Homework #4

CSE 446/546: Machine Learning
Prof. Matt Golub & Pang Wei Koh
Due: **Wednesday** Dec 4, 2024 11:59pm
Points A: 96; B: 25

Please review all homework guidance posted on the website before submitting to Gradescope. Reminders:

- All code must be written in Python and all written work must be typeset (e.g. LaTeX).
- Make sure to read the "What to Submit" section following each question and include all items.
- Please provide succinct answers and supporting reasoning for each question. Similarly, when discussing experimental results, concisely create tables and/or figures when appropriate to organize the experimental results. All explanations, tables, and figures for any particular part of a question must be grouped together.
- For every problem involving generating plots, please include the plots as part of your PDF submission.
- When submitting to Gradescope, please link each question from the homework in Gradescope to the location of its answer in your homework PDF. Failure to do so may result in deductions of up to 10% of the value of each question not properly linked. For instructions, see https://www.gradescope.com/get_started#student-submission.

Important: By turning in this assignment (and all that follow), you acknowledge that you have read and understood the collaboration policy with humans and AI assistants alike: https://courses.cs.washington.edu/courses/cse446/24au/assignments/. Any questions about the policy should be raised at least 24 hours before the assignment is due. There are no warnings or second chances. If we suspect you have violated the collaboration policy, we will report it to the college of engineering who will complete an investigation. Not adhering to these reminders may result in point deductions.

Conceptual Questions

A1. The answers to these questions should be answerable without referring to external materials. Briefly justify your answers with a few words.

a. *[2 points]* True or False: Given a data matrix $X \in \mathbb{R}^{n \times d}$ where d is much smaller than n and $k = \text{rank}(X)$, if we project our data onto a k-dimensional subspace using PCA, our projection will have zero reconstruction error (in other words, we find a perfect representation of our data, with no information loss).

b. *[2 points]* True or False: Suppose that an $n \times n$ matrix X has a singular value decomposition of $USV^\top$, where S is a diagonal $n \times n$ matrix. Then, the rows of V are equal to the eigenvectors of $X^\top X$.

c. *[2 points]* True or False: choosing k to minimize the k-means objective (see Equation (4) below) is a good way to find meaningful clusters.

d. *[2 points]* True or False: The singular value decomposition of a matrix is unique.

e. *[2 points]* True or False: The rank of a square matrix equals the number of its unique nonzero eigenvalues.

What to Submit:

- **Parts a-e:** 1-2 sentence explanation containing your answer.

Think before you train

A2. **The first part of this problem (part a)** explores how you would apply machine learning theory and techniques to a real-world problem. There is one scenario detailing a setting, a dataset, and a specific result we hope to achieve. Your job is to describe how you would handle the scenario with the tools we've learned in this class. Your response should include:

(1) any pre-processing steps you would take (e.g. any data processing),

(2) the specific machine learning pipeline you would use (i.e., algorithms and techniques learned in this class),

(3) how your setup acknowledges the constraints and achieves the desired result.

You should also aim to leverage some of the theory we have covered in this class. Some things to consider may be: the nature of the data (i.e., *How hard is it to learn? Do we need more data? Are the data sources good?*), the effectiveness of the pipeline (i.e., *How strong is the model when properly trained and tuned?*), and the time needed to effectively perform the pipeline.

a. *[10 points]* **Scenario: Disease Susceptibility Predictor**

- Setting: You are tasked by a research institute to create an algorithm that learns the factors that contribute most to acquiring a specific disease.
- Dataset: A rich dataset of personal demographic information, location information, risk factors, and whether a person has the disease or not.
- Result: The company wants a system that can determine how susceptible someone is to this disease when they enter in their own personal information. The pipeline should take limited amount of personal data from a new user and infer more detailed metrics about the person.

The second part of this problem (parts b, c) focuses on exploring possible shortcomings of machine learning models, and what real-world implications might follow from ignoring these issues.

b. *[5 points]* Briefly describe (1) some potential shortcomings of your training process from the disease susceptibility predictor scenario above that may result in your algorithm having different accuracy on different populations, and (2) how you may modify your procedure to address these shortcomings.

c. *[5 points]* Recall in Homework 2 we trained models to predict crime rates using various features. It is important to note that **datasets describing crime have many shortcomings in describing the entire landscape of illegal behavior in a city, and that these shortcomings often fall disproportionately on minority communities**. Some of these shortcomings include that crimes are reported at different rates in different neighborhoods, that police respond differently to the same crime reported or observed in different neighborhoods, and that police spend more time patrolling in some neighborhoods than others. What real-world implications might follow from ignoring these issues?

What to Submit:

- **For part (a):** One clearly-written short paragraph (approximately 4-7 sentences).
- **For part (b):** Clearly-written and well-thought-out answers addressing (1) and (2) (as described in the problem). Two short paragraphs or one medium paragraph suffice.
- **For part (c):** One clearly-written short paragraph on real-world implications that may follow from ignoring dataset issues.

Image Classification on CIFAR-10

A3. In this problem we will explore different deep learning architectures for image classification on the CIFAR-10 dataset. Make sure that you are familiar with `torch.Tensors`, two-dimensional convolutions (`nn.Conv2d`) and fully-connected layers (`nn.Linear`), ReLU non-linearities (`F.relu`), pooling (`nn.MaxPool2d`), and tensor reshaping (`view`).

Hint: For loops are costly. Can you vectorize it or use Numpy operations to make it faster in some ways?

A few preliminaries:

- Make sure to read the "Tips for HW4" EdStem post for additional tips about training your models.
- Each network f maps an image $x^{\text{in}} \in \mathbb{R}^{32\times32\times3}$ (3 channels for RGB) to an output $f(x^{\text{in}}) = x^{\text{out}} \in \mathbb{R}^{10}$. The class label is predicted as $\arg\max_{i=0,1,\ldots,9} x_i^{\text{out}}$. An error occurs if the predicted label differs from the true label for a given image.
- The network is trained via multiclass cross-entropy loss.
- Create a validation dataset by appropriately partitioning the train dataset. *Hint*: look at the documentation for `torch.utils.data.random_split`. Make sure to tune hyperparameters like network architecture and step size on the validation dataset. Do **NOT** validate your hyperparameters on the test dataset.
- At the end of each epoch (one pass over the training data), compute and print the training and validation classification accuracy.
- While one could train a network for hundreds of epochs to reach convergence and maximize accuracy, this can be prohibitively time-consuming, so feel free to train for just a dozen or so epochs.

For parts (a) and (b), apply a hyperparameter tuning method (e.g. random search, grid search, etc.) using the validation set, report the hyperparameter configurations you evaluated and the best set of hyperparameters from this set, and plot the training and validation classification accuracy as a function of epochs. Produce a separate line or plot for each hyperparameter configuration evaluated (top 3 configurations is sufficient to keep the plots clean). Finally, evaluate your best set of hyperparameters on the test data and report the test accuracy.

Note 1: Please refer to the provided notebook with starter code for this problem, on the course website. That notebook provides a complete end-to-end example of loading data, training a model using a simple network with a fully-connected output and no hidden layers (this is equivalent to logistic regression), and performing evaluation using canonical Pytorch. We recommend using this as a template for your implementations of the models below.
Note 2: If you are attempting this problem and do not have access to GPU we highly recommend using Google Colab. The provided notebook includes instructions on how to use GPU in Google Colab.

Here are the network architectures you will construct and compare.

a. *[18 points]* **Fully-connected output, 1 fully-connected hidden layer:** this network has one hidden layer denoted as $x^{\text{hidden}} \in \mathbb{R}^M$ where M will be a hyperparameter you choose (M could be in the hundreds). The nonlinearity applied to the hidden layer will be the `relu` ($\text{relu}(x) = \max\{0, x\}$. This network can be written as

$$x^{out} = W_2\text{relu}(W_1(x^{in}) + b_1) + b_2$$

where $W_1 \in \mathbb{R}^{M\times3072}$, $b_1 \in \mathbb{R}^M$, $W_2 \in \mathbb{R}^{10\times M}$, $b_2 \in \mathbb{R}^{10}$. Tune the different hyperparameters and train for a sufficient number of epochs to achieve a *validation accuracy* of at least 50%. Provide the hyperparameter configuration used to achieve this performance.

b. *[18 points]* **Convolutional layer with max-pool and fully-connected output:** for a convolutional layer W_1 with filters of size $k \times k \times 3$, and M filters (reasonable choices are $M = 100$, $k = 5$), we have that $\text{Conv2d}(x^{\text{in}}, W_1) \in \mathbb{R}^{(33-k)\times(33-k)\times M}$.

- Each convolution will have its own offset applied to each of the output pixels of the convolution; we denote this as $\text{Conv2d}(x^{\text{in}}, W) + b_1$ where b_1 is parameterized in $\mathbb{R}^M$. Apply a `relu` activation to the result of the convolutional layer.
- Next, use a max-pool of size $N \times N$ (a reasonable choice is $N = 14$ to pool to 2×2 with $k = 5$) we have that $\text{MaxPool}(\text{relu}(\text{Conv2d}(x^{\text{in}}, W_1) + b_1)) \in \mathbb{R}^{\lfloor \frac{33-k}{N} \rfloor \times \lfloor \frac{33-k}{N} \rfloor \times M}$.
- We will then apply a fully-connected layer to the output to get a final network given as

$$x^{output} = W_2(\text{MaxPool}(\text{relu}(\text{Conv2d}(x^{\text{input}}, W_1) + b_1))) + b_2$$

  where $W_2 \in \mathbb{R}^{10 \times M(\lfloor \frac{33-k}{N} \rfloor)^2}$, $b_2 \in \mathbb{R}^{10}$.

The parameters M, k, N (in addition to the step size and momentum) are all hyperparameters, but you can choose a reasonable value. Tune the different hyperparameters (number of convolutional filters, filter sizes, dimensionality of the fully-connected layers, step size, etc.) and train for a sufficient number of epochs to achieve a *validation accuracy* of at least 65%. Provide the hyperparameter configuration used to achieve this performance.

The number of hyperparameters to tune, combined with the slow training times, will hopefully give you a taste of how difficult it is to construct networks with good generalization performance. State-of-the-art networks can have dozens of layers, each with their own hyperparameters to tune. Additional hyperparameters you are welcome to play with, if you are so inclined, include: changing the activation function, replace max-pool with average-pool, adding more convolutional or fully connected layers, and experimenting with batch normalization or dropout.

What to Submit:

- **For parts (a)-(b):** A single plot of the training and validation accuracy for the top 3 hyperparameter configurations you evaluated (x-axis is training epoch; y-axis is accuracy; this plot should contain 6 lines total). If it took fewer than 3 hyperparameter configurations to pass the performance threshold, plot all hyperparameter configurations you evaluated. A horizontal line should be plotted at the targeted threshold (50% or 65%). Validation lines should be dotted, and training lines should be solid.
- **For parts (a)-(b):** List the hyperparameter values you searched over and your search method (random, grid, etc.). Provide the values of best performing hyperparameters, and accuracy of best models on test data.
- **For parts (a)-(b):** Code. You should convert your code (the .ipynb notebook) into a Python (.py) file, rename it to `hw4-a3.py`, and submit it to the corresponding Gradescope submission. To download the file from Google Colab, you can go to File > Download > Download as .py.

Matrix Completion and Recommendation System

A4.

Note: Please refer to the provided notebook with starter code for this problem, on the course website. The notebook provides a template for each part of the problem, and includes code to help load the data properly. We recommend creating a copy of the starter notebook and completing the assignment by filling out the template.

Hint: **For loops are costly.** Can you vectorize it or use Numpy operations to make it faster in some ways?

You will build a personalized movie recommendation system. We will use the 100K MovieLens dataset available at https://grouplens.org/datasets/movielens/100k/. There are $m = 1682$ movies and $n = 943$ users. Each user rated at least 20 movies, but some watched many more. The total dataset contains $100,000$ total ratings from all users. The goal is to recommend movies the users haven't seen.

Consider a matrix $R \in \mathbb{R}^{m \times n}$ where the entry $R_{i,j} \in \{1, \ldots, 5\}$ represents the jth user's rating on movie i. A higher value represents that the user is more satisfied with the movie.

We may think of our historical data as some observed entries of this matrix while many remain unknown, and we wish to estimate the unknown ratings that each user would assign to each movie. We could use these ratings to recommend the "best" movies for each user.

The dataset contains user and movie metadata which we will ignore. We strictly use the ratings contained in the `u.data` file. Use this data file and the following python code to construct a training and test set:

```
import csv
import numpy as np
data = []
with open('u.data') as csvfile:
    spamreader = csv.reader(csvfile, delimiter='\t')
    for row in spamreader:
        data.append([int(row[0])-1, int(row[1])-1, int(row[2])])
data = np.array(data)

num_observations = len(data)   # num_observations = 100,000
num_users = max(data[:,0])+1   # num_users = 943, indexed 0,...,942
num_items = max(data[:,1])+1   # num_items = 1682 indexed 0,...,1681

np.random.seed(1)
num_train = int(0.8*num_observations)
perm = np.random.permutation(data.shape[0])
train = data[perm[0:num_train],:]
test = data[perm[num_train::],:]
```

The arrays `train` and `test` contain R_{train} and R_{test}, respectively. Each line takes the form "j, i, s", where j is the user index, i is the movie index, and s is the user's score.

Using `train`, you will train a model that can predict $\widehat{R} \in \mathbb{R}^{m \times n}$, how every user would rate every movie. You will evaluate your model based on the average squared error on `test`:

$$\mathcal{E}_{\text{test}}(\widehat{R}) = \frac{1}{|\text{test}|} \sum_{(i,j,R_{i,j}) \in \text{test}} (\widehat{R}_{i,j} - R_{i,j})^2.$$

Low-rank matrix factorization is a baseline method for personalized recommendation. It learns a vector representation $u_i \in \mathbb{R}^d$ for each movie and a vector representation $v_j \in \mathbb{R}^d$ for each user, such that the inner product

$\langle u_i, v_j \rangle$ approximates the rating $R_{i,j}$. You will build a simple latent factor model.

You will implement multiple estimators and use the inner product $\langle u_i, v_j \rangle$ to predict if user j likes movie i in the test data. For simplicity, we will put aside best practices and choose hyperparameters by using those that minimize the test error. You may use fundamental operators from `numpy` or `pytorch` in this problem (`numpy.linalg.lstsq, SVD, autograd,` etc.) but not any precooked algorithm from a package like `scikit-learn`. If there is a question whether some package is not allowed for use in this problem, it probably is not appropriate.

a. *[5 points]* Our first estimator pools all users together and, for each movie, outputs as its prediction the average user rating of that movie in `train`. That is, if $\mu \in \mathbb{R}^m$ is a vector where μ_i is the average rating of the users that rated the ith movie, write this estimator $\widehat{R}$ as a rank-one matrix.
Compute the estimate $\widehat{R}$. What is $\mathcal{E}_{\text{test}}(\widehat{R})$ for this estimate?

b. *[5 points]* Allocate a matrix $\widetilde{R}_{i,j} \in \mathbb{R}^{m \times n}$ and set its entries equal to the known values in the training set, and 0 otherwise. Let $\widehat{R}^{(d)}$ be the best rank-d approximation (in terms of squared error) approximation to $\widetilde{R}$. This is equivalent to computing the singular value decomposition (SVD) and using the top d singular values. This learns a lower-dimensional vector representation for users and movies, assuming that each user would give a rating of 0 to any movie they have not reviewed.

- For each $d = 1, 2, 5, 10, 20, 50$, compute the estimator $\widehat{R}^{(d)}$. We recommend using an efficient solver such as `scipy.sparse.linalg.svds`.
- Plot the average squared error of predictions on the training set and test set on a single plot, as a function of d.

Note that, in most applications, we would not actually allocate a full $m \times n$ matrix. We do so here only because our data is relatively small and it is instructive.

c. *[10 points]* Replacing all missing values by a constant may impose strong and potentially incorrect assumptions on the unobserved entries of R. A more reasonable choice is to minimize the MSE (mean squared error) only on rated movies. Define a loss function:

$$\mathcal{L}\Big(\{u_i\}_{i=1}^m, \{v_j\}_{j=1}^n\Big) := \sum_{(i,j,R_{i,j}) \in \text{train}} (\langle u_i, v_j \rangle - R_{i,j})^2 + \lambda \sum_{i=1}^m \|u_i\|_2^2 + \lambda \sum_{j=1}^n \|v_j\|_2^2 \tag{1}$$

where $\lambda > 0$ is the regularization coefficient. We will implement algorithms to learn vector representations by minimizing (1). Note: we define the loss function here as the sum of squared errors; be careful to calculate and plot the mean squared error for your results.

Since this is a non-convex optimization problem, the initial starting point and hyperparameters may affect the quality of $\widehat{R}$. You may need to tune λ and σ to optimize the loss you see.

- *Alternating minimization*: First, randomly initialize $\{u_i\}$ and $\{v_j\}$. Then, alternatate between (1) minimizing the loss function with respect to $\{u_i\}$ by treating $\{v_j\}$ as fixed; and (2) minimizing the loss function with respect to $\{v_j\}$ by treating $\{u_i\}$ as fixed. Repeat (1) and (2) until both $\{u_i\}$ and $\{v_j\}$ converge. Note that when one of $\{u_i\}$ or $\{v_j\}$ is given, minimizing the loss function with respect to the other part has a closed-form solution. Indeed, it can be shown that when minimizing with respect to a *single* u_i (with $\{v_j\}$ fixed), the gradient is given by:

$$\nabla_{u_i} L\Big(\{u_i\}_{i=1}^m, \{v_j\}_{j=1}^n\Big) = 2\left(\sum_{j \in r(i)} v_j v_j^T + \lambda I\right) u_i - 2 \sum_{j \in r(i)} R_{i,j} v_j \tag{2}$$

where here $r(i)$ is a shorthand for the set of users who have reviewed movie i in the training set, or more formally, $r(i) = \{j : (j, i, R_{i,j}) \in \text{train}\}$. Setting the overall gradient to be equal to 0 gives us that

$$\arg\min_{u_i} L\Big(\{u_i\}_{i=1}^m, \{v_j\}_{j=1}^n\Big) = \left(\sum_{j\in r(i)} v_j v_j^T + \lambda I\right)^{-1} \left(\sum_{j\in r(i)} R_{i,j} v_j\right) \tag{3}$$

Note that this update rule is for a single vector u_i, whereas you should update all of the $\{u_i\}_{i=1}^m$ in one round. When it comes to the alternate step which involves fixing $\{u_i\}$ and minimizing $\{v_j\}$, an analogous calculation will give you a very similar update rule.

- Try $d \in \{1, 2, 5, 10, 20, 50\}$ and plot the mean squared error of train and test as a function of d.

Some hints:

- Common choices for initializing the vectors $\{u_i\}_{i=1}^m$, $\{v_j\}_{j=1}^n$ include: entries drawn from `np.random.rand()` scaled by some scale factor $\sigma > 0$ (σ is an additional hyperparameter), or using one of the solutions from part b or c.
- The only $m \times n$ matrix you need to allocate is probably for $\widetilde{R}$.
- It is **crucial** that the squared error part of the loss is only defined w.r.t. $R_{i,j}$ that actually exist in the training set. Consider implementing some type of data structures that allow you to keep track of $r(i)$ as well as the reverse mapping $r^{-1}(j)$ from movies to relevant users.

What to Submit:

- **For part a**: A mathematical expression for $\hat{R}$. Value for $\mathcal{E}_{test}(\hat{R})$.
- **For part b**: Plot of MSE on training and test set vs. d.
- **For part c**: Plot of MSE on training and test set vs. d.
- **For parts a-c**: Code. You should convert your code (the .ipynb notebook) into a Python (.py) file, rename it to `hw4-a4.py`, and submit it to the corresponding Gradescope submission. To download the file from Google Colab, you can go to File > Download > Download as .py.

k-means clustering

A5. Given a dataset $\mathbf{x}_1, ..., \mathbf{x}_n \in \mathbb{R}^d$ and an integer $1 \leq k \leq n$, recall the following k-means objective function

$$\min_{\pi_1,...,\pi_k} \sum_{i=1}^{k} \sum_{j \in \pi_i} \|\mathbf{x}_j - \mu_i\|_2^2 \ , \quad \mu_i = \frac{1}{|\pi_i|} \sum_{j \in \pi_i} \mathbf{x}_j \ . \tag{4}$$

Above, $\{\pi_i\}_{i=1}^k$ is a partition of $\{1, 2, ..., n\}$. The objective (4) is NP-hard[1] to find a global minimizer of. Nevertheless, Lloyd's algorithm (discussed in lecture) typically works well in practice.[2]

a. *[5 points]* Implement Lloyd's algorithm for solving the k-means objective (4). Do not use any off-the-shelf implementations, such as those found in `scikit-learn`.

b. *[5 points]* Run Lloyd's algorithm on the *training* dataset of MNIST with $k = 10$. Show the image representing the center of each cluster, as a set of k 28×28 images.

Note on Time to Run — The runtime of a good implementation for this problem should be fairly fast (a few minutes); if you find it taking upwards of one hour, please check your implementation! (Hint: **For loops are costly.** Can you vectorize it or use Numpy operations to make it faster in some ways? If not, is looping through data-points or through centers faster?)

What to Submit:

- **For part (a):** Nothing required in PDF submission.
- **For part (b):** 10 images of cluster centers.
- **For parts (a)-(b):** Code through corresponding Gradescope coding submission.

[1] To be more precise, it is both NP-hard in d when $k = 2$ and k when $d = 2$.
[2] See the references on the Wikipedia page for k-means and k-means++ for more details.

Random Fourier Features

B1. Kernel methods such as Logistic Regression are considered memory-based learners. Rather than learning a mapping from a set of input features $\mathcal{X} \subset \mathbb{R}^d$ to outputs in $\mathcal{Y}$, they *remember* all training examples $(\mathbf{x_i}, y_i)$ and learn a corresponding weight for them.

$$\hat{f}(\mathbf{x}) = \sum_{i=1}^{N} \omega_i k(\mathbf{x_i}, \mathbf{x})$$

After learning the weight vector $\mathbf{w} = [\mathbf{w}_1, ..., \mathbf{w}_N]$, we can make prediction on unseen samples using the *kernel function* k between all training samples and $\mathbf{x}$. Kernel methods are attractive because they rely on the *kernel trick*. Any positive definite function $k(\mathbf{x}, \mathbf{x}')$ with $\mathbf{x}, \mathbf{x}' \in \mathbb{R}^d$ defines a function ψ mapping $\mathbb{R}^d$ to a higher-dimensional space such that the inner product between datapoints can be quickly computed as $\langle \psi(\mathbf{x}), \psi(\mathbf{x}') \rangle = k(\mathbf{x}, \mathbf{x}')$. In essence, the kernel trick is an efficient way to learn a linear decision boundary in a higher dimension space than that of $\mathcal{X}$.

The kernel trick can be prohibitively expensive for large datasets. This is because the memory-based algorithm accesses the data through evaluations of the kernel matrix $k(x, x')$ which grows in proportion to the dataset size N.
Instead of relying on the implicit feature mapping ψ provided by the kernel trick, suppose we can approximate the kernel function k as the inner product of two vectors in $\mathbb{R}^D$. Mathematically, we would like to find a mapping $\mathbf{z}$:

$$\mathbf{z} : \mathbb{R}^d \to \mathbb{R}^D \qquad \text{such that} \qquad k_p(\mathbf{x}, \mathbf{x}') = \langle \psi(\mathbf{x}), \psi(\mathbf{x}') \rangle \approx \langle \mathbf{z}(\mathbf{x}), \mathbf{z}(\mathbf{x}') \rangle$$

With this approximation, we no longer require the *kernel trick* to express $\langle \psi(\mathbf{x}), \psi(\mathbf{x}') \rangle$ as $k(\mathbf{x}, \mathbf{x}')$. Rather, we can approximate it by directly computing the tractable inner product $\langle \mathbf{z}(\mathbf{x}), \mathbf{z}(\mathbf{x}') \rangle$.

$$\hat{f}(\mathbf{x}) = \sum_{i=1}^{N} \omega_i k(\mathbf{x}_i, \mathbf{x}) = \sum_{i=1}^{N} \omega_i \langle \psi(\mathbf{x}_i), \psi(\mathbf{x}) \rangle \approx \sum_{i=1}^{N} \omega_i \langle \mathbf{z}(\mathbf{x}_i), \mathbf{z}(\mathbf{x}) \rangle = \left(\sum_{i=1}^{N} \omega_i \mathbf{z}(\mathbf{x}_i)^T \right) \mathbf{z}(\mathbf{x}) = \beta^T \mathbf{z}(\mathbf{x})$$

Assuming $\mathbf{z}(\mathbf{x}) = \sigma(M\mathbf{x} + b)$ for some nonlinear function σ, this "approximate" Logistic Regression *can potentially be evaluated much quicker* than the kernel Logistic Regression. To see why, note that the left-hand-side requires evaluating $k(\mathbf{x_i}, \mathbf{x})$ for all $i \in \{1, \ldots, N\}$, in general, if ω_i is not sparse. On the other hand, the right-hand-side just requires computing $\mathbf{z}(\mathbf{x}) = \sigma(M\mathbf{x} + b)$ which is dominated by the time to compute a $D \times d$ matrix-vector product, and then inner product with β which is $\mathbb{R}^D$. Thus, the total computation time for the left-hand-side scales linearly with N, and the right-hand-side scales with just d and D, independent of N! When training the approximate Logistic Regression we also get similar computational savings if $N \gg \max\{d, D\}$.

a. *[15 points]* **Deriving Random Fourier Features**: Bochner's theorem states that a continuous kernel $k(\mathbf{x}, \mathbf{x}') = k(\mathbf{x} - \mathbf{x}')$ on $\mathbb{R}^d$ is positive definite if and only if k is the Fourier transform of a non-negative measure. While we won't delve into the logic of Fourier transforms here, this theorem lets us express the kernel as follows: for any probability distribution $p(\mathbf{w})$ define

$$k_p(\mathbf{x}, \mathbf{x}') := \int_{\mathbb{R}^d} p(\mathbf{w}) e^{i\mathbf{w}^T(\mathbf{x} - \mathbf{x}')} dw = \mathbb{E}_{\mathbf{w}} \left[e^{i\mathbf{w}^T(\mathbf{x} - \mathbf{x}')} \right]$$

where $i = \sqrt{-1}$, the imaginary unit. While any choice of $p(\mathbf{w})$ induces a valid kernel, in this problem we'll be focusing on the Gaussian distribution, namely

$$p(\mathbf{w}) = (2\pi\sigma^2)^{-\frac{D}{2}} e^{-\frac{1}{2\sigma^2}||\mathbf{w}||_2^2} = (2\pi/\gamma^2)^{-\frac{D}{2}} e^{-\gamma^2||\mathbf{w}||_2^2/2} \quad \text{where } \gamma = \frac{1}{\sigma}$$

In this sub-problem, we'll use this Fourier-transform interpretation of k to derive a randomized mapping $\mathbf{z} : \mathbb{R}^d \to \mathbb{R}^D$ which is an unbiased estimate of the kernel function i.e.

$$\mathbb{E}_{\mathbf{w}}[\mathbf{z}(\mathbf{x})^T \mathbf{z}(\mathbf{x}')] = k_p(\mathbf{x}, \mathbf{x}')$$

If $\mathbf{z}(x)^T \mathbf{z}(x')$ serves as a good approximation to the kernel matrix, we can apply the aforementioned approximation algorithm.

i Use Euler's formula $e^{iy} = \cos(y) + i\sin(y)$ to show that $k_p(\mathbf{x}, \mathbf{x}') = E_{\mathbf{w}}\left[\cos(\mathbf{w}^T(\mathbf{x} - \mathbf{x}'))\right]$.

Hint: If both x and A are real, then $A = \int f(x) + ig(x)dx = \int f(x)dx$.

ii We begin by defining $z_{\mathbf{w}} : \mathbb{R}^d \to \mathbb{R}$ as

$$z_{\mathbf{w}}(\mathbf{x}) = \sqrt{2}\cos(\mathbf{w}^T\mathbf{x} + b) \qquad \text{where } \mathbf{w} \sim p(\mathbf{w}),\ b \sim \text{Uniform}(0, 2\pi)$$

Note that this is not yet the mapping vector $\mathbf{z}$, but rather a mapping to $\mathbb{R}$. Use part (i) to show that the expected product of $z_{\mathbf{w}}(\mathbf{x})$s is an unbiased estimate of the kernel function i.e.

$$E_{\mathbf{w},b}\left[z_{\mathbf{w}}(\mathbf{x}) z_{\mathbf{w}}(\mathbf{x}')\right] = k_p(\mathbf{x}, \mathbf{x}')$$

Hint: For this problem you may use the following identity: $2\cos(\alpha)\cos(\beta) = \cos(\alpha + \beta) + \cos(\alpha - \beta)$.

iii Now we're ready to define our random Fourier features $\mathbf{z} : \mathbb{R}^d \to \mathbb{R}^D$. Let $\mathbf{z}$ be the d-dimensional concatenation of $z_{\mathbf{w}}(\mathbf{x})$s:

$$\mathbf{z}(\mathbf{x}) = \left[\tfrac{1}{\sqrt{D}} z_{\mathbf{w}_1}(\mathbf{x}),\ \tfrac{1}{\sqrt{D}} z_{\mathbf{w}_2}(\mathbf{x}),\ \ldots,\ \tfrac{1}{\sqrt{D}} z_{\mathbf{w}_D}(\mathbf{x})\right]^T$$

Use parts (i) and (ii) to show that the expected inner product of the mapping $\mathbf{z}$ is an unbiased estimate of the kernel function i.e.

$$E_{\mathbf{w}}[\mathbf{z}(\mathbf{x})^T \mathbf{z}(\mathbf{x}')] = k_p(\mathbf{x}, \mathbf{x}')$$

b. *[5 points]* **Random Fourier Features and the RBF Kernel**. As mentioned in part (b), using different distributions $p(\mathbf{w})$ induces different valid kernels. Using the $p(\mathbf{w})$ given in part (a), show that expected value of the inner product between random Fourier features is the RBF kernel i.e.

$$E_{\mathbf{w}}[\mathbf{z}(\mathbf{x})^T \mathbf{z}(\mathbf{x}')] = \exp\left(-\frac{||\mathbf{x} - \mathbf{x}'||_2^2}{2\gamma^2}\right)$$

Hint: The PDF for a variable $X \in \mathbb{R}^d$ following normal distribution with mean μ and covariance matrix Σ is as follows:

$$P(X = x) = ((2\pi)^d |\Sigma|)^{-1/2} \exp\left(-\frac{1}{2}(x - \mu)^\top \Sigma^{-1}(x - \mu)\right)$$

where $|\Sigma| = \det(\Sigma)$ denote the determinant of matrix Σ. In addition, if $\Sigma = \text{diag}(\sigma^2, \ldots, \sigma^2)$, then $|\Sigma| = \sigma^{2d}$, and $\Sigma^{-1} = \text{diag}(\sigma^{-2}, \ldots, \sigma^{-2})$.

c. *[5 points]* **Concentration Bounds** In part (a) we derived our function $\mathbf{z}$ which serve as a good approximation to the kernel function. Our results let us get an upper bound our approximation error for the kernel function. Explain why we can apply Hoeffding's inequality to obtain

$$p(|\mathbf{z}(\mathbf{x})^T \mathbf{z}(\mathbf{x}') - k(\mathbf{x}, \mathbf{x}')| \geq \epsilon) \leq 2\exp\left(-D\epsilon^2/8\right)$$

What to Submit:

- Part a: Separate proofs for subproblems (i), (ii) and (iii)
- Part b: proof
- Part c: proof, 1-2 sentence explanation about which conditions are met that allow us to apply Hoeffding's inequality e.g. "B is bounded by [...]".

Administrative

A6.

a. *[2 points]* About how many hours did you spend on this homework? There is no right or wrong answer :)