CSCC11 – Introduction to Machine Learning and Data Mining

Assignment 1

Logistics

- This assignment is worth 12% of the course grade and is due on Nov 12, 2023.
- It can be done individually or in groups of two. Either way, you are required to self-signup to one of the A1 groups on Quercus and provide a single archive file (.rar or .zip) containing all the files/documents related to your solution.
- The answers/code you submit (excluding the starter code) must be your own.
- A starter code folder has been provided to help you complete this assignment. Please do not change the headers of the given Python functions.
- We prefer that you ask any related questions during the tutorial sessions or on Piazza. Otherwise, for your first point of contact, please reach out to your TAs:

Arash Rasti Meymandi, through email: arash.rasti@mail.utoronto.ca
Anindro Bhattacharya, through email: anindro.bhattacharya@mail.utoronto.ca
Noor Nasri, through email: noor.nasri@mail.utoronto.ca

Part 1: Linear Regression [3 points]

You are required to use linear regression to estimate the probability of being admitted to a graduate program using this dataset.

a) Express the corresponding optimization problem in matrix-vector form and make sure that all the involved entities are well-defined, along with their dimensions. [1 point]

b) Follow the steps below to build and test the linear regression model: [2 points]

i. Read the dataset *Admission_Predict.csv* file into a data frame.

ii. Split the data into training and testing subsets (you can make use of the `train_test_split` function in the `sklearn` library). The suggested way to split the dataset is a proportion of 70:30 where 70% of the data is used for training and 30% is used for testing. Feel free to try various other proportions to see the effect of the having different sizes during training and testing.

iii. Compute the closed form solution for linear regression using the starter code. Remember that the goal is to estimate the *Chance of Admit* value.

iv. A function `get_pred_Y` has been defined to provide predictions on the test data. Complete the function.

v. Report the *Mean Squared Error (MSE)* resulting from applying the model to the test set.

Part 2: RBF Regression* [5 points]

Suppose you're given an image with missing or corrupted pixels, like the image below corrupted by red text (Fig 1 (b)). Inpainting is a process of predicting the corrupted pixels, for which we'll use RBF regression.

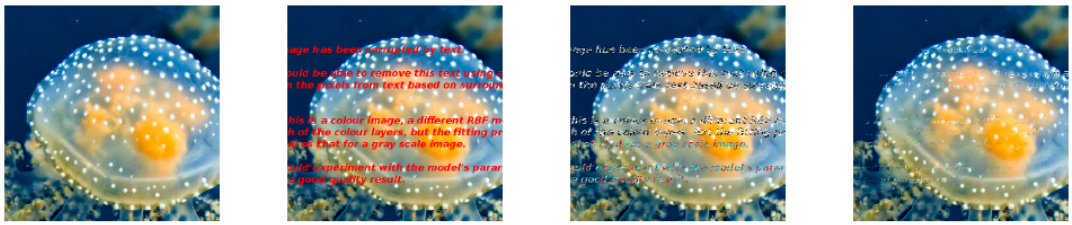

Figure 1: From left to right: (a) The original image. (b) An image corrupted with red text. (c) A badly restored image. (d) A well restored image.

Background: An image is a 2D array of pixels. For grayscale images (Fig 2 (left)) each pixel is a scalar, representing brightness. (Color images have three values per pixel, for red, green and blue components, but let's focus on greyscale for now.) We'll model an image as a function I(x) that specifies brightness as a function of position $x \in \mathbb{R}^2$. Pixel values are between 0 (black) and 1 (white).

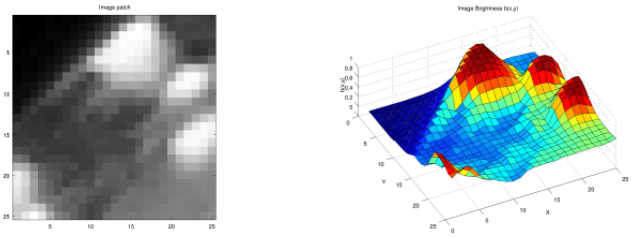

Figure 2: *(left)* Greyscale image. *(right)* Depiction of image as a height map where brighter pixels are higher.

We don't have an equation for I(x) that we can evaluate at the corrupted pixels, but we can use basis function regression to find one. Because images are smooth and correlated in local regions, we want basis functions that represent the brightness in one region more or less independently of other regions. So let's use radial basis functions (RBFs) assuming the model is:

$$m(x) = w_0 + \sum_k w_k b_k(x)$$

where b_k is the k[th] basis function:

$$b_k(x) = \exp\left(-\frac{\|x - c_k\|^2}{2\sigma^2}\right)$$

* This question was originally prepared by Profs. Francisco Estrada and Bryan Chan

The RBF is a smooth bump centered at location $c_k \in \mathbb{R}^2$ with width σ^2. We'll assume the basis functions are centred on a square grid, all with the same width. You will specify the grid spacing and the RBF width, and then find the weights w_k (including the bias) using regularized least squares regression on image patches. For example, for a 3×3 grid of RBFs, with all weights equal to 1, the model might look like Figure 3 left. If instead we use the LS weights (Fig. 3 middle) we get a better approximation to our image patch (Fig. 2 right). We get an even better approximation if we use 64 RBFs on an 8×8 grid (Fig. 3 right). Once we have the model, we can use it to evaluate (or predict) the image values at any pixel (e.g. corrupted pixels), or in between two pixels.

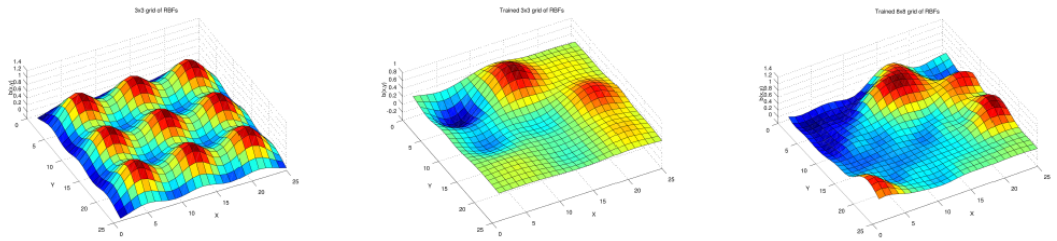

Figure 3: *(left)* An RBF model with all weight $w_k = 1$. *(middle)* A model with 3×3 RBFs with LS regression weights. *(right)* A model with 8×8 RBFs with LS regression weights. (cf. Fig. 2 (right)).

Your Task: The starter code (see *StarterCode/Part2*) contains methods for handling images. They can crop image patches, reshape the pixels values into vectors of the correct dimensions, and feed these to your regression code. Your task will be to write code to compute the least-squares weights for the RBF model. You will then explore how your model performs for inpainting, varying the RBF spacing, the RBF width, and the regularization constant. The amount of code required is minimal, but it requires an understanding of LS regression, and careful implementation of the right equations.

1. Read through the methods in the starter code. There are two key files, namely, `rbf_regression.py` and `rbf_image_inpainting.py`. Pay attention to the examples provided that show how the functions are called, and what the expected output should be.
2. **Implement 2D RBF regression code**. The file `rbf_regression.py` contains the class `RBFRegression`, along with various methods:
    a. `__init__(self, centers, widths)`: This is the constructor of the class. In it, `centers` is a $K \times 2$ matrix that specifies the centers of K RBF basis functions (*bumps*), and `widths` specifies a vector of K corresponding widths.
    b. `_rbf_2d(self, X, rbf_i)`: This computes the output of the i^{th} RBF given a set of 2D inputs. X is an $M \times 2$ matrix of inputs, where each row (in total M rows) stores a 2D input, and `rbf_i` specifies the i^{th} RBF.
    c. `predict(self, X)`: This method predicts the output for the given inputs, using the model parameters. X is an $M \times 2$ matrix of inputs, where each row (in total M rows) corresponds to a 2D input. The method outputs a vector containing the corresponding predicted outputs.
    d. `fit_with_l2_regularization(self, train_X, train_Y, l2_coef)`: This method finds the regularized LS solution. Here, `train_X` is an $N \times 2$ matrix of training inputs, where each row (in total N rows) corresponds to a 2D training input, `train_Y` is a vector of training outputs,

and `l2_coef` is the regularization constant, λ, that controls the amount of regularization.

You should complete the body of the two methods, `predict` and `fit_with_l2_regularization`. Similar to the `PolynomialRegression` class, the `RBFRegression` class also has the variable `self.parameters`. It is the column vector containing w_k. You will be using `self.parameters` for prediction, and updating it after computing the LS model fit. Again, once you have completed `RBFRegression`, you should do some testing to verify your solution. The starter code provides some basic checks on whether your implementations are correct, but they do not cover all cases. Remember that your regression code should include the bias term with the RBFs.

You can now run `RBF_image_inpainting.py` to remove texts on images. If working correctly and proper hyperparameters, you should obtain a reasonably clean jelly fish (see Fig 1 (d)) with the image `Amazing_jellyfish_corrupted_by_text.tif`. You may also change the settings of `RBF_image_inpainting.py` by changing the image, the colour to be filled in, the patch size, similarity tolerance, centers, widths, and the regularization constant λ. Examine what happens when you change the distance between RBF centers. What is the effect of RBF width on the reconstruction? What can you say about tuning the model as you increase the amount of hyperparameters? There is no deliverable for this analysis, but make sure you understand how the RBF regression works and choose appropriate parameters to obtain a clean image.

Part 3: Clustering [4 points]

Compression is a fundamental technique in image processing. It involves reducing the amount of data required to represent an image, thus saving storage space and reducing transmission time. This can be achieved through clustering by grouping similar pixels and representing them with cluster centroids. For example, in the case of the image below, compression using K-means with K=20 reduces the size nearly by half (assuming a *png* format).

a) You are required to implement an image compression approach using K-means. More specifically, you need to write Python code to cluster the pixels of an image into K clusters and then replace each pixel with the centroid of the cluster it belongs to. The starter code includes the following functions: [2 points]

1. `kmeans_numpy(X, n_clusters, max_iters=100):`
    - Initializes cluster centroids randomly.
    - Iteratively assigns pixels to the nearest centroid and recalculates centroids.
    - Returns the final centroids and labels.
2. `create_compressed_image(labels, centroids, height, width, channels):`
    - Creates a compressed image by assigning each pixel to the cluster centroid.
3. `kmeans_image_compression(image_path, num_clusters):`
    - Loads the original image.
    - Applies K-Means clustering using the `kmeans_numpy` function.
    - Creates a compressed image using `create_compressed_image`.
    - Saves the original and compressed images.
    - Calculates and prints the compression ratio.
    - Displays the original and compressed images using matplotlib.

Implementation Steps

1. **Loading the image:**
    - Use the `cv2.imread()` function to load an image from a given file path.
    - Convert the image into the RGB color space using `cv2.cvtColor()` if it's not in RGB.
2. **K-Means clustering:**
    - Implement the K-Means clustering algorithm using the `kmeans_numpy` function.
    - Determine the number of clusters (`num_clusters`) based on your choice.
3. **Creating the compressed image:**
    - Implement the `create_compressed_image` function to create the compressed image based on cluster centroids and labels.
4. **Saving the image:**
    - Save the compressed image in *png* format using `cv2.imwrite()`

```
compressed_image_path = "compressed_image.png"
cv2.imwrite(compressed_image_path, cv2.cvtColor(compressed_image, cv2.COLOR_RGB2BGR))
```

    - Calculate the sizes of the original and compressed images using `os.path.getsize()` and compute the compression ratio which is original_size divided by compressed_size.
5. **Visualization:**
    - Display the original and compressed images side by side using matplotlib.

Test your implementation with different values of `num_clusters` and briefly explain the impact of increasing the number of clusters.

b) How would you use a *Gaussian Mixture Model* to perform the compression? Note that you are not required to provide any code for this part – just make sure to explain all the steps in

detail, along with the equations to be used for the computations. All the involved entities and their dimensions should be well-defined. [2 points]