EECS 189 Introduction to Machine Learning
Fall 2020 HW13

This homework is due **Wednesday, December 9 at 11:59 p.m.**

1 Getting Started

Read through this page carefully. You may typeset your homework in latex or submit neatly handwritten/scanned solutions. Please start each question on a new page. Deliverables:

1. Submit a PDF of your writeup, **with an appendix for your code**, to the appropriate assignment on Gradescope. If there are graphs, include those graphs in the correct sections. Do not simply reference your appendix.
2. If there is code, submit all code needed to reproduce your results.
3. If there is a test set, submit your test set evaluation results.

After you've submitted your homework, watch out for the self-grade form.

(a) Who else did you work with on this homework? In case of course events, just describe the group. How did you work on this homework? Any comments about the homework?

(b) Please copy the following statement and sign next to it. We just want to make it *extra* clear so that no one inadvertently cheats.

I certify that all solutions are entirely in my words and that I have not looked at another student's solutions nor have I looked at any online solutions to any of these problems. I have credited all external sources in this write up.

2 CNNs on Fruits and Veggies

Note that running the CNN for once on Google Colab or a Intel (R) i5-6360U CPU takes about 50 mins, so please start early.

In this problem, we will use the dataset of fruits and vegetables that was collected by students in a different semester. The goal is to accurately classify the produce in the image. In prior homework, we explored how to select features and then use linear classification to learn a function. There, we just used the color histograms as the basic data. We will now explore using Convolutional Neural Networks to optimize feature discovery jointly with learning a classification policy.

Denote the input state $x \in \mathbb{R}^{90\times90\times3}$, which is a downsampled RGB image with the fruit centered in it. Each data point will have a corresponding class label, which corresponds to their matching produce. Given 25 classes, we can denote the label as $y \in \{0, ..., 24\}$.

The goal of this problem is make sure you learn how to implement a Convolutional Neural Network (CNN) using PyTorch. We will use it to compare with both a fully-connected neural network as well as a nearest-neighbor approach.

Note all python packages needed for the project, will be imported already. **DO NOT import new Python libraries.** Also, this project will be computationally expensive on your computer's CPU. Please use Google Colab if you do not have a strong CPU. The dataset for this project can be download here(link). We recommend using PyTorch version 1.7.0 with Python3 interface.

(a) To begin the problem, we need to implement a CNN in PyTorch. In the starter code is a file named *cnn_pytorch.py*, the network architecture and the loss function are currently blank. You will have to write a convolutional neural network that has the following architecture:

   (a) Layer 1: A convolutional layer with 5 filters of size 15 by 15
   (b) Non-Linear Response: Rectified Linear Units
   (c) A max pooling operation with filter size of 3 by 3
   (d) Layer 2: A Fully Connected Layer with output size 512.
   (e) Non-Linear Response: Rectified Linear Units
   (f) Layer 3: A Fully Connected Layer with output size 25 (i.e. the class labels in one-hot encoding)
   (g) Loss Layer: Softmax Cross-Entropy Loss

   In the file *example_cnn_pytorch.py*, we show how to implement a network in PyTorch. Please use this as a reference. Once the network is implemented **run the corresponding part in the jupyter notebook on the dataset and report the resulting confusion matrix**. The goal is to ensure that your network compiles, but we should not expect the results to be good because it is randomly initialized.

(b) The next step to train the network is to complete the pipeline which loads the datasets and offers it as mini-batches into the network. **Fill in the missing code in** *data_manager_pytorch.py* **and report your code.** If you are using google colab, you can directly open and edit corresponding files on the file tab $\mapsto$ Drive $\mapsto$ MyDrive $\mapsto$ Colab Notebooks $\mapsto$ prob2.

(c) We will now complete the iterative optimization loop. Fill in the missing code in *trainer_pytorch.py* to iteratively apply SGD for a fixed number of iterations. In our system, we will be using an extra momentum term to help speed up the SGD optimization. **Run the jupyter notebook and report the resulting chart.**

(d) We want to better understand how the network was able to achieve better performance on our fruits and veggies dataset. Somehow, it is learning features to reduce the dimensionality of

the data. We can see what features were learned by examining the response maps after our convolutional layer.

The response map is the output image after the convolutional has been applied. This image can be interpreted as what features are interesting for classification. **Fill in the missing code in** *viz_features_pytorch.py* **and report the images specified**.

(e) Data augmentation are techniques used to increase the amount of data by adding slightly modified copies of already existing data or newly created synthetic data from existing data. It acts as a regularizer and can help improve generalization when training a machine learning model. It is a staple in image classification and usually improves the performance of CNNs.

We have implemented a image horizontal flip augmentation for you. Specifically, in the training phase, there is 50% possibility that the loaded image will be horizontally flipped. There will be no augmentation in the evaluation phase.

Change "FLIPAUG = True" in *data_manager_pytorch.py* **and run the corresponding part in the jupyter notebook. Compare the performance with part (c) which has no augmentation.**

(Bonus) Implement another augmentation algorithm that outperforms the given method. To start with, you can think of random crop, color jittering, etc.

(f) We have seen that the convolutional neural network can achieve a good performance on this dataset. However, we are not sure about whether a fully connected neural network could do the same job. We would like to replace the layers (a)-(c) in the original network by a fully connected layer and a ReLU layer. We want to have approximately the same number of parameters between layer (a) in the original network and the new fully connected layer. **How many hidden units will the new fully connected layer have? You can round up fractional numbers to the nearest integer. Do you observe any wierd aspect about this new network?**

Hint: For a convolution layer with input size $W \times H \times C$ and N convolution filters with each of them having a size of $X \times Y$, it has $C \times X \times Y \times N$ parameters.

(g) **Implement the fully connected network in the previous part and redo part (c). How's the performance of the fully connected network?**

(h) Given that our CNN has achieved some generalization with such low training error, it suggests that we should try other "interpolating" type of estimators. The easiest to understand such approach is 1-nearest neighbors. **Fill in the missing code in** *nn_classifier.py* **and report the performance as the numbers of neighbors is swept across when the corresponding jupyter notebook part is run.**

3 Gradient boosting and early stopping

In this problem we show how the greedy algorithms Matching Pursuit (a simplified version of the Orthogonal Matching Pursuit that you might have seen in EECS16A) and AdaBoost which you have seen in class, can both be interpreted as taking steps greedily in a direction which is closest

to the negative gradient in a restricted set/space. We start by connecting gradient descent on the square loss with Matching Pursuit.

(a) Let us explore a naive approach to doing gradient descent on square loss:

So far, we have considered the squared error loss $L(\mathbf{w}) := \frac{1}{2}\|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2$ as a function of the variable $\mathbf{w}$. When running gradient descent we took the derivative of L with respect to $\mathbf{w}$ and obtained iterations

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \alpha_t \mathbf{X}^\top(\mathbf{X}\mathbf{w}^t - \mathbf{y}) \tag{1}$$

We can substitute $\mathbf{v} = \mathbf{X}\mathbf{w} \in \mathbb{R}^n$ and instead consider the loss function as a map $\mathcal{L} : \mathcal{V} \mapsto \mathbb{R}$ mapping from the Euclidean space $\mathcal{V} = \mathbb{R}^n$ to real values:

$$\mathcal{L}(\mathbf{v}) = \frac{1}{2}\|\mathbf{y} - \mathbf{v}\|^2.$$

In order to minimize this new loss $\mathcal{L}$ with respect to $\mathbf{v}$, we could imagine simply running gradient descent on this loss as follows:

$$\mathbf{v}^{t+1} = \mathbf{v}^t - \alpha_t \nabla_{\mathbf{v}} \mathcal{L}(\mathbf{v})|_{\mathbf{v}=\mathbf{v}^t}, \tag{2}$$

for some suitably chosen step size $\alpha_t > 0$.

Compute the gradient $\nabla_{\mathbf{v}}\mathcal{L}(\mathbf{v})$. Write down the gradient descent update for $\mathbf{v}^{t+1}$.

What is the gradient $\nabla_{\mathbf{v}}\mathcal{L}(\mathbf{v})$ evaluated at $\mathbf{v} = \mathbf{X}\mathbf{w}$ which we denote as $\nabla_{\mathbf{v}}\mathcal{L}(\mathbf{v})\big|_{\mathbf{v}=\mathbf{X}\mathbf{w}}$? For $\mathbf{v}^t = \mathbf{X}\mathbf{w}^t$, compute the expression for $\mathbf{v}^{t+1}$ using update (2).

(b) Now we compare this update with the one obtained by running the gradient descent update of $\mathbf{w}$ on the loss L. In particular, using equation (1), **write down the expression for $\mathbf{X}\mathbf{w}^{t+1}$. How do you interpret the difference between the updates $\mathbf{X}\mathbf{w}^{t+1}$ and $\mathbf{v}^{t+1}$?** Think in terms of underlying subspaces in $\mathbb{R}^d$ in which the updates lie.

(c) As we see in the previous part, the updates $\mathbf{X}\mathbf{w}^{t+1}$ and $\mathbf{v}^{t+1}$ derived by taking gradients of L and $\mathcal{L}$ respectively, are not equivalent. In general, we may not be able to run the gradient updates (2) exactly, as there might be constraints on the update we want to make. Sometimes, these constraints are consequences of the problem formulation (in the previous part for example, we wanted a linear fit for y depending on $\mathbf{x}$). Sometimes, there are further constraints that come from computational considerations, as in the follow-up parts dealing with AdaBoost. In this problem, we will see that if instead of (2) we take a direction closest to the gradient in some restricted space, both procedures actually match.

In the specific example of linear regression, we want to find a vector $\widetilde{\mathbf{v}}^t$ which can be expressed as a linear function of our data matrix $\mathbf{X}$. Note that $\widetilde{\mathbf{v}}^t$ will be used for updating $\mathbf{v}$ later (equation (4)). For this purpose we restrict our search space to the column space of $\mathbf{X}$ (a subspace of $\mathbb{R}^n$) in which all vectors $\widetilde{\mathbf{v}}$ can be expressed by $\widetilde{\mathbf{v}} = \mathbf{X}\mathbf{w}$ for some vector $\mathbf{w}$ which has norm smaller than one (we'll see later why this is necessary), i.e. the set

$$\widetilde{\mathcal{V}} = \{\widetilde{\mathbf{v}} = \mathbf{X}\mathbf{w} : \|\mathbf{w}\|_2 \leq 1, \mathbf{w} \in \mathbb{R}^d\} \subset \mathcal{V}.$$

Notice that the gradient of $\mathcal{L}$ with respect to $\mathbf{v}^t$ need not be in this smaller subspace, i.e. $\nabla_{\mathbf{v}}\mathcal{L}(\mathbf{v}^t) \notin \widetilde{\mathcal{V}}$. In that case, we consider a greedy strategy: We first find the direction in $\widetilde{\mathcal{V}}$ which best aligns with the negative gradient. That is at each iteration, we find

$$\widetilde{\mathbf{v}}^t = \arg\max_{\widetilde{\mathbf{v}}\in\widetilde{\mathcal{V}}} \langle -\nabla\mathcal{L}(\mathbf{v}^t), \widetilde{\mathbf{v}}\rangle. \tag{3}$$

and then compute $\mathbf{v}^{t+1}$ using the update equation

$$\mathbf{v}^{t+1} = \mathbf{v}^t + \alpha_t \widetilde{\mathbf{v}}^t, \tag{4}$$

for an appropriate choice of step size α_t, where $\widetilde{\mathbf{v}}^t$ now belongs to the subset $\widetilde{\mathcal{V}}$. We now show how this new procedure yields updates in the same direction as performing the gradient step with respect to $\mathbf{w}$ explicitly.

Again assume that the current iterate is given by $\mathbf{v}^t = \mathbf{X}\mathbf{w}^t$ for some vector $\mathbf{w}^t$. Now, using equation (3) and the expression for $\nabla_{\mathbf{v}}\mathcal{L}(\mathbf{v})\big|_{\mathbf{v}=\mathbf{X}\mathbf{w}^t}$ derived previously, **show that $\widetilde{\mathbf{v}}^t$ in (3) is exactly parallel to the vector $-\mathbf{X}\mathbf{X}^\top(\mathbf{X}\mathbf{w}^t - \mathbf{y})$ (i.e. they are the same up to a positive scalar). Now write the update equation (4) in terms of $\mathbf{X}, \mathbf{w}^t, \mathbf{y}$ and α_t.** Thus, we have seen that gradient descent on the losses L and $\mathcal{L}$ are essentially the same (and in fact the same if adjusting the stepsize α_t accordingly) by choosing the set $\widetilde{\mathcal{V}}$ accordingly.

Hint: For any function h mapping to a scalar you may use that

$$\arg\max_{\mathbf{v}=\mathbf{X}\mathbf{w}:\|\mathbf{w}\|_2\le 1} h(\mathbf{v}) = \mathbf{X}\arg\max_{\|\mathbf{w}\|_2\le 1} h(\mathbf{X}\mathbf{w}).$$

(d) *Towards Matching Pursuit in the gradient descent framework* In this problem we are going to interpret the direction-selection part of Matching Pursuit as a general gradient-type method as described in (b) for an appropriate choice of the set $\widetilde{\mathcal{V}}$.

As you have seen in earlier homeworks, Orthogonal Matching Pursuit is a greedy method to pick relevant coordinates that can explain the data best. It searches for a direction (among the columns) such that the residual error can be explained best and updates the entire fit in the augmented space. Matching Pursuit (MP) is a simpler variant where the entire fit is not updated at every step, just the fit along the new coordinate that was chosen.

Let $\mathbf{X} \in \mathbb{R}^{n\times d}$ be the training feature matrix, $\mathbf{y}$ the observed training target variables from which we want to infer (potentially sparse) weights $\mathbf{w} \in \mathbb{R}^d$. At every iteration t, MP picks the column of $\mathbf{X}$ with maximal absolute inner product with the current residual $\mathbf{r}^t = \mathbf{y} - \mathbf{X}\mathbf{w}^t$, i.e.

$$i_t = \arg\max_{i=1,\dots,d} |\langle \mathbf{r}^t, \boldsymbol{\phi}_i(\mathbf{X})\rangle|. \tag{5}$$

where $\mathbf{X} \in \mathbb{R}^{n\times d}$ is the data matrix with columns $\boldsymbol{\phi}_i(\mathbf{X})$ and rows $\mathbf{x}_1, \dots, \mathbf{x}_n$, i.e. $\mathbf{X} = (\mathbf{x}_1, \dots, \mathbf{x}_n)^\top = (\boldsymbol{\phi}_1(\mathbf{X}), \dots, \boldsymbol{\phi}_d(\mathbf{X}))$. Note that $\mathbf{x}_i$ denotes the i-th data point and $\boldsymbol{\phi}_i(\mathbf{X})$ denotes the vector of the i-th features of all the data points (i.e. the i-th column of the data matrix $\mathbf{X}$).

The updates of MP at each iteration t then read

$$\mathbf{X}\mathbf{w}^{t+1} = \mathbf{X}\mathbf{w}^t + \frac{\langle \mathbf{r}^t, \boldsymbol{\phi}_{i_t}(\mathbf{X})\rangle}{\|\boldsymbol{\phi}_{i_t}(\mathbf{X})\|^2}\boldsymbol{\phi}_{i_t}(\mathbf{X}). \tag{6}$$

Now your task in this part is to establish the correspondence between the MP selection rule (5) and the general approach in (restricted) gradient descent given by (3).

In particular, set $\mathbf{v}^t = \mathbf{X}\mathbf{w}^t$ and **find the set $\widetilde{\mathcal{V}}$ such that the selection represented by (5) can be understood as just being (3) in action, that is**

$$\begin{aligned}\widetilde{\mathbf{v}}^t &= \arg\max_{\widetilde{\mathbf{v}}\in\widetilde{\mathcal{V}}} \langle -\nabla\mathcal{L}(\mathbf{v}^t), \widetilde{\mathbf{v}}\rangle \\ &= \text{sign}(\langle \mathbf{r}^t, \boldsymbol{\phi}_{i_t}(\mathbf{X})\rangle)\boldsymbol{\phi}_{i_t}(\mathbf{X}).\end{aligned}$$

Hint: don't forget how to deal with the absolute value.

(e) In traditional gradient descent, the rule for choosing the step-size α_t is left open. One way of doing this is adaptively by *linesearch* — picking the step-size α that reduces the loss the most. Formally, choosing stepsize α_t via linesearch with respect to the search direction $\widetilde{\mathbf{v}}^t$ at each iteration for a loss $\mathcal{L}$ means finding the best α such that $\alpha_t = \arg\min_{\alpha\geq 0} \mathcal{L}(\mathbf{v}^t + \alpha\widetilde{\mathbf{v}}^t)$. **Argue that the update (4) with $\widetilde{\mathbf{v}}^t = \text{sign}(\langle \mathbf{r}^t, \boldsymbol{\phi}_{i_t}(\mathbf{X})\rangle)\boldsymbol{\phi}_{i_t}(\mathbf{X})$ you derived in (c), and choosing α_t via linesearch along that $\widetilde{\mathbf{v}}^t$ is the same as doing the MP update (6).**

(f) *MP as gradient boosting* The two previous greedy principles of matching-pursuit (column selection and linesearch) can be extended to more general settings of non-linear and non-parametric function classes $\mathcal{F}$ and general loss functions $\mathcal{L}$ (which need not be squared loss). Consider the general learning problem where again we are given a bunch of pairwise training samples in the form of $(\mathbf{x}_i, y_i)$ for $i = 1, \ldots, n$, with $\mathbf{x}_i \in \mathbb{R}^d$ and $y_i \in \mathcal{Y} \subset \mathbb{R}$ and we want to learn a function $f : \mathbb{R}^d \to \mathcal{Y}$ from a function space $\mathcal{F}$ (with possibly non-linear and non-parametric basis elements, such as trees) which we can then use to predict y_{test} by $f(\mathbf{x}_{test})$ for a test sample $\mathbf{x}_{test}$.

For this purpose we minimize the following (average) loss $\mathcal{L} : \mathbb{R}^n \to \mathbb{R}$

$$\mathcal{L}(\mathbf{f}(\{\mathbf{x}\}_{i=1}^n)) = \mathcal{L}(f(\mathbf{x}_1), \ldots, f(\mathbf{x}_n)) = \frac{1}{n}\sum_{i=1}^n \mathcal{L}(y_i, f(\mathbf{x}_i)).$$

for some point-wise loss function $\mathcal{L} : \mathcal{Y} \times \mathcal{Y} \mapsto \mathbb{R}$ for one sample. Notice that the loss $\mathcal{L}$ maps a vector to a scalar, as in the linear settings. Here, we use $\mathbf{f}(\{\mathbf{x}\}_{i=1}^n) := (f(\mathbf{x}_1), \ldots, f(\mathbf{x}_n))$ to denote the vector of function values of $f \in \mathcal{F}$ at the points $\mathbf{x}_1, \ldots, \mathbf{x}_n$.

As in gradient descent (4), we want to minimize the loss using an iterative algorithm with updates of the form

$$f^{t+1} = f^t + \alpha_t \widetilde{f^t}.$$

Note that our notation follows the convention that $f^t \in \mathcal{F}$ is the current iterate and $\widetilde{f^t} \in \mathcal{W}$ is the direction that we are adding at each step. The main question is how to choose $\widetilde{f^t}$. Let's assume that it is for example computationally much simpler to search over a smaller set of functions $\mathcal{W} \subset \mathcal{F}$ at each timestep.

Similar to the parametric update in (3), we then choose the function $\widetilde{f} \in \mathcal{W}$ for which the function value vector $\widetilde{\mathbf{f}}(\{\mathbf{x}\}_{i=1}^n)$ has the maximal inner product with the negative gradient, i.e.

$$\widetilde{f^t} = \arg\max_{\widetilde{f}\in\mathcal{W}} \langle -\nabla\mathcal{L}(\mathbf{f}^t(\{\mathbf{x}\}_{i=1}^n)), \widetilde{\mathbf{f}}(\{\mathbf{x}\}_{i=1}^n)\rangle \tag{7}$$

where the gradient $\nabla\mathcal{L}(\mathbf{f}^t(\{\mathbf{x}\}_{i=1}^n)), \widetilde{\mathbf{f}}(\{\mathbf{x}\}_{i=1}^n) = \nabla_{\mathbf{v}}\mathcal{L}(\mathbf{v})\big|_{\mathbf{v}=\mathbf{f}^t(\{\mathbf{x}\}_{i=1}^n)}$ which you have already computed above for the square loss.

Now we want to see how Matching Pursuit can be viewed as an instance of this framework, also called *gradient boosting* or *functional gradient descent*. Specifically, **show that the particular choices of**

$$\mathcal{F} = \{f(\mathbf{x}) = \mathbf{w}^\top\mathbf{x} : \mathbf{w} \in \mathbb{R}^d\} \textbf{ and } \mathcal{W} = \{\widetilde{f}(\mathbf{x}) = \pm\mathbf{e}_i^\top\mathbf{x} : i = 1, \ldots, d\},$$

where $\mathbf{e}_i$ is the i-th standard basis element with a 1 in the i-th position and zeros elsewhere, yield Matching Pursuit as a form of gradient boosting. Notice that $\mathcal{F}$ is the function space of all linear functions and $\mathcal{W}$ is the set of functions on vectors $\mathbf{x} \in \mathbb{R}^d$ which pick out signed coordinates of $\mathbf{x}$.

Hint: For matching pursuit, what are $\widetilde{\mathbf{f}}^{\mathbf{t}}(\{\mathbf{x}\}_{i=1}^n)$ and $\mathbf{f}^t(\{\mathbf{x}\}_{i=1}^n)$?

(g) *AdaBoost as gradient boosting* Now let's have a look at non-linear and non-parametric functions such as decision trees for classification. In this problem we will establish that AdaBoost is also an instance of gradient boosting on a particular loss function (which is not the zero-one loss!).

Notice how each decision tree can be viewed as a function $\widetilde{f} : \mathbb{R}^d \to \{-1, +1\}$. Since in each iteration we fit a tree to the weighted samples, our function set $\mathcal{W}$ now corresponds to the set of decision trees with a fixed set of parameters.

Remember the AdaBoost algorithm from lecture and the previous problem. At each iteration t, it upweights each sample i by a different w_i^t in every iteration and finds a new tree which solves the reweighted classification problem

$$\widetilde{f}_{Ada}^t = \arg\min_{\widetilde{f}\in\mathcal{W}} \sum_{i=1}^n w_i^t \mathbb{I}(y_i \neq \widetilde{f}(\mathbf{x}_i)) \text{ where } w_i^t = \frac{\exp(-y_i f^t(\mathbf{x}_i))}{\sum_{i=1}^n \exp(-y_i f^t(\mathbf{x}_i))}$$

where f^t is a linear combination of functions in $\mathcal{W}$ found by previous iterations and $\mathbb{I}(A) = 1$ if A is true and 0 otherwise (the indicator function). Notice that the samples which the current classifier f^t gets wrong are assigned a larger weight.

The update for the overall classifier reads

$$f^{t+1} = f^t + \alpha_t \widetilde{f}_{Ada}^t \tag{8}$$

The classification rule is then obtained by taking the sign of the final iterate at time τ, i.e. $f_{\text{final}}(\mathbf{x}) = \text{sign}(f^\tau(\mathbf{x}))$ which lies in a bigger space $\mathcal{F} \supset \mathcal{W}$.

Show that $\widetilde{\mathbf{f}}^{\mathbf{t}}_{\mathbf{Ada}}(\{\mathbf{x}\}_{i=1}^n) = \widetilde{\mathbf{f}}^{\mathbf{t}}(\{\mathbf{x}\}_{i=1}^n)$ in the gradient boosting framework (7) for the loss $\ell(y, f(\mathbf{x})) = \exp\{-yf(\mathbf{x})\}$. This implies that finding the best fit for the reweighted samples is equivalent to fitting the gradient. For the gradient in (7), notice that we again view $\mathcal{L}$ as a function of a vector $\mathbf{v}$ with

$$\mathcal{L}(\mathbf{v}) = \frac{1}{n}\sum_{i=1}^n \ell(y_i, f(\mathbf{x}_i)) = \frac{1}{n}\sum_{i=1}^n \exp\{-y_i v_i\} \tag{9}$$

and the gradient $\nabla\mathcal{L}(\mathbf{f}^t(\{\mathbf{x}\}_{i=1}^n)) = \nabla_{\mathbf{v}}\mathcal{L}(\mathbf{v})\big|_{\mathbf{v}=\mathbf{f}^t(\{\mathbf{x}\}_{i=1}^n)}$.

Hint: Now note that for $y \in \{-1,+1\}$ and $\widetilde{f} : \mathbb{R}^d \to \{-1,+1\}$, we have

$$y_i\widetilde{f}(\mathbf{x}_i) = 2(1 - \mathbb{I}(y_i \neq \widetilde{f}(\mathbf{x}_i))) - 1. \tag{10}$$

Hint: Recall that the maximizer does not change if you scale by or add to the function a constant factor which is independent of the object you are searching over, i.e.

$$\arg\max_{\widetilde{f}\in\mathcal{W}} \; CH(\widetilde{\mathbf{f}}(\{\mathbf{x}\}_{i=1}^n)) + G(\mathbf{f}(\{\mathbf{x}\}_{i=1}^n)) = \arg\max_{\widetilde{f}\in\mathcal{W}} \; H(\widetilde{\mathbf{f}}(\{\mathbf{x}\}_{i=1}^n))$$

for some functions H, G that depend on the function value vectors $\mathbf{f}, \widetilde{\mathbf{f}}$, and a scalar $C \in \mathbb{R}$. $\mathbf{f}$ does not depend on $\widetilde{f}$.

(h) *AdaBoost weight update as linesearch steps* In this problem we show that the weights used in the AdaBoost algorithm at each time step t correspond to the "best" stepsize in the direction $\widetilde{f}_{Ada}^t$ using linesearch.

Let's define the error of the classifier $\widetilde{f}_{Ada}^t$ as $\epsilon_t = \sum_{i=1}^n w_i^t \mathbb{I}(y \neq \widetilde{f}_{Ada}^t(\mathbf{x}_i))$. In practice, AdaBoost uses the stepsize $\alpha_t = \frac{1}{2}\ln\frac{1-\epsilon_t}{\epsilon_t}$.

Show that the AdaBoost stepsize at any time t corresponds to the stepsize chosen via linesearch for the direction $\widetilde{\mathbf{v}} = \widetilde{\mathbf{f}}_{\mathbf{Ada}}^{\mathbf{t}}(\{\mathbf{x}\}_{i=1}^n)$, i.e. it solves the optimization problem

$$\min_{\alpha\in\mathbb{R}} \frac{\mathcal{L}(\mathbf{f}^t(\{\mathbf{x}\}_{i=1}^n) + \alpha\widetilde{\mathbf{f}}_{\mathbf{Ada}}^{\mathbf{t}}(\{\mathbf{x}\}_{i=1}^n))}{\mathcal{L}(\mathbf{f}^t(\{\mathbf{x}\}_{i=1}^n))}$$

Hint: division by a constant factor (independent of the variable we are minimizing over) does not change the minimizer of the optimization problem, i.e.

$$\arg\min_{\alpha} \frac{g(\alpha; \{\mathbf{x}\}_{i=1}^n)}{h(\{\mathbf{x}\}_{i=1}^n)} = \arg\min_{\alpha} g(\alpha; \{\mathbf{x}\}_{i=1}^n)$$

(i) Now we will explore how gradient boosting does in practice for both examples we have theoretically studied in the previous sections: Matching Pursuit and AdaBoost. In particular, we will see how the number of updates, equivalent to the number of trees that we are fitting, effects the test and training error.

We have written some code (see the corresponding jupyter notebook) that trains several classifiers on the Spam dataset you worked with in the previous problem. **Run the code for this part of the question.** It generates a training error plot that uses several different decision trees (depths 1, 2, and 4) as well as several AdaBoost-trained classifiers, each built off of one of these decision tree classifiers. **Do you see a trend in the performance of different trees by themselves? Do you observe a trend in the training error as you use deeper trees for AdaBoost? Why might this happen?**

(j) Now we examine the test error, and compare against a baseline classifier (a decision tree of depth 9). **Run the code for this part. Is there a difference in the training and test error? Which decision tree depth works best for AdaBoost? Explain your observations.**

(k) In the last part, you noticed that the test error decreases as a function of boosting iterations in the beginning but eventually it starts to increase when the number of decision trees in Adaboost is pretty large. In practice, this phenomenon motivates us to stop training early and limit the number of classifiers used in a boosting setting. Here "stopping early" means that training error has not reduced to zero but we have stopped training! Refer to the plot from the previous part and answer: **Do you think limiting the number of base classifiers used for AdaBoost would help? Which base classifier can we run more boosting iterations on before the test error starts increasing? Justify your answer intuitively.**

(l) In this part, we connect this phenomenon to matching pursuit. The provided code generates a synthetic dataset:

$$\mathbf{y}_{\text{train}}, \mathbf{y}_{\text{test}} \in \mathbb{R}^n$$
$$\mathbf{X}_{\text{train}}, \mathbf{X}_{\text{test}} \in \mathbb{R}^{n \times d}$$
$$\mathbf{w} \in \mathbb{R}^d,$$

with (for both test and train data)

$$\mathbf{y} = \mathbf{X}\mathbf{w} + \mathbf{z}.$$

Here, $\mathbf{w}$ is a sparse vector (it has only a few non-zero entries) and $\mathbf{z}$ denotes noise. Given the observations $\mathbf{y}_{\text{train}}$ and the feature matrix $\mathbf{X}_{\text{train}}$, we are tasked with finding a sparse solution $\hat{\mathbf{w}}$ for which we use the matching pursuit algorithm.

Let $\hat{\mathbf{w}}^t_{\text{MP}}$ denote the estimate of $\mathbf{w}$ recovered by MP after t-iterations. The code also plots the training error $\|\mathbf{y}_{\text{train}} - \mathbf{X}_{\text{train}}\hat{\mathbf{w}}^t_{\text{MP}}\|^2$ and the test error $\|\mathbf{y}_{\text{test}} - \mathbf{X}_{\text{test}}\hat{\mathbf{w}}^t_{\text{MP}}\|^2$ as a function of iterations t (as t increases matching pursuit builds an estimate using a greater number of features). **Run the code for this part. Explain the shape of the training error plot. Does the plot for test error look similar to the one from part (i)? Comment on the similarities.** You may use conclusions obtained in previous parts to justify your comments.

4 Neural Tangent Kernels

This problem shows how we can understand neural network behavior better if we leverage a very standard tool that you have seen in EECS16B (or wherever you learned similar ideas) — linearization of nonlinear functions. This staple tool from control theory and analog transistor circuits (where it is called "small signal analysis" and is developed in courses like EE105) turns out to be vital to give us insight into modern neural networks. In hindsight, this is perhaps not surprising given that a neural network is basically an analog circuit with nonlinear elements in it. The key is to view the network as a function not of its supposed inputs, but instead of its parameter weights. Since the learning dynamics evolve on the weights, a "small signal" view of the response to the weights sheds some important insight into what is going on.

Consider the two-layer neural network which computes the following function:

$$f(\mathbf{x}|\mathbf{a}, \mathbf{W}) = \frac{1}{\sqrt{m}} \sum_{r=1}^{m} a_r \sigma(\mathbf{w}_r^\top \mathbf{x}).$$

We impose the following assumptions:

- Initial values of final layer weights $\{a_i\}$ are centered i.i.d. random variables from the segment $[-A, A]$ (not necessarily from the uniform distribution).
- Initial values of weight vectors $\{w_r\}$ are i.i.d. random vectors independent of the $\{a_r\}$,
- The nonlinear activation function σ and its first and second derivatives σ' and σ'' are all bounded (for example you can think of sigmoid activation function).

For understanding convenience, the model that we will use to make predictions from the learned weights is the following:

$$\hat{f}(\mathbf{x}|\mathbf{a}, \mathbf{W}) := f(\mathbf{x}|\mathbf{a}, \mathbf{W}) - f(\mathbf{x}|\mathbf{a}_{\text{initial}}, \mathbf{W}_{\text{initial}})$$

i.e. we subtract out the guess that our initial random weights make. (This corresponds to the concept of bias point in analog circuits and the idea of operating point in control theory.)

Why? Because the spirit of this problem is to take the Taylor expansion of the neural network with respect to the weights that we're learning and just keep the first two terms — the constant term (corresponding to the random initial state's predictions) and the linear term (that captures how the network's behavior will change locally as we learn). The prediction model above lets us focus entirely on the linear term where all the action is — the jupyter parts will help you see this more clearly.

Our further goal is to understand what happens for large neural networks. Here, we'll use another standard approach — we'll take limits in two ways. We'll let $m \to \infty$ and we will also use a continuous-time perspective since that lets us leverage the core differential-equation thinking that you've developed in other courses (like EECS16B).

Consequently, this problem studies training of such a neural net with gradient descent in the "infinite width" regime, which means that we will set parameters of σ, distributions of a_r and $\mathbf{w}_r$, the number of data points n and the data $\mathbf{X}$ itself to be **constant**, and will only have m going to infinity. What happens when we take m to infinity? As it turns out, in this regime training a neural net is no different from just doing a kernel regression with kernel $K(\mathbf{x}_i, \mathbf{x}_j) = \langle \nabla_{\mathbf{a},\mathbf{W}} f(\mathbf{x}_i|\mathbf{a}, \mathbf{W}), \nabla_{\mathbf{a},\mathbf{W}} f(\mathbf{x}_j|\mathbf{a}, \mathbf{W}) \rangle$, which in turn itself converges to a fixed kernel in the infinite-width limit. Sure, there are an infinite number of "features," but that is not an obstacle since the actual (delta-) weights being applied to them are also infinitesimal. This limiting kernel is often called the "Neural Tangent Kernel" (NTK) for obvious reasons. This is what "small signal" analysis as applied to a neural net is about.

The rough outline of our argument in this problem is the following: we spend most of our time showing that we don't go very far in the weight space during gradient descent, and then in the last part we show why it means that we are doing kernel regression.

We start with computing the gradient of our prediction:

$$\frac{\partial}{\partial a_r} f(\mathbf{x}|\mathbf{a}, \mathbf{W}) = \frac{1}{\sqrt{m}} \sigma(\mathbf{w}_r^\top \mathbf{x}),$$

$$\nabla_{\mathbf{w}_r} f(\mathbf{x}|\mathbf{a}, \mathbf{W}) = \frac{1}{\sqrt{m}} a_r \sigma'(\mathbf{w}_r^\top \mathbf{x})\mathbf{x},$$

For given training data $\mathbf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_n]^\top$ and $\mathbf{y} = [y_1, \ldots, y_n]^\top$ consider the MSE loss:

$$L(\mathbf{X}, \mathbf{W}, \mathbf{a}) = \frac{1}{2} \sum_{i=1}^{n} \Big(y_i - \hat{f}(\mathbf{x}_i|\mathbf{a}, \mathbf{W})\Big)^2$$

To simplify the math, instead of considering (usual) discrete-time gradient descent, we will assume that during the training the parameters of our NN evolve according to the continuous-time gradient descent defined by the system of differential equations:

$$\frac{d\mathbf{w}_r(t)}{dt} = -\nabla_{w_i} L(\mathbf{X}, \mathbf{W}(t), \mathbf{a}(t)) = \sum_{i=1}^{n} \Big(y_i - \hat{f}(\mathbf{x}_i|\mathbf{a}(t), \mathbf{W}(t))\Big) \nabla_{\mathbf{w}_r} f(\mathbf{x}_i|\mathbf{a}(t), \mathbf{W}(t)),$$
$$\frac{d\mathbf{a}(t)}{dt} = -\nabla_{\mathbf{a}} L(\mathbf{X}, \mathbf{W}(t), \mathbf{a}(t)) = \sum_{i=1}^{n} \Big(y_i - \hat{f}(\mathbf{x}_i|\mathbf{a}(t), \mathbf{W}(t))\Big) \nabla_{\mathbf{a}} f(\mathbf{x}_i|\mathbf{a}(t), \mathbf{W}(t)).$$

You can think of this process as of your usual gradient descent with infinitesimally small step size and a rescaling of time.

It turns out that looking at the evolution of weights directly is not very convenient. The main idea is to look at the dynamics of the predictions on the training data: denote

$$u_i(t) := \hat{f}(\mathbf{x}_i|\mathbf{a}(t), \mathbf{W}(t)), \quad \mathbf{u}(t) := [u_1(t), \ldots, u_n(t)]^\top.$$

Note that $\mathbf{u}(0) = \mathbf{0}$ because of how we simplified things. Then during the GD we have

$$\begin{aligned}
\frac{du_i(t)}{dt} &= \left\langle \nabla_{\mathbf{a}} f(\mathbf{x}_i|\mathbf{a}, \mathbf{W}), \frac{d\mathbf{a}(t)}{dt} \right\rangle + \sum_{r=1}^{m} \left\langle \nabla_{\mathbf{w}_i} f(\mathbf{x}_i|\mathbf{a}, \mathbf{W}), \frac{d\mathbf{w}_r(t)}{dt} \right\rangle \\
&= \sum_{j=1}^{n} \Big(y_j - u_j(t)\Big) \Big\langle \nabla_{\mathbf{a}} f(\mathbf{x}_i|\mathbf{a}, \mathbf{W}), \nabla_{\mathbf{a}} f(\mathbf{x}_j|\mathbf{a}, \mathbf{W}) \Big\rangle \\
&+ \sum_{j=1}^{n} \sum_{r=1}^{m} \Big(y_j - u_j(t)\Big) \Big\langle \nabla_{\mathbf{w}_r} f(\mathbf{x}_i|\mathbf{a}, \mathbf{W}), \nabla_{\mathbf{w}_r} f(\mathbf{x}_j|\mathbf{a}, \mathbf{W}) \Big\rangle
\end{aligned}$$

So, looking at the differential equation satisfied by the residuals (this should remind you of what you say in the review problem in HW0 when we looked at discrete-time gradient-descent for least-squares), we see:

$$\frac{d}{dt}(\mathbf{y} - \mathbf{u}(t)) = -(\mathbf{G}(t) + \mathbf{H}(t))(\mathbf{y} - \mathbf{u}(t)),$$

where $\mathbf{G}(t)$ and $\mathbf{H}(t)$ are empirical kernel matrices:

$$\mathbf{G}_{i,j}(t) := \frac{1}{m} \sum_{r=1}^{m} \sigma(\mathbf{w}_r(t)^\top \mathbf{x}_i)\sigma(\mathbf{w}_r(t)^\top \mathbf{x}_j)$$

$$\mathbf{H}_{i,j}(t) := \mathbf{x}_i^\top \mathbf{x}_j \frac{1}{m} \sum_{r=1}^{m} a_r(t)^2 \sigma'(\mathbf{w}_r(t)^\top \mathbf{x}_i)\sigma'(\mathbf{w}_r(t)^\top \mathbf{x}_j)$$

As m goes to infinity, $\mathbf{G}(0)$ and $\mathbf{H}(0)$ converge to kernel Gram matrices with kernels $\mathbf{G}^\infty$ and $\mathbf{H}^\infty$ correspondingly:

$$\begin{aligned}
\mathbf{G}^\infty_{i,j} &:= \mathbb{E}_{\mathbf{w}}[\sigma(\mathbf{w}^\top \mathbf{x}_i)\sigma(\mathbf{w}^\top \mathbf{x}_j)], \\
\mathbf{H}^\infty_{i,j} &:= \mathbf{x}_i^\top \mathbf{x}_j \mathbb{E}[a^2]\mathbb{E}_{\mathbf{w}}[\sigma'(\mathbf{w}^\top \mathbf{x}_i)\sigma'(\mathbf{w}^\top \mathbf{x}_j)].
\end{aligned}$$

Denote

$$\mathbf{K}(t) := \mathbf{G}(t) + \mathbf{H}(t), \quad \mathbf{K}^\infty := \mathbf{G}^\infty + \mathbf{H}^\infty.$$

Note that $\mathbf{K}^\infty$ is exactly the kernel Gram matrix that corresponds to NTK.

In this problem we assume that the matrix $\mathbf{K}^\infty$ is non-degenerate, i.e. $\lambda_{\min}(\mathbf{K}^\infty) > 0$. (This is a reasonable assumption since there are more features than data points.) Our goal is to show that under this condition training with gradient descent leads to the same prediction as kernel regression with the neural tangent kernel.

(a) The first observation that we are going to make is that if the lowest eigenvalue of our matrix $\mathbf{K}(t)$ remains separated from zero, then the residuals converge rapidly to zero loss. This rapid convergence will help us prove that our weights don't go very far during the training.

Suppose that for some $T > 0$ for any $t \in [0, T]$ the lowest eigenvalue $\lambda_{\min}(\mathbf{K})$ of $\mathbf{K}(t)$ is at least $\bar{\lambda} > 0$. **Show that for any** $t \in [0, T]$

$$\|\mathbf{y} - \mathbf{u}(t)\|^2 \le \|\mathbf{y}\|^2 \exp\left(-2t\bar{\lambda}\right)$$

Hint: consider $\frac{d}{dt}\|\mathbf{y} - \mathbf{u}(t)\|^2$. *Don't forget that* $\mathbf{u}(0) = \mathbf{0}$.

(b) As promised, now we want to show that our weights don't go very far and we will not get out of the local regime where the NTK is a good approximation. Suppose for some $T > 0$ there exist $\gamma > 0$ and $\bar{\lambda} > 0$ s.t. for any $t < T$ the following holds:

- for any r, $|a_r(t)| < \gamma$ (i.e. there is some bound on how big the final-layer weights become),
- $\lambda_{\min}(\mathbf{K}(t)) \ge \bar{\lambda}$

Show that for any $t < T$ **and any** r

$$\begin{aligned}
\left\| \frac{d\mathbf{w}_r(t)}{dt} \right\| &\le \frac{1}{\sqrt{m}} \|\mathbf{y}\| \gamma C(\sigma, \mathbf{X}, n) \exp\left(-2t\bar{\lambda}\right), \\
\left| \frac{da_r(t)}{dt} \right| &\le \frac{1}{\sqrt{m}} \|\mathbf{y}\| C(\sigma, \mathbf{X}, n) \exp\left(-2t\bar{\lambda}\right)
\end{aligned}$$

where $C(\sigma, \mathbf{X}, n)$ **is some constant that only depends on** σ**,** $\mathbf{X}$ **and** n**.**

(c) In the previous part we controlled the speed with which our weights may travel. Now let's control the distance: **show that for any** $T, \bar{\lambda}, \gamma, \delta > 0$ **there exists** m_0 **s.t. for any** $m > m_0$ **the following holds: if we can control the condition number and the magnitude of the weights a i.e.**

- $\forall t \in [0, T]\ \lambda_{\min}(\mathbf{K}(t)) > \bar{\lambda}$**,**
- $\forall t \in [0, T]\ \max_r |a_r(t)| < \gamma$

then for any $t \in [0, T]$ **and any** $r\ \|\mathbf{w}_r(t) - \mathbf{w}_r(0)\| < \delta$ **and** $|a_r(t) - a_r(0)| < \delta$**.**

Hint: you need to integrate the bounds from the previous part. How do they behave when m grows?

(d) So far we were assuming that the lowest singular value of $\mathbf{K}(t)$ stays separated from zero. We know that it is true at the initialization as $\mathbf{K}(0)$ is close to $\mathbf{K}^\infty$ for m large enough. However, it is not clear why it should stay like that throughout the training. To show that we are going to use something like a "recursion": we already showed that if $\lambda_{\min}(\mathbf{K}(t))$ stays separated from zero, then the weights don't go very far. Now we are going to show that if the weights don't go very far, then $\lambda_{\min}(\mathbf{K}(t))$ stays separated from zero! In the next part we will put everything together and finalize the argument, but for now

Show that if for any r**,** $\|\mathbf{w}_r(t) - \mathbf{w}_r(0)\| < \delta$ **and** $|a_r(t) - a_r(0)| < \delta$ **then**

$$\begin{aligned}\|\mathbf{G}(t) - \mathbf{G}(0)\| &\leq C_G(n, \sigma, \mathbf{X})\delta,\\ \|\mathbf{H}(t) - \mathbf{H}(0)\| &\leq C_H(n, \sigma, A, \mathbf{X})(\delta + \delta^2)\end{aligned}$$

where $C_G(n, \sigma, \mathbf{X})$ **is a constant that only depends on** $n, \sigma, \mathbf{X}$**, and** $C_H(n, \sigma, A, \mathbf{X})$ **is a constant that only depends on** $n, \sigma, A, \mathbf{X}$**.**

Hint: you have explicit formulas for $\mathbf{G}$ *and* $\mathbf{H}$*. How does it help that the derivatives of* σ *are bounded, and the distribution of* a_r *also lies in a bounded segment?*

(e) Finally, we can put everything together: **Show that for any** $\varepsilon > 0$ **we can choose** m **to be so large that with probability at least** $1 - \varepsilon$ **for any** t **the following hold:**

- $\|\mathbf{K}(t) - \mathbf{K}^\infty\| \leq \varepsilon$**,**
- $\max_r \|w_r(t) - w_r(0)\| \leq \varepsilon$**,**
- $\max_r |a_r(t) - a_r(0)| < \varepsilon$

Your argument should (probably) go as follows:

- For large m, $\mathbf{K}(0)$ is close to $\mathbf{K}^\infty$, so $\lambda_{\min}(\mathbf{K}(0))$ is separated from zero.
- The weights $\{a_r\}$ are always all bounded at initializaiton.
- While $\lambda_{\min}(\mathbf{K}(0))$ is separated from zero and the weights $\{a_r\}$ are bounded the weights don't go very far if m is large enough.
- However, if the weights don't go very far, then $\lambda_{\min}(\mathbf{K}(0))$ stays separated from zero and the weights $\{a_r\}$ stay bounded.

- Therefore, if m is large enough, the weights will not go far away from initialization, and the matrix $\mathbf{K}(t)$ will not go far away from $\mathbf{K}^\infty$.

(f) Finally we can show that going not far from the initialization means that we are not actually doing anything non-linear! We can write our prediction at some new point $\mathbf{x}$ as

$$\begin{aligned}\hat{f}(\mathbf{x}) &= \int_0^\infty \frac{d}{dt} f(\mathbf{x}|\mathbf{a}(t), \mathbf{W}(t))\, dt \\ &= \int_0^\infty \left\langle \nabla_{\mathbf{a},\mathbf{W}} f(\mathbf{x}|\mathbf{a}(t), \mathbf{W}(t)), \sum_{i=1}^n (y_i - u_i(t)) \nabla_{\mathbf{a},\mathbf{W}} f(\mathbf{x}_i|\mathbf{a}(t), \mathbf{W}(t)) \right\rangle dt \\ &= \int_0^\infty \sum_{i=1}^n (y_i - u_i(t)) k(\mathbf{x}, \mathbf{x}_i, t)\, dt\end{aligned}$$

where we introduced the notation $k(\mathbf{x}, \mathbf{x}_i, t)$ for the scalar product of gradients. It's not hard to see that our argument could be applied almost as it is to show that for m large enough the vector $[k(\mathbf{x}, \mathbf{x}_1, t), \ldots, k(\mathbf{x}, \mathbf{x}_n, t)]^\top$ stays close to the vector $[k(\mathbf{x}, \mathbf{x}_1), \ldots, k(\mathbf{x}, \mathbf{x}_n)]^\top$ where k is the NTK. Plugging this approximation we get

$$\hat{f}(\mathbf{x}) = \sum_{i=1}^n k(\mathbf{x}, \mathbf{x}_i) \cdot \left(\int_0^\infty (y_i - u_i(t))\, dt \right)$$

Thus, to show that our predictions converge to the predictions of NTK we just need to show that the coefficients $\left\{ \int_0^\infty (y_i - u_i(t))\, dt \right\}_{i=1}^n$ converge to the coefficients learned by kernel regression with NTK. **Show that convergence for m going to infinity.**

Hint: look at the learned predictions in the data points. You don't have to be completely rigorous, for example you may assume $\mathbf{K}(t) = \mathbf{K}^\infty$ for large m and pull $\mathbf{K}^\infty$ out of the integral as we did above with $k(\mathbf{x}, \mathbf{x}_i)$.

(g) For the final part of this problem you will implement the key steps for neural tangent kernel regression and explore the behavior of the kernel compared to the network under gradient descent for varying hidden layer widths. **Complete the missing code and respond to the prompts for each subpart in the notebook.**

5 Double descent

As we increase the number of features, we increase the kinds of functions that our model can approximate well. This reduces the approximation error, and if the learning method can take advantage of it, could be a good thing. Unfortunately, there is a price that generally is paid by the learning process — as you have seen in earlier homeworks, the additional parameters can provide a place for noise/errors to enter the learned model and thereby degrade the prediction performance on unseen test data.

Traditionally, the test error is depicted as a ∪-shaped curve as a function of the number of parameters of the model, and one wants to find the "sweet spot" (the minimum of the test error). However,

in this course you have learned that the story is not so simple. You have seen how more features can have a regularizing effect — they can reduce the contamination effect — even as they allow you to exactly interpolate the training data. In earlier homeworks, you have used both a Fourier model and a Gaussian model to see this effect.

This homework problem is an homage to (see this celebrated paper) where the consequences of this underlying phenomenon were observed empirically and showed that this wasn't something that was restricted to deep neural nets, but was actually something ubiquitous. The paper showed that as we increase the number of parameters even further than the interpolation threshold (the point where the model is rich enough to exactly interpolate the data), the test error begins to decrease again in many problems — and in many cases, actually deceases below the best possible model with a few parameters. This phenomenon is called "Double Descent."

The aim of this problem is to show theoretically how double descent happens in a simple model where we add features one by one and perform linear regression for each number of features. This simple model is one that you have already seen in an earlier homework where we gave you a theoretical model that exhibits adversarial examples. You've done analyses in this style before — the goal here is to help you put the pieces together.

Suppose our signal is $\phi^*(\mathbf{x})$ where ϕ^* is a "master feature." We don't know the feature ϕ^*, but we have some intuition about it. Let's model it in the following way: suppose that we can construct features $\phi_i = \phi^* + \Delta_i$ s.t. the following conditions hold: if $\mathbf{x}$ is a point from the data distribution, then

- $\mathbb{E}_{\mathbf{x}}\phi^*(\mathbf{x})\Delta_i(\mathbf{x}) = \mathbb{E}_{\mathbf{x}}\Delta_i(\mathbf{x})\Delta_j(\mathbf{x}) = 0 \ \forall i \neq j$,
- $\mathbb{E}_{\mathbf{x}}\phi^*(\mathbf{x}) = 0, \quad \mathbb{E}_{\mathbf{x}}\phi^*(\mathbf{x})^2 = 1$,
- $\mathbb{E}_{\mathbf{x}}\Delta_i(\mathbf{x}) = 0, \quad \mathbb{E}_{\mathbf{x}}\Delta_i(\mathbf{x})^2 = \sigma^2$,
- $(\phi^*(\mathbf{x}), \Delta_1(\mathbf{x}), \Delta_2(\mathbf{x}), \dots)$ are jointly Gaussian (note that it implies independence since correlations are zero).

Notice how this model can be thought of as posessing a single dimensional latent space (using the style of thinking that you have also seen in earlier homeworks) where each visible feature is a noisy view of that latent space.

We want to estimate the generalization error of linear regression if we create m features. If $m < n$ we do least squares, if $m \geq n$ we find the min norm interpolating solution. Let's denote the corresponding solution as $\hat{\mathbf{w}}_{\mathbf{y}}$ (note that we emphasize the dependence on $\mathbf{y}$ here).

Suppose we find weighs $\{w_i\}_{i=1}^m$. The generalization error will be

$$R(\mathbf{w}) := \left(1 - \sum_{i=1}^m w_i\right)^2 + \sigma^2 \sum_{i=1}^m w_i^2.$$

Let $O \in \mathbb{R}^{m \times m}$ be a unitary matrix (i.e. a matrix with orthonormal columns) whose first row is $[1/\sqrt{m}, \dots, 1/\sqrt{m}]$. We can change coordinates in the weight space in the following way:

$$\mathbf{X} \to \mathbf{X}O^\top =: [\sqrt{m}\phi^*(\mathbf{X}) + \mathbf{c}_1, \mathbf{c}_2, \dots, \mathbf{c}_m] \quad \mathbf{w} \to O\mathbf{w} =: \boldsymbol{\theta}$$

where $[\mathbf{c}_1, \ldots, \mathbf{c}_m]$ a the matrix with i.i.d. entries from $\mathcal{N}(0, \sigma^2)$.

The generalization error becomes

$$L(\boldsymbol{\theta}) := R(O\mathbf{w}) = (1 - \sqrt{m}\theta_1)^2 + \sigma^2 \|\boldsymbol{\theta}\|^2,$$

where θ_1 is the first coordinate of $\boldsymbol{\theta}$.

Notice that the last group of terms are contamination terms in spirit and the first term is fundamentally about how much of the true signal survives. However, the scenario here has an approximation-error dimension as well since the true pattern/signal here cannot actually be represented accurately using a finite number of features. The first feature in rotated coordinates is becoming an ever-better approximation to the true pattern as we add more features, and the approximation error goes to zero as the number of features goes to infinity. It is this improved approximation error that permits us the possibility of doing better with lots of features than we could possibly do with a few features.

To allow us to get a better handle on the contamination, denote by $\boldsymbol{\theta_{-1}}$ the vector comprised of the rest of the components after the first one and let

$$\mathbf{C}_{-1} := [\mathbf{c}_2, \ldots, \mathbf{c_m}]$$

This problem has many parts to help you walk through the derivation in detail, but it is good to remember what you already know from earlier homeworks. As the number of features increases, the effective "weight" (remember the meta-learning problem on the previous homework) on the first feature in rotated coordinates is increasing. This is going to make the min-norm interpolating solution put a lot of weight on that relative to its aliases — this is what is going to allow the signal to survive, even as we have to learn from a few samples. Meanwhile, the contaminating aliases are going to become less and less contaminating to test points as we add more features, in the same way that you have seen in earlier homeworks about Fourier-based models as well as Gaussian-based models.

(a) Let us give the first feature in the rotated coordinates its own name and notation: $\boldsymbol{\eta} := \sqrt{m}\phi^*(\mathbf{X}) + \mathbf{c}_1$. This is where we want a lot of weight to end up in the learned solution. **Prove the following formula for the solution**:

$$(\hat{\theta}_1)_{\mathbf{y}} = \begin{cases} \mathbf{y}^\top \boldsymbol{\eta}_\perp / \|\boldsymbol{\eta}_\perp\|^2, & m \le n \\ (\mathbf{C}_{-1}^\dagger \boldsymbol{\eta})^\top \mathbf{C}_{-1}^\dagger \mathbf{y} / (1 + \|\mathbf{C}_{-1}^\dagger \boldsymbol{\eta}\|^2). & \text{else.} \end{cases}$$

$$(\hat{\boldsymbol{\theta}}_{-1})_{\mathbf{y}} = \begin{cases} \mathbf{C}_{-1}^\dagger (\mathbf{I}_m - \boldsymbol{\eta}\boldsymbol{\eta}_\perp^\top / \|\boldsymbol{\eta}_\perp\|^2)\mathbf{y}, & m \le n \\ \left(\mathbf{I}_m - \mathbf{C}_{-1}^\dagger \boldsymbol{\eta} (\mathbf{C}_{-1}^\dagger \boldsymbol{\eta})^\top / (1 + \|\mathbf{C}_{-1}^\dagger \boldsymbol{\eta}\|^2)\right) \mathbf{C}_{-1}^\dagger \mathbf{y}, & \text{else,} \end{cases}$$

where $\boldsymbol{\eta}_\perp$ is the projection of $\boldsymbol{\eta}$ on the space orthogonal to the column span of $\mathbf{C}_{-1}$.

Hint 1: the least squares solution finds the linear combination of columns that is closest to the target $\mathbf{y}$. *Therefore for* $m < n$, *we know* $(\hat{\theta}_1)_{\mathbf{y}} \boldsymbol{\eta}$ *is the dilation of* $\boldsymbol{\eta}$ *that is closest to the projection of* $\mathbf{y}$ *on the orthogonal complement to the span of columns of* $\mathbf{C}$.

Hint 2: $(\hat{\boldsymbol{\theta}}_{-1})_{\mathbf{y}} = \mathbf{C}_{-1}^\dagger (\mathbf{y} - \hat{\theta}_1 \boldsymbol{\eta})$.

Hint 3: for $m > n$ plug the equation from the previous hint into the definition of the min norm interpolating solution.

(b) We can already see that relation between $\mathbf{y}$ and $\boldsymbol{\eta}$ is somewhat inconvenient: $\boldsymbol{\eta} = \sqrt{m}\mathbf{y} + \mathbf{c}_1$ so it is easy to express $\boldsymbol{\eta}$ in terms of $\mathbf{y}$, but the equations depend on $\boldsymbol{\eta}$ in a complicated way, so we would like to leave $\boldsymbol{\eta}$ as a free variable. Fortunately, because we assumed that the distributions are jointly Gaussian this dependence can be simplified too! **Show that**

$$\mathbf{y} = \frac{\sqrt{m}}{m+\sigma^2}\boldsymbol{\eta} + \sqrt{\frac{\sigma^2}{m+\sigma^2}}\boldsymbol{\varepsilon},$$

where $\boldsymbol{\varepsilon} \in \mathcal{N}(\mathbf{0}, \mathbf{I}_n)$ — independent of $\boldsymbol{\eta}$ and $\mathbf{C}_{-1}$.

This decomposition is great because it lets us pretend that the y is actually a noisy version of a scaled first feature — bringing us closer to cases that we have studied earlier. Notice how the noise gets smaller as m.

Hint: use the conditional distribution formula for multivariate Gaussians.

(c) Treating $\boldsymbol{\varepsilon}$ as something like noise, we can derive a bias-variance style decomposition:

$$\begin{aligned} L(\boldsymbol{\theta}) =& (1 - \sqrt{m}\theta_1)^2 + \sigma^2\|\boldsymbol{\theta}\|^2 \\ \mathbb{E}_{\boldsymbol{\varepsilon}} L(\hat{\boldsymbol{\theta}}_{\mathbf{y}}) =& \mathbb{E}_{\boldsymbol{\varepsilon}} L\left(\frac{\sqrt{m}}{m+\sigma^2}\hat{\boldsymbol{\theta}}_{\boldsymbol{\eta}} + \sqrt{\frac{\sigma^2}{m+\sigma^2}}\hat{\boldsymbol{\theta}}_{\boldsymbol{\varepsilon}}\right) \\ =& L\left(\frac{\sqrt{m}}{m+\sigma^2}\hat{\boldsymbol{\theta}}_{\boldsymbol{\eta}}\right) + \frac{\sigma^2}{m+\sigma^2}\mathbb{E}_{\boldsymbol{\varepsilon}}\left[(\hat{\theta}_1)_{\boldsymbol{\varepsilon}}^2 m + \sigma^2\|\hat{\boldsymbol{\theta}}_{\boldsymbol{\varepsilon}}\|\right] \end{aligned}$$

Here, notice that we have introduced this subscript notation for convenience on the estimated parameters in the rotated space — this refers to what happens if we run the relevant least-squares/min-norm estimator using the subscript as the target.

Now we can compute each part of the error separately. Let's start with the bias (the part that doesn't depend on the realization of the "noise" $\boldsymbol{\varepsilon}$): **show that**

$$(\hat{\theta}_1)_{\boldsymbol{\eta}} = \begin{cases} 1, & m \le n \\ 1 - \frac{1}{1+\|\mathbf{C}_{-1}^{\dagger}\boldsymbol{\eta}\|^2}. & \text{else.} \end{cases}$$

$$(\hat{\boldsymbol{\theta}}_{-1})_{\boldsymbol{\eta}} = \begin{cases} \mathbf{0}, & m \le n \\ \frac{1}{1+\|\mathbf{C}_{-1}^{\dagger}\boldsymbol{\eta}\|^2}\mathbf{C}_{-1}^{\dagger}\boldsymbol{\eta}, & \text{else.} \end{cases}$$

(Conceptual Hint: When $m \le n$, there is no alias for $\boldsymbol{\eta}$ among the features. When $m > n$, what is the relationship of $\mathbf{C}_{-1}^{\dagger}\boldsymbol{\eta}$ to an alias for $\boldsymbol{\eta}$?)

(d) Now **plug in the result of the previous part to obtain**

$$L\left(\frac{\sqrt{m}}{m+\sigma^2}\hat{\boldsymbol{\theta}}_{\boldsymbol{\eta}}\right) = \begin{cases} \frac{\sigma^2}{m+\sigma^2}, & m \le n \\ \left(\frac{\sigma^2}{m+\sigma^2} + \frac{m}{m+\sigma^2}\frac{1}{1+\|\mathbf{C}_{-1}^{\dagger}\boldsymbol{\eta}\|^2}\right)^2 + \frac{\sigma^2 m}{(m+\sigma^2)^2}\frac{\|\mathbf{C}_{-1}^{\dagger}\boldsymbol{\eta}\|^2}{1+\|\mathbf{C}_{-1}^{\dagger}\boldsymbol{\eta}\|^2} & , m > n \end{cases}$$

(e) Now let's deal with the "variance" part $\frac{\sigma^2}{m+\sigma^2}\mathbb{E}_{\boldsymbol{\varepsilon}}\left[(\hat{\theta}_1)_{\boldsymbol{\varepsilon}}^2 m + \sigma^2\|\hat{\boldsymbol{\theta}}_{\boldsymbol{\varepsilon}}\|^2\right]$ (don't forget that $\boldsymbol{\varepsilon}$ is a vector with i.i.d. standard normal entries, that doesn't depend on $\boldsymbol{\eta}$ or $\mathbf{C}_{-1}$.)

The variance term turns out to be a bit more complicated then the bias term, so we start with the following technical step: **show that**

$$\mathbb{E}_{\boldsymbol{\varepsilon}}(\hat{\theta}_1)_{\boldsymbol{\varepsilon}}^2 = \begin{cases} 1/\|\boldsymbol{\eta}_{\perp}\|^2, & m \le n \\ \|(\mathbf{C}_{-1}^{\dagger})^{\top}\mathbf{C}_{-1}^{\dagger}\boldsymbol{\eta}\|^2/(1+\|\mathbf{C}_{-1}^{\dagger}\boldsymbol{\eta}\|^2)^2. & \textbf{else.} \end{cases}$$

$$\mathbb{E}_{\boldsymbol{\varepsilon}}\|(\hat{\boldsymbol{\theta}}_{-1})_{\boldsymbol{\varepsilon}}\|^2 = \begin{cases} \mathrm{tr}\left((\mathbf{C}_{-1}^{\dagger})^{\top}\mathbf{C}_{-1}^{\dagger}(\mathbf{I}_m - \boldsymbol{\eta}_{\|}\boldsymbol{\eta}_{\perp}^{\top}/\|\boldsymbol{\eta}_{\perp}\|^2 - \boldsymbol{\eta}_{\perp}\boldsymbol{\eta}_{\|}^{\top}/\|\boldsymbol{\eta}_{\perp}\|^2 + \boldsymbol{\eta}_{\|}\boldsymbol{\eta}_{\|}^{\top}/\|\boldsymbol{\eta}_{\perp}\|^2)^{\top}\right), & m \le n \\ \mathrm{tr}\left(\mathbf{C}_{-1}^{\dagger}\left(\mathbf{C}_{-1}^{\dagger}\right)^{\top}\right) - \frac{2+\|\mathbf{C}_{-1}^{\dagger}\boldsymbol{\eta}\|^2}{(1+\|\mathbf{C}_{-1}^{\dagger}\boldsymbol{\eta}\|^2)^2}\|(\mathbf{C}_{-1}^{\dagger})^{\top}\mathbf{C}_{-1}^{\dagger}\boldsymbol{\eta}\|^2, & \textbf{else,} \end{cases}$$

where we denoted $\boldsymbol{\eta}_{\|} := \boldsymbol{\eta} - \boldsymbol{\eta}_{\perp}$. The formula may look bulky but the computation is very straightforward.

Hint 1: if some matrix $\mathbf{A}$ is independent from $\boldsymbol{\varepsilon}$, then $\mathbb{E}_{\boldsymbol{\varepsilon}}\|\mathbf{A}\boldsymbol{\varepsilon}\|^2 = \|\mathbf{A}\|_F = \mathrm{tr}\left(\mathbf{A}\mathbf{A}^{\top}\right) = \mathrm{tr}\left(\mathbf{A}^{\top}\mathbf{A}\right)$.

Hint 2: explain why $\mathbf{C}_{-1}^{\dagger}\boldsymbol{\eta} = \mathbf{C}_{-1}^{\dagger}\boldsymbol{\eta}_{\|}$

Hint 3: $\mathrm{tr}\left(\mathbf{A}\mathbf{B}^{\top}\right) = \mathrm{tr}\left(\mathbf{B}^{\top}\mathbf{A}\right)$ for any matrices $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{a\times b}$ for any natural a, b.

(f) We will not ask you to finish the computation of the generalization error for the case $m > n$. The steps of that computation are exactly the same as the steps for the case $m \le n$, but it just turns out to be (substantially) more bulky. So, from now on you will only deal with the underparametrized case $m \le n$.

Moreover, even in this case we'll do some work for you: since $\boldsymbol{\eta}_{\|}$ is zero-mean and independent of $\boldsymbol{\eta}_{\perp}$ and $\mathbf{C}_{-1}$, the expectation of the second and the third term is equal to zero, so we can remove them without changing the expectation:

$$\begin{aligned} &\mathbb{E}\left[\mathrm{tr}\left((\mathbf{C}_{-1}^{\dagger})^{\top}\mathbf{C}_{-1}^{\dagger}(\mathbf{I}_m - \boldsymbol{\eta}_{\|}\boldsymbol{\eta}_{\perp}^{\top}/\|\boldsymbol{\eta}_{\perp}\|^2 - \boldsymbol{\eta}_{\perp}\boldsymbol{\eta}_{\|}^{\top}/\|\boldsymbol{\eta}_{\perp}\|^2 + \boldsymbol{\eta}_{\|}\boldsymbol{\eta}_{\|}^{\top}/\|\boldsymbol{\eta}_{\perp}\|^2)^{\top}\right)\right] \\ =&\mathbb{E}\left[\mathrm{tr}\left((\mathbf{C}_{-1}^{\dagger})^{\top}\mathbf{C}_{-1}^{\dagger}(\mathbf{I}_m + \boldsymbol{\eta}_{\|}\boldsymbol{\eta}_{\|}^{\top}/\|\boldsymbol{\eta}_{\perp}\|^2)^{\top}\right)\right] \\ =&\mathbb{E}\left[\mathrm{tr}\left(\mathbf{C}_{-1}^{\dagger}\left(\mathbf{C}_{-1}^{\dagger}\right)^{\top}\right) + \|\mathbf{C}_{-1}^{\dagger}\boldsymbol{\eta}_{\|}\|^2/\|\boldsymbol{\eta}_{\perp}\|^2\right]. \end{aligned}$$

So overall for the expectation of the variance term we have

$$\frac{\sigma^2}{m+\sigma^2}\mathbb{E}\left[(\hat{\theta}_1)_{\boldsymbol{\varepsilon}}^2 m + \sigma^2\|\hat{\boldsymbol{\theta}}_{\boldsymbol{\varepsilon}}\|^2\right]$$

$$= \begin{cases} \frac{\sigma^2}{m+\sigma^2}\mathbb{E}\left[\sigma^2 \operatorname{tr}\left(\mathbf{C}_{-1}^{\dagger}\left(\mathbf{C}_{-1}^{\dagger}\right)^{\top}\right) + (\sigma^2\|\mathbf{C}_{-1}^{\dagger}\boldsymbol{\eta}_{\|}\|^2 + \sigma^2 + m)/\|\boldsymbol{\eta}_{\perp}\|^2\right], & m \le n, \\ \frac{\sigma^2}{m+\sigma^2}\mathbb{E}\left[\sigma^2 \operatorname{tr}\left(\mathbf{C}_{-1}^{\dagger}\left(\mathbf{C}_{-1}^{\dagger}\right)^{\top}\right) + \frac{m-\sigma^2(1+\|\mathbf{C}_{-1}^{\dagger}\boldsymbol{\eta}\|^2)}{(1+\|\mathbf{C}_{-1}^{\dagger}\boldsymbol{\eta}\|^2)^2}\|(\mathbf{C}_{-1}^{\dagger})^{\top}\mathbf{C}_{-1}^{\dagger}\boldsymbol{\eta}\|^2\right], & m > n. \end{cases}$$

Now **use Lemma 1 to show that for** $m \le n$

$$\mathbb{E}\left[(\sigma^2\|\mathbf{C}_{-1}^{\dagger}\boldsymbol{\eta}_{\|}\|^2 + \sigma^2 + m)/\|\boldsymbol{\eta}_{\perp}\|^2\right] = \begin{cases} +\infty, & n - m + 1 \le 2 \\ \frac{1}{(m+\sigma^2)(n-m-1)}\mathbb{E}\left[(\sigma^2\|\mathbf{C}_{-1}^{\dagger}\boldsymbol{\eta}\|^2 + \sigma^2 + m)\right], & n - m + 1 > 2, \end{cases}$$

Lemma 1. *Suppose a r.v.* ξ *has chi-squared distribution with* d *degrees of freedom. Then*

$$\mathbb{E}[\xi^{-1}] = \begin{cases} +\infty, & d \le 2 \\ \frac{1}{d-2}, & d > 2. \end{cases}$$

(g) At this point we've obtained expressions for the generalization error that only depend on 4 random quantities: $\mathbb{E}\operatorname{tr}\left((\mathbf{C}_{-1}^{\dagger})^{\top}\mathbf{C}_{-1}^{\dagger}\right)$, $\mathbb{E}\|(\mathbf{C}_{-1}^{\dagger})^{\top}\mathbf{C}_{-1}^{\dagger}\|_F^2$,, $\mathbb{E}\|\mathbf{C}_{-1}^{\dagger}\boldsymbol{\eta}\|^2$, $\mathbb{E}\|(\mathbf{C}_{-1}^{\dagger})^{\top}\mathbf{C}_{-1}^{\dagger}\boldsymbol{\eta}\|^2$. To simplify the last step in estimation of the expected error we are going to cheat a little bit: instead of computing true expectations we will simply substitute those 4 quantities with their expectations in our equations.

Hint: don't forget that $\boldsymbol{\eta}$ *is gaussian and independent of* $\mathbf{C}$.

Use Lemma 2 to compute the following quantities:

$$\mathbb{E}\operatorname{tr}\left((\mathbf{C}_{-1}^{\dagger})^{\top}\mathbf{C}_{-1}^{\dagger}\right), \quad \mathbb{E}\|(\mathbf{C}_{-1}^{\dagger})^{\top}\mathbf{C}_{-1}^{\dagger}\|_F^2, \quad \mathbb{E}\|\mathbf{C}_{-1}^{\dagger}\boldsymbol{\eta}\|^2, \quad \mathbb{E}\|(\mathbf{C}_{-1}^{\dagger})^{\top}\mathbf{C}_{-1}^{\dagger}\boldsymbol{\eta}\|^2.$$

Lemma 2. *Suppose* $\mathbf{G}$ *is a* $a \times b$ *matrix with i.i.d. standard normal entries and* $a \ge b$. *Then*

$$\mathbb{E}\operatorname{tr}\left(\mathbf{G}^{\dagger}(\mathbf{G}^{\dagger})^{\top}\right) = \begin{cases} +\infty, & a \in \{b, b+1\}, \\ \frac{b}{a-b-1}, & a > b+1, \end{cases}$$

and

$$\mathbb{E}\|\mathbf{G}^{\dagger}(\mathbf{G}^{\dagger})^{\top}\|_F^2 = \begin{cases} +\infty, & a \le b+3, \\ \frac{(a-1)b}{(a-b)(a-b-1)(a-b-3)}, & a > b+3. \end{cases}$$

(h) Finally, we are ready to compute the answer. As stated before, plug in the following in the formulas:

$$\|\mathbf{C}_{-1}^{\dagger}\boldsymbol{\eta}\|^2 \approx \mathbb{E}\|\mathbf{C}_{-1}^{\dagger}\boldsymbol{\eta}\|^2,$$
$$\|(\mathbf{C}_{-1}^{\dagger})^{\top}\mathbf{C}_{-1}^{\dagger}\boldsymbol{\eta}\|^2 \approx \mathbb{E}\|(\mathbf{C}_{-1}^{\dagger})^{\top}\mathbf{C}_{-1}^{\dagger}\boldsymbol{\eta}\|^2$$

Moreover, let's assume that we are in the regime when $m, n, |m-n| \gg 1$, so the following also holds:

$$|m-n| \approx |m-n| - 1 \approx |m-n| - 2 \approx |m-n| - 3,$$

$$m \approx m-1 \approx m-2$$

etc.

(note that σ^2 is not a constant, so you cannot assume $m + \sigma^2 \approx m$).

Combine the expressions for bias and variance and plug in our approximations to show that for $m \leq n$

$$\mathbb{E}L(\hat{\boldsymbol{\theta}}_{\mathbf{y}}) \approx \frac{n\sigma^2}{(m+\sigma^2)(n-m)}.$$

As promised, we don't ask you to do the same computation for $m > n$. However, we provide you with the answer: for $m > n$

$$EL(\hat{\boldsymbol{\theta}}_{\mathbf{y}}) \approx \frac{\sigma^2(n^2+m\sigma^2)}{(n+\sigma^2)^2(m-n)}.$$

(i) Look once again at our result:

$$\mathbb{E}L(\hat{\boldsymbol{\theta}}_{\mathbf{y}}) \approx \begin{cases} \frac{n\sigma^2}{(m+\sigma^2)(n-m)}, & m \leq n, \\ \frac{\sigma^2(n^2+m\sigma^2)}{(n+\sigma^2)^2(m-n)}, & m > n. \end{cases}$$

Answer the following questions qualitatively:

- What happens if m is close to n? What happens to the training error in this case?
- Suppose n and σ are constant. What is the shape of the curve if you plot $\mathbb{E}L(\hat{\boldsymbol{\theta}}_{\mathbf{y}})$ against m? Do we always get double descent?
- Suppose n and σ are fixed. What is the smallest achievable error if $m \leq n$? What if $m > n$? (don't forget that σ^2 may be larger than n)

(j) **Compare your theoretical result to the simulation result in the Jupyter notebook. Report your observations.**

(k) (Optional) **Copy-paste-and-modify the simulation in the Jupyter notebook to plot what happens if you hold the number of features constant at something moderately large and sweep the number of training points from 0 to something large. Report your observations.**

This should remind you of plots you saw in the meta-learning problem as well. The fact that more high-quality training data can actually at times make performance worse was not widely appreciated until recently, but is important to be aware of.

The second part on this homework is designed to further exercise skills and concepts by giving you past exam problems to look at. You can consider all these problems optional.

6 Short answer (5 parts, Est. 14 minutes)

For each of the following questions, write at most one sentence in answer.
If you write more than one sentence, we will only grade the first sentence.

(a) (Est. 4 minutes) For the following loss functions $\ell : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, **write a YES or a NO in the table** for whether it

(1) assigns a loss of 0 to any prediction with the right sign, i.e., for any $\hat{y}, y_* \in \mathbb{R}$,

$$\ell(\hat{y}, y_*) = 0 \text{ if } \operatorname{sign}(\hat{y}) = \operatorname{sign}(y_*) ,$$

(2) is bounded, i.e. $\max_{\hat{y} \in \mathbb{R}, y_* \in \mathbb{R}} \left|\ell(\hat{y}, y_*)\right| \leq C$ for some finite constant $C \geq 0$,

(3) is convex on $\mathbb{R}$ in its first argument, i.e. $\hat{y} \mapsto \ell(\hat{y}, y_*)$ is convex on $\mathbb{R}$,

(4) is differentiable everywhere on $\mathbb{R}$ in its first argument, i.e. $\hat{y} \mapsto \ell(\hat{y}, y_*)$ is differentiable on $\mathbb{R}$.

The losses are defined as follows (recall that both $\hat{y}$ and y_* are scalars):

- 0-1 loss: $\ell(\hat{y}, y_*) = \begin{cases} 1 & \text{if } \operatorname{sign}(\hat{y}) \neq \operatorname{sign}(y_*) \\ 0 & \text{otherwise} \end{cases}$
- squared loss: $\ell(\hat{y}, y_*) = (\hat{y} - y_*)^2$
- absolute loss: $\ell(\hat{y}, y_*) = |\hat{y} - y_*|$
- sigmoid loss: $\ell(\hat{y}, y_*) = \frac{1}{1+\exp(-\hat{y} \cdot y_*)}$
- hinge loss: $\ell(\hat{y}, y_*) = \max\{0, 1 - \hat{y} \cdot y_*\}$

The first column of answers is provided for you as an example.

	0-1 loss	squared loss	absolute loss	sigmoid loss	hinge loss
(1) sign	YES				
(2) bounded	YES				
(3) convex	NO				
(4) everywhere differentiable	NO				

(b) (Est. 2 minutes) **Give an example of two random variables $X, Y \in \mathbb{R}$ that are uncorrelated, but *not* independent.**

(c) (Est. 2 minutes) **Give a necessary and sufficient condition on a matrix $\mathbf{X} \in \mathbb{R}^{n\times n}$, under which we can write $\mathbf{X} = \mathbf{U}\mathbf{D}\mathbf{U}^\top$ for $\mathbf{D} \in \mathbb{R}^{n\times n}$ diagonal and $\mathbf{U} \in \mathbb{R}^{n\times n}$ orthonormal.**

(d) (Est. 2 minutes) Recall that when describing the optimal regression weights $\mathbf{w}$ using ridge regression, the pseudo-inverse, and PCA, we can write the weights using the SVD of the feature matrix $\mathbf{X}$ as follows, with rescaling on the singular values s_i according to the spectral function $\rho(s_i)$:

$$\mathbf{w} = \sum_{i=1}^{n} \rho(s_i)\mathbf{v}_i\mathbf{u}_i^\top y$$

Figure 1 plots the spectral function $\rho(s_i)$ for (1) Pseudo-Inverse, (2) PCA, and (3) Ridge Regression. **In the boxes on Figure 1, label the graphs of the spectral function with the method that they correspond to**.

(e) (Est. 4 minutes) Assume that $\mathbf{X} \in \mathbb{R}^{n\times d_1}$ and $\mathbf{Y} \in \mathbb{R}^{n\times d_2}$ are two full column-rank datasets ($n \geq d_1, n \geq d_2$). Recall that in the computations for canonical correlation analysis (CCA), we would like to maximize the correlation coefficient,

$$\rho_{\text{cca}}(\mathbf{X}, \mathbf{Y}) := \max_{\substack{\mathbf{u}\in\mathbb{R}^{d_1},\, \mathbf{v}\in\mathbb{R}^{d_2} \\ \mathbf{u}\neq 0, \mathbf{v}\neq 0}} \frac{\mathbf{v}^\top\mathbf{X}^\top\mathbf{Y}\mathbf{u}}{\sqrt{\mathbf{v}^\top\mathbf{X}^\top\mathbf{X}\mathbf{v}\cdot\mathbf{u}^\top\mathbf{Y}^\top\mathbf{Y}\mathbf{u}}}.$$

Now consider a full column-rank matrix $\mathbf{Z} \in \mathbb{R}^{n\times d}$ (i.e. $\text{rank}(\mathbf{Z}) = d$). **Compute the following correlation coefficients, and write down any corresponding pair of vectors $\mathbf{u}$, $\mathbf{v} \in \mathbb{R}^d$ that achieves the maximum.**

(1) $\rho_{\text{cca}}(\mathbf{Z}, \mathbf{Z})$

(2) $\rho_{\text{cca}}(\mathbf{Z}, -\mathbf{Z})$

7 Gaussian Kernels (3 parts, Est. 12 minutes)

In this question, we will look at training a binary classifier with a Gaussian kernel. Specifically given a labelled dataset $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^n \subseteq \mathbb{R}^d \times \{\pm 1\}$ and a kernel function $k(\mathbf{x}_1, \mathbf{x}_2)$, we consider

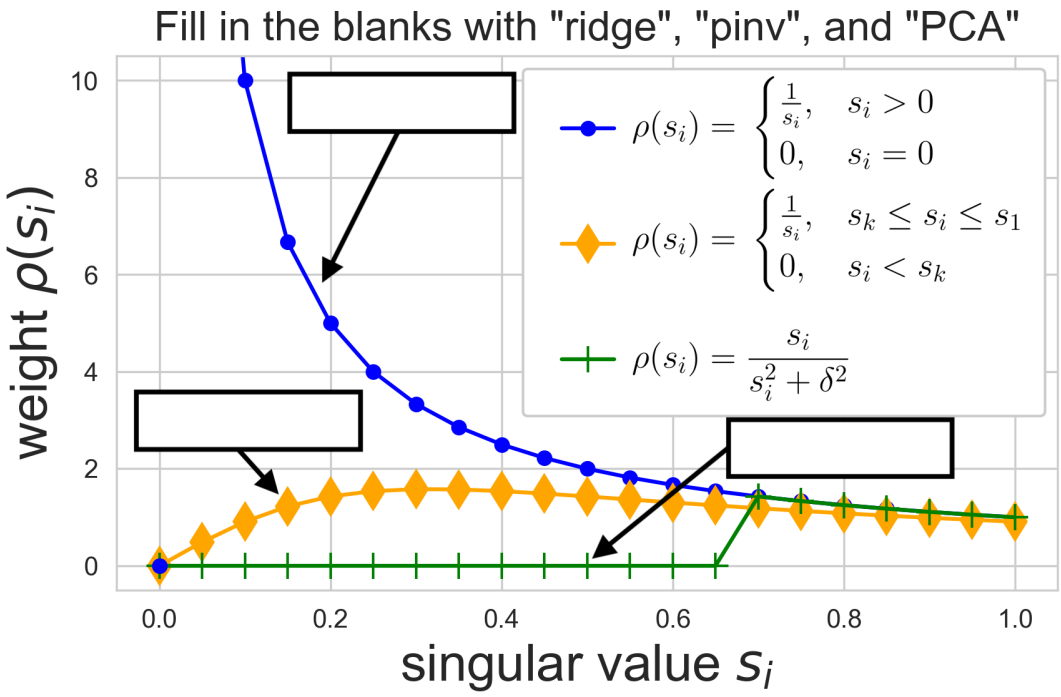


Figure 1: Fill in the boxes with "ridge", "pinv", and "PCA."

classifiers of the form:

$$\widehat{f}(\mathbf{x}) = \text{sign}\left(\sum_{i=1}^{n} \alpha_i k(\mathbf{x}_i, \mathbf{x})\right),$$

where we define sign(u) to be 1 if $u \geq 0$ or -1 if $u < 0$. In order to choose the weights $\alpha_i, i = 1, ..., n$, we will consider the least-squares problem:

$$\boldsymbol{\alpha} \in \arg\min_{\boldsymbol{\alpha} \in \mathbb{R}^n} \|\mathbf{K}\boldsymbol{\alpha} - \mathbf{y}\|_2^2, \tag{11}$$

where $\mathbf{K} = (k(\mathbf{x}_i, \mathbf{x}_j))_{i=1,j=1}^n$ is the kernel matrix and $\mathbf{y} = (y_1, ..., y_n)$ is the vector of labels. We will work with the Gaussian kernel. Recall that the Gaussian kernel with bandwidth $\sigma > 0$ is defined as:

$$k(\mathbf{x}_i, \mathbf{x}_j) := \exp\left(-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|_2^2}{2\sigma^2}\right).$$

(a) (Est. 4 minutes) When the bandwidth parameter $\sigma \to 0$, observe that the off-diagonal entries of the kernel matrix $\mathbf{K}$ tend also to zero. Consider a two sample dataset S (i.e. $n = 2$) with $(x_1, y_1) = (1, 1)$ and $(x_2, y_2) = (-1, -1)$. **Assuming that as $\sigma \to 0$ the off-diagonal entries of $\mathbf{K}$ are equal to zero (and the diagonal entries unmodified), what is the optimal solution of α for the optimization problem (11) and what is the resulting classifier $\widehat{f}(x)$?**

(b) (Est. 4 minutes) Now we consider the regime when the bandwidth parameter $\sigma \to +\infty$. Observe in this regime, the off-diagonal entries of the kernel matrix $\mathbf{K}$ tend to one. Given a

dataset S, suppose we solve the optimization problem (11) with all the off-diagonal entries of $\mathbf{K}$ equal to one (and the diagonal entries unmodified). **Prove that if the number of $+1$ labels in S equals the number of -1 labels in S, then $\alpha = \mathbf{0}$ is an optimal solution of (11). What is the resulting classifier $\widehat{f}(x)$?**

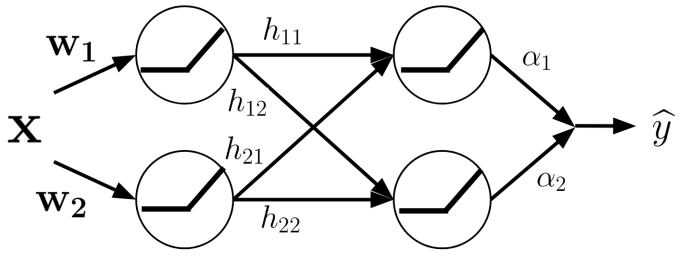
(c) (Est. 4 minutes) Now we consider the regime when the bandwidth parameter is large but finite. Consider again the two sample dataset S with $(x_1, y_1) = (1, 1)$ and $(x_2, y_2) = (-1, -1)$. When $\sigma \gg 1$, we can approximate $k(x_1, x_2) \approx 1 + \frac{x_1 x_2}{2\sigma^2}$. **Show that the solution of the optimization problem (11) with the kernel $k_a(x_1, x_2) = 1 + \frac{x_1 x_2}{2\sigma^2}$ is given by $\alpha = (\sigma^2, -\sigma^2)$. What is the resulting classifier $\widehat{f}(x)$?**

Hint: The inverse of a 2×2 matrix is given by the formula $\begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{ad-bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$.

8 Neural Networks (3 parts, Est. 12 minutes)

We consider the following neural network structure, where:

$$\mathbf{x},\ \mathbf{w}_1,\ \mathbf{w}_2, \in \mathbb{R}^7, \quad \text{and}$$
$$\alpha_1,\ \alpha_2,\ h_{11},\ h_{12},\ h_{21},\ h_{22},\ \widehat{y} \in \mathbb{R}\,.$$



The output of this network $\widehat{y}(\mathbf{x})$ is given by $\widehat{y} = \alpha_1 v_1 + \alpha_2 v_2$, where

$$\begin{aligned} u_1 &= \max\{0, \mathbf{w}_1^\top \mathbf{x}\} \\ u_2 &= \max\{0, \mathbf{w}_2^\top \mathbf{x}\} \\ v_1 &= \max\{0, h_{11} u_1 + h_{21} u_2\} \\ v_2 &= \max\{0, h_{12} u_1 + h_{22} u_2\}\,. \end{aligned}$$

For a single data pair $(\mathbf{x}, y)$ we consider the loss function $\ell(\mathbf{x}, y) = \frac{1}{2}(\widehat{y}(\mathbf{x}) - y)^2$.

(a) (Est. 4 minutes) The average loss over an arbitrary set of labeled points $\{\mathbf{x}_i, y_i\}$ is given by $\frac{1}{n}\sum_{i=1}^n \ell(\mathbf{x}_i, y_i)$. In general, this function does not have a unique minimizer. **Give a specific reason why this is true.**

(b) (Est. 4 minutes) **Compute the partial derivative $\frac{\partial}{\partial h_{ij}} \ell(\mathbf{x}, y)$. Your answer may be in terms of intermediate variables but must not contain any partial derivatives.** Please give an answer

in terms of a general $\frac{\partial}{\partial h_{ij}}$ and do not simply give an explicit enumeration of the four different $\frac{\partial}{\partial h_{ij}}$ partial derivatives.

Note: You may take the derivative of a ReLu at zero to be zero.

(c) (Est. 4 minutes) Assuming $n > 2$, suppose we train this network with stochastic gradient descent using minibatches of size 2 and step size η. **Write down a procedure to update h_{ij} at time t.** (You may write it in terms of an unexpanded $\frac{\partial \ell}{\partial h_{ij}}$, i.e. you do not need to rely on a correct part (b).)

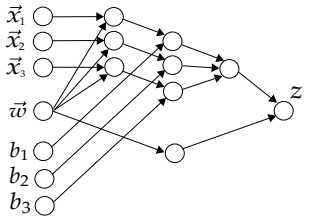
9 Multiple Choice (Est. 10 minutes)

For the following multiple choice questions, select all that apply.

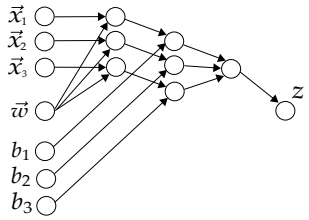
(a) **Which of the following computation graphs describes the function**

$$z = \sum_{i=1}^{n} \phi(\mathbf{x}_i^\top \mathbf{w} + b_i) + \|\mathbf{w}\|_2^2$$

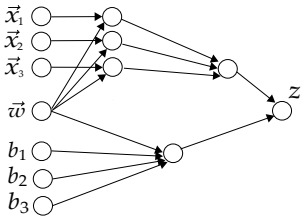
where $z \in \mathbb{R}$, $\mathbf{w}, \mathbf{x}_i \in \mathbb{R}^d$, and $b_i \in \mathbb{R}$?



(A)



(B)



(B)

(a) ◯ computation graph (A)

(b) ◯ computation graph (B)

(c) ◯ computation graph (C)

(b) Consider a distribution $\mathcal{D}$ on labelled pairs $(x, y) \in \{\pm 1\} \times \{\pm 1\}$ where the conditional distribution of y given x is:

$$y = \begin{cases} x & \text{with probability } \alpha \\ -x & \text{with probability } 1 - \alpha \end{cases}.$$

The $0/1$ population risk of a classifier the form $\text{sign}(w \cdot x)$, $w \in \mathbb{R}$ is defined as:

$$R_{\mathcal{D}}[w] = \mathbb{E}_{(x,y)\sim\mathcal{D}} \left[\mathbb{I}(\text{sign}(w \cdot x) \neq y) \right].$$

What is $\min_{w\in\mathbb{R}} R_{\mathcal{D}}[w]$**?**

(a) $1/2$.

(b) $\max\{\alpha, 1-\alpha\}$.

(c) $\min\{\alpha, 1-\alpha\}$.

(d) Depends on the marginal distribution of x.

(c) Using a fully connected neural net with one hidden layer and ReLU activations, **what is the minimum number of hidden nodes needed to achieve perfect classification, with respect to the hinge loss, of the data shown in Figure 2** (the plus signs denote data points labelled $+1$ while the minus signs denote data points labelled -1)? Mathematically, the neural network we are considering has the form

$$f(\mathbf{x}) = \beta + \sum_{j=1}^{N} \alpha_j \max\{\langle \mathbf{w}_j, \mathbf{x} \rangle + b_j, 0\},$$

where β, α_j, $\mathbf{w}_j$, and b_j are the parameters of the network.

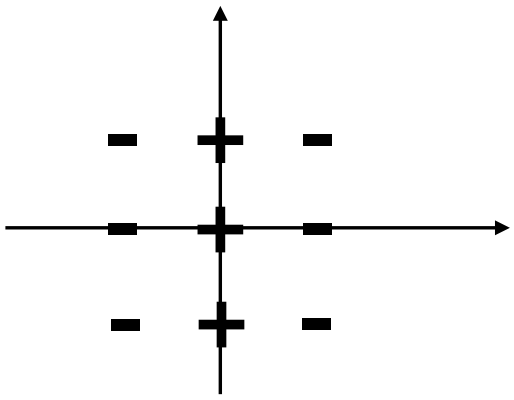


Figure 2: Data set for part (d).

(a) $N = 1$

(b) $N = 2$

(c) $N = 3$

(d) $N = 4$

10 When testing is "unfair" (Est. 35 minutes)

In this problem we explore what happens when we train an estimator on a data set which has a different distribution than the test set.

For concreteness, let us consider the problem of linear regression where we want to learn something close to the true linear model $f(\mathbf{x}) = \mathbf{x}^\top \mathbf{w}^\star$. Our training set consists of a feature matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$ with rows $\mathbf{x}_1^\top, \ldots, \mathbf{x}_n^\top$ and noisily observed target vector $\mathbf{y} = \mathbf{X}\mathbf{w}^\star + \boldsymbol{\epsilon}$ with Gaussian observation noise $\boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I})$, where $\mathbf{w}^\star \in \mathbb{R}^d$ is the true weight vector.

After fitting a linear regression model to $(\mathbf{X}, \mathbf{y})$ which results in the estimator $\widehat{\mathbf{w}}$, we want to know how well this learnt model will perform on a test set, where inputs have been drawn from a different distribution than the training points, but still generating true outputs in the same manner (namely with $y_{\text{test}} = \mathbf{x}_{\text{test}}^\top \mathbf{w}^\star$, for the same true $\mathbf{w}^\star$). This is sometimes referred to as "covariate shift" in the machine learning and statistical literature.

In the following we assume that $\mathbf{X}$ is full rank and has compact singular value decomposition $\mathbf{X} = \mathbf{U}\mathbf{\Lambda}^{1/2}\mathbf{V}^\top$ with diagonal matrix $\mathbf{\Lambda}$ which has entries $\lambda_1, \ldots, \lambda_d$ on the diagonal and matrices $\mathbf{U} \in \mathbb{R}^{n \times d}, \mathbf{V} \in \mathbb{R}^{d \times d}$. We refer to the matrix $\mathbf{\Sigma} = \mathbf{X}^\top \mathbf{X}$ as the empirical covariance matrix of $\mathbf{X}$.

(a) (Est. 5 minutes) **Write down the least squares estimator $\widehat{\mathbf{w}}$ you obtain by minimizing the ordinary least squares (OLS) objective, i.e. $\widehat{\mathbf{w}} := \arg\min_{\mathbf{w} \in \mathbb{R}^d} \frac{1}{2}\|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2$, in terms of $\mathbf{w}^\star, \mathbf{V}, \mathbf{\Lambda}$ and $\widetilde{\boldsymbol{\epsilon}} = \mathbf{U}^\top \boldsymbol{\epsilon}$.** You do not need to derive the standard OLS solution but can start with that if you choose.

(b) (Est. 5 minutes) **What is the mean and covariance matrix of the Gaussian random vector $\widetilde{\boldsymbol{\epsilon}} = \mathbf{U}^\top \boldsymbol{\epsilon}$?**

(c) (Est. 15 minutes) We are now given a different test set $\mathbf{X}_{\text{test}}$ with empirical covariance $\mathbf{\Sigma}_{\text{test}}$, where $\mathbf{X}_{\text{test}}$ has the same eigenbasis (and hence implicity, the same number n of points) for simplicity, i.e. $\mathbf{X}_{\text{test}} = \mathbf{U}\mathbf{\Lambda}_{\text{test}}^{1/2}\mathbf{V}^\top$. Use parts (a), (b) to **derive the following expected average prediction error in terms of the eigenvalues λ_i of $\mathbf{X}^\top\mathbf{X}$ and eigenvalues λ_i^{test} of $\mathbf{X}_{\text{test}}^\top \mathbf{X}_{\text{test}}$:**

$$\frac{1}{n}\mathbb{E}_{\boldsymbol{\epsilon}}\|\mathbf{X}_{\text{test}}\widehat{\mathbf{w}} - \mathbf{X}_{\text{test}}\mathbf{w}^\star\|^2 = \frac{1}{n}\sum_{i=1}^{n} \frac{\lambda_i^{\text{test}}}{\lambda_i}.$$

Recall that these eigenvalues, λ_i^{test}, correspond to the same i-th eigenvector in $\mathbf{V}$, for $i = 1, \ldots, d$. Note that the expectation above is taken over the only randomness in the problem, the training noise $\boldsymbol{\epsilon}$.

Hint: Depending on how you do it, you might find these facts useful: (i) $\mathbf{u}^\top\mathbf{v} = \text{tr}(\mathbf{v}\mathbf{u}^\top)$ *and (ii)* $\mathbb{E}[\text{tr}(A)] = \text{tr}(\mathbb{E}[A])$ *and (iii)* $\text{tr}(A) = \sum_i \lambda_i$ *where λ_i are the eigenvectors of matrix A. But there are other ways to get to the answer as well.*

(d) (Est. 5 minutes) In practice, we sometimes have a choice of training sets. Let's consider a concrete scenario with $d = 2$. We assume for simplicity that $\mathbf{V} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ for all covariance matrices to follow. Assume that we can choose to obtain noisy observations $\mathbf{y} = \mathbf{X}^\top\mathbf{w}^\star + \boldsymbol{\epsilon}$ for three possible training feature matrices $\mathbf{X}_1, \mathbf{X}_2, \mathbf{X}_3$ whose covariance matrices have the following diagonal eigenvalue matrices

$$\mathbf{\Lambda}_1 = \begin{pmatrix} 10 & 0 \\ 0 & 10 \end{pmatrix} \qquad \mathbf{\Lambda}_2 = \begin{pmatrix} 1 & 0 \\ 0 & 100 \end{pmatrix} \qquad \mathbf{\Lambda}_3 = \begin{pmatrix} 100 & 0 \\ 0 & 0.1 \end{pmatrix}.$$

First, we are given a test set $\mathbf{X}_{\text{test}}$ (with the same singular vectors as $\mathbf{X}$) with diagonal eigenvalue matrix $\mathbf{\Lambda}_{\text{test}} = \begin{pmatrix} 0.01 & 0 \\ 0 & 100 \end{pmatrix}$ and are asked to **minimize the average expected prediction error $\frac{1}{n}\mathbb{E}_{\boldsymbol{\epsilon}}\|\mathbf{X}_{\text{test}}\widehat{\mathbf{w}} - \mathbf{X}_{\text{test}}\mathbf{w}^\star\|_2^2$. Which training feature matrix do you choose and why?**

(e) (Est. 5 minutes) Now you are asked to make a choice of training data from the previous part's $\mathbf{X}_1, \mathbf{X}_2, \mathbf{X}_3$ which **minimizes the worst-case expected error under any possible $\mathbf{x}_{\text{test}}$ with unit norm. Which feature matrix do you choose and why?**

11 Coin tossing with unknown coins (Est. 35 minutes)

This question is about adapting EM and the spirit of k-means to a discrete problem of tossing coins.

We have a bag that contains two kinds of coins that look identical. The first kind has probability of heads p_a and the other kind has probability of heads p_b, but we don't know these. We also don't know how many of each kind of coin are in the bag; so the probability, α_a, of drawing a coin of the first type is also unknown (and since $\alpha_a + \alpha_b = 1$, we do not need to separately estimate α_b, the probability of drawing a coin of the second type).

What we have is n pieces of data: for each data point, someone reached into the bag, pulled out a random coin, tossed it ℓ times and then reported the number h_i which was the number of times it came up heads. The coin was then put back into the bag. This was repeated n times. The resulting n head-counts $(h_1, h_2, h_3, \ldots, h_n)$ constitute our data.

Our goal is to estimate p_a, p_b, α_a from this data in some reasonable way.

For this problem, the binomial distribution can be good to have handy:

$$P(H = h) = \binom{\ell}{h} p^h (1-p)^{\ell - h}$$

for the probability of seeing exactly h heads having tossed a coin ℓ times with each toss independently having probability p of turning up heads. Also recall that the mean and variance of a binomial distribution are given respectively by ℓp and $\ell p(1-p)$.

(a) (Est. 10 minutes) **How would you adapt the main ideas in the k-means algorithm to construct an analogous approach to estimating $\widehat{p}_a, \widehat{p}_b, \widehat{\alpha}_a$ from this data set? Give an explicit algorithm, although it is fine if it is written just in English.**

(b) (Est. 8 minutes) Suppose that the true $p_a = 0.4$ and the true $p_b = 0.6$ and $\alpha_a = 0.5$, and $\ell = 5$. For $n \to \infty$, **will your "k-means" based estimates (those from the preceding question) for $\widehat{p}_a$ and $\widehat{p}_b$ yield the correct parameter estimates (namely, $\widehat{p}_a = 0.4$ and $\widehat{p}_b = 0.6$)? Why or why not?**

Hint: Draw a sketch of the typical histograms of the number of heads of each coin on the same axes.

(c) (Est. 17 minutes) How would you adapt the EM for Gaussian Mixture Models that you have seen to construct an EM algorithm for estimating $\widehat{p}_a, \widehat{p}_b, \widehat{\alpha}_a$ from this data set?

You don't have to solve for the parameters in closed form, but (i) **write down the E-step update equations (i.e. write down the distributions that should be computed for the E-step — not in general, but specifically for this problem) and (ii) the objective function that gets maximized for the M-step and also what you are maximizing with respect to (again,**

not just the general form, but specific to this problem). If you introduce any notation, be sure to **explain what everything means. Explain in words what the E- and M-steps are doing on an intuitive level.**

12 SVMs for Novelty Detection (Est. 40 minutes)

This problem is an SVM-variant that works with training data from only one class.

The classification problems we saw in class are two-class or multi-class classification problems. What would one-class classification even mean? In a one-class classification problem, we want to determine whether our new test sample is *normal* (not as in Gaussian), namely whether it is a member of the class represented by the training data or whether it is *abnormal*. One-class classification is also called *outlier detection*. In particular, we assume that all/most of the training data are from the normal class, and want to somehow model them, such that for new unseen test points, we can tell whether they "look like" these points, or whether they are different (i.e, abnormal).

For example, Netflix may want to predict whether a user likes a movie but only have thumbs-up data about movies that the user liked and no thumbs-down votes at all. How can we deal with learning with no negative training samples?

(a) (Est. 7 minutes) Let $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ be your training data for the one-class classification problem (all supposedly belonging to one class — the normal class). One way to formulate one-class classification using SVMs is to have the goal of finding a decision plane which goes through the origin, and for which all the training points are on one side of it. We also want to maximize the distance between the decision plane and the data points. Let the equation of the decision plane H be

$$H := \{\mathbf{x} \in \mathbb{R}^d | \mathbf{w}^\top \mathbf{x} = 0\}. \tag{12}$$

Let the margin m be the distance between the decision plane and the data points

$$m = \min_i \frac{|\mathbf{w}^\top \mathbf{x}_i|}{\|\mathbf{w}\|}. \tag{13}$$

If the convex hull of the training data does not contain the origin, then it is possible to solve the following optimization problem:

$$\widehat{\mathbf{w}} = \arg\min_{\mathbf{w}} \quad \frac{1}{2}\|\mathbf{w}\|_2^2 \tag{14}$$

$$\text{subject to} \quad \mathbf{w}^\top \mathbf{x}_i \geq 1, \quad 1 \leq i \leq n \tag{15}$$

Argue that in the above hard one-class SVM optimization (assuming that the convex hull of the training data does not contain the origin), **the resulting margin is given by** $\widehat{m} = \frac{1}{\|\widehat{\mathbf{w}}\|}$**.**

(b) (**Bonus** Est. 10 minutes) The optimal $\widehat{\mathbf{w}}$ in the hard one-class SVM optimization problem defined by (14) and (15) is identical to the optimal $\widehat{\mathbf{w}}_{\text{two-class}}$ in the traditional two-class hard-margin SVM you saw in class using the augmented training data $(\mathbf{x}_1, 1)$, $(\mathbf{x}_2, 1)$, …, $(\mathbf{x}_n, 1)$, $(-\mathbf{x}_1, -1)$, $(-\mathbf{x}_2, -1)$, …, $(-\mathbf{x}_n, -1)$.

Argue why this is true by comparing the objective functions and constraints of the two optimization problems, as well as the optimization variables.

(c) (Est. 3 minutes) It turns out that the hard one-class SVM optimization cannot deal with problems in which the origin is in the convex hull of the training data. To extend the one-class SVM to such data, we use the hinge loss function

$$\max(0, 1 - \mathbf{w}^\top \mathbf{x}_i) \tag{16}$$

to replace the hard constraints used in the one-class SVM so that the optimization becomes

$$\widehat{\mathbf{w}} = \arg\min_{\mathbf{w}} \ \frac{1}{2}\|\mathbf{w}\|_2^2 + C \sum_{i=1}^{n} \max(0, 1 - \mathbf{w}^\top \mathbf{x}_i). \tag{17}$$

Explain how the hyper-parameter $C > 0$ affects the behavior of the soft one-class SVM in (17).

(d) (Est. 5 minutes) Let $\widehat{\mathbf{w}}^\top \mathbf{x}$ be the score of sample $\mathbf{x}$ where $\widehat{\mathbf{w}}$ is the optimal $\mathbf{w}$ according to the training data and the soft one-class SVM in (17) with some particular C. We will classify a sample as an outlier if its score is below a threshold η. **Describe a procedure to find a threshold η so that about 5% of your training data will be classified as outliers.**

(e) (Est. 10 minutes) The Lagrangian dual of the soft one-class SVM optimization problem is:

$$\begin{aligned} \widehat{\boldsymbol{\alpha}} = \arg\max_{\boldsymbol{\alpha}} \quad & \boldsymbol{\alpha}^\top \mathbf{1} - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j \mathbf{x}_i^\top \mathbf{x}_j \\ & 0 \le \alpha_i \le C, \quad 1 \le i \le n. \end{aligned} \tag{18}$$

Given a sample $\mathbf{x}_{\text{test}}$ and the $\widehat{\mathbf{w}}$ solution to the primal one-class-SVM optimization in (17), you could test whether it is an outlier by checking if $\widehat{\mathbf{w}}^\top \mathbf{x}_{\text{test}} < \eta$. **How can you test whether $\mathbf{x}_{\text{test}}$ is an outlier using the dual formulation given the training data and the optimal $\widehat{\boldsymbol{\alpha}}$ from (18)?**

(f) (Est. 5 minutes) Your friend claims that linear models like the one-class SVM are too simple to be useful in practice. After all, for the example training data in Figure 3, it is impossible to find a sensible decision line to separate the origin and the raw training data. Suppose that we believe the right pattern for "normalcy" here is everything within an approximate annulus around the unit circle. **How could you use the one-class SVM to do the right thing for outlier detection with such data? Explain your answer.**

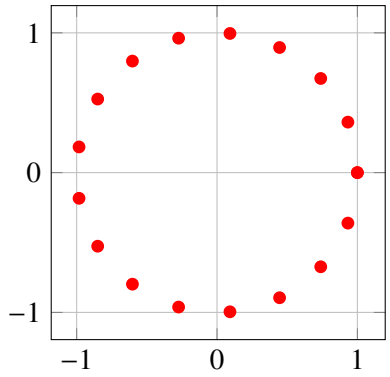


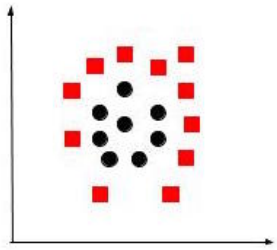
Figure 3: Counter example provided by your friend.

13 Multiple Choice Questions (Est. 30 minutes)

For these questions, **select all the answers which are correct.** You will get full credit for selecting all the right answers. On some questions, real-valued partial credit will be assigned.

You will be graded on your best 15 of the 18 questions below. So feel free to skip three of them if you want.

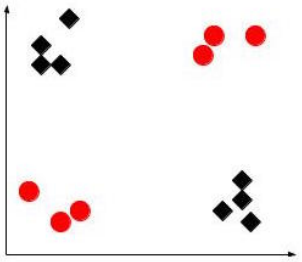
(a) Which of the following classifiers can be used to classify the training data *perfectly*? Methods are not kernelized or with additional features unless explicitly stated so.



- ◯ Logistic regression
- ◯ 3-nearest neighbors
- ◯ SVM with a quadratic kernel
- ◯ Hard margin SVM
- ◯ LDA
- ◯ QDA

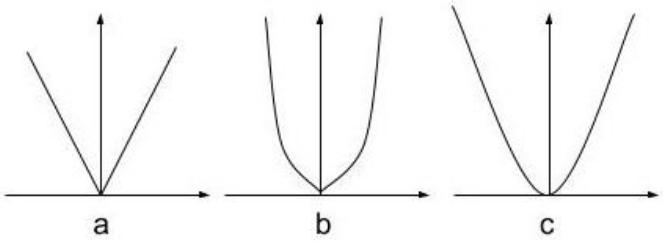
(b) Which of the following classifiers can be used to classify the data *perfectly*? Methods are not

kernelized or with additional features unless explicitly stated so.



- ◯ Logistic regression without a kernel
- ◯ 3-nearest neighbors
- ◯ SVM with a quadratic kernel
- ◯ Hard margin SVM
- ◯ LDA
- ◯ QDA

(c) Which of the following loss functions tends to result in a sparse solution? Select all that apply.



- ◯ a
- ◯ b
- ◯ c
- ◯ None

(d) "Regularization" is a critical aspect of machine learning. Which of the following statements about regularization is true?

- ◯ the main function of regularization is to reduce the bias of your learned predictor
- ◯ the main function of regularization is to reduce the variance of your learned predictor
- ◯ regularization serves to combat overfitting
- ◯ regularization serves to combat underfitting
- ◯ Removing training data points is regularizing
- ◯ Adding noisy copies of training data points is regularizing
- ◯ Dropout is regularizing

- ◯ Doing dimensionality reduction as a part of the learning process has a regularizing effect.
- ◯ Adding a penalty term on the learned weights can be regularizing.
- ◯ Adding a prior for the learned weights can be regularizing.
- ◯ Averaging together the results of diverse predictors is regularizing.
- ◯ Adding features to a learning problem is regularizing.
- ◯ Using kernelized methods is always more regularizing than using direct methods.

(e) The form of the decision boundary for a Linear Discriminant Analysis (LDA) based classifier

- ◯ is always linear
- ◯ is always quadratic
- ◯ is always cubic
- ◯ can take on any form
- ◯ depends on if the covariance matrices are diagonal
- ◯ depends on if the means for each class are are all the same

(f) K-means

- ◯ is like LDA and QDA in that it is a generative model for classification.
- ◯ is similar to EM in that during training it alternates between (i) assigning points to clusters and (ii) updating the parameters of the model – centroids for each class.
- ◯ is like Mixtures of Gaussians because it works on unsupervised data.
- ◯ always optimizes a loss which is equivalent to cross-entropy loss.
- ◯ is dissimilar to EM in that it hard assigns each training point to only one cluster during each training iteration.
- ◯ is guaranteed to find a global optimum.

(g) Which of the following statements are true about loss functions?

- ◯ loss functions are usually learned from the data set
- ◯ cross-entropy loss and mean-squared loss are always identical.
- ◯ the lower the value of the loss function at the end of training, the better the generalization error.
- ◯ MSE loss can be derived from doing maximum likelihood on a linear regression model (with Gaussian iid noise).

(h) Your organization wants to develop a predictor as a component of a self-driving car. It collects a lot of labeled data and randomly divides it into two parts A and B. It creates different teams to try different prediction techniques and gives every team access to A. The organization holds back B so that it can evaluate the performance of each team's predictor on B to choose which predictor to commit to for further development in the self-driving car. Your team has further divided the A data into two disjoint sets: A_V and A_T. Which of the following are true?

- ◯ If you use A_T to train your neural net predictor, you should use A_V to decide how many layers to use.
- ◯ If you use A_T to train a kernelized SVM, you should use A_V to pick the kernel parameters.
- ◯ If you use A_T to create a nearest-neighbor based predictor, you should use A_V to pick the number of neighbors.
- ◯ If you use A_T to train a boosted decision tree, you should use A_V to decide how many stages of boosting to use.
- ◯ Your team should try to figure out a way to get access to B so you can make the best predictor.

(i) Consider using k-nearest-neighbors for the problem of classification. Which of the following are true?

- ◯ The training classification error on the training data used to create the 1-nearest-neighbor classifier is always zero.
- ◯ As the number of neighbors k gets very large and the number of training points goes to infinity, the probability of error for a k-nearest-neighbor classifier will tend to approach the probability of error for a MAP classifier that knew the true underlying distributions.
- ◯ Reducing k has a regularizing effect.
- ◯ If the training data is linearly separable with a positive margin, then the decision boundary found by 1-nearest-neighbors will always be a hyperplane.
- ◯ k-nearest-neighbors can be regularized by dimensionality reduction via PCA applied to the training data.

(j) Consider using a random forest of depth-1 trees for least-squares regression where the individual trees are formed by randomly picking a feature and then randomly chosing a value for it to split on. All random choices for the trees in the random forest are made i.i.d. Which of the following are true?

- ◯ Such individual regression trees are always going to be unbiased estimators.
- ◯ The expected (over both the training data and the random choices used to construct the trees) bias of the random forest estimator will always be the same as the expected bias of a single random tree.
- ◯ The variance (with an expectation over both the training data and the random choices used to construct the trees) of the random forest estimator will be a non-increasing function of the number of trees in the random forest.
- ◯ The random forest estimator will tend to become unbiased as the number of trees in it increases.

(k) You have a function that is implemented as a black box executable to which you have lost the source code. (You can run it as much as you would like though.) The function has three inputs that are a True/False Boolean, a real number from $[-1, +1]$, and an integer from 1 to 10. The function returns true or false. You decide to learn an approximation to this function

using machine learning and use the black box executable to generate training data. Which of the following are reasonable things to do in this situation?

- ◯ Representing the Boolean input using ± 1.
- ◯ Check our performance using a validation set that was drawn separately from the training set
- ◯ Try using AdaBoost with decision trees to fit the training data
- ◯ Try using OLS to fit the training data
- ◯ Represent the integer input using a real number but scaling and shifting it to be in the range $[-1, +1]$.
- ◯ Represent the integer input using a vector of length 10 and one-hot-encoding.
- ◯ Divide the problem into 20 sub-problems based on the Boolean and integer values, and just learn a 1-dimensional classifier for each one using a binary decision tree on the remaining real input.
- ◯ Do dimensionality reduction using random projections to regularize this problem

(l) Suppose we have a neural network with one hidden layer and all linear activations. Then we create a second model which is similar, but with 50 hidden layers (of the same size). No parameters are tied. Which of the following are true?

- ◯ the 50-layer network has the same representational capacity as the 1-layer network.
- ◯ the 50-layer network has greater representational capacity as the 1-layer network.
- ◯ the 50-layer network is more likely to suffer from unstable gradients than the 1-layer network.
- ◯ if we let the 1-layer network be arbitrarily wide and replace the linear activations with sigmoid activations, this updated 1-layer network would have higher representational capacity (be able to model a larger class of funtions) than the original 50-layer network (with only linear activatoins).

(m) Which of the following statements about Convolutional Neural Networks (CNN) are true?

- ◯ the convolution operation is invariant to image rotation
- ◯ a CNN has fewer parameters than a fully connected network when the number of hidden units (and input size and number of outputs) is the same
- ◯ a CNN has more parameters than a fully connected network when the number of hidden units (and input size and number of outputs) is the same
- ◯ the convolutional filters must be designed by hand because they cannot be learned by gradient descent

(n) You designed a 3-layer, fully connected neural network with sigmoid activation functions for a classification task. The output layer is a linear layer followed by a softmax function. The whole network is trained by stochastic gradient descent with cross entropy loss. You observe that the accuracy on the training set is close to 100%, however, it performs poorly on the hold-out validation set. Which of these are likely to improve the validation performance?

- ◯ Increase the number of hidden units
- ◯ Reduce the number of hidden units
- ◯ Increase the ℓ_2 regularization weight
- ◯ Reduce the ℓ_2 regularization weight
- ◯ Add the use of dropout to the neural network training
- ◯ Train the network for more iterations/epochs

(o) The ReLU activation functions, $r(z)$, is less likely to contribute to vanishing gradients than the sigmoid activation function, $s(z)$, because:

- ◯ $s(z) >= r(z)$ for $z < 0$
- ◯ $r(z)$ always has a constant non-zero gradient for $z > 0$
- ◯ $r(z)$ is constant when $z < 0$
- ◯ $r(z = 0) = 0$ whereas $s(z = 0) = 0.5$

(p) Backpropagation

- ◯ is a type of neural network model
- ◯ is a way of computing the gradient in neural networks
- ◯ yields better parameter estimation than using the chain rule for the same number of training iterations
- ◯ only works for cross-entropy loss
- ◯ prevents vanishing gradients by efficient local computation
- ◯ is efficient in part because it caches factors of the gradient that can be re-used
- ◯ has time complexity that is linear in the number of layers
- ◯ has time complexity that is quadratic in the number of layers

14 Your Own Question

Write your own question, and provide a thorough solution.

Writing your own problems is a very important way to really learn the material. The famous "Bloom's Taxonomy" that lists the levels of learning is: Remember, Understand, Apply, Analyze, Evaluate, and Create. Using what you know to create is the top-level. We rarely ask you any HW questions about the lowest level of straight-up remembering, expecting you to be able to do that yourself. (e.g. make yourself flashcards) But we don't want the same to be true about the highest level.

As a practical matter, having some practice at trying to create problems helps you study for exams much better than simply counting on solving existing practice problems. This is because thinking about how to create an interesting problem forces you to really look at the material from the perspective of those who are going to create the exams.

Besides, this is fun. If you want to make a boring problem, go ahead. That is your prerogative. But it is more fun to really engage with the material, discover something interesting, and then come up with a problem that walks others down a journey that lets them share your discovery. You don't have to achieve this every week. But unless you try every week, it probably won't happen ever.

Contributors:

- Alexander Tsigler
- Anant Sahai
- Fanny Yang
- Garrett Thomas
- Jennifer Listgarten
- Joshua Sanz
- Kailas Vodrahalli
- Lisa Jian
- Mark Velednitsky
- Michael Laskey
- Peter Wang
- Raaz RSK Dwivedi
- Ruta Jawale
- Yang Gao
- Yaodong Yu
- Yichao Zhou