Homework 1 – Machine Learning (CS4342, Whitehill, Spring 2022)

This homework is intended to help you (1) learn (or refresh your understanding of) how to implement linear algebraic operations in Python using `numpy` (to which we refer in the code below as `np`); (2) practice implementing one of the simpler machine learning algorithms: step-wise classification.

1 Part 1: Python and numpy

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_WPIUSERNAME.py` (e.g., `homework1_lleshin.py`). See the starter file `homework1_template.py`. In all problems, you may assume that the the dimensions of the matrices and/or vectors are compatible for the requested mathematical operations.

Note 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]`.

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

Note 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)` 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.

1. Given matrices $\mathbf{A}$ and $\mathbf{B}$, compute and return an expression for $\mathbf{A} + \mathbf{B}$. [**1 pts**]
   Answer (freebie!): While it is completely valid to use `np.add(A, B)`, this is unnecessarily verbose; you really should make use of the "syntactic sugar" provided by Python's/`numpy`'s operator overloading and just write: `A + B`. Similarly, you should use the more compact notation for the rest of the questions as well.

2. 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` or `np.dot`. [**2 pts**]

3. Given matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$, return $\mathbf{A} \odot \mathbf{B} + \mathbf{C}^\top$, where $\odot$ represents the element-wise (Hadamard) product and $\top$ represents matrix transpose. In `numpy`, the element-wise product is obtained simply with `*`. [**2 pts**]

4. Given column vectors $\mathbf{x}$ and $\mathbf{y}$ and square matrix $\mathbf{S}$, compute $\mathbf{x}^\top \mathbf{S} \mathbf{y}$. [**2 pts**]

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

6. 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. [**3 pts**]

7. Given square matrix $\mathbf{A}$ and (scalar) α, compute $\mathbf{A} + \alpha\mathbf{I}$, where $\mathbf{I}$ is the identity matrix with the same dimensions as $\mathbf{A}$. Use `np.eye`. [**2 pts**]

8. Given matrix $\mathbf{A}$ and integers i, j, return the ith column of the jth row of $\mathbf{A}$, i.e., $\mathbf{A}_{ji}$. [**2 pts**]

9. 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. [**4 pts**]

10. 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 $\frac{1}{|\mathcal{S}|} \sum_{(i,j)\in\mathcal{S}} \mathbf{A}_{ij}$. **[5 pts]**

11. 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. **[5 pts]**

12. 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` to compute the inverse explicitly; this is numerically unstable and can, in some situations, give incorrect results. **[4 pts]**

13. 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`. **[3 pts]**

14. 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 $\mathbf{A}$. **[3 pts]**

2 Part 2: Step-wise Classification

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:

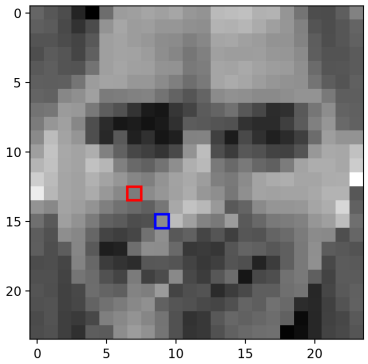
1. Download from Canvas the starter Python file `homework1_smile.py` as well as the following data files: `trainingFaces.npy,trainingLabels.npy,testingFaces.npy,testingLabels.npy`.
2. Write code to train a step-wise classifier for $m = 6$ features of the binary comparison type described above; the greedy 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.) [**25 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. **[3 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.