CSCC11 Introduction to Machine Learning, Winter 2021

Assignment 1, Due Thursday, February 4, 10am

This assignment makes use of material from the first 3 weeks, but much of it can be completed after week 2 (Chapters 1-7). It includes both written work and programming (in Python). To begin the programming component, download `a1.tar` from the calendar page on the course website. When you untar the file a directory A1 is created. Please do not change the directory structure, nor the headers of the Python functions contained therein. Please read the assignment handout carefully. Instructions for electronic submission are included below.

We prefer you ask questions on Piazza. Otherwise, for your first point of contact, please reach out to your TA, Aravind Shaj, through email: aravind.shaj@mail.utoronto.ca.

1) Written Component

1. Consider Least-Squares (LS) basis function regression for a single variable input/output problem, i.e.,

$$y \;=\; f(x) \;=\; w_0 + \sum_{k=1}^{K} w_k \, b_k(x) \; .$$

Let the training data be denoted by $\{(x_i, y_i)\}_{i=1}^{N}$.

   (a) Please formulate the LS objective in terms of a vector of known inputs, a corresponding vector of known outputs, and the appropriate matrices. Remember to include a bias term in the model (incorporated into the basis function matrix and the weight vector). **Note**: Please put the bias related terms as the first element/row/column of the vector/matrix.

   (b) Then show each step of taking the gradient of the objective function. **Note**: You may use the matrix identities handout on the course website.

   (c) Then solve for the optimal weight vector $\mathbf{w}$ (which includes the bias w_0).

2. In the following problem we consider cases in which the model above is not well constrained by the data alone.

   (a) Assume that you have at least one data point, $N \geq 1$. In general, under what conditions will the solution to the normal equations in Q1(c) *not* be unique? Set up and provide a simple regression problem where the optimal weight vector ($\mathbf{w}$) is not unique.

   (b) Consider L2 regularized regression (a.k.a. ridge regression), which adds a term to the LS objective that penalizes the squared L2 norm of $\mathbf{w}$, scaled by a positive regularization parameter λ (see Eqn (10) in Chapter 3 of the online notes). Use the gradient of this regularized objective to derive the normal equations for the solution in this case, and explain the way in which this regularization helps overcome the problem above when the data alone do not fully constrain the solution.

   (c) Show that the solution for regularized regression in part (b) can alternatively be obtained via (ordinary) least squares regression with augmented basis function matrix $\hat{\mathbf{B}}$ and known outputs $\hat{\mathbf{y}}$ as defined below:

$$\hat{\mathbf{B}} = \begin{pmatrix} \mathbf{B} \\ \sqrt{\lambda}\, \mathbf{I}_{K+1} \end{pmatrix} \in \mathbb{R}^{(N+K+1)\times(K+1)} \;,\; \hat{\mathbf{y}} = \begin{pmatrix} \mathbf{y} \\ \mathbf{0}_{K+1} \end{pmatrix} \in \mathbb{R}^{(N+K+1)}$$

where $\mathbf{B}$ is the basis function matrix and $\mathbf{y}$ is the vector of known outputs in Q1. $\mathbf{I}_{K+1}$ represents the identity matrix in $\mathbb{R}^{(K+1)\times(K+1)}$ and $\mathbf{0}_{K+1}$ is a zero vector in $\mathbb{R}^{(K+1)}$. Note that the quantities inside brackets are block matrices. Explain in short how the above formulation constrains $\mathbf{w}$.

3. Now consider a probabilistic formulation of basis function regression. This is useful as a way to incorporate measurement noise. For example, images may contain white noise due to lighting variations, hardware issues, and other reasons. Here, we'll assume the target output y is equal to $f(x)$ plus Gaussian noise. Specifically, we assume y given x follows a Gaussian distribution with mean $f(x)$, and variance σ^2. We write this as $y \sim \mathcal{N}(f(x), \sigma^2)$.

   As above, we assume a single variable input/output problem, with training data $\{(x_i, y_i)\}_{i=1}^N$, and that $f(x)$ is a weighted sum of basis functions evaluated at x, with weights $\mathbf{w} = [w_0, \ldots, w_K]^T$.

   (a) Formulate the Maximum Likelihood (ML) objective (without solving for the weights).

   (b) What can you say about the negative log likelihood as compared to the LS objective above in Q1?

   (c) Now, suppose that the model parameters (weights) follow a Gaussian distribution with zero mean with some fixed isotropic covariance $\alpha^{-1}\mathbf{I}$, i.e., $\mathbf{w} \sim \mathcal{N}(\mathbf{0}, \alpha^{-1}\mathbf{I})$. Formulate and take the negative log of the Maximum a Posteriori (MAP) objective. **Note**: You may ignore the evidence term as it does not depend on the parameters of interest.

   (d) What can you say about minimizing the negative log posterior compared to the LS objective above?

   (e) What happens if we assume that the model parameters follow a Uniform distribution? What can you say about ML and MAP estimates in that case?

Coding Component

Software requirements: Setup and use the virtual environment following the instructions posted on Quercus.

2) Least-Squares Polynomial Regression

Preliminary. Familiarize yourself with linear regression. Get the demo regression code from the course website. Run the demo, cell by cell. It shows a couple of ways to solve linear least-squares problems, first for scalar input and output, and then for two-dimensional inputs. It also shows a solution in matrix-vector form.

Now, let's access and visualize the datasets. Download and untar the starter file, `a1.tar`, and look in `A1/Q2`. Load the training data files in the starter directory into Python. The data files are stored as a dictionary in **Pickle** format. To read them from file,

1. Run one of Python Shell, Jupyter Notebook, etc.
2. Import the Pickle module (Note the underscore): `import _pickle as pickle`
3. Open the Pickle file and read the content (replacing **data.pkl** below with the actual file name)
   ```
   with open("data.pkl", "rb") as f:
       data = pickle.load(f)
   ```
4. You may access the training inputs and outputs with `data["X"]` and `data["Y"]`, respectively. They are stored as **numpy** arrays with shape $(N, 1)$. These are the N-dimensional column vectors comprising the N training inputs and outputs for the regression problem. We'll denote them here as

$$\mathbf{x} = [x_1, x_2, ..., x_N]^T \quad , \quad \mathbf{y} = [y_1, y_2, ..., y_N]^T \; . \tag{1}$$

The starter code includes three datasets. Each is generated from a polynomial function of a scalar input x, to which noise has been added. Each dataset comes with a *small* training set, a *large* training set, and a *test* set. Among other things this will allow you to see how different amounts of data affect model fitting.

You can visualize the training datasets. This allows you to get some ideas on which models are potentially get a good fit. You can do this using **matplotlib**, which generates a scatter plot showing the input output pairs:

1. Import matplotlib's pyplot module:
   `import matplotlib.pyplot as plt`
2. Plot the points:
   ```
   plt.scatter(data["X"], data["Y"])
   plt.show()
   ```

Once you understand how linear regression works and how to manipulate the given data, you may proceed to the next task. Note that there is no deliverable for the above tasks.

Polynomial Regression. We aim to find LS polynomial models for this problem. Each polynomial is expressed as a weighted sum of monomials, where the largest monomial degree is denoted K.

$$y = f(x) = w_0 + \sum_{k=1}^{K} w_k \, x^k \tag{2}$$

Your goals are to find the LS parameter estimates for $\mathbf{w}_K = [w_0, w_1, ..., w_K]^T$, for each K from 1 to 10. Then, you will select a single model (i.e., find a good value for K). You want a model that will give good predictions on future (unseen) test data. What follows are more detailed instructions. **Note:** w_0 is the bias term and is the first element of the vector $\mathbf{w}_K$.

In this section, You need to complete 3 methods in `polynomial_regression.py`. This file contains the class `PolynomialRegression` with various methods:

1. `__init__(self, K)`: This is the constructor of the class. `K` specifies the degree of the polynomial.
2. `predict(self, X)`: This method predicts the output for a given input, using the model parameters. `X` is a vector of inputs. The method outputs a vector of predicted outputs.
3. `fit(self, train_X, train_Y)`: This method find the LS model parameters, given training data. `train_X` is a vector of training inputs, and `train_Y` is the vector of corresponding outputs.
4. `fit_with_l2_regularization(self, train_X, train_Y, l2_coef)`: This method finds the regularized LS solution. `train_X` is a vector of training inputs, `train_Y` is a vector of training outputs, and `l2_coef` is the regularization constant, λ, that controls the amount of regularization.
5. `compute_mse(self, X, observed_Y)`: This method computes the mean squared error between predictions and observed outputs. `X` and `observed_Y` are vectors of inputs and observed outputs.

Complete the body of the methods: `predict`, `fit`, and `fit_with_l2_regularization`. Notice that the `PolynomialRegression` class has the variable `self.parameters`. It is the column vector containing $\mathbf{w}_K$. You will be using `self.parameters` for prediction, and updating it when performing model fitting (both with and without regularization). Once you have completed `PolynomialRegression`, you should do some testing to verify your solution! We have provided some basic checks on whether your implementations are correct, but they do not cover all cases. You might want to generate your own noisy polynomial data where you know the ground truth and control the amount of noise.

Written Report. Once you are confident with your implementation, you are ready to explore the ideas of over-fitting, under-fitting, and generalization. In real life, the model that you trained may not perform well in practice. There can be various reasons, but over-fitting and under-fitting are common. You will be learning about what factors contribute to these problems, and what approaches you can use to mitigate them. To this end, the scripts described below will make use of the three datasets provided in the starter code to examine how the results depend on dataset size, model complexity and the regularization constant:

1. `compare_dataset_size.py`: Compares training and test error on various training dataset sizes (under the same data distribution).

2. `compare_model_complexity.py`: Compares training and test error on various degrees of the model polynomial.

3. `compare_regularization.py`: Compares training and test error for various values of the regularization constant λ, where $\lambda \in [0, 1]$.

Look at the plots of squared errors on training and test data as functions of dataset size, polynomial degree K, and regularization coefficient λ. The plots will be generated within the dataset directories. Using short and concise paragraphs or bullet points, answer and explain in `questions.txt` what you can conclude about over-fitting, under-fitting, and generalization. What degree of polynomial was used to generate the data?

3) Image Inpainting with RBF Regression

Suppose you're given an image with missing or corrupted pixels, like the image above corrupted by red text (Fig 1 (2)). Inpainting is a process of predicting the corrupted pixels, for which we'll use RBF regression.

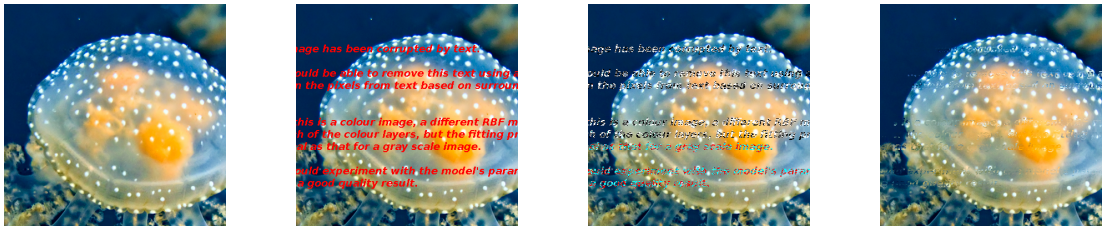

Figure 1: From left to right: (a) The original image. (b) An image corrupted with red text. (c) A badly restored image. (d) A well restored image.

Background: An image is a 2D array of pixels. For greyscale 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 $I(\mathbf{x})$ that specifies brightness as a function of position $\mathbf{x} \in \mathbb{R}^2$. Pixel values are between 0 (black) and 1 (white).

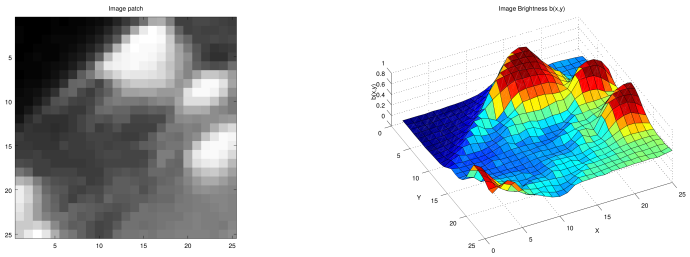

Figure 2: *(left)* Greyscale image. *(right)* Depiction of image as a height map where brighter pixels are higher.

We don't have an equation for $I(\mathbf{x})$ 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); i.e., as in Chapter 3 of the notes, let the model be

$$m(\mathbf{x}) \;=\; w_0 \;+\; \sum_k w_k \, b_k(\mathbf{x}) \;. \tag{3}$$

where b_k is the kth basis function:

$$b_k(\mathbf{x}) \;=\; \exp(-||\mathbf{x} - \mathbf{c}_k||^2/2\sigma^2)\;. \tag{4}$$

The RBF is a smooth *bump* centered at location $\mathbf{c}_k \in \mathbb{R}^2$ 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 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 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 (eg. corrupted pixels), or in between two pixels.

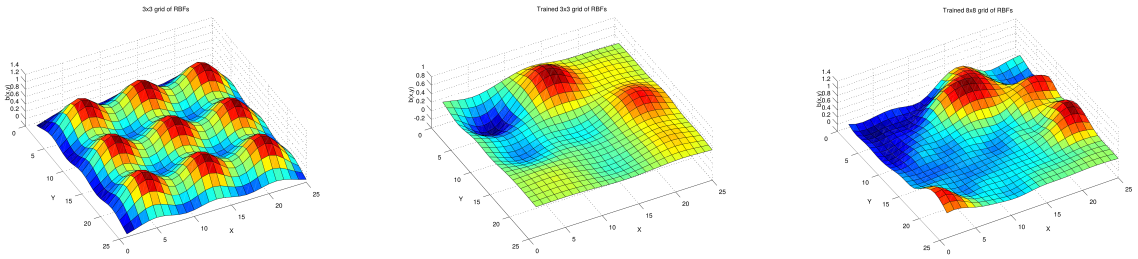

Figure 3: *(left)* An RBF model with all weight $w_k = 1$. *(middle)* A model with 3×3 RBFs with LS regression weights. *(right)* A model with 8×8 RBFs with LS regression weights. (cf. Fig. 2 (right)).

Your Task: The starter code (see `A1/Q3`) 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. To begin, decompress a1.tar, and look in `A1/Q3`.

1. Read through the 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 outputs of the i^{th} RBF. given a set of the 2D inputs. `X` is a $M \times 2$ 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 a $M \times 2$ 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 a $N \times 2$ 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 $\mathbf{w}_K$. 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.

Submission

1. Written Component

   You may hand write or type up (e.g. Word, LateX, etc.) the solutions. We strongly recommend typing it up for legibility. Compile the document as a PDF named **A1sol_StudentUTORID.pdf** (e.g. **A1sol_student1.pdf**). Then you should submit the tar file onto Quercus.

2. Programming Component

   Compress the `A1` directory into a tar file named **A1sol_StudentUTORID.tgz** (e.g., **A1sol_student1.tgz**).

   Please don't modify the method/function headers or the structure of the A1 directory tree since the automarker will fail to run. Failing to follow the instructions will result in a 25% penalty.

   To tar and zip the contents, use the command: `tar -cvzf name_of_tar.tgz A1`. Make sure to decompress the tar file to ensure the entire A1 directory is there. Then you should submit the tar file onto Quercus.