CS 229, Summer 2020
Problem Set #2 Solutions

YOUR NAME HERE (`YOUR SUNET HERE`)

Due Monday, July 27 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 `https://piazza.com/stanford/summer2020/cs229`. (3) This quarter, Summer 2020, students may submit in pairs. If you do so, make sure both names are attached to the Gradescope submission. However, students are not allowed to work with the same partner on more than one assignment. 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 27 at 11:59 pm. If you submit after Monday, July 27 at 11:59 pm, you will begin consuming your late days. If you wish to submit on time, submit before Monday, July 27 at 11:59 pm.

All students must submit an electronic PDF version of the written questions. We highly recommend typesetting your solutions via LaTeX, and we will award one bonus point for typeset submissions. 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: We see that for dataset A we quickly achieve convergence after 30374 iterations, whereas for dataset B it takes many more iterations to get θ to converge. For both the size of the gradient vector decreases over time, just much more slowly for dataset B.

```
==== Training model on data set A ====
Finished 10000 iterations
Theta: [-20.81394174  21.45250215  19.85155266]
Size of theta: 1287.5141623056304
Size of grad: 5.222218480921017e-11
Finished 20000 iterations
Theta: [-20.81437785  21.45295156  19.85198173]
Size of theta: 1287.5686341528296
Size of grad: 2.8439949007650234e-19
Finished 30000 iterations
Theta: [-20.81437788  21.45295159  19.85198176]
Size of theta: 1287.568638172836
Size of grad: 1.5326309220390799e-27
Converged in 30374 iterations
```

Compared with

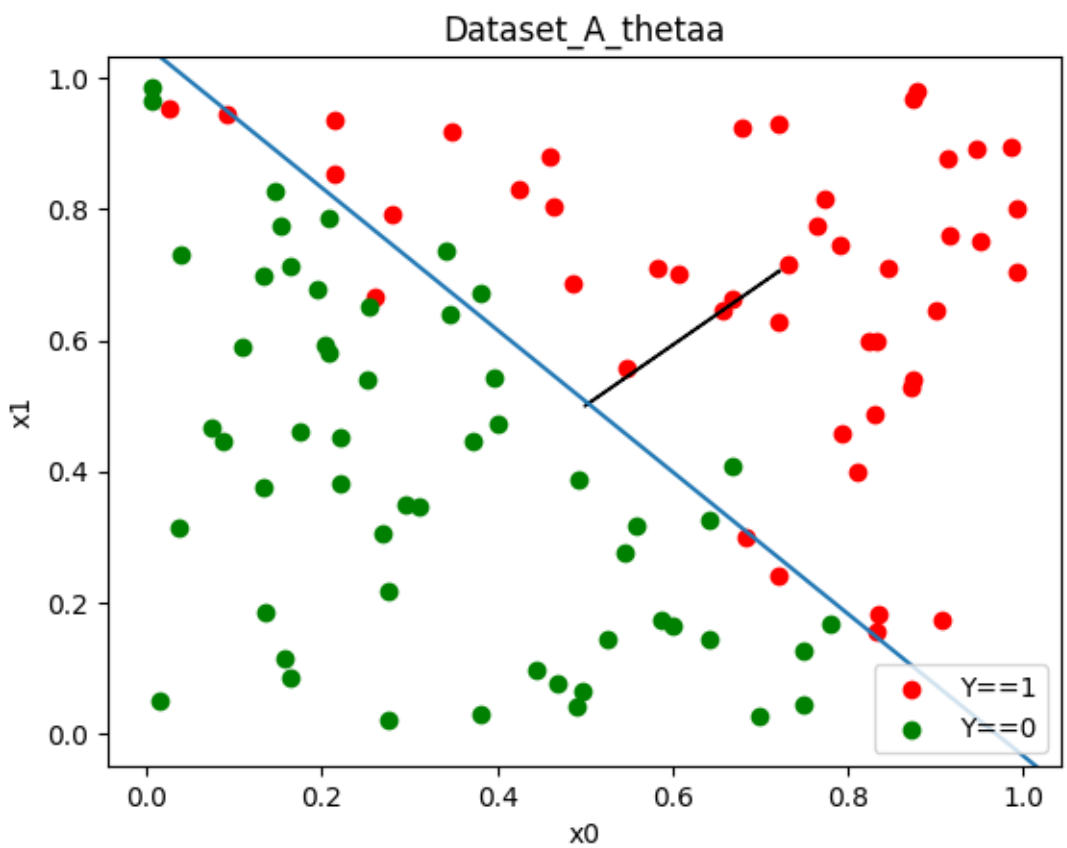
```
==== Training model on data set B ====
Finished 10000 iterations
Theta: [-52.74109217  52.92982273  52.69691453]
Size of theta: 8360.153738379526
Size of grad: 0.001129658631566177
Finished 20000 iterations
Theta: [-68.10040977  68.26496086  68.09888223]
Size of theta: 13935.228454051608
Size of grad: 0.00047228214977839664
Finished 30000 iterations
Theta: [-79.01759142  79.17745526  79.03755803]
Size of theta: 18759.78475534314
...
```

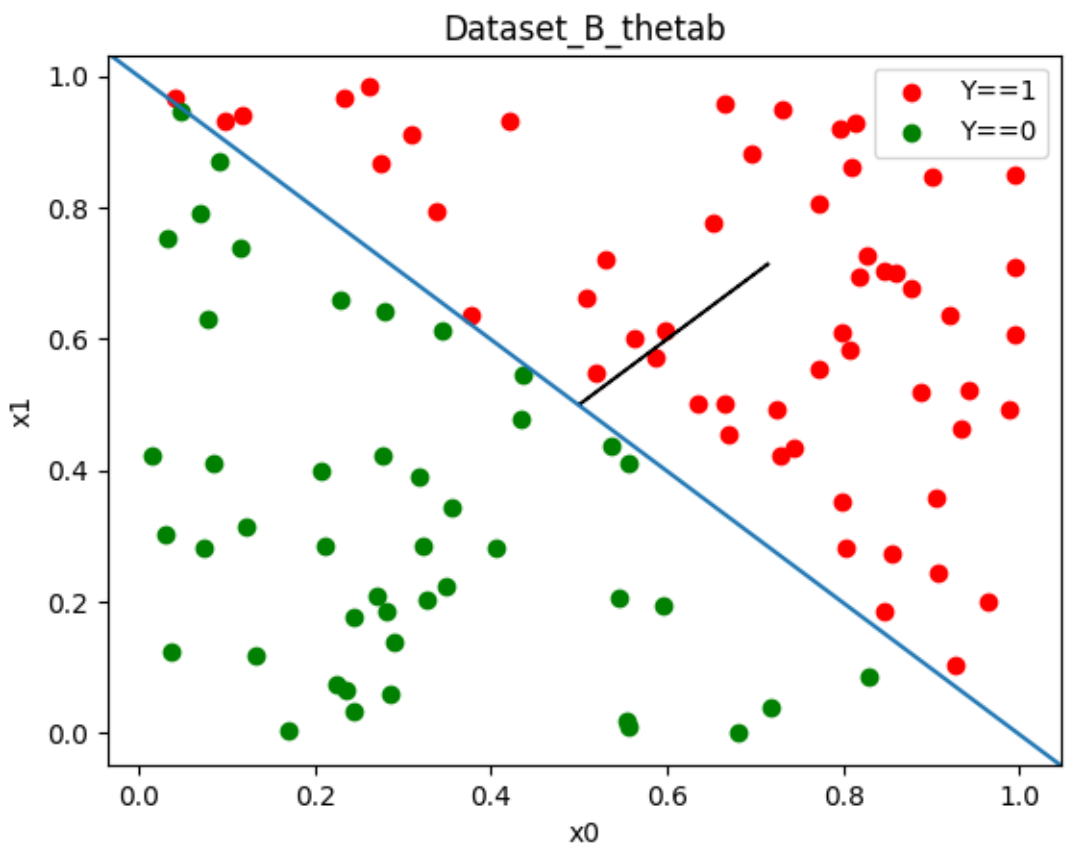
It appears that with dataset B the theta vector is unhelpfully just growing in size, not really changing its direction at all.

(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.

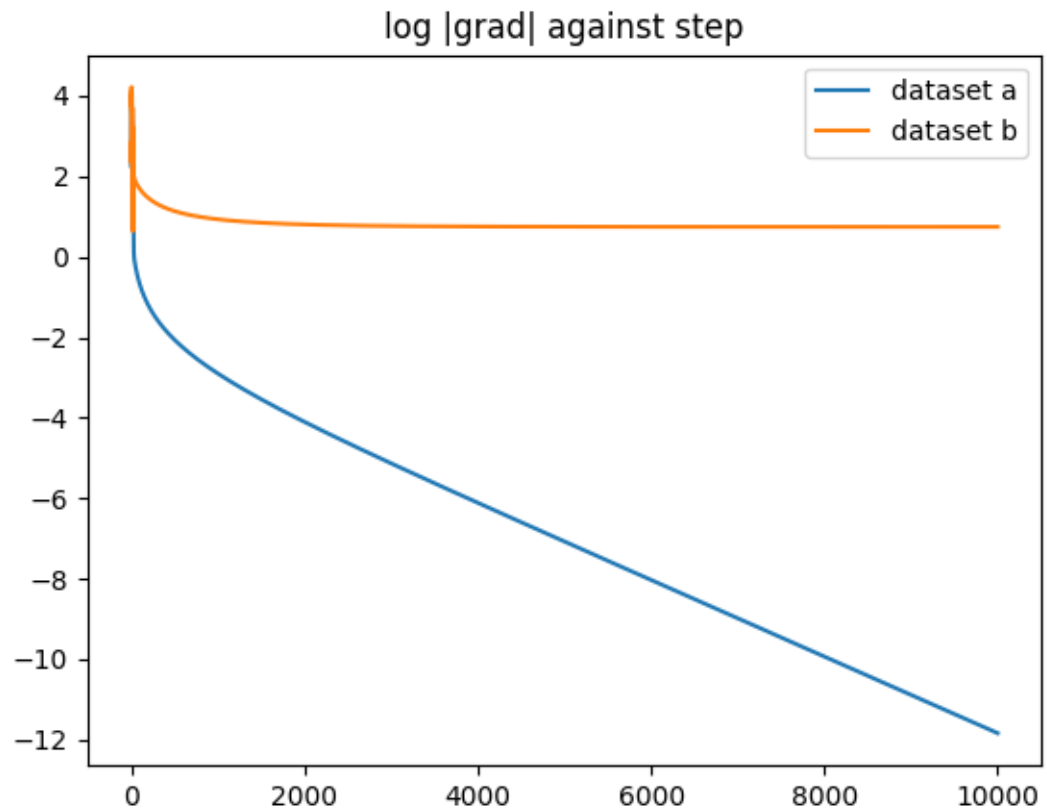
Answer:





We observe that dataset A seems less well differentiated than dataset B, the former has a few points mixing around the boundaries, whereas the latter can almost perfectly be separated by a dividing line. Otherwise the two are very comparable.

The classification line shown are derived many iterations on through gradient descent. For Dataset B we achieve perfect classification, whereas for Dataset A there are a few points that are misclassified, but this is unavoidable.



Our hypothesis is confirmed by this figure which shows the log size of the log likelihood gradient vector, for both they seem to eventually decrease exponentially quickly (linearly in log), but for B it's much shallower a slope than A. This would seem to be because perfect classification is possible with B, such that further improvements can only be made by increasing the magnitude of θ without changing direction. Indeed if $h_\theta(x) = g(z)$ then $h_{2\theta}(x) = g(2z)$. If classification is good, then $z >> 0$ or $z << 0$ s.t. writing $e^{-z} = y$, have $g(z) \approx 1 - y,\ g(2z) \approx 1 - y^2$ or $g(z) \approx y^{-1},\ g(2z) \approx y^{-2}$, i.e. the size $|y - g(z)| = \epsilon \mapsto \epsilon^2$.

Hence the grad step $\frac{\partial l}{\partial \theta} = \sum(y_i - h_\theta(x_i))x_i$ roughly gets smaller as we step. I.e. if $\theta \mapsto 2\theta$ then $\epsilon \mapsto \epsilon^2$. I.e. the size of $\frac{\partial l}{\partial \theta}$ is approximately proportional to $\epsilon^{2^k} = \epsilon^{M_k}$ where the parameter θ_n itself has size $2^k\theta$. The time of doubling, $n_k \propto \theta 2^k \epsilon^{-2^k} \propto M_k \epsilon^{-M_k}$, and so if we roughly map $y = |\text{grad}|$ against $x \approx n_k$, we get a relation like $x \approx\propto M_k/y \approx\propto 1/y$ since $y = \epsilon^{-M_k}$ shrinks much faster than M_k itself. Then $\log|\text{grad}|$ shrinks rather like $-\log x$, i.e. in the long run not like the linear decay we get by dataset A.

That was overly complicated and I'm not even sure if quite correct, but the essence is having no target to aim for size of the gradient slides decreases very slowly, like $1/$time step, while the parameter itself just gets bigger and bigger (though bigger at a very gradually slowing rate). Some intuition would be better here...

(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. a higher constant learning rate would normally make convergence happen faster, but here there is no real convergence point. In fact the doubling time $n_k \propto \theta/(\alpha\text{grad})$ would decrease as expected, but the theta diff to determine stopping point is also increased, i.e. waiting for αgrad small enough in size. I.e. overall the effect cancels out, given that $y \approx\propto 1/x$

ii. Decreasing the learning rate over time would help speed stopping of the algorithm, e.g. $\alpha = 1/t^2$. This will definitely squeeze the size of the steps to quickly be small enough.

iii. Linear scaling of the input features is unhelpful, it's akin to starting further or nearer along in the θ doubling process - doesn't remove the need for the increasingly lengthy doubling times

iv. A regularization term of $||\theta||_2^2$ would help, as it would force an optimum to exist within an acceptable bounds of size of parameter, to which we could converge to much more naturally.

v. Adding zero-mean gaussian noise to he training data or labels would be useful, as either now the points aren't all split in half, or doubling θ endlessly is no longer a good strategy (some maximum θ over which we're counter productive).

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

Answer:

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

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

In recent years, spam on electronic media has been a growing concern. Here, we'll build a classifier to distinguish between real messages, and spam messages. For this class, we will be building a classifier to detect SMS spam messages. We will be using an SMS spam dataset developed by Tiago A. Almedia and José María Gómez Hidalgo which is publicly available on `http://www.dt.fee.unicamp.br/~tiago/smsspamcollection` [1]

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

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

The provided code will then run your functions and save the resulting dictionary into `spam_dictionary` and a sample of the resulting training matrix into `spam_sample_train_matrix`.

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

Answer: I end up with a dictionary of 1678 words, most pretty weird such as

(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:

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

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

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

Complete the `get_top_five_naive_bayes_words` function within the provided code using the above formula in order to obtain the 5 most indicative tokens.

Report the top five words in your writeup.

Answer: Our top 5 spammiest words (by spam metric - ratio of prob of token given it is spam vs not spam) from highest score to lowest is

`['claim', 'prize', '150p', 'won', 'uk']`

(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:

3. **[18 points] Constructing kernels**

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

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

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

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

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

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

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

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

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

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

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

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

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

Answer:

(a) $K_{ij} = K'_{ij} + K''_{ij}$ is still symmetric, and still positive definite via $a^TKa = a^TK'a + a^TK''a$

(b) $K_ij = K'_ij - K''_ij$ is not necessarily symmetric semi-pos-def as e.g. $K'' = 2K'$ then $K = -K'$ is now symmetric semi-neg-def

(c) $K = aK'$ where $a > 0$ is clearly still a kernel

(d) $K = -aK'$ where $a > 0$ is clearly never a kernel

(e) $K(x, z) = K'(x, z)K''(x, z)$ i.e. $K_{ij} = K'_{ij}K''_{ij}$ (no summation) is a bit tricky. Consider that since K', K'' are both kernels, then e.g. $K'(x, z) = \alpha(x)^T\alpha(z)$ and $K''(x, z) = \beta(x)^T\beta(z)$

for some unknown dimensional maps α, β. Then writing α_i for $\alpha(x_i)$ for our arbitrary set of vectors under consideration $\{x_1, ..., x_n\}$ (that gives us the corresponding $n \times n$ matrix K_{ij}),

$$\begin{aligned} a^T K a &= \sum_{i,j} a_i K_{ij} a_j = \sum_{i,j} a_i (\alpha_i^T \alpha_j \times \beta_i^T \beta_j) a_j \\ &= \sum_{i,j} a_i (\sum_k \alpha_{ik}^T \alpha_{jk})(\sum_l \beta_{il} \beta_{jl}) a_j \\ &= \sum_{k,l} \sum_i \left[a_i \alpha_{ik} \beta_{il} \left(\sum_j a_j \alpha_{jk} \beta_{jl} \right) \right] \\ &= \sum_{k,l} \left(\sum_i a_i \alpha_{ik} \beta_{il} \right)^2 \geq 0 \end{aligned}$$

(f) $K(x, z) = f(x)f(z)$ for a real valued func $f : \mathbb{R}^d \to \mathbb{R}$, then $a^T K a = a_i f_i f_j a_j = (\sum_i a_i f_i)^2 \geq 0$

(g) $K(x, z) = K'(\phi(x), \phi(z))$ for some func $\phi : \mathbb{R}^d \to \mathbb{R}^p$ then still as above, $a^T K a = a_i \phi_i \phi_j a_j \geq 0$

(h) $K(x, z) = p(K'(x, z))$ for some polynomial p with positive coefficients, then we can use the previous parts of the question to construct K and still be a valid kernel, e.g. $K', K'K', K'K'K', ...$ are all valid kernels from part (e), and we can add positive scaled kernels together and still have a kernel by other parts.

4. **[15 points] Kernelizing the Perceptron**

Let there be a binary classification problem with $y \in \{0, 1\}$. The perceptron uses hypotheses of the form $h_\theta(x) = g(\theta^T x)$, where $g(z) = \text{sign}(z) = 1$ if $z \geq 0$, 0 otherwise. In this problem we will consider a stochastic gradient descent-like implementation of the perceptron algorithm where each update to the parameters θ is made using only one training example. However, unlike stochastic gradient descent, the perceptron algorithm will only make one pass through the entire training set. The update rule for this version of the perceptron algorithm is given by

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

where $\theta^{(i)}$ is the value of the parameters after the algorithm has seen the first i training examples. Prior to seeing any training examples, $\theta^{(0)}$ is initialized to $\vec{0}$.

(a) [3 points] Let K be a kernel corresponding to some very high-dimensional feature mapping ϕ. Suppose ϕ is so high-dimensional (say, ∞-dimensional) that it's infeasible to ever represent $\phi(x)$ explicitly. Describe how you would apply the "kernel trick" to the perceptron to make it work in the high-dimensional feature space ϕ, but without ever explicitly computing $\phi(x)$. [**Note:** You don't have to worry about the intercept term. If you like, think of ϕ as having the property that $\phi_0(x) = 1$ so that this is taken care of.] Your description should specify:

i. [1 points] How you will (implicitly) represent the high-dimensional parameter vector $\theta^{(i)}$, including how the initial value $\theta^{(0)} = 0$ is represented (note that $\theta^{(i)}$ is now a vector whose dimension is the same as the feature vectors $\phi(x)$);

ii. [1 points] How you will efficiently make a prediction on a new input $x^{(i+1)}$. I.e., how you will compute $h_{\theta^{(i)}}(x^{(i+1)}) = g({\theta^{(i)}}^T \phi(x^{(i+1)}))$, using your representation of $\theta^{(i)}$; and

iii. [1 points] How you will modify the update rule given above to perform an update to θ on a new training example $(x^{(i+1)}, y^{(i+1)})$; *i.e.,* using the update rule corresponding to the feature mapping ϕ:

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

Answer: In the notes we considered the LMS algorithm for fitting the model $h_\theta(x) = \theta^T x$ with update rule $\theta := \theta + \alpha \sum_{i=1}^n (y_i - h_\theta(x_i))x_i$. Here $x \in \mathbb{R}^d$, and in order to introduce a feature map $\phi(x) \in \mathbb{R}^p$ e.g. $(1, x_1, x_2, x_1^2, x_1x_2, ...)$ etc., we had then $\theta := \theta + \alpha \sum_{i=1}^n (y_i - \theta^T \phi(x_i))\phi(x_i)$.

Applying the Kernel trick in this scenario to deal with high/infinite dimensional feature involved taking $\theta^{(0)} = \sum_{i=1}^n \beta_i^{(0)} \phi(x_i)$ and observing that the update becomes $\beta := \beta + \alpha(y - K\beta)$ where K is the matrix $K_{ij} =< \phi(x_i), \phi(x_j) >$, and we've "replaced" θ with a new parametrisation of the n-vector β.

Then resultant prediction $\theta^T \phi(x) = (\sum_j \beta_j \phi(x_j))^T \phi(x) = \sum_j \beta_j K(x_j, x)$. The important is that we've replaced keeping track of high dimensional $\phi(x)$ by instead computing easier $K(x, z) = \phi(x)^T \phi(z)$ which generally should be easier to compute.

In the perceptron learning algorithm we have a slightly different setup: $h_\theta(x) = \text{sgn}(\theta^T x)$, and with update rule $\theta^{(i+1)} := \theta^{(i)} + \alpha(y_{i+1} - h_{\theta^{(i)}}(x_{i+1}))x_{i+1}$. For some reason we make an update based on just one training example, and make just one pass through the training set. $\theta^{(0)} = 0$. The higher-dimensional kernel analogue is then taking at the i+1th step, $\theta^{(i+1)} := \theta^{(i)} + \alpha(y_{i+1} - h_{\theta^{(i)}}(\phi(x_{i+1})))\phi(x_{i+1})$, and so $\beta_{i+1}^{(i+1)} := \beta_{i+1}^{(i)} + \alpha(y_{i+1} - \text{sgn}(\sum_j K_{i+1,j}\beta_j^{(i)}))$.

I.e. at each step we oddly only update one index of β. so $\beta^0 = 0$. Then $\beta_1^1 = \alpha y_1$. Then $\beta_2^2 = \alpha(y_2 + \sum_j \beta_j^1 K_{2,j}) = \alpha(y_2 + \beta_1^1 K_{2,1})$, and so on...

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

We provide three functions to be used as kernel, a dot-product kernel defined as:

$$K(x, z) = x^\top z, \tag{1}$$

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

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

and finally the following function:

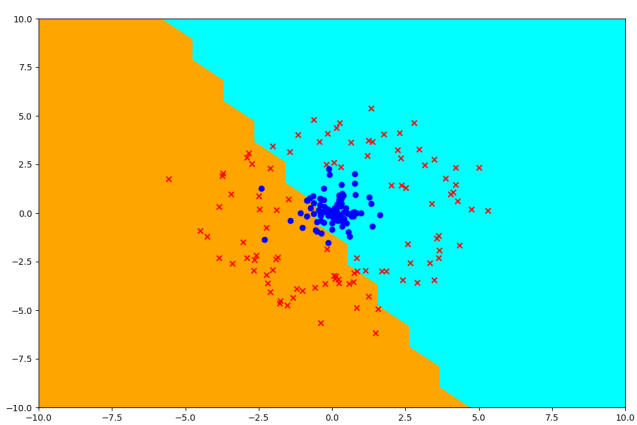
$$K(x, z) = \begin{cases} -1 & x = z \\ 0 & x \neq z \end{cases} \tag{3}$$

Note that the last function is not a kernel function (since its corresponding matrix is not a PSD matrix). However, we are still interested to see what happens when the kernel is invalid. Run `src/perceptron/perceptron.py` to train kernelized perceptrons on `src/perceptron/train.csv`. The code will then test the perceptron on `src/perceptron/test.csv` and save the resulting predictions in the `src/perceptron/` folder. Plots will also be saved in `src/perceptron/`.

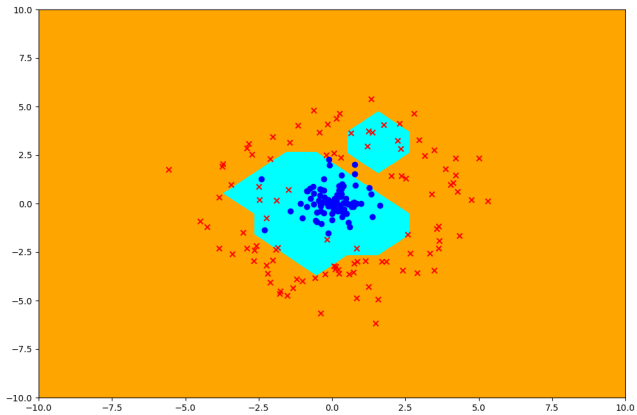
Include the three plots (corresponding to each of the kernels) in your writeup, and indicate which plot belongs to which function.

Answer: Plots for test set predictions of different kernel perceptron predictions:

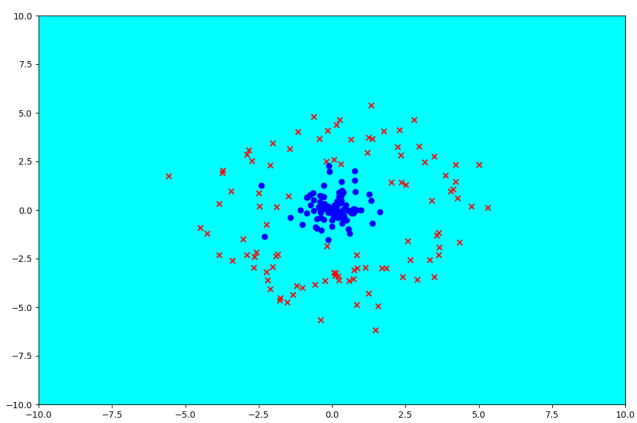
dot kernel predictions



rbf kernel predictions



non psd predictions



(c) [2 points] One of the choices in Q4b completely fails, one works a bit, and one works well in classifying the points. Discuss the performance of different choices and why do they fail or perform well?

Answer: The perceptron training algorithm of $\theta \mapsto \theta + \alpha(y_i - g(\theta^T \phi(x_i)))\phi(x_i)$ is an odd data point wise stochastic gradient ascent of maximising the overall likelihood of a logistic classifier $g(z) = 1/(1 + e^{-z})$ for a certain feature map and parameter vector $z = \theta^T \phi(x)$. Using the kernel we find a way to better represent high dimensional feature maps, replacing $\theta = \sum_j \beta_j \phi(x_j)$. So with the dot product kernel $K(x, z) = x^T z$, this is taking $\phi(x) = x$, so since x is a 2-d vector here we're just training a line classifier (without even an intercept term). The result is actually not bad as you can see by the graph, it's pretty much the best you can do with a line.

The gaussian kernel classifier is great as expected being an infinite dimensional kernel of so many combinations.

The non psd kernel is weird, just computationally you can see that $K_{i+1,j} = 0$ always, s.t. $\beta_i = \alpha(y_i - 1)$ is essentially random choice of $-0.5, 0$. Then prediction $g(\sum_j \beta_j K(x_j, x)) = 0$ since $K(x_j, x) = 0$ always (as $x_j \neq x$) generally. Another observation is that $K_{ij} = -I$ negative identity matrix, not sure how it helps though :(.

5. **[30 points] Neural Networks: MNIST image classification**

In this problem, you will implement a simple neural network to classify grayscale images of handwritten digits (0 - 9) from the MNIST dataset. The dataset contains 60,000 training images and 10,000 testing images of handwritten digits, 0 - 9. Each image is 28×28 pixels in size, and is generally represented as a flat vector of 784 numbers. It also includes labels for each example, a number indicating the actual digit (0 - 9) handwritten in that image. A sample of a few such images are shown below.

The data and starter code for this problem can be found in

- `src/mnist/nn.py`
- `src/mnist/images_train.csv`
- `src/mnist/labels_train.csv`
- `src/mnist/images_test.csv`
- `src/mnist/labels_test.csv`

The starter code splits the set of 60,000 training images and labels into a set of 50,000 examples as the training set, and 10,000 examples for dev set.

To start, you will implement a neural network with a single hidden layer and cross entropy loss, and train it with the provided data set. Use the sigmoid function as activation for the hidden layer, and softmax function for the output layer. Recall that for a single example (x, y), the cross entropy loss is:

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

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

For clarity, we provide the forward propagation equations below for the neural network with a single hidden layer. We have labeled data $(x^{(i)}, y^{(i)})_{i=1}^n$, where $x^{(i)} \in \mathbb{R}^d$, and $y^{(i)} \in \mathbb{R}^K$ is a

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

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

where σ is the sigmoid function.

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

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

The starter code already converts labels into one hot representations for you.

Instead of batch gradient descent or stochastic gradient descent, the common practice is to use mini-batch gradient descent for deep learning tasks. In this case, the cost function is defined as follows:

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

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

(a) **[5 points]**
For a single input example $x^{(i)}$ with one-hot label vector $y^{(i)}$, show that

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

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

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

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

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

Answer: To find $\frac{\partial}{\partial z_j}(-\sum y_k \log \hat{y}_k) = (*)$ we note that $\hat{y}_k = \text{softmax}(\hat{y})_k = \frac{e^{z_k}}{\sum_i e^{z_i}}$, and so $\frac{\partial}{\partial z_j}\hat{y}_k = \frac{e^{z_k}(\sum_i e^{z_i})\delta_{jk} - e^{z_j}e^{z_k}}{(\sum_i e^{z_i})^2} = \hat{y}_k\delta_{jk} - \hat{y}_j\hat{y}_k$. So then $(*) = -\sum_k (y_k/\hat{y}_k)(\hat{y}_k\delta_{jk} - \hat{y}_j\hat{y}_k) = \hat{y}_j - y_j$.

I guess this sort of makes sense. In gradient descent we'll try to decrease cross entropy loss, i.e. stepping along $-\nabla_z = y - \hat{y}$, i.e. seeking to make the final pre-softmax layer increase/decrease s.t. $\hat{y}$ is more like y

(b) **[15 points]**

Implement both forward-propagation and back-propagation for the above loss function $J_{MB} = \frac{1}{B}\sum_{i=1}^{B} CE(y^{(i)}, \hat{y}^{(i)})$. Initialize the weights of the network by sampling values from a standard normal distribution. Initialize the bias/intercept term to 0. Set the number of hidden units to be 300, and learning rate to be 5. Set $B = 1,000$ (mini batch size). This means that we train with 1,000 examples in each iteration. Therefore, for each epoch, we need 50 iterations to cover the entire training data. The images are pre-shuffled. So you don't need to randomly sample the data, and can just create mini-batches sequentially.

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

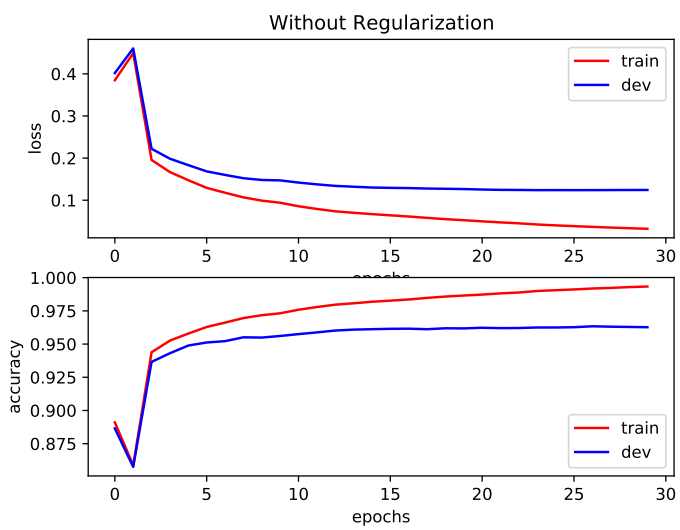
Similarly, in a new image, plot the accuracy (on y-axis) over the training set, measured as the fraction of correctly classified examples, versus the number of epochs (x-axis). In the same image, also plot the accuracy over the dev set versus number of epochs.

Submit the two plots (one for loss vs epoch, another for accuracy vs epoch) in your writeup.

Also, at the end of 30 epochs, save the learnt parameters (i.e all the weights and biases) into a file, so that next time you can directly initialize the parameters with these values from the file, rather than re-training all over. You do NOT need to submit these parameters.

Hint: Be sure to vectorize your code as much as possible! Training can be very slow otherwise.

Answer:



We see that as expected with each epoch the train and dev loss decreases and the accuracy increases. Sharpest initially, but plateauing towards the end. The dev set is slightly worse, especially at later epochs in both metrics, and plateaus off more sharply.

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

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

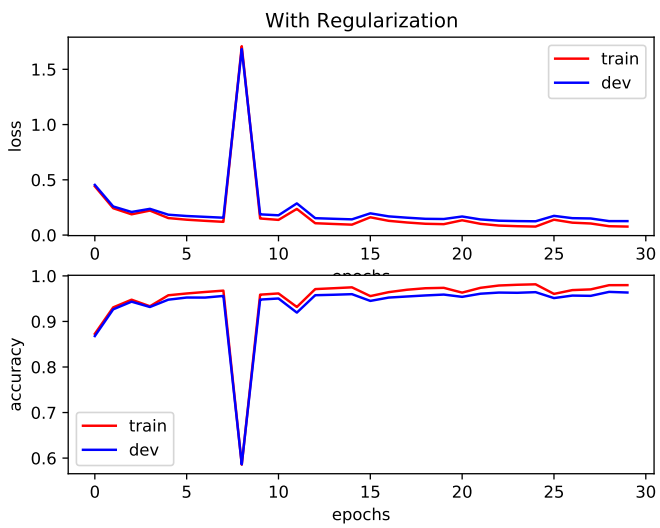
Be careful not to regularize the bias/intercept term. Set λ to be 0.0001. Implement the regularized version and plot the same figures as part (a). Be careful NOT to include the regularization term to measure the loss value for plotting (i.e., regularization should only be used for gradient calculation for the purpose of training).

Submit the two new plots obtained with regularized training (i.e loss (without regularization term) vs epoch, and accuracy vs epoch) in your writeup.

Compare the plots obtained from the regularized model with the plots obtained from the non-regularized model, and summarize your observations in a couple of sentences.

As in the previous part, save the learnt parameters (weights and biases) into a different file so that we can initialize from them next time.

Answer:



We see that as expected with each epoch the train and dev loss decreases and the accuracy increases. Sharpest initially, but plateauing towards the end. The dev set is slightly worse, especially at later epochs in both metrics, and plateaus off more sharply.

(d) **[3 points]** All this while you should have stayed away from the test data completely. Now that you have convinced yourself that the model is working as expected (i.e, the observations you made in the previous part matches what you learnt in class about regularization), it is finally time to measure the model performance on the test set. Once we measure the test set performance, we report it (whatever value it may be), and NOT go back and refine the model any further.

Initialize your model from the parameters saved in part (a) (i.e, the non-regularized model), and evaluate the model performance on the test data. Repeat this using the parameters saved in part (b) (i.e, the regularized model).

Report your test accuracy for both regularized model and non-regularized model. Briefly (in one sentence) explain why this outcome makes sense" You should have accuracy close to 0.92870 without regularization, and 0.96760 with regularization. Note: these accuracies assume you implement the code with the matrix dimensions as specified in the comments, which is not the same way as specified in your code. Even if you do not precisely these numbers, you should observe good accuracy and better test accuracy with regularization.

Answer:

```
>>> test_labels = one_hot_labels(test_labels)
>>> mean = np.mean(test_data)
>>> std = np.std(test_data)
>>> test_data = (test_data - mean) / std
>>> accuracy_base_test = nn_test(test_data, test_labels, base['params'])
>>> accuracy_reg_test = nn_test(test_data, test_labels, reg['params'])
>>> accuracy_base_test
0.9691
>>> accuracy_reg_test
0.9647
```

This doesn't quite match the expected 0.982870 and 0.96760, not sure what's up there... I did run the training on smaller batch sizes to suit my memory but when training on larger I still didn't seem to have much difference between with/without 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). You will also explore how regularization strengths affect generalization in part (d).

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

Answer:

$$\theta_{\mathsf{MAP}} = \mathsf{argmax}_\theta\, p(\theta|x, y) = \mathsf{argmax}_\theta\, \frac{p(\theta, x, y)}{p(x, y)} = \mathsf{argmax}_\theta\, \frac{p(\theta, x, y)}{p(x, \theta)} \frac{p(x, \theta)}{p(x, y)}$$
$$= \mathsf{argmax}_\theta\, p(y|x, \theta) \frac{p(x, \theta)}{p(x)} \frac{p(x)}{p(x, y)} = \mathsf{argmax}_\theta\, p(y|x, \theta)p(x)$$

Where at the last step we have used the assumption that $p(\theta) = p(\theta|x)$ (i.e. θ is independent from x (if no knowledge of y)), and that we can drop the terms with no θ since we're argmaxing of θ

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

$$\theta_{\text{MAP}} = \arg\min_{\theta} -\log p(y|x,\theta) + \lambda||\theta||_2^2. \tag{4}$$

Also, what is the value of λ?

Answer: We'll convert our MLE with L_2 regularization to the form of θ_{MAP} by doing $\text{argmin}_\theta - \log p(y|x,\theta) + \lambda||\theta||_2^2 = \text{argmax}_\theta p(y|x,\theta)\ \exp(-\lambda||\theta||_2^2)$.
Then the problem is really simplified, as by (a) we have

$$\theta_{MAP} = \text{argmax}_\theta p(y|x,\theta)p(\theta) =$$
$$\text{argmax}_\theta p(y|x,\theta)(2\pi)^{-k/2}|\Sigma|^{-1/2}\exp(-\frac{\theta^T\Sigma^{-1}\theta}{2})$$
$$\text{i.e. we're done by the prior of } \theta \sim N(0,\eta^2 I);\ \Sigma = \eta^{-2}I$$
$$\text{so then } \lambda = \frac{\eta^{-2}}{2}$$

(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:

$$\begin{aligned}\theta_{MAP} &= \text{argmin}_\theta - \log p(y|x,\theta) + \lambda||\theta||_2^2 \\ &= \text{argmin}_\theta - \log\left(\exp(-\frac{(y-X\theta)^T\Sigma(y-X\theta)}{2})\right) + \lambda||\theta||_2^2 \\ &= \text{argmin}_\theta \frac{\sigma^{-2}}{2}||y-X\theta||_2^2 + \lambda||\theta||_2^2\end{aligned}$$

We take ∇_θ of the RHS and set to zero to find the minimal θ,

$$\begin{aligned}\frac{\sigma^{-2}}{2}(-2X^Ty + 2X^TX\theta) + 2\lambda\theta &= 0 \\ (\sigma^{-2}X^TX + 2\lambda I)\,\theta &= \sigma^{-2}X^Ty \\ \theta &= (\sigma^{-2}X^TX + 2\lambda I)^{-1}\,\sigma^{-2}X^Ty\end{aligned}$$

(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:

$$\begin{aligned}
\theta_{MAP} &= \text{argmax}_\theta \ \log p(y|x,\theta) + \log p(\theta) \quad \text{using fact that LR model has } p(\theta|x) = p(\theta) \\
&= \text{argmax}_\theta \ \log\left(\exp(-\frac{(y-X\theta)^T\Sigma(y-X\theta)}{2})\right) + \log(\prod_i \frac{1}{2b}\exp\left(\frac{-|\theta_i|}{b}\right)) \\
&= \text{argmin}_\theta \ \frac{\sigma^{-2}}{2}||y - X\theta||_2^2 + \frac{1}{b}||\theta||_1
\end{aligned}$$

Hence finding θ_{MAP} is equivalent to solving the linear regression problem with L_1 regularization, i.e. minimising the loss $J(\theta) = ||y - X\theta||_2^2 + \gamma||\theta||_1$, where here $\gamma = \frac{2\sigma^2}{b}$

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.