Summer 2020 CX4641/CS7641 Homework 4

Instructor: Dr. Mahdi Roozbahani ¶

Deadline: July 20th, 11:59 pm

- No unapproved extension of the deadline is allowed. Late submission will lead to 0 credit.
- Discussion is encouraged on Piazza as part of the Q/A. However, all assignments should be done individually.
- You are allowed to resubmit your homework until July 26th 11:59 PM without any penalty.

Instructions for the assignment

- In this assignment, we have programming and writing questions.
- To switch between cell for code and for markdown, see the menu -> Cell -> Cell Type
- You could directly type the Latex equations in the markdown cell.
- Typing with Latex\markdown is required for all the written questions. Handwritten answers would not be accepted.
- If a question requires a picture, you could use this syntax "$< imgsrc =$"" $style =$" $width : 300px;$ " $/ >$" to include them within your ipython notebook.
- Questions marked with **[P]** are programming only and should be submitted to the autograder. Questions marked with **[W]** may required that you code a small function or generate plots, but should **NOT** be submitted to the autograder. It should be submitted on the writing portion of the assignment on gradescope
- The outline of the assignment is as follows:
  - Q1 [25 pts] > Decision Tree Utilities | **[P]**
  - Q2 [20 pts] > Decision Tree | **[P]**
  - Q3 [10 pts (bonus for Undergrad)] > Pruning | **[P]**
  - Q4 [20 pts] > Random Forest | **[P]**
  - Q5 [35 pts] > SVM **[W]** items
  - Q6 [Bonus for all][30 pts] > Neural Network **[P]**

Using the autograder

- You will find two assignments on Gradescope that correspond to HW4: "HW4 - Programming" and "HW4 - Non-programming".
- You will submit your code for the autograder on "HW4 - Programming" in the following format:
  - util.py
  - decision_tree.py
  - random_forest.py
  - NN.py
- All you will have to do is to copy your implementations of the classes "DecisionTree", "RandomForest", "dlnet" onto the respective files. We provided you different .py files and we added libraries in those files please DO NOT remove those lines and add your code after those lines. Note that these are the only allowed libraries that you can use for the homework.
- You are allowed to make as many submissions until the deadline as you like. Additionally, note that the autograder tests each function separately, therefore it can serve as a useful tool to help you debug your code if you are not sure of what part of your implementation might have an issue. However, we encourage you to first debug locally since you will be able to solve most issues in the jupyter notebook.
- **For the "HW4 - Non-programming" part, you will download your jupyter notbook as HTML, print it as a PDF from your browser and submit it on Gradescope. To download the notebook as html, click on "File" on the top left corner of this page and select "Download as > HTML". The non-programming part corresponds to Q5.**

Environment Setup

In [ ]:
```
import numpy as np
from collections import Counter
from scipy import stats
from math import log2, sqrt
import pandas as pd
import time
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.tree import DecisionTreeClassifier

import time
```

Part 1: Utility Functions (25 pts)

Part 1.1: Evaluation Utility Functions

Here, we ask you to develop a few functions that will be the main building blocks of your decision tree and random forest algorithms.

Entropy and information gain [10pts]

First, we define and implement a function that computes entropy of the data.\ Then use this entropy function to compute the information gain for the partitioned data.

Part 1.2: Splitting Utility Functions

Building a decision tree requires us to evaluate the best feature and value to split a node on. Now we will implement functions that help us determine these splits for the dataset.

(1) partition_classes: [5pts]

One of the basic operations is to split a tree on one attribute (features) with a specific value for that attribute.

In partition_classes(), we split the data (X) and labels (y) based on the split feature and value - BINARY SPLIT.

You will have to first check if the split attribute is numerical or categorical. If the split attribute is numeric, split_val should be a numerical value. For example, your split_val should go over all the values of attributes. If the split attribute is categorical, split_val should include all the categories one by one.

You can perform the partition in the following way:

- Numeric Split Attribute:

  Split the data X into two lists(X_left and X_right) where the first list has all the rows where the split attribute is less than or equal to the split value, and the second list has all the rows where the split attribute is greater than the split value. Also create two lists(y_left and y_right) with the corresponding y labels.
- Categorical Split Attribute:

  Split the data X into two lists(X_left and X_right) where the first list has all the rows where the split attribute is equal to the split value, and the second list has all the rows where the split attribute is not equal to the split value. Also create two lists(y_left and y_right) with the corresponding y labels.

Hint: You could find out if the feature is categorical by checking if it is the instance of 'str'

(2) find_best_split [5pts]

Given the data and labels, we need to find the order of splitting features, which is also the importance of the feature. For each attribute (feature), we need to calculate its optimal split value along with the corresponding information gain and then compare with all the features to find the optimal attribute to split.

First, we specify an attribute. After computing the corresponding information gain of each value at this attribute list, we can get the optimal split value, which has the maximum information gain.

(3) find_best_feature [5pts]

Based on the above functions, we can find the most important feature that we will split first.

In [ ]:

```
import numpy as np
from math import log2, sqrt

def entropy(class_y):
    """
    Input:
        - class_y: list of class labels (0's and 1's)

    TODO: Compute the entropy for a list of classes
    Example: entropy([0,0,0,1,1,1,1,1]) = 0.9544
    """
    if np.count_nonzero(class_y) == 0 or np.count_nonzero(class_y) == len(class_y):
        return 0
    else:
        num_ones = np.count_nonzero(class_y)
        num_zeros = len(class_y)-num_ones
        prob_ones = num_ones/(num_ones+num_zeros)
        prob_zeros = num_zeros/(num_ones+num_zeros)
        entropy = -prob_ones*np.log2(prob_ones)-prob_zeros*np.log2(prob_zeros)
        return entropy

def information_gain(previous_y, current_y):
    """
    Inputs:
        - previous_y : the distribution of original labels (0's and 1's)
        - current_y  : the distribution of labels after splitting based on a particular
                     split attribute and split value

    TODO: Compute and return the information gain from partitioning the previous_y labels into the current_y labels.

    Reference: http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15381-s06/www/DTs.pdf

    Example: previous_y = [0,0,0,1,1,1], current_y = [[0,0], [1,1,1,0]], info_gain = 0.4591
    """
    prev_entropy = entropy(previous_y)
    curr_entropy = 0

    for elem in current_y:
        curr_entropy += entropy(elem)*len(elem)/len(previous_y)
    info_gain = prev_entropy - curr_entropy
    return info_gain


def partition_classes(X, y, split_attribute, split_val):
    """
    Inputs:
    - X               : (N,D) list containing all data attributes
    - y               : a list of labels
    - split_attribute : column index of the attribute to split on
    - split_val       : either a numerical or categorical value to divide the split_attribute

    TODO: Partition the data(X) and labels(y) based on the split value - BINARY SPLIT.

    Example:

    X = [[3, 'aa', 10],                 y = [1,
         [1, 'bb', 22],                      1,
         [2, 'cc', 28],                      0,
         [5, 'bb', 32],                      0,
         [4, 'cc', 32]]                      1]

    Here, columns 0 and 2 represent numeric attributes, while column 1 is a categorical attribute.

    Consider the case where we call the function with split_attribute = 0 (the index of attribute) and split_val = 3 (the value o
f attribute).
    Then we divide X into two lists - X_left, where column 0 is <= 3 and X_right, where column 0 is > 3.

    X_left = [[3, 'aa', 10],                y_left = [1,
              [1, 'bb', 22],                          1,
              [2, 'cc', 28]]                          0]

    X_right = [[5, 'bb', 32],               y_right = [0,
               [4, 'cc', 32]]                         1]

    Consider another case where we call the function with split_attribute = 1 and split_val = 'bb'
    Then we divide X into two lists, one where column 1 is 'bb', and the other where it is not 'bb'.

    X_left = [[1, 'bb', 22],                y_left = [1,
              [5, 'bb', 32]]                          0]

    X_right = [[3, 'aa', 10],               y_right = [1,
               [2, 'cc', 28],                         0,
               [4, 'cc', 32]]                         1]


    Return in this order: X_left, X_right, y_left, y_right
    """

    X = np.array(X, dtype = object)
    y = np.array(y)

    split_col = X[:, split_attribute]
    if type(split_val) == int or type(split_val) == float:
```

```
        X_left = X[split_col <= split_val]
        y_left = y[split_col <= split_val]

        X_right = X[split_col > split_val]
        y_right = y[split_col > split_val]

    else:
        X_left = X[split_col == split_val]
        y_left = y[split_col == split_val]

        X_right = X[split_col != split_val]
        y_right = y[split_col != split_val]

    return X_left, X_right, y_left, y_right

def find_best_split(X, y, split_attribute):
    """Inputs:
        - X               : (N,D) list containing all data attributes
        - y               : a list array of labels
        - split_attribute : Column of X on which to split

    TODO: Compute and return the optimal split value for a given attribute, along with the corresponding information gain

    Note: You will need the functions information_gain and partition_classes to write this function

    Example:

        X = [[3, 'aa', 10],                 y = [1,
             [1, 'bb', 22],                      1,
             [2, 'cc', 28],                      0,
             [5, 'bb', 32],                      0,
             [4, 'cc', 32]]                      1]

        split_attribute = 0

        Starting entropy: 0.971

        Calculate information gain at splits:
           split_val = 1  -->  info_gain = 0.17
           split_val = 2  -->  info_gain = 0.01997
           split_val = 3  -->  info_gain = 0.01997
           split_val = 4  -->  info_gain = 0.32
           split_val = 5  -->  info_gain = 0

       best_split_val = 4; info_gain = .32;
    """
    ig = 0
    best_split_val = 0
    column_vals = list(set([r[split_attribute] for r in X]))

    for val in column_vals:
        split_y = []
        X_left, X_right, y_left, y_right = partition_classes(X, y, split_attribute, val)
        split_y.append(y_left)
        split_y.append(y_right)
        ig_curr = information_gain(y,split_y)
        if ig_curr>=ig:
            ig = ig_curr
            best_split_val = val
    return best_split_val,ig

def find_best_feature(X, y):
    """
    Inputs:
        - X: (N,D) list containing all data attributes
        - y : a list of labels

    TODO: Compute and return the optimal attribute to split on and optimal splitting value

    Note: If two features tie, choose one of them at random

    Example:

        X = [[3, 'aa', 10],                 y = [1,
             [1, 'bb', 22],                      1,
             [2, 'cc', 28],                      0,
             [5, 'bb', 32],                      0,
             [4, 'cc', 32]]                      1]

        split_attribute = 0

        Starting entropy: 0.971

        Calculate information gain at splits:
           feature 0:  -->  info_gain = 0.32
           feature 1:  -->  info_gain = 0.17
           feature 2:  -->  info_gain = 0.4199

       best_split_feature: 2 best_split_val: 22
    """

    ig = 0
    best_split_val = 0
    best_split_feature = 0
```

```
    best_ig = 0
    for split_feature in range(len(X[0])):
        split_val,ig = find_best_split(X,y,split_feature)
        if ig>best_ig:
            best_ig = ig
            best_split_val = split_val
            best_split_feature = split_feature
    return best_split_feature,best_split_val
```

In [ ]:
```
X = np.array([[3, 'aa', 10],[1, 'bb', 22],[2, 'cc', 28],[5, 'bb', 32],[4, 'cc', 32]],dtype=object)
y=[1,1,0,0,1]
partition_classes(X,y,1,'bb')
find_best_feature(X,y)
```

Part 2: Decision Tree (20 pts)

Please read the following instructions carefully before you dive into coding

In this part, you will implement your own ID3 decision tree class and make it work on training and test set.

You may use a recursive way to construct the tree and make use of helper functions in Part1.

Please keep in mind that we use information gain to find the best feature and value to split the data for ID3 tree.

To save your training time, we have added a `max_depth` parameter to control the maximum depth of the tree. You may adjust its value to pre-prune the tree. If set to None, it has no control of depth.

You need to have a stop condition for splitting. The stopping condtion is reached when one of the two following conditions are met:

1. If all data points in that node have the same label
2. If the current node is at the maximum depth. In this case, you may assign the mode of the labels as the class label

The MyDecisionTree class should have some member variables. We highly encourage you to use a dict in Python to store the tree information. For leaves nodes, this dict may have just one element representing the class label. For non-leaves node, the list should at least store the feature and value to split, and references to its left and right child.

An example of the dict that you may use for non-leaf nodes:

```
node = {
            'isLeaf': False,
            'split_attribute': split_attribute,
            'split_value': split_val,
            'is_categorical': is_categorical,
            'leftTree': leftTree,
            'rightTree': rightTree
        };
```

In the above example, the leftTree and rightTree are instances of MyDecisonTree itself.

If you use different ways to represent and store the information, please include clear comments or documentations with your code. If your result is not correct, partial credits can only be awarded if we are able to understand your code

In [ ]:

```
import numpy as np
from collections import Counter
from scipy import stats


class MyDecisionTree(object):
    def __init__(self, max_depth=10):
        """
        TODO: Initializing the tree as an empty dictionary, as preferred.
        [5 points]

        For example: self.tree = {}

        Args:

        max_depth: maximum depth of the tree including the root node.
        """
        self.tree = {}
        self.max_depth = max_depth


    def fit(self, X, y, depth):
        """
        TODO: Train the decision tree (self.tree) using the the sample X and labels y.
        [10 points]

        NOTE: You will have to make use of the utility functions to train the tree.
        One possible way of implementing the tree: Each node in self.tree could be in the form of a dictionary:
        https://docs.python.org/2/library/stdtypes.html#mapping-types-dict

        For example, a non-leaf node with two children can have a 'left' key and  a  'right' key.
        You can add more keys which might help in classification (eg. split attribute and split value)


        While fitting a tree to the data, you will need to check to see if the node is a leaf node(
        based on the stopping condition explained above) or not.
        If it is not a leaf node, find the best feature and attribute split:
        X_left, X_right, y_left, y_right, for the data to build the left and
        the right subtrees.

        Remember for building the left subtree, pass only X_left and y_left and for the right subtree,
        pass only X_right and y_right.

        Args:

        X: N*D matrix corresponding to the data points
        Y: N*1 array corresponding to the labels of the data points
        depth: depth of node of the tree
        """
        if len(y) == 0:
            return self
        if depth>=self.max_depth:
            self.tree = {'isLeaf': True,'mode':stats.mode(y).mode[0]}
            return self
        elif np.max(y) == np.min(y):
            self.tree = {'isLeaf': True,'mode':stats.mode(y).mode[0]}
            return self

        most_common_y = stats.mode(y).mode[0]
        best_split_feature,best_split_val = find_best_feature(X,y)

        if type(best_split_val) == str:
            is_categorical = True
        else:
            is_categorical = False

        X_left, X_right, y_left, y_right = partition_classes(X,y,best_split_feature,best_split_val)


        self.tree = {'isLeaf': False,'split_attribute': best_split_feature,\
                'split_value': best_split_val,'mode': most_common_y,'is_categorical':is_categorical,\
                     'left':MyDecisionTree(max_depth=self.max_depth).fit(X_left,y_left,depth+1),\
                    'right':MyDecisionTree(max_depth=self.max_depth).fit(X_right,y_right,depth+1)}
        return self

    def predict(self, record):
        """
        TODO: classify a sample in test data set using self.tree and return the predicted label
        [5 points]
        Args:

        record: D*1, a single data point that should be classified

        Returns: True if the predicted class label is 1, False otherwise
        """

        curr_layer = self.tree
        while (curr_layer['isLeaf'] == False):
            if curr_layer['is_categorical']:
                if record[curr_layer['split_attribute']] == curr_layer['split_value']:
                    curr_layer = curr_layer['left'].tree
                elif record[curr_layer['split_attribute']] != curr_layer['split_value']:
                    curr_layer = curr_layer['right'].tree
```

```
            elif not(curr_layer['is_categorical']):
                if record[curr_layer['split_attribute']] <= curr_layer['split_value']:
                    curr_layer = curr_layer['left'].tree
                elif record[curr_layer['split_attribute']] > curr_layer['split_value']:
                    curr_layer = curr_layer['right'].tree
        if curr_layer['mode'] == 1:
            return True
        else:
            return False

    # helper function. You don't have to modify it
    def DecisionTreeEvalution(self,X,y, verbose=False):
        # Make predictions
        # For each test sample X, use our fitting dt classifer to predict
        y_predicted = []
        for record in X:
            y_predicted.append(self.predict(record))

        # Comparing predicted and true labels
        results = [prediction == truth for prediction, truth in zip(y_predicted, y)]

        # Accuracy
        accuracy = float(results.count(True)) / float(len(results))
        if verbose:
            print("accuracy: %.4f" % accuracy)
        return accuracy

    def DecisionTreeError(self, y):
        # helper function for calculating the error of the entire subtree if converted to a leaf with majority class label.
        # You don't have to modify it
        num_ones = np.sum(y)
        num_zeros = len(y) - num_ones
        return 1.0 - max(num_ones, num_zeros) / float(len(y))

    #  Define the post-pruning function
    def pruning(self, X, y):
        """
        TODO:
        1. Prune the full grown decision trees recursively in a bottom up manner.
        2. Classify examples in validation set.
        3. For each node:
        3.1 Sum errors over the entire subtree. You may want to use the helper function "DecisionTreeEvalution".
        3.2 Calculate the error on same example if converted to a leaf with majority class label.
        You may want to use the helper function "DecisionTreeError".
        4. If error rate in the subtree is greater than in the single leaf, replace the whole subtree by a leaf node.
        5. Return the pruned decision tree.
        """
        if self.tree['isLeaf'] or len(y)==0:
            return self
        else:
            X_left, X_right, y_left, y_right = partition_classes(X, y, self.tree['split_attribute'], self.tree['split_value'])
            self.tree['left'] = self.tree['left'].pruning(X_left,y_left)
            self.tree['right'] = self.tree['right'].pruning(X_right,y_right)
            most_common_y = stats.mode(y).mode[0]
            if (1-self.DecisionTreeEvalution(X,y))>self.DecisionTreeError(y):
                self.tree = {'isLeaf':'yes','mode':most_common_y}
        return self
```

Dataset Objective

We are the founders of a new e-commerce company that uses machine learning to optimize the user experience. We are tasked with the responsibility of coming up with a method for determining the likelihood of a shopping session ending in a purchase being made. We will then use this information to adjust pricing and services to encourage more purchasing.

After much deliberation amongst the team, you come to a conclusion that we can use past online shopping data to predict the future occurence of revenue sessions.

Our task is to use the decision tree algorithm to predict if a shopping session ends in a purchase.

You can find more information on the dataset here (https://archive.ics.uci.edu/ml/datasets/Online+Shoppers+Purchasing+Intention+Dataset#).

Loading the dataset

The dataset that the company has collected has the following features:

1. Administrative : continuous variable
2. Administrative_Duration : continuous variable
3. Informational : continuous variable
4. Informational_Duration : continuous variable
5. ProductRelated : continuous variable
6. ProductRelated_Duration : continuous variable
7. BounceRates : continuous variable
8. ExitRates : continuous variable
9. PageValues : continuous variable
10. SpecialDay : continuous variable
11. Month : categorical variable
12. OperatingSystems : continuous variable
13. Browser : continuous variable
14. Region : continuous variable
15. TrafficType : continuous variable
16. VisitorType : categorical variable
17. Weekend : continuous variable
18. Revenue : target variable

Splitting the Dataset

The original dataset explained above was split into four separate datasets. You are provided only three of these. The fourth is hidden and will be used to test your implementations via the gradescope autograder.

Training Data: For training decision tree and random forest algorithms in parts 2 and 4.

Validation Data: For pruning your decision tree in part 3. (optional)

Testing Data: For testing your decision tree and random forest algorithms in parts 2 and 4

Hidden Data: This data will be left out and will instead be used to grade your imlementations on gradescope.

In [ ]:
```
# helper function. You don't have to modify it
data_test = pd.read_csv("hw4_summer2020_data_test.csv")
data_valid = pd.read_csv("hw4_summer2020_data_valid.csv")
data_train = pd.read_csv("hw4_summer2020_data_train.csv")
data_hidden = pd.read_csv("hw4_summer2020_data_hidden.csv")

categorical = ['Month']
numerical = ['Administrative','Administrative_Duration','Informational',
             'Informational_Duration','ProductRelated','ProductRelated_Duration','BounceRates','ExitRates','PageValues','SpecialD
ay','Month','OperatingSystems','Browser','Region','TrafficType','VisitorType','Weekend']

X_train = data_train.drop(columns = 'Revenue')
y_train = data_train['Revenue']
X_test = data_test.drop(columns = 'Revenue')
y_test = data_test['Revenue']
X_train, y_train, X_test, y_test = np.array(X_train), np.array(y_train), np.array(X_test), np.array(y_test)

X_valid = data_valid.drop(columns = 'Revenue')
y_valid = data_valid['Revenue']
X_valid, y_valid = np.array(X_valid), np.array(y_valid)

X_hidden = data_hidden.drop(columns = 'Revenue')
y_hidden = data_hidden['Revenue']
X_hidden, y_hidden = np.array(X_hidden), np.array(y_hidden)
```

Let us train and evaluate the performance of our decision tree on the test set. Note that it is trivially possible to achieve 84% accuracy because of the distribution of "revenue" shopping sessions in the dataset. You can use the provided test set to evaluate your implementation, however, your implementation will be tested using a left out hidden test set. You will need to obtain 87% on the hidden test set to receive full credit. Change the default parameters in your MyDecisionTree class to be the ones that you would like to be used in grading.

In [ ]:
```
### Initializing a decision tree.
t1 = time.time()
dt = MyDecisionTree(max_depth=10)

# Building a tree
print("fitting the decision tree")
dt.fit(X_train, y_train, 0)

# Evaluating the decision tree
dt.DecisionTreeEvalution(X_test,y_test, True)
t2 = time.time()
print(t2-t1)
```

Part 3

Pruning [10 Pts] [Bonus for undergrads]

In order to avoid overfitting, you can:

1. Acquire more training data;
2. Remove irrelevant attributes;
3. Grow full tree, then post-prune;
4. Ensemble learning.

In this part, you are going to apply reduced error post-pruning to prune the fully grown tree in a bottom-up manner. The idea is basically about, starting at the leaves, each node is replaced with its most popular class. If the prediction accuracy is not affected then the change is kept. You may also try recursive function to apply the post-pruning. Please notice we use validation set to get the accuracy for each node during the pruning

Now, you should make use of the decision tree you trained in part1. You can use the provided test set to evaluate your implementation, however, your implementation will be tested using a left out hidden test set. You will receive full credit if your decision tree is more accurate when pruned and you achieve at least 88% accuracy.

In [ ]:
```
# helper function. You don't have to modify it.
# pruning the full grown decision tree using validation set
# dt should be a decision tree object that has been fully trained
dt_pruned=dt.pruning(X_valid, y_valid)

# Evaluate the decision tree using test set

dt_pruned.DecisionTreeEvalution(X_test, y_test, False)
```

Part 4: Random Forests [35pts]

The decision boundaries drawn by decision trees are very sharp, and fitting a decision tree of unbounded depth to a list of examples almost inevitably leads to **overfitting**. In an attempt to decrease the variance of our classifier we're going to use a technique called 'Bootstrap Aggregating' (often abbreviated 'bagging'). This stems from the idea that a collection of weak learners can learn decision boundaries as well as a strong learner.

Part 4.1 Random Forest Implementation (30 pts)

A Random Forest is a collection of decision trees, built as follows:

1) For every tree we're going to build:

```
a) Subsample the examples with replacement. Note that in this question, the size of the subsample data is equal to the original datase
t.

b) From the subsamples in a), choose attributes at random to learn on in accordance with a provided attribute subsampling rate. Based
 on what it was mentioned in the class, we randomly pick features in each split. We use a more general approach here to make the progr
amming part easier. Let's randomly pick some features (70% percent of features) and grow the tree based on the pre-determined randomly
selected features. Therefore, there is no need to find random features in each split.

c) Fit a decision tree to the subsample of data we've chosen to a certain depth.
```

Classification for a random forest is then done by taking a majority vote of the classifications yielded by each tree in the forest after it classifies an example.

We will be using the Out of Bag (OOB) score to test our random forest implementations. Since we are subsampling the datapoints with replacement, each individual decision tree in the random forests has some samples that were not used to train the tree. These samples are called "out of bag." We first calculate the accuracy of each decision tree as the percentage of OOB samples that are correctly classified. Then we take the average of these accuracies to obtain the random forest's OOB score.

In RandomForests Class,

1. X is assumed to be a matrix with num_training rows and num_features columns where num_training is the number of total records and num_features is the number of features of each record.
2. y is assumed to be a vector of labels of length num_training.

NOTE: Lookout for TODOs for the parts that needs to be implemented.

In [ ]:
```
import numpy as np

"""
NOTE: For graduate student, you are required to use your own decision tree MyDecisionTree() to finish random forest.
      Undergraduate students may use the decision tree library from sklearn.
"""

class RandomForest(object):
    def __init__(self, n_estimators=25, max_depth=6, max_features=0.98):
        # helper function. You don't have to modify it
        # Initialization done here
        self.n_estimators = n_estimators
        self.max_depth = max_depth
        self.max_features = max_features
        self.bootstraps_row_indices = []
        self.feature_indices = []
        self.out_of_bag = []
        self.decision_trees = [MyDecisionTree(max_depth=max_depth) for i in range(n_estimators)]

    def _bootstrapping(self, num_training, num_features, random_seed = None):
        """
        TODO:
        - Randomly select a sample dataset of size num_training with replacement from the original dataset.
        - Randomly select certain number of features (num_features denotes the total number of features in X,
          max_features denotes the percentage of features that are used to fit each decision tree) without replacement from the t
otal number of features.

        Return:
        - row_idx: the row indices corresponding to the row locations of the selected samples in the original dataset.
        - col_idx: the column indices corresponding to the column locations of the selected features in the original feature lis
t.

        Reference: https://en.wikipedia.org/wiki/Bootstrapping_(statistics)
        """
        np.random.seed(seed=random_seed)
        all_rows_ind = np.arange(num_training)
        row_idx = np.random.choice(all_rows_ind,size=int(1*len(all_rows_ind)),replace=True)

        all_ft_ind = np.arange(num_features)
        col_idx = np.random.choice(all_ft_ind,size=int(self.max_features*len(all_ft_ind)),replace=False)

        return row_idx,col_idx

    def bootstrapping(self, num_training, num_features):
        # helper function. You don't have to modify it
        # Initializing the bootstap datasets for each tree
        for i in range(self.n_estimators):
            total = set(list(range(num_training)))
            row_idx, col_idx = self._bootstrapping(num_training, num_features,random_seed=5)
            total = total - set(row_idx)
            self.bootstraps_row_indices.append(row_idx)
            self.feature_indices.append(col_idx)
            self.out_of_bag.append(total)

    def fit(self, X, y):
        """
        TODO:
        Train decision trees using the bootstrapped datasets.
        Note that you need to use the row indices and column indices.
        """
        self.bootstrapping(len(X),len(X[0]))
        for i in range(self.n_estimators):
            print(i,' estimators fitted')
            train_subset_X = X[self.bootstraps_row_indices[i]][:,self.feature_indices[i]]
            train_subset_y = y[self.bootstraps_row_indices[i]]
            self.decision_trees[i].fit(train_subset_X,train_subset_y,depth=0)

    def OOB_score(self, X, y):
        # helper function. You don't have to modify it
        accuracy = []
        for i in range(len(X)):
            predictions = []
            for t in range(self.n_estimators):
                if i in self.out_of_bag[t]:
                    predictions.append(self.decision_trees[t].predict(X[i][self.feature_indices[t]]))
            if len(predictions) > 0:
                accuracy.append(np.sum(predictions == y[i]) / float(len(predictions)))
        return np.mean(accuracy)
```

Part 4.2 Hyperparameter tuning(5pts)

Change the hyperparamters below to obtain atleast a 89% accuracy on the hidden test dataset. Change the default parameters in your RandomForest class to be the ones that you would like to be used in grading.

In [ ]:
```
"""
TODO:
n_estimators defines how many decision trees are fitted for the random forest (at least 10).
max_depth defines a stop condition when the tree reaches to a certain depth.
max_features controls the percentage of features that are used to fit each decision tree.
Tune these three parameters to achieve a better accuracy. You will need to obtain 89% on the
hidden test set to receive full credit. You can use the provided test set to evaluate your implementation.
The random forest fitting may take 5 - 15 minutes. We will not take running time into account when grading
this part, however, you need to make sure that the gradescope autograder does not time out.
"""
n_estimators = 25
max_depth = 6
max_features = 0.98

start = time.time()
random_forest = RandomForest(n_estimators, max_depth, max_features)

random_forest.fit(X_train, y_train)

accuracy=random_forest.OOB_score(X_test, y_test)
print("accuracy: %.4f" % accuracy)

end = time.time()
print(end-start)
```

Part 5: SVM (35 Pts)

5.1 Fitting an SVM classifier by hand (20 Pts)

Consider a dataset with 2 points in 1-dimensional space: $(x_1 = -2, y_1 = -1)$ and $(x_2 = 3, y_2 = 1)$. Here x are the point coordinates and y are the classes.

Consider mapping each point to 3-dimensional space using the feature vector $\phi(x) = [1, 2x, x^2]$. (This is equivalent to using a second order polynomial kernel.) The max margin classifier has the form

$$min||\theta||^2 s.t.$$

$$y_1(\phi(x_1)\theta + b) \geq 1$$

$$y_2(\phi(x_2)\theta + b) \geq 1$$

Hint: $\phi(x_1)$ and $\phi(x_2)$ are the suppport vectors. We have already given you the solution for the suppport vectors and you need to calculate back the parameters. Margin is equal to $\frac{1}{||\theta||}$ and full margin is equal to $\frac{2}{||\theta||}$.

(1) Find a vector parallel to the optimal vector θ. (4pts)

(2) Calculate the value of the margin achieved by this θ? (4pts)

(3) Solve for θ, given that the margin is equal to $1/||\theta||$. (4pts)

(4) Solve for b using your value for θ. (4pts)

(5) Write down the form of the discriminant function $f(x) = \phi(x)\theta + b$ as an explicit function of x. (4pts)

Solution:

When mapped to the 3-dimensional space, the points become:

$$p_1 \equiv [-2, -1] \Rightarrow \phi(x_1) = [1, -4, 4]$$

$$p_2 \equiv [3, 1] \Rightarrow \phi(x_2) = [1, 6, 9]$$

(1) Now, a vector parallel to the optimal vector θ is parallel to the vector joining the two points. One such vector may be $\phi(x_2) - \phi(x_1) = (0, 10, 5)$.

(2) The value of the margin achieved by this θ is given by

$$\gamma = \frac{1}{2} ||\phi(x_2) - \phi(x_1)|| = \frac{5\sqrt{5}}{2}$$

(3) If the optimal vector is given by $[\theta_1, \theta_2, \theta_3]$,

$$\theta_1^2 + \theta_2^2 + \theta_3^2 = \frac{1}{\gamma} = \frac{2}{5\sqrt{5}}$$

Given that this vector is parallel to the vector in (1), we get the following constraints:

$$\theta_1 = 0$$

$$\theta_2 = 2\theta_3$$

Plugging the constraints into the above equation, we get

$$5\theta_3^2 = \frac{4}{125} \Rightarrow \theta_3 = 0.08, \theta_2 = 0.16$$

So, we get the optimal vector set as

$$[\theta_1, \theta_2, \theta_3] \equiv [0, 0.16, 0.08]$$

(4) The constraint of the maximum margin classifier is given by

$$y_1(\phi(x_1)\theta + b) \geq 1$$

$$y_2(\phi(x_2)\theta + b) \geq 1$$

Plugging in the results into the above inequalities, we get

$$-1 \times ([0, 0.16, 0.08] \cdot [1, -4, 4] + b) \geq 1$$

$$\Rightarrow 0.32 - b \geq 1$$

$$1 \times ([0, 0.16, 0.08] \cdot [1, 6, 9] + b) \geq 1$$

$$\Rightarrow 1.68 + b \geq 1$$

Solving the above set of inequalities, we find that $b = \frac{-17}{25}$.

(5) Expressing the function in the required form, we get

$$f(x) = [1, 2x, x^2] \cdot [0, 0.16, 0.08] - \frac{17}{25}$$

$$\Rightarrow f(x) = 0.32x + 0.08x^2 - \frac{17}{25}$$

5.2 SVM Kernel (15 Pts)

(1) (5 points) We know that SVM can be used to classify linearly inseparable data by transforming it to a different feature space with a kernel $K(x,z) = \phi(x)^T\phi(z)$, where $\phi(x)$ is a feature mapping. Let K_1 and K_2 be $R^n \times R^n$ kernels, $c \in R^+$ be a positive constant., and $\phi_1, \phi_2 : R^n \rightarrow R^d$ be the respective feature mappings of K_1 and K_2. Explain how to use ϕ_1, ϕ_2 to obtain the following kernels:

a. $K(x,z) = cK_1(x,z)$

b. $K(x,z) = K_1(x,z)K_2(x,z)$

(2) (10 points)

a. Consider the polynomial kernel

$$K(x,z) = (x^Tz + 1)^d$$

with d=2. Let $x, z \in R$ for simplicity. Define one calculation as one multiplication, addition or square operation. Assume that constants (like $\sqrt{2}$) are already calculated and given. What is the number of calculations required to find $K(x,z)$ through direct computation?

b. Can you find the corresponding feature mapping $\phi(x)$?

c. What is the number of calculations required for calculating the above feature map for a scalar x ?

d. What is the number of calculations to find $K(x,z)$ using $\phi(x)^T\phi(z)$? Comment on this with respect to your answer in (a).

e. Consider the Radial Basis Kernel

$$K(x,z) = exp(-\frac{||x-z||^2}{2\sigma^2})$$

. Is it possible to find a feature map in this case? Do you think it's necessary that an explicit feature map exists with all kernels?

Solution:

(1) a. The kernel when multiplied by the constant as $K(x,z) = cK_1(x,z)$ is given by the expression $\phi(x) = \sqrt{c}\phi_1(x)$.

b. The kernel of the product $K_1(x,z)K_2(x,z)$ is given by the expression $\phi(x) = \phi_1(x)\phi_2(x)$.

(2) a. The kernel provided may be expressed as $K(x,z) = (x^Tz + 1)^2$. In the two-dimensional space, the kernel may be expressed as

$$K(x,z) = (x_1z_1 + 1)^2 = (x_1z_1)^2 + 1 + 2x_1z_1$$

Clearly, the above expression has 3 calculations in it.

b. The corresponding feature map is given by:

$$\phi(x) = [x_1^2, \sqrt{2}x_1, 1]$$

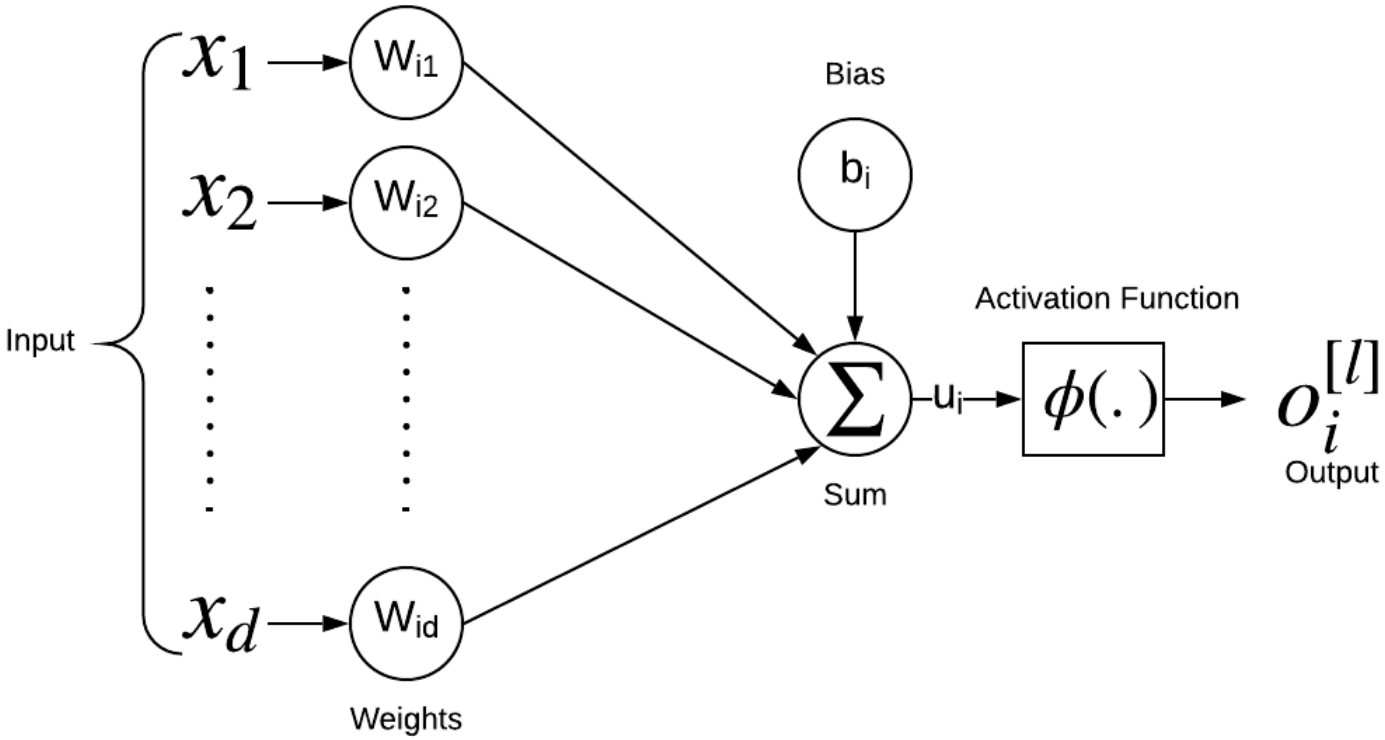
c. Clearly, the above feature mapping requires 3 calculations.

d. Since this computation requires a dot product of two 3-dimensional vectors, 9 calculations are required to find $K(x,z)$ in terms of the product of the feature mappings.

e. For the RBF kernel, the feature map is infinite-dimensional (i.e. may be expressed in the form of a Tayolr series). As such, an explicit finite-dimensional feature map need not exist for all kernels.

Part 6: Two Layer Neural Network

Perceptron



A single perceptron can be thought of as a linear hyperplane as in SVMs followed by a non-linear function.

$$u_i = \phi\left(\sum_{j=1}^{n} w_{ij}x_j + b_i\right) = \phi(w_i^T x + b_i)$$

where $w_i \in R^n$ is the weight vector, $x \in R^n$ is ONE data point with n features, $b_i \in R$ is the bias element, and $\phi(.)$ is any non linear function that will be described later.

Fully-connected Layer

Typically, a modern neural network contains millions of perceprons as the one shown in the previous image. Preceptrons interact in different configurations. Such as cascaded or parallel. In this part we decribe a fully connected layer configuration in a neural network which comprises multiple parallel perceptrons forming one layer.

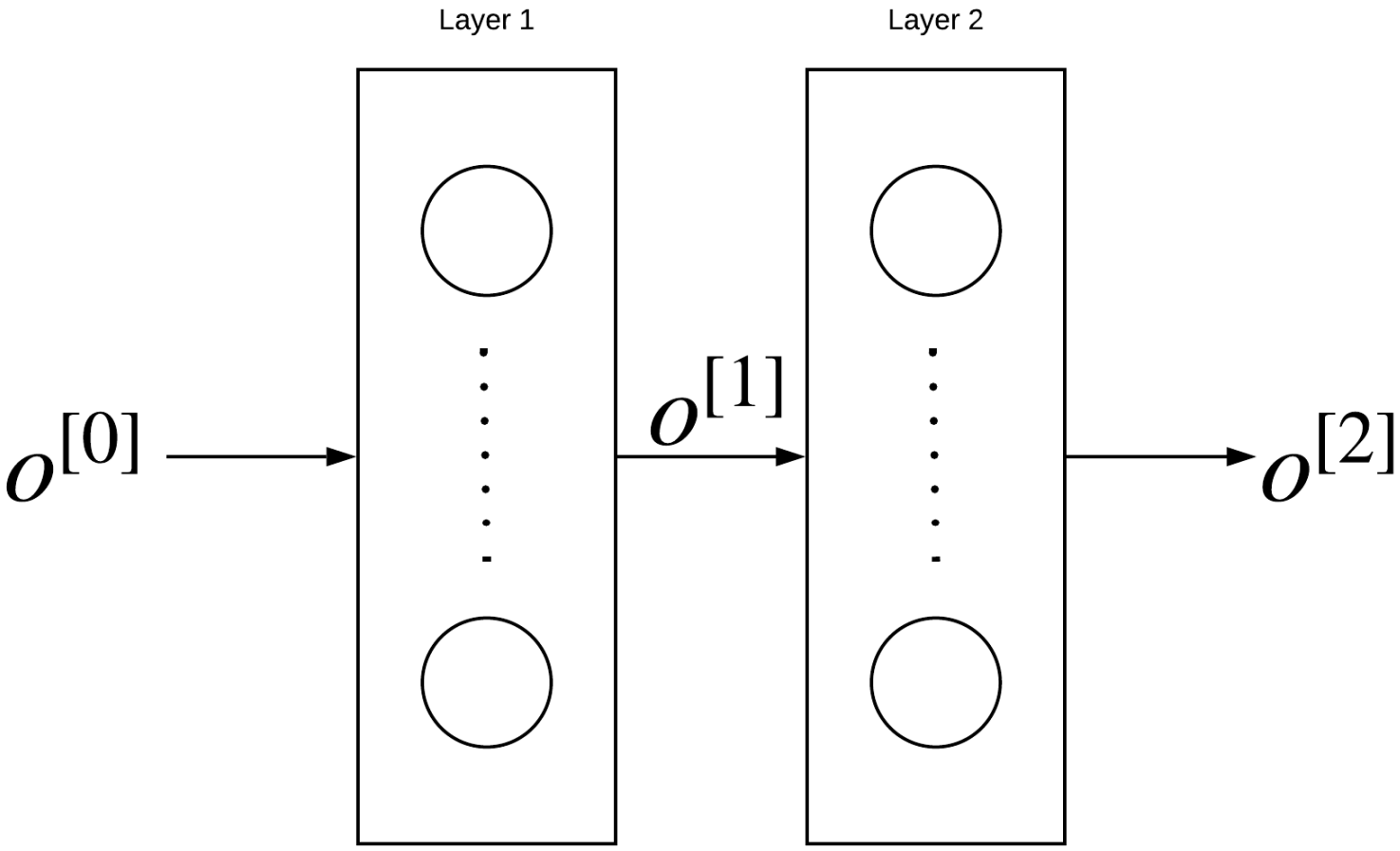
We extend the previous notation to describe a fully connected layer as follows

$$o = Wx + b$$

where $o \in R^m$ is the output vector after appling linear operations, $W \in R^{m \times d}$ is the weight matrix, and $b \in R^m$ is the bais vector. This is followed by element wise non-linear function $u^{[l]} = \phi(o)$. The whole operation can be summarized as,

$$u^{[l]} = \phi(Wu^{[l-1]} + b)$$

where $u^{[l-1]}$ is the output of the previous layer as shown in figure.

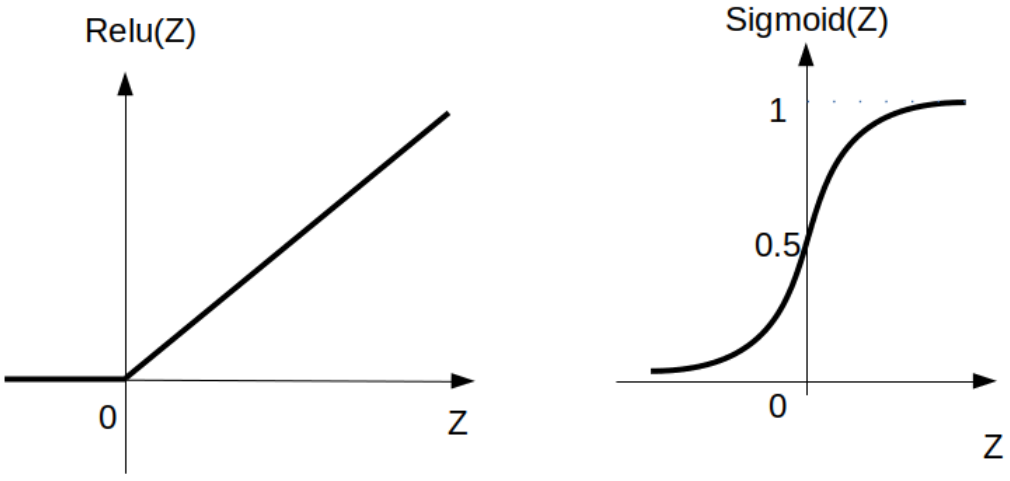


Activation Function

There are many activation functions in the literature, but for this question we are going to use Relu and Sigmoid only.

Relu

The rectified linear unit (Relu) is one of the most commonly used activation function in deep learning models. The mathematical form is $\phi(o) = max(0, o)$ while the graph form is



Sigmoid

The sigmoid function is another non-linear function with S-shaped curve. This function is useful in the case of binary classification as its output is between 0 and 1. The mathematical form of the function is $\phi(o) = \frac{1}{1+e^{-o}}$ while the graphical form is shown in the above figure.

Cross Entropy Loss

An essential piece in training a neural network, is the loss function. The whole purpose of gradient decent algorithm, is to find some network parameters that minimizes the loss function. In this excercise we minimize Cross Entropy (CE) loss, that represents on an intutive level, the distance between true data distribution and estimated distribution by neural network. So during training of the neural network, we will be looking for network parameters that minimizes the distance between true and estimated distribution. The mathematical form of the CE loss is given by

$$CE(p, q) = -\sum_i p(x_i)\log q(x_i)$$

where $p(x)$ is the true distribution and $q(x)$ is the estimated distribution.

Implementation details

For binary classification problems as in this exercise, we have probability distribution of a label y_i is given by

$$y_i = \begin{cases} 1 & \text{with probability } p(x_i) \\ 0 & \text{with probability } 1 - p(x_i) \end{cases}$$

A frequentist estimate of $p(x_i)$ can be written as $p(x_i) = \sum_{i=1}^{N} \frac{y_i}{N}$, therefore the cross entropy for binary estimation can be written as

$$CE(y_i, \hat{y_i}) = -\frac{1}{N}\sum_{i=1}^{N} y_i \log(\hat{y_i}) + (1 - y_i)\log(1 - \hat{y_i})$$

where $y_i \in \{0, 1\}$ is the true label and $\hat{y_i} \in [0, 1]$ is the estimated distribution.

Forward Propagation

We start by intializing the weights of the fully connected layer using Xavier initialization Xavier initialization (http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf). During training we pass all the data points throught the network layer by layer.

$$\begin{aligned} u^{[0]} &= x \\ o^{[1]} &= W^{[1]}u^{[0]} + b^{[1]} \\ u^{[1]} &= Relu(o^{[1]}) \\ o^{[2]} &= W^{[2]}u^{[1]} + b^{[2]} \\ \hat{y} = u^{[2]} &= Sigmoid(o^{[2]}) \end{aligned}$$

Then we get the output and compute the loss

$$l = y^T.\log(\hat{y})$$

Backward propagation

After the forward pass, we do back propagation to update the weights and biases in the direction of the negative gradient of the loss function. So we update the weights and biases using the following formulas

$$W^{[2]} := W^{[2]} - lr \times \frac{\partial l}{\partial W^{[2]}} \quad b^{[2]} := b^{[2]} - lr \times \frac{\partial l}{\partial b^{[2]}} \quad W^{[1]} := W^{[1]} - lr \times \frac{\partial l}{\partial W^{[1]}} \quad b^{[1]} := b^{[1]} - lr \times \frac{\partial l}{\partial b^{[1]}}$$

where lr is the learning rate.

To compute the terms $\frac{\partial l}{\partial W^{[i]}}$ and $\frac{\partial l}{\partial b^{[i]}}$ we use chain rule for differentiation as follows

$$\begin{aligned} \frac{\partial l}{\partial W^{[2]}} &= \frac{\partial l}{\partial u^{[2]}}\frac{\partial u^{[2]}}{\partial o^{[2]}}\frac{\partial o^{[2]}}{\partial W^{[2]}} \\ \frac{\partial l}{\partial b^{[2]}} &= \frac{\partial l}{\partial u^{[2]}}\frac{\partial u^{[2]}}{\partial o^{[2]}}\frac{\partial o^{[2]}}{\partial b^{[2]}} \end{aligned}$$

So $\frac{\partial l}{\partial u^{[2]}}$ is the differentiation of the cross entropy function at point $u^{[2]}$, $\frac{\partial u^{[2]}}{\partial o^{[2]}}$ is the differentiation of the Sigmoid function at point $o^{[2]}$, $\frac{\partial o^{[2]}}{\partial W^{[2]}}$ is equal to $u^{[1]}$, and $\frac{\partial o^{[2]}}{\partial b^{[2]}}$ is equal to 1. To compute $\frac{\partial l}{\partial W^{[2]}}$, we need $u^{[2]}, o^{[2]} \& u^{[1]}$ which are calculated during forward propagation. So we need to store these values in a cache variable during the forward propagation to be able to access them during bacward propagation. Also, the functional form of the CE differentiation and Sigmoid differentiation are given by

$$\begin{aligned} \frac{\partial l}{\partial u^{[2]}} &= \frac{-1}{N}\left(\frac{y}{u^{[2]}} - \frac{1-y}{1-u^{[2]}}\right) \\ \frac{\partial u^{[2]}}{\partial o^{[2]}} &= \frac{1}{1+e^{-o^{[2]}}}\left(1 - \frac{1}{1+e^{-o^{[2]}}}\right) \\ \frac{\partial o^{[2]}}{\partial W^{[2]}} &= u^{[1]} \\ \frac{\partial o^{[2]}}{\partial b^{[2]}} &= 1 \end{aligned}$$

While

$$\frac{\partial l}{\partial W^{[1]}} = \frac{\partial l}{\partial u^{[2]}} \frac{\partial u^{[2]}}{\partial o^{[2]}} \frac{\partial o^{[2]}}{\partial u^{[1]}} \frac{u^{[1]}}{o^{[1]}} \frac{o^{[1]}}{W^{[1]}}$$

$$\frac{\partial l}{\partial b^{[1]}} = \frac{\partial l}{\partial u^{[2]}} \frac{\partial u^{[2]}}{\partial o^{[2]}} \frac{\partial o^{[2]}}{\partial u^{[1]}} \frac{u^{[1]}}{o^{[1]}} \frac{o^{[1]}}{b^{[1]}}$$

Where

$$\frac{\partial o^{[2]}}{\partial u^{[1]}} = W^{[1]}$$

$$\frac{\partial u^{[1]}}{\partial o^{[1]}} = \begin{cases} 0 & \text{if } o^{[1]} \leq 0 \\ 1 & \text{if } o^{[1]} > 0 \end{cases}$$

$$\frac{\partial o^{[1]}}{\partial W^{[1]}} = x$$

$$\frac{\partial o^{[1]}}{\partial b^{[1]}} = 1$$

Note that $\frac{\partial u^{[1]}}{\partial o^{[1]}}$ is the differentiation of the Relu function at $o^{[1]}$.

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_breast_cancer
from sklearn.preprocessing import MinMaxScaler

from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from sklearn.metrics import plot_confusion_matrix
```

In [ ]:

```
'''
We are going to use Breast Cancer Wisconsin (Diagnostic) Data Set provided by sklearn
https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_breast_cancer.html
to train a 2 fully connected layer neural net. We are going to buld the neural network from scratch.
'''

class dlnet:

    def __init__(self, x, y, lr=0.003):
        '''
        This method initializes the class, its implemented for you.
        Args:
            x: data
            y: labels
            Yh: predicted labels
            dims: dimensions of different layers
            param: dictionary of different layers parameters
            ch: Cache dictionary to store forward parameters that are used in backpropagation
            loss: list to store loss values
            lr: learning rate
            sam: number of training samples we have

        '''
        self.X = x  # features
        self.Y = y  # ground truth labels

        self.Yh = np.zeros((1, self.Y.shape[1]))  # estimated labels
        self.dims = [30, 15, 1]  # dimensions of different layers

        self.param = {}  # dictionary for different layer variables
        self.ch = {}  # cache variable
        self.loss = []

        self.lr = lr  # learning rate
        self.sam = self.Y.shape[1]  # number of training samples we have
        self._estimator_type = 'classifier'

    def nInit(self):
        '''
        This method initializes the neural network variables, its already implemented for you.
        Check it and relate to mathematical the description above.
        You are going to use these variables in forward and backward propagation.
        '''
        np.random.seed(1)
        self.param['W1'] = np.random.randn(self.dims[1], self.dims[0]) / np.sqrt(self.dims[0])
        self.param['b1'] = np.zeros((self.dims[1], 1))
        self.param['W2'] = np.random.randn(self.dims[2], self.dims[1]) / np.sqrt(self.dims[1])
        self.param['b2'] = np.zeros((self.dims[2], 1))
        return

    def Relu(self, x):
        '''
        In this method you are going to implement element wise Relu.
        Make sure that all operations here are element wise and can be applied to an input of any dimension.
        Input: Z of any dimension
        return: Relu(Z)
        '''
        relu = np.maximum(0, x)
        return(relu)


    def Sigmoid(self, x):
        '''
        In this method you are going to implement element wise Sigmoid.
        Make sure that all operations here are element wise and can be applied to an input of any dimension.
        Input: Z of any dimension
        return: Sigmoid(Z)
        '''
        sigmoid = 1/(1+np.exp(-Z))
        return sigmoid


    def dRelu(self, x):
        '''
        In this method you are going to implement element wise differentiation of Relu.
        Make sure that all operations here are element wise and can be applied to an input of any dimension.
        Input: Z of any dimension
        return: dRelu(Z)
        '''
        x[x<=0] = 0
        x[x>0] = 1
        return x

    def dSigmoid(self, x):
        '''
        In this method you are going to implement element wise differentiation of Sigmoid.
        Make sure that all operations here are element wise and can be applied to an input of any dimension.
        Input: Z of any dimension
        return: dSigmoid(Z)
        '''
        sigm = self.Sigmoid(x)
        dSig = sigm*(1-sigm)
        return dSig
```

```
def nloss(self, y, yh):
    '''
    In this method you are going to implement Cross Entropy loss.
    Refer to the description above and implement the appropriate mathematical equation.
    Input: y 1xN: ground truth labels
           yh 1xN: neural network output after Sigmoid

    return: CE 1x1: loss value
    '''
    #  Delete this line when you implement the function
    nloss = -np.sum(y*np.log(yh+1e-8))/y.shape[1]
    return nloss

def forward(self, x):
    '''
    Fill in the missing code lines, please refer to the description for more details.
    Check nInit method and use variables from there as well as other implemeted methods.
    Refer to the description above and implement the appropriate mathematical equations.
    donot change the lines followed by #keep.
    '''
    #Todo: uncomment the following 7 lines and complete the missing code
    #u1 =
    #o1 =
    #self.ch['u1'], self.ch['o1'] = u1, o1  # keep
    #u2 =
    #o2 =
    #self.ch['u2'], self.ch['o2'] = u2, o2  # keep
    #return A2  # keep

    #  Delete this line when you implement the function
    raise NotImplementedError



def backward(self, y, yh):
    '''
    Fill in the missing code lines, please refer to the description for more details
    You will need to use cache variables, some of the implemeted methods, and other variables as well
    Refer to the description above and implement the appropriate mathematical equations.
    donot change the lines followed by #keep.
    '''
    #Todo: uncomment the following 13 lines and complete the missing code

    # dLoss_o2 =
    # dLoss_u2 =
    # dLoss_W2 =
    # dLoss_b2 =
    # dLoss_o1 =
    # dLoss_u1 =
    # dLoss_W1 =
    # dLoss_b1 =
    # self.param["W2"] = self.param["W2"] - self.lr * dLoss_W2  # keep
    # self.param["b2"] = self.param["b2"] - self.lr * dLoss_b2  # keep
    # self.param["W1"] = self.param["W1"] - self.lr * dLoss_W1  # keep
    # self.param["b1"] = self.param["b1"] - self.lr * dLoss_b1  # keep
    #return dLoss_W2, dLoss_b2, dLoss_W1, dLoss_b1 #keep


    #  Delete this line when you implement the function
    raise NotImplementedError


def gradient_decent(self, x, y, iter=60000):
    '''
    This function is an implementation of the gradient decent algorithm,
    Its implemented for you.
    '''
    self.nInit()
    for i in range(0, iter):
        yh = self.forward(x)
        loss = self.nloss(y, yh)
        dLoss_W2, dLoss_b2, dLoss_W1, dLoss_b1 = self.backward(y, yh)
        self.loss.append(loss)
        if i % 2000 == 0: print("Loss after iteration %i: %f" % (i, loss))
    return

def predict(self, x):
    '''
    This function predicts new data points
    Its implemented for you
    '''
    Yh = self.forward(x)
    return np.round(Yh).squeeze()
```

In [ ]:
```
'''
Training the Neural Network, you donot need to modify this cell
We are going to use Breast Cancer Wisconsin (Diagnostic) Data Set provided by sklearn
https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_breast_cancer.html
'''
dataset = load_breast_cancer() # load the dataset
x, y = dataset.data, dataset.target
x = MinMaxScaler().fit_transform(x) #normalize data
x_train, x_test, y_train, y_test = train_test_split(x, y, random_state=1) #split data
x_train, x_test, y_train, y_test = x_train.T, x_test.T, y_train.reshape(1,-1), y_test #condition data

nn = dlnet(x_train,y_train,lr=0.1) # initalize neural net class
nn.gradient_decent(x_train, y_train, iter = 66000) #train

# create figure
fig = plt.plot(np.array(nn.loss).squeeze())
plt.title('Training')
plt.xlabel("Epoch")
plt.ylabel("Loss")
```

In [ ]:
```
'''
Testing Neural Network
'''
y_predicted = nn.predict(x_test) # predict

#plot
print(classification_report(y_test, y_predicted, target_names=dataset.target_names))
plot_confusion_matrix(nn, x_test, y_test, cmap=plt.cm.Blues, display_labels=dataset.target_names)
plt.show()
```