HW1 Decision Tree, Linear Regression

In [ ]:

```
import pandas as pd
import statistics
import numpy as np
from scipy.stats import entropy, zscore
```

In [ ]:

```
trainingHouseDf = pd.read_csv(r"housing_train.txt", header=None, delim_whitespace=True)
testingHouseDf = pd.read_csv(r"housing_test.txt", header=None, delim_whitespace=True)

spambaseDf = pd.read_csv(r"spambase.data", header=None)
```

In [ ]:

```
print(trainingHouseDf.head())
print ("======")
print(spambaseDf.head())
```

```
         0     1     2  3      4      5     6       7  8      9    10  \
0  0.00632  18.0  2.31  0  0.538  6.575  65.2  4.0900  1  296.0  15.3
1  0.02731   0.0  7.07  0  0.469  6.421  78.9  4.9671  2  242.0  17.8
2  0.02729   0.0  7.07  0  0.469  7.185  61.1  4.9671  2  242.0  17.8
3  0.03237   0.0  2.18  0  0.458  6.998  45.8  6.0622  3  222.0  18.7
4  0.06905   0.0  2.18  0  0.458  7.147  54.2  6.0622  3  222.0  18.7

       11    12    13
0  396.90  4.98  24.0
1  396.90  9.14  21.6
2  392.83  4.03  34.7
3  394.63  2.94  33.4
4  396.90  5.33  36.2
======
     0     1     2    3     4     5     6     7     8     9  ...    48  \
0  0.00  0.64  0.64  0.0  0.32  0.00  0.00  0.00  0.00  0.00  ...  0.00
1  0.21  0.28  0.50  0.0  0.14  0.28  0.21  0.07  0.00  0.94  ...  0.00
2  0.06  0.00  0.71  0.0  1.23  0.19  0.19  0.12  0.64  0.25  ...  0.01
3  0.00  0.00  0.00  0.0  0.63  0.00  0.31  0.63  0.31  0.63  ...  0.00
4  0.00  0.00  0.00  0.0  0.63  0.00  0.31  0.63  0.31  0.63  ...  0.00

      49   50     51     52     53     54   55    56  57
0  0.000  0.0  0.778  0.000  0.000  3.756   61   278   1
1  0.132  0.0  0.372  0.180  0.048  5.114  101  1028   1
2  0.143  0.0  0.276  0.184  0.010  9.821  485  2259   1
3  0.137  0.0  0.137  0.000  0.000  3.537   40   191   1
4  0.135  0.0  0.135  0.000  0.000  3.537   40   191   1

[5 rows x 58 columns]
```

PROBLEM 1 [50 points]

Using each dataset, build a decision tree (or regression tree) from the training set. Since the features are numeric values, you will need to use thresholds mechanisms. Report (txt or pdf file) for each dataset the training and testing error for each of your trials:

- simple decision tree using something like Information Gain or other Entropy-like notion of randomness
- regression tree
- try to limit the size of the tree to get comparable training and testing errors (avoid overfitting typical of deep trees)

In [ ]:

```
# 10-folds for spambase

foldSize = int(len(spambaseDf)/10)
spambaseFolds = []
for i in range(9):
  spambaseFolds += [spambaseDf[i*foldSize:(i+1)*foldSize]]
spambaseFolds += [spambaseDf[9*foldSize:len(spambaseDf)]]
```

In [ ]:

```
def labelsVariance(labels: list) -> float:
  n = len(labels)
  mean = sum(labels) / n
  deviations = [(x - mean) ** 2 for x in labels]
  return sum(deviations) / n
```

In [ ]:

```
def findFeatureSplitVar(df: pd.DataFrame, featureIndexesUsed: list) -> tuple:
  """ finds best feature split for decision tree

  Args:
      df (pd.DataFrame): dataframe to split
      featureIndexesUsed (list): index of columns that were already used to split

  Returns:
      tuple:
      - Tuple of split sides
      - the index of columns that have been used to split
      - split theta criteria
  """
  bestFeatureIndex = None
  lowestVar = None
  bestTheta = None
  for i in range(len(df.columns) - 1):
    if (i in featureIndexesUsed):
      continue
    for j in range(0, len(df)):
      tempTheta = df.iloc[j, i]
      left = df[df[i] <= tempTheta]
      right = df[df[i] > tempTheta]
      if len(left) > 1:
        leftLabels = left[left.columns[-1]].to_list()
        varLeft = labelsVariance(leftLabels)
      else:
        varLeft = 0
      if len(right) > 1:
        rightLabels = right[right.columns[-1]].to_list()
        varRight = labelsVariance(rightLabels)
      else:
        varRight = 0
      totalVar = varLeft + varRight
      if lowestVar is None or totalVar < lowestVar:
```

```
            bestTheta = tempTheta
            lowestVar = totalVar
            bestFeatureIndex = i
    leftSplit: pd.DataFrame = df[df[bestFeatureIndex] <= bestTheta]
    rightSplit: pd.DataFrame = df[df[bestFeatureIndex] > bestTheta]
    featureIndexesUsed.append(bestFeatureIndex)
    return (leftSplit, rightSplit), featureIndexesUsed, bestTheta
```

In [ ]:

```
splitDf, featureIndexesUsed, splitTheta = findFeatureSplitVar(trainingHouseDf, [])
print(f"{len(splitDf[0])}")
print(f"{len(splitDf[1])}")
print("<=>")
print(featureIndexesUsed)
print(splitTheta)
```

```
2
431
<=>
[7]
1.1691
```

In [ ]:

```
def trainingVar(df: pd.DataFrame, featureIndexesUsed) -> tuple:
  if df[len(df.columns) - 1].nunique() == 1 or len(df) == 1:
    return df.iloc[0, len(df.columns) - 1]

  if len(df) < 10 or len(featureIndexesUsed) == len(df.columns) -1:
    return statistics.mean(df[len(df.columns) - 1])

  splitDf, featureIndexesUsed, splitTheta = findFeatureSplitVar(df, featureIndexesUsed)

  if len(splitDf[0]) == 0:
    return statistics.mean(splitDf[1][len(df.columns) - 1])
  elif len(splitDf[1]) == 0:
    return statistics.mean(splitDf[0][len(df.columns) - 1])
  else:
    return (
    (featureIndexesUsed[-1], splitTheta),
    trainingVar(splitDf[0], featureIndexesUsed),
    trainingVar(splitDf[1], featureIndexesUsed)
    )
```

In [ ]:

```
regularizedHouseTraining = trainingHouseDf.copy()
regularizedHouseTraining.iloc[:, :-1] = regularizedHouseTraining.iloc[:, :-1].apply(zsc
housingDecisionTree = trainingVar(regularizedHouseTraining, [])
print("Decision Tree:")
print(housingDecisionTree)
```

```
Decision Tree:
((7, -1.2815292139270065), 50.0, ((12, -1.479904560646471), 50.0, ((2, -1.54901232919687
27), 50.0, ((0, 1.273545812919041), ((11, -4.247750516782579), 13.433333333333334, 23.42
14463840399), ((4, 1.025012987051451), 11.771428571428572, 8.047058823529412)))))
```

In [ ]:

```
def findFeatureSplitEntropy(df: pd.DataFrame, featureIndexesUsed: list) -> tuple:
  """ finds best feature split for decision tree

  Args:
```

```
        df (pd.DataFrame): dataframe to split
        featureIndexesUsed (list): index of columns that were already used to split

    Returns:
        tuple:
        - Tuple of split sides
        - the index of columns that have been used to split
        - split theta criteria
    """
    bestFeatureIndex = None
    lowestEntropy = None
    bestTheta = None
    for i in range(len(df.columns) - 1):
      if (i in featureIndexesUsed):
        continue
      for j in range(0, len(df), len(df)//25):
        tempTheta = df.iloc[j, i]
        left = df[df[i] <= tempTheta]
        right = df[df[i] > tempTheta]
        if len(left) > 1:
          leftLabels = left[len(df.columns) - 1].to_list()
          numLabels = len(leftLabels)
          zeroProb = leftLabels.count(0)/numLabels
          oneProb = leftLabels.count(1)/numLabels
          entropyLeft = entropy([zeroProb, oneProb])
        else:
          entropyLeft = 0
        if len(right):
          rightLabels = right[len(df.columns) - 1].to_list()
          numLabels = len(rightLabels)
          zeroProb = rightLabels.count(0)/numLabels
          oneProb = rightLabels.count(1)/numLabels
          entropyRight = entropy([zeroProb, oneProb])
        else:
          entropyRight = 0
        totalEntropy = entropyLeft + entropyRight
        if lowestEntropy is None or totalEntropy < lowestEntropy:
          bestTheta = tempTheta
          lowestEntropy = totalEntropy
          bestFeatureIndex = i
    leftSplit: pd.DataFrame = df[df[bestFeatureIndex] <= bestTheta]
    rightSplit: pd.DataFrame = df[df[bestFeatureIndex] > bestTheta]
    featureIndexesUsed.append(bestFeatureIndex)
    return (leftSplit, rightSplit), featureIndexesUsed, bestTheta
```

In [ ]:

```
def trainingEntropy(df: pd.DataFrame, featureIndexesUsed) -> tuple:
  if df[len(df.columns) - 1].nunique() == 1 or len(df) == 1:
    return df.iloc[0, len(df.columns) - 1]

  if len(df) < 500 or len(featureIndexesUsed) == len(df.columns) -1:
    labels = df[len(df.columns) - 1].to_list()
    return 0 if labels.count(0) >= labels.count(1) else 1

  splitDf, featureIndexesUsed, splitTheta = findFeatureSplitEntropy(df, featureIndexesU

  if len(splitDf[0]) == 0:
    labels = splitDf[1][len(splitDf[1].columns) - 1].to_list()
    return 0 if labels.count(0) >= labels.count(1) else 1
  elif len(splitDf[1]) == 0:
```

```
    labels = splitDf[0][len(splitDf[0].columns) - 1].to_list()
    return 0 if labels.count(0) >= labels.count(1) else 1
  else:
    return (
    (featureIndexesUsed[-1], splitTheta),
    trainingEntropy(splitDf[0], featureIndexesUsed),
    trainingEntropy(splitDf[1], featureIndexesUsed)
    )
```

In [ ]:

```
# test cell

trainingSpam = pd.concat(spambaseFolds[:9])
spambaseDecisionTree = trainingEntropy(trainingSpam, [])
print("===")
print(spambaseDecisionTree)
```

```
===
((52, 5.3), ((26, 25.0), 0, 0), 1)
```

In [ ]:

```
def testing(testPoint: list, tree):
  if type(tree) is tuple:
    return testing(testPoint, tree[1]) if testPoint[tree[0][0]] <= tree[0][1] else test
  else:
    return tree
```

In [ ]:

```
# mean square error
def calcMSE(yPred: list, yActual: list) -> float:
    squaredErrors = []
    for i in range(len(yPred)):
        squaredErrors.append((yPred[i] - yActual[i])**2)
    mse = sum(squaredErrors) / len(yPred)
    return mse
```

In [ ]:

```
predictions = []
for i in range(len(regularizedHouseTraining)):
  testingPoint = regularizedHouseTraining.iloc[i].to_list()
  prediction = testing(testingPoint[:-1], housingDecisionTree)
  predictions.append(prediction)

y_true = trainingHouseDf.iloc[:,-1].to_list()
print(f"Training MSE: {calcMSE(predictions, y_true)}")
```

```
Training MSE: 69.97861170073773
```

In [ ]:

```
regularIzedHouseTesting = testingHouseDf.copy()
regularIzedHouseTesting.iloc[:, :-1] = regularIzedHouseTesting.iloc[:, :-1].apply(zscor

predictions = []
for i in range(len(regularIzedHouseTesting)):
  testingPoint = regularIzedHouseTesting.iloc[i].to_list()
  prediction = testing(testingPoint[:-1], housingDecisionTree)
  predictions.append(prediction)

y_true = testingHouseDf.iloc[:,-1].to_list()
print(f"Testing MSE: {calcMSE(predictions, y_true)}")
```

```
Testing MSE: 53.713238377726036
```

In [ ]:

```
# test cell

testing(spambaseFolds[9][0].to_list(), spambaseDecisionTree)
print(spambaseFolds[9][0].to_list()[-1])
```

```
0.0
```

In [ ]:

```
# k-fold validation full code

trainingErrorRates = []
errorRates = []
for k in range(10):
  trainingWrongCount = 0
  wrongCount = 0
  trainingData = spambaseFolds[:k] + spambaseFolds[k+1:]
  test: pd.DataFrame = spambaseFolds[k]
  training = pd.concat(trainingData)
  training.iloc[:, :-1] = training.iloc[:, :-1].apply(zscore)
  test.iloc[:, :-1] = test.iloc[:, :-1].apply(zscore)
  decisionTree = trainingEntropy(training, [])

  for i in range(len(training)):
    testingPoint = training.iloc[i].to_list()
    prediction = testing(testingPoint[:-1], decisionTree)
    if prediction != testingPoint[-1]:
      trainingWrongCount += 1
  trainingErrorRates.append(trainingWrongCount/len(training))

  for i in range(len(test)):
    testingPoint = test.iloc[i].to_list()
    prediction = testing(testingPoint[:-1], decisionTree)
    if prediction != testingPoint[-1]:
      wrongCount += 1
  errorRates.append(wrongCount/len(test))

print(f"Training Error: {statistics.mean(trainingErrorRates)}")
print(f"Testing Error: {statistics.mean(errorRates)}")
```

```
C:\Users\Ethan Yu\AppData\Local\Temp\ipykernel_12152\1018614307.py:12: SettingWithCopyWa
rning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_
guide/indexing.html#returning-a-view-versus-a-copy
  test.iloc[:, :-1] = test.iloc[:, :-1].apply(zscore)
Training Error: 0.394045832472961
Testing Error: 0.3941304347826087
```

PROBLEM 2 [50 points]

Using each of the two datasets above, apply regression on the training set to find a linear fit with the labels. Implement linear algebra exact solution (normal equations).

- Compare the training and testing errors (mean sum of square differences between prediction and actual label).
- Compare with the decision tree results

In [ ]:

```
# re-import

trainingHouseDf = pd.read_csv(r"housing_train.txt", header=None, delim_whitespace=True)
testingHouseDf = pd.read_csv(r"housing_test.txt", header=None, delim_whitespace=True)

spambaseDf = pd.read_csv(r"spambase.data", header=None)

foldSize = int(len(spambaseDf)/10)
spambaseFolds = []
for i in range(9):
  spambaseFolds += [spambaseDf[i*foldSize:(i+1)*foldSize]]
spambaseFolds += [spambaseDf[9*foldSize:len(spambaseDf)]]
```

In [ ]:

```
def addIntercept(matrix: np.array) -> np.array:
  x = matrix.copy()
  x.insert(0, "intercept", [1 for i in range(len(x))])
  return x
```

In [ ]:

```
x = addIntercept(trainingHouseDf)
print(x.head())
```

```
   intercept        0     1     2  3      4      5     6       7  8      9  \
0          1  0.00632  18.0  2.31  0  0.538  6.575  65.2  4.0900  1  296.0
1          1  0.02731   0.0  7.07  0  0.469  6.421  78.9  4.9671  2  242.0
2          1  0.02729   0.0  7.07  0  0.469  7.185  61.1  4.9671  2  242.0
3          1  0.03237   0.0  2.18  0  0.458  6.998  45.8  6.0622  3  222.0
4          1  0.06905   0.0  2.18  0  0.458  7.147  54.2  6.0622  3  222.0

     10      11    12    13
0  15.3  396.90  4.98  24.0
1  17.8  396.90  9.14  21.6
2  17.8  392.83  4.03  34.7
3  18.7  394.63  2.94  33.4
4  18.7  396.90  5.33  36.2
```

In [ ]:

```
housingLabels = x[x.columns[len(x.columns)-1]]
training = x.drop(columns=x.columns[-1], axis=1)
print(housingLabels.head())
print(training.head())
```

```
0    24.0
1    21.6
2    34.7
3    33.4
4    36.2
Name: 13, dtype: float64
   intercept        0     1     2  3      4      5     6       7  8      9  \
0          1  0.00632  18.0  2.31  0  0.538  6.575  65.2  4.0900  1  296.0
1          1  0.02731   0.0  7.07  0  0.469  6.421  78.9  4.9671  2  242.0
2          1  0.02729   0.0  7.07  0  0.469  7.185  61.1  4.9671  2  242.0
3          1  0.03237   0.0  2.18  0  0.458  6.998  45.8  6.0622  3  222.0
```

```
4          1  0.06905   0.0  2.18  0  0.458  7.147  54.2  6.0622  3  222.0

     10      11     12
0  15.3  396.90   4.98
1  17.8  396.90   9.14
2  17.8  392.83   4.03
3  18.7  394.63   2.94
4  18.7  396.90   5.33
```

In [ ]:

```
trainingMatrix = training.to_numpy()
labelsMatrix = housingLabels.to_numpy()
print(trainingMatrix.shape)
print(labelsMatrix.shape)
```

```
(433, 14)
(433,)
```

In [ ]:

```
def getW(x: np.array, y: np.array) -> np.array:
  xTranspose = np.transpose(x)
  dxdInverse = np.linalg.inv(np.matmul(xTranspose, x))
  return np.matmul(dxdInverse, np.matmul(xTranspose, y))
```

In [ ]:

```
w = getW(trainingMatrix, labelsMatrix)
print(w)
```

```
[ 3.95843212e+01 -1.01137046e-01  4.58935299e-02 -2.73038670e-03
  3.07201340e+00 -1.72254072e+01  3.71125235e+00  7.15862492e-03
 -1.59900210e+00  3.73623375e-01 -1.57564197e-02 -1.02417703e+00
  9.69321451e-03 -5.85969273e-01]
```

In [ ]:

```
# training predictions

trainPredictions = np.matmul(trainingMatrix, w)
print(f"Testing Error: {calcMSE(trainPredictions, labelsMatrix)}")
```

```
Testing Error: 22.081273187013167
```

In [ ]:

```
housingTesting = addIntercept(testingHouseDf)

testLabels = housingTesting[housingTesting.columns[len(housingTesting.columns)-1]]
testFeatures = housingTesting.drop(columns=housingTesting.columns[-1], axis=1)
print(testFeatures.head())
print(testLabels.head())
```

```
   intercept        0    1     2  3      4      5     6       7  8      9  \
0          1  0.84054  0.0  8.14  0  0.538  5.599  85.7  4.4546  4  307.0
1          1  0.67191  0.0  8.14  0  0.538  5.813  90.3  4.6820  4  307.0
2          1  0.95577  0.0  8.14  0  0.538  6.047  88.8  4.4534  4  307.0
3          1  0.77299  0.0  8.14  0  0.538  6.495  94.4  4.4547  4  307.0
4          1  1.00245  0.0  8.14  0  0.538  6.674  87.3  4.2390  4  307.0

     10      11     12
0  21.0  303.42  16.51
1  21.0  376.88  14.81
2  21.0  306.38  17.28
3  21.0  387.94  12.80
```

```
4  21.0  380.23  11.98
0    13.9
1    16.6
2    14.8
3    18.4
4    21.0
Name: 13, dtype: float64
```

In [ ]:

```
testingMatrix = testFeatures.to_numpy()
testLabelsMatrix = testLabels.to_numpy()
print(testingMatrix.shape)
print(testLabelsMatrix.shape)
```

```
(74, 14)
(74,)
```

In [ ]:

```
testPredictions = np.matmul(testingMatrix, w)
print(f"Testing Error: {calcMSE(testPredictions, testLabelsMatrix)}")
```

```
Testing Error: 22.63825629658623
```

In [ ]:

```
trainErrors = []
testErrors = []

for k in range(10):
  trainingData = spambaseFolds[:k] + spambaseFolds[k+1:]
  test: pd.DataFrame = addIntercept(spambaseFolds[k])
  training = addIntercept(pd.concat(trainingData))

  trainingLabels = training[training.columns[len(training.columns)-1]]
  trainingFeatures = training.drop(columns=training.columns[-1], axis=1)

  trainingMatrix = trainingFeatures.to_numpy()
  trainingLabelsMatrix = trainingLabels.to_numpy()
  w = getW(trainingMatrix, trainingLabelsMatrix)
  trainResults = np.matmul(trainingMatrix, w)
  trainPredictions = list(map(lambda x: 0 if x <= 0.5 else 1, trainResults))
  wrongCount = 0
  for i in range(len(trainPredictions)):
    if trainPredictions[i] != trainingLabelsMatrix[i]:
      wrongCount += 1
  trainErrors.append(wrongCount/len(trainPredictions))

  testingLabels = test[test.columns[len(test.columns)-1]]
  testingFeatures = test.drop(columns=test.columns[-1], axis=1)

  testingMatrix = testingFeatures.to_numpy()
  testingLabelsMatrix = testingLabels.to_numpy()
  testingResults = np.matmul(testingMatrix, w)
  testPredictions = list(map(lambda x: 0 if x <= 0.5 else 1, testingResults))
  wrongCount = 0
  for i in range(len(testPredictions)):
    if testPredictions[i] != testingLabelsMatrix[i]:
      wrongCount += 1
  testErrors.append(wrongCount/len(testPredictions))

print(f"Training Error: {statistics.mean(trainErrors)}")
print(f"Testing Error: {statistics.mean(testErrors)}")
```

```
Training Error: 0.10558060843199908
Testing Error: 0.15062670942186174
```

PROBLEM 3 [20 points]

DHS chapter8, Pb1. Given an arbitrary decision tree, it might have repeated queries splits (feature f, threshold t) on some paths root-leaf. Prove that there exists an equivalent decision tree only with distinct splits on each path.

If there are two splits with the same feature and threshold on the same path, second split will have one side have all of the data and the other side will have none. In training, there will be no reduction of variance or entropy reduction when a split has all of the data on one side and none on the other. Therefore, removing the duplicate splits will keep the same variance reductions at each node with the same paths leaving an equivalent decision tree only with distinct splits on each path.

PROBLEM 4 [20 points]

DHS chapter8, Consider a binary decision tree using entropy splits.

1. Prove that the decrease in entropy by a split on a binary yes/no feature can never be greater than 1 bit.
2. Generalize this result to the case of arbitrary branching B>1.

a: The range that entropy is measured is between 1 and 0 where 1 is the highest amount of entropy and 0 is the lowest amount of entropy. By definition, a perfect split would result in both sides have 0 entropy and would therefore sum to 0. Therefore, the highest decrease of entropy that can happen is 1 - 0, which is 1.

b: The maximum amount of entropy of the node before the split is still 1. A perfect split will result in 0 entropy for every branch. When summed together, the entropy will be 0. 1 - 0 is one, so the decrease in entropy is still maximum 1 bit. Any imperfect split would result in an entropy > 0 which would make the decrease less than 1.

PROBLEM 5 [20 points]

Derive explicit formulas for normal equations solution presented in class for the case of one input dimension. (Essentially assume the data is (x_i,y_i) i=1,2,..., m and you are looking for h(x) = ax+b that realizes the minimum mean square error. The problem asks you to write down explicit formulas for a and b.)

HINT: Do not simply copy the formulas from here (but do read the article): either take the general formula derived in class and make the calculations (inverse, multiplications, transpose) for one dimension or derive the formulas for a and b from scratch; in either case show the derivations. You can compare your end formulas with the ones linked above.

PROBLEM 5

$$J(a,b) = \frac{1}{N}\sum_{i=1}^{N} (a x_i + b - y_i)^2 \rightarrow MSE$$

Sample means: $\bar{x} = \frac{\sum_{i=1}^{N} x_i}{N}$

$\bar{y} = \frac{\sum_{i=1}^{N} y_i}{N}$

$$\frac{\partial J(a,b)}{\partial (b)} = \frac{1}{N}\sum_{i=1}^{N} 2(b + a x_i - y_i) = 0$$

$$\Rightarrow b + a\bar{x} - \bar{y} = 0$$

$$\Rightarrow \boxed{b = \bar{y} - a\bar{x}}$$

$$\frac{\partial J(a,b)}{\partial (a)} = \frac{1}{N}\sum_{i=1}^{N} 2(b + a x_i - y_i) x_i = 0$$

$$\Rightarrow b\bar{x} + a\frac{\sum x_i^2}{N} - \frac{\sum x_i y_i}{N} = 0$$

$$\Rightarrow a = \frac{\sum (x_i y_i) - N\bar{x}\bar{y}}{\sum x_i^2 - N\bar{x}^2}$$

$$\Rightarrow \boxed{a = \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{\sum (x_i - \bar{x})^2}}$$

PROBLEM 7 [20points]

DHS chapter5, The convex hull of a set of vectors xi,i = 1,...,n is the set of all vectors of the form

$$\mathbf{x} = \sum_{i=1}^{n} \alpha_i \mathbf{x}_i$$

where the coefficients αi are nonnegative and sum to one. Given two sets of vectors, show that either they are linearly separable or their convex hulls intersect. Hint on easy part: that the two conditions cannot happen simultaneously. Suppose that both statements are true, and consider the classification of a point in the intersection of the convex hulls.

PROBLEM 7

Covex hull $X = \sum_{i=1}^{N} \alpha_i x_i$

$$Y = \sum_{j=1}^{D} \beta_j y_j$$

A linear discriminant function can be written

$g(x) = w^T x_i + w_{0i}$

To plug that into a convex hull

$$X = \sum \alpha_i (w^T x_i + w_{0i})$$

$$Y = \sum \beta_j (w^T y_j + w_{0j})$$

An intersection between the two convex hulls means

$$\sum \alpha_i (w^T x_i + w_{0i}) = \sum \beta_j (w^T y_j + w_{0j})$$

The seperation surface is $g(x) = 0$. Therefore to be linearly seperable

$$g(X) > 0 \quad \text{and} \quad g(Y) \leq 0$$

But since α_i and β_j are both non-negative, it is impossible to simultaneously be greater than and less than equal to zero.

Therefore, two convex hulls

cannot be linearly separated
and intersected at the same time