COMP24112 – Exercise 3: Face Recognition Report
April 2023

3.2 Explain briefly the knowledge supporting the implementation and design step by step. Explicitly comment on the role of any arguments you have added to your functions.

The “l2_rls_train(data, labels, lmbd)” function is used to train a linear regression model using the L2 regularized least squares method. The function takes in three arguments: data, which is a numpy array of shape (n_samples, n_features) containing the training data, “labels”, which is a numpy array of shape (n_samples,) containing the corresponding labels for the training data, and lmbd, which is a float representing the regularization strength. A larger lambda value results in a bigger regularization penalty, leading to a more constrained model with smaller coefficient values. A smaller lambda value, on the other hand, leads to a less constrained model with greater coefficient values. The function begins by appending a column of ones to the data matrix “X”. This is a typical approach for making the bias component in the linear regression model easier to compute. The weight vector “w" is then calculated based on whether lmbd value is 0 (no regularization) or not. Firstly we should check to see if the hyper-parameter is zero. If it is, we will use pseudo inverse to compute the weight vector, here we use a method called “linalg.pinv” from NumPy to calculate the (Moore-Penrose) pseudo-inverse of a matrix. Otherwise, we are going to use the Ridge Regression formula, which is the formula below:

$$\mathbf{W} = \left(\tilde{\mathbf{X}}^T\tilde{\mathbf{X}} + \lambda\mathbf{I}\right)^{-1}\tilde{\mathbf{X}}^T\mathbf{Y}.$$

Here is the formula for calculating the weight vector where X is the data with “1”s as the first column, Y is the label of X, λ is the hyperparameter and I is the identity matrix.
The l2_rls_predict(w, data) function takes in two arguments: “w”, which is the numpy array containing the model parameters obtained from the training function, and “data”, which is a numpy array containing the input features for prediction. To predict the output, firstly, a column of ones is added to the input data matrix, just like in the training function. This accounts for the bias term in the linear regression model, then the function computes the predicted labels by multiplying this matrix with the weight vector “w”.

4.2 Explain the classification steps, and report your chosen hyper–parameter and results on the test set.

There are a few steps in classification. Before the whole classification starts, we split the data into training and testing sets, each set accounts for 50% of the total data. It also implements “one hot encoding” using the library named “sklearn.preprocessing” to represent categorical data as numerical data. In this code, the labels are first transformed into numerical values, and then into binary vectors. This is done separately for the training and test sets. Here are some additional details on how the labels are encoded using one-hot encoding for multi-class classification:
The “LabelEncoder” class is used to transform the labels into numerical values. This assigns a unique numerical value to each class label. The “OneHotEncoder” function is then used to transform the numerical labels into binary vectors using one-hot encoding. This creates a binary vector of length “n” for each label, where “n” is the number of unique classes in the dataset. The binary vector has a 1 in the position corresponding to the index of the class and 0s in all other positions.
The lambda value is chosen from 10^{-5} to 10^{5}, each time the index increases by 0.375. We used the K-fold method for hyperparameter selection. This means the whole data set divides into k sections, where k is 5. In different iterations, one part becomes the validation set, and the other parts of the data set for training. The model makes a prediction by applying the trained L2-regularized least squares regression model to the test data. The “l2_rls_predict(w, data)” function takes in the trained model parameters (w) and the test data (te_data), and returns the predicted

labels. The “argmax()” function is used to select the class with the highest predicted probability for each test instance.
The best hyper-parameter according to my experiment is 1.1, and the accuracy on the test set is 89.12%. The confusion matrix for testing data is shown in part 4.1.

Did you notice any common features among the easiest and most difficult subjects to classify? Describe your observations and analyze your results.

For the most difficult objects, If there is a significant difference in facial expression between the left and right sides of the person's face, such as opening the mouth or closing the eyes, making strange faces(e.g.😜), the model may be disturbed. The model may also be affected if the lighting circumstances in which the face was captured change, such as shadows or reflections. For instance, subjects with glasses may become harder to classify as glasses will add obscurity to the greyscale levels due to extra bold features (reflections, etc).
For the easiest subjects the model predicted, the left face and the right face are symmetrical, and the shooting light is pointed directly at the face (the light is not projected on the face from the side). All photos are taken from the middle front side of faces.

5.2 Report the MAPE and make some observations regarding the results of the face completion model. How well has your model performed? Offer one suggestion for how it can be improved.

The mean absolute percentage error (MAPE) for the face completion model is 20.78 % (round off to 2 decimals place). According to my observations, the model performed better when the missing components are simpler, such as the lower half of the face with little facial hair or accessories. More complex face features, such as beards, glasses, or sophisticated haircuts, are difficult for the model to anticipate, resulting in fewer accurate predictions in these situations.
As the model's performance is not consistent across all test situations, particularly for complex facial characteristics and variable lighting circumstances, I would suggest using more high-quality images to improve the performance of the face completion model, which means using higher-resolution images, making sure photographs are captured under perfect lighting conditions.

6.3 Analyze the impact that changing the learning rate has on the cost function and obtained testing accuracies over each iteration in experiment 6.2. Drawing from what you observed in your experiments, what are the consequences of setting the learning rate and iteration number too high or too low?

Here we executed the experiment three times, the only changing variable is the learning rate. The learning rate we utilized are $10^{(-2)}$, $10^{(-3)}$,$10^{(-4)}$, the iteration number N is 200. Please check graphs under part 6.2.
Generally, we obtained a decreasing sum-of-squares error loss as the number of iterations increased for learning rates of $10^{(-3)}$ and $10^{(-4)}$, It drops from 3.0 at the beginning to 0 at the end. The testing accuracies have been improved from 0.5 to 1. For the learning rate of $10^{(-4)}$, the algorithm starts with a very small step size and may converge very slowly to the minimum of the cost function. This can explain why the loss drops slower than the performance of $10^{(-3)}$ case but does not exhibit any sudden changes. For the learning rate $10^{(-2)}$, we gained a consistent testing error of 0.5 across all iterations, the sum-of-squares error loss remains constant for a while and then suddenly jumps very quickly. This behavior is known as divergence, and it suggests that the learning rate $10^{(-2)}$ is too large for the data. As a result, the weight will be overly heavy, affecting accuracy. The gradient update function could explain it directly for this:

$$w^{(t+1)} = w^{(t)} - \eta\left(\tilde{X}^T\tilde{X}w^{(t)} - \tilde{X}^T Y\right)$$

In summary, the learning rate chosen influences the optimization algorithm's convergence rate and stability. If the learning rate is too high, the algorithm may overshoot the minimum of the cost function and fail to converge, while a low learning rate can lead to slow convergence. Setting the learning rate and iteration number too high or too low may lead to poor model performance, either by overshooting, getting stuck in a local minimum, over-fitting, or under-fitting.

Explain in your report the following: (1) Your implementation of "hinge_gd_train". (2) Your experiment design, comparative result analysis and interpretation of obtained results.

The "hinge_gd_train" function trains a linear model using the hinge loss and L2 regularization. The hinge loss is a loss function used in SVMs for classification problems. It penalizes predictions that are on the wrong side of the margin and gives no penalty for predictions that are on the correct side of the margin or inside the margin. The function takes the feature matrix "data", target labels "labels", learning rate "learning_rate", number of iterations "N", and regularization hyperparameter "C" as inputs. The implementation first expands the feature matrix by adding a column of ones for the bias term. It also initializes the weight vector to zeros and creates empty arrays to store the weights and costs for each iteration. It then calculates the derivative of the hinge loss.
This is the gradient function:

$$g(w) = \min_{(\mathbf{w},w_0)\in\Re^{d+1}} C\sum_{i=1}^{N}\max\left(0,1-y_i\left(\mathbf{w}^T\mathbf{x}_i + w_0\right)\right) + \frac{1}{2}\mathbf{w}^T\mathbf{w}$$

To get the gradient we differentiate the loss with respect to **i**th component of w. Then we rewrite hinge loss in terms of w as $f(g(w))$ where f(g)=max(0,1−y·g) and $g(w)$=$\mathbf{x}\cdot\mathbf{w}$, then we could use chain rule to get

$$\frac{\partial}{\partial w_i} f(g(w)) = \frac{\partial f}{\partial g}\frac{\partial g}{\partial w_i}$$

First derivative term is evaluated at $g(w)$=$x\cdot w$ becoming $-y_i$ when $y_i\left(w^T x_i + w_0\right) < 1$, and 0 when $y_i\left(w^T x_i + w_0\right) >= 1$. Second derivative term becomes x_i. So in the end we get

$$\frac{\partial g(\mathbf{w})}{\partial \mathbf{w}} = \mathbf{w} + C \times \sum_{i=1}^{n}\begin{cases} 0 & \text{if } y_i\left(w^T x_i + w_0\right) \geq 1 \\ -y_i\mathbf{x}_i & \text{otherwise} \end{cases}$$

The cost function used in this implementation is the regularized hinge loss, which adds a regularization term to the hinge loss to prevent overfitting. The regularization term is controlled by the hyper-parameter C, which determines the strength of the regularization. A higher C value means a greater regularization term, which penalizes the model for having large parameter values. This can help to avoid overfitting. The cost function is calculated for each iteration and stored in the "cost_all" array. The weight vector is updated using the gradient descent update rule, and it is stored in the "w_all" array for each iteration.
For the experiment design, we used the same parameters, both with an iteration number of 200 and a learning rate of 0.001. The hyper-parameters(C) for hinge loss we set to 1 and 0.5. We plotted the performances of both least square loss and hinge loss model in the same graph, and we can see in the sum of square error loss graph, the line of least square loss declines faster and eventually reaches zero, but hinge loss does not, although it is eventually infinitely close to 0. We can also find the line of hinge loss declines more slowly and with some fluctuations. In the test accuracy plot, the hinge loss also performs less well than the least squares loss and is followed by some strong fluctuations. This is because the hinge loss is a non-smooth, non-convex loss function, which means that it has a less well-behaved optimization landscape than the least squares loss. We may also discover that a higher C value (C=1) results in a smaller margin, which may result in overfitting and poor performance on the test data as compared to C=0.5.