CS 229, Spring 2020
Problem Set #2 Solutions

Naoufal Layad (`06387243`) & Youssef Aitousarrah (`06363218`)

Due Wednesday, May 13 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/spring2020/cs229`. (3) This quarter, Spring 2020, students may submit in pairs. 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 Wednesday, May 13 at 11:59 pm. If you submit after Wednesday, May 13 at 11:59 pm, you will begin consuming your late days. If you wish to submit on time, submit before Wednesday, May 13 at 11:59 pm.

All students must submit an electronic PDF version of the written questions. We highly recommend typesetting your solutions via LaTeX. All students must also submit a zip file of their source code to Gradescope, which should be created using the `make_zip.py` script. You should make sure to (1) restrict yourself to only using libraries included in the `environment.yml` file, and (2) make sure your code runs without errors. Your submission may be evaluated by the auto-grader using a private test set, or used for verifying the outputs reported in the writeup.

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

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

We have provided an implementation of logistic regression in `src/stability/stability.py`, and two labeled datasets A and B in `src/stability/ds1_a.csv` and `src/stability/ds1_b.csv`.

Please do not modify the code for the logistic regression training algorithm for this problem. First, run the given logistic regression code to train two different models on A and B. You can run the code by simply executing `python stability.py` in the `src/stability` directory.

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

Answer: We notice that dataset A converges after a finite number of iterations (30410) but dataset B doesn't converge at all.

(b) [5 points] Investigate why the training procedure behaves unexpectedly on dataset B, but not on A. Provide hard evidence (in the form of math, code, plots, etc.) to corroborate your hypothesis for the misbehavior. Remember, you should address why your explanation does *not* apply to A.

Hint: The issue is not a numerical rounding or over/underflow error.

Answer: If we look at the data we can see that for dataset A the classes are not perfectly separable, which is the reason why the gradient descent converges. But, for dataset B the two classes are perfectly separable, this leads to the gradient descent not being able to converge because minimizing the cost function can only be done by increasing θ infinitely as we can see if we plot its value after each iteration. If we want to understand this mathematically, we can write the cost function:

$$J(\theta) = -\frac{1}{n}\sum_{i=1}^{n} y^{(i)} \log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)})) \tag{1}$$

Let's say the we found a θ that is capable of perfectly separating the examples so that:
For $y^{(i)} = 1$ we have $\theta^T x^{(i)} > 0$, and for $y^{(i)} = 0$ we have $\theta^T x^{(i)} < 0$.
This means that to minimize $J(\theta)$ we have to push $\theta^T x^{(i)}$ further towards $+\infty$ for a positive example and towards $-\infty$ for a negative one. This means that theta will keep increasing and we will not be able to minimize the cost function.

We have used a trick to see if this is the right explanation. What we did was to add a new data point to the dataset B. We added a class 0 point to domain where we have class 1 and consequently we obtained a convergent model.

We plotted also the dataset A with the boundary decision separating the classes. We also plotted the boundary decision with different thetas for dataset B (we took three theta elements after 10 000 iteration each time) and we can see that they are very close from each other. Therefore, even if we find a decision boundary that separates the classes, theta keep changing because the cost function cannot be optimized for a finite theta.

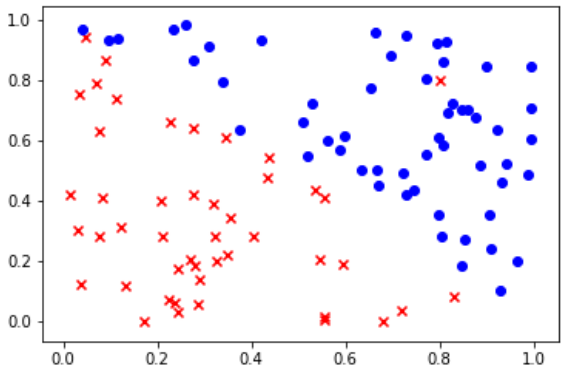

Figure 1: Adding a new data point to the dataset B to achieve convergence

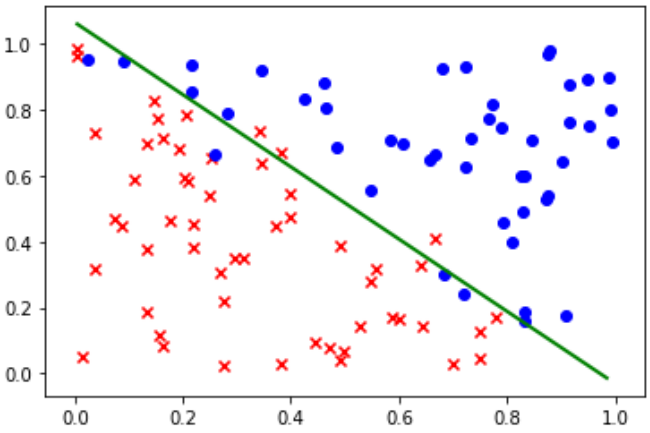

Figure 2: Decision boundary separating the classes for dataset A.

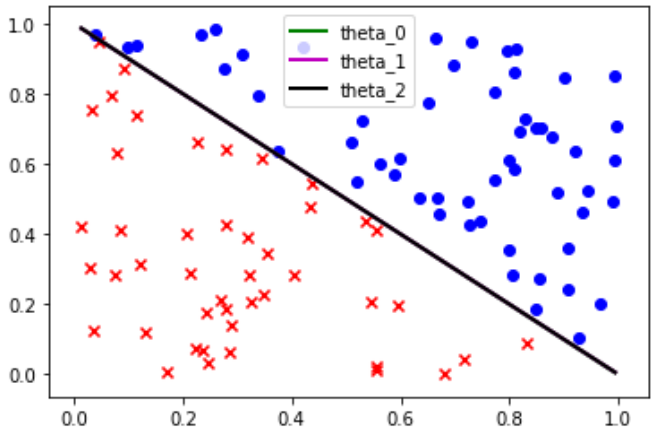

Figure 3: Decision boundary separating the classes for different thetas, for dataset B.

(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) and ii) will not fix the problem because failing to convergence is due to the absence of a minima (We need to go to ∞ to optimize the function). Therefore changing the learning rate will not help fix the problem.

iii) It is not going to work because a linear scaling of the input features will keep them perfectly separable and this is the reason we are not converging.

iv) This option will work. Intuitively by adding a regularization effect we prevent the θ from becoming too large. Adding $\lambda\|\theta\|_2^2$ to the cost function will make it optimizable at finite θ and not at ∞.

v) This will move the data points a little bit and if the variance is enough we can obtain a data set that is not perfectly separable which will solve the convergence problem. We tested this option by adding random normal noise with a standard deviation of 0.03 and we obtained a convergence.

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

Answer: This dataset will not cause any problems for the SVM model because it is exactly what is needed for SVM to work well. SVM needs inputs to be perfectly separable in order to work and it works by separating the classes by a gap. SVM works effectively when the data set is linearly separable.

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: After the pre-processing step, we obtain a dictionary with a size 1721 (not taking into account the word if it got repeated more than once in the same message). If we opt for the other solution were we account for words even if they are repeated in the same message we get a vocabulary size of 1757. You will find the pre-processing functions in the attached spam.py file.

(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: After building a naive Bayes classifier for spam classification with **multinomial event model and Laplace smoothing**, we obtain an accuracy of 0.978494623655914 on the testing set.

[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.

(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:

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

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 five most indicative tokens gotten using the above formula are: ['claim', 'won', 'prize', 'tone', 'urgent!']

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

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

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

Answer: The best kernel radius we obtained was 10. This SVM model had an accuracy of 0.8799283154121864 on the testing set. This accuracy is lower than the one obtained using the naive bayes model.

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) We know that K_1 and K_2 are kernels, so using Mercer's theorem, for any finite set $(x^1, ..., x^n)$ the square matrix K given by $K_{ij} = K(x^{(i)}, x^{(j)})$ is symmetric and positive semidefinite.

First, we know that the sum of two symmetric matrices is symmetric. So $K = K_1 + K_2$ is symmetric.

Furthermore, $\forall Y \in \mathbb{R}^n$, $0 \leq Y^T K_1 Y$ and $0 \leq Y^T K_2 Y$, so by summing these two quantities we obtain $\forall Y \in \mathbb{R}^n$, $0 \leq Y^T K_1 Y + Y^T K_2 Y = Y^T(K_1 + K_2)Y = Y^T KY$. Therefore, $K = K_1 + K_2$ is positive semidefinite, and the Mercer's theorem implies that $K = K_1 + K_2$ is a kernel.

We will answer (c) and (d) before (b)

(c) $K = aK_1$ where a is positive, we know that K_1 is a kernel, so for any finite set $(x^1, ..., x^n)$ the square matrix K_1 given by $K_{1ij} = K_1(x^{(i)}, x^{(j)})$ is symmetric and positive semidefinite. First,

$K = aK_1$ still a symmetric matrix. Moreover, $\forall Y \in \mathbb{R}^n$, $0 \leq Y^T K_1 Y$ and since a is positive so $\forall Y \in \mathbb{R}^n$, $0 \leq Y^T a K_1 Y = Y^T K Y$. Thus, K is positive semidefinite. So the Mercer's theorem implies that $K = aK_1$ is a kernel.

(d) $K = -aK_1$ where a is positive, we know that K_1 is a kernel, so for any finite set $(x^1, ..., x^n)$ the square matrix K_1 given by $K_{1ij} = K_1(x^{(i)}, x^{(j)})$ is symmetric and positive semidefinite. First, $K = -aK_1$ still a symmetric matrix. Moreover, $\forall Y \in \mathbb{R}^n$, $0 \leq Y^T K_1 Y$ and since a is positive so $\forall Y \in \mathbb{R}^n$, $Y^T K Y = Y^T(-aK_1)Y \leq 0$. Thus, K is negative semidefinite. So, $K = -aK_1$ is not kernel.

(b) $K = K_1 - K_2$ is not necessarily a kernel, if we choose K_1 an arbitrary kernel and $K_2 = 2K_1$, since K_1 is a kernel and 2 is positive so (c) implies that $K_2 = 2K_1$ is a kernel. Therefore, $K = K_1 - K_2 = -K_1$. Using (d) we deduce that $K = K_1 - K_2$ is not a kernel since -1 is negative.

(e) We know that K_1 and K_2 are kernels, we can write them as $K_1(x, z) = \phi_1(x)^T \phi_1(z)$ and $K_2(x, z) = \phi_2(x)^T \phi_2(z)$. We take a finite set $(x^1, ..., x^n)$, so the square matrices K_1 given by $K_{1ij} = K_1(x^{(i)}, x^{(j)}) = \phi_1(x^{(i)})^T \phi_1(x^{(j)})$ and $K_{1ij} = K_2(x^{(i)}, x^{(j)}) = \phi_2(x^{(i)})^T \phi_2(x^{(j)})$ are symmetric positive semidefinite.

For $K = K_1 K_2$ and $Y \in \mathbb{R}^n$, we have

$$Y^T K Y = Y^T K_1 K_2 Y = \sum_{k=1}^{k=n} \sum_{r=1}^{r=n} Y_k K_{1kr} K_{2kr} Y_r$$

So

$$Y^T K Y = \sum_{k=1}^{k=n} \sum_{r=1}^{r=n} Y_k K_1(x^{(k)}, x^{(r)}) K_2(x^{(k)}, x^{(r)}) Y_r,$$

then

$$Y^T K Y = \sum_{k=1}^{k=n} \sum_{r=1}^{r=n} Y_k \phi_1(x^{(k)})^T \phi_1(x^{(r)}) \phi_2(x^{(k)})^T \phi_2(x^{(r)}) Y_r,$$

therefore,

$$Y^T K Y = \sum_{k=1}^{k=n} \sum_{r=1}^{r=n} Y_k \sum_{i=1}^{i=d} \phi_1(x_i^{(k)}) \phi_1(x_i^{(r)}) \sum_{j=1}^{j=d} \phi_2(x_j^{(k)}) \phi_2(x_j^{(r)}) Y_r$$

By inverting the order of the sums, we obtain

$$Y^T K Y = \sum_{i=1}^{i=d} \sum_{j=1}^{j=d} \sum_{k=1}^{k=n} \sum_{r=1}^{r=n} Y_k \phi_1(x_i^{(k)}) \phi_1(x_i^{(r)}) \phi_2(x_j^{(k)}) \phi_2(x_j^{(r)}) Y_r.$$

So

$$Y^T K Y = \sum_{i=1}^{i=d} \sum_{j=1}^{j=d} \left(\sum_{k=1}^{k=n} Y_k \phi_1(x_i^{(k)}) \phi_2(x_j^{(k)}) \right)^2 \geq 0.$$

We conclude that $K = K_1 K_2$ is a kernel.

(f) We consider $K(x,z) = f(x)f(z)$. For a a finite set $(x^1, ..., x^n)$, the square matrix K given by $K_{ij} = K(x^{(i)}, x^{(j)}) = f(x^{(i)})f(x^{(j)})$.

For $Y \in \mathbb{R}^n$, we have

$$Y^T KY = \sum_{k=1}^{k=n} \sum_{r=1}^{r=n} Y_k K_{kr} Y_r = \sum_{k=1}^{k=n} \sum_{r=1}^{r=n} Y_k f(x^{(k)}) f(x^{(r)}) Y_r = \left(\sum_{k=1}^{k=n} Y_k f(x^{(k)}) \right)^2 \geq 0.$$

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

(g) We consider $K(x,z) = K_3(\phi(x), \phi(z))$. For a a finite set $(x^1, ..., x^n)$ of $\mathbb{R}^d$, the square matrix K given by $K_{ij} = K(x^{(i)}, x^{(j)}) = K_3(\phi(x^{(i)}), \phi(x^{(j)}))$.

For $Y \in \mathbb{R}^n$, we have

$$Y^T KY = \sum_{k=1}^{k=n} \sum_{r=1}^{r=n} Y_k K_{kr} Y_r = \sum_{k=1}^{k=n} \sum_{r=1}^{r=n} Y_k K_3(\phi(x^{(k)}), \phi(x^{(r)})) Y_r$$

We know that K_3 is a kernel, so for the finite set $(\phi(x^1), ..., \phi(x^n))$ of $\mathbb{R}^p$, the square matrix $K^{\phi}_{kr} = K_3(\phi(x^{(k)}), \phi(x^{(r)}))$ is symmetric positive semidefinite. Thus,

$$Y^T KY = \sum_{k=1}^{k=n} \sum_{r=1}^{r=n} Y_k K_{kr} Y_r = \sum_{k=1}^{k=n} \sum_{r=1}^{r=n} Y_k K^{\phi}_{kr} Y_r \geq 0.$$

Thus $K(x,z) = K_3(\phi(x), \phi(z))$ is a kernel.

(h) Let $p(x)$ a polynomial over x with positive coefficients. We can write $\forall x \in \mathbb{R}\ p(x) = \sum_{k=0}^{k=n} a_k x^k$ with $\forall k a_k \geq 0$. We have proved in (2) that the product $K = K_1 K_2$ is a kernel, so we can prove by an trivial induction that $\forall k\ K(x,z) = K_1(x,z)^k$ is a kernel. And in (c), we have that for $a_k \geq 0$ $a_k K_1(x,z)^k$. Now using (a) and a trivial induction, we obtain that $p(K_1(x,z)) = \sum_{k=0}^{k=n} a_k K_1(x,z)^k$ is a kernel, as a sum of kernels.

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

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

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

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

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

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

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

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

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

Answer: i. In order to avoid representing $\theta^{(i)}$ with as much elements as the dimension of $\phi(x)$ which can be very large, we can write it as the linear combination of the $\phi(x^{(i)})$. Therefore, we can write $\theta^{(0)} = 0.\phi(x^{(0)})$. The subsequent $\theta^{(i)}$ can be written as: $\theta^{(i)} = \sum_{j=0}^{i} \beta_j^{(i)} \phi(x^{(i)})$. This can be easily proven by induction.

ii. We can write:

$$\begin{aligned} h_{\theta^{(i)}}(x^{(i+1)}) &= g((\theta^{(i)})^T \phi(x^{(i+1)})) \\ &= g((\sum_{j=0}^{i} \beta_j^{(i)} \phi(x^{(j)}))^T \phi(x^{(i+1)})) \\ &= g((\sum_{j=0}^{i} \beta_j^{(i)} \phi(x^{(j)})^T) \phi(x^{(i+1)})) \\ &= g(\sum_{j=0}^{i} \beta_j^{(i)} \phi(x^{(j)})^T \phi(x^{(i+1)})) \\ &= g(\sum_{j=0}^{i} \beta_j^{(i)} K(x^{(j)}, x^{(i+1)})) \end{aligned} \tag{2}$$

By using the Kernel trick we will avoid representing $\phi(x)$ as it can be very high-dimensional, and we will use the Kernel to compute $\phi(x^{(i)})^T\phi(x^{(j)})$

iii. We can make use of the representation of $\theta^{(i)}$ to update the $\beta^{(i)}$ instead. We can write:

$$\begin{aligned}\theta^{(i+1)} &= \theta^{(i)} + \alpha(y^{(i+1)} - h_{\theta^{(i)}}(x^{(i+1)}))\phi(x^{(i+1)}) \\ &= \sum_{j=0}^{i} \beta_j^{(i)}\phi(x^{(j)}) + \alpha(y^{(i+1)} - h_{\theta^{(i)}}(x^{(i+1)}))\phi(x^{(i+1)}) \\ &= \sum_{j=0}^{i+1} \beta_j^{(i+1)}\phi(x^{(j)})\end{aligned} \tag{3}$$

Therefore, the update rule is as follows: $\beta_j^{(i+1)} = \beta_j^{(i)}$ for $j = 1, .., i$, and
$\beta_{i+1}^{(i+1)} = \alpha(y^{(i+1)} - h_{\theta^{(i)}}(x^{(i+1)})) = \alpha(y^{(i+1)} - g(\sum_{j=0}^{i} \beta_j^{(i)} K(x^{(j)}, x^{(i+1)})))$

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

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

$$K(x, z) = x^\top z, \tag{4}$$

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

$$K(x, z) = \exp\left(-\frac{\|x - z\|_2^2}{2\sigma^2}\right), \tag{5}$$

and finally the following function:

$$K(x, z) = \begin{cases} -1 & x = z \\ 0 & x \neq z \end{cases} \tag{6}$$

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: After implementing the Kernelized Perceptron, we get the following figures:

i. For the dot product:

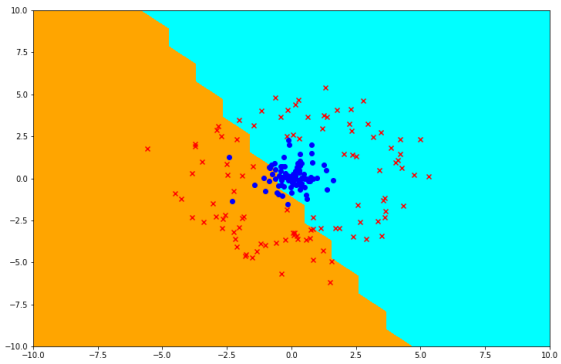

Figure 4: Perceptron prediction using the dot product kernel

ii. For the radial basis function (RBF):

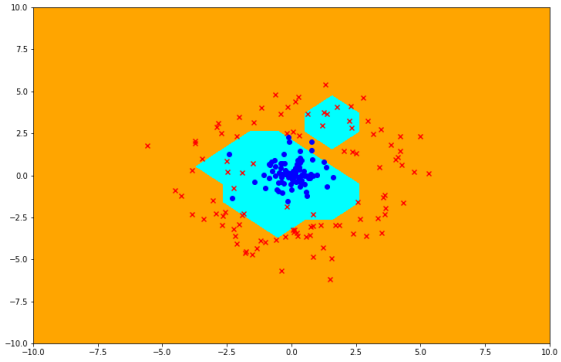

Figure 5: Perceptron prediction using the RBF kernel

iii. For the non kernel function:

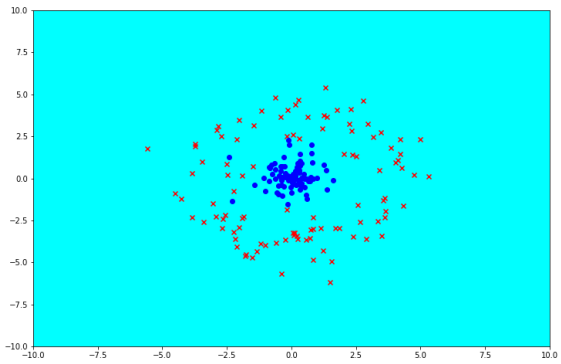

Figure 6: Perceptron prediction using the invalid kernel

(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: We see that the dot product kernel works a bit but it is incapable of totally separating the classes. This is due to the fact that $\phi(x)$ here is just the identity function. For this reason this Kernel is only capable of yielding a linear decision boundary.

The RBF Kernel on the other hand performs very good and this is because $\phi(x)$ is ∞ dimensional here, and therefore the classes can be separated in this high dimensional space.

The invalid kernel fails completely to give any meaningful results and this is because it is doing the opposite of what a Kernel is supposed to do. This Kernel yields a low value for similar vectors ($K(x, x) = -1$) and a high one for different vectors ($K(x, z) = 0$ for $x \neq z$) which is the opposite of what the two other Kernels are doing.

5. **[25 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:

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

Using the Bayes formula, $p(\theta|x, y) = \frac{p(\theta, y|x)}{p(y|x)} = \frac{p(y|x,\theta)p(\theta|x}{p(y|x)}$), we obtain

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

Since $p(y|x)$ does not depend on θ, so

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

(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,

$$\theta_{\text{MAP}} = \arg\min_{\theta} -\log p(y|x,\theta) + \lambda||\theta||_2^2. \tag{7}$$

Also, what is the value of λ?

Answer: We proved in (a) that

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

since the logarithmic function is an increasing function, so

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

thus

$$\theta_{\text{MAP}} = \arg\max_{\theta} log(p(y|x,\theta)) + log(p(\theta)),$$

therefore

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

we know the prior over $\theta, where \theta \sim \mathcal{N}(0, \eta^2 I)$, which implies

$$log(p(\theta)) = log(\frac{exp(\frac{-||\theta||_2^2}{2\eta^2})}{(2\pi\eta^{2n})^{1/2}}) = -\frac{1}{2}log(2\pi\eta^{2n}) - \frac{1}{2\eta^2}||\theta||_2^2.$$

Using the fact that $\frac{1}{2}log(2\pi\eta^{2n})$ does not depend on θ , we obtain

$$\theta_{\text{MAP}} = \arg\min_{\theta} -\log p(y|x,\theta) + \lambda||\theta||_2^2, \tag{8}$$

where $\lambda = \frac{1}{2\eta^2}$

(c) [7 points] Now consider a specific instance, a linear regression model given by $y = \theta^T x + \epsilon$ where $\epsilon \sim \mathcal{N}(0, \sigma^2)$. Assume that the random noise $\epsilon^{(i)}$ is independent for every training example $x^{(i)}$. Like before, assume a Gaussian prior on this model such that $\theta \sim \mathcal{N}(0, \eta^2 I)$. For notation, let X be the design matrix of all the training example inputs where each row vector is one example input, and $\vec{y}$ be the column vector of all the example outputs.

Come up with a closed form expression for θ_{MAP}.

Answer: We have from the previous question:

$$\theta_{\text{MAP}} = \arg\min_{\theta} -\log(p(y|x,\theta)) + \lambda||\theta||_2^2 \tag{9}$$

We have also: $p(Y|x,\theta) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi}\sigma} \exp(-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2})$.

Therefore, $\log(p(y|x,\theta)) = \sum_{i=1}^{n} -\log(\sqrt{2\pi}\sigma) - \frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}$.

Thus,

$$\begin{aligned}\theta_{\text{MAP}} &= \arg\min_{\theta} -\log(p(y|x,\theta)) + \lambda||\theta||_2^2 \\ &= \arg\min_{\theta} \sum_{i=1}^{n} \left[\log(\sqrt{2\pi}\sigma) + \frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}\right] + \lambda||\theta||_2^2 \\ &= \arg\min_{\theta} \sum_{i=1}^{n} \frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2} + \lambda||\theta||_2^2 \\ &= \arg\min_{\theta} \frac{1}{2\sigma^2}(Y - X\theta)^T(Y - X\theta) + \lambda||\theta||_2^2\end{aligned} \tag{10}$$

If we take the derivative of $L(\theta) = \frac{1}{2\sigma^2}(Y - X\theta)^T(Y - X\theta) + \lambda||\theta||_2^2$ w.r.t θ in the previous expression, we find:

$$\begin{aligned}\frac{\partial L(\theta)}{\partial \theta} &= \frac{1}{2\sigma^2}(\nabla_\theta(\theta^T X^T X\theta) - \nabla_\theta(Y^T X\theta) - \nabla_\theta(\theta^T X^T Y)) + 2\lambda\theta \\ &= \frac{1}{2\sigma^2}(2X^T X\theta - 2X^T Y) + 2\lambda\theta \\ &= \frac{1}{\sigma^2}(X^T X\theta - X^T Y) + 2\lambda\theta\end{aligned} \tag{11}$$

Setting this equal to zero, we get:
$\frac{\partial L(\theta)}{\partial \theta} = 0 \rightarrow (X^T X + 2\lambda\sigma^2\mathbf{I})\theta = X^T Y \rightarrow \theta = (X^T X + 2\lambda\sigma^2\mathbf{I})^{-1}X^T Y$

Therefore,

$$\theta = (X^T X + 2\lambda\sigma^2\mathbf{I})^{-1}X^T Y$$

(d) [5 points] Coding question: double descent phenomenon and the effect of regularization.
The Bayesian perspective provides a justification of using the L_2 regularization term as in equation (8), and it also provides a formula for the optimal choice of λ, assuming that we know the true prior for θ. For real-world applications, we often do not know the true prior, so λ is tuned based on the performance on the validation dataset. In this problem, you will be asked to verify empirically that the choice of λ you derived in part (c) is close to optimal, when the generating process of the data, θ, and the label y are as exactly described in part (c).

Meanwhile, you will empirically observe an interesting phenomenon, often called double descent, which was first discovered in 1970s and recently returned to spotlight. Sample-wise double descent is the phenomenon that the validation loss of some learning algorithm or estimator does not monotonically decrease as we have more training examples, but instead has a curve with two U-shaped parts. Model-wise double descent is a similar phenomenon as we increase the size of the model. For simplicity, here we only focus on the sample-wise double descent.

You will be asked to train on various datasets by minimizing the regularized cost function with various choices of λ:

$$-\log p(\vec{y}|X,\theta) + \lambda||\theta||_2^2 \tag{12}$$

and plot the validation losses for the choices of datasets and λ. You will observe the double descent phenomenon for relatively small λ but not the optimal choice of λ.

Problem details: We assume that the data are generated as described in part (c) with $d = 500, \sigma = 0.5$. You are asked to have a Gaussian prior $\theta \sim \mathcal{N}(0, \eta^2 I)$ with $\eta = 1/\sqrt{d}$ on the parameter θ. (The teaching staff indeed generated the ground-truth θ from this prior.)

You are given 13 training datasets of sample sizes $n = 200, 250, \ldots, 750$, and 800, and a validation dataset, located at

- `src/bayesianreg/train200.csv`, `train250.csv`, etc.
- `src/bayesianreg/validation.csv`

Let λ_{opt} denote the regularization strength that you derived from part (c). (λ_{opt} is a function of σ and η.)

For each training dataset $(X, \vec{y})$ and each $\lambda \in \{0, 1/32, 1/16, 1/8, 1/4, 1/2, 1, 2, 4\} \times \lambda_{opt}$, compute the optimizer of equation (12), and evaluate the mean-squared-error of the optimizer on the validation dataset.

Complete the `ridge_regression` method of `src/doubledescent/doubledescent.py` which takes in a training file and a validation file, computes the θ that minimizes training objective under different regularization strengths, and returns a list of validation errors (one for each choice of λ).

Include in your writeup a plot of the validation losses of these models. The x-axis is the size of the training dataset (from 200 to 800); the y-axis is the MSE on the validation dataset. Draw one line for each choice of λ connecting the validation errors across different training dataset sizes. Therefore, the plot should contain 9×13 points and 9 lines connecting them.

You should observe that for those choices of λ with $\lambda < \lambda_{opt}$, the validation error may increase and then decrease as we increase the sample size. However, double descent does not occur for λ_{opt} or any regularization larger than λ_{opt}.

Note: Use the Moore-Penrose pseudo-inverse as implemented in `numpy.linalg.pinv` if your matrix is singular.

Remark: If you would like to know more about double descent, please feel free to check out the partial list of references in the introduction and related work [2] but knowing the references or papers are unnecessary for solving this homework problem. Roughly speaking, it is mostly caused by the lack of regularization when $n \approx d$. When $n \approx d$, the data matrix is particularly ill-conditioned and stronger and more explicit regularization is needed.

Answer:

[2] Nakkiran, P., Venkat P., Kakade, S., Ma, T. Optimal Regularization Can Mitigate Double Descent. arXiv e-prints (Mar. 2020). arXiv:2003.01897

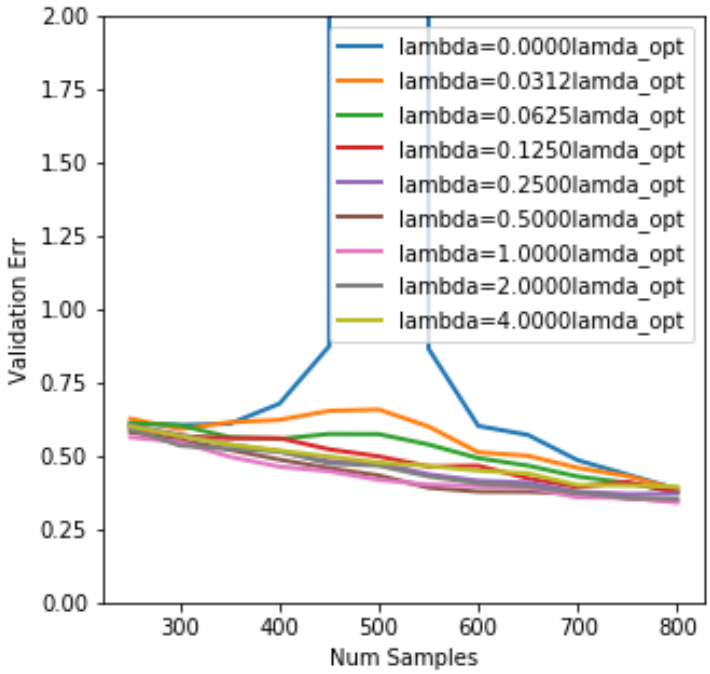

Figure 7: The validation error for the different data sets and λ choices

We notice that for small datasets the validation error increases then decreases.

(e) [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: We have, $\theta_{\text{MAP}} = \arg\min_\theta - \log(p(y|x,\theta)) - \log(p(\theta))$

And, $\log(p(\theta)) = \sum_{i=1}^{d} \log(\frac{1}{2b}\exp(-\frac{|\theta_i|}{b})) = \sum_{i=1}^{d} -\log(2b) - \frac{|\theta_i|}{b}$
Therefore,

$$\begin{aligned}
\theta_{\text{MAP}} &= \arg\min_{\theta} -\log(p(y|x,\theta)) + \log(p(\theta)) \\
&= \arg\min_{\theta} -\log(p(y|x,\theta)) - \sum_{i=1}^{d} -\frac{|\theta_i|}{b} \\
&= \arg\min_{\theta} \sum_{i=1}^{n} \frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2} - \sum_{i=1}^{d} -\frac{|\theta_i|}{b} \\
&= \arg\min_{\theta} ||X\theta - Y||_2^2 + \frac{2\sigma^2}{b}||\theta||_1
\end{aligned} \tag{13}$$

Finally,

$$J(\theta) = ||X\theta - Y||_2^2 + \gamma||\theta||_1$$

Where, $\gamma = \frac{2\sigma^2}{b}$

Remark: Linear regression with L_2 regularization is also commonly called *Ridge regression*, and when L_1 regularization is employed, is commonly called *Lasso regression*. These regularizations can be applied to any Generalized Linear models just as above (by replacing $\log p(y|x,\theta)$ with the appropriate family likelihood). Regularization techniques of the above type are also called *weight decay*, and *shrinkage*. The Gaussian and Laplace priors encourage the parameter values to be closer to their mean (*i.e.,* zero), which results in the shrinkage effect.

Remark: Lasso regression (*i.e.,* L_1 regularization) is known to result in sparse parameters, where most of the parameter values are zero, with only some of them non-zero.