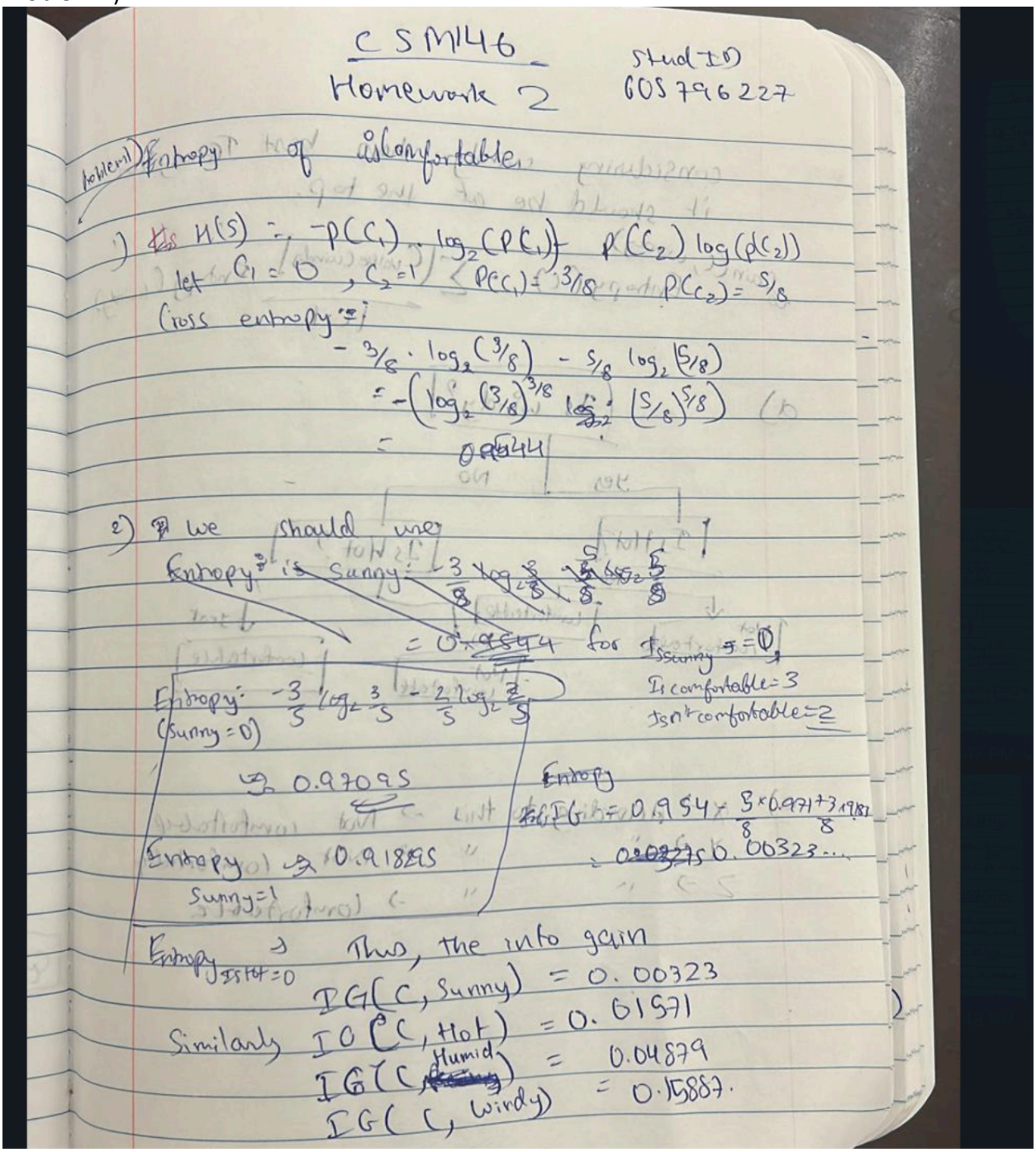
CSM146 Homework 2:

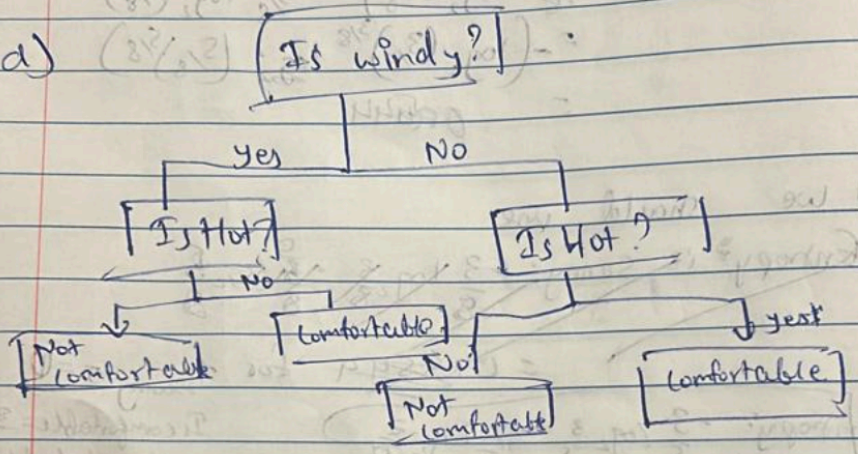
Problem 1)



considering windy has best IGain, thus it should be at the top.

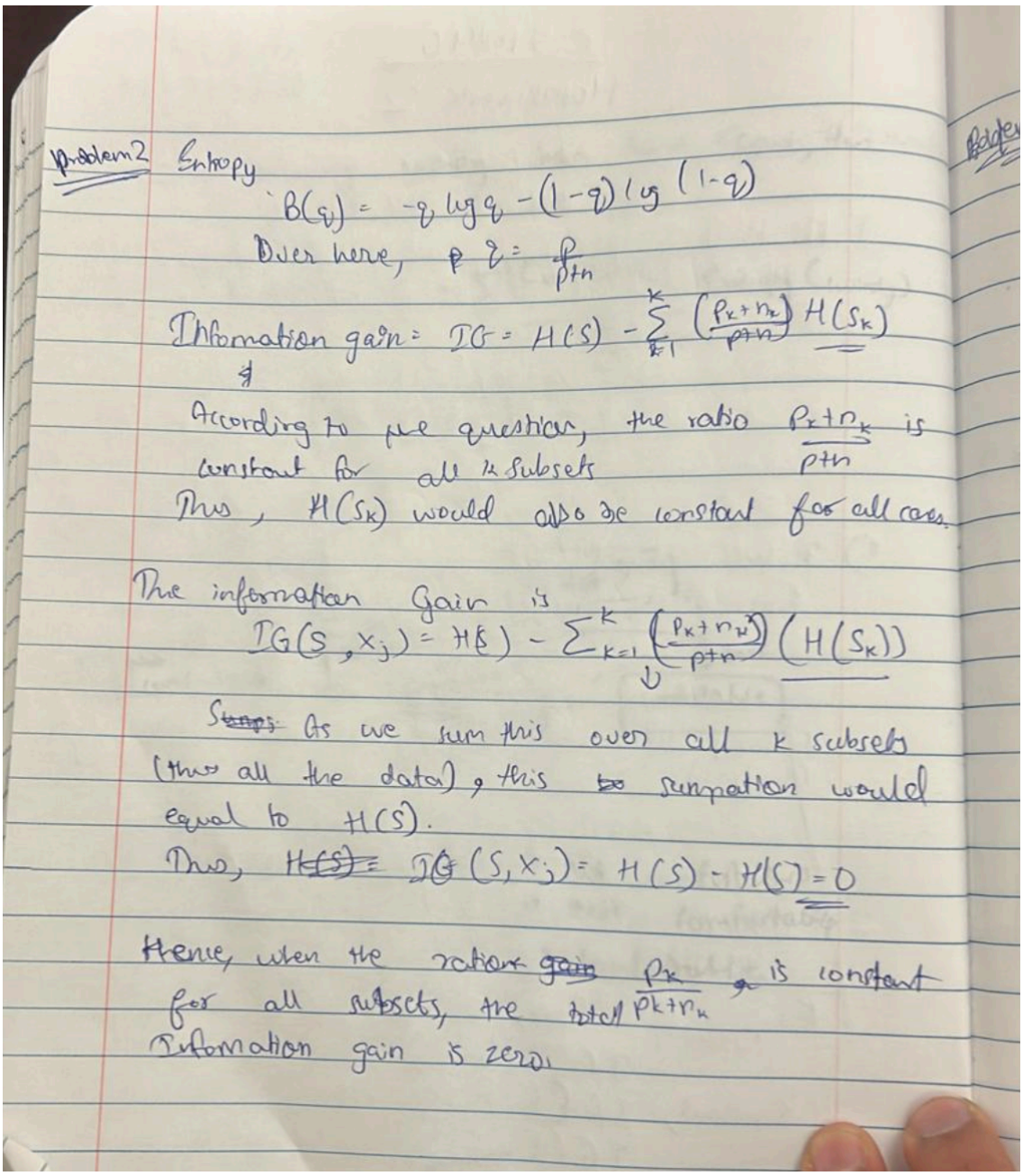
$$Gain(C, windy) = Entropy(C) - \sum \frac{|C_{value(windy)}|}{|C|} Entropy(C_{(windy)})$$

d)




e) X → According to this → Not comfortable
Y → " " → Not comfortable
Z → " " → comfortable.

Problem 2:



Problem 2 Entropy

$$B(q) = -q \lg q - (1-q) \lg (1-q)$$

Over here, $q = \frac{p}{p+n}$

Information gain = $IG = H(S) - \sum_{k=1}^{k} \left(\frac{p_k + n_k}{p+n}\right) H(S_k)$

According to the question, the ratio $\frac{p_k + n_k}{p+n}$ is constant for all k subsets

Thus, $H(S_k)$ would also be constant for all cases.

The information Gain is

$$IG(S, X_j) = H(S) - \sum_{k=1}^{k} \left(\frac{p_k + n_k}{p+n}\right) (H(S_k))$$

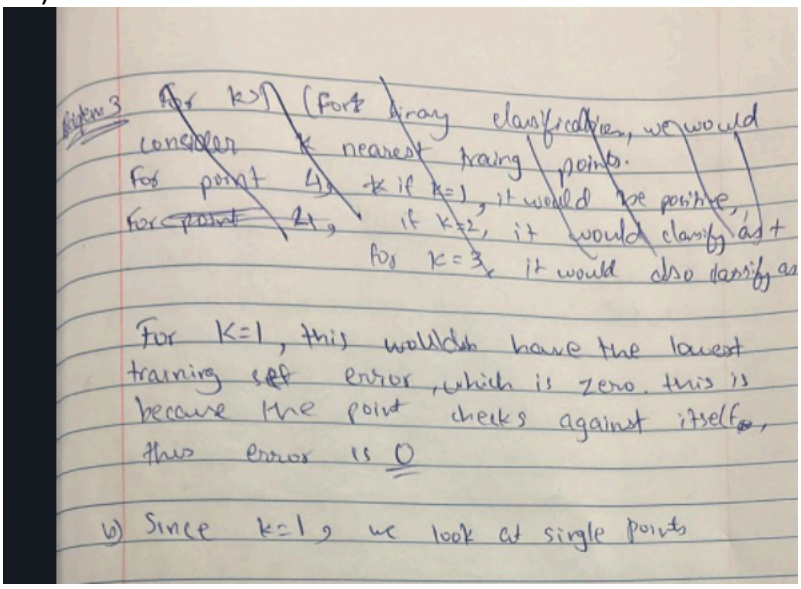
As we sum this over all k subsets (thus all the data), this summation would equal to $H(S)$.

Thus, $IG(S, X_j) = H(S) - H(S) = 0$

Hence, when the ratio $\frac{p_k}{p_k + n_k}$ is constant for all subsets, the total information gain is zero.

Problem 3)

a)




b) For this, the errors can be calculated as:
For each test point: [
(1, 5) closest neighbor is -1, thus it is classified correctly
, (4, 7), closest neighbor is -1, thus it is classified incorrectly
(6, 2), closest neighbor is -1, thus it is classified incorrectly
(9, 5), closest neighbor is 1, thus it is classified correctly
(5, 5)] closest neighbor is 1, thus it is classified incorrectly

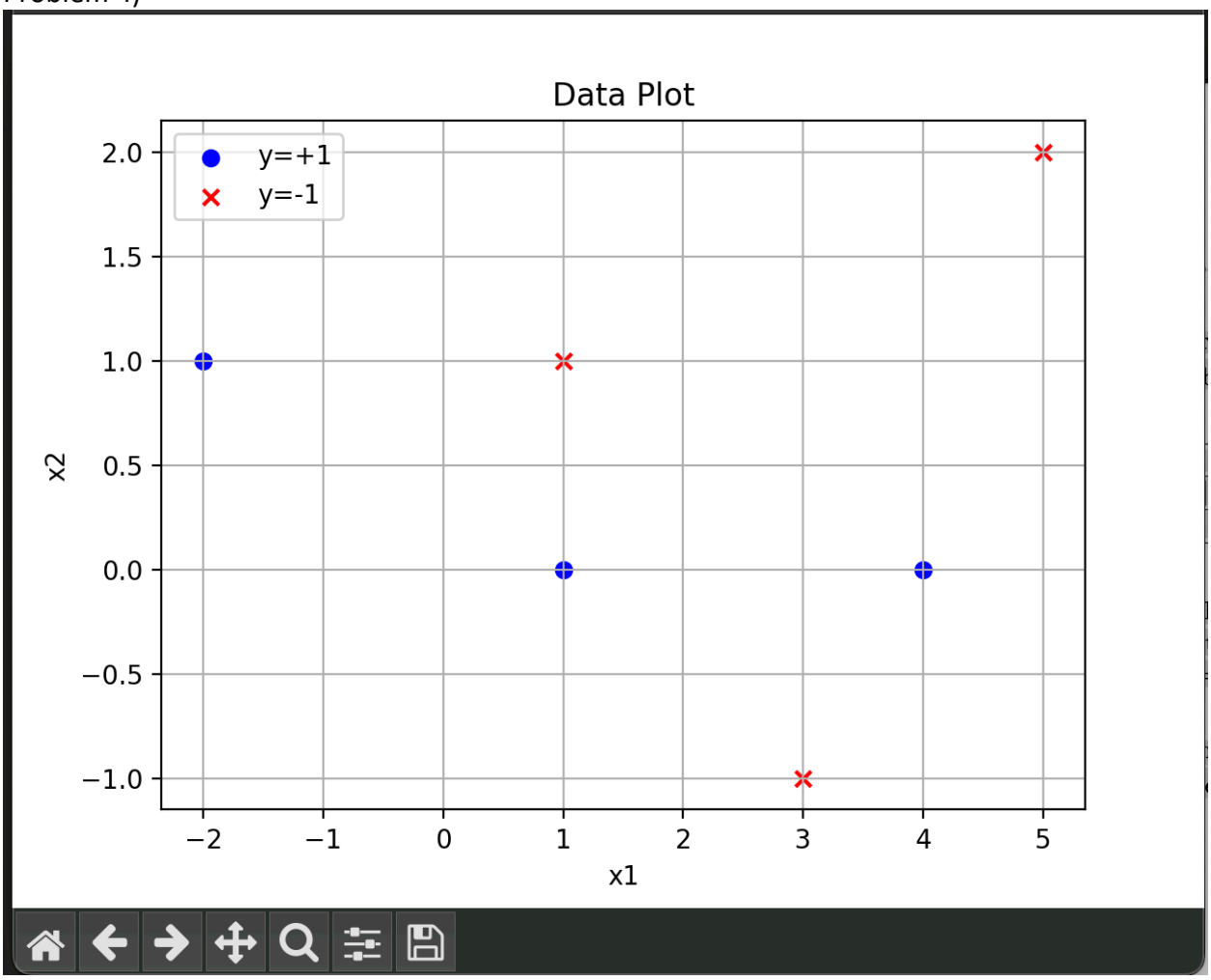
Hence the accuracy of this model would be 2/5 which is 0.4

c)

```
(base) krishpatel@wifi-131-179-23-139 desktop % python3 csm146_hw2.py
point in consideration[1, 5]
Predicted value:  1
Actual:  -1
point in consideration[4, 7]
Predicted value:  1
Actual:  1
point in consideration[6, 2]
Predicted value:  -1
Actual:  1
point in consideration[9, 5]
Predicted value:  1
Actual:  1
point in consideration[5, 5]
Predicted value:  -1
Actual:  -1
```

According to this, values 1 and 3 are incorrectly predicted, thus the accuracy would be 3/5 which is 0.6.

Problem 4)




X1 represents the first parameter while X2 represents the second parameter

b) as we can see from this, no the data is not linearly separable, and a singular line can't be drawn to classify and separate two boundary points. Thus we would need a more complex model.

c)

Problem 4

c) $\theta_{new} = \theta_{old} + \alpha(y^{(i)} - y_{pred}) \times x_i$

for the first case, $\theta = x^{(1)} = [1, 4, 0]$

- $x^{(1)} \Rightarrow$ y-pred $= [1,4,0] \cdot [1,4,0] = 17$ which is $> 0$, thus, $y^{(1)}$ is predicted is 1. hence, no need for update

- $x^{(2)} = [1,1,1] \cdot [1,4,0] = 5$ which gives y-pred $= 1$, hence incorrect. update is

$\theta - 0.1 \times 2 [1,1,1] = [0.8, 3.8, -0.2)$.

- $x^{(3)} = [1,2,1]$ which gives a value of $-7$. which is incorrect ie $y^{(3)} = 1$

Thus, updated weights are $[1, 3.4, 0.0]$

$x^{(4)} = [1,1,0]$. prediction is $1 + 3.4 = 4.4$.

hence correct as y-pred $= 1$

no update

$x^{(5)} = [1,5,2] \cdot [1, 3.4, 0.0] = 18$.

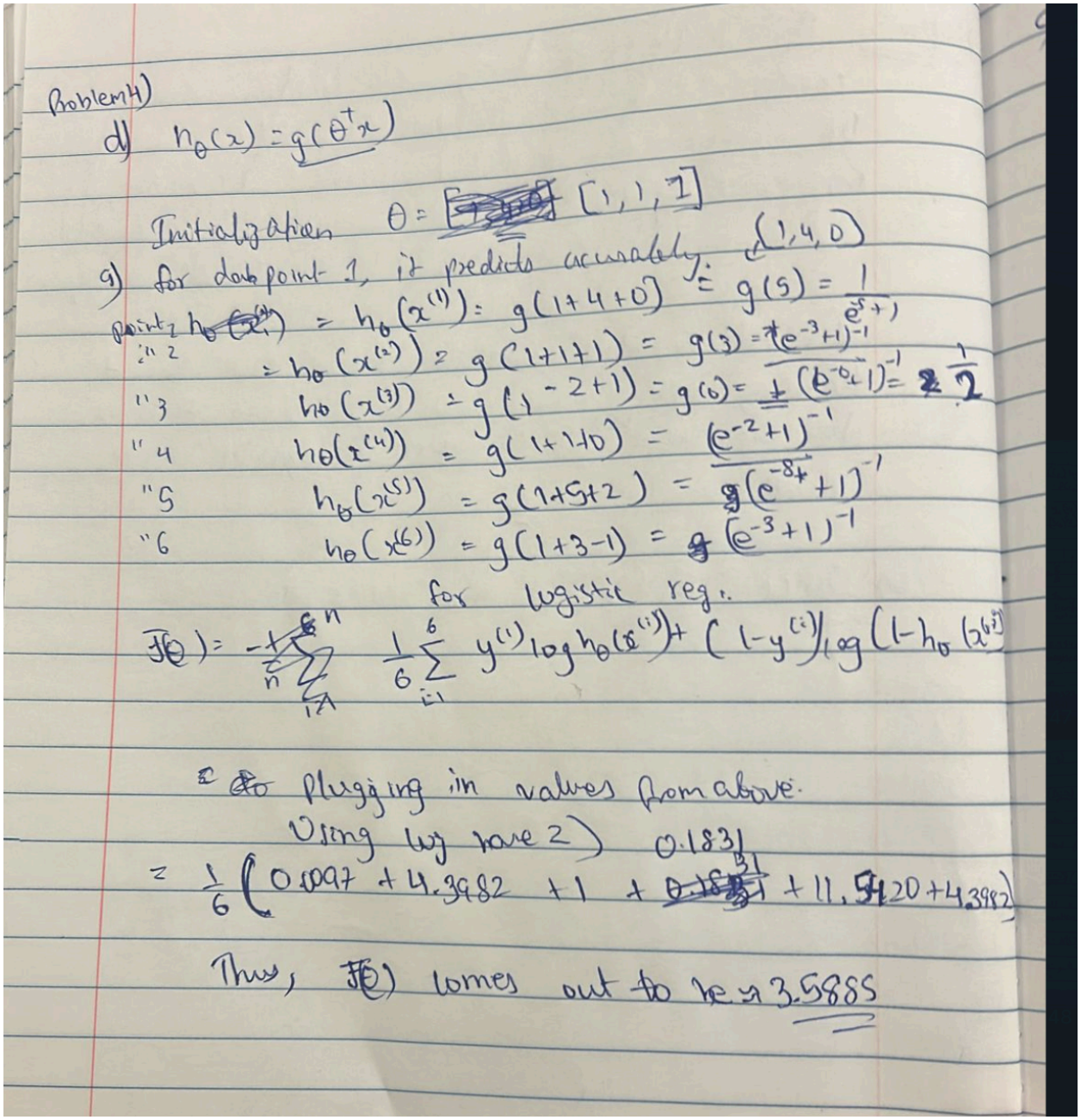
Incorrect. Thus updated $\theta = [0.8, 2.4, -0.4)$

$x^{(6)} = [1, 3, -1]$ y-pred $= 8.4$

which give y-pred $= 1$, which is incorrect

$\theta = \theta - (0.1 \times 2 \times [1, 3, -1]) = [0, 1.8, -0.2)$

d)



Problem 4)

d) $h_\theta(x) = g(\theta^T x)$

Initialization $\theta = [1, 1, 1]$

a) for data point 1, it predicts accurately. $\therefore (1, 4, 0)$

point 2 $h_\theta(x^{(2)}) = h_\theta(x^{(1)}) = g(1+4+0) = g(5) = \frac{1}{e^{-5}+1}$

" 2 $= h_\theta(x^{(2)}) = g(1+1+1) = g(3) = (e^{-3}+1)^{-1}$

" 3 $h_\theta(x^{(3)}) = g(1-2+1) = g(0) = (e^{-0}+1)^{-1} = \frac{1}{2}$

" 4 $h_\theta(x^{(4)}) = g(1+1+0) = (e^{-2}+1)^{-1}$

" 5 $h_\theta(x^{(5)}) = g(1+5+2) = g(e^{-8}+1)^{-1}$

" 6 $h_\theta(x^{(6)}) = g(1+3-1) = g(e^{-3}+1)^{-1}$

for logistic reg.:

$$J(\theta) = -\frac{1}{n}\sum_{i=1}^{n} \frac{1}{6}\sum_{i=1}^{6} y^{(i)} \log h_\theta(x^{(i)}) + (1-y^{(i)}) \log(1-h_\theta(x^{(i)}))$$

= Plugging in values from above.

Using (a) we have 2) 0.1831

$$= \frac{1}{6}(0.0097 + 4.3982 + 1 + 0.1831 + 11.5420 + 4.3982)$$

Thus, $J(\theta)$ comes out to be $\approx 3.5885$

Q5)

a)


b)

```python
### ========== TODO : START ========== ###
# part b: evaluate training error of Random classifier
randomclassifier = RandomClassifier()
randomclassifier.fit(X, y)
y_pred = randomclassifier.predict(X)
from sklearn.metrics import accuracy_score
# 3. Calculate the training error
error = 1 - accuracy_score(y, y_pred)
print(f"Training error of Random Classifier: " + str(error))
### ========== TODO : END ========== ###
```

✓ 0.0s

```
Training error of Random Classifier: 0.374
```

```
    self.probabilities_ = None

def fit(self, X, y) :
    """
    Build a random classifier from the training set (X, y).

    Parameters
    ----------
        X    -- numpy array of shape (n,d), samples
        y    -- numpy array of shape (n,), target classes

    Returns
    -------
        self -- an instance of self
    """

    ### ========== TODO : START ========== ###
    # part b: set self.probabilities_ according to the training set
    unique, counts = np.unique(y, return_counts= True)
    total = len(y)
    probability = counts/total
    self.probabilities_ = dict(zip(unique, probability))

    ### ========== TODO : END ========== ###

    return self

def predict(self, X, seed=1234) :
    """
    Predict class values.

    Parameters
    ----------
        X    -- numpy array of shape (n,d), samples
        seed -- integer, random seed

    Returns
    -------
        y    -- numpy array of shape (n,), predicted classes
    """
    if self.probabilities_ is None :
        raise Exception("Classifier not initialized. Perform a fit first.")
    np.random.seed(seed)

    ### ========== TODO : START ========== ###
    classes = list(self.probabilities_.keys())
    probabilities = list(self.probabilities_.values())
    y = np.random.choice(classes, size = len(X), p = probabilities)

    # part b: predict the class for each test example
    # hint: use np.random.choice (be careful of the parameters)
    #pass

    ### ========== TODO : END ========== ###

    return y
```

c)

```
### ========== TODO : START ========== ###
# part c: evaluate training error of Decision Tree classifier

dt_classifier = DecisionTreeClassifier(random_state=0)
dt_classifier.fit(X, y)
y_pred = dt_classifier.predict(X)
accuracy = accuracy_score(y, y_pred)
training_error = 1 - accuracy
print(f"Training Error of Decision Tree classifier: {training_error}")

### ========== TODO : END ========== ###
```

✓ 0.0s

```
Training Error of Decision Tree classifier: 0.0
```

d)

```
### ========== TODO : START ========== ###
# part d: evaluate training error of k-Nearest Neighbors classifier
# use k = 3, 5, 7 for n_neighbors
k_values = [3, 5, 7]
for k in k_values:
    knn_classifier = KNeighborsClassifier(n_neighbors=k)
    knn_classifier.fit(X, y)
    y_pred = knn_classifier.predict(X)
    accuracy = accuracy_score(y, y_pred)
    training_error = 1 - accuracy
    print(f"Training Error of k-NN classifier (k={k}):" + str(training_error))
### ========== TODO : END ========== ###
```

✓ 0.1s

```
Training Error of k-NN classifier (k=3):0.15300000000000002
Training Error of k-NN classifier (k=5):0.19499999999999995
Training Error of k-NN classifier (k=7):0.21299999999999997
```

e)

```
### ========== TODO : START ========== ###
# part e: evaluate training error of Logistic Regression
# use lambda_ = 0.1, 1, 10 for n_neighbors
lambda_values = [0.1, 1, 10]
# Note: Make sure you initialize your classifier with the appropriate parameters: random_state=0 and max_iter=1000, using the default solver is fine.
C_values = [1.0 / lambda_ for lambda_ in lambda_values]

for C in C_values:
    logisticreg_classifier = LogisticRegression(C=C, random_state=0, max_iter=1000)
    logisticreg_classifier.fit(X, y)
    y_pred = logisticreg_classifier.predict(X)
    accuracy = accuracy_score(y, y_pred)
    training_error = 1 - accuracy
    print(f"Training Error of Logistic Regression (lambda_={1.0/C}): " + str(training_error))



### ========== TODO : END ========== ###
```

✓ 0.5s

```
Training Error of Logistic Regression (lambda_=0.1): 0.20799999999999996
Training Error of Logistic Regression (lambda_=1.0): 0.20799999999999996
Training Error of Logistic Regression (lambda_=10.0): 0.21999999999999997
```

f)

```python
from sklearn.metrics import f1_score

def evaluate_classifier(clf, name, X, y):
    avg_train_error, avg_val_error, avg_f1_score = error(clf, X, y)
    print(f"{name} - Avg Training Error: {avg_train_error}, Avg Validation Error: {avg_val_error}, Avg F1 Score: {avg_f1_score}")
print('Investigating various classifiers...')

evaluate_classifier(DecisionTreeClassifier, "Decision Tree Classifier", X, y)
evaluate_classifier(KNeighborsClassifier, "K-Nearest Neighbors Classifier (k=5)", X, y)
evaluate_classifier(LogisticRegression, "Logistic Regression (λ=1)", X, y)
```

✓ 11.1s

```
Investigating various classifiers...
Decision Tree Classifier - Avg Training Error: 0.0, Avg Validation Error: 0.20133333333333334, Avg F1 Score: 0.7986666666666666
K-Nearest Neighbors Classifier (k=5) - Avg Training Error: 0.19977647058823536, Avg Validation Error: 0.2544, Avg F1 Score: 0.7455999999999999
Logistic Regression (λ=1) - Avg Training Error: 0.20741176470588232, Avg Validation Error: 0.21286666666666665, Avg F1 Score: 0.7871333333333332
```

```python
def error(clf, X, y, ntrials=100, test_size=0.15):
    """
    Computes the classifier error over a random split of the data,
    averaged over ntrials runs.
    """
    sss = StratifiedShuffleSplit(n_splits=ntrials, test_size=test_size, random_state=0)

    train_errors = []
    val_errors = []
    f1_scores = []

    for train_i, test_i in sss.split(X, y):
        X_train, X_test = X[train_i], X[test_i]
        y_train, y_test = y[train_i], y[test_i]
        if clf == DecisionTreeClassifier:
            model = clf(criterion='entropy')
        elif clf == KNeighborsClassifier:
            model = clf(n_neighbors=5)
        elif clf == LogisticRegression:
            model = clf(C=1.0, random_state=0, max_iter=1000)
        else:
            model = clf()

        model.fit(X_train, y_train)
        y_predicted_train = model.predict(X_train)
        train_errors.append(np.mean(y_train != y_predicted_train))

        y_predicted_test = model.predict(X_test)
        val_errors.append(np.mean(y_test != y_predicted_test))

        f1_scores.append(f1_score(y_test, y_predicted_test, average="micro"))

    avg_train_error = np.mean(train_errors)
    avg_val_error = np.mean(val_errors)
    avg_f1_score = np.mean(f1_scores)

    return avg_train_error, avg_val_error, avg_f1_score
```

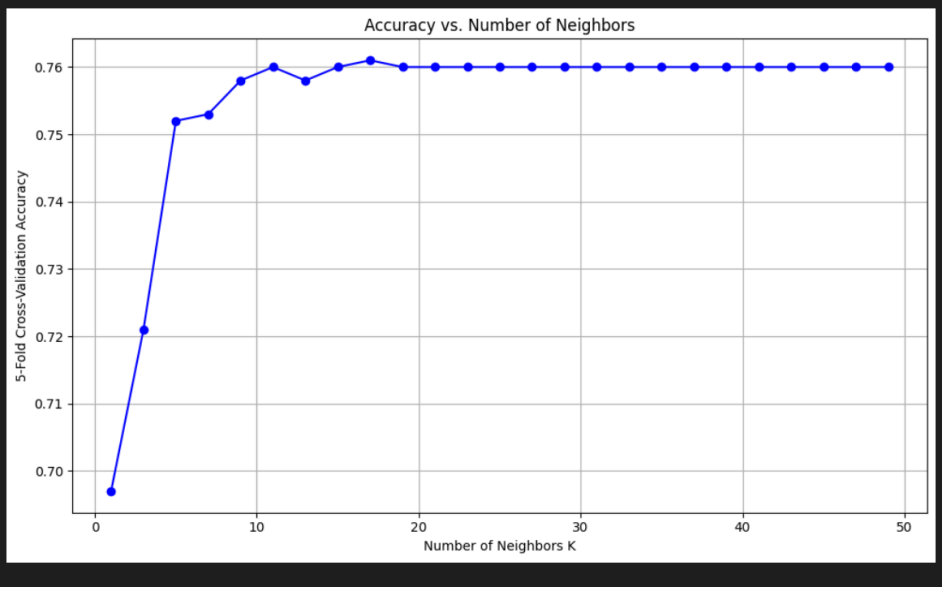
✓ 0.0s

g)

```
from sklearn.model_selection import cross_val_score
print('Finding the best k...')
cv_scores = []
k_values = list(range(1, 51, 2))
for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k)
    scores = cross_val_score(knn, X, y, cv=5, scoring='accuracy')
    cv_scores.append(scores.mean())
best_k = k_values[cv_scores.index(max(cv_scores))]
best_score = max(cv_scores)
print(f"The best value of k is {best_k} with a score of {best_score:.4f}")

plt.figure(figsize=(10, 6))
plt.plot(k_values, cv_scores, marker='o', linestyle='-', color='b')
plt.xlabel('Number of Neighbors K')
plt.ylabel('5-Fold Cross-Validation Accuracy')
plt.title('Accuracy vs. Number of Neighbors')
plt.grid(True)
plt.tight_layout()
plt.show()
```

✓ 1.0s

```
Finding the best k...
The best value of k is 17 with a score of 0.7610
```



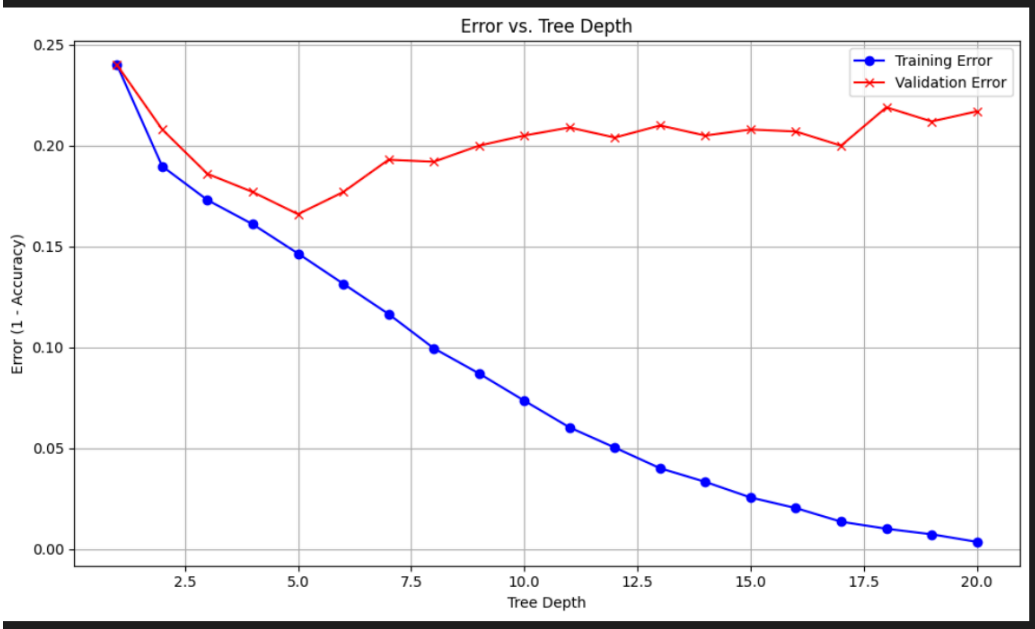

The accuracy of the KNeighborsClassifier initially increases sharply as the number of neighbors (k) increases, reaching a peak at k=17. This suggests that a small k might be too sensitive to noise in the data. After k=17, the accuracy flattens and remains relatively constant for larger values of k, indicating that increasing the number of neighbors beyond this point doesn't provide significant benefits in terms of accuracy for this dataset.

h)

```
# part h: investigate decision tree classifier with various depths
print('Investigating depths...')


depths = list(range(1, 21))
train_errors = []
val_errors = []
for depth in depths:
    tree = DecisionTreeClassifier(max_depth=depth, criterion='entropy')
    scores = cross_validate(tree, X, y, cv=5, return_train_score=True, scoring='accuracy')
    train_error = 1 - scores['train_score'].mean()
    val_error = 1 - scores['test_score'].mean()
    train_errors.append(train_error)
    val_errors.append(val_error)
plt.figure(figsize=(10, 6))
plt.plot(depths, train_errors, marker='o', linestyle='-', color='b', label='Training Error')
plt.plot(depths, val_errors, marker='x', linestyle='-', color='r', label='Validation Error')
plt.xlabel('Tree Depth')
plt.ylabel('Error (1 - Accuracy)')
plt.title('Error vs. Tree Depth')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()



best_depth = depths[val_errors.index(min(val_errors))]   #best depth
print(f"The best depth is {best_depth} with a validation error of {min(val_errors):.4f}")

### ========== TODO : END ========== ###
```

✓ 0.6s

```
nvestigating depths...
```




```
he best depth is 5 with a validation error of 0.1660
```

As the tree depth increases, the training error consistently decreases, indicating that the decision tree becomes more fitted to the training data with increasing depth.
However, the validation error initially decreases with tree depth but starts to flatten out and even slightly increases after a depth of around 5, suggesting that deeper trees might overfit to the training data without significantly improving generalization to new data.
Based on the plot, the optimal depth for the decision tree on this dataset is 5, where the validation error is at its lowest (approximately 0.1660). Beyond this depth, it indicates potential overfitting.