COMP 3105 Introduction to Machine Learning Assignment 3

Instructor: Junfeng Wen (junfeng.wen [AT] carleton.ca)

Fall 2023
School of Computer Science
Carleton University

Deadline: 11:59 pm, Sunday, Nov. 12, 2023

Instruction: **Submit the following three files** to Brightspace for marking

- A Python file `A3codes.py` that includes all your implementations of the required functions
- A pip requirements file named `requirements.txt` that specifies the running environment including a list of Python libraries/packages and their versions required to run your codes.
- A PDF file `A3report.pdf` that includes all your answers to the written questions. It should also specify your team members (names and student IDs). Please clearly specify question/sub-question numbers in your submitted PDF report so TAs can see which question you are answering.

Do not submit a compressed file, or it may result in a mark deduction. We recommend trying your code using Colab or Anaconda/Virtualenv before submission.

Rubrics: This assignment is worth 15% of the final grade. Your codes and report will be evaluated based on their scientific qualities including but not limited to: Are the implementations correct? Is the analysis rigorous and thorough? Are the codes easily understandable (with comments)? Is the report well-organized and clear?

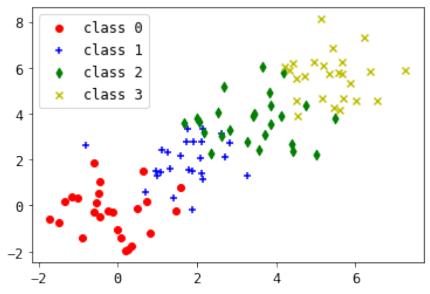
Policies:

- You can finish this assignment in groups of two. All members of a group will receive the same mark when the workload is shared.
- You may consult others (classmates/TAs/LLMs) about general ideas but don't share codes/answers. Please specify in the PDF file any individuals or programs (e.g., ChatGPT) you consult for the assignment. If you use large language models (LLMs), clearly show us how you use it. Any group found to cheat or violating this policy will receive a score of 0 for this assignment.
- Remember that you have **three** excused days *throughout the term* (rounded up to the nearest day), after which no late submission will be accepted.
- Specifically for this assignment, you can use libraries with general utilities, such as matplotlib, numpy/scipy, cvxopt, and pandas for Python. **However, you must implement everything by yourselves without using any pre-existing implementations of the algorithms or any functions from an ML library (such as scikit-learn)**. The goal is for you to really understand, step by step, how the algorithms work.

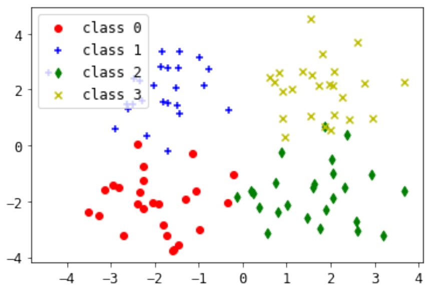
Question 0: Data Generation & Helper Functions

In this assignment, you will implement various algorithms, including multi-class classification, principle component analysis (PCA) and k-means clustering. In the `A3helpers.py` file, we provided several functions that will be used in this assignment:

- `augmentX, unAugmentX` adds/removes the augmented column of all ones to/from the input matrix X.
- `convertToOneHot` converts the multi-class labels into one-hot encoding.
- `generateData` is used to generate training and test data points. There are two generative models, each generate a type of data in 2D space (controlled by the `gen_model` argument). The following shows what the generated data may look like:



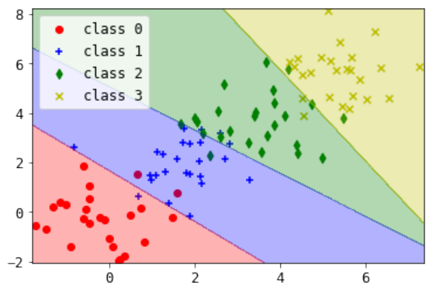
(a) Generative model 1



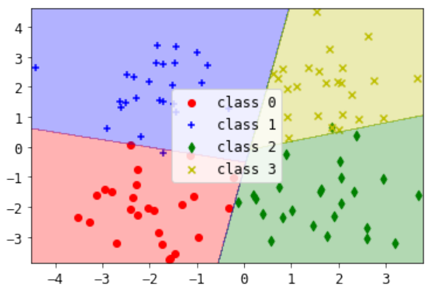
(b) Generative model 2

Figure 1: Generated data

- `plotModel` can be used to visualize your trained/learned models once you finish your implementation. The models you learned may look like the following:



(a) Generative model 1



(b) Generative model 2

Figure 2: Classification Boundaries

- `plotImg` can help you visualize the image for Question 2.

You can also use the `A3testbed.py` to visualize your results.

Question 1 (5%) Linear Multi-Class classifier

In this question, you will implement k-class classification using the multinomial deviance loss (or equivalently the KL divergence or cross-entropy) from scratch, in Python using NumPy/SciPy, and evaluate their performances on the synthetic datasets from above.

The input vectors are assumed to be **augmented** in this question (i.e. we assume that the input matrix `X` or `Xtest` has a column of all 1s). All of the following functions must be able to handle arbitrary $n > 0$, $m > 0$, $d > 0$ and $k > 1$. The vectors and matrices are represented as NumPy arrays. Your functions shouldn't print additional information to the standard output.

(a) (2%) Implement a Python function

```
W = minMulDev(X, Y)
```

that takes an $n \times d$ input matrix `X` and an $n \times k$ label matrix `Y` (where each row is a one-hot encoding of the label), and returns a $d \times k$ matrix of weights/parameters `W` corresponding to the solution of the multinomial deviance loss:

$$W^* = \operatorname*{argmin}_{W \in \mathbb{R}^{d \times k}} \frac{1}{n} \sum_{i=1}^{n} \log(\mathbf{1}_k^\top e^{W^\top \mathbf{x}_i}) - \mathbf{y}_i^\top W^\top \mathbf{x}_i$$

where the exponential and logarithm are elementwise. This problem can be solved by `scipy.optimize.minimize`.

Note that `scipy.optimize.minimize` can only handle 1D array as unknowns. Therefore, you will need to reshape your unknown variable to a $d \times k$ matrix W **inside** your objective function.

Numerical issue: `exp` and/or `log` may cause numerical issues. You may want to check the `scipy.special.logsumexp` function and see how its `axis` argument works.

(b) (1%) Implement a Python function

```
Yhat = classify(Xtest, W)
```

that takes an $m \times d$ input matrix `Xtest` and a $d \times k$ matrix of weights/parameters `W`, and returns an $m \times k$ prediction matrix `Yhat`. Recall that the prediction of a data point $\mathbf{x}$ is given by $\widehat{\mathbf{y}} = \text{indmax}(W^\top \mathbf{x})$, a one-hot vector. To predict on the whole test dataset, $\widehat{Y}_{\text{test}} = \text{indmax}(X_{\text{test}} W)$, where indmax is applied to each row.

(c) (1%) Implement a Python function

```
acc = calculateAcc(Yhat, Y)
```

that takes an $m \times k$ prediction matrix `Yhat` and an $m \times k$ label matrix `Y`, and returns a scalar `acc` that is the accuracy of the prediction. Recall that the one-hot encoding of the label shows you which one is the ground-truth/predicted class. The prediction would be correct if it matches the ground-truth label.

Debug Hint: Once you finish all the previous functions, you can run `A3testbed.py` with a controlled random seed of zero. You should get an accuracy of 87% on generative model 1 and 94% on generative model 2.

(d) (1%) In this part, you will evaluate your implementation on the synthetic datasets from above.

We have implemented a helper function

```
train_acc, test_acc = synClsExperiments(minMulDev, classify, calculateAcc)
```

in `A3helpers.py` that calls your functions, and returns a 4×2 matrix `train_acc` of average training accuracies and a 4×2 matrix `test_acc` of average test accuracies (See Table 1 and Table 2 below).

In the PDF file, report the *averages* (over 10 runs) for each accuracy in two tables (one for training and the other for test).

Table 1: Training accuracies with different number of training dataset sizes

n	Model 1	Model 2
16		
32		
64		
128		

Table 2: Test accuracies with different number of training dataset sizes

n	Model 1	Model 2
16		
32		
64		
128		

Runtime: For efficient implementation, it will run for about 10 seconds. You may want to reduce the number of runs when testing your code.

Looking at your tables from above, analyze the results and discuss any findings you may have and the possible reason behind them.

Question 2 (6%) Principle Component Analysis (PCA)

In this question, you will implement principle component analysis (PCA) from scratch, in Python using NumPy/SciPy, and check how it works for synthetic and real-world data.

The input vectors are assumed to be **augmented** in this question (i.e. we assume that the input matrix `X` or `Xtest` has a column of all 1s). All of the following functions must be able to handle arbitrary $n > 0$, $m > 0$, $d > 0$ and $1 \leq k \leq d$. The vectors and matrices are represented as NumPy arrays. Your functions shouldn't print additional information to the standard output.

(a) (2%) Implement a Python function

```
U = PCA(X, k)
```

that takes an $n \times d$ input matrix `X` and a scalar integer `k` ($1 \leq k \leq d$), and returns a $k \times d$ matrix of projecting directions `U` whose *rows* correspond to the top-k projecting directions with the largest variances. That is, it consists of the eigenvectors of $X^\top X$ with largest eigenvalues. This problem can be solved by `scipy.linalg.eigh`.

Recall that PCA is applied **after** subtracting the mean $\boldsymbol{\mu} = \frac{1}{n} \sum_{i=1}^{n} \mathbf{x}_i$ of X from the dataset. In other words, you need to first compute the mean $\boldsymbol{\mu} \in \mathbb{R}^d$ of the input matrix X, subtract it from every row of X, and finally compute the eigenvectors of $X^\top X$.

Hints: Please read the doc of `scipy.linalg.eigh` very carefully, including how it orders the eigenvalues in *ascending* order and the shape of the eigenvectors. You can also check its `subset_by_index` argument.

(b) (1%) The A3files.zip includes an image dataset, A3train.csv, of shirt images taken from the Fashion-MNIST dataset. Every row in the file is one training example. Once loaded, you can call the `plotImgs` helper function to see some samples of the images as in Fig. 3.

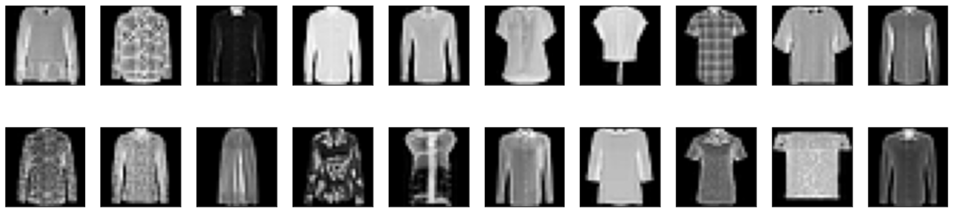
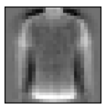


Figure 3: Sample images from the dataset

Your task here is to call your `PCA` on this dataset with $k = 20$ to learn the matrix U, whose *rows* represent the "prototypes" of the dataset. Then call the `plotImgs` helper function with the U to see the "eigen-shirts". In the PDF file, show us the "eigen-shirts" you get and discuss any findings you may have from them.

Debug Hint: One of your "eigen-shirts" should look like the following (maybe with flipped color)



(c) (1%) Implement a Python function

```
Xproj = projPCA(Xtest, mu, U)
```

that takes an $m \times d$ input matrix `Xtest`, a $d \times 1$ *training* mean vector `mu` (i.e., $\boldsymbol{\mu}$ from part (a)) and a $k \times d$ projection matrix `U`, and returns an $m \times k$ projected matrix Xproj that consists of the projected features/representations of Xtest onto the directions U.

Recall that the projection of a data point $\mathbf{x}$ is given by $U(\mathbf{x} - \boldsymbol{\mu})$. To project the whole test dataset: $(X_{\text{test}} - \boldsymbol{\mu}^\top)U^\top$. You may want to check the broadcast rules for NumPy arrays.

(d) (1%) In this part, you will evaluate the effect of PCA on the classification problem. Implement a Python function

```
train_acc, test_acc = synClsExperimentsPCA()
```

that returns a 2×2 matrix train_acc of average training accuracies and a 2×2 matrix test_acc of average test accuracies (See Table 3 and Table 4 below). It repeats 100 runs as follows

```
def synClsExperimentsPCA():
    n_runs = 100
    n_train = 128
    n_test = 1000
    dim_list = [1, 2]
    gen_model_list = [1, 2]
    train_acc = np.zeros([len(dim_list), len(gen_model_list), n_runs])
    test_acc = np.zeros([len(dim_list), len(gen_model_list), n_runs])
    for r in range(n_runs):
        for i, k in enumerate(dim_list):
            for j, gen_model in enumerate(gen_model_list):
                Xtrain, Ytrain = generateData(n=n_train, gen_model=gen_model)
                Xtest, Ytest = generateData(n=n_test, gen_model=gen_model)

                U = PCA(Xtrain, k)
                Xtrain_proj = # TODO: call your projPCA to find the new features
                Xtest_proj = # TODO: call your projPCA to find the new features

                Xtrain_proj = augmentX(Xtrain_proj) # add augmentation
                Xtest_proj = augmentX(Xtest_proj)

                W = minMulDev(Xtrain_proj, Ytrain) # from Q1
                Yhat = classify(Xtrain_proj, W) # from Q1
                train_acc[i, j, r] = calculateAcc(Yhat, Ytrain) # from Q1

                Yhat = classify(Xtest_proj, W)
                test_acc[i, j, r] = calculateAcc(Yhat, Ytest)

    # TODO: compute the average accuracies over runs
    # TODO: return 2-by-2 train accuracy and 2-by-2 test accuracy
```

In the PDF file, report the *averages* (over 100 runs) for each accuracy in two tables (one for training and the other for test).

Table 3: Training accuracies with different number of dimensions

Dim k	Model 1	Model 2
1		
2		

Table 4: Test accuracies with different number of dimensions

Dim k	Model 1	Model 2
1		
2		

Runtime: For efficient implementation, it will run for about 30 seconds. You may want to reduce the number of runs when testing your code.

(e) (1%) Looking at your tables from above, analyze the results and discuss any findings you may have and the possible reason behind them.

Question 3 (4%) k-means

In this question, you will implement k-means from scratch, in Python using NumPy/SciPy, and check how it works for synthetic data.

All of the following functions must be able to handle arbitrary $n > 0$, $m > 0$, $d > 0$ and $1 < k < n$. The vectors and matrices are represented as NumPy arrays. Your functions shouldn't print additional information to the standard output.

(a) (2%) Implement a Python function

```
Y, U, obj_val = kmeans(X, k, max_iter=1000)
```

that takes an $n \times d$ input matrix `X`, a scalar integer `k` ($1 < k < n$) and a scalar integer `max_iter` indicating the maximum number of iterations, and returns an $n \times k$ membership/assignment matrix `Y`, a $k \times d$ matrix of cluster centers `U` and a scalar `obj_val` that is the objective value achieved by the solutions. It solves the following problem locally

$$\begin{aligned} \min_{Y \in \mathcal{Y}, U} \quad & J(Y, U) = \frac{1}{2n} \|X - YU\|_F^2 \\ \text{where } & \mathcal{Y} = \{Y | Y \in \{0,1\}^{n \times k}, Y\mathbf{1}_k = \mathbf{1}_n\} \end{aligned} \tag{1}$$

by alternating the following steps (with an initial guess of U)

1. Fix U, solve for an optimal Y^*. Specifically, compute the pairwise distance matrix D between the input matrix X and the given centers U such that $D_{ij} = \|\mathbf{x}_i - \mathbf{u}_j\|_2^2$ where $\mathbf{x}_i$ is the ith row of X and $\mathbf{u}_j$ is the jth row of U. Then assign each point $\mathbf{x}_i$ to the closest cluster center. That is, the ith row of Y is given by
$$Y_{i:} = [0, \dots, \underbrace{1}_{j^*\text{th position}}, \dots, 0]$$
where $j^* = \operatorname{argmin}_j D_{ij}$.
2. Fix Y, solve for an optimal U^*, which is given by $(Y^\top Y)^{-1} Y^\top X$.

After convergence, compute the objective value $J(Y, U)$ (see Eq. (1)) achieved by the solutions Y and U. The function should look like the following

```python
def kmeans(X, k, max_iter=1000):
    n, d = X.shape
    assert maxIter > 0
    U = # TODO: Choose k random points from X as initial centers
    for i in range(max_iter):
        D = # TODO: Compute pairwise distance between X and U
        Y = # TODO: Find the new cluster assignments
        old_U = U
        U = # TODO: Update cluster centers
        if np.allclose(old_U, U):
            break
    obj_val = # TODO: Compute objective value
    return Y, U, obj_val
```

Note: Do not control random seed inside your function. You may find `scipy.spatial.distance.cdist`, `numpy.linalg.solve` and/or `numpy.linalg.inv` helpful.

Numerical issue: $Y^\top Y$ may not be invertible when we have empty clusters. To handle this situation, you need to add a small constant before inverting the matrix: `Y.T @ Y + 1e-8 * np.eye(k)`. If this is still insufficient, try to increase `1e-8` or debug your code using Colab or Anaconda.

(b) (1%) Implement a Python function

```
Y, U, obj_val = repeatKmeans(X, k, n_runs=100)
```

that takes an $n \times d$ input matrix `X`, a scalar integer `k` ($1 < k < n$) and a scalar integer `n_runs` indicating the number of restarts, and returns an $n \times k$ membership/assignment matrix `Y`, a $k \times d$ matrix cluster centers `U` and a scalar `obj_val` that corresponds to the best solutions and objective value obtained during `n_runs` of your `kmeans`. It runs as follows

```python
def repeatKmeans(X, k, n_runs=100):
    best_obj_val = float('inf')
    for r in range(n_runs):
        Y, U, obj_val = kmeans(X, k)
        # TODO: Compare obj_val with best_obj_val. If it is lower,
        #       then record the current Y, U and update best_obj_val

    # TODO: Return the best Y, U and best_obj_val
```

Recall that the problem in Eq. (1) is NP-hard and k-means only solves it locally instead of globally, which means every time you run `kmeans`, it may give you a different solution. That's why you should set the initial guess U *randomly.* However, we can call `kmeans` multiple (specifically, `n_runs`) times, and choose the solutions that give the *minimum* objective value among those runs.

Note: Do not set random seed in your function.

(c) (1%) Implement a Python function

```
obj_val_list = chooseK(X, k_candidates=[2,3,4,5,6,7,8,9])
```

that takes an $n \times d$ input matrix `X` and a Python list of integers `k_candidates`, and returns a list of objective values obtained for each k value in the candidate list when calling `repeatKmeans` with that k value.

Once you finish this function, call it with the following

```python
Xtrain, Ytrain = generateData(n=100, gen_model=2)
obj_val_list = chooseK(Xtrain)
```

and report the `obj_val_list` in the following table.

Table 5: Objective values for different k

k	2	3	4	5	6	7	8	9
`obj_val`								

In the PDF file, explain how many number of clusters k you think is appropriate for this dataset, and why.

Note: Do not control random seed in your function and use the default input arguments.