7/4/25

Lab 4

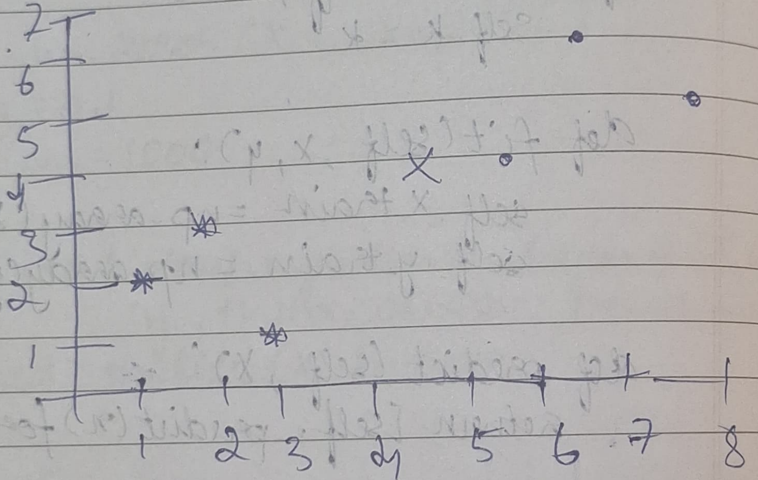
KNN

```
import numpy as np
import matplot.pyplot as plt
from collections import Counter

def ed (x1, x2):
    return np.sqrt(np.sum((x1 - x2)** 2))

class knn:
    def __init__(self, k=3):
        self.k = k

    def fit(self, x, y):
        self.x train = np.array(x)
        self.y train = np.array(y)

    def predict (self, x):
        return [self._predict(x) for x in x]

    def _predict (self, x):
        distances = [euclidean_distance (x, x_train)
                      for x train in self.x_train]
        k_indices = np.argsort(distances)[:self.k]
        k_nearest_labels = [self.y_train[i] for i in
                      k_indices]
        most_common = counter(k_nearest_labels).
                       most_common(1)
        return most_common[0][0]

    def score (self, x, y):
        predictions = self.predict(x)
        return np.mean( predictions == y)

x_train = np.array([[1,2],[2,3],[3,1],
          [6,5],[7,7],[8,6]])
```

```
y_train = np.array([0, 0, 0, 1, 1, 1])
x_test = np.array([[5, 5]])

knn = knn(k=3)
knn.fit(x_train, y_train)
prediction = knn.predict(x_test)
```

output
KNN classification Predicted class: 1

SVM.



* -> class 0
· -> Class 1
x -> Test point

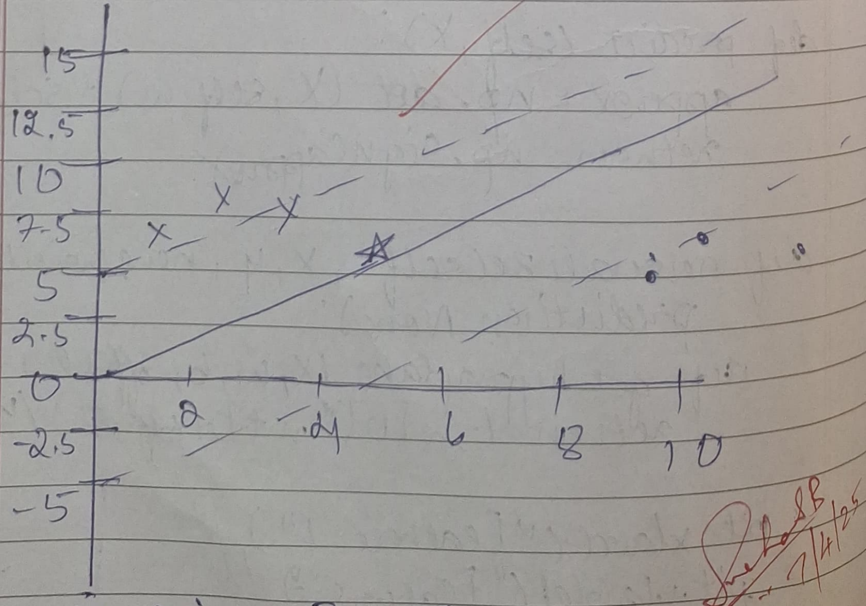
SVM

```
import numpy as np
import matplotlib.pyplot as plt

class SVM:
    def __init__(self, learning_rate=0.001,
                 lambda_param=0.01, n_iters=1000)
        self.lr = learning_rate
        self.lambda_param = lambda_param
```

```
self.n_iters = n_iters
self.w = None
self.b = None

def fit(self, X, y):
    y = np.where(y <= 0, -1, 1)
    n_samples, n_features = X.shape
    self.w = np.zeros(n_features)
    self.b = 0

    for _ in range(self.n_iters):
        for idx, x_i in enumerate(X):
            condition = y[idx] * (np.dot(x_i,
            self.w) + self.b) >= 1
        if condition:
            self.w -= self.lr * (2 * self.lambda_
            param * self.w)
        else:
            self.w -= self.la * (2 * self.lambda_
            param * self.w - np.dot(x_i,
            y[idx]))
            self.b += self.lr * y[idx]

def predict(self, X):
    approx = np.dot(X, self.w) + self.b
    return np.sign(approx

def visualize(self, X, y, new_point = None,
    prediction = None):
    def get_hyperplane(x, w, b, offset):
        return (-w[0] * x + b + offset) / w[1]

    plt.xlabel("Feature 1")
    plt.ylabel("Feature 2")
```

```python
if __name__ == "__main__":

    X = np.array([
    [1, 7], [2, 8], [3, 8], [8, 1], [9, 8],
    [10, 2]])

    y = np.array([0, 0, 0, 1, 1, 1])

    new_point = np.array([[5, 5]])

    svm = SVM()
    svm.fit(X, y)
    prediction = svm.predict(new_point)[0]

    svm.visualize(X, y, new_point=new_point,
                  prediction=prediction)

    print(f"New point {new_point[0]} classified as {'Class 1' if prediction else 'Class 0'}")
```

output
New Point - Class 0



— Decision Boundary
--- Margins.