ECE421: Introduction to Machine Learning
Programming Assignment 1
Assigned: Jan 16, 2023; Due: Feb 3, 2023 @ 11:59 p.m.

Objectives

In this assignment, you will be implementing the following algorithms: **Pocket Algorithm**, and **Linear Regression**; using two different methods of coding approaches. In the first approach, you will implement these algorithms using `Python` and functions in the `NumPy` library only. In the second approach, you will use `scikit-learn` to gauge how well your initial implementation using `NumPy` functions fares in comparison to off-the-shelf modules available in `scikit-learn`. You will also be asked to answer several questions related to your implementations. **To avoid any potential installation issue, you are encouraged to develop your solution using Google Colab notebooks.**

Requirements

In your implementations, please use the function prototype provided (i.e. name of the function, inputs and outputs) in the detailed instructions presented in the remainder of this document. We will be testing your code using a test function that which evokes the provided function prototype. If our testing file is unable to recognize the function prototype you have implemented, you can lose significant portion of your marks. In the assignment folder, the following files are included in the `starter_code` folder:

- `PerceptronImp.py`
- `LinearRegressionImp.py`

These files contain the test function and an outline of the functions that you will be implementing. You also need to submit a separate `PA1_qa.pdf` file that answer questions related to your implementations.

1 Pocket Algorithm

In the `PerceptronImp.py` file, you will implement the Pocket Algorithm to classify two different classes in the IRIS dataset. You do not need to download the dataset as it is included in the `scikit-learn` library. For evaluation, we provide you the test function `test_Part1()`(**Note: keep this function as it is in your submission**). This function loads the IRIS dataset, runs your implementation and outputs the confusion matrix on the test set (for each coding approach). Follow the below instructions to get started.

Part 1a: Pocket Algorithm using NumPy and Python

You will be implementing the Pocket Algorithm using the `NumPy` library functions in the `PerceptronImp.py` file. You will be computing parameters of a linear plane that best separates input features belonging to two classes. Specifically, you will be implementing four functions which are detailed in the following:

- Function 1: `def fit_Perceptron(X_train, y_train)`
  - Inputs: `X_train, y_train`
    The first input `X_train` represents the matrix of input features that belongs to $\mathbb{R}^{NXd}$ where N is the total number of training samples and d is the dimension of each input feature vector. The second input `y_train` is an N dimensional vector where the i^{th} component represents the output observed in the training set for the i^{th} row in `X_train` matrix which corresponds to the i^{th} input feature. Each element in `y_train` takes the value $+1$ or -1 to represent the first class and second class respectively.

- Output: `w`
  The output of this function is the vector `w` that represents the coefficients of the line computed by the pocket algorithm that best separates the two classes of training data points. The dimensions of this vector is $d + 1$ as the offset term is accounted in the computation.
- Function implementation considerations:
  This function computes the parameters `w` of a linear plane which separates the input features from the training set into two classes specified by the training dataset. As the pocket algorithm is used, you will set the maximum number of epochs (the maximum number of passes over the training data) to 5000. Useful functions in `NumPy` for implementing this function are: `zeros` (for initializing the weight vector with 0s), `shape` (for identifying the number of rows and columns in a matrix), `hstack` (to add an additional column to the front of the original input matrix), `ones` (for setting the first column of the input feature matrix to 1), `dot` function to take the dot product of two vectors of the same size. You will also use the function `errorPer` that you will implement next to compute the average number of misclassifications for the plane you are currently considering with respect to the training dataset.

- Function 2: `def errorPer(X_train, y_train, w)`
  - Inputs: `X_train`, `y_train` and `w`
    The inputs to this function are `X_train` and `y_train`. `X_train` is defined with the additional column of ones and `y_train` is defined in a manner similar to Function 1. `w` represents the coefficients of a linear plane and is of $d + 1$ dimensions.
  - Output: `avgError`
    The output of this function is the average number of points that are misclassified by the plane defined by `w`.
  - Function implementation considerations:
    You will use the `pred` function that you will implement in Function 3 to find the output of the classifier defined by `w`.
- Function 3: `def pred(X_i,w)`
  - Inputs: `X_i`, `w`
    The first input is `X_i` which is the feature vector of $d + 1$ dimensions of the i^{th} test datapoint. `w` is defined in a manner similar to Function 2.
  - Output: Class label
    The class predicted for linear classifier defined by `w` for the input datapoint `X_i` is computed. This output can take one of the following values: $+1$ and -1.
  - Function implementation considerations:
    If the dot product of the vector `w` and the input point `X_i` is **strictly** positive, then you can consider the input point to map to a point that is above the line (i.e. belongs to class 1). Otherwise, it will belong to class -1.
- Function 4: `def confMatrix(X_train,y_train,w)`
  - Inputs: `X_train`, `y_train` and `w`
    These inputs are defined in the same manner as Function 1.
  - Output: A two-by-two matrix composed of integer values
  - Function implementation considerations:
    This function will populate a two-by-two matrix. Using the zero-index, the $(0, 0)$ position represents a count of the total number of points correctly classified to be class -1 (True Negative). The $(0, 1)$ position contains a count of total number of points that are in class -1 but are classified to be class $+1$ by the classifier (False Positive). The $(1, 0)$ position contains a count of total number of points that are in class $+1$ but are classified to be class -1 by the classifier (False Negative). The $(1, 1)$ position represents a count of the total number of points correctly classified to be class $+1$ (True Positive). Refer to the table below.

		Predicted	
		-1	+1
Labels	-1	True Negative	False Positive
	+1	False Negative	True Positive

The following is the mark breakdown for Part 1a:

- Test file successfully runs all four implemented functions: 8 marks
- Outputs of all four functions are close to the expected output: 12 marks
- Code content is organized well and annotated with useful comments: 10 marks

Part 1b: Pocket Algorithm using scikit-learn

In this part, you will use the `scikit-learn` library to train the binary linear classification model. You will then compare the performance of your implementation in Part 1a with the one available in the `scikit-learn` library. You will implement one function in this part in the `PerceptronImp.py` file. You can refer to the `scikit-learn` demo covered in the lecture to aid you with this completing this part or refer to this page: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Perceptron.html.

- Function: `def test_SciKit(X_train, X_test, Y_train, Y_test)`
  - Inputs: `X_train, X_test, Y_train, Y_test`
    The first input `X_train` represents the matrix of input features that belongs to $\mathbb{R}^{NXd}$ where N is the total number of training samples and d is the dimension of each input feature vector. The second input `X_test` represents the matrix of input features that belongs to $\mathbb{R}^{MXd}$ where M is the total number of testing samples and d is the dimension of each input feature vector. The third input `Y_train` is an N dimensional vector where the i^{th} component represents the output observed in the training set for the i^{th} row in `X_train` matrix which corresponds to the i^{th} input feature. The similar counterpart to `X_test` is `Y_test`.
  - Output: A two-by-two matrix composed of integer values
    This function will output the result obtained from the confusion matrix function imported from `sklearn.metrics` library to report the performance of the model fitted using the Perceptron algorithm available in the `sklearn.metrics` library.
  - Function implementation considerations:
    `Perceptron` and `confusion_matrix` functions imported from `sklearn.linear_model`, `sklearn.metrics` will be utilized to fit the linear classifer using the Perceptron learning algorithm and evaluate the performance of this algorithm. As presented in the `scikit-learn` demo in the lecture, you will initiate an object of the Perceptron type, you will run the `fit` function to train the classifier, you will use the `predict` function to perform predictions using the trained algorithm and finally you will use the `confusion_matrix` function to identify how many points have been classified correctly and incorrectly in the test dataset. **Refer to the below questions** in order to set the parameters for your `Perceptron` model.

Answer the following question(s), write and save your answer in a separate `PA1_qa.pdf` file. Remember to submit this file together with your code.

1. Refer to the documentation, what is the functionality of the `tol` parameter in the Perceptron class? (2 marks)
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, which parameters (and values) should we set to ensure that the algorithm will pass over the training data 5000 times? (2 marks)
3. How can we set the weights of the model to a certain value? (2 marks)

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)

The following is the mark breakdown for Part 1b:

- Test file successfully runs implemented function: 4 marks
- Output is close to the expected output from the test file: 5 marks
- Code content is organized well and annotated with comments: 3 marks
- Questions are answered correctly: 8 marks

2 Linear Regression

In the `LinearRegressionImp.py` file, you will implement the Linear Regression algorithm and test its performance using the diabetes dataset. You do not need to download the dataset as it is included in the `scikit-learn` library. For evaluation, we provide you the test function `test_Part2()` (**Note: keep this function as it is in your submission**). This function loads the diabetes dataset, runs your implementation and outputs the mean-squared-error on the test set (for each coding approach). Follow the below instructions to get started.

Part 2a: Linear Regression using NumPy and Python

You will be implementing in `LinearRegressionImp.py` the exact computation of the solution for linear regression using the `NumPy` library functions via the least squares method. You will be computing the parameters of a linear plane that best fits the training dataset. Specifically, you will be implementing three functions which are detailed in the following:

- Function 1: `def fit_LinRegr(X_train, y_train)`
  - Inputs: `X_train, y_train`
    The first input `X_train` represents the matrix of input features that belongs to $\mathbb{R}^{NXd}$ where N is the total number of training samples and d is the dimension of each input feature vector. The second input `y_train` is an N dimensional vector where the i^{th} component represents the output observed in the training set for the i^{th} row in `X_train` matrix which corresponds to the i^{th} input feature. Each element in `y_train` takes a value in $\mathbb{R}$.
  - Output: `w`
    The output of this function is the vector `w` that represents the coefficients of the line computed using the least square method that best fits the training data points. The dimensions of this vector is $d + 1$ as the offset term is accounted in the computation.
  - Function implementation considerations:
    This function computes the parameters `w` of a linear plane which best fits the training dataset. Useful functions in `NumPy` for implementing this function are: `shape` (for identifying the number of rows and columns in a matrix), `hstack` (to add an additional column to the front of the original input matrix), `ones` (for setting the first column of the input feature matrix to 1), `dot` function to take the dot product of two vectors of the same size, `transpose` function for taking the transpose of a vector, and `linalg.inv` function for finding the inverse of a square matrix.
- Function 2: `def mse(X, y, w)`
  - Inputs: `X`, `y` and `w`
    The inputs to this function, `X` and `y`, are defined in a manner similar to `X_train` and `y_train` in Function 1. `w` represents the coefficients of a linear plane and is of $d + 1$ dimensions.

- Output: `avgError`
  The output of this function is the mean squared error introduced by the linear plane defined by `w`.
- Function implementation considerations:
  You will use the `pred` function that you will implement in Function 3 to find the output of the linear plane defined by `w`. Functions from `NumPy` that will be useful are: `shape` (for identifying the number of rows and columns in a matrix), `hstack` (to add an additional column to the front of the original input matrix), `ones` (for setting the first column of the input feature matrix to 1), and `dot` function to take the dot product of two vectors of the same size.

- Function 3: `def pred(X_i,w)`
  - Inputs: `X_i`, `w`
    The first input is `X_i` which is the feature vector of $d+1$ dimensions of the i^{th} test datapoint. `w` is defined in a manner similar to Function 2.
  - Output: Predicted value
    The output predicted by the linear regression model defined by `w` for the input datapoint `X_i` is computed. This output can take values in the Real space $\mathbb{R}$.
  - Function implementation considerations:
    The `dot` product function in `NumPy` will be useful for this function implementation.

For this implementation, we also provide the test function `subtestFn()`. This function loads a toy dataset, runs your NumPy implementation and return a message indicating whether your solution works when `X_train` is not a full-column rank matrix, i.e. the input features are not linearly independent.

Answer the following question(s), write and save your answer in a separate `PA1_qa.pdf` file. Remember to submit this file together with your code.

- 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)
- 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)
- 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)

The following is the mark breakdown for Part 2a:

- Test file successfully runs all three implemented functions: 8 marks
- Outputs of all four functions are close to the expected output: 12 marks
- Code content is organized well and annotated with useful comments: 5 marks
- Questions are answered correctly: 5 marks

Part 2b: Linear Regressions using scikit-learn

In this part, you will use the `scikit-learn` library to train the linear regression model. You will then compare the performance of your implementation in Part 2a with the one available in the `scikit-learn` library. You will implement one function in this part in the `LinearRegressionImp.py` file. You can refer to the `scikit-learn` demo covered in the lecture to aid you with this completing this part.

- Function: `def test_SciKit(X_train, X_test, Y_train, Y_test)`

- Inputs: `X_train, X_test, Y_train, Y_test`
  The first input `X_train` represents the matrix of input features that belongs to $\mathbb{R}^{NXd}$ where N is the total number of training samples and d is the dimension of each input feature vector. The second input `X_test` represents the matrix of input features that belongs to $\mathbb{R}^{MXd}$ where M is the total number of testing samples and d is the dimension of each input feature vector. The third input `Y_train` is an N dimensional vector where the i^{th} component represents the output observed in the training set for the i^{th} row in `X_train` matrix which corresponds to the i^{th} input feature. The similar counterpart to `X_test` is `Y_test`.
- Output: `error`
  This function will output the mean squared error on the test set, which is obtained from the `mean_squared_error` function imported from `sklearn.metrics` library to report the performance of the model fitted using the linear regression algorithm available in the `sklearn.metrics` library.
- Function implementation considerations:
  `LinearRegression` and `mean_squared_error` functions imported from `sklearn.linear_model`, `sklearn.metrics` will be utilized to fit the linear classifier using the linear regression algorithm and evaluate the performance of this algorithm. As presented in the `scikit-learn` demo in the lecture, you will initiate an object of the `LinearRegression` type, you will run the `fit` function to train the model, you will use the `predict` function to perform predictions using the trained algorithm and finally you will use the `mean_squared_error` function to compute the mean squared error of the trained model.

How close is the performance of your implementation in comparison to the existing modules in the `scikit-learn` library? Place this comment at the end of the code file.

The following is the mark breakdown for Part 2b:

- Test file successfully runs implemented function: 6 marks
- Output is close to the expected output from the test file: 8 marks
- Code content is organized well and annotated with comments: 6 marks