Machine Learning Coursework Part II
ELEC60019/70059
Autumn 2024/2025

Student CID: 02215217 Student Username: zj722 Student Level: 3^{rd} year

1. Perceptron

Task 1 **[40 marks]**

I selected the Setosa and Versicolor classes from the Iris dataset,focusing on petal length and sepal length as features. by extracting praticular row in iris dataset (row 0 to 49 for Setosa and 50 to 99 for Versicolor) as shown in list 1.

```
setosa_label = iris_data.iloc[0:50,4].values
versicolor_label = iris_data.iloc[50:100,4].values
labels = np.concatenate((setosa_label,versicolor_label))
```

Listing 1. data extraction for iris

The dataset showed clear linear separability, as I visualized in the scatter plot, with perceptron decision boundary as shown in figure1.

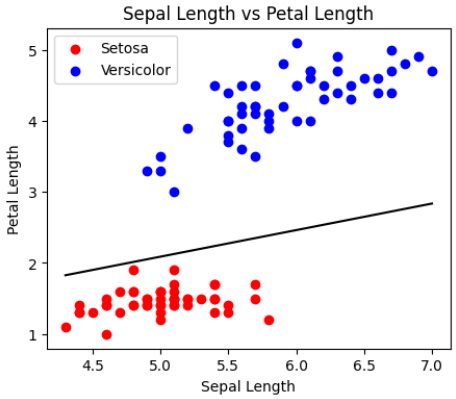

Figure 1. linear separability of the Setosa and Versicolor.

I executed the perceptron algorithm for 8 iterations. Initially, I observed 50 misclassified samples, but by the 5th iteration, misclassifications dropped to zero, indicating convergence. Figure 2shows the change in misclassified samples across the iterations.
I obtained final weight values of:

$$\mathbf{W} = \begin{bmatrix} -2.0 & -3.4 & 9.1 \end{bmatrix} \quad (1)$$

which defined the model's decision boundary.

Task 2 **[60 marks]**

In cases where data is not linearly separable, as shown in Figure 3, simple PLA fails to classify labels correctly due to the impossibility of separating nonlinear data with a straight line. The pocket algorithm addressed this by storing the model with the fewest misclassification errors, for best predicted model.

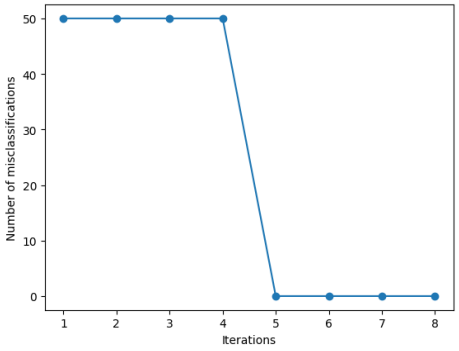

Figure 2. change in misclassified samples across the iterations.

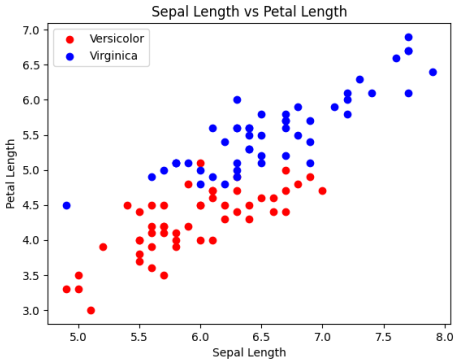

Figure 3. Versicolor and Virginica classes

As shown in Listing 2, the code iteratively compares errors of each predicted model, storing the one with the least error as W_{pocket}.

```
def fit_pocket(n_iter, X, y):
    W = np.zeros(1 + X.shape[1])
    W_pocket = np.copy(W)
    best_error = len(y)
    errors_ = []
    for i in range(n_iter):
        for j, (xi, target) in enumerate(zip(X,y)):
          if predict(W, xi) != target:
            W = W + target * np.append(1, xi)
        errors = 0
        for j, (xi, target) in enumerate(zip(X, y)):
            if predict(W, xi) != target:
                errors += 1
        if errors < best_error:
            W_pocket = np.copy(W)
            best_error = errors
        plot_boundary(W_pocket, X)
```

Listing 2. pocket alogrithm

The performance of the pocket algorithm depends on the number of iterations: more iterations mean more cases considered, leading to a better model. I set it to 100 iterations, revealing three distinct stages.
First, the model begins iterating to find the best prediction, as shown in Figure4

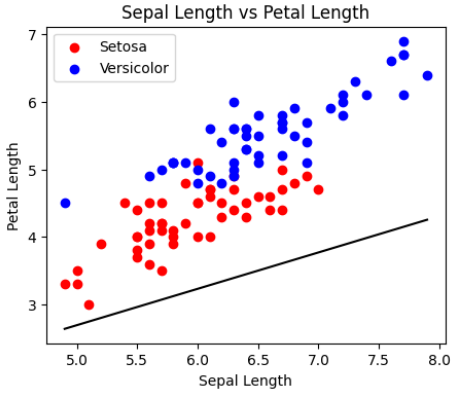

Figure 4. model at 1st iteration

After 70 iterations, it reaches the model with the least error (misclassified error=16), as illustrated in Figure 5.

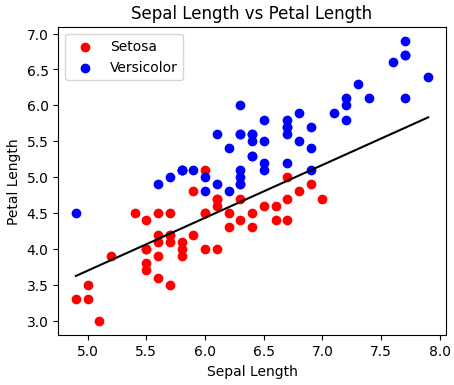

Figure 5. least error model occur at 70th iteration

However, due to the nonlinearly separable data, the model cannot fully converge and eventually diverges. By the 100th iteration, the model ends with an error of 44, as shown in Figure6

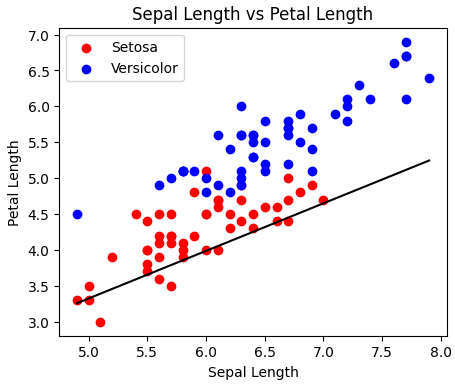

Figure 6. diverged model at 100th iteration

The overall error trend shown in figure 7, which clearly shows the convergence and divergence trend.

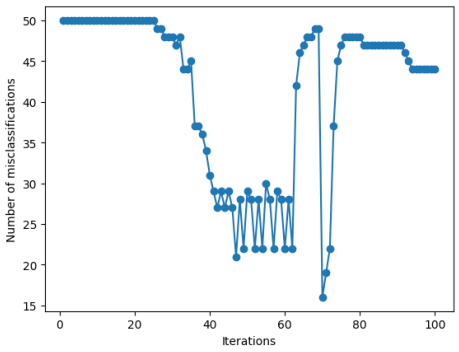

Figure 7. error change along 100th iteration

Overall for the best predicted model I got with weights value of:

$$\mathbf{W} = \begin{bmatrix} -1.0 & -47.2 & 64.1 \end{bmatrix} \tag{2}$$

with misclassified error of 16

2. Linear Regression

Task 1 **[10 marks]**

The following plots show the predicted model of disease progression with respect to BMI) (figure8) and blood pressure (figure9), respectively.

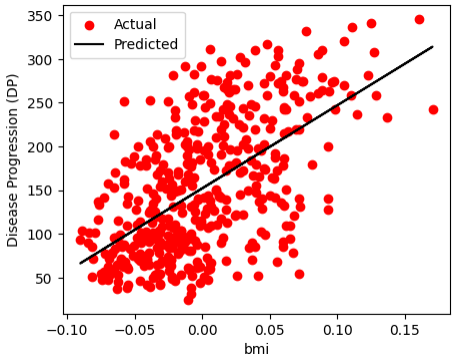

Figure 8. Linear regression: BMI vs disease propagation

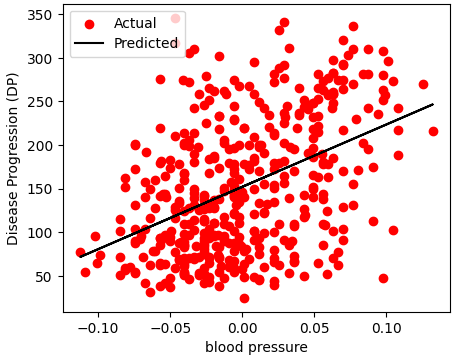

Figure 9. Linear regression: blood pressure vs disease propagation

The corresponding MSE are shown in figure10 and figure11, clearly illustrating rapid convergence with only a few iterations.

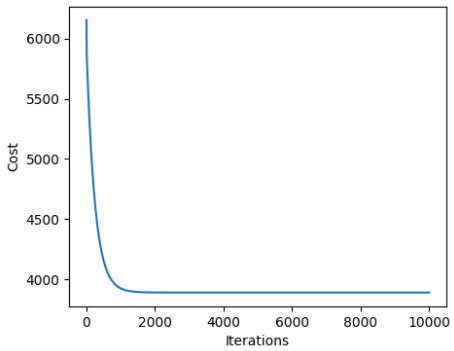

Figure 10. BMI: MSE change with iterations

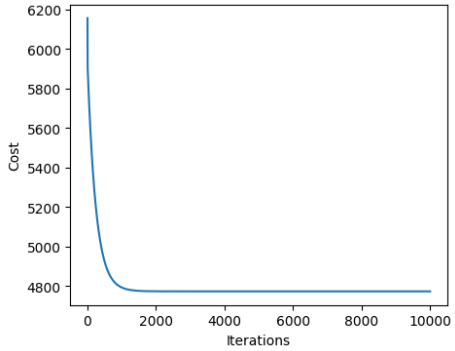

Figure 11. blood pressure: Mse change with iterations

Task 2 **[40 marks]**

```python
def fit_closed_form(X, Y, attribute_x, attribute_y):
    global df
    X = np.column_stack((np.ones(len(X)), X))
    W = np.linalg.pinv(X.T.dot(X)).dot(X.T).dot(Y)

    mse = cost(X, W, Y)
    costs = [mse] * 10

    plot_line(X, W, Y)

    plot_cost(costs)

    df = add_result(df, attribute_x, attribute_y, "
     closed-form", W[0], W[1], mse)
    return W
```

Listing 3. closed form alogrithm

The table 1 compares the Mean Squared Error (MSE), bias, and weight for Gradient Descent and Closed-Form, across three features: Age, BMI, and BP.
Both methods yield the same MSE and weights for each feature.

Figure121314 shows the predicted model founded by Closed form method.

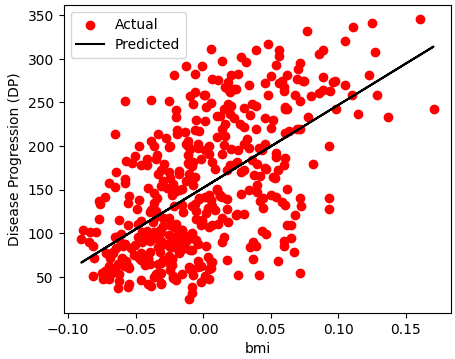

Figure 12. closed-form: BMI

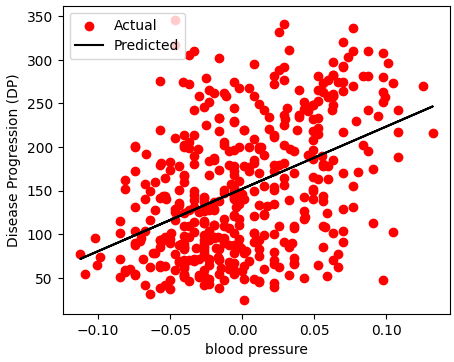

Figure 13. closed-form: blood pressure

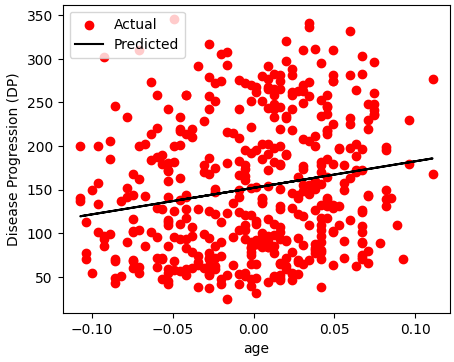

Figure 14. closed-form: age

Task 3 **[15 marks]**

```
def sklearn(X, Y, attribute_x, attribute_y):
    global df
    X_2d = X.values.reshape(-1, 1)

    model = LinearRegression()
    model.fit(X_2d, Y)
    W = np.array([model.intercept_, model.coef_[0]])
    predictions = model.predict(X_2d)

    mse = mean_squared_error(Y, predictions)

    plot_line_sklearn(X_2d, W, Y)

    costs = [mse] * 10
    plot_cost(costs)

    df = add_result(df, attribute_x, attribute_y, "
     sklearn", W[0], W[1], mse)
```

Listing 4. sckit-learn method

```
def sklearn(X, Y, attribute_x, attribute_y):
    global df
    X_2d = X.values.reshape(-1, 1)

    model = LinearRegression()
    model.fit(X_2d, Y)
    W = np.array([model.intercept_, model.coef_[0]])
    predictions = model.predict(X_2d)

    mse = mean_squared_error(Y, predictions)

    plot_line_sklearn(X_2d, W, Y)

    costs = [mse] * 10
    plot_cost(costs)

    df = add_result(df, attribute_x, attribute_y, "
     sklearn", W[0], W[1], mse)
```

Listing 5. sckit-learn method with 10 features

The figure 16 1715 shows the predicted model generated by sckit-learn method.

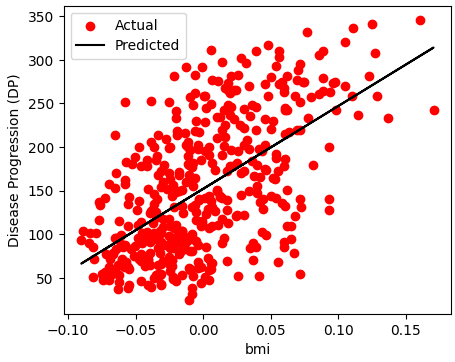

Figure 15. sckit-learn: BMI

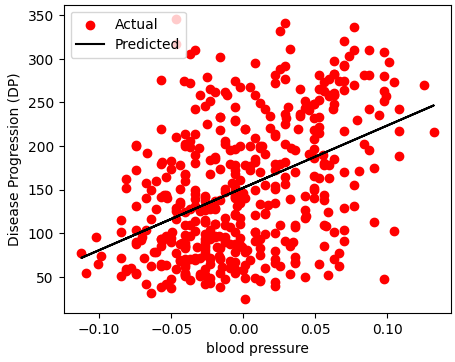

Figure 16. sckit-learn: blood pressure

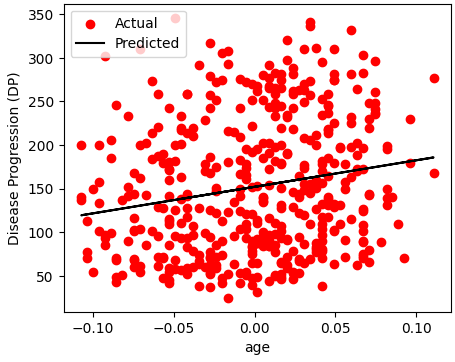

Figure 17. sckit-learn: age

The table 1 compares the MSE, bias, and weight for different regression methods (Gradient Descent, Closed-Form, and Scikit-learn) across three features: Age, BMI, and BP. All methods yield

the same MSE and weight for each feature, indicating consistent performance across methods.

Feature	Method	Cost (MSE)	Bias (W[0])	Weight (W[1])
Age	Gradient Descent	5720.55	152.13	304.18
Age	Closed-Form	5720.55	152.13	304.18
Age	Scikit-learn	5720.55	152.13	304.18
BMI	Gradient Descent	3890.46	152.13	949.44
BMI	Closed-Form	3890.46	152.13	949.44
BMI	Scikit-learn	3890.46	152.13	949.44
BP	Gradient Descent	4774.11	152.13	714.74
BP	Closed-Form	4774.11	152.13	714.74
BP	Scikit-learn	4774.11	152.13	714.74

Table 1. Final Weights and MSE for Different Methods and Features

Task 4 **[10 marks]**

By using Sckit-learn method, I can process 10 features together which end up with. a 10th order weight matrix in 10th dimension:

$$\mathbf{W} = \begin{bmatrix} -10.0098663 \\ -239.81564367 \\ 519.84592005 \\ 324.3846455 \\ -792.17563855 \\ 476.73902101 \\ 101.04326794 \\ 177.06323767 \\ 751.27369956 \\ 67.62669218 \end{bmatrix} \tag{3}$$

With much smaller MSE = 2859.69634758675

The results differ from those obtained using only one feature because this time, the predictor operates in a 10-dimensional space instead of a 2-dimensional one. This higher dimensionality allows the model to capture more complex relationships, leading to different weights predicted model which considered the interrelationship between features.

Task 5 **[25 marks]**

In this implementation, I applied 10-fold validation on dataset to assess the model's generalization performance as shown in list6. The dataset was shuffled and split into 10 equal folds, with each fold being used as a validation set once while the remaining 9 folds served as the training set.

The model trained on each fold was a linear regression model. The MSE for each fold were: Fold 1: MSE = 3024.23 Fold 2: MSE = 3841.19 Fold 3: MSE = 2801.36 Fold 4: MSE = 2166.46 Fold 5: MSE = 2625.90 Fold 6: MSE = 3998.63 Fold 7: MSE = 1631.99 Fold 8: MSE = 2813.44 Fold 9: MSE = 3194.62 Fold 10: MSE = 4060.77

The average MSE across all folds was 3015.86, reflecting the model's general performance, which is a more accurate evaluation.

```
X = diabetes_data.drop(columns=['DP']).values
Y = diabetes_data['DP'].values

k = 10
n_samples = len(X)
fold_size = n_samples // k

indices = np.arange(n_samples)
np.random.shuffle(indices)
X_shuffled = X[indices]
Y_shuffled = Y[indices]
mse_list = []

for i in range(k):
    start = i * fold_size
    end = start + fold_size

    X_val = X_shuffled[start:end]
    Y_val = Y_shuffled[start:end]

    X_train = np.concatenate([X_shuffled[:start],
     X_shuffled[end:]], axis=0)
    Y_train = np.concatenate([Y_shuffled[:start],
     Y_shuffled[end:]], axis=0)

    model = LinearRegression()
    model.fit(X_train, Y_train)

    Y_pred = model.predict(X_val)

    mse = mean_squared_error(Y_val, Y_pred)
    mse_list.append(mse)
    print(f"Fold {i+1}: MSE = {mse}")

average_mse = np.mean(mse_list)
print(f"Average MSE across all folds: {average_mse}")
```

Listing 6. K-fold validation

3. Non-linear Transformation

Task 1 **[40 marks]**

A specific transformation can convert nonlinearly separable data into a linearly separable pattern. Once this transformation is applied, the Perceptron Learning Algorithm (PLA) can be used to obtain the desired linear classifier.

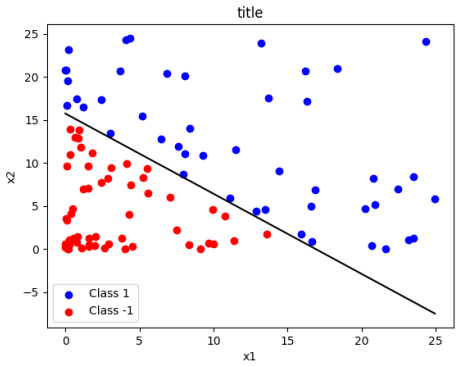

Figure 18. visualization of data with boundary after transformation

The result after 16 iteration achieved zero miss classification error with weight of classifier as follow:

$$\mathbf{W} = \begin{bmatrix} -137.0 & 8.108 & 8.707 \end{bmatrix} \tag{4}$$

Task 2 **[60 marks]**

```
def polynomial_transform(x, d, include_bias=True):
    if include_bias:
        z = np.ones((x.shape[0], 1))
    else:
        z = np.empty((x.shape[0], 0))

    for i in range(1, d + 1):
        z = np.column_stack((z, x**i))
    return z
```

Listing 7. polynomial transformation

The code in Listing 7 demonstrates how to transform an input x, a single-variable input, into a polynomial of order d. By performing training on the data 8 times, using polynomial orders 1, 3, 5, 7, 9, 11, 12, and 15, the results are summarized as follows: shown in figure 19

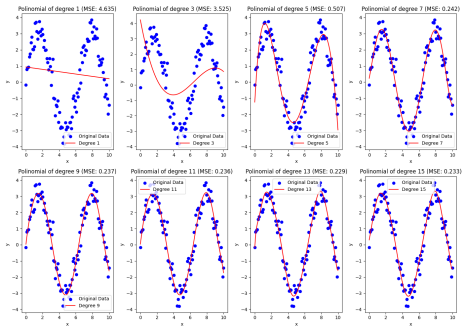

Figure 19. polynomial transformation

When training the data using polynomial models of orders less than 5, the resulting hypotheses show high MSE, with values of 4.64 and 3.52 for orders 1 and 3, respectively. This indicates that the models are overly simplified for such training data's. For polynomial orders of 5 and above, the models better align with the data pattern, achieving MSE values below 0.5. Specifically, with a polynomial of order 15, the MSE decreases approximately 0.23, demonstrating an excellent fit to the training data. However, increasing model complexity to a large order is not always ideal, as it raises the risk of overfitting, which can significantly worse the model's performance on test data. The test result with and without bias are identical

4. Logistic regression

Task 1 **[40 marks]**

To train a model accurately, it's essential to split the dataset into training and testing sets. A larger training set allows the model to fit more effectively to the data, while the reserved testing set is used to evaluate the model's performance, helping to avoid overfitting.

The $train\_test\_split()$ function, shown in Listing 8, ensures a random and unbiased distribution of samples.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y
    , test_size=0.2, random_state=42)
alpha = 0.01
num_iter = 3000
W = fit(alpha, num_iter, X=X_train, y=y_train)
predictions = predict(X_test, W)
pred = predictions.astype('int')
true = y_test.astype('int')
conf_matrix = compute_confusion_matrix(true, pred)
print('Confusion matrix result: ')
print(conf_matrix)

diagonal_sum = conf_matrix.trace()
sum_of_all_elements = conf_matrix.sum()
accuracy = diagonal_sum / sum_of_all_elements
print('accuracy = {:0.2f}%'.format(accuracy*100))
```

Listing 8. assessing model using confusion matrix

The model's accuracy is quantified using a confusion matrix, shown in Equation 5, with an overall accuracy of 100%. The matrix sums to 30 samples, representing 20% of the entire dataset (150 samples), which verifies the correctness of the split and calculated accuracy.

$$\textbf{comfusion matrix\_Setosa} = \begin{bmatrix} 10 & 0 \\ 0 & 20 \end{bmatrix} \tag{5}$$

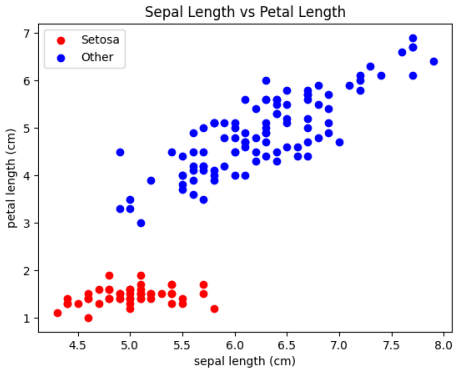

Figure 20. Setosa vs others

The above Figure20 shows the dataset of Setosa vs others.

Task 2 **[20 marks]**

To generate models that classify Iris-Virginica and Iris-Versicolor versus other classes separately, I created two datasets: one for classifying Versicolor versus other classes and another for Virginica versus other classes. These datasets are shown in figure21 and figure22. Still, the training and testing datasets are 80% and 20% respectively.

```
y_versicolor = np.where(labels == 1, 0, 1)
y_virginica = np.where(labels == 2, 0, 1)

X = iris_data.iloc[:,[0,2]]
intercept = np.ones((X.shape[0], 1))
X = np.concatenate((intercept, X), axis=1)

X_versicolor = iris_data[iris_data.target==1]
X_otherthanversicolor = iris_data[iris_data.target!=1]
X_train_versicolor, X_test_versicolor,
    y_train_versicolor, y_test_versicolor =
    train_test_split(X, y_versicolor, test_size=0.2,
    random_state=42)

X_virginica = iris_data[iris_data.target==1]
X_otherthanvirginica = iris_data[iris_data.target!=1]
X_train_virginica, X_test_virginica, y_train_virginica,
    y_test_virginica = train_test_split(X, y_virginica,
     test_size=0.2, random_state=42)
```

Listing 9. changing dataset to virginica and versicolor respectively

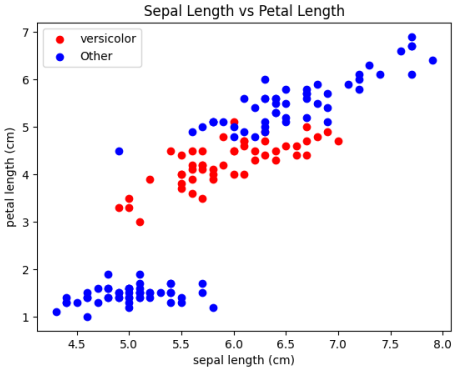

Figure 21. versicolor vs others

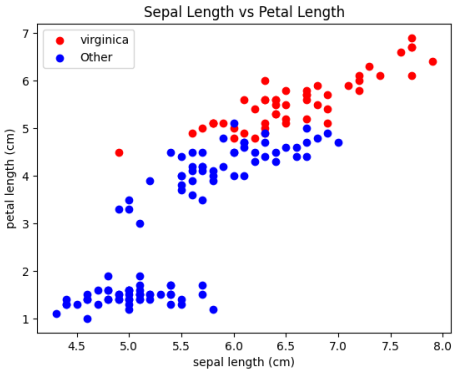

Figure 22. virginica vs others

After applying Logistic regression, the weight for two above sets are shown in equation6 and 7.

$$\textbf{weight versicolor} = \begin{bmatrix} 0.00668389 & 0.43189967 & -0.47953932 \end{bmatrix} \tag{6}$$

$$\textbf{weight virginica} = \begin{bmatrix} 0.72815421 & 1.64637642 & -2.2601223 \end{bmatrix} \tag{7}$$

Setosa's distinct separation in feature space (see Figure 20) allows for high classification accuracy(100%). In contrast, Versicolor's intermediate position leads to greatest overlap with both Setosa and Virginica, making it more challenging for the classifier and thus reducing accuracy(70%) with confusion matrix shown in equation8. Virginica, with less overlap mainly with Versicolor, has relatively distinct attributes, resulting in higher classification accuracy(90%) with confusion matrix shown in equation9.

$$\textbf{comfusion matrix\_versicolor} = \begin{bmatrix} 0 & 9 \\ 0 & 21 \end{bmatrix} \tag{8}$$

$$\textbf{comfusion matrix\_virginica} = \begin{bmatrix} 9 & 2 \\ 1 & 18 \end{bmatrix} \tag{9}$$

Task 3 **[40 marks]**

By using different learning rate : α (0.0001, 0.01, 1, and 100) it is clear to see the convergence process difference between each other. see Listing 10 show the code to track cost in training process.

```
def fit(alpha, num_iter, X, y, initial_tracking_iters
    =100):
    W = np.zeros(X.shape[1])
    loss_list = []

    for i in range(num_iter):
        z = np.dot(X, W)
        Y = sigmoid(z)
        W += alpha * np.dot(X.T, (y - Y)) / y.size
        loss = cost(Y, y)

        if i < initial_tracking_iters or i % 100 == 0:
            loss_list.append(loss)

    return W, loss_list
```

Listing 10. Cost Tracking in Training Function

Below are weight for each learning rate.

- **Learning Rate = 0.0001:**

$$\mathbf{W}_{0.0001} = \begin{bmatrix} 0.0001 & 0.0359 & -0.0412 \end{bmatrix} \tag{10}$$

$$\textbf{Confusion Matrix}_{0.0001} = \begin{bmatrix} 0 & 10 \\ 0 & 20 \end{bmatrix} \tag{11}$$

  Accuracy: 66.67%.

- **Learning Rate = 0.01:**

$$\mathbf{W}_{0.01} = \begin{bmatrix} 0.0270 & 0.4065 & -0.4630 \end{bmatrix} \tag{12}$$

$$\textbf{Confusion Matrix}_{0.01} = \begin{bmatrix} 10 & 0 \\ 0 & 20 \end{bmatrix} \tag{13}$$

  Accuracy: 100%.

- **Learning Rate = 1:**

$$\mathbf{W}_{1} = \begin{bmatrix} 0.7432 & 1.6742 & -2.2962 \end{bmatrix} \tag{14}$$

$$\textbf{Confusion Matrix}_{1} = \begin{bmatrix} 10 & 0 \\ 0 & 20 \end{bmatrix} \tag{15}$$

  Accuracy: 100%.

- **Learning Rate = 100:**

$$\mathbf{W}_{100} = \begin{bmatrix} 96.9317 & 3.3141 & -2.8361 \end{bmatrix} \tag{16}$$

$$\textbf{Confusion Matrix}_{100} = \begin{bmatrix} 10 & 0 \\ 0 & 20 \end{bmatrix} \tag{17}$$

  Accuracy: 100%.

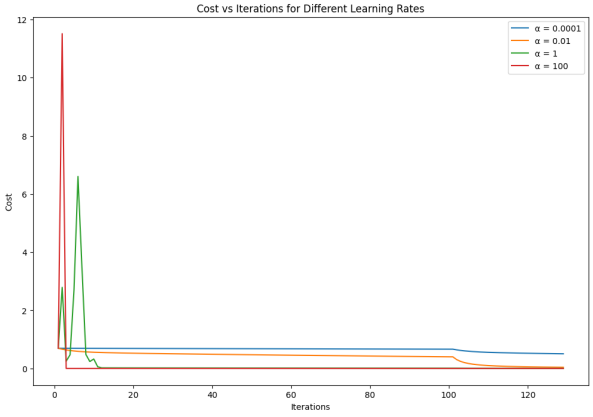

Figure 23. Cost vs. Iterations for Different Learning Rates

The learning rate α influences the gradient descent process by determine different size step. A small α, such as 0.0001, leads to slow convergence, which result in 66.6% accuracy for 120 iterations. but high stability. Larger values, such as $\alpha = 1$ and $\alpha = 100$, enable fast convergence but higher risk of overshoots, make training process unstable.

5. SVM

Task 1 **[10 marks]**

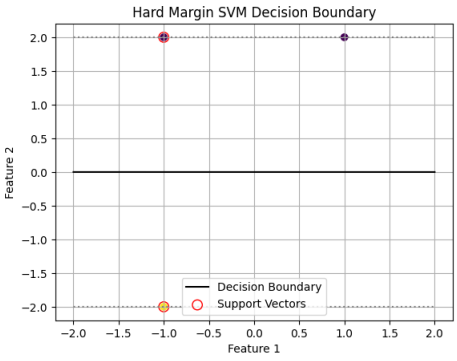

Figure 24. classification using SVM

Both manual calculations and the SVM function identified the same support vectors, weight values, bias, and $alpha$ values. Classifcation result is shown in figure 24. However, the code provided higher accuracy with more decimal places.
Although x_1 lies on the margin, it isn't a support vector since $\alpha_1 = 0$. The dual formulation minimized the set of support vectors, indicating SVM's efficiency in selecting the minimal numbers of necessary support vectors.

Task 2 **[60 marks]**
Code listed below11 are modified version of hard margin SVM which tolerant outliers and provide better fit models.

```
def soft_margin(X, y, C):
    m, n = X.shape

    W = cp.Variable(n)
    b = cp.Variable()
    slack = cp.Variable(m)

    objective = cp.Minimize(0.5 * cp.norm(W) ** 2 + C *
     cp.sum(slack))

    constraints = [y[i] * (X[i] @ W + b) >= 1 - slack[i]
      for i in range(m)]
    constraints += [slack[i] >= 0 for i in range(m)]

    prob = cp.Problem(objective, constraints)
    prob.solve()

    if prob.status != cp.OPTIMAL:
        print("SVM optimization failed. Please check the
      data and constraints.")
        return

    W_optimal = W.value
    b_optimal = b.value
    print("Optimal value of W:", W_optimal)
    print("Optimal value of b:", b_optimal)

    alphas = np.array([constraints[i].dual_value for i
     in range(m)])
    support_vectors = X[np.where(alphas > 1e-4)]
    non_zero_alphas = alphas[np.where(alphas > 1e-4)]
    print("Non-zero alphas:", non_zero_alphas)
    print("Support Vectors:\n", support_vectors)
```

Listing 11. soft margin

Using the soft-margin SVM, I evaluated various C values observing the impact on decision boundaries, support vectors, and margin width.

$$\mathbf{C} = \left[1e-3, 1e-1, 1, 1e2, 1e5\right] \tag{18}$$

For smaller C values (e.g. $1e-3$ and $1e-1$ The margin is wider, allowing more flexibility. Several misclassifications are tolerated, and more points lie within the margin, which results in a greater number of support vectors. As shown in figure 25

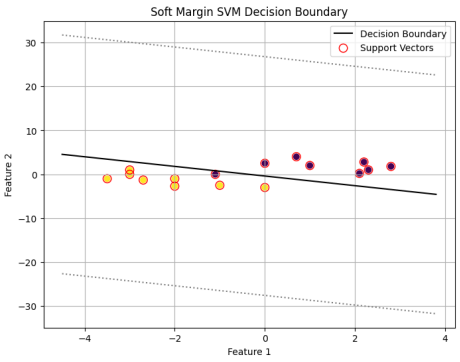

Figure 25. C = 1e-3

As C increases (e.g.1, $1e2$ and $1e5$. The margin becomes narrower, aiming for stricter separation of classes. Fewer support vectors are required, as the model penalizes misclassifications more heavily. as shown figure 26

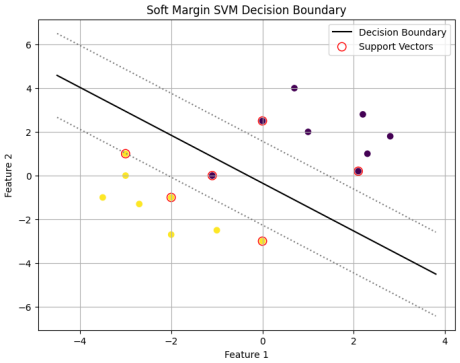

Figure 26. C = 1e-1

At very high C, the margin approaches the behavior of a hard-margin SVM, minimizing misclassifications but at the risk of overfitting. as shown in figure 27

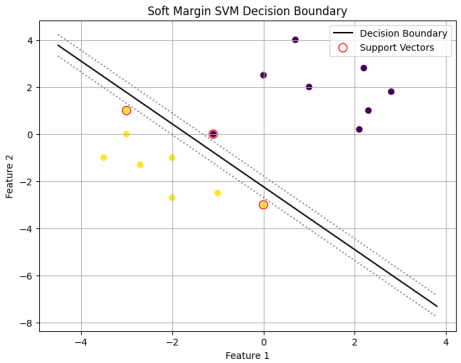

Figure 27. c = 1e5

For this dataset, moderate C values, such as C=1e-1, provide an optimal balance between flexibility and accuracy.

Task 3 **[30 marks]**

The code listed in 12 provide the process of soft SVM throgh sklearn.

```
def sklearn_soft_margin(X, y, C):
    model = svm.SVC(kernel='linear', C=C)
    model.fit(X, y)


    W = model.coef_[0]
    b = model.intercept_[0]
    print(f'C = {C}: W = {W}, b = {b}')

    plt.figure(figsize=(8, 6))
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap='coolwarm',
     edgecolors='k')

    x_vals = np.array([np.min(X[:, 0]) - 1, np.max(X[:,
     0]) + 1])
    slope = -W[0] / W[1]
    intercept = -b / W[1]
    y_vals = slope * x_vals + intercept
    plt.plot(x_vals, y_vals, label='Decision Boundary',
     color='black')

    margin = 1 / np.linalg.norm(W)
    y_down = y_vals - margin
    y_up = y_vals + margin
    plt.plot(x_vals, y_down, linestyle='dotted', color='
     grey')
    plt.plot(x_vals, y_up, linestyle='dotted', color='
     grey')
    plt.scatter(model.support_vectors_[:, 0], model.
     support_vectors_[:, 1], s=100, facecolors='none',
     edgecolors='r', label='Support Vectors')

    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.legend()
    plt.title(f'Soft Margin SVM Decision Boundary (C={C
     })')
    plt.grid()
    plt.show()
C_values = [1e-3, 1e-1, 1, 1e2, 1e5]
for C in C_values:
    sklearn_soft_margin(X, y, C)
```

Listing 12. sklearnsoft margin

The hard margin method are implement by setting c to a large number which in that case, soft margin becomes to a hard margin.

```
def hard_margin_svm(X, y):
    model = svm.SVC(kernel='linear', C=C)
    model.fit(X, y)

    # Extract weights and bias from the trained model
    W = model.coef_[0]
    b = model.intercept_[0]
```

Listing 13. sklearn hard margin

By repeating the C values in task 2, using soft margin learn, I ended up with following graphs. Which is identical to plots in Task2.

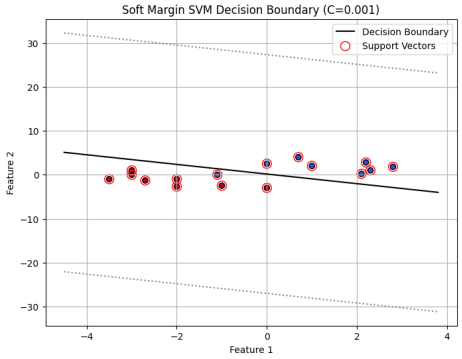

Figure 28. sklearn soft margin c = 0.001

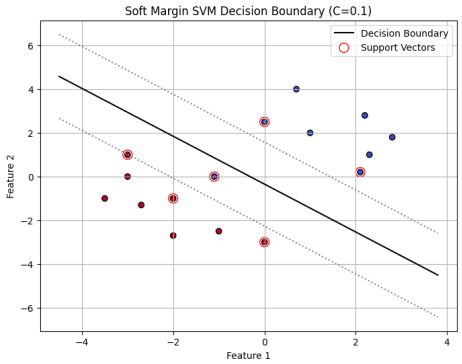

Figure 29. sklearn soft margin c = 0.1

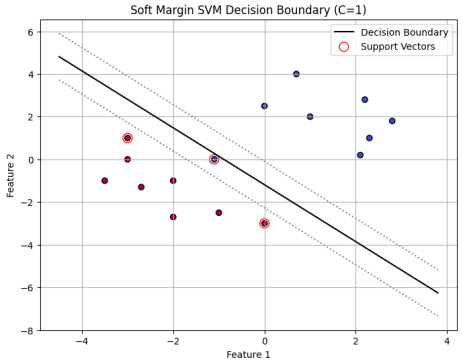

Figure 30. sklearn soft margin c = 1

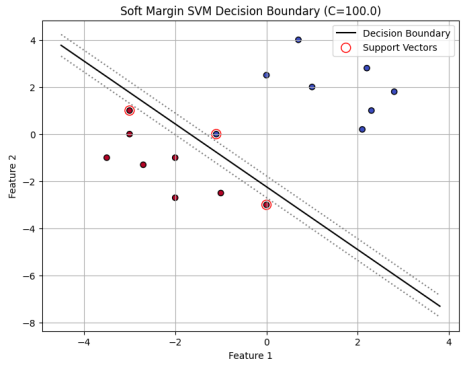

Figure 31. sklearn soft margin c = 100

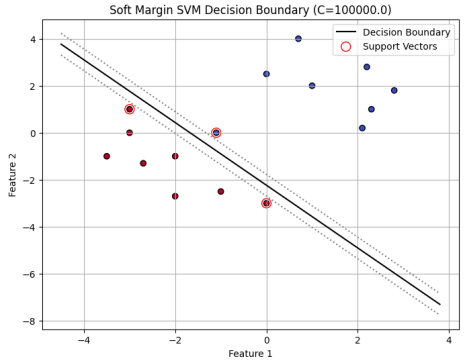

Figure 32. sklearn soft margin c = 100000

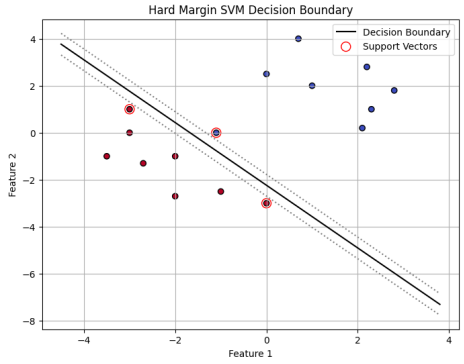

Figure 33. sklearn hard margin

6. k-NN

Task 1 **[60 marks]**

```
def manhattan_distance(point1, point2):
    point1 = np.array(point1)
    point2 = np.array(point2)
    return np.sum(np.abs(point1 - point2))

def minkowski_distance(point1, point2, p):
    point1 = np.array(point1)
    point2 = np.array(point2)
    return np.sum(np.abs(point1 - point2) ** p) ** (1 / p)
```

Listing 14. calculate Manhatton and Minkowski distance

The calculation of Manhatton and Minkowki distance is shown in list14 above.

k	Euclidean (%)	Manhattan (%)	Minkowski(p=3)(%)
1	93.33	93.33	93.33
3	96.67	96.67	93.33
5	96.67	96.67	93.33
7	96.67	96.67	96.67
9	96.67	96.67	96.67
11	96.67	96.67	100.0
13	100.0	96.67	100.0
15	100.0	96.67	100.0
17	96.67	100.0	96.67
19	96.67	100.0	96.67
21	96.67	100.0	96.67
23	96.67	96.67	96.67
25	96.67	96.67	96.67
27	96.67	96.67	96.67
29	96.67	93.33	96.67

Table 2. Accuracy for Different Distance Measures and k Values

For Euclidean distance, the accuracy reaches 100% at k=13 and k=15. And this method maintains high accuracy for most k values and remains stable above 96.67%, suggesting its robustness in distinguishing clusters when spatially separated.
With Manhattan distance, accuracy also reaches 100%, between k = 17 to k = 19 indicating that the classifier may require a slightly larger neighborhood size to achieve optimal performance. Manhattan distance is less sensitive to outliers in certain directions, making it useful when features might exhibit grid-like distributions or structured separations. Still, this method shows the robust performance across large k range.
The Minkowski distance with p=3 effectively emphasizes mid-range distances(close point become closer and far point further due to the power of 3), balancing sensitivity between Euclidean and Manhattan characteristics. Still, this method works nicely across k values.

Task 2 **[40 marks]**

```

from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
for metric in metrics:
    for k in k_values:
        knn = KNeighborsClassifier(n_neighbors=k, metric=metric, p=3 if metric == 'minkowski' else 2)
        knn.fit(X_train, y_train)

        y_pred = knn.predict(X_test)

        accuracy = accuracy_score(y_test, y_pred)
        results[metric].append(accuracy)
```

Listing 15. KNN Task 2

The code list in 15 shows the code to perform KNN in by using sklearn function.

k	Euclidean (%)	Manhattan (%)	Minkowski(p=3)(%)
1	93.33	93.33	93.33
3	96.67	96.67	93.33
5	96.67	96.67	93.33
7	96.67	96.67	96.67
9	96.67	96.67	96.67
11	96.67	96.67	100.00
13	100.00	96.67	100.00
15	100.00	96.67	100.00
17	96.67	100.00	96.67
19	96.67	100.00	96.67
21	96.67	100.00	96.67
23	96.67	96.67	96.67
25	96.67	96.67	96.67
27	96.67	96.67	96.67
29	96.67	93.33	96.67

Table 3. Accuracy for Euclidean, Manhattan, and Minkowski (p=3) Distances

The table 3 presents the accuracy results obtained by performing KNN using sklearn. These results are compared with those in2 which includes the accuracy results obtained by performing KNN in Task 1. The comparison shows no differences between the two sets of results.