AI701: Foundations of Artificial Intelligence (Fall 2022)

Assignment 1

Mario Cantero
22010372

Farah Alharthi
22010348

Salma Alrashdi
22010399

{mario.cantero, farah.alharti, salma.alrashdi}@mbzuai.ac.ae

September 23, 2022

1 Linear Regression

The first Jupyter Notebook corresponds to the design of a linear regression model. This statistical tool is used to predict the petal width given an observed dataset of petal length. An analytical solution is studied for this regression task.

Using a dataset of ScikitLearn, a small amount of observations of the data is already provided, as well as some functions like scatter plotting the observations, plotting the model, and showing the residuals. Some cells provide explanations of the linear model using such functions. This is all done before commencing the tasks.

1.1 Computing the MSE: task 1

The first task consists on implementing the mean square error, which is a way to track how well is our model fitting the data using the residuals:

$$MSE = \frac{1}{N}\sum_{i=0}^{N-1}(\hat{y}_i - y_i)^2 \tag{1}$$

We defined a function named *mean_squared_error* and passed three parameters: a as the coefficient of the variable, b as the intercept value, and *data* as the dataset containing the observations and their corresponding labels. Using vectorization, the MSE function was defined returning this variable:

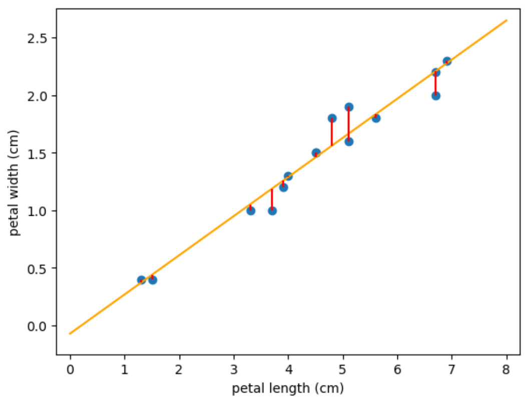
```
mse = np.sum(np.power(a*x+b-y, 2))/y.shape[0]
```

1.2 Computing the least squares estimate: task 2

The least squares estimate is an analytical approach to calculate the coefficients a and b. We defined a function named *iris_least_squares_estimate* that calculates the LSE. Some of its calculations include:

```
sum_xy = np.sum((x-mean_x)*(y-mean_y)) # Numerator
sum_x2 = np.sum(np.power(x-mean_x,2))  # Denominator
a = sum_xy/sum_x2
b = mean_y-a*mean_x
```

Next, we plotted the resulting model from the coefficients calculated by our function using a provided procedure. Lastly, by mean squared error function we calculate the error of our points corresponding to the line which is approximately 0.013.



After plotting our model, we added the test data to the plot and noticed that our estimator does not pass through them perfectly. In addition to this, we calculated the predicted petal widths and compared them to the ground truths. To get a sense of how well the model is doing, the MSE was calculated next with the test data, giving a value of approximately 0.0598.

The linear regression model can also be applied to higher dimensional features. In this case, we changed the problem to house pricing prediction using features like size and location. First, we concatenated all the attributes into a big data matrix X and then we add the vector of ones to consider the intercept term of the model. To compute the least square estimate of this dataset, we just performed matrix operations like so:

```
    a = np.linalg.inv(X.T@X)@X.T@y
```

Finally, we created a 3D plane to visualize our model.

2 Logistic Regression

In this part of the assignment, we defined a decision boundary that separated the data into two groups using logistic regression with a normally-distributed, toy dataset.

2.1 Sigmoid function + predictive probability: task 1 & 2

First, we implemented the sigmoid function $\sigma(\mathbf{x}) = \frac{1}{1+exp^{-(ax_1+bx_2+c)}}$ in a function called *sigmoid(x)*. This sigmoid function can be used as a predictive probability function just by passing the linear model as an input like so:

```
    pred_prob = sigmoid(a*x[:,0]+b*x[:,1]+c)
```

2.2 Loss function: task 3

For a loss function, we used Binary Cross Entropy (BCE) loss to work out the ideal decision boundary that maximizes the probabilities of the training data of being in their true classes. We added $1.0 * e - 15$ in the calculation of the logarithm function to avoid $log(0)$ cases. It measures how well our decision boundary classifies correctly.

2.3 Gradient descent: tasks 4 & 5

We implemented the derivatives of BCE with respect to each weight in our linear equation in the gradient descend.

```
def compute_gradients(x, a, b, c, y):
    grad_a = np.sum(np.multiply(predictive_prob(x, a, b, c)-y, x[:,0]))
    grad_b = np.sum(np.multiply(predictive_prob(x, a, b, c)-y, x[:,1]))
    grad_c = np.sum(predictive_prob(x, a, b, c)-y)
    return grad_a, grad_b, grad_c
```

In gradient descent, we changed the weights in the direction that minimizes the loss function for a number of iterations with $a^{t+1} = a^t - \alpha * \frac{\delta L}{\delta a}$. Note that the values are updated after the calculation of the gradients in each iteration.

After each iteration of the gradient descent algorithm, the BCE decreased until finding the optimal values of $a \approx 3.7$, $b \approx 2.8$, and $c \approx -3.1$. We used a learning rate of 0.005 and 100 iterations.

2.4 Evaluation: task 6

We evaluated the accuracy of our algorithm using a test set. First, we rounded every predicted probability higher or equal than 0.5 to 1, and to a value of 0 in the contrary case. Then, we counted the number of times the prediction matched the actual label, then divided by the number of total observations.

Our accuracy was 1 because it predicted all test observations correctly. The accuracy of our model was good because our training data was spread in a way that made it easier to fit a decision boundary between two clusters.

3 Support Vector Machines

The third and last notebook includes some tasks on text classification using support vector machines (SVM). A partial dataset of tweets by American democrats and republicans is used and provided by the professor. The objective of this activity is to study the robustness of SVM on text classification.

Initially, there is a really basic exploratory data analysis on not processed data. Using some Pandas functions, we know that there's 222 Republicans and 211 Democrats. In addition, if we want to know how many tweets per politician are there we can use a method for grouping by politician. Before commencing with the tasks, the data was processed by tokenizing each tweet using TF-IDF, followed by the splitting of training and testing data with a 80-20 proportion.

3.1 Train an SVM: task 1

Using the ScikitLearn framework's SVM model, it is being asked to iterate over four kernels using a fixed C. In this case a value of 1 for C was used with *linear*, *rbf*, *poly*, and *sigmoid* kernel. In every iteration the method *.score()* is used and its result is printed to see how well the model did on both the seen and the unseen data. The results obviously show that the model performs well on trained data in contrast to the test data. Most of the time the best model uses the radial-basis function, but with a random state of 5, it is better to use a sigmoid function.

3.2 Find the best model parameters: task 2

Later, a list of C values and kernels are passed to the object *gcv* created with the class *GridSearchCV* fitting the training data. The result of this search saves the best model's hyperparameters on the object's attributes, which corresponds (depending on the random state) to using a an *rbf* kernel and a C value of 1.7. The time taken by the grid search cross-validation to go through all combinations of the hyperparameters was relatively large, and that's why the time was estimated. The total time it took was approximately 29 seconds.

Finally, a classification report is used to learn what type of party is easier to classify by using f1-score as the metric. This metric takes the harmonic mean between precision and recall that maps to a number between [0, 1). Using the same random state, it is easier to classify the Republican party.