CS 229 Problem Set 2 Solutions

Justina Žurauskienė
SUNet:

2023 Summer

Problem 1: Spam classification

a) **[5 points]** Implement code for processing the spam messages into numpy arrays that can be fed into machine learning models. Do this by completing the `get_words`, `create_dictionary`, and `transform_text` functions within our provided `src/spam.py`. Do note the corresponding comments for each function for instructions on what specific processing is required. The provided code will then run your functions and save the resulting dictionary into `spam_dictionary` and a sample of the resulting training matrix into `spam_sample_train_matrix`. In your writeup, report the vocabulary size after the preprocessing step. You do not need to include any other output for this subquestion.

Answer:

For completeness here I include both results (see Ed post #345).

Size of dictionary is **1721 words**, when using *python* command

```
split()
```

Size of dictionary is **1722 words**, when using *python* command

```
split(" ")
```

These options do not change final results (i.e. accuracy and top five words).

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:

The accuracy of the trained Naive Bayes model on the test-set is: **0.978** (rounded to three decimal places)

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:

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

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

Answer:

The top five words are:

['claim', 'won', 'prize', 'tone', 'urgent!']

Problem 2: 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 the direct construction of kernels. In other words, 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, although the description above is sufficient for this problem.

In this question, we are interested in seeing which operations preserve the validity of kernels. Let K_1 and K_2 be kernels over $\mathbb{R}^d \times \mathbb{R}^d$, let $a \in \mathbb{R}^+$ be a positive real number, let $f : \mathbb{R}^d \to \mathbb{R}$ 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 $p(x)$ be 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.

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

Answer:

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

Here, we will be checking if the following properties hold; Thus, for K to be a valid kernel it should be PSD, i.e. satisfy two properties:

- Underlying matrix G must be symmetric,
- and $z^T G z \geq 0$.

Here we assume that G_1, G_2 are constructed using kernels K_1, K_2 respectively; Thus,

- This G is symmetric, because

$$\boxed{G_{ij}} = G_1(x^{(i)}, z^{(j)}) + G_2(x^{(i)}, z^{(j)}) = \phi_1(x^{(i)})^T \phi_1(z^{(j)}) + \phi_2(x^{(i)})^T \phi_2(z^{(j)}) =$$

$$= \phi_1(z^{(j)})^T \phi_1(x^{(i)}) + \phi_2(z^{(j)})^T \phi_2(x^{(i)}) = G_1(z^{(j)}, x^{(i)}) + G_2(z^{(j)}, x^{(i)}) = \boxed{G_{ji}}.$$

- Matrix G is also PSD, because for arbitrary y, we have the following,

$$\begin{aligned}
y^T G y &= \sum_i \sum_j y_i G_{ij} y_j \\
&= \sum_i \sum_j y_i \left[G_{1_{ij}} + G_{2_{ij}} \right] y_j \\
&= \sum_i \sum_j y_i \left[\phi_1(x^{(i)})^T \phi_1(z^{(j)}) + \phi_2(x^{(i)})^T \phi_2(z^{(j)}) \right] y_j = \\
&= \sum_i \sum_j y_i \phi_1(x^{(i)})^T \phi_1(z^{(j)}) y_j + \sum_i \sum_j y_i \phi_2(x^{(i)})^T \phi_2(z^{(j)}) y_j = \\
&= \sum_i \sum_j y_i G_{1_{ij}} y_j + \sum_i \sum_j y_i G_{2_{ij}} y_j = \\
&= \underbrace{y^T G_1 y}_{\geq 0} + \underbrace{y^T G_2 y}_{\geq 0} \geq 0 \quad \text{see PS1Q1a supporting result.}
\end{aligned}$$

Thus, kernel constructed as $K(x, z) = K_1(x, z) + K_2(x, z)$ is <u>valid kernel.</u>

Alternative strategy to show this:

If we denote G_1 to be a Gram matrix (symmetric by definition) constructed using K_1 and G_2 – a Gram matrix constructed using kernel K_2; then a Gram matrix constructed using $K = K_1 + K_2$ can be denoted as $G = G_1 + G_2$. Thus, $\forall z : z^T G z = z^T (G_1 + G_2) z = \underbrace{z^T G_1 z}_{\geq 0} + \underbrace{z^T G_2 z}_{\geq 0} \geq 0$.

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

- Underlying matrix G is symmetric following logic/proof from above;
- However it is not guaranteed that resultant G will comply with PSD property; For this we can consider a simple example with matrices A and B, defined as,

$$A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad \text{and} \quad B = 2 * A = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \quad \Rightarrow \quad D = A - B = \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}$$

From [ps0q2] we know that identity matrix is PSD, thus A and B defined in such way are PSD matrices; However matrix D, defined as a difference between A and B, is not a PSD, because we can find at least one vector $x^T = [1, 1]$ that results in $x^T D x = -2$.

In more general case, if we denote G_1 and G_2 be two Gram matrices constructed using kernels K_1 and K_2 respectively; also $g_1, g_2 \in \mathrm{R}^+ : g_2 > g_1$ and $x^T G_1 x = g_1$ and $x^T G_2 x = g_2$, then $x^T G x = x^T (G_1 - G_2) x = x^T G_1 x - x^T G_2 x = g_1 - g_2 \not\geq 0$, meaning matrix G is not PSD and underlying K is <u>not a valid kernel.</u>

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

We know that:

> If A is a PSD matrix, i.e., it is symmetric and $x^TAx \geq 0$ for all $x \in \mathbb{R}^n$, then for any positive real scalar $a \in \mathrm{R}^+$, the matrix aA is also PSD, because $x^T(aA)x = a(x^TAx) \geq 0$ for all $x \in \mathbb{R}^n$.

Let G_1 be Gram matrix constructed using kernel K_1. Then aG_1 will be symmetric and PSD, using statement from above; Thus kernel K constructed in such way will be a valid kernel.

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

Following logic presented in (b) we can express $D = -1 \cdot I$, we see that such K is not a valid kernel.

In more general case, using above supporting statement in (c): Let G_1 be Gram matrix corresponding to kernel K_1, then $x^TGx = x^T(-aG_1)x = -a\underbrace{x^TG_1x}_{\geq 0} \not\geq 0$.

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

Let G_1 and G_2 be two Gram matrices (by definition symmetric and PSD) that correspond to kernels K_1 and K_2 respectively. We can define G to be the Hadamard product between G_1 and G_2, i.e., $G = G_1 \circ G_2$. Since both G_1 and G_2 are symmetric and positive semidefinite, we can express them in terms of their eigenvalues and eigenvectors using spectral theorem; we also know that if matrix $A \succeq 0$ it follows that all $\lambda_i(A) \geq 0, i = 1..n$; Therefore, for an arbitrary vector y, we have

$$
\begin{aligned}
y^TGy &= y^T(G_1 \circ G_2)y = y^T\left[U_1\Lambda_1U_1^T \circ U_2\Lambda_2U_2^T\right]y = \\
&= y^T\left[\sum_i\sum_j \lambda_1^{(i)}u_1^{(i)}u_1^{(i)^T} \circ \lambda_2^{(j)}u_2^{(j)}u_2^{(j)^T}\right]y = \\
&= y^T\left[\sum_i\sum_j \underbrace{\lambda_1^{(i)}\lambda_2^{(j)}}_{\geq 0}\underbrace{(u_1^{(i)} \circ u_2^{(j)})(u_1^{(i)} \circ u_2^{(j)})^T}_{\text{outer product}}\right]y \geq 0 \quad \text{also see matching result here}
\end{aligned}
$$

where Λ_1, Λ_2 and U_1, U_2 hold eigenvalues and eigenvectors for each Gram matrix (kernel) respectively.

Therefore, K constructed in such a way is a valid kernel.

Alternatively, we know that Hadamard product is also known as (element-wise product, Schur product). Therefore, as per Schur product theorem, which states that the Hadamard (element-wise) product of two positive semi-definite matrices is also positive semi-definite, therefore K constructed in such a way is a valid kernel.

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

For simplicity here, we assume that f is a square-integrable function from $\mathbb{R}^d$ to $\mathbb{R}$; Mercer's theorem can be applied to such functions as well. Therefore for $K(x,z) = f(x)f(z)$ to be a valid kernel, it should satisfy Mercer's condition that states: for all square-integrable functions g this expression is non-negative:

$$\int\int g(x)K(x,y)g(y)dxdy \geq 0$$

In our case, $K(x,z) = f(x)f(z)$, thus the integral would become,

$$\int\int g(x)f(x)f(z)g(z)dxdz = \int f(x)g(x)dx \int f(z)g(z)dz \equiv \left(\int f(x)g(x)dx\right)^2 \geq 0$$

This follows because we assume that we are integrating over the whole domain and $\int f(x)g(x)dx = \int f(z)g(z)dz$ with x, z being arbitrary. Thus, K here is a valid kernel.

Alternatively, we could argue that because function's f output is a real number, we can denote $\phi(x) = f(x)$ and $\phi(z) = f(z)$; then kernel defined as $f(x)f(z)$ can be expressed as per provided definition $K(x,z) = \phi(x)^T\phi(z)$ meaning that scalar (obtained as inner product in 1D, which correspond to product of two numbers) is a valid kernel.

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

We know that K_3 is a valid kernel function defined over $\mathbb{R}^p \times \mathbb{R}^p$ space; whereas, given the feature mapping $\phi(x)$ and $\phi(z)$ will output $\mathbb{R}^p$ objects, that will fall into K_3 operation domain. Thus, here $K(x,z)$ is a valid kernel.

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

Here, polynomial p is defined using kernel K_1 as variable with positive coefficients; because such polynomial can be expressed as,

$$\sum_{l=0}^{n} a_l K_1^l(x,z)$$

it represents a linear combination of kernel taken to different powers that is weighted by positive coefficients. We know that power operation in polynomial can be substituted using product operation of same variable, therefore multiplying kernel by kernel will result in valid kernel. Further, we previously have demonstrated that we can get valid kernels by adding kernels together and by weighting kernels with positive scalars. Thus, here $K(x,z)$ is valid kernel.

Additional clarifying statement: For clarity here we specifically assume that "power of the kernel function K", refers to the situation where function K is applied l times, thus resulting in the Hadamard product l times of underlying Gram matrices $(G\circ, .., \circ G)$, and not to the matrix product $(G\cdot, ..., \cdot G)$.

Problem 3: 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 $z \geq 0$, and 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

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

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

Answer:

Here, we can replace all instances of scalar products, $\theta^T x$ with kernel function K. This kernel corresponds to scalar product in some (possible infinite dimensional) feature space, meaning K implicitly represents original product of $\theta^T x$. Thus, we can express original parameters θ as,

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

here β_j are coefficients that we need to learn, and $\phi(x^{(j)})$ is the high-dimensional feature mapping of the data points. The initial value $\theta^{(0)} = \vec{0}$ can be represented by setting $\beta_j = 0$. Since we are here interested in $\theta^{(i)}$, rather θ, the expression will become:

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

which depends on the first i data points.

ii. **[1 point]** 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 $\theta^{(i)}$;

Answer:

From lecture notes (in general), we know that,

$$h_\theta(x) \equiv \theta^T x = \left[\sum_{j=1}^{n} \beta_j \phi(x^{(j)})\right]^T \phi(x) = \sum_{j=1}^{n} \beta_j \langle \phi(x^{(j)}), \phi(x) \rangle = \sum_{j=1}^{n} \beta_j K(x^{(j)}, x).$$

Thus, in order to amend the expression for making new predictions:

$$h_{\theta^{(i)}}(x^{(i+1)}) = g(\theta^{(i)T}\phi(x^{(i+1)})$$

we replace right side of $h_{\theta^{(i)}}(x^{(i+1)})$ with kernel-based formulation:

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

iii. **[1 point]** 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:

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

Answer:

Here, to modify given update rule we will replace $h_{\theta^{(i)}}(x^{(i+1)})$ with derived, kernel-based, expression:

$$\theta^{(i+1)} := \theta^{(i)} + \alpha \left(y^{(i+1)} - g \left[\sum_{j=1}^{i} \beta_j K(x^{(j)}, x^{(i+1)}) \right] \right) \phi(x^{(i+1)})$$

However, we do not explicitly compute or store the "high-dimensional" parameter vector θ, and instead, we store the coefficients β; thus replacing $\theta^{(i)}$, with sum term,

$$\theta^{(i+1)} := \sum_{j=1}^{i} \beta_j \phi(x^{(j)}) + \underbrace{\alpha \left(y^{(i+1)} - g \left[\sum_{j=1}^{i} \beta_j K(x^{(j)}, x^{(i+1)}) \right] \right)}_{\text{notice rule for updating } \beta\text{'s}} \phi(x^{(i+1)})$$

Thus, in case of perceptron algorithm, we have a new efficient way given by,

$$\boxed{\beta_{i+1} = \alpha \left(y^{(i+1)} - g \left[\sum_{j=1}^{i} \beta_j K(x^{(j)}, x^{(i+1)}) \right] \right).}$$

(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 kernels: a dot-product kernel defined as

$$K(x, z) = x^T z \quad (1)$$

a radial basis function (RBF) kernel, defined as

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

and finally the following function:

$$K(x, z) = \begin{cases} -1 & x = z \\ 0 & x \neq z \end{cases} \quad (3)$$

Note that the last function is not a kernel function (since its corresponding matrix is not a positive semi-definite 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 write-up, and indicate which plot belongs to which function.

Answer:

Please see corresponding plots (kernels are specified in captions).

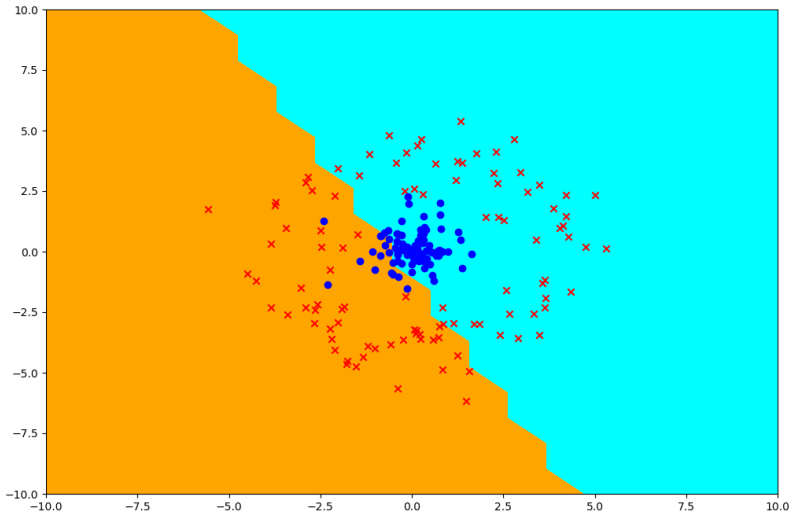


Figure 1: This figure corresponds to dot-product kernel given in eq:(1)

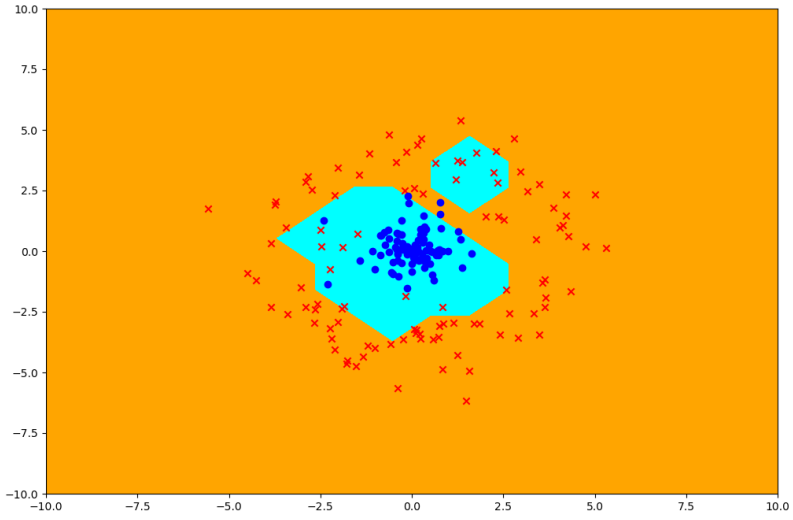


Figure 2: This figure corresponds to RBF kernel given in eq:(2)

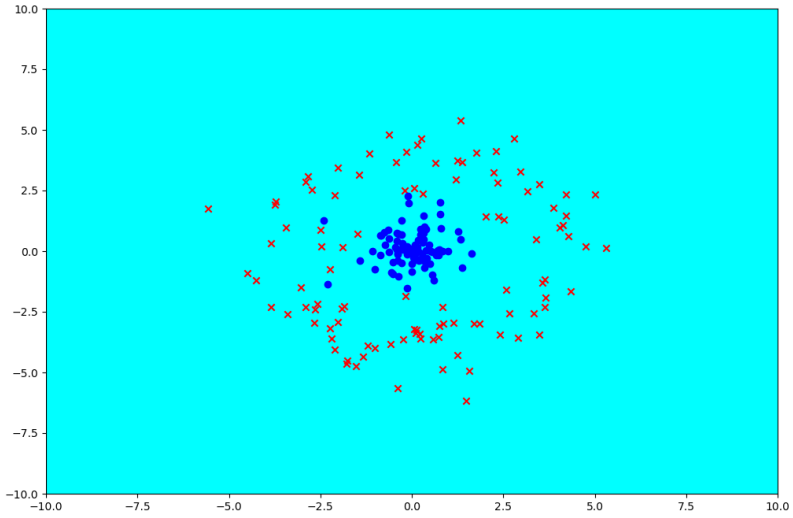


Figure 3: This figure corresponds to non-PSD kernel given in eq:(3)

(c) **[2 points]** One of the choices in Q3b completely fails, one works a bit, and one works well in classifying the points. Discuss the performance of different choices and why do they fail or perform well?

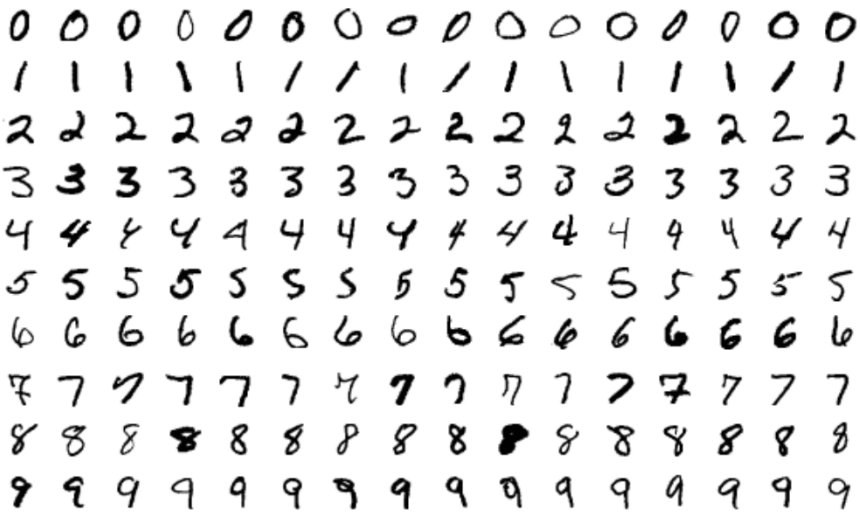
Answer:

From analysis and plots we can see the following outcomes:

- When the kernel is defined as a dot product, $K(x, z) = x^T z$, we have a linear kernel function. In this case, we are essentially performing the standard perceptron algorithm, as the kernel trick reduces to the original formulation. However, since the data forms complex patterns in 2D space that are not linearly separable, a linear decision boundary (which is what we get with a linear kernel) does not perform well.
- By introducing the Radial Basis Function (RBF) kernel, we achieve a non-linear decision boundary, which improves the separation of the classes. The RBF kernel computes here a non-linear transformation of the input vectors, meaning the perceptron algorithm can learn more complex relationships present in our data. This kernel maps the input data to a higher-dimensional space where data could possibly be separated-out in linear way; this could explain observable better performance.
- Using function that does not pass PSD kernel property, present completely failed results. This is because by definition we expect a kernel function that is able to measure similarity between data points. If function is PSD (kernel), it follows that there is a corresponding dot product in (possibly) high-dimensional feature space. Given, that provided function is not a real kernel, it cannot capture/measure similarity between data because it does not correspond to a dot product in any feature space. Lack of this PSD property can result in algorithm to not converging or producing nonsensical results.

Problem 4: 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 the 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:

$$a^{(i)} = \sigma\left({W^{[1]}}^\top x^{(i)} + b^{[1]}\right) \in \mathbb{R}^h$$

$$z^{(i)} = {W^{[2]}}^\top a^{(i)} + b^{[2]} \in \mathbb{R}^K$$

$$\hat{y}^{(i)} = \text{softmax}(z^{(i)}) \in \mathbb{R}^K$$

where σ is the sigmoid function.

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

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

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:

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

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

(a) **[5 points]**

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

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

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

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

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

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

Answer:

The cross-entropy loss for one-hot encoded labels can be defined in the following way (we disregard the outer sum here, because we are considering only a single pair of example/label),

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

Because we only have a single "1" in lth position of the one-hot vector with the rest being zeros, we can simplify above expression by only considering non-zero position l; thus

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

here l is the index with non-zero entry. Given this expression, we can compute $\frac{\partial \text{CE}(y^{(i)}, \hat{y}^{(i)})}{\partial z_j^{(i)}}$ for $j \neq l$ and $j = l$ separately.

Derivative of softmax function:

Softmax is defined as,

$$y^{(i)} = \frac{e^{z_l^{(i)}}}{\sum_{k=1}^{K} e^{z_k^{(i)}}}$$

- If $j \neq l$, we have,

$$\frac{\partial y^{(i)}}{\partial z_j^{(i)}} = \frac{0 \cdot \sum_{k=1}^{K} e^{z_k^{(i)}} - e^{z_j^{(i)}} e^{z_l^{(i)}}}{\left(\sum_{k=1}^{K} e^{z_k}\right)^2} = -\frac{e^{z_j^{(i)}}}{\sum_{k=1}^{K} e^{z_k}} \cdot \frac{e^{z_l^{(i)}}}{\sum_{k=1}^{K} e^{z_k}} = -y_l^{(i)} \cdot y_j^{(i)}.$$

- If $j = l$, we have,

$$\frac{\partial y^{(i)}}{\partial z_j^{(i)}} = \frac{e^{z_l^{(i)}} \cdot \sum_{k=1}^{K} e^{z_k^{(i)}} - e^{z_j^{(i)}} e^{z_l^{(i)}}}{\left(\sum_{k=1}^{K} e^{z_k^{(i)}}\right)^2} = \frac{e^{z_l^{(i)}}}{\sum_{k=1}^{K} e^{z_k^{(i)}}} \cdot \left(1 - \frac{e^{z_j^{(i)}}}{\sum_{k=1}^{K} e^{z_k^{(i)}}}\right) = y_l^{(i)} \cdot (1 - y_j^{(i)})$$

These cases can be absorbed into a single expression by considering Kronecker notation:

$$\frac{\partial y^{(i)}}{\partial z_j^{(i)}} = y_l^{(i)} \cdot (\delta_{jl} - y_j^{(i)}), \quad \text{where} \quad \delta_{jl} = \begin{cases} 1 & j = l \\ 0 & j \neq l \end{cases}.$$

- **Case $j \neq l$**:

$$\frac{\partial \text{CE}(y^{(i)}, \hat{y}^{(i)})}{\partial z_j^{(i)}} = \frac{\partial CE}{\partial \hat{y}_j^{(i)}} \cdot \frac{\partial \hat{y}_j^{(i)}}{\partial z_j^{(i)}} = -0 \cdot \hat{y}_l^{(i)} \cdot (\delta_{jl} - \hat{y}_j^{(i)}) = 0, \quad \text{where} \quad \delta_{jl} = 0.$$

- **Case $j = l$**:

$$\frac{\partial \text{CE}(y^{(i)}, \hat{y}^{(i)})}{\partial z_l^{(i)}} = \frac{\partial CE}{\partial \hat{y}_l^{(i)}} \cdot \frac{\partial \hat{y}_l^{(i)}}{\partial z_l^{(i)}} = -\frac{1}{\hat{y}_l^{(i)}} \cdot \hat{y}_l^{(i)} \cdot (\delta_{ll} - \hat{y}_l^{(i)}) \quad \text{where} \quad \delta_{ll} = 1.$$

$$\Rightarrow \quad \frac{\partial \text{CE}(y^{(i)}, \hat{y}^{(i)})}{\partial z_l^{(i)}} = \hat{y}_l^{(i)} - 1$$

Returning to vector and one-hot encoding notation, we have that

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

here $y^{(i)}$ is a one-hot vector with a 1 in position l and 0s elsewhere.

(b) **[15 points]**

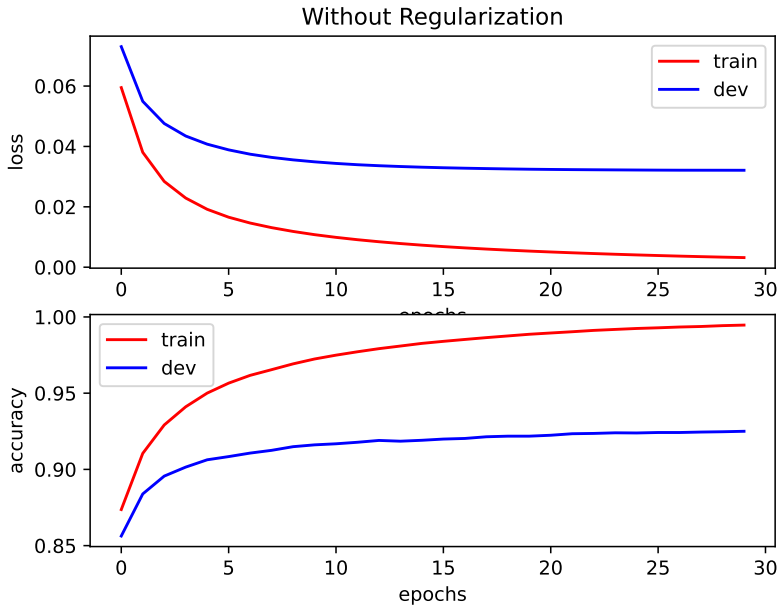
Implement both forward-propagation and back-propagation for the above loss function $J_{\text{MB}} = \frac{1}{B}\sum_{i=1}^{B} CE(y^{(i)}), \hat{y}^{(i)})$. Initialize the weights of the network by sampling values from a standard normal distribution. Initialize the bias/intercept term to 0. Set the num- ber 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:



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

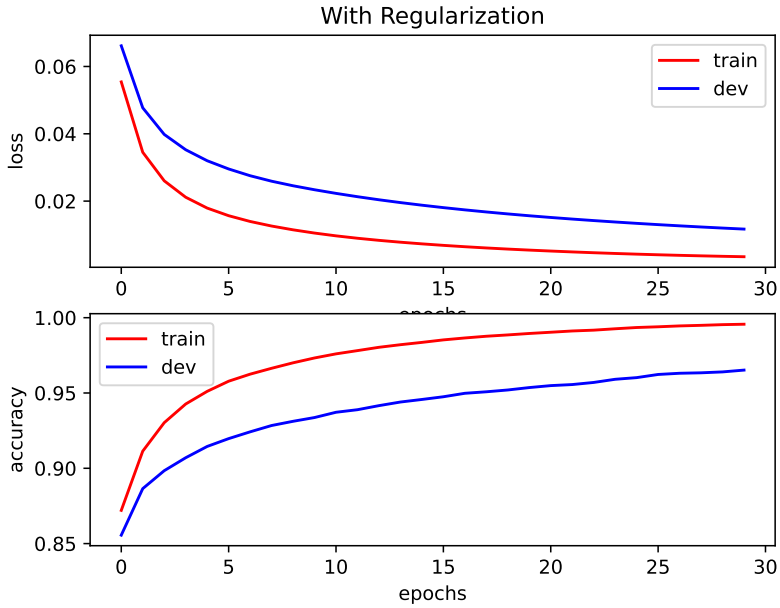
$$J_{\text{MB}} = \frac{1}{B}\sum_{i=1}^{B} \text{CE}(y^{(i)}, \hat{y}^{(i)}) + \lambda\|W^{[1]}\|^2 + \|W^{[2]}\|^2$$

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:



When comparing the plots (which display accuracy and loss against the number of epochs) of the regularized model with those of the non-regularized model, we can observe a better alignment between loss and accuracy across the training and development sets for the regularized model. Given their complexity, neural networks can be prone to overfitting. Therefore, the inclusion of a regularization term can enhance our model's ability to generalize to new data.

(d) **[3 points]** All this while you should have stayed away from the test data completely. Now that you have convinced yourself that the model is working as expected (i.e., the observations you made in the previous part matches what you learnt in class about regularization), it is finally time to measure the model performance on the test set. Once we measure the test set performance, we report it (whatever value it may be), and NOT go back and refine the model any further. Initialize your model from the parameters saved in part (a) (i.e., the non-regularized model), and evaluate the model performance on the test data. Repeat this using the parameters saved in part (b) (i.e., the regularized model). Report your test accuracy for both regularized model and non-regularized model. Briefly (in one sentence) explain why this outcome makes sense. You

should have accuracy close to 0.92870 without regularization, and 0.96760 with regularization. Note: these accuracies assume you implement the code with the matrix dimensions as specified in the comments, which is not the same way as specified in your code. Even if you do not precisely these numbers, you should observe good accuracy and better test accuracy with regularization.

Answer:

For model baseline, got accuracy: **0.928700**. Please see below corresponding plots for baseline model:

For model regularized, got accuracy: **0.967600**. Please see below corresponding plots for baseline model:

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

Regularization discourages learning a more complex model, thereby simplifying our model. This aligns with the idea that simpler models are less likely to overfit, thus enhancing their ability to generalize to unseen data. This effect is observed in our case as well; the regularized model achieved better accuracy on unseen data when compared to the baseline non-regularized neural network model.

Problem 5: 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,

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

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

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

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., L2 or L1 regularization). You will also explore how regularization strengths affect generalization in part (d).

(a) **[3 points]** Show that $\theta_{\text{MAP}} = \arg\max_\theta p(y|x,\theta)p(\theta)$ 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 is not explicitly modeled by θ. (Note that this means x and θ are marginally independent, but not conditionally independent when y is given.)

Answer:

We can use Bayes rule to write down the posterior distribution for parameters θ, as,

$$p(\theta|x,y) = \frac{\text{likelihood} \times \text{prior}}{\text{marginal probability}} = \frac{p(y|x,\theta)p(\theta|x)}{\int p(y|x,\theta)p(\theta|x)d\theta}$$

MAP can be obtained by maximising posterior probability; Since here we are interested in comparing θ's, and because marginal probability is just a normalising constant that does not depend on θ but integrates over all θ's, we can disregard it for the purpose of obtaining $\hat{\theta}_{MAP}$; Give this reasoning, posterior becomes,

$$p(\theta|x,y) \propto p(y|x,\theta)p(\theta|x); \quad | \quad \text{wher we take} \quad \arg\max_\theta$$

Assuming, x represents our input/feature space, the θ's are independent of x because of modelling assumptions; i.e. in Bayesian analysis, most often, we use prior distributions to express initial beliefs about model parameters before observing any data. This also means that the distribution over θ is the same regardless of x. This implies that the parameter θ does not depend on the input x explicitly. Therefore, we can re-write $p(\theta|x) = p(\theta)$, and in turn obtain desired expression,

$$\hat{\theta}_{MAP} \equiv \arg\max_\theta p(\theta|x,y) = \arg\max_\theta p(y|x,\theta)p(\theta).$$

(b) **[5 points]** Recall that L2 regularization penalizes the L2 norm of the parameters while minimizing the loss (i.e., negative log-likelihood in the case of probabilistic models). Now we will show that MAP estimation with a zero-mean Gaussian prior over θ, specifically $\theta \sim N(0, \eta^2 I)$, is equivalent to applying L2 regularization with MLE estimation. Specifically, show that for some scalar λ,

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

Also, what is the value of λ?

Answer:

For probabilistic model, the MAP estimate can be re-written using negative *log*-posterior, defined as

$$-\mathscr{L}(\theta) \propto -\log\left[p(y|x,\theta) \cdot p(\theta)\right]$$

Therefore, maximisation problem is replaced by the minimisation of NLP problem; this gives us,

$$\hat{\theta}_{\text{MAP}} \equiv \arg\min_\theta\{-\log(p(y|x,\theta)) - \log p(\theta)\}.$$

We will now consider special case, where prior $p(\theta) = \mathcal{N}(0, \eta^2 I)$. Assuming $\log \equiv \ln$, we have,

$$\log p(\theta) = \ln\left(\frac{1}{(2\pi)^{d/2}|\eta^2 I|^{1/2}} \exp\left\{-\frac{1}{2}\theta^T(\eta^2 I)^{-1}\theta\right\}\right) =$$

$$= -\frac{d}{2}\ln(2\pi) - \frac{1}{2}\ln|\eta^2 I| - \frac{1}{2}\theta^T(\eta^2 I)^{-1}\theta \propto \quad \text{(keeping terms with } \theta \text{ and using sum notation)}$$

$$\propto -\frac{1}{2}\sum_i \frac{\theta_i^2}{\eta^2} = -\frac{1}{2\eta^2}\|\theta\|_2^2,$$

where we denoted L_2 (Euclidean) norm of a vector θ as:

$$\|\theta\|_2 = \sqrt{\sum_i \theta_i^2}.$$

Thus, using this simplification we obtain final $\hat{\theta}_{MAP}$ expression,

$$\boxed{\hat{\theta}_{\text{MAP}} = \arg\min_\theta\{-\log p(y|x,\theta) + \lambda\|\theta\|_2^2\},} \quad \text{where } \boxed{\lambda = \frac{1}{2\eta^2}}.$$

(c) **[7 points]** Now consider a specific instance, a linear regression model given by $y = \theta^T x + \epsilon$ where $\epsilon \sim N(0, \sigma^2)$. 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 $\theta \sim N(0, \eta^2 I)$. 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:

To obtain $\hat{\theta}_{MAP}$ for linear regression model, we recall results obtained in (b), i.e.

$$\log p(\theta) \propto -\frac{1}{2}\theta^T(\eta^2 I)^{-1}\theta.$$

From lecture 6, we recall that in Bayesian Linear Regression we assume that the output $y^{(i)}$ is

$$y^{(i)} \sim \mathcal{N}(\theta^T x^{(i)}, \sigma^2 I)$$

normally distributed about the mean, which is given by the linear combination of the input and parameters, with variance σ^2; i.e. the expected value of $y^{(i)}$ is $\theta^T x^{(i)}$, and the variance is σ^2 shared across n examples. Therefore, multivariate Gaussian likelihood, is given by this expression,

$$p(\vec{y}|X,\theta,\sigma^2) = \frac{1}{(2\pi\sigma^2)^{n/2}} \exp\left\{-\frac{1}{2\sigma^2}(\vec{y} - X\theta)^T(\vec{y} - X\theta)\right\}$$

where: $y = (y^{(1)}, y^{(2)}, ..., y^{(n)})^T$ is the vector of output values, $X = (x^{(1)}, x^{(2)}, ..., x^{(n)})^T$ is the matrix of input features (design matrix), θ is vector containing regression coefficients, and σ^2 is the noise variance. Taking *log* of likelihood we get,

$$\log p(y|x,\theta) = -\frac{n}{2}\ln(2\pi\sigma^2) - \frac{1}{2\sigma^2}(\vec{y} - X\theta)^T(\vec{y} - X\theta) \propto -\frac{1}{2\sigma^2}(\vec{y} - X\theta)^T(\vec{y} - X\theta)$$

We can ignore terms that do not involve parameters θ; thus combining previously achieved results, we get,

$$-\log p(\theta|x,y) \propto -\log p(y|x,\theta) - \log p(\theta) = \frac{1}{2\sigma^2}(\vec{y} - X\theta)^T(\vec{y} - X\theta) + \frac{1}{2}\theta^T(\eta^2 I)^{-1}\theta$$

Similarly to the MLE case, we take derivatives of the negative *log*-posterior with respect to parameters θ. For this we can use supporting facts eq. (84, 81) from Matrix Cookbook (Version: November 15, 2012). Thus, we get,

$$\frac{\partial \log p(\theta|x,y)}{\partial \theta} = \frac{1}{2\sigma^2} \cdot \left[-2X^T(\vec{y} - X\theta)\right] + \frac{1}{2}\left[(\eta^{-2}I) + (\eta^{-2}I)\right]\theta = \eta^{-2}I\theta - \frac{1}{\sigma^2}X^T(\vec{y} - X\theta).$$

Equating this to 0, and solving for minimum, we will obtain,

$$\eta^{-2}I\theta - \frac{1}{\sigma^2}X^T(\vec{y} - X\theta) = 0 \quad \Rightarrow$$

$$\eta^{-2}I\theta - \frac{1}{\sigma^2}X^T\vec{y} + \frac{1}{\sigma^2}X^TX\theta = 0$$

$$\left(\eta^{-2}I + \frac{1}{\sigma^2}X^TX\right)\theta = \frac{1}{\sigma^2}X^T\vec{y} \quad | \cdot \sigma^2$$

$$\left(\frac{\sigma^2}{\eta^2}I + X^TX\right)\theta = X^T\vec{y} \quad \Rightarrow \quad \boxed{\hat{\theta}_{MAP} = \left[\frac{\sigma^2}{\eta^2}I + X^TX\right]^{-1}X^T\vec{y}}$$

Here, the additional term $\frac{\sigma^2}{\eta^2}I$ acts as a regulariser.

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

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

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

Show that θ_{MAP} in this case is equivalent to the solution of linear regression with L1 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 the linear regression problem with L1 regularization does not exist. To optimize this, we use gradient descent with a random initialization and solve it numerically.

Answer:

Here we are given Laplace prior distribution, which has the following expression,

$$p(\theta_i|0,b) = \frac{1}{2b}\exp\left\{-\frac{|\theta_i|}{b}\right\} \quad \Rightarrow \quad \log p(\theta_i|0,b) \propto -\frac{1}{b}|\theta_i|, \quad \text{for each } i = 1,2,...,d.$$

Then, because of provided independence property, we can sum obtained *log*-probabilities to get:

$$\log p(\theta|0,b) = \sum_{i=1}^{d} \log p(\theta_i|0,b) \propto -\frac{1}{b}\|\theta\|_1, \quad \text{where } \|\theta\|_1 = |\theta_1| + ... + |\theta_d|.$$

We know, that posterior is proportional to likelihood times prior, meaning we can express $J(\theta)$ as negative *log*-posterior using terms that involve only parameters under consideration,

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

The expression $(\vec{y} - X\theta)^T(\vec{y} - X\theta)$ here represents the squared Euclidean distance (or L_2 norm) between the vectors $\vec{y}$ and $X\theta$; this follows directly from results in [ps2q5b], where L_2 is defined. Thus, our cost function becomes,

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

To put $J(\theta)$ in desired form, we can absorb all constants into single term $\gamma = \frac{2\sigma^2}{b}$, this gives:

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

Remark: Linear regression with L2 regularization is also commonly called Ridge regression, and when L1 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., L1 regularization) is known to result in sparse parameters, where most of the parameter values are zero, with only some of them non-zero.