Spring 2019 CX4240 Homework 4

Dr. Mahdi Roozbahani

Deadline: Tuesday July 23, 2019, 11:59 pm

- No unapproved extension of the deadline is allowed. Late submission will lead to 0 credit.
- Discussion is encouraged, but each student must write his own answers and explicitly mention any collaborators.

submited by: Huy Thong Nguyen

Environment Setup

In [1]:
```
import csv
import numpy as np
import ast
from collections import Counter
from scipy import stats
```

Part 1: Utility Functions [50pts]

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 [20pts]

First, we computes entropy and then use this entropy for information gain.

In [2]:
```
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
    """
    prob = np.array(list(Counter(class_y).values()))
    prob = prob*1.0/len(class_y)   #Normalize to 1
    entropy = np.sum(-prob*np.log2(prob))
    return entropy
```

In [3]:
```python
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 cur
rent_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
    """
    H_prev = entropy(previous_y)

    left, right = current_y
    assert len(previous_y) == len(left) + len(right)
    p_left, p_right = len(left)*1.0/(len(previous_y)), len(right)*1.0/(len(previous_y))
    H_left, H_right = entropy(left), entropy(right)

    info_gain = H_prev - (p_left*H_left + p_right*H_right)

    return info_gain
```

In [4]:
```python
# TEST CASE
test_class_y = [0,0,0,1,1,1,1,1]
print(entropy(test_class_y))

previous_y = [0,0,0,1,1,1]
current_y = [[0,0], [1,1,1,0]]
print(information_gain(previous_y, current_y))
```

```
0.954434002924965
0.4591479170272448
```

Build a simple desicion tree step by step [30pts]

Now we will implement three functions to build a decision tree from the scratch.

(1) partition_classes: [10pts]

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 could be the mean of the values of split_attribute. If the split attribute is categorical, split_val should be one of the categories.

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.

In [5]:
```python
import numbers
```

In [6]:
```python
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 and split_val = 3 (mean of c
olumn 0)
    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]

    """
    X_left, X_right, y_left, y_right = [], [], [], []

    def is_left(X,ii, split_attribute, split_val):
        if isinstance(split_val,numbers.Number):
            return X[ii][split_attribute] <= split_val
        else:
            return X[ii][split_attribute] == split_val

    for ii in range(len(y)):
        if is_left(X,ii, split_attribute, split_val):
            X_left.append(X[ii])
            y_left.append(y[ii])
        else:
            X_right.append(X[ii])
            y_right.append(y[ii])

    # Return in this order
    return (X_left, X_right, y_left, y_right)
```

(2) find_best_split [10pts]

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.

In [7]:
```python
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 correspondi
ng 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 = 1  -->  info_gain = 0.17
           split = 2  -->  info_gain = 0.02
           split = 3  -->  info_gain = 0.02
           split = 4  -->  info_gain = 0.32
           split = 5  -->  info_gain = 0.

       best_split_val = 4; info_gain = .32;
    """
    best_split_val, max_IG = X[0][split_attribute], 0

    possible_vals = set([X[ii][split_attribute] for ii in range(len(X))])

    for split_val in possible_vals:
        X_left, X_right, y_left, y_right = partition_classes(X, y, split_attribute, split_val)
        current_IG = information_gain(y, [y_left, y_right])

        if len(y_left) == 0 or len(y_right) == 0:
            continue

        if current_IG>max_IG:
            best_split_val, max_IG = split_val, current_IG

    return best_split_val, max_IG
```

(3) find_best_feature [10pts]

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

In [8]:
```
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 = 0.42

       best_split_feature = 2; best_split_val = 22;
    """
    best_feature, best_split_val, best_IG = 0, X[0][0], 0

    for split_feature in range(len(X[0])):
        current_best_split_val, current_max_IG = find_best_split(X, y, split_feature)

        if current_max_IG>best_IG:
            best_feature, best_split_val, best_IG = split_feature, current_best_split_val, current_max
_IG

    return best_feature, best_split_val
```

In [9]:
```
# TEST CASE
test_X = [[3, 'aa', 10],[1, 'bb', 22],[2, 'cc', 28],[5, 'bb', 32],[4, 'cc', 32]]
test_y = [1,1,0,0,1]
print(partition_classes(test_X, test_y, 0, 3))
print(partition_classes(test_X, test_y, 1, 'bb'))

split_attribute = 0
best_split_val, info_gain = find_best_split(test_X, test_y, split_attribute)
print("best_split_val:", best_split_val, "info_gain:", info_gain)

best_feature, best_split_val = find_best_feature(test_X, test_y)
print("best_split_feature:", best_feature, "best_split_val:", best_split_val)
```

```
([[3, 'aa', 10], [1, 'bb', 22], [2, 'cc', 28]], [[5, 'bb', 32], [4, 'cc', 32]], [1, 1, 0], [0, 1])
([[1, 'bb', 22], [5, 'bb', 32]], [[3, 'aa', 10], [2, 'cc', 28], [4, 'cc', 32]], [1, 0], [1, 0, 1])
best_split_val: 4 info_gain: 0.3219280948873623
best_split_feature: 2 best_split_val: 22
```

Part 2: Decision Tree [30 pts]

In this part, you will train the decision tree and then use this tree to make predictions.

For starters, let's consider the following algorithm (a variation of C4.5) for the construction of a decision tree:

```
1) Check for base cases:
     a)If all elements of a list are of the same class, return a leaf node with the appropriate class labe
l.
     b)If a specified depth limit is reached, return a leaf labeled with the most frequent class.

2) For each attribute alpha: evaluate the normalized information gain gained by splitting on alpha

3) Let alpha_best be the attribute with the highest normalized information gain

4) Create a decision node that splits on alpha_best

5) Recur on the sublists obtained by splitting on alpha_best, and add those nodes as children of node
```

You need to finish the following three functions. In the __init__(), we initialize the tree as an empty dictionary or list. In the fit(), we fit a decision tree (self.tree) using the the features X and labels y. You could see the dataset (hw4data.csv). In the predict(), write a function to produce classifications for a list of features once your decision tree has been build.

[reference: [http://www.cs.cmu.edu/~cga/ai-course/dtree.pdf (http://www.cs.cmu.edu/~cga/ai-course/dtree.pdf)](http://www.cs.cmu.edu/~cga/ai-course/dtree.pdf)]

Hint: We need to use the recursion, so we need to select a stop condition which showed in step 1). Meanwhile, you could also define a max depth to stop. Here, we provide a simple recursion example for the tree traversal.

In [10]:

```
class Node():
    def __init__(self, data, left, right):
        self.data = data
        self.left = left
        self.right = right

class BTree:
    def __init__(self):
        self.root = None

    def insert(self, data):   #insert nodes
        r = self.root
        if r is None:
            self.root = Node(data, None, None)
            return
        while True:
            # if the node is smaller than the root
            if r.data > data:
                if r.left is None:
                    r.left = Node(data, None, None)
                    break
                else:
                    r = r.left
            else:
            # if the node is larger than the root
                if r.right is None:
                    r.right = Node(data, None, None)
                    break
                else:
                    r = r.right
    def preorder(self, root):
            if root is None:
                return
            else:
                print(root.data)
                self.preorder(root.left)
                self.preorder(root.right)

if __name__ == '__main__':
    bt = BTree()
    L = [27, 14, 10, 19, 35, 31, 42] #the first one is the node
    for i in L:
        bt.insert(i)
    # The tree will have three layers and from the top to the bottom, it will look like [27, [14,35],
 [10,19,31,42]]
    bt.preorder(bt.root)
    # Pre-order traversal
    # Reference: https://en.wikipedia.org/wiki/Tree_traversal
```

```
27
14
10
19
35
31
42
```

In [11]:
```python
class MyDecisionTree(object):
    def __init__(self, max_depth = None):
        """
        TODO: Initializing the tree as an empty dictionary or list, as preferred.

        For example: self.tree = [] or self.tree = {}
        """
        self.left = None
        self.right = None

        self.best_feature = None
        self.best_split_val = None

        self.is_leaf = False
        self.leaf_value = None

        self.max_depth = max_depth if max_depth != None else float('inf')

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

        NOTE: You will have to make use of the functions in utils.py to train the tree.
        One possible way of implementing the tree: Each node in self.tree could be in the form of a di
ctionary:
        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)
        """
        if self.max_depth <= 0:
            self.is_leaf = True
            self.leaf_value = Counter(y).most_common(1)[0][0]
            return

        if entropy(y) == 0:
            self.is_leaf = True
            self.leaf_value = y[0]
            return

        best_feature, best_split_val = find_best_feature(X, y)
        self.best_feature = best_feature
        self.best_split_val = best_split_val

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

        assert len(y_left) != 0
        self.left = MyDecisionTree(max_depth = self.max_depth-1)
        self.left.fit(X_left, y_left)

        assert len(y_right) != 0
        self.right = MyDecisionTree(max_depth = self.max_depth-1)
        self.right.fit(X_right, y_right)


    def predict(self, record):
        """
        TODO: classify a sample in test data set using self.tree and return the predicted label
        """
        if self.is_leaf:
            return self.leaf_value

        value = record[self.best_feature]
        check_left = (value <= self.best_split_val) if isinstance(value,numbers.Number) else (value ==
self.best_split_val)

        if check_left:
            return self.left.predict(record)
        else:
            return self.right.predict(record)
```

Now, let us use the Decision Tree to build a classifier and then to make predictions.

The accuracy may be difference since it will depends on the stop condition.

In [12]:
```
def DecisionTreeEvalution(depth = None):
    X = list()
    y = list()
    numerical_cols = set([0,10,11,12,13,15,16,17,18,19,20]) # indices of numeric attributes (columns)
    training_num = 2500
    # Loading data set
    print("reading hw4-data")
    with open("hw4-data.csv") as f:
        next(f, None)

        for line in csv.reader(f, delimiter=","):
            xline = []
            for i in range(len(line)):
                if i in numerical_cols:
                    xline.append(ast.literal_eval(line[i]))
                else:
                    xline.append(line[i])

            X.append(xline[:-1])
            y.append(xline[-1])

    print(X[0]) # print a data sample

    # Initializing a decision tree.
    dt = MyDecisionTree(max_depth = depth)
    print(dt.max_depth)

    # Building a tree
    print("fitting the decision tree")
    dt.fit(X[:training_num], y[:training_num])

    # Make predictions
    # For each test sample X, use our fitting dt classifer to predict
    y_predicted = []
    for record in X[training_num:]:
        y_predicted.append(dt.predict(record))

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

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

DecisionTreeEvalution()
```

```
reading hw4-data
[30, 'blue-collar', 'married', 'basic.9y', 'no', 'yes', 'no', 'cellular', 'may', 'fri', 487, 2, 999,
0, 'nonexistent', -1.8, 92.893, -46.2, 1.313, 5099.1]
inf
fitting the decision tree
accuracy: 0.8746
```

In [13]:
```
DecisionTreeEvalution(depth = 2)
```

```
reading hw4-data
[30, 'blue-collar', 'married', 'basic.9y', 'no', 'yes', 'no', 'cellular', 'may', 'fri', 487, 2, 999,
0, 'nonexistent', -1.8, 92.893, -46.2, 1.313, 5099.1]
2
fitting the decision tree
accuracy: 0.8987
```

Part 3: Random Forests [30pts]

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').

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 provided (with replacement) in accordance with a provided example subsampling rat
e.

b) From the sample in a), choose attributes at random to learn on (in accordance with a provided attribute
 subsampling rate)

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.

In RandomForests Class,

1. X is assumed to be a matrix with n rows and d columns where n is the number of total records and d is the number of features of each record.
2. y is assumed to be a vector of labels of length n.
3. XX is similar to X, except that XX also contains the data label for each record.

NOTE:

1. Lookout for TODOs for the parts that needs to be implemented.
2. Do NOT change the signature of the given functions.
3. Do NOT change any part of the randomForestClassifier function APART from the forest_size parameter.

In [14]:

```
from sklearn import tree

"""
NOTE: We import the tree from sklearn so that even if you can not build a decision tree sucessfully,
you could still finish the random forest classifer.

You are welcome to try to use your own decision tree MyDecisionTree() to finish random forest as well,
but for grading we will use the sklearn tree provided here.
"""

class RandomForest(object):
#     num_trees = 0
#     decision_trees = []

    # the bootstrapping datasets for trees
    # bootstraps_datasets is a list of lists, where each list in bootstraps_datasets is a bootstrapped
dataset.
#     bootstraps_datasets = []

    # the true class labels, corresponding to records in the bootstrapping datasets
    # bootstraps_labels is a list of lists, where the 'i'th list contains the labels corresponding to
 samples in the 'i'th bootstrapped dataset.
#

    def __init__(self, num_trees = 10, seed = 1):
        # Initialization done here
        self.num_trees = num_trees
        self.decision_trees = [tree.DecisionTreeClassifier() for i in range(num_trees)] # from sklearn
        self.bootstraps_datasets = []
        self.bootstraps_labels = []
        np.random.seed(seed=seed)

    def _bootstrapping(self, XX, n):

        """
        TODO: Create a sample dataset of size n by randomly sampling with replacement from the origina
l dataset XX.
        Note that you would also need to record the corresponding class labels for the sampled records
for training purposes.
        Reference: https://en.wikipedia.org/wiki/Bootstrapping_(statistics)
        """

        samples = [] # sampled dataset
        labels = []  # class labels for the sampled records

        N = len(XX)
        indexes = list(set(np.random.choice(np.arange(N),N-1)))
        for ii in indexes:
            samples.append(XX[ii][:-1])
            labels.append(XX[ii][-1])

        return (samples, labels)


    def bootstrapping(self, XX):
        # Initializing the bootstap datasets for each tree
        for i in range(self.num_trees):
            data_sample, data_label = self._bootstrapping(XX, len(XX))
            self.bootstraps_datasets.append(data_sample)
            self.bootstraps_labels.append(data_label)


    def fitting(self):
        """
        TODO: Train `num_trees` decision trees using the bootstraps datasets and labels by calling the
sklearn DecisionTree class.
        """
        for ii, (X, y) in enumerate(zip(self.bootstraps_datasets, self.bootstraps_labels)):
            self.decision_trees[ii].fit(X,y)
```

```python
    def voting(self, X):
        y = []

        for record in X:
            # Following steps have been performed here:
            #   1. Find the set of trees that consider the record as an out-of-bag sample.
            #   2. Predict the label using each of the above found trees.
            #   3. Use majority vote to find the final label for this record.
            votes = []
            for i in range(len(self.bootstraps_datasets)):
                dataset = self.bootstraps_datasets[i]
                if record not in dataset:
                    OOB_tree = self.decision_trees[i]
                    effective_vote = OOB_tree.predict([record])
                    votes.append(effective_vote[0])


            counts = np.bincount(votes)

            if len(counts) == 0:
            #  Special case
                #  Handle the case where the record is not an out-of-bag sample for any of the trees.
                y = np.append(y, 0)
            else:
                y = np.append(y, np.argmax(counts))

        return y
```

In [15]:

```python
"""
NOTE: Do not change this function apart from the forest_size parameter.
The accuracy will be a little different due to the random bootstrapping sampling.
"""
def randomForestClassifier():
    X = list()
    y = list()
    XX = list()  # Contains data features and data labels
    numerical_cols = set([0,10,11,12,13,15,16,17,18,19,20]) # indices of numeric attributes (columns)

    # Loading data set
    print("Reading the data.")
    with open("hw4-data.csv") as f:
        next(f, None)

        for line in csv.reader(f, delimiter=","):
            xline = []
            for i in range(len(line)):
                if i in numerical_cols:
                    xline.append(ast.literal_eval(line[i]))

            X.append(xline[:-1])
            y.append(xline[-1])
            XX.append(xline[:])

    """
    TODO: Initialize forest_size according to your implementation
    """
    # Minimum forest_size should be 10
    forest_size = 10

    # Initializing a random forest.
    randomForest = RandomForest(forest_size)

    # Creating the bootstrapping datasets
    print("Creating the bootstrap datasets.")
    randomForest.bootstrapping(XX)

    # Building trees in the forest
    print("Fitting the forest.")
    randomForest.fitting()

    # Calculating an unbiased error estimation of the random forest
    # based on out-of-bag (OOB) error estimate.
    y_predicted = randomForest.voting(X)

    # 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))

    print("Accuracy: %.4f" % accuracy)
    print("OOB estimate: %.4f" % (1-accuracy))

randomForestClassifier()
```

```
Reading the data.
Creating the bootstrap datasets.
Fitting the forest.
Accuracy: 0.8988
OOB estimate: 0.1012
```

Bouns Part : Challenge!

This part of the assignment is an opportunity to get extra credit for your final grade. To do so, we will be holding a competition to see who can write the best classifier to make predictions on a private research dataset. Your classifier should follow the standard sklearn format with .fit() and .predict() methods. This problem will not give you points on this assignment, but will count towards your final grade as follows:

First place: +3% on your final grade

Second place: +2% on your final grade

Third place: +1% on your final grade

If there are ties in accuracy, winner will be determined by submission time.

You've been provided with a sample of data from a research dataset in 'challenge_data.pickle'. It is serialized as a tuple of (features, classes). I have reserved a part of the dataset for testing. The classifier that performs most accurately on the holdout set wins (so optimize for accuracy).

As a minimum bar for getting extra credit, you'll need to get at least an average accuracy of 80% on the data you have, and at least an average accuracy of 60% on the holdout set.

Other rules:

- You are NOT allowed to import any pre-built classifiers (i.e. sklearn, xgboost, lightgbm, statsmodels, etc)
- You can import utilities from the following libraries to build neural networks:
  - keras
  - tensorflow
  - pytorch
- You are allowed to enter classifiers you have built in this or other assigments (though you'll probably want to improve them a bit)
- You may add any feature engineering you wish to your .fit() method to improve your results

In [32]:
```
from keras.models import Sequential
from keras.layers import Dense, Dropout, BatchNormalization
```

In [61]:
```
class ChallengeClassifier():

    def __init__(self, batch_size = 25, epoches = 30):
        # initialize whatever parameters you may need here-
        # this method will be called without parameters
        # so if you add any to make parameter sweeps easier, provide defaults

        self.batch_size = batch_size
        self.epoches = epoches

        self.model = Sequential()
        self.model.add(BatchNormalization())
        self.model.add(Dense(500,activation='relu'))
        self.model.add(Dropout(0.1))
        self.model.add(BatchNormalization())
        self.model.add(Dense(300,activation='relu'))
        #self.model.add(Dropout(0.1))
        self.model.add(Dense(1, activation='sigmoid'))

        self.model.compile(loss='binary_crossentropy',
                      optimizer='adam',
                      metrics=['accuracy'])

    def fit(self, features, classes, epoches = 10):
        # fit your model to the provided features
        e = epoches if epoches else self.epoches
        self.model.fit(features, classes,
          batch_size=self.batch_size,
          epochs=e,
          verbose = False)

    def predict(self, features):
        # classify each feature in features as either 0 or 1.
        y_pred = 1.0*(self.model.predict(features)>0.5)
        return y_pred.reshape(-1)
```

Prepare data

In [18]:
```
import pandas as pd
df = pd.read_csv('hw4-data.csv')
df =  df.select_dtypes(exclude='object')

X, y = df[df.columns[:-1]], df[df.columns[-1]].values
training_num = 2500
X_train, X_test = X.iloc[:training_num].values, X.iloc[training_num:].values
y_train, y_test = y[:training_num], y[training_num:]
```

Fit the model

In [68]:
```
model = ChallengeClassifier(batch_size = 25, epoches = 10)
model.fit(X_train,y_train, epoches = 30)
```

In [69]:
```python
best_score = -1
for _ in range(20):
    model.fit(X_train,y_train, epoches = 1)
    score = np.mean(model.predict(X_test) == y_test)*100
    print(score)
    best_score = max(score, best_score)
print('Best Score:', best_score)
```

```
90.7350216182829
90.36442248301421
91.59975293390981
90.61148857319333
90.92032118591723
90.98208770846202
90.48795552810377
90.79678814082767
90.79678814082767
90.54972205064855
90.85855466337244
90.85855466337244
90.85855466337244
90.54972205064855
90.79678814082767
90.79678814082767
90.24088943792464
89.87029030265596
90.48795552810377
90.0555898702903
Best Score: 91.59975293390981
```

Best prediction score is 91.599% (third one from the top)

In [ ]:
```python

```