$

(Since the Kernel matrix K_3 defined $\forall \phi(x^{(i)}), i \in (1,n)$ is PSD since K_3 is a valid kernel)

$\longrightarrow$ K is PSD $\longrightarrow$ K is a valid kernel since both symmetric and PSD $\diamond$

(h) $K(x,z) = p(K_1(x,z))$ is a valid kernel

<u>Proof</u>

First, we'll prove that $(K_1(x,z))^n$ is a valid Kernel by induction

<u>Proof</u>:

Note that for n=1, $K_1(x,z)^1$ is a valid kernel since K_1 is valid

Assume true for n=k: $(K_1(x,z))^k$ is a valid Kernel

Consider n=k+1: $(K_1(x,z))^{k+1} = K_1(x,z)(K_1(x,z))^k$ is a valid kernel due to part (e) where we proved that $K(x,z) = K_1(x,z)K_2(x,z)$ is valid if K_1 and K_2 are valid. In this case, $K_1 = K_1 and K_2 = (K_1(x,z))^k$ which we've assumed to be valid in the inductive step.

$\longrightarrow$ Thus, $(K_1(x,z))^n$ is a valid Kernel $\diamond$

Now we'll show that $p(K_1(x,z))$ can be written as the inner product over a feature map ϕ of x and z and is thus a valid kernel

<u>Proof</u>

$$p(K_1(x,z)) = \sum_{i=0}^{n} a_i (K_1(x,z))^i$$

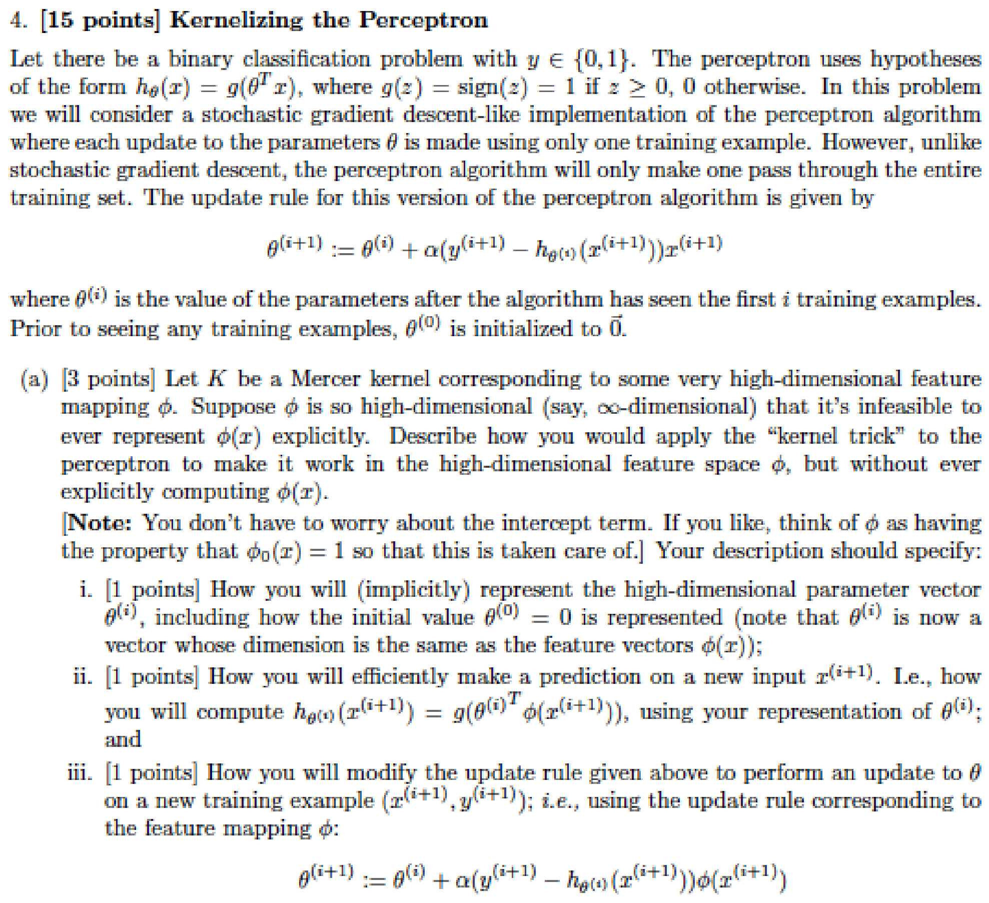
$$= \sum_{i=0}^{n} a_i \phi_i(x)\phi_i(z) \quad (1)$$

$$= \phi(x) \cdot \phi(z) \quad (2)$$

$\longrightarrow$ Thus, $K(x,z) = p(K_1(x,z))$ is a valid Kernel $\diamond$

(1) $\phi_i(x)\phi_i(z)$ refers to the corresponding inner product of $K_1(x,z)^i$ (2) $\phi(x) = [a_0^{0.5}, a_1^{0.5}\phi_1(x), ..., a_n^{0.5}]$

2.4 4. [15 points] Kernelizing the Perceptron



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:

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

Solution

(i) $\theta^{(i)} = \sum_{j=1}^{i} \beta_j \phi(x^{(j)})$ where $\beta_j = \alpha(y^{(j)} - g({\theta^{(j-1)}}^T \phi(x^{(j)})))$
(ii)

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

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

$$\text{Thus, } \beta_j = \beta_j \forall (1 \leq j \leq i) \text{ and}$$

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

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

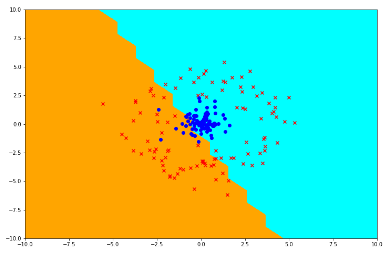
We provide two kernels, a dot-product kernel and a radial basis function (RBF) kernel. Run `src/perceptron/perceptron.py` to train kernelized perceptrons on `src/perceptron/train.csv`. The code will then test the perceptron on `src/perceptron/test.csv` and save the resulting predictions in the `src/perceptron/` folder. Plots will also be saved in `src/perceptron/`.

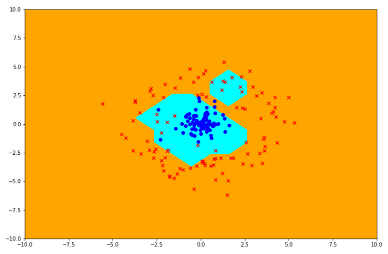
Include the two plots (corresponding to each of the kernels) in your writeup, and indicate which plot belongs to which kernel.

(c) [2 points]

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

Solution





dot kernel (left); rbf kernel (right)

The reason the dot product kernel performs poorly is that it does not transform the input into a different feature space and thus, it maintains the non-linear separability of the data.

2.5 5. [25 points] Neural Networks: MNIST image classification

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.

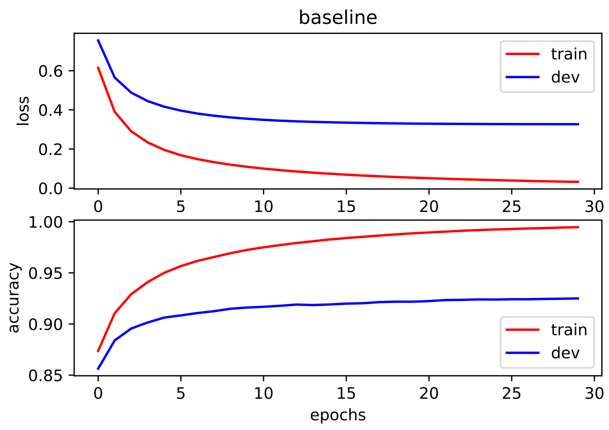
Train the model with mini-batch gradient descent as described above. Run the training for 30 epochs. At the end of each epoch, calculate the value of loss function averaged over the entire training set, and plot it (y-axis) against the number of epochs (x-axis). In the same image, plot the value of the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



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

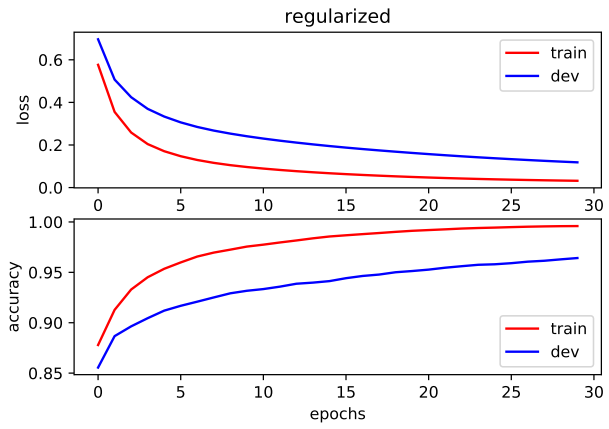
$$J_{MB} = \left(\frac{1}{B} \sum_{i=1}^{B} CE(y^{(i)}, \hat{y}^{(i)}) \right) + \lambda \left(||W^{[1]}||^2 + ||W^{[2]}||^2 \right)$$

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.



The baseline model appears to begin over-fitting compared to the regularized model as can be seen by the dev set accuracy flattening compared to the train set accuracy. This can also be seen in the losses of the regularized model continually decreasing with more epochs.

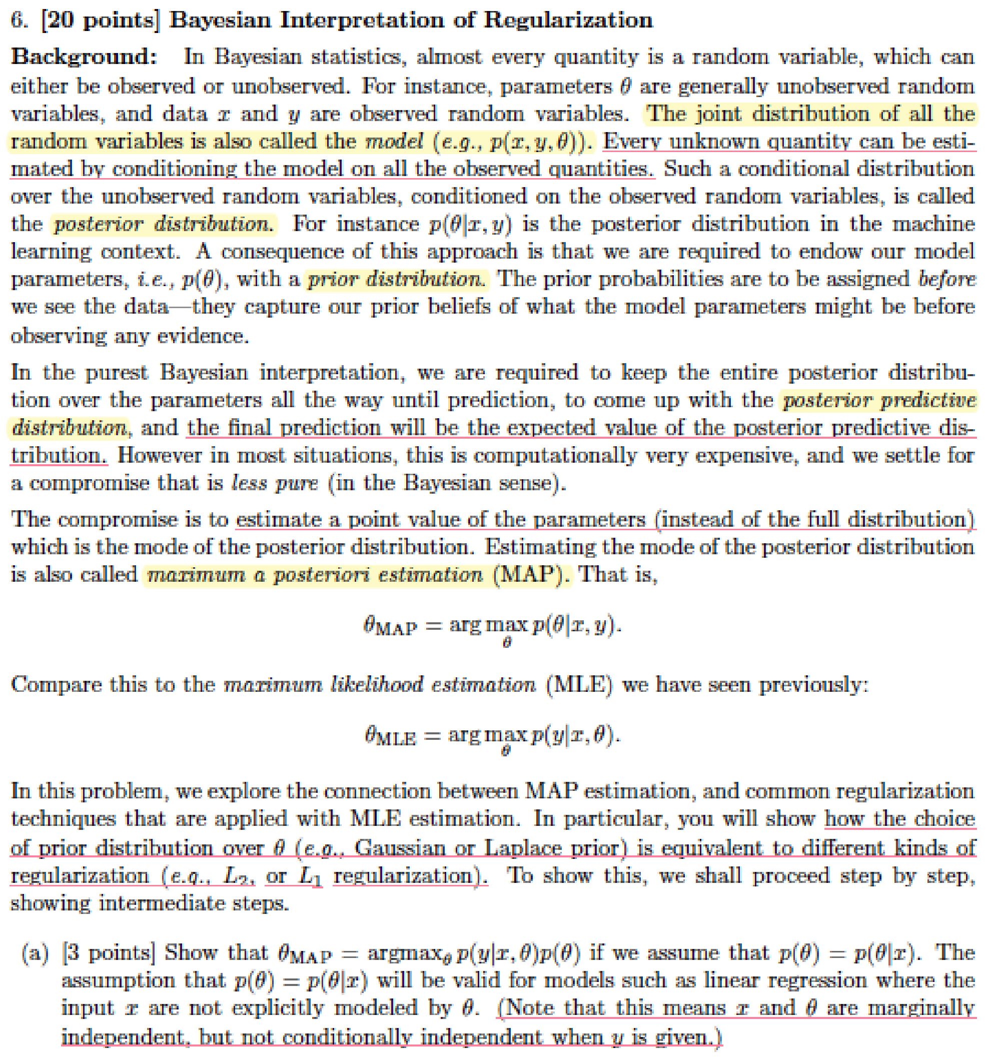
(c) **[3 points]** All this while you should have stayed away from the test data completely. Now that you have convinced yourself that the model is working as expected (i.e, the observations you made in the previous part matches what you learnt in class about regularization), it is finally time to measure 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.

Baseline model test accuracy : 0.929
Regularized model test accuracy : 0.967

2.6 6. [20 points] Bayesian Interpretation of Regularization



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.)

Solution

$\theta_{MAP} = argmax_\theta\; p(\theta|x, y) \;= argmax_\theta\; \frac{p(y,x|\theta)p(\theta)}{p(y,x)} = argmax_\theta\; p(y, x|\theta)p(\theta)$
$= argmax_\theta\; p(y|x, \theta)p(x|\theta)p(\theta) \;= argmax_\theta\; p(y|x, \theta)p(\theta|x)p(x)$
$= argmax_\theta\; p(y|x, \theta)p(\theta|x)$
$= argmax_\theta\; p(y|x, \theta)p(\theta) \quad \diamond$

(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?

Solution

$\theta_{MAP} = argmax_\theta\ P(y|x,\theta)P(\theta)$ (from previous answer)
$= argmax_\theta\ log(P(y|x,\theta)P(\theta)) = argmax_\theta\ log(P(y|x,\theta)) + log(P(\theta))$ (since log is concave)
$= argmax_\theta\ log(P(y|x,\theta)) + log(\frac{1}{(2\pi)^{d/2}|n^2 I|^{1/2}} exp(\frac{-1}{2}\theta^T(n^2 I)^{-1}\theta))$
$= argmax_\theta\ log(P(y|x,\theta)) + \frac{-1}{2}(\theta^T(n^2 I)^{-1}\theta)$
$= argmin_\theta\ -log(P(y|x,\theta)) + \frac{1}{2n^2}(\theta^T\theta)$
$= argmin_\theta\ -log(P(y|x,\theta)) + \lambda|\theta|_2^2$ where $\lambda = \frac{1}{2n^2}$ $\diamond$

(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}.

Solution

Continuing from the previous question, what we want to minimize is:

$$L(\theta) = -log(P(\vec{y}|X,\theta)) + \lambda|\theta|_2^2$$

$$= -log(\frac{1}{\sqrt{2\pi}\sigma} exp(\frac{-(\vec{y} - X\theta)^2}{2\sigma^2})) + \lambda|\theta|_2^2$$

$$= log(\sqrt{2\pi}\sigma) + \frac{(\vec{y} - X\theta)^2}{2\sigma^2}) + \lambda|\theta|_2^2$$

$$= log(\sqrt{2\pi}\sigma) + \frac{(\vec{y}^T\vec{y} + \theta^T(X^TX)\theta - 2\vec{y}^TX\theta}{2\sigma^2} + \lambda\theta^T\theta$$

Taking the derivative with respect to θ of this gives

$$\frac{\partial L}{\partial \theta} = 0 = \frac{1}{2\sigma^2}(0 + 2X^TX\theta - 2(X^T\vec{y})) + 2\lambda\theta$$

$$0 = (\frac{X^TX}{\sigma^2} + 2\lambda I)\theta - \frac{X^T\vec{y}}{\sigma^2}$$

$$\rightarrow X^T\vec{y} = (X^TX + 2\lambda\sigma^2 I)\theta$$

$$\longrightarrow \theta = (X^T X + 2\lambda\sigma^2 I)^{-1} X^T \vec{y} \quad \diamond$$

We know the inverse exists because $X^T X + 2\lambda\sigma^2 I$ is positive-definite

(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.

Solution

$$\theta_{MAP} = argmax_\theta \; p(\theta|x, y) = argmax_\theta \; P(y|x, \theta)P(\theta) \quad (pt\ a)$$

$$= argmax_\theta \; log(P(\vec{y}|X, \theta)) + log(P(\theta))$$

$$= argmax_\theta \; log(\frac{1}{\sqrt{2\pi}\sigma} exp(\frac{-(\vec{y} - X\theta)^2}{2\sigma^2})) + log(\prod_{i=1}^{d} P(\theta_i))$$

$$= argmin_\theta \; \frac{(\vec{y} - X\theta)^2}{2\sigma^2}) - \sum_{i=1}^{d} log(P(\theta_i))$$

$$= argmin_\theta \; \frac{(\vec{y} - X\theta)^2}{2\sigma^2}) - \sum_{i=1}^{d} log(\frac{1}{2b} exp(\frac{-|\theta_i|}{b}))$$

$$= argmin_\theta \; \frac{(\vec{y} - X\theta)^2}{2\sigma^2}) + \sum_{i=1}^{d} \frac{|\theta_i|}{b}$$

$$= argmin_\theta \; \frac{(\vec{y} - X\theta)^2}{2\sigma^2}) + \frac{1}{b}||\theta||_1$$

$$= argmin_\theta \; \frac{(\vec{y} - X\theta)^2 + \frac{2\sigma^2}{b}||\theta||_1}{2\sigma^2}$$

$$= argmin_\theta \; (\vec{y} - X\theta)^2 + \frac{2\sigma^2}{b}||\theta||_1$$

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

where $\gamma = \frac{2\sigma^2}{b}$