CSCC11 – Introduction to Machine Learning and Data Mining

Fall 2022

Assignment 1

Logistics

- This assignment is due on Oct 26, 2022.
- It can be done individually or in groups of two.
- The answers/code you submit (excluding the starter code) must be your own.
- Please do not change the headers of the Python functions included with the assignment.
- Some questions require written answers while others require writing Python code. Make sure to upload the various files/documents of your solution as a single archive file (e.g. rar, zip) to "Assignment 1" on Quercus. The file should be named using the student IDs of the group members.
- We prefer that you ask any related questions on Piazza. Otherwise, for your first point of contact, please reach out to your TAs:

  Dhiraj Tawani, through email: dhiraj.tawani@mail.utoronto.ca
  Srinath Dama, through email: srinath.dama@mail.utoronto.ca

Part 1: Exploring a combination of clustering and linear regression

In some cases, using clustering as a pre-processing step helps improve the accuracy of linear regression. Instead of building a single linear regression model using the entire dataset, it might be beneficial to cluster the data first and then build a linear model for each cluster separately. As such, estimating the response for a new data point *d* would be done by identifying the cluster *d* belongs to, followed by applying the linear model associated with that cluster. You are required to explore the effectiveness of this approach using the graduate admissions dataset that can be found here.

1.1. Applying and evaluating a single linear regression model [20 pts]

This part involves applying linear regression without clustering. To do that, you can use the starter code provided in *StarterCode/Part1*.

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

1.1.2. 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.

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

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

1.1.5. We will use two metrics to evaluate the effectiveness of the implementation: *Mean Absolute Error (MAE)* & *Mean Squared Error (MSE)*. Define these metrics and discuss why one would be preferred over the other. Complete the functions `get_mae` and `get_mse` in the starter code file.

1.1.6. Find the optimal parameters using your training dataset and use these parameters to estimate the *Chance of Admit* values for the testing dataset. In addition, you are required to report the corresponding *MAE* and *MSE* values.

1.2. Applying and evaluating clustering-based linear regression [30 pts]

In this part, you are required to perform K-Means clustering as a pre-processing step in the training phase (i.e. clustering the training dataset) and then build a linear regression model for each cluster separately.

1.2.1. To determine the number of clusters, you can use the *Silhouette Coefficient* which ranges between -1 and 1 where higher values indicate better clustering. That is, you can run K-Means with varying values of K and use the value that results in the highest *Silhouette Coefficient*. You can learn more about the Silhouette coefficient here and use the method provided in the `sklearn` library.
Complete the code to determine the *Silhouette Coefficient* associated with each value of K ranging between 2 and 10. Which value would be the most appropriate?

1.2.2. Using the number of clusters obtained in the previous step, apply the K-Means implementation provided in the `sklearn` library to the training dataset.

1.2.3. Build a separate linear regression model for each cluster.

1.2.4. Final Steps:

a. Report the *Mean Absolute Error* and *Mean Squared Error* for the testing dataset and compare them to the errors from the previous section. Note that, for each data point in the testing dataset, you are required to determine the corresponding cluster using the result of 1.2.2 and then estimate the *Chance of Admit* value using the result of 1.2.3.
b. Provide a brief discussion regarding the factors that might have contributed to this result.

Part 2: Image Inpainting with RBF Regression*

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.

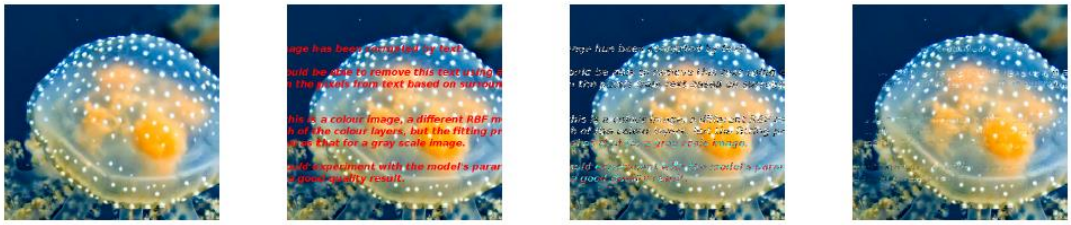

Figure 1: From left to right: (a) The original image. (b) An image corrupted with red text. (c) A badly restored image. (d) A well restored image.

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).

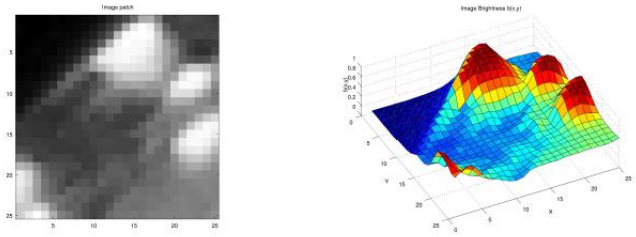

Figure 2: *(left)* Greyscale image. *(right)* Depiction of image as a height map where brighter pixels are higher.

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:

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

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

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

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.

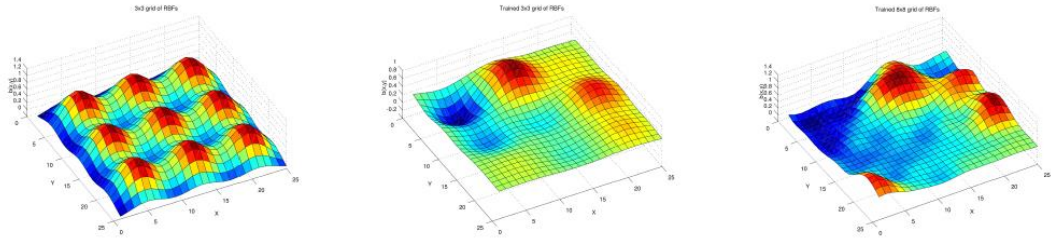

Figure 3: *(left)* An RBF model with all weight $w_k = 1$. *(middle)* A model with 3×3 RBFs with LS regression weights. *(right)* A model with 8×8 RBFs with LS regression weights. (cf. Fig. 2 (right)).

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. **[50 pts]**

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.