COMP 3105A Introduction to Machine Learning Assignment 2

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

Fall 2025
School of Computer Science
Carleton University

Deadline: 11:59 pm, Sunday, Oct. 19, 2025

Instruction:

Submit the following three files to Brightspace for marking

- A Python file `A2codes.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. The current template uses Python=3.10.
- A PDF file `A2report.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 will 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:

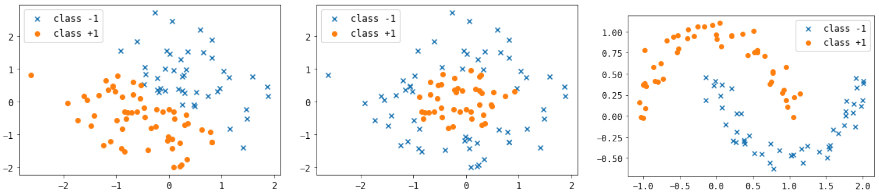
- You can finish this assignment in groups of two. All members of a group will receive the same mark when the workload is shared.
- You may consult others (classmates/TAs/LLMs) about general ideas but don't share codes/answers. Please specify in the PDF file any individuals or programs (e.g., ChatGPT) you consult for the assignment. If you use large language models (LLMs), clearly show us how you use them (either include the prompts and replies in the report, or include URL links showing chat history in the report). Any group found to cheat or violate 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, Helpers & A New Loss

In this assignment, we will focus on binary classification problems and solve them in several different ways. The class label is assumed to be -1 or $+1$. In the `A2helpers.py` file, we provided several functions that will be used in this assignment:

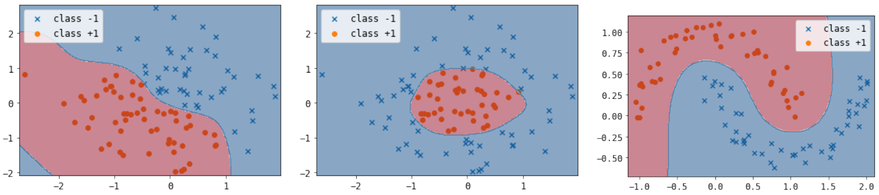
- `linearKernel, polyKernel, gaussKernel` are the kernel functions
- `generateData` is used to generate training and test data points. There are three generative models, each generates a type of data in 2D space (controlled by the `gen_model` argument). The following shows what the generated data may look like:



(a) Generative model 1 (b) Generative model 2 (c) Generative model 3

Figure 1: Generated data

- `plotModel, plotAdjModel, plotDualModel` 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 (c) Generative model 3

Figure 2: Classification boundaries

- `plotDigit` can help you visualize the digit image for Question 3.

You can also use the `A2testbed.py` to visualize your results.

ExpLinear Loss. In this assignment, you will use a new surrogate loss for binary classification (see Lecture 4: Output Transforms). Specifically, given the margin $m = y\widehat{z}$, the *ExpLinear loss* is defined as

$$L(m) = \max(0, -m) + e^{\min(0, -m)} = \begin{cases} 1 - m & \text{if } m \leq 0 \\ e^{-m} & \text{if } m > 0 \end{cases}$$

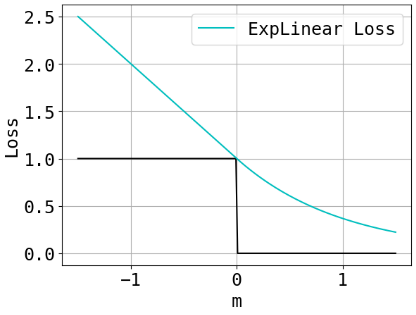


Figure 3: ExpLinear Loss

Question 1 (5%) Binary Classifier (Primal Form)

In this question, you will implement binary classification with different losses from scratch, in Python using NumPy/SciPy, and evaluate their performances on the synthetic datasets from above with different regularization hyper-parameters. You will learn some essential built-in functions like `scipy.optimize.minimize` to solve unconstrained problems and `cvxopt.solvers.qp` to solve quadratic programmings.

The input vectors are assumed to be **un-augmented** in this question (i.e. we do not add a constant feature of 1 to it). All of the following functions must be able to handle arbitrary $n > 0$ and $d > 0$. The vectors and matrices are represented as NumPy arrays. Your functions shouldn't print additional information to the standard output.

(a) (1%) Implement a Python function

```
w, w0 = minExpLinear(X, y, lamb)
```

that takes an $n \times d$ input matrix `X`, an $n \times 1$ target/label vector `y` and a regularization hyper-parameter `lamb` (i.e., $\lambda > 0$), and returns a $d \times 1$ vector of weights/parameters `w` and a scalar intercept `w0` corresponding to the solution of the regularized ExpLinear loss:

$$\mathbf{w}^*, w_0^* = \operatorname*{argmin}_{\mathbf{w}\in\mathbb{R}^d, w_0\in\mathbb{R}} \sum_{i=1}^{n} \max(0, -m_i) + e^{\min(0,-m_i)} + \frac{\lambda}{2}\|\mathbf{w}\|_2^2$$

where $m_i = y_i \cdot (\mathbf{x}_i^\top \mathbf{w} + w_0)$ is the margin of the ith data point. This problem can be solved by `scipy.optimize.minimize`.

(b) (1%) Implement a Python function

```
w, w0 = minHinge(X, y, lamb, stablizer=1e-5)
```

that takes an $n \times d$ input matrix `X`, an $n \times 1$ target/label vector `y` and a regularization hyper-parameter `lamb` (i.e., $\lambda > 0$), a positive scalar `stabilizer` (see below on numerical issue), and returns a $d \times 1$ vector of weights/parameters `w` and a scalar intercept `w0` corresponding to the solution of the regularized hinge loss:

$$\mathbf{w}^*, w_0^* = \operatorname*{argmin}_{\mathbf{w}\in\mathbb{R}^d, w_0\in\mathbb{R}} \sum_{i=1}^{n} \max(0, 1 - y_i \cdot (\mathbf{x}_i^\top \mathbf{w} + w_0)) + \frac{\lambda}{2}\|\mathbf{w}\|_2^2$$

Recall that the optimization can be expressed as a quadratic programming with the joint parameters $\begin{bmatrix} \mathbf{w} \\ w_0 \\ \boldsymbol{\xi} \end{bmatrix} \in \mathbb{R}^{d+1+n}$

$$\begin{aligned} \min_{\mathbf{w}, w_0, \boldsymbol{\xi}} \quad & \boldsymbol{\xi}^\top \mathbf{1}_n + \frac{\lambda}{2}\|\mathbf{w}\|_2^2 \\ \text{s.t.} \quad & \boldsymbol{\xi} \succeq \mathbf{0}_n \\ & \mathbf{1}_n - \Delta(\mathbf{y})(X\mathbf{w} + w_0\mathbf{1}_n) \preceq \boldsymbol{\xi} \end{aligned}$$

This problem can be solved by `cvxopt.solvers.qp`.

Numerical issue: Due to numerical issue, the `P` matrix for `cvxopt.solvers.qp` could fail to be stable even when you correctly specify it. To ensure numerical stability, you need to add a small positive stabilizer to the diagonal: `P = P + stabilizer * np.eye(n+d+1)`.

(c) (1%) Implement a Python function

```
yhat = classify(Xtest, w, w0)
```

that takes an $m \times d$ test matrix `Xtest` consisting of m (any) points to be predicted on, a $d \times 1$ vector of weights/parameters `w` and a scalar intercept `w0`, and returns an $m \times 1$ prediction vector `yhat`. Recall that the prediction of a data point $\mathbf{x}$ is given by $\widehat{y} = \text{sign}(\mathbf{x}^\top \mathbf{w} + w_0)$. Note that you don't have to loop over points to predict on the whole test dataset. Instead, you can simply calculate $\widehat{\mathbf{y}}_{\text{test}} = \text{sign}(X_{\text{test}}\mathbf{w} + w_0)$.

Your function must be able to handle arbitrary number of test points $m > 0$.

(d) (1%) In this part, you will evaluate your implementation with different regularization hyper-parameters. Implement a Python function

```
train_acc, test_acc = synExperimentsRegularize()
```

that returns a 4×6 matrix `train_acc` of average training accuracies and a 4×6 matrix `test_acc` of average test accuracies (See Table 1 and Table 2 below). It repeats 100 runs as follows

```
def synExperimentsRegularize():
    n_runs = 100
    n_train = 100
    n_test = 1000
    lamb_list = [0.001, 0.01, 0.1, 1.]
    gen_model_list = [1, 2, 3]
    train_acc_explinear = np.zeros([len(lamb_list), len(gen_model_list), n_runs])
    test_acc_explinear = np.zeros([len(lamb_list), len(gen_model_list), n_runs])
    train_acc_hinge = np.zeros([len(lamb_list), len(gen_model_list), n_runs])
    test_acc_hinge = np.zeros([len(lamb_list), len(gen_model_list), n_runs])

    # TODO: Change the following random seed to your GROUP ID
    np.random.seed(0)

    for r in range(n_runs):
        for i, lamb in enumerate(lamb_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)

                w, w0 = minExpLinear(Xtrain, ytrain, lamb)
                train_acc_explinear[i, j, r] = # TODO: compute accuracy on training set
                test_acc_explinear[i, j, r] = # TODO: compute accuracy on test set

                w, w0 = minHinge(Xtrain, ytrain, lamb)
                train_acc_hinge[i, j, r] = # TODO: compute accuracy on training set
                test_acc_hinge[i, j, r] = # TODO: compute accuracy on test set

    # TODO: compute the average accuracies over runs
    # TODO: combine accuracies (explinear and hinge)
    # TODO: return 4-by-6 train accuracy and 4-by-6 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).

Runtime. For efficient implementation, this function can be finished in around 20 seconds. Consider avoiding for loops and vectorizing your method if your code is running slow.

Table 1: Training accuracies with different hyper-parameters

λ	ExpLinear			Hinge		
	Model 1	Model 2	Model 3	Model 1	Model 2	Model 3
0.001						
0.01						
0.1						
1.0						

Table 2: Test accuracies with different hyper-parameters

λ	ExpLinear			Hinge		
	Model 1	Model 2	Model 3	Model 1	Model 2	Model 3
0.001						
0.01						
0.1						
1.0						

(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 2 (5%) Binary Classification (Adjoint Form)

In this question, you will implement binary classification with different losses (again) using the *adjoint formula* coming from the representer theorem, and evaluate their performances on the synthetic datasets from above with different kernels.

The input vectors are assumed to be **un-augmented** in this question (i.e. we do not add a constant feature of 1 to it). All of the following functions must be able to handle arbitrary $n > 0$ and $d > 0$. The vectors and matrices are represented as NumPy arrays. Your functions shouldn't print additional information to the standard output.

(a) (1%) Implement a Python function

```
a, a0 = adjExpLinear(X, y, lamb, kernel_func)
```

that takes an $n \times d$ input matrix `X`, an $n \times 1$ target/label vector `y`, a regularization hyper-parameter `lamb` (i.e., $\lambda > 0$), and a callable kernel function `kernel_func`, and returns an $n \times 1$ vector of weights/parameters `a` and a scalar intercept `a0` corresponding to the solution of the adjoint form of the regularized ExpLinear loss:

$$\boldsymbol{\alpha}^*, \alpha_0^* = \underset{\boldsymbol{\alpha}\in\mathbb{R}^n, \alpha_0\in\mathbb{R}}{\operatorname{argmin}} \sum_{i=1}^{n} \max(0, -m_i) + e^{\min(0,-m_i)} + \frac{\lambda}{2}\boldsymbol{\alpha}^\top K \boldsymbol{\alpha}$$

where $m_i = y_i \cdot (\mathbf{k}_i^\top \boldsymbol{\alpha} + \alpha_0)$ is the margin of the ith data point, $\mathbf{k}_i$ is the ith column of K and they are computed based on `X` and the kernel function `kernel_func`, e.g.,

```
K = kernel_func(X, X).
```

The following are examples of the kernel function constructed using the `lambda` function in Python:

```
kernel_func = lambda X1, X2:  linearKernel(X1, X2)
kernel_func = lambda X1, X2:  polyKernel(X1, X2, 2)
```

where the `linearKernel` and `polyKernel` can be found in the A2helpers.py.

This problem can be solved by `scipy.optimize.minimize`.

Debug hint: One way to verify your solution is to compare it to the solution from Q1(a). Recall that due to L_2 regularization, $\mathbf{w}^* = X^\top \boldsymbol{\alpha}^*$ so $\mathbf{x}_i^\top \mathbf{w}^* + w_0^*$ should be equal to $\mathbf{k}_i^\top \boldsymbol{\alpha}^* + \alpha_0^*$. That is, *when using linear kernel*, they should give you similar results with small numerical difference.

(b) (1%) Implement a Python function

```
a, a0 = adjHinge(X, y, lamb, kernel_func, stabilizer=1e-5)
```

that takes an $n \times d$ input matrix `X`, an $n \times 1$ target/label vector `y`, a regularization hyper-parameter `lamb` (i.e., $\lambda > 0$), a callable kernel function `kernel_func`, a positive scalar `stabilizer` (see below on numerical issue), and returns an $n \times 1$ vector of weights/parameters `a` and a scalar intercept `a0` corresponding to the solution of the adjoint form of the regularized hinge loss:

$$\boldsymbol{\alpha}^*, \alpha_0^* = \underset{\boldsymbol{\alpha}\in\mathbb{R}^n, \alpha_0\in\mathbb{R}}{\operatorname{argmin}} \sum_{i=1}^{n} \max(0, 1 - y_i \cdot (\mathbf{k}_i^\top \boldsymbol{\alpha} + \alpha_0)) + \frac{\lambda}{2}\boldsymbol{\alpha}^\top K \boldsymbol{\alpha}$$

where $\mathbf{k}_i$ is the ith column of K and they are computed based on `X` and the kernel function `kernel_func`.

Recall that the optimization can be expressed as a quadratic programming with the joint parameters $\begin{bmatrix} \boldsymbol{\alpha} \\ \alpha_0 \\ \boldsymbol{\xi} \end{bmatrix} \in \mathbb{R}^{2n+1}$

$$\begin{aligned} \min_{\boldsymbol{\alpha},\alpha_0,\boldsymbol{\xi}} \quad & \boldsymbol{\xi}^\top \mathbf{1}_n + \frac{\lambda}{2}\boldsymbol{\alpha}^\top K \boldsymbol{\alpha} \\ \text{s.t.} \quad & \boldsymbol{\xi} \succeq \mathbf{0}_n \\ & \mathbf{1}_n - \Delta(\mathbf{y})(K\boldsymbol{\alpha} + \alpha_0 \mathbf{1}_n) \preceq \boldsymbol{\xi} \end{aligned}$$

This problem can be solved by `cvxopt.solvers.qp`.

Numerical issue: Again due to numerical issue, the `P` matrix for `cvxopt.solvers.qp` could fail to be stable even when you correctly specify it. To ensure numerical stability, you need to add a small positive stabilizer to the diagonal: `P = P + stabilizer * np.eye(2*n+1)`.

Debug hint: One way to verify your solution is to compare to the solution from Q1(b). *When using linear kernel*, $\mathbf{w}^* = X^\top \boldsymbol{\alpha}^*$ so the solution here should be similar to that of Q1(b).

(c) (1%) Implement a Python function

```
yhat = adjClassify(Xtest, a, a0, X, kernel_func)
```

that takes an $m \times d$ test input matrix `Xtest` consisting of (any) points to be predicted on, a $n \times 1$ vector of weights/parameters `a`, a scalar intercept `a0`, an $n \times d$ **training** input matrix `X` and a kernel function `kernel_func`, and returns an $m \times 1$ prediction vector `yhat`. Recall that the prediction of a data point $\mathbf{x}$ is given by $\hat{y} = \text{sign}(\text{kernel\_func}(\mathbf{x}^\top, X)\boldsymbol{\alpha} + \alpha_0)$. Note that you don't have to loop over points to predict on the whole test dataset. Instead, you can simply calculate $\hat{\mathbf{y}}_{\text{test}} = \text{sign}(\text{kernel\_func}(X_{\text{test}}, X)\boldsymbol{\alpha} + \alpha_0)$.

Your function must be able to handle arbitrary number of test points $m > 0$.

(d) (1%) In this part, you will evaluate your implementation with different kernels. Implement a Python function

```
train_acc, test_acc = synExperimentsKernel()
```

that returns a 5×6 matrix `train_acc` of average training accuracies and a 5×6 matrix `test_acc` of average test accuracies (See Table 3 and Table 4 below). It repeats 10 runs as follows

```
def synExperimentsKernel():
    n_runs = 10
    n_train = 100
    n_test = 1000
    lamb = 0.001
    kernel_list = [linearKernel,
                   lambda X1, X2: polyKernel(X1, X2, 2),
                   lambda X1, X2: polyKernel(X1, X2, 3),
                   lambda X1, X2: gaussKernel(X1, X2, 1.0),
                   lambda X1, X2: gaussKernel(X1, X2, 0.5)]
    gen_model_list = [1, 2, 3]
    train_acc_explinear = np.zeros([len(kernel_list), len(gen_model_list), n_runs])
    test_acc_explinear = np.zeros([len(kernel_list), len(gen_model_list), n_runs])
    train_acc_hinge = np.zeros([len(kernel_list), len(gen_model_list), n_runs])
    test_acc_hinge = np.zeros([len(kernel_list), len(gen_model_list), n_runs])

    # TODO: Change the following random seed to your GROUP ID
    np.random.seed(0)
```

```
for r in range(n_runs):
    for i, kernel in enumerate(kernel_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)

            a, a0 = adjExpLinear(Xtrain, ytrain, lamb, kernel)
            train_acc_explinear[i, j, r] = # TODO: compute accuracy on training set
            test_acc_explinear[i, j, r] = # TODO: compute accuracy on test set

            a, a0 = adjHinge(Xtrain, ytrain, lamb, kernel)
            train_acc_hinge[i, j, r] = # TODO: compute accuracy on training set
            test_acc_hinge[i, j, r] = # TODO: compute accuracy on test set

# TODO: compute the average accuracies over runs
# TODO: combine accuracies (explinear and hinge)
# TODO: return 5-by-6 train accuracy and 5-by-6 test accuracy
```

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 3: Training accuracies with different kernels

Kernel	ExpLinear			Hinge		
	Model 1	Model 2	Model 3	Model 1	Model 2	Model 3
Linear						
Poly($d = 2$)						
Poly($d = 3$)						
Gauss($\sigma = 1$)						
Gauss($\sigma = 0.5$)						

Table 4: Test accuracies with different kernels

Kernel	ExpLinear			Hinge		
	Model 1	Model 2	Model 3	Model 1	Model 2	Model 3
Linear						
Poly($d = 2$)						
Poly($d = 3$)						
Gauss($\sigma = 1$)						
Gauss($\sigma = 0.5$)						

Runtime. For efficient implementation, this function can be finished in around 5 minutes.

(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 (5%) Binary Classification (SVM Dual Form)

In this question, you will implement binary classification with the hinge loss (yet again) using the *dual formula*, and choose the best hyper-parameter and kernel for some real-world problems via cross-validation.

The input vectors are assumed to be **un-augmented** in this question (i.e. we do not add a constant feature of 1 to it). All of the following functions must be able to handle arbitrary $n > 0$ and $d > 0$. The vectors and matrices are represented as NumPy arrays. Your functions shouldn't print additional information to the standard output.

(a) (1%) Implement a Python function

```
a, b = dualHinge(X, y, lamb, kernel_func, stabilizer=1e-5)
```

that takes an $n \times d$ input matrix `X`, an $n \times 1$ target/label vector `y`, a regularization hyper-parameter `lamb` (i.e., $\lambda > 0$), and a callable kernel function `kernel_func`, and returns an $n \times 1$ vector of weights/parameters `a` and a scalar intercept `b` corresponding to the solution of the dual form of support vector machine (i.e., regularized hinge loss):

$$\begin{aligned} \max_{\boldsymbol{\alpha}} \quad & \boldsymbol{\alpha}^\top \mathbf{1}_n - \frac{1}{2\lambda}\boldsymbol{\alpha}^\top \Delta(\mathbf{y}) K \Delta(\mathbf{y}) \boldsymbol{\alpha} \\ \text{s.t.} \quad & \mathbf{0}_n \preceq \boldsymbol{\alpha} \preceq \mathbf{1}_n \\ & \boldsymbol{\alpha}^\top \mathbf{y} = 0 \end{aligned} \tag{1}$$

where K is computed based on `X` and the kernel function `kernel_func`. This problem can be solved by `cvxopt.solvers.qp`.

The output `a` corresponds to $\boldsymbol{\alpha}^*$ (the solution you get from optimizing Eq. (1)) and the offset `b` can be recovered as follows: Find any α_i such that $0 < \alpha_i < 1$, then the offset `b` can be computed using that data point as $b = y_i - \frac{1}{\lambda}\mathbf{k}_i^\top \Delta(\mathbf{y})\boldsymbol{\alpha}^*$.

Debug hint: One way to verify your solution is to compare it to the solutions from Q1(b) and Q2(b). Recall that *when using linear kernel*, the primal solution satisfies $\mathbf{w}^* = \frac{1}{\lambda} X^\top \Delta(\mathbf{y})\boldsymbol{\alpha}^*$. That is, the $\mathbf{w}^*$ you get here should be similar to those of Q1(b) and Q2(b).

Numerical issue: Some α_i may appear $0 < \alpha_i < 1$ but is actually 0 or 1 (e.g., $\alpha_i = 0.00000001$ or $\alpha_i = 0.99999999$). This could be due to lost of precision during training. To ensure numerical stability, it is recommended to pick the α_i that is closest to 0.5 to compute the offset `b`. Also, don't forget the stabilizer `P = P + stabilizer * np.eye(n)`.

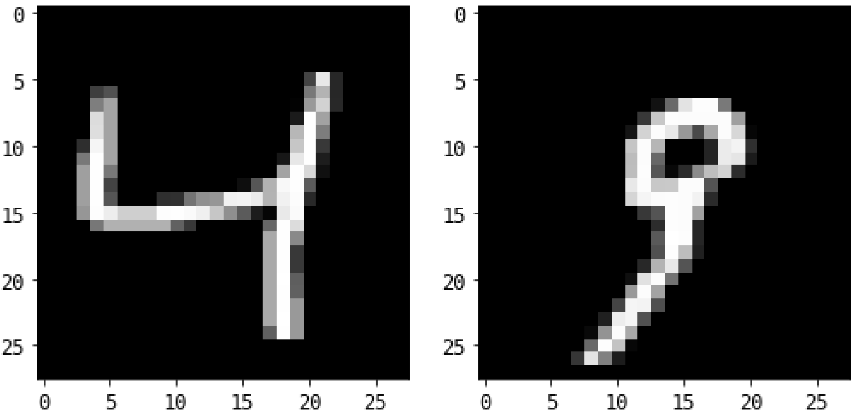
(b) (1%) Implement a Python function

```
yhat = dualClassify(Xtest, a, b, X, y, lamb, kernel_func)
```

that takes an $m \times d$ test matrix `Xtest` consisting of (any) points to be predicted on, a $n \times 1$ vector of weights/parameters `a`, a scalar intercept `b`, an $n \times d$ **training** input matrix `X`, an $n \times 1$ training output vector, a scalar regularization hyper-parameter `lamb` a kernel function `kernel_func`, and returns an $m \times 1$ prediction vector `yhat`. Recall that the prediction of a data point $\mathbf{x}$ is given by $\hat{y} = \text{sign}\left(\frac{1}{\lambda}\text{kernel\_func}(\mathbf{x}^\top, X)\Delta(\mathbf{y})\boldsymbol{\alpha}^* + b^*\right)$. To predict on the whole test dataset, $\hat{\mathbf{y}}_{\text{test}} = \text{sign}\left(\frac{1}{\lambda}\text{kernel\_func}(X_{\text{test}}, X)\Delta(\mathbf{y})\boldsymbol{\alpha}^* + b^*\right)$.

Your function must be able to handle arbitrary number of test points $m > 0$.

(c) (2%) The A2files.zip includes an image dataset, A2train.csv, of handwritten digits taken from the MNIST dataset. Each image is either digit 4 or digit 9 (once loaded, you can call the `plotDigit` function to see some samples of the images as in Fig. 4). The first column of the csv file is the class label. Treat digit 4 as the -1 class and digit 9 as the $+1$ class, your task is to use your `dualHinge` function to learn a good binary classifier.



(a) Digit 4

(b) Digit 9

Figure 4: Sample images from the training dataset

In this part, you need to perform cross-validation and select the best hyperparameters and kernels for this dataset. Implement a Python function

```
cv_acc, best_lamb, best_kernel = cvMnist(dataset_folder, lamb_list, kernel_list, k=5)
```

that takes the (absolute) dataset path `dataset_folder`, a candidate (Python) list of regularization parameter `lamb_list`, a candidate (Python) list of kernel functions `kernel_list`, an integer `k` indicating k-fold cross-validation, and returns a "`len(lamb_list)`-by-`len(kernel_list)`" matrix `cv_acc` of average (across folds) accuracy on the validation data. It runs as follows

```
def cvMnist(dataset_folder, lamb_list, kernel_list, k=5):

    train_data = pd.read_csv(f"{dataset_folder}/A2train.csv", header=None).to_numpy()
    X = train_data[:, 1:] / 255.
    y = train_data[:, 0][:, None]
    y[y == 4] = -1
    y[y == 9] = 1

    cv_acc = np.zeros([k, len(lamb_list), len(kernel_list)])

    # TODO: Change the following random seed to your GROUP ID
    np.random.seed(0)

    # TODO: perform any necessary setup

    for i, lamb in enumerate(lamb_list):
        for j, kernel_func in enumerate(kernel_list):
            for l in range(k):
                Xtrain = # TODO: obtain the training input
                ytrain = # TODO: obtain the corresponding training label
                Xval = # TODO: obtain the validation input
                yval = # TODO: obtain the corresponding validation label
                a, b = dualHinge(Xtrain, ytrain, lamb, kernel_func)
                yhat = dualClassify(Xval, a, b, Xtrain, ytrain, lamb, kernel_func)
                cv_acc[l, i, j] = # TODO: calculate validation accuracy
    # TODO: compute the average accuracies over k folds
    # TODO: identify the best lamb and kernel function
    # TODO: return a "len(lamb_list)-by-len(kernel_list)" accuracy variable,
    #       the best lamb and the best kernel
```

In the PDF file, use a table to clearly show your average validation accuracy for some `lamb_list` and `kernel_list` of your choice. Explain how you choose the best performing hyper-parameters and kernel function.

Note: Each image pixel takes a value from 0 to 255. The pixel values are divided by 255 so that values in the input matrix `X` are in [0, 1] for `dualHinge` and `dualClassify`.

Runtime. For efficient implementation, this function can be finished in around 30 seconds, when using two λs and three kernels (linearKernel, polyKernel and gaussKernel).

(d) (1%) We will evaluate your choices from (c) on a *new* test dataset (that you don't have access to). You will get full mark here if your chosen hyper-parameters, kernel function, `dualHinge` and `dualClassify` can achieve acceptable performance on the test dataset.

Note: We will also normalize the test input matrix `Xtest` to [0, 1] when calling your `dualClassify`.