King Fahd University of Petroleum & Minerals
College of Computing and Mathematics
Information and Computer Science Department

ICS 381: Principles of AI – Second Semester 2021-2022 (212)
PA 3 – Programming Instructions

General Helpful Tips:

- You can submit your code on Gradescope any time before the submission deadline.
- Do not copy others' work. You can discuss general approaches with students, but do not share specific coding solutions.
- **NOTE1:** You are free to use numpy. So, you can make use of np.argmax, np.argmin, np.argsort, etc. You can also use pandas package.
- **NOTE2:** You are **not** allowed to use scikit-learn package for this assignment.
- **NOTE3:** Many of the function arguments for this assignment are 2D or 1D numpy arrays.
- **NOTE4:** Be sure to implement the functions as specified in this document. In addition, you can implement any extra helper functions as you see fit with whatever names you like.
- Submit the required files only: [**machine_learning.py, image_processing.py**]

- References: the citrus dataset is adapted from link.

machine_learning.py: Implement data preprocessing

Implement the following data preprocessing functions:

Function Name: `preprocess_classification_dataset()`
Arguments: `No arguments`
Returns: `X_train, y_train, X_val, y_val, X_test, y_test`; these are all numpy arrays, the feature matrices are of shape=(num_instances, features) and the y labels are of shape=(num_instances, 1).

Implementation: For this function, you want to setup the train, val, and test datasets for the citrus dataset. Be sure to import the pandas package and download the required dataset files from blackboard. There are three dataset files: [train.csv, val.csv, test.csv]. Open one of the csv files to become familiar. The target binary label is the 'output' column. To setup the training dataset, do the following:

1. Use pandas pd.read_csv function to read citrus_train.csv into a train_df dataframe object. You can assume that the data files exist in the same directory as machine_learning.py.
2. Grab the features columns from train_df using panda's column slicing. Grab the output label column. As follows:

```
train_feat_df = train_df.iloc[:,:-1] # grab all columns except the last one
train_output = train_df[['output']]
```

3. Convert the feature and output dataframes into numpy arrays. As follows:

```
X_train = train_feat_df.values
y_train = train_output.values
```

Do the same to setup val and test datasets. Return `X_train, y_train, X_val, y_val, X_test, y_test`

Note: your function should work for any dataset for any number of features. The autograder will test your code against randomized datasets. You can assume the datasets will have the 'output' column as the last column, and previous columns are all features and are all continuous numbers.

machine_learning.py: Implement k-nearest neighbors

Implement the following function for k-nearest neighbors:

Function Name: `knn_classification`
Arguments: `X_train, y_train, x_new, k=5`
Returns: `y_new_pred`; the prediction of x_new (a scalar value).

Implementation: X_train and y_train are the training set's feature matrix and label vector, respectively. x_new is the feature vector of a new instance (1D numpy array). Implement k-nearest neighbors to predict the label of x_new using the training set. Use **Euclidean distance**; you can assume all features are continuous real values. For the prediction, it is the majority vote of the k-nearest neighbors (hint: you can use numpy's np.unique to get counts). Return the predicted label of x_new (a scalar value).

machine_learning.py: Implement logistic regression gradient descent

Implement the following functions for logistic regression:

Function Name: `logistic_regression_training`
Arguments: `X_train, y_train, alpha=0.01, max_iters=5000, random_seed=1`
Returns: `weights`; numpy array of the trained weights vector for logistic regression (first element is the bias weight).

Implementation: Implement logistic regression using gradient descent (see the update formula in the slides). Note that y_train contains {0, 1} binary labels.

1. First, add a column of ones in X_train for the bias weight. I encourage you to make use of numpy's np.ones and np.hstack functions to accomplish this. Note: be careful not to modify X_train directly since it is passed by reference, and it maybe used for other things by the caller.
2. Randomly initialize the weight vector. Do not forget about the bias which is the first element of this weight vector. You can do this using numpy's np.random.normal function as follows:

   ```
   np.random.seed(random_seed) # for reproducibility
   weights = np.random.normal(loc=0.0, scale=1.0, size=(num_of_features, 1))
   ```

   Note that num_of_features counts the number of features + the bias.

3. Now perform max_iters iterations of gradient descent using logistic regression update formula. Hint: make use of numpy's matrix multiplication operator @ (see a quick tutorial here); this will allow you write the update in very few lines of code.
4. After max_iters updates, return the final weight vector. Note that the shape should be (num_of_features, 1) including bias.

Function Name: `logistic_regression_prediction`
Arguments: `X, weights, threshold=0.5`
Returns: `y_preds`; numpy array of the logistic regression {0, 1} label predictions of X of shape=(num_instances, 1)..

Implementation: Given a numpy feature matrix X of instances (can be more than one instance), you want to perform logistic regression predictions using weights. Be sure to add a column of ones in X. Recall that logistic regression outputs probability scores, so you will threshold them using threshold argument to get final {0, 1} label predictions.

machine_learning.py: Apply model selection & evaluation on citrus dataset

Now we are going to use the implemented machine learning algorithms to apply model selection and evaluation on the binary classification citrus dataset. Implement the following function.

Function Name: `model_selection_and_evaluation`
Arguments: `alpha=0.01, max_iters=5000, random_seed=1, threshold=0.5`
Returns: `best_method, val_accuracy_list, test_accuracy`

Implementation: Perform the following steps

1. Setup the train, val, and test sets using `preprocess_classification_dataset()`
2. For model selection, evaluate the following four models on the validation dataset: ['1nn', '3nn', '5nn', 'logistic regression']
    a. For the knn models, use your implemented `knn` function. Hint: loop on the validation instances to compute their predictions using `knn`.
    b. For logistic regression, first train on the training set to get trained weights, then use the trained weights to compute predictions on the validation set instances. Use your implemented logistic regression functions with alpha, max_iters, and random_seed as arguments.
    c. For the four models, compute the validation set accuracy of the predictions. Identify the model that has the highest validation set accuracy. For the logistic regression model, use the threshold argument when calling logistic_regression_prediction.
3. With the best model, evaluate the test set accuracy using the merged train Val datasets. You can use np.vstack for vertically merging train and validation datasets.

```
X_train_val_merge = np.vstack([X_train, X_val])
y_train_val_merge = np.vstack([y_train, y_val])
```

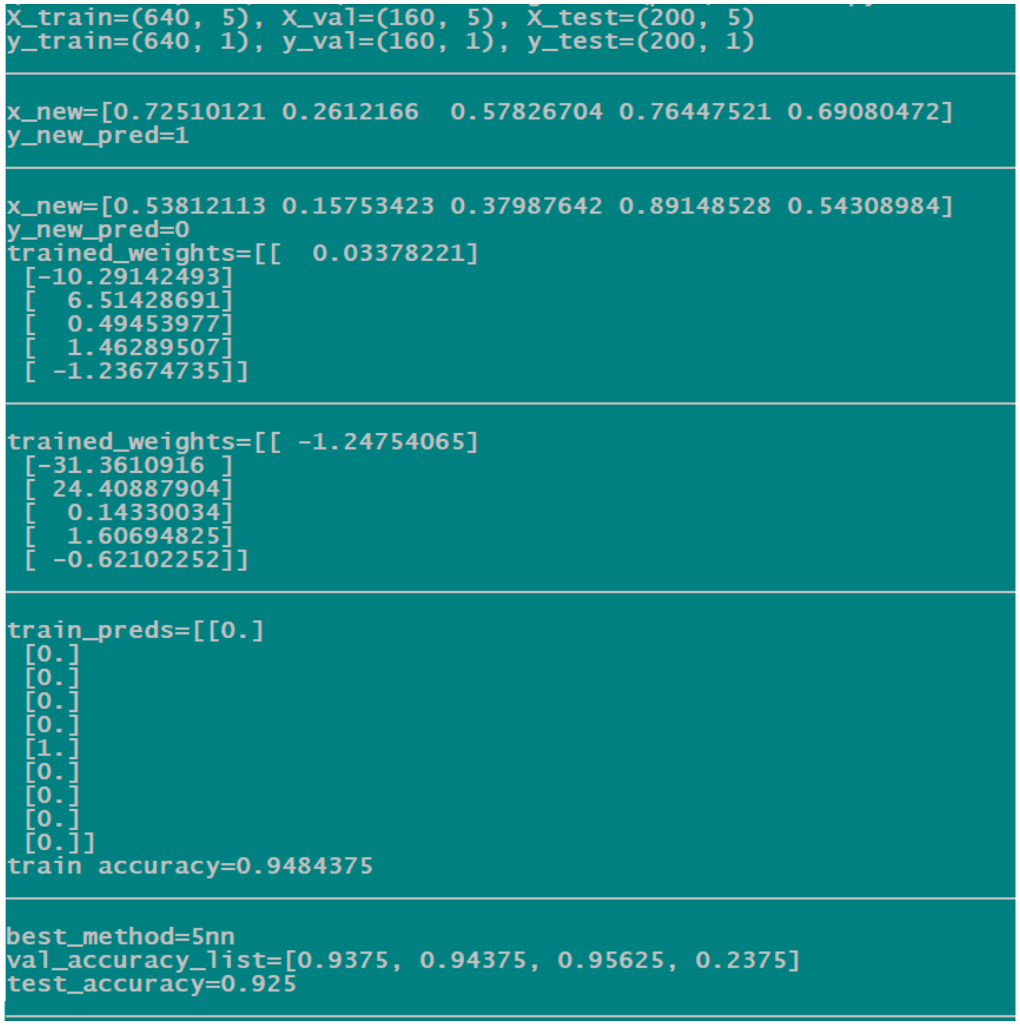
Return the name of the best method as a string ('1nn', '3nn', '5nn', or 'logistic regression'), the **list** of validation set accuracies in the same order of ['1nn', '3nn', '5nn', 'logistic regression'], and the test set accuracy. Accuracy should be float value between 0 and 1.

Hint: I recommend making a helper function for computing the knn predictions for an entire dataset. Another hint is that accuracy can be computed in a single statement `(y_true.flatten() == y_pred.flatten()).sum() /y_true.shape[0]` assuming y_true and y_pred are numpy arrays for the true labels and the corresponding predictions, respectively.

Test your code on test_ml.py

You can test your machine learning code by running test_ml.py. You can inspect the code and add your own problem cases.

The following are my implementation results:



```
X_train=(640, 5), X_val=(160, 5), X_test=(200, 5)
y_train=(640, 1), y_val=(160, 1), y_test=(200, 1)

x_new=[0.72510121 0.2612166  0.57826704 0.76447521 0.69080472]
y_new_pred=1

x_new=[0.53812113 0.15753423 0.37987642 0.89148528 0.54308984]
y_new_pred=0
trained_weights=[[  0.03378221]
 [-10.29142493]
 [  6.51428691]
 [  0.49453977]
 [  1.46289507]
 [ -1.23674735]]

trained_weights=[[ -1.24754065]
 [-31.3610916 ]
 [ 24.40887904]
 [  0.14330034]
 [  1.60694825]
 [ -0.62102252]]

train_preds=[[0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [1.]
 [0.]
 [0.]
 [0.]
 [0.]]
train accuracy=0.9484375

best_method=5nn
val_accuracy_list=[0.9375, 0.94375, 0.95625, 0.2375]
test_accuracy=0.925
```

[Optional Bonus] image_processing.py: Implement cross-correlation and convolution operations

Your task is to implement the cross-correlation and convolution operations from scratch. You are only allowed to import numpy. You should **not** use numpy's correlation and convolve functions.

Name: `cross_correlation`
Arguments: `image, filter, mode='valid'`
Returns: `result_image`

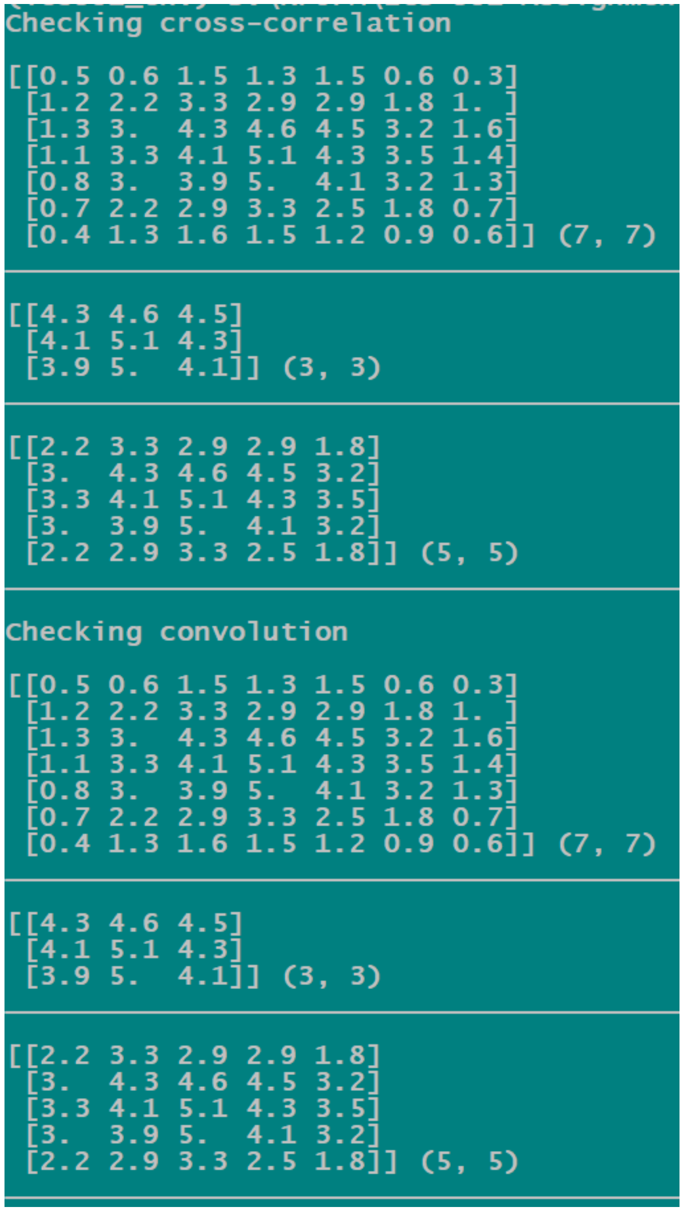
Implementation: you should implement this function for the three modes `'valid', 'same', 'full'`. To help with the padding in case of same and full modes, you can numpy's np.pad function. To compute the summations, you can implement this operation as a double loop of applying the filter to the image. For this bonus, you might need to get familiar with numpy's slicing capabilities. Particularly when grabbing the needed parts of the image to apply the filter on.

Assumptions: Image and filter are 2D numpy arrays. You can assume that filter will always be smaller than image. You can assume that the filter will be odd sized shape (e.g. shape=(3, 5)); this makes computing padding easier.

Name: `convolution`
Arguments: `image, filter, mode='valid'`
Returns: `result_image`

Implementation: If you implemented cross-correlation correctly, then you can implement convolution as a call to your cross_correlation function with a flipped filter. You can use numpy's flip function to achieve this.

Test your code by running test_cv.py and check that your implementation matches scipy’s function results.



```
Checking cross-correlation

[[0.5 0.6 1.5 1.3 1.5 0.6 0.3]
 [1.2 2.2 3.3 2.9 2.9 1.8 1. ]
 [1.3 3.  4.3 4.6 4.5 3.2 1.6]
 [1.1 3.3 4.1 5.1 4.3 3.5 1.4]
 [0.8 3.  3.9 5.  4.1 3.2 1.3]
 [0.7 2.2 2.9 3.3 2.5 1.8 0.7]
 [0.4 1.3 1.6 1.5 1.2 0.9 0.6]] (7, 7)

[[4.3 4.6 4.5]
 [4.1 5.1 4.3]
 [3.9 5.  4.1]] (3, 3)

[[2.2 3.3 2.9 2.9 1.8]
 [3.  4.3 4.6 4.5 3.2]
 [3.3 4.1 5.1 4.3 3.5]
 [3.  3.9 5.  4.1 3.2]
 [2.2 2.9 3.3 2.5 1.8]] (5, 5)

Checking convolution

[[0.5 0.6 1.5 1.3 1.5 0.6 0.3]
 [1.2 2.2 3.3 2.9 2.9 1.8 1. ]
 [1.3 3.  4.3 4.6 4.5 3.2 1.6]
 [1.1 3.3 4.1 5.1 4.3 3.5 1.4]
 [0.8 3.  3.9 5.  4.1 3.2 1.3]
 [0.7 2.2 2.9 3.3 2.5 1.8 0.7]
 [0.4 1.3 1.6 1.5 1.2 0.9 0.6]] (7, 7)

[[4.3 4.6 4.5]
 [4.1 5.1 4.3]
 [3.9 5.  4.1]] (3, 3)

[[2.2 3.3 2.9 2.9 1.8]
 [3.  4.3 4.6 4.5 3.2]
 [3.3 4.1 5.1 4.3 3.5]
 [3.  3.9 5.  4.1 3.2]
 [2.2 2.9 3.3 2.5 1.8]] (5, 5)
```