Homework 1 – Machine Learning (CS4342, Whitehill, B-Term 2025)

This homework is intended to help you (1) refresh your knowledge of linear algebra and multivariate calculus; (2) learn how to implement linear algebraic operations in Python using `numpy` (to which we refer in the code below as `np`); and (3) practice implementing one of the simpler machine learning algorithms: step-wise classification. **Collaboration policy**: You may complete this assignment with a partner, if you choose. In that case, both partners should sign up on Canvas in a **pre-made group** as a team, and only one of you should submit the assignment. Note: you are allowed to use ChatGPT (or another AI tool) to get *general help* about the topics in this homework assignment, e.g., `numpy` syntax and array manipulation, and plotting of graphs. However, you may *not* use it to solve the specific exercises.

1 Linear Algebra and Multivariate Calculus [9 pts]

1. Let $h(x, y, z) = 2xy - xz + xyz^2$. Find the expressions for all three partial derivatives $\frac{\partial h}{\partial x}, \frac{\partial h}{\partial y}, \frac{\partial h}{\partial z}$. [**2 pts**].

2. Let $g(u, v) = u \sin v + \log(v^2)$. (In this course, always assume that the base of log is e.) Find the expressions for $\frac{\partial g}{\partial u}, \frac{\partial g}{\partial v}$. [**2 pts**].

3. Let $\mathbf{p} = \begin{bmatrix} -1 & 1 & -3 \end{bmatrix}^\top$, $\mathbf{Q} = \begin{bmatrix} 1 & 2 \\ 4 & 1 \\ 2 & 0 \end{bmatrix}$, and $\mathbf{r} = \begin{bmatrix} x \\ y \end{bmatrix}$. Let $g(x, y) = \mathbf{p}^\top \mathbf{Q} \mathbf{r}$. Find the expressions for the two partial derivatives $\frac{\partial g}{\partial x}$ and $\frac{\partial g}{\partial y}$. [**2 pts**]

4. Let $f(x_1, x_2) = \begin{bmatrix} 2x_1 + x_2^2 \\ \exp(-x_1) \\ x_2 \end{bmatrix}$ be a function that takes two inputs x_1, x_2 and produces three outputs f_1, f_2, f_3. Find the expressions for all six partial derivatives $\frac{\partial f_1}{\partial x_1}, \ldots, \frac{\partial f_3}{\partial x_2}$. [**3 pts**].

Please put your answers to these problems in either a text file (you can use the ^ character to represent exponents) `homework1_calc_WPIUSERNAME.txt` or a PDF file (for LaTeX or a scanned piece of handwritten solutions) `homework1_calc_WPIUSERNAME.pdf`.

2 Python and numpy [24 pts]

See the starter file `homework1_numpy.py`. For each of the problems below, write a method (e.g., `problem1`) that returns the answer for the corresponding problem. Put all your methods in one file called `homework1_numpy_WPIUSERNAME.py` (e.g., `homework1_numpy_gwang.py`). In all problems, you may assume that the the dimensions of the matrices and/or vectors are compatible for the requested mathematical operations. Notes:

1. In mathematical notation we usually start indices with $j = 1$. However, in `numpy` (and many other programming settings), it is more natural to use 0-based array indexing. When answering the questions below, do not worry about "translating" from 1-based to 0-based indexes. For example, if the (i, j)th element of some matrix is requested, you can simply write `A[i,j]`.

2. To represent vectors & matrices, please use `numpy`'s `array` class (*not* the `matrix` class).

3. While the difference between a row vector and a column vector is important when doing math, `numpy` does not care about this difference *as long as the array is 1-D*. This means, for example, that if you want to compute the inner product between two vectors `x` and `y`, you can just write `x.dot(y)` or `x @ y` without needing to transpose the `x`. If `x` and `y` are 2-D arrays, however, then it *does* matter whether they are row-vectors or column-vectors, and hence you might need to transpose accordingly.

4. You are *not* required to write unit tests for these questions, and we will not try to "trick" you during the grading process. The purpose here is just to learn useful `numpy` skills. Nonetheless, you may find it useful to write tests to verify the correctness of your solutions.

Problems:

1. Given matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$, compute and return $\mathbf{AB} - \mathbf{C}$ (i.e., right-multiply matrix $\mathbf{A}$ by matrix $\mathbf{B}$, and then subtract $\mathbf{C}$). Use `@`, `dot`, `np.dot`. [**1 pts**]

2. Given matrix $\mathbf{A}$, return a vector with the same number of rows as $\mathbf{A}$ but that contains all ones. Use `np.ones`. [**1 pts**]

3. Given matrix $\mathbf{A}$, return a matrix with the same shape and contents as $\mathbf{A}$ *except* that the diagonal terms ($\mathbf{A}_{ii}$ for every valid i) are all zero. [**1 pts**]

4. Given matrix $\mathbf{A}$ and integer i, return the sum of all the entries in the ith row, i.e., $\sum_j \mathbf{A}_{ij}$. Do **not** use a loop, which in Python is very slow. Instead use the `np.sum` function. [**1 pts**]

5. Given matrix $\mathbf{A}$ and scalars c, d, compute the arithmetic mean (you can use `np.mean`) over all entries of A that are between c and d (inclusive). In other words, if $\mathcal{S} = \{(i,j) : c \leq \mathbf{A}_{ij} \leq d\}$, then compute $\frac{1}{|\mathcal{S}|} \sum_{(i,j)\in\mathcal{S}} \mathbf{A}_{ij}$. [**2 pts**]

6. Given an $(n \times n)$ matrix $\mathbf{A}$ and integer k, return an $(n \times k)$ matrix containing the right-eigenvectors of $\mathbf{A}$ corresponding to the k eigenvalues of $\mathbf{A}$ with the largest magnitude. Use `np.linalg.eig` to compute eigenvectors. [**2 pts**]

7. Given square matrix $\mathbf{A}$ and column vector $\mathbf{x}$, use `np.linalg.solve` to compute $\mathbf{A}^{-1}\mathbf{x}$. Do **not** use `np.linalg.inv` or `** -1` for any part of your solution, as that is numerically unstable and can give inaccurate results. [**2 pts**]

8. Given an n-vector $\mathbf{x}$ and a non-negative integer k, return a $n \times k$ matrix consisting of k copies of $\mathbf{x}$. You can use numpy methods such as `np.newaxis`, `np.atleast_2d`, and/or `np.repeat`. [**2 pts**]

9. Given a matrix $\mathbf{A}$ with n rows, return a matrix that results from **randomly permuting** (use `np.random.permutation`) the rows (but not the columns) in $\mathbf{A}$. Do *not* modify the input array $\mathbf{A}$. [**2 pts**]

10. Given an $(m \times n)$ matrix $\mathbf{A}$, return the m-vector resulting from computing the mean within each row of $\mathbf{A}$. Use `np.mean` and the `axis` parameter. [**2 pts**]

11. Given positive integers n and k, first generate a random 1-d array $\mathbf{A}$ of n integers (each between 0 and k, inclusive). Replace all the even numbers in $\mathbf{A}$ with -1. Then, return the result. [**2 pts**]

12. Given a matrix $\mathbf{A}$ with n rows and an n-vector $\mathbf{b}$, add $\mathbf{b}$ to every column of $\mathbf{A}$ and return the result. **Hint**: convert $\mathbf{b}$ to a 2-d array, either using `np.newaxis` or `np.reshape`. [**2 pts**]

13. Given a 3-d array $\mathbf{A}$ of size $(n \times m \times m)$, which represent a collection of n images (each with $m \times m$ pixels), create and return 2-d array $\mathbf{B}$ of size $(m^2 \times n)$ such that each column i contains the image pixels of image i in row-major order (i.e., `B[0,0]` should contain the value of the upper-left-most pixel in the first image, and `B[m*m-1,0]` should contain the value of the lower-right-most pixel in the first image). [**4 pts**]

3 Step-wise Classification [35 pts]

For the tasks below, write your code in a file called `homework1_smile_WPIUSERNAME.py`, and put your experimental results in `homework1_smile_WPIUSERNAME.pdf`.

In this part of the assignment you will train a very simple smile classifier that analyzes a grayscale image $\mathbf{x} \in \mathbb{R}^{24\times 24}$ and outputs a prediction $\hat{y} \in \{0, 1\}$ indicating whether the image is smiling (1) or not (0). The classifier will make its decision based on very simple **features** of the input image consisting of *binary comparisons* between pixel values. Each feature is computed as

$$\mathbb{I}[\mathbf{x}_{r_1,c_1} > \mathbf{x}_{r_2,c_2}]$$

where $r_i, c_i \in \{0, 1, 2, \ldots, 23\}$ are the row and column indices, respectively, and $\mathbb{I}[\cdot]$ is an indicator function whose value is 1 if the condition is true and 0 otherwise. In general, these features are not very good, but nonetheless they will enable the classifier to achieve an accuracy (f_{PC}) much better than just guessing or just choosing the dominant class. Based on these features, you should train an *ensemble* smile classifier using **step-wise classification** for $m = 6$ features. The output of the ensemble (1 if it thinks the image is smiling, 0 otherwise) is determined by the *average* prediction across all m members of the ensemble. If more than half of the m ensemble predictors $g^{(1)}, \ldots, g^{(m)}$ think that the image is smiling, then the ensemble says it's a smile; otherwise, the ensemble says it's not smiling.

Step-wise classification/regression is a **greedy algorithm**: at each round j, choose the jth feature (r_1, c_1, r_2, c_2) such that – when it is added to the set of $j - 1$ features that have *already been selected* – the accuracy (f_{PC}) of the overall classifier on the training set is maximized. More specifically, at every round j, consider *every possible* tuple of pixel locations (r_1, c_1, r_2, c_2): if you construct an ensemble with j predictors (the $j - 1$ you've already chosen, plus the current "candidate" (r_1, c_1, r_2, c_2)), is the resulting ensemble more accurate (in terms of PC on training data) than *any other* tuple of pixel locations during this round? If the current tuple is the best yet (for round j), then save it as your "best seen yet" for round j. If not, ignore it. Move on to the next possible tuple of pixel locations, and repeat until you've searched all of them. At the end of round j, you will have selected the best feature for that round, and you add it to your set of selected features. Once added, it stays in the set forever – it can never be removed. (Otherwise, it wouldn't be a greedy algorithm at all.) Now you move on to round $j + 1$ until you've completed $m = 6$ rounds.

To measure the ensemble's accuracy (f_{PC}), you should run it on *all* the images in the *training* set, and then compare the output of the ensemble to the corresponding ground-truth labels. At the end of the entire training procedure, you should estimate how well your "machine" (ensemble smile classifier) works on a set of images not used for training, i.e., the *test set*.

Skeleton code: While how you write your code is up to you (subject to the vectorization constraint and also basic readability); however, to get you started, we sketched in a few functions:

- `fPC (y, yhat)`: this takes in a vector of ground-truth labels and corresponding vector of guesses, and then computes the accuracy (PC). The implementation (in vectorized form) should only take 1-line.
- `measureAccuracyOfPredictors (predictors, X, y)`: this takes in a *set* of predictors, a set of images to run it on, as well as the ground-truth labels of that set. For each image in the image set, it runs the ensemble to obtain a prediction. Then, it computes and returns the accuracy (PC) of the predictions w.r.t. the ground-truth labels.
- `stepwiseRegression (trainingFaces, trainingLabels, testingFaces, testingLabels)`: I've included some visualization code, but otherwise it's empty. You need to implement the step-wise classification described above.

Tasks:

1. Download from Canvas the starter Python file `homework1_smile.py` as well as the following data files: `trainingFaces.npy,trainingLabels.npy,testingFaces.npy,testingLabels.npy`.

2. Write code to train a step-wise classifier for $m = 6$ features of the binary comparison type described above; the greedy procedure should maximize f_{PC} (which you will also need to implement). At each round, the code should examine *every possible feature* (r_1, c_1, r_2, c_2).. Make sure your code is **vectorized** to improve run-time performance (wall-clock time). In particular, you should compute the values of a single predictor over *all* the images in one-fell-swoop *without* using a loop. (It's also possible to compute the values of *multiple* predictors over all images in one-fell-swoop, but this is not required.) [**24 pts**].

3. Write code to analyze how training/testing accuracy changes as a function of number of examples $n \in \{400, 600, 800, \ldots, 2000\}$ (implement this in a for-loop):

   (a) Run your step-wise classifier training code only on the first n examples of the *training set*.

   (b) Measure and record the *training accuracy* of the trained classifier on the n examples.

   (c) Measure and record the *testing accuracy* of the classifier on the (entire) *test set*.

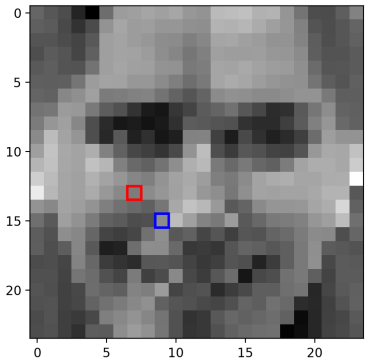
   Important: you **must** write code (a simple loop) to do this – do **not** just do it manually for each value of n. This is good experimental practice in general and is especially important in machine learning to ensure reproducibility of results. Using the entire training set, you should achieve a test accuracy of at least 75%. [**8 pts**].

4. In a PDF document (you can use whatever tool you like – LaTeX, Word, Google Docs, etc. – but make sure you export to PDF), show the training accuracy and testing accuracy as a function of n. Please use the following simple format:

```
n trainingAccuracy testingAccuracy
400 ... ...
600 ... ...
800 ... ...
1000 ... ...
1200 ... ...
1400 ... ...
1600 ... ...
1800 ... ...
2000 ... ...
```

   Moreover, characterize in words (and write them in the PDF) how the training accuracy and testing accuracy changes as a function of n, *and* how the two curves relate to each other; what trends do you observe? [**4 pts**]

5. **Visualizing the learned features**: It is important in empirical machine learning to visualize what was actually learned during training. This can be useful for debugging to make sure that your training code is working as it should, and also to make sure your training and testing sets are selected wisely. For $n = 2000$, visualize the $m = 6$ features that were learned by (a) displaying any face image from the test set; and (b) drawing a square around the specific pixel locations ((r_1, c_1) and (r_2, c_2)) that are examined by the feature. You can use the example code in the `homework1_smile.py` template to render the image. Insert the graphic (just one showing all 6 features) into your PDF file. [**4 pts**]. Here's an example that shows just one feature:



Tip on vectorization: Implement your training algorithm so that, for any particular feature (r_1, c_1, r_2, c_2), *all* the feature values (over all the n training images) are extracted at once using `numpy` – do not use a loop. Also, even after vectorizing your code, you will still have some nested `for`-loops to iterate over the different pairs of pixels; that is fine. Also, do *not* use `np.vectorize`.

4 Discrete Autoregression [22 pts]

Here you will tackle a very different machine learning problem about text rather than images. In particular, you will both train and test an autoregressive model to generate sentences of English text. The training dataset ("TinyStories") you will use consists of children's stories; hence, the language is quite simple. (Brace yourself for lots of sentences about a little girl named Lily.) In particular, you will train a "3-gram" (i.e., sentences are modeled in terms of probabilities of 3 words appearing in sequence) word prediction model for a fixed vocabulary of 501 words (which includes the end-of-sentence word "."). This involves estimating several probability distributions over each word X_t (for $t = 1, 2, \ldots$):

1. $P(X_1 = i)$ for each word $i \in \{1, \ldots, 501\}$. This is the probability that word i *begins* a sentence.
2. $P(X_2 = j \mid X_1 = i)$ for each second word $j \in \{1, \ldots, 501\}$ and for each (already sampled) first word i. This is the probability that word j is the *second* word in a sentence, given that word i was already chosen as the first word.
3. $P(X_{t+2} = k \mid X_t = i, X_{t+1} = j)$. Whereas the previous two probability distributions constituted the "base case" of the model, this distribution represents the "recursive case": Given the previous two already selected words i and j, this function gives the probability of choosing word k next.

To get started, you'll first need to install the Huggingface `datasets` Python package (e.g., `pip install datasets`) and then download the "TinyStories" dataset (the latter is performed automatically when the starter code is run). Alternatively, and to save your own disk space and electricity, you can use Google Colab.

Tasks:

1. **Training**: Define and estimate the three probability distributions described above on the first 100,000 stories in the TinyStories dataset. (It's fine but not necessary to train on more data.) To do so, you can apply the definition of conditional probability ($P(A|B) = P(A, B)/P(B)$) and count how many times each word sequence occurs and then "normalize" these counts to form probability distributions. For instance, to estimate $P(X_2 = \text{HAPPY} \mid X_1 = \text{THE}) = P(X_2 = \text{HAPPY}, X_1 = \text{THE})/P(X_1 = \text{THE})$, you should count how many times the words "THE HAPPY" occur as the first two words in a sentence; then, divide that count by the total number of times that "THE" occurs at the start of a sentence.

When processing each story, make sure to ignore any sentence that contains a word that is not in the top 500 word list. **[15 pts.]**

2. **Testing/Inference**: Implement code to autoregressively sample from these probability distributions. Then, run your code to generate 100 "sentences". To generate a sentence, you first randomly sample a word $i \sim P(X_1 = i)$. Next, after having chosen the first word, you sample a second word $j \sim P(X_2 = j \mid X_1 = i)$. Then, iteratively, *until you generate the end-of-sentence word "."*, you sample the next word $k \sim P(X_{t+2} = k \mid X_t = i, X_{t+1} = j)$ given the two previously generated words, thereby extending the generated sentence by one word in each iteration. This is similar to the generative process used in large language models (LLMs) such as ChatGPT, except that (1) LLMs use a neural network to estimate probability distributions instead of fixed look-up tables like in this assignment, and (2) LLMs consider the entire history of words $X_1, \ldots, X_t$ – not just the previous two – when generating word X_{t+1}. Note that, since the 3-gram model you are training is quite weak, some of the "sentences" you generate will be grammatically incorrect. This is ok. **[7 pts.]**

For the tasks above, complete the code implementation in the file `homework1_autoregression_WPIUSERNAME.py` (which contains skeleton code to download the dataset, etc.), and put your generated sentences in `homework1_autoregression_WPIUSERNAME.txt`.