CS189 HW5

David Chen

March 2022

Contents

1 Write-up and Honor Code

Collaborated with: N/A.

I certify that all solutions are entirely in my own words and that I have not looked at another student's solutions. I have given credit to all external sources I consulted.

Signed: David Chen

2 Random Forest Motivation

2.1 Part 1

Note that the average of the uncorrelated random variables is given by $\frac{1}{n}\sum_i Y_i$. Therefore,

$$\begin{aligned} E\left[\frac{1}{n}\sum_i Y_i\right] &= \frac{1}{n}\sum_i E\left[Y_i\right] \\ &= \frac{1}{n}\sum_i \mu \\ &= \frac{1}{n} n\mu \\ &= \mu. \end{aligned}$$

$$\begin{aligned} Var\left(\frac{1}{n}\sum_i Y_i\right) &= \frac{1}{n^2}\sum_i Var\left(Y_i\right) \\ &= \frac{1}{n^2}\sum_i \sigma^2 \\ &= \frac{1}{n^2} n\sigma^2 \\ &= \frac{\sigma^2}{n}. \end{aligned}$$

The mean of the average is the same as that of a random variable, while the variance decreases by a factor of $\frac{1}{n}$.

2.2 Part 2

2.2.1 Section (a)

When $n' = n$, the probability of a point not being chosen is

$$\begin{aligned} P\left(\text{Point } i \text{ not chosen}\right) &= \left(\frac{n-1}{n}\right)^n \\ &= \left(1 - \frac{1}{n}\right)^n. \end{aligned}$$

Note that this is approximately equal to $\frac{1}{e}$ as $n \to \infty$, as

$$\begin{aligned} \left(\frac{n}{n-1}\right)^n &= \left(1 + \frac{1}{n-1}\right)^n \\ &\approx \left(1 + \frac{1}{n}\right)^n \\ &\approx e \\ \therefore \left(\left(\frac{n}{n-1}\right)^n\right)^{-1} &\approx \frac{1}{e}. \end{aligned}$$

Therefore, the probability of a point being chosen is

$$\begin{aligned} 1 - P\left(\text{Point } i \text{ not chosen}\right) &= 1 - \frac{1}{e} \\ &\approx 0.63. \end{aligned}$$

2.2.2 Section (b)

The hyperparameter T can be tuned by training learners from a training data set and testing it on a validation data set that was set aside beforehand. For instance, K-fold cross-validation can be used to tune the hyperparameter T.

2.3 Part 3

First note that, for any two random variables X and Y, $\mathrm{Var}(X+Y) = \mathrm{Var}\,(X) + \mathrm{Var}\,(Y) + 2\mathrm{Cov}\,(X,Y)$.

For the variance of our average $\frac{1}{n}\sum_i Z_i$, the covariance of each pair of Z_i needs to be added twice on to the individual variances of each Z_i. Since there are $\frac{n(n+1)}{2}$ unique pairs and each pair has covariance ρ, the variance of our average becomes

$$\begin{aligned} Var\left(\frac{1}{n}\sum_i Z_i\right) &= \frac{1}{n^2}Var\left(\sum Z_i\right) \\ &= \frac{1}{n^2}\left(n\cdot\sigma^2 + 2\cdot\frac{n\,(n+1)}{2}\cdot\rho\right) \\ &= \frac{\sigma^2+\rho\,(n+1)}{n}. \end{aligned}$$

2.4 Part 4

A random forest of stumps could potentially be a good idea.

First, let us consider the performance of an individual stump. The stump has low variance, as the difference in classifications are based on one feature. However, it has high bias, since it can only use the one feature to make decisions rather than a normal decision tree, which utilizes multiple features.

Now, if we were to combine multiple stumps and let them vote for a classification, the result could change. The variance would still stay low, as the variance of each stump is low and averaging, as shown above, leads to low variance. The difference now is the bias, since we are taking decisions from stumps that use different features. Since each stump covers a section of the feature space, combining them and letting them vote could lead to a decision that covers most of the feature space and therefore leads to low bias. Overall, this could lead to less overfitting while maintaining strong performance of a classifier.

The assumption made for the earlier conclusion is that there is a decent amount of stumps and they cover a wide range of features. If this were not true, then much of the feature space would be left uncovered and the bias would remain high.

3 Decision Trees for Classification

Entire question is included in the attached Python Notebook.

CS189 Homework 5

Imports

In [41]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy
from scipy import io
from scipy import stats
import random

from collections import Counter

import pandas as pd
from numpy import genfromtxt
from sklearn.tree import DecisionTreeClassifier, export_graphviz
from sklearn.base import BaseEstimator, ClassifierMixin
from sklearn.model_selection import cross_validate
import sklearn.tree
from sklearn.impute import SimpleImputer

import random
random.seed(246810)
np.random.seed(246810)

eps = 1e-5  # a small number
```

Decision Trees for Classification

Setup

In [42]:
```
spamData = io.loadmat("data/spam_data/spam_data.mat")
spamData["training_labels"] = spamData["training_labels"].reshape((spamData["tra

titanicData = pd.read_csv("data/titanic/titanic_training.csv")
titanicLabels = titanicData["survived"]
del titanicData["survived"]
del titanicData["ticket"]
# del titanicData["cabin"]
# del titanicData["parch"]
# del titanicData["embarked"]

titanicFeatureLabels = list(titanicData.columns)
titanicLabels = titanicLabels.to_numpy()

# Fills NaNs in the data with the mode of the column
imp_mode = SimpleImputer(missing_values=np.nan, strategy='most_frequent')
titanicData = imp_mode.fit_transform(titanicData)

titanicTest = pd.read_csv("data/titanic/titanic_testing_data.csv")
del titanicTest["ticket"] # this is the only one that increased val accuracy
# del titanicTest["cabin"]
# del titanicTest["parch"]
# del titanicTest["embarked"]
titanicTest = imp_mode.fit_transform(titanicTest)


# Node used in Decision Trees
class Node:
    def __init__(self, left, right, splitFeature, thresh, pred=True):
        self.left = left
        self.right = right
        self.splitFeature = splitFeature
        self.thresh = thresh
        # For leaf nodes, this is true if you want to predict larger class when
        # when you want to predict larger class when x < thresh
        self.pred = pred


# Vectorized function for hashing for np efficiency
def w(x):
    return np.int(hash(x)) % 1000

h = np.vectorize(w)

# Vectorize input dataset, which gets rid of strings and uses boolean features i
def vectorize(dataset, featureLabels=None, minFrequency=3):
    stringColumns = []
    booleanFeatureList = []
    for i in range(dataset.shape[1]):
        if not isinstance(dataset[0][i], str):
            continue
        if featureLabels:
            featureLabel = featureLabels[i]
        else:
            featureLabel = f"Feature #{i + 1}"
        uniqueVals, counts = np.unique(dataset[:,i], return_counts=True)
        uniqueVals = uniqueVals[np.where(counts >= minFrequency)]
        booleanFeatures = [(np.zeros((dataset.shape[0])), f"{featureLabel}: {val
        for j in range(dataset.shape[0]):
            value = dataset[j][i]
            indexArray = np.where(uniqueVals==value)[0]
            if indexArray.size == 0:
                continue
            idx = indexArray[0]
            booleanFeatures[idx][0][j] = 1
        booleanFeatureList += booleanFeatures
        stringColumns.append(i)
    stringColumns.reverse()
    for i in stringColumns:
        dataset = np.delete(arr=dataset, obj=i, axis=1)
        if featureLabels:
            featureLabels.pop(i)
    colToFeature = {}
    for feature in booleanFeatureList:
        idx = dataset.shape[1]
        featureCol = feature[0]
        featureCol = np.reshape(featureCol, (featureCol.shape[0], 1))
        dataset = np.hstack((dataset, featureCol))
        colToFeature[idx] = feature[1]
        if featureLabels:
            featureLabels.append(feature[1])
    if featureLabels:
        return featureLabels, dataset
    return colToFeature, dataset

stackedTitanicData = np.vstack((titanicData, titanicTest))
colFeatureDict, stackedTitanicData = vectorize(stackedTitanicData, titanicFeatur
titanicData = stackedTitanicData[:titanicData.shape[0]]
titanicTest = stackedTitanicData[titanicData.shape[0]:]
print(titanicData.shape)
print(titanicData)

# for key in colFeatureDict.keys():
#     titanicFeatureLabels.append(colFeatureDict[key])

# colFeatureDict, titanicData = vectorize(titanicData, titanicFeatureLabels)
# __, titanicTest = vectorize(titanicTest, titanicFeatureLabels)

def results_to_csv(y_test, path):
    y_test = y_test.astype(int)
    df = pd.DataFrame({'Category': y_test})
    df.index += 1  # Ensures that the index starts at 1.
    df.to_csv(path, index_label='Id')
```

```
(1000, 26)
[[3.0 22.0 0.0 ... 0.0 0.0 1.0]
 [1.0 22.0 0.0 ... 1.0 0.0 0.0]
 [2.0 23.0 0.0 ... 1.0 0.0 0.0]
 ...
 [2.0 63.0 1.0 ... 0.0 0.0 1.0]
 [3.0 41.0 0.0 ... 0.0 0.0 1.0]
 [2.0 34.0 1.0 ... 0.0 0.0 1.0]]
```

Part 1: Implement Decision Trees

In [60]:

```
class DecisionTree:
    def __init__(self, max_depth=3, feature_labels=None, m=0, verbose=False):
        # TODO implement __init__ function
        self.max_depth = max_depth
        self.feature_labels = feature_labels
        self.head = None # Head of the tree that will be fitted later
        self.m = m
        self.verbose = verbose

    @staticmethod
    def information_gain(X, y, feature, thresh):
        # TODO implement information gain function
        H_S = DecisionTree.entropy(y)

        # Weighted average entropy
        H_after = 0

        S_l = y[np.where(X[:, feature] < thresh)]
        H_after += len(S_l) * DecisionTree.entropy(S_l)

        S_r = y[np.where(X[:, feature] >= thresh)]
        H_after += len(S_r) * DecisionTree.entropy(S_r)

        H_after /= len(y)
        return H_S - H_after
    @staticmethod
    def entropy(y):
        # TODO implement entropy (or optionally gini impurity) function
        classes, counts = np.unique(ar=y, return_counts=True)

        H_s = 0
        for count in counts:
            count /= len(y)
            H_s -= count * np.log2(count)
        return H_s

    def split(self, X, y):
        # TODO implement split function
        # Feature and thresh that give the largest information gain
        maxFeature = 0
        maxThresh = 0
        maxInfoGain = 0

        if self.m > 0:
            featureIndices = random.sample(range(X.shape[1]), self.m)
        else:
            featureIndices = range(X.shape[1])

        for featureIdx in featureIndices:
            threshList = np.unique(X[:,featureIdx])

            for thresh in threshList:
                infoGain = self.information_gain(X, y, featureIdx, thresh + eps)

                if infoGain > maxInfoGain:
                    maxFeature = featureIdx
                    maxThresh = thresh + eps
                    maxInfoGain = infoGain
        return maxFeature, maxThresh

    def fit(self, X, y):
        # TODO implement fit function
        self.head = self.fitRecursive(X, y, 0)

    def fitRecursive(self, X, y, depth):
        feature, thresh = self.split(X, y)
        lowerSplit = np.where(X[:,feature] < thresh)
        upperSplit = np.where(X[:,feature] >= thresh)

        X_l = X[lowerSplit]
        y_l = y[lowerSplit]
        X_r = X[upperSplit]
        y_r = y[upperSplit]

        # If we are at max depth or have no information gain, we can stop our re
        if depth == self.max_depth - 1 or X_l.shape[0] == 0 or X_r.shape[0] == 0
        X.shape[0] == 0 or DecisionTree.information_gain(X, y, feature, thresh)
            uniqueLabels, counts = np.unique(y, return_counts=True)
            maxCount = np.argmax(counts)
            pred = uniqueLabels[maxCount]
            return Node(None, None, feature, thresh, pred)

        left = self.fitRecursive(X_l, y_l, depth + 1)
        right = self.fitRecursive(X_r, y_r, depth + 1)
        return Node(left, right, feature, thresh)
    def predict(self, X):
        # TODO implement predict function
        predictions = np.zeros((X.shape[0]))

        for idx in range(X.shape[0]):
            x = X[idx]

            ptr = self.head
            while not (ptr.left == None and ptr.right == None):
                feature = ptr.splitFeature
                thresh = ptr.thresh

                if x[feature] < thresh:
                    if self.verbose:
                        print(f"{self.feature_labels[feature]} < {thresh}")
                    ptr = ptr.left
                else:
                    if self.verbose:
                        print(f"{self.feature_labels[feature]} >= {thresh}")
                    ptr = ptr.right
            feature = ptr.splitFeature
            thresh = ptr.thresh
#             if ptr.pred:
#                 prediction = int(x[feature] >= thresh)
#             else:
#                 prediction = int(x[feature] < thresh)
            prediction = ptr.pred
            if self.verbose:
                if x[feature] < thresh:
                    print(f"{self.feature_labels[feature]} < {thresh}")
                else:
                    print(f"{self.feature_labels[feature]} >= {thresh}")
                print() # Empty line to look better
            predictions[idx] = prediction
        return predictions
```

Part 2: Implement a Random Forest

In [44]:

```
class BaggedTrees(BaseEstimator, ClassifierMixin):
    def __init__(self, maxDepth=3, feature_labels=None, n=200, verbose=False):
        self.n = n
        self.decision_trees = [DecisionTree(max_depth=maxDepth,\
                                            feature_labels=feature_labels,\
                                            verbose=verbose) for i in range(self

    def fit(self, X, y):
        # TODO implement function
        for decisionTree in self.decision_trees:
            # remember to do BAGGING
            bagIndices = random.choices(range(X.shape[0]), k=self.n)
            bagX = np.array([X[idx] for idx in bagIndices]) # Subsample chosen w
            bagY = np.array([y[idx] for idx in bagIndices])

            decisionTree.fit(bagX, bagY)

    def predict(self, X):
        # TODO implement function
        totalPredictions = []

        for decisionTree in self.decision_trees:
            prediction = decisionTree.predict(X)
            totalPredictions.append(prediction)
        totalPredictions = np.array(totalPredictions)
        predictions = np.zeros((totalPredictions.shape[1]))
        for i in range(totalPredictions.shape[1]):
            classes, counts = np.unique(ar=totalPredictions[:,i], return_counts=
            predictionIdx = np.argmax(counts)
            predictions[i] = classes[predictionIdx]
        return predictions

class RandomForest(BaggedTrees):
    def __init__(self, maxDepth=3, feature_labels=None, n=200, m=1, verbose=Fals
        # TODO implement function
        self.n = n
        self.decision_trees = [DecisionTree(maxDepth, feature_labels, m, verbose
```

Part 3: Describe implementation details

1. For both categorical and numerical features, I dealt with missing values by using the mode of the features' values. The goal of this is to basically have the tree dismiss the feature when a categorical value is missing, as some categorical features had more NaNs than inputs from samples. For numerical features, using the mode was done so that it would be closest to other sample's values, so the tree would likely not prune the sample due to that feature value being missing.
2. My stopping criterion was maximum depth.
3. I first implemented BAGGING by making multiple samples(chosen with replacement), training a decision tree on each sample, then letting the decision trees vote for the result. Afterwards, I let my Random Forest class inherit the Bagged Tree class, with the only change being the addition of an m value. I added an extra change in the Decision Tree class to accommodate this m value by chosing m features before splitting and training.
4. Not yet. I'm just trying to optimize wherever possible.
5. Not yet.

Part 4: Performance Evaluation

In [61]:

```
datasets = [[spamData["training_data"], spamData["training_labels"], "spam datas
            [titanicData, titanicLabels, "titanic dataset"]]

valProportion = 0.2
for dataset in datasets:
    X = dataset[0]
    y = dataset[1]
    name = dataset[2]

    dataSize = X.shape[0]
    indices = np.arange(dataSize)
    np.random.shuffle(indices)
    shuffledData = X[indices]
    shuffledLabels = y[indices]

    valSize = int(valProportion * dataSize)
    trainingData = shuffledData[valSize:]
    trainingLabels = shuffledLabels[valSize:]
    validationData = shuffledData[:valSize]
    validationLabels = shuffledLabels[:valSize]

    numFeatures = trainingData.shape[1]

    decisionTree = DecisionTree(max_depth=5)
    randomForest = RandomForest(m=int(np.sqrt(numFeatures)))

    decisionTree.fit(trainingData, trainingLabels)
    randomForest.fit(trainingData, trainingLabels)

    dtTrainingPredictions = decisionTree.predict(trainingData)
    dtValPredictions = decisionTree.predict(validationData)

    rfTrainingPredictions = randomForest.predict(trainingData)
    rfValPredictions = randomForest.predict(validationData)

    # Get and output the accuracies
    dtTrainingAccuracy = np.sum(dtTrainingPredictions == trainingLabels) / train
    dtValAccuracy = np.sum(dtValPredictions == validationLabels) / validationDat
    print(f"The training accuracy for the {name} using a Decision Tree is: {dtTr
    print(f"The validation accuracy for the {name} using a Decision Tree is: {dt

    rfTrainingAccuracy = np.sum(rfTrainingPredictions == trainingLabels) / train
    rfValAccuracy = np.sum(rfValPredictions == validationLabels) / validationDat
    print(f"The training accuracy for the {name} using a Random Forest is: {rfTr
    print(f"The validation accuracy for the {name} using a Random Forest is: {rf

    print() # Empty line to look better
```

```
The training accuracy for the spam dataset using a Decision Tree is: 0.804494925
0845819.
The validation accuracy for the spam dataset using a Decision Tree is: 0.7562862
669245648.
The training accuracy for the spam dataset using a Random Forest is: 0.722087965
2005799.
The validation accuracy for the spam dataset using a Random Forest is: 0.6953578
33655706.

The training accuracy for the titanic dataset using a Decision Tree is: 0.82375.
The validation accuracy for the titanic dataset using a Decision Tree is: 0.845.
The training accuracy for the titanic dataset using a Random Forest is: 0.7775.
The validation accuracy for the titanic dataset using a Random Forest is: 0.81.
```

In [46]:

```
#spamModel = DecisionTree(max_depth=20)
spamModel = RandomForest(m=int(np.sqrt(spamData["training_data"].shape[1])))
spamModel.fit(spamData["training_data"], spamData["training_labels"])
spamPredictions = spamModel.predict(spamData["test_data"])
path = "SpamPredictions.csv"
results_to_csv(spamPredictions, path)
print("Spam Done")

#titanicModel = DecisionTree(max_depth=10)
titanicModel = RandomForest(m=int(np.sqrt(titanicData.shape[1])))
```

```
titanicModel.fit(titanicData, titanicLabels)
titanicPredictions = titanicModel.predict(titanicTest)
path = "TitanicPredictions.csv"
results_to_csv(titanicPredictions, path)
print("Titanic Done")
```

```
Spam Done
Titanic Done
```

Kaggle Display Name: David Chen Spam Score: 0.74765 Titanic Score: 0.75161

Part 5

In [62]:

```
spamFeatureLabels = [
            "pain", "private", "bank", "money", "drug", "spam", "prescription",
            "creative", "height", "featured", "differ", "width", "other",
            "energy", "business", "message", "volumes", "revision", "path",
            "meter", "memo", "planning", "pleased", "record", "out",
            "semicolon", "dollar", "sharp", "exclamation", "parenthesis",
            "square_bracket", "ampersand"
        ]
spamDT = DecisionTree(max_depth=20, feature_labels=spamFeatureLabels, verbose=Tr
spamDT.fit(spamData["training_data"], spamData["training_labels"])
singleTest = spamData["test_data"][0]
singleTest = singleTest.reshape(1, singleTest.shape[0])
testPrediction = spamDT.predict(singleTest)
result = "spam" if int(testPrediction[0]) else "ham"
print(f"Therefore, the prediction is {result}.")
```

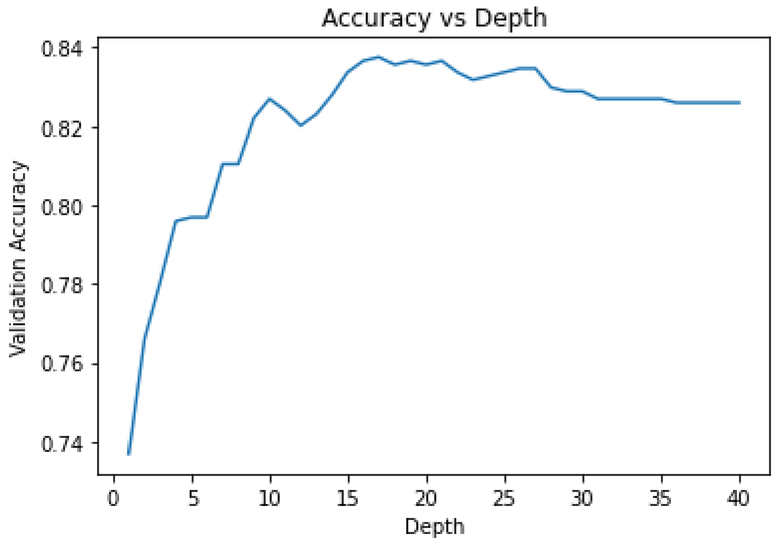
```
exclamation < 1e-05
meter < 1e-05
parenthesis < 1e-05
volumes < 1e-05
ampersand < 1e-05
pain < 1e-05
semicolon < 1e-05
prescription < 1e-05
square_bracket < 1e-05
energy < 1.00001
bank < 1.00001
drug < 1e-05
differ < 1e-05
memo < 1e-05
path < 1e-05
spam < 1e-05
planning < 1e-05
dollar < 6.00001
sharp < 1.00001
revision < 1e-05

Therefore, the prediction is ham.
```

In [48]:

```
# Validation with 80/20 split to tune depth hyperparameter
valProportion = 0.2

X = spamData["training_data"]
y = spamData["training_labels"]
name = "Spam Dataset"

dataSize = X.shape[0]
```

```
indices = np.arange(dataSize)
np.random.shuffle(indices)
shuffledData = X[indices]
shuffledLabels = y[indices]

valSize = int(valProportion * dataSize)
trainingData = shuffledData[valSize:]
trainingLabels = shuffledLabels[valSize:]
validationData = shuffledData[:valSize]
validationLabels = shuffledLabels[:valSize]

depths = range(1, 41)
valAccuracies = []
for depth in depths:
    decisionTree = DecisionTree(max_depth=depth)
    decisionTree.fit(trainingData, trainingLabels)

    valPredictions = decisionTree.predict(validationData)
    valAccuracy = np.sum(valPredictions == validationLabels) / validationData.sh
    valAccuracies.append(valAccuracy)
    print(f"Finished training with depth {depth}")
plt.plot(depths, valAccuracies, label="Validation Data")
plt.xlabel("Depth")
plt.ylabel("Validation Accuracy")
plt.title("Accuracy vs Depth")
plt.show()
# plot that shit here(prolly use matplotlib)
```

```
Finished training with depth 1
Finished training with depth 2
Finished training with depth 3
Finished training with depth 4
Finished training with depth 5
Finished training with depth 6
Finished training with depth 7
Finished training with depth 8
Finished training with depth 9
Finished training with depth 10
Finished training with depth 11
Finished training with depth 12
Finished training with depth 13
Finished training with depth 14
Finished training with depth 15
Finished training with depth 16
Finished training with depth 17
Finished training with depth 18
Finished training with depth 19
Finished training with depth 20
Finished training with depth 21
Finished training with depth 22
Finished training with depth 23
Finished training with depth 24
Finished training with depth 25
Finished training with depth 26
Finished training with depth 27
Finished training with depth 28
Finished training with depth 29
Finished training with depth 30
Finished training with depth 31
Finished training with depth 32
Finished training with depth 33
Finished training with depth 34
Finished training with depth 35
Finished training with depth 36
Finished training with depth 37
Finished training with depth 38
Finished training with depth 39
Finished training with depth 40
```



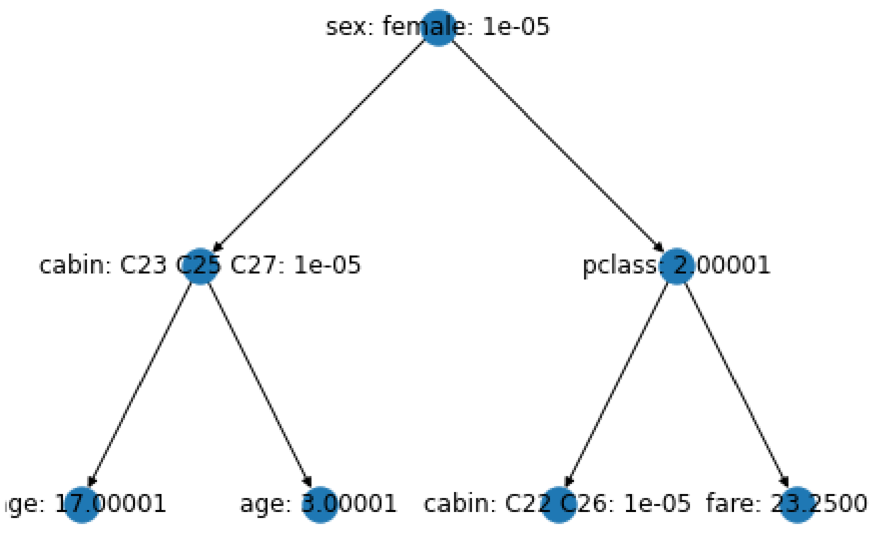
Depth 18 had the highest validation accuracy. This is likely because the bias was too high before depth 18, but the variance got too high after depth 18. Therefore, depth 18 was the depth that minimized the variance plus the bias squared.

Part 6

In [63]:
```
# Setup
import matplotlib.pyplot as plt
import networkx as nx
import pydot
from networkx.drawing.nx_pydot import graphviz_layout

def hierarchy_pos(G, root=None, width=1., vert_gap = 0.2, vert_loc = 0, xcenter

    '''
    From Joel's answer at https://stackoverflow.com/a/29597209/2966723.
    Licensed under Creative Commons Attribution-Share Alike

    If the graph is a tree this will return the positions to plot this in a
    hierarchical layout.

    G: the graph (must be a tree)

    root: the root node of current branch
    - if the tree is directed and this is not given,
      the root will be found and used
    - if the tree is directed and this is given, then
      the positions will be just for the descendants of this node.
    - if the tree is undirected and not given,
      then a random choice will be used.

    width: horizontal space allocated for this branch - avoids overlap with othe

    vert_gap: gap between levels of hierarchy
    vert_loc: vertical location of root

    xcenter: horizontal location of root
    '''
    if not nx.is_tree(G):
        raise TypeError('cannot use hierarchy_pos on a graph that is not a tree'

    if root is None:
        if isinstance(G, nx.DiGraph):
            root = next(iter(nx.topological_sort(G)))  #allows back compatibilit
        else:
            root = random.choice(list(G.nodes))

    def _hierarchy_pos(G, root, width=1., vert_gap = 0.2, vert_loc = 0, xcenter
        '''
        see hierarchy_pos docstring for most arguments

        pos: a dict saying where all nodes go if they have been assigned
        parent: parent of this branch. - only affects it if non-directed

        '''

        if pos is None:
            pos = {root:(xcenter,vert_loc)}
        else:
            pos[root] = (xcenter, vert_loc)
        children = list(G.neighbors(root))
        if not isinstance(G, nx.DiGraph) and parent is not None:
            children.remove(parent)
        if len(children)!=0:
            dx = width/len(children)
            nextx = xcenter - width/2 - dx/2
            for child in children:
                nextx += dx
                pos = _hierarchy_pos(G,child, width = dx, vert_gap = vert_gap,
                                    vert_loc = vert_loc-vert_gap, xcenter=nextx,
                                    pos=pos, parent = root)
        return pos


    return _hierarchy_pos(G, root, width, vert_gap, vert_loc, xcenter)

# Plots the Decision Tree
def plotDT(DT, featureLabels):
    graph = nx.DiGraph()
    rootNode = DT.head
    queue = [rootNode]
    parentDict = {}
    labelDict = {}
    labelDict[rootNode] = f"{featureLabels[rootNode.splitFeature]}: {rootNode.th
    while queue:
        node = queue.pop(0)
        left = node.left
        right = node.right
        if left:
            queue.append(left)
            parentDict[left] = node
            labelDict[left] = f"{featureLabels[left.splitFeature]}: {left.thresh
        if right:
                queue.append(right)
                parentDict[right] = node
                labelDict[right] = f"{featureLabels[right.splitFeature]}: {right.thr

    for child in parentDict.keys():
        parent = parentDict[child]
        graph.add_edge(parent, child)

    pos = hierarchy_pos(graph, root=rootNode)
    nx.draw(graph, pos=pos, with_labels=True, labels=labelDict)
    plt.show()
```

In [64]:

```
titanicDT = DecisionTree(max_depth=3, feature_labels=titanicFeatureLabels, verbo
titanicDT.fit(titanicData, titanicLabels)
plotDT(titanicDT, titanicFeatureLabels)


# Im just doing the stuff below to debug the tree and titanic preprocessing
singleTest = titanicTest[0]
singleTest = singleTest.reshape(1, singleTest.shape[0])
testPrediction = titanicDT.predict(singleTest)
result = "survived" if int(testPrediction[0]) else "died"
print(f"Therefore, the prediction is that the passenger {result}.")
```



```
sex: female < 1e-05
cabin: C23 C25 C27 < 1e-05
age >= 17.00001

Therefore, the prediction is that the passenger died.
```