ECE421 Assignment 1 - Question and Answer

Edwin Chacko
1009149716
edwin.chacko@mail.utoronto.ca

Balakannan Murali
1009454494
balakannan.murali@mail.utoronto.ca

September 26, 2024

1 Pocket Algorithm

1.1 The Pocket Algorithm Using NumPy

1.2 The Pocket Algorithm Using scikit-learn

1.2.1 Refer to the documentation, what is the functionality of the tol parameter in the Perceptron class? (2 marks)

In the pocket algorithm, the tolerance (tol) parameter introduces a stopping criterion to prevent unnecessary training when the improvement in the loss function becomes negligible. Specifically, training stops when the loss satisfies the condition:

"If it is not None, the iterations will stop when ($loss > previous\_loss - tol$)."

The tol parameter controls the tolerance for stopping the algorithm. If the change in loss is smaller than tol, the algorithm stops to prevent unnecessary iterations. This helps ensure efficient training by halting the process once further updates provide minimal gains, preventing overfitting and saving computational resources. The value of tol should balance between avoiding early stopping and avoiding excessive training time.

1.2.2 If we set max_iter=5000 and tol=1e-3 (the rest as default), does this guarantee that the algorithm will pass over the training data 5000 times? If not, ensure that the algorithm will pass over the training data 5000 times? (2 marks)

Setting max_iter=5000 and tol=1e-3 **does not** guarantee the algorithm will pass over the training data 5000 times (epochs) because the tol parameter (tolerance) introduces an early stopping criteria. If the loss between iteration becomes less than 1e-3, training stops and the model converges.
To ensure that the algorithm will pass over the training data 5000 times, we can set the tol=None which forces the algorithm to run over the maximum number of iterations set (max_iter=5000) regardless of the loss function. The early stopping criteria is stopped by doing this resulting in the desired number of iterations.
No other parameters need changing as the default parameters, along with max_iter=5000 and tol=None, support running 5000 iterations. This can be verified by setting the parameter "verbose=1" to see the output for each epoch.

1.2.3 How can we set the weights of the model to a certain value? (2 marks)

Manually setting weights to certain value useful for controlled experiments, debugging, transfer learning, and fine-tuning. It also ensures model comparisons and supports simulations or analysis by controlling certain learning dynamics.
In the Perceptron class, "self.coef_" represents the weights assigned to the features, with a shape of "(1, N_features)" for binary classification. Similarly, "self.intercept_" refers to the bias (constants) in the decision function, with a shape of "(1,)" for binary classification. To explicitly set the weights of the model to a certain value, we can modify "self.coef_" appropriately. And for the bias, modify "self.intercept_".

1.2.4 How close is the performance (through confusion matrix) of your NumPy implementation in comparison to the existing modules in the scikit-learn library? (2 marks)

Comparing the NumPy and scikit-learn implementations through their respective Confusion Matrices reveals that they are close but different. However there is some nuance to be considered. Both implementations share 8 true positives (TP) and 0 false negatives (FN) but differ as the NumPy has 9 true negatives (TN) and 3 false positives (FP) while the scikit-learn has 10 TN and 2 FP.
The nuance arises as the scikit-learn Perceptron.fit(), has parameter shuffle which is True by default. This leads to a different order in training cases than the NumPy implementation. This has slight effects on the final weights and can still perform the same on the training data but differently on the test set.

When setting shuffle=False, we see the exact same confusion matrix for both sets, where there are 3 FP and 9 TN. This reveals that the difference in performance is a result of shuffling which causes one of the test cases to be classified correctly compared in the scikit-learn implementation.

2 Linear Regression

2.1 Linear Regression Using NumPy

2.1.1 When we input a singular matrix, the function linalg.inv often returns an error message. In your fit_LinRegr(X_train, y_train) implementation, is your input to the function linalg.inv a singular matrix? Explain why. (2 marks)

Matrix inversion of a square matrix $A \in \mathbb{R}^{n \times n}$, A^{-1}, is given by,

$$A^{-1} = \frac{1}{det(A)} \cdot adj(A) \tag{1}$$

where $det(A)$ is the determinant of A and $adj(A)$ is adjoint of A. This equation is not defined for singular matrices (matrices whose determinant is 0).

When there is dependence in the rows or columns of a matrix, the matrix is singular. In our fit_LinRegr(X_train, y_train), the input given in test 6 becomes,

$$X\_train = \begin{bmatrix} 1 & 1 & 2 \\ 1 & 2 & 4 \\ 1 & 3 & 6 \\ 1 & 4 & 8 \end{bmatrix} \tag{2}$$

which has dependent columns and rows (eg, $2 \cdot col_2 = col_3$). This dependence means X_train is a singular matrix - a matrix whose determinant is 0. Therefore, the matrix inverse is not defined for this example, which is why an error is thrown.

2.1.2 As you are using linalg.inv for matrix inversion, report the output message when running the function subtestFn(). We note that inputting a singular matrix to linalg.inv sometimes does not yield an error due to numerical issue. (1 marks)

The try-except block handles the error fine and prints $ERROR$ to the terminal. Removing this guardrail invokes NumPy's output for this error:

```
Traceback (most recent call last):
    ...
    w = fit_LinRegr(X, y
    ...
numpy.linalg.LinAlgError: Singular matrix
```

This result is consistent with our earlier analysis of what causes this error (we are attempting to invert a matrix for which the matrix inverse operation is not defined).

2.1.3 Replace the function linalg.inv with linalg.pinv, you should get the model's weight and the "NO ERROR" message after running the function subtestFn(). Explain the difference between linalg.inv and linalg.pinv, and report the model's weight. (2 marks)

Using the linalg.pinv function leads to different output:

```
------------subtestFn------------
weights:  [-2.10942375e-15  2.00000000e-01  4.00000000e-01]
NO ERROR
```

According to the NumPy documentation, the linalg.inv method raises the LinAlgError if a is detected to be singular. The precise implementation regarding how the inverse is calculated is not mentioned, however similar libraries use LU Decomposition.
According to the NumPy documentation, the linalg.pinv computes the pseudo inverse of a matrix, A, using singular value decomposition (SVD) of A. The method involves the SVD of A into $A = Q_1 \Sigma Q_2^T$ and using a theorem that states $A^+ = Q_2 \Sigma^+ Q_1^T$, where $Q_{1,2}$ are orthogonal matrices, Σ is a diagonal matrix consisting of A's singular values, and Σ^+ is the diagonal matrix consisting of the reciprocals of A's singular values.
The difference is in the methodology used to compute the pseudo-inverse. linalg.inv is used in the approach introduced in class, which required taking the inverse of $X^T X$, which may not be invertible. linalg.pinv uses SVD which does not need to compute $(X^T X)^{-1}$.

2.2 Linear Regressions Using scikit-learn

Discussion available as a comment at the end of the code file, "LinearRegressionImp.py", as per the handout.

3 Discussion

3.1 How much time did you spend on each part of this assignment? (optional)

Overall maybe 10hrs combined in the assingment. I spent the longest time on the pocket algorithm (after already implementing it) as the part of the data chosen is such that the algorithm finds the best weights on the first epoch. This led to me thinking my copy of w was shallow. I ultimately realized it was fine after changing the train_test_split seed.

3.2 Any additional feedback? (optional)

Enjoyed it overall