Homework 1 – Machine Learning (CS453X, Whitehill, Spring 2018)

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 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_jrwhitehill.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, you should "translate" from 1-based indexing to 0-based indexing as appropriate.

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

1. Given matrices $\mathbf{A}$ and $\mathbf{B}$, compute and return an expression for $\mathbf{A}+\mathbf{B}$. **[2 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 (and arguably more elegant) 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 $\mathbf{y}$, compute the inner product of $\mathbf{x}$ and $\mathbf{y}$ (i.e., $\mathbf{x}^\top\mathbf{y}$). **[2 pts]**

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

6. 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]**

7. Given (invertible) matrix $\mathbf{A}$, compute $\mathbf{A}^{-1}$. **[2 pts]**

   (Now that you know how to compute the inverse of a matrix, you should **almost never need to do so ever again**. The reasons are that (1) in most numerical algorithms, you are not interested in the matrix inverse itself but rather in the inverse **multiplied** by something else (e.g., a vector); and (2) computing the matrix inverse explicitly is numerically unstable.)

8. 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]**

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

10. 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]**

11. Given matrix $\mathbf{A}$ and scalars c, d, compute the arithmetic 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 $\frac{1}{|\mathcal{S}|}\sum_{(i,j)\in\mathcal{S}} \mathbf{A}_{ij}$. Use `np.nonzero` along with `np.mean`. [**5 pts**]

12. 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 largest eigenvalues of $\mathbf{A}$. Use `np.linalg.eig` to compute eigenvectors. [**5 pts**]

13. Given square matrix $\mathbf{A}$ and column vector $\mathbf{x}$, use `np.linalg.solve` to compute $\mathbf{A}^{-1}\mathbf{x}$. [**4 pts**]

14. Given square matrix $\mathbf{A}$ and row vector $\mathbf{x}$, use `np.linalg.solve` to compute $\mathbf{x}\mathbf{A}^{-1}$. Hint: $\mathbf{AB} = (\mathbf{B}^\top\mathbf{A}^\top)^\top$. [**4 pts**]

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 a smile classifier using **step-wise classification** for $m = 5$ features. 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 is maximized. Once you select a feature, it stays in the set forever – it can never be removed. (Otherwise, it wouldn't be a greedy algorithm at all.)

Your training code should perform step-wise classification by analyzing the images and maximizing the accuracy only on the *training set*. To estimate how well the "machine" (smile classifier) you are building would work on a set of images not used for training, you should then measure it accuracy on the *test set*.

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 $m = 5$ 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) [**25 pts**].

3. Write code to analyze how training/testing accuracy changes as a function of number of examples $n \in \{400, 800, 1200, 1600, 2000\}$ (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*.

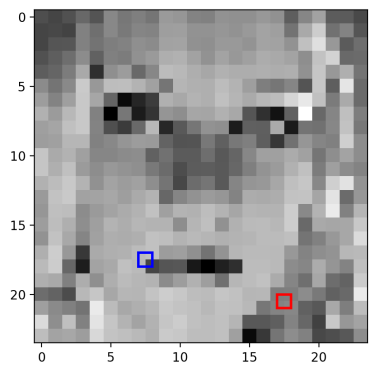
   Important: you **must** write code (a simple loop) to do this – do **not** just do it manually for each value of n. This is good experimental practice in general and is especially important in machine learning to ensure reproducibility of results. [**8 pts**].

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 ... ...
800 ... ...
1200 ... ...
1600 ... ...
2000 ... ...
```

Moreover, characterize in words 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 very important in empirical machine learning work 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 $m = 5$ 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 5 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. Even after vectorizing my own code, running the experiments required in this assignment took about 1 hour (on a single CPU of a MacBook 2016 laptop).