Exercise 1 - Machine learning for Communication Networks and Systems

Shani Klein – 205781909

First , I use Google Colab notebook to write my code for Q1,Q2. The note book is in the link : https://colab.research.google.com/drive/1l8RJ27H2R_fP8c_hNk88jgTnzql3yGAH?usp=sharing

I highly recommend watching my solution both in here and in Google .The notebook contains explanations and visualization of the data in each part . However , this document also include all the theoretical information and results so you may read only this document if you prefer to .

For the random forest question I attached some .py file (I ran this part on local computer with strong CPU because the process is heavy and ineffiecient).

1. In this assignment we were given a set of rules in a file. Each row describe a different rule. The structure of the a rule is as follows (the @ sign is the first character of each rule): **@SourceIP/sourceMask DestinationIP/destinationMask sourcePort destinationPort protocol** The file is given in a TSV format .

1.1 The file we got contains a set of n rules $R = \{r_1, r_2, \ldots., r_n\}$. The rules need to be classified using the technique of Decision Tree, using only bits from source IP and destination IP field hence the first step I did was rearrange the data so I could build the decision tree easily.

Rearranging the data:

1. Extracting relevant columns- because we are interested only on the Source IP ,Destination IP and the mask for each IP I change the data we work on so it will include only the relevant parts.
After the extraction our data is in the form of :

SourceIP	sourceMask	DestinationIP	destinationMask

As we can see in the snippet from the notebook code:

0	180.171.147.187	32	83.167.110.154	32
1	180.171.147.161	32	86.109.168.32	32
2	180.171.147.254	32	85.199.42.4	32
3	180.171.147.63	32	25.178.135.86	32
4	180.171.147.89	32	25.184.48.239	32
...	...	...	...	...
205	85.52.83.70	31	199.45.140.2	32
206	85.52.83.112	31	199.45.140.2	32
207	85.52.83.98	31	25.184.48.129	32

2. Represent the IP's as 64 bits with '*' on wildcards- Due to the fact we need to work with bits with our decision tree I change the representation of the IPs to be one array of 64 bits. In addition, I wanted to mark each wild card as '*' so in the next sections I would be able to differ from original bit and a wild card bit.
So now our data is in the form of :

IP in bits	sourceMask	destinationMask

As we can see in the snippet from the notebook code:

	IP	s_wildCard	d_wildCard
0	10110100101010111001001110111011010100111010001...	32	32
1	10110100101010111001001110100001010101100110111...	32	32
2	10110100101010111001001111111110010101011100011...	32	32
3	10110100101010111001001100111111000110011011000...	32	32
4	10110100101010111001001101011001000110011011101...	32	32
...	...	...	...
695	01010101001101000101001101001**10110100101010...	30	22
696	01010101001101000101001101100**11000111001000...	30	22
697	01010101001101000101001101100**00011001101110...	30	22
698	01010101001101000101001101100**11000111001001...	30	22
699	0101010100110100010100********10110110101111...	23	26

3. Add rule number for each rule- I added a column with the rule number which can be like the "label" . I add this column so I could see the rule decision when printing the tree.

So After all the data manipulation our data look like this:

	IP	s_wildCard	d_wildCard	rule_number
0	10110100101010111001001110111011010100111010001...	32	32	0
1	10110100101010111001001110100001010101100110111...	32	32	1
2	10110100101010111001001111111110010101011100011...	32	32	2
3	10110100101010111001001100111111000110011011000...	32	32	3
4	10110100101010111001001101011001000110011011101...	32	32	4
...	...	...	...	...
715	0101010100110100010100********10011011001101...	23	16	715
716	0101010100110100010100********01001001000111...	23	16	716
717	0101010100110100010100********10110111001010...	23	16	717
718	0101010100110100010100********10110110101111...	23	16	718
719	0101010100110100010100********10110100101011...	23	16	719

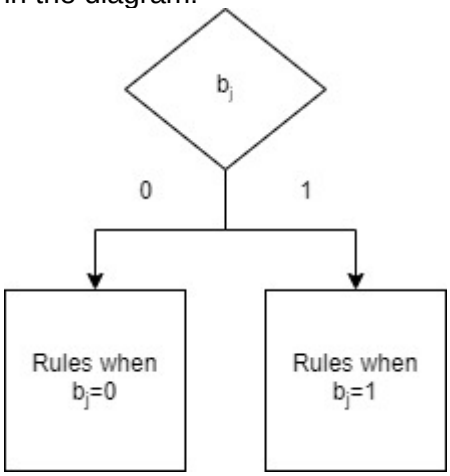
1.2. Using the criteria of maximum information gain, as described in the lecture, find the analytical expression of the $IG(R_i, b_j)$

To get the analytical expression of the $IG(R_i, b_j)$ we will first define some definition:
First, lets look at the entire set of rule:

- We must take under consideration the effect of the wild cards. The wild cards are like don't cares – so if we have a rule 000* when * denotes the don't care then the rule should be considered as both 0000 and 00001 .
  Hence for m don't cares we get 2^m option for a rule.
- The entropy of set of rules- $H(R)$ we define in the following matter:
  - For each rule R_i let's denote $N_{R_i} = 2^{|\phi_i|}$ when ϕ_i is the don't cares (the bits with number greater than the mask) of the rule R_i.
  - We will define $N_{total} = \sum_{i=1}^{n} N_{R_i}$.

Hence $H(R) = -\sum_{i=1}^{n} \frac{N_{R_i}}{N_{total}} \log_2\left(\frac{N_{R_i}}{N_{total}}\right) = -\frac{1}{N_{total}} \sum_{i=1}^{n} 2^{|\phi_i|} \log_2\left(\frac{2^{|\phi_i|}}{N_{total}}\right)$

Second, now lets look at specific bit b_j . the value of b_j can be only 1 or zero (it is possible that for some rules b_j is a don't care, we refer to this later on). Thus , given a bit b_j we can split the rules to rules when $b_j = 0$ and rules when $b_j = 1$ as describe in the diagram:



So now we can say that $H(R|b_j = 0)$ is the entropy of the left group and :$H(R|b_j = 1)$ is the entropy of the right group.
Now lets look at the effect of wild cards in the conditional entropy.
We say earlier that $H(R) = -\sum_{i=1}^{n} \frac{N_{R_i}}{N_{total}} \log_2\left(\frac{N_{R_i}}{N_{total}}\right) = -\frac{1}{N_{total}} \sum_{i=1}^{n} 2^{|\phi_i|} \log_2\left(\frac{2^{|\phi_i|}}{N_{total}}\right)$

but given a specific bit the counting for each rule may be different.
Lets Look at some random rule R_i and a bit b_j.
If R_i in bit b_j is '1' (not a wild card) , and R_i has m wild cards then R_i will appears 2^m times in the set of "rules when $b_j = 1$" (i.e. left group) and zero times on the right group.
But if R_i in bit is a wild card , and R_i has m wild cards then R_i will appears 2^m times in the total set of rules- half of them in the left group and half in the right group.
Meaning- the counting of R_i in each group will be $\frac{2^m}{2} = 2^{m-1}$ times.

Hence $H(R|b_j) = -\sum_{i=1}^{n} \frac{N_{R_i}}{N_{total}} \log_2\left(\frac{N_{R_i}}{N_{total}}\right)$
When N_{R_i} will be define as I mention above (the counting of each rule in the group) , and N_{total} will be the sum of all N_{R_i} in the group.

Then the information gain for a group R will be $IG(R, b_j) = H(R) - H(R|b_j)$ when $H(R|b_j) = p(b_j = 1) \cdot H(R|b_j = 1) + p(b_j = 0) \cdot H(R|b_j = 0)$.

Now that we define all the definition above we can define:
$IG(R, b_j) = H(R) - H(R|b_j)$
When $H(R|b_j) = p(b_j = 1) \cdot H(R|b_j = 1) + p(b_j = 0) \cdot H(R|b_j = 0)$

```
def compute_HR(rows,bit):
  counts = update_counts(rows,bit)
  Ntotal=0
  for num in counts:
     Ntotal+=counts[num]
  HR = 0
  for lbl in counts:
      prob_of_lbl = counts[lbl] / float(Ntotal)
      HR -= prob_of_lbl*math.log2(prob_of_lbl)
  return HR
```

when update_counts(rows,bit) work in the following manner:
we iterate the rows (which equivalent to the rules) and for each rule – if the rule in the bit 'bit' is not a wild card- then we update $N_{R_i} = 2^{|\phi_i|}$, else. If the bit is a wildcard that means the the rule split equally between the bits value 0 ,1 and therefor we will count only half of the occurrences , meaning $N_{R_i} = \frac{2^{|\phi_i|}}{2} = 2^{|\phi_i|-1}$
the code:

```
def update_counts(rows,bit):
  """Counts the number of each rule including wildcards"""
  counts = {}  # a dictionary of label -> count.
  num_of_rule=0;
  for row in rows:
     # the number of wildCard in source IP is 32 number of relavent bits
      s_wildCard=32-int(row["s_wildCard"])
      # same for dest's wildCard
      d_wildCard=32-int(row["d_wildCard"])
       # bit is a wild card so given bit we count only half of the wild card
      total_wildCard=s_wildCard+d_wildCard
      if(bit!='all' and row["IP"][bit]=='*'):
        total_wildCard=total_wildCard-1

      counts[num_of_rule] =2**total_wildCard
      num_of_rule+=1
  return counts
```

When computing the HR for the entire set in I got $\boldsymbol{H(R) = 1.094003600304795.}$

1.3 Using the criteria developed in 1.2, I wrote a Python code that compute the appropriate decision tree for all the versions given in the assignment.

First, because the tree itself is built according to the questions we ask , I created a class named Question which it's constructor receive 2 arguments- a bit and a value .

The class Question has a method match which receive a rule and return true if and only if the IP of the rule in bit 'bit' equals to 'value'.
For example if we run the following lines of code:

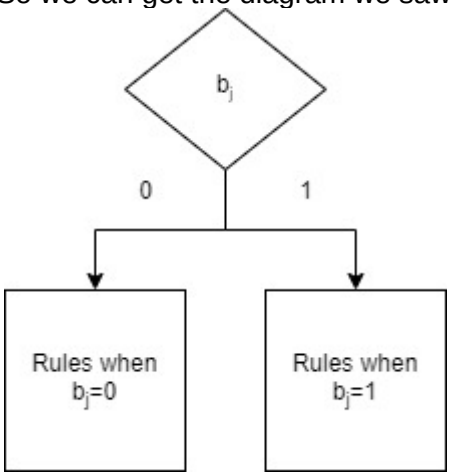
```
q=Question(0,1)
```

```
print("Question:",q)
# get rows from the data
rows=get_data_rows(data)
# choose the first row to be the example
example=rows[0]
print("example row: ",example)
print("Match result: ",q.match(example))
print("\n")
example=rows[3]
q=Question(0,0)
print("Question:",q)
print("example row: ",example)
print("Match result: ",q.match(example))
```

We will get the results:

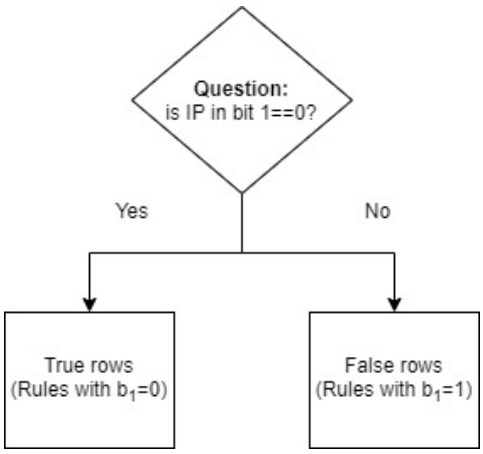
```
Question: Is IP in bit 0 == 1?
example row:  {'IP': '10110100101010111001001110111011010100111010011101101110100110101', 's_wildCard': '32', 'd_wildCard': '32'}
Match result:  True

Question: Is IP in bit 0 == 0?
example row:  {'IP': '10110100101010111001001100111111000110011011001010000111010101101', 's_wildCard': '32', 'd_wildCard': '32'}
Match result:  False
```

A Question is used to partition a dataset , so now that we have built the Question class we can split the data given a question to the rows which match the question (i.e. true rows) and the rows which do not match the data (i.e. true rows).
So we can get the diagram we saw earlier:



by asking "is bit b_j==0" and split the rules to rules whose answer "yes" and "no.

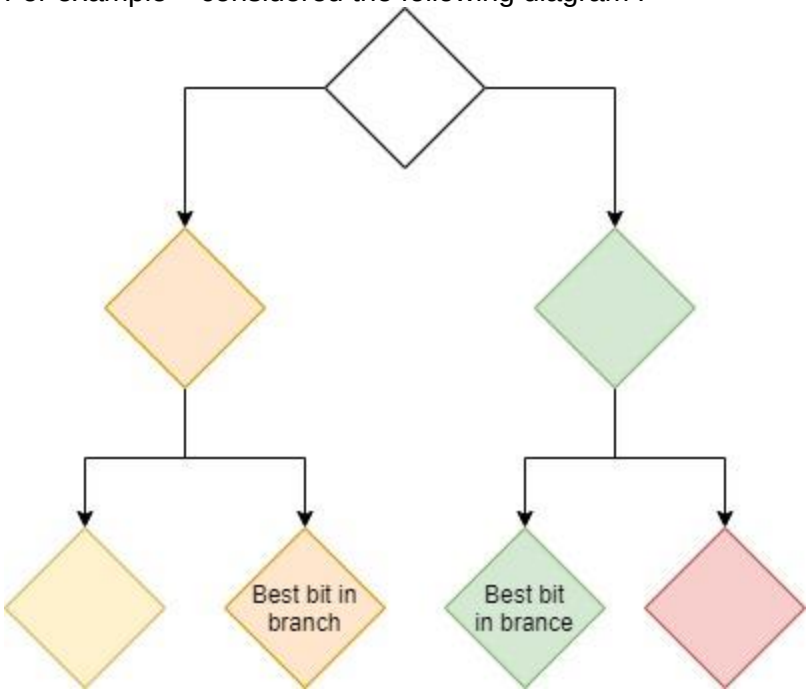
For the example lets take $b_j = 1$ By build a Question(1,0) which return true if and only if the row's ip in bit 1 is equals to 0 and split the data accordingly
Meaning:



So beside writing the Question Class I wrote the partition function that splits the data:

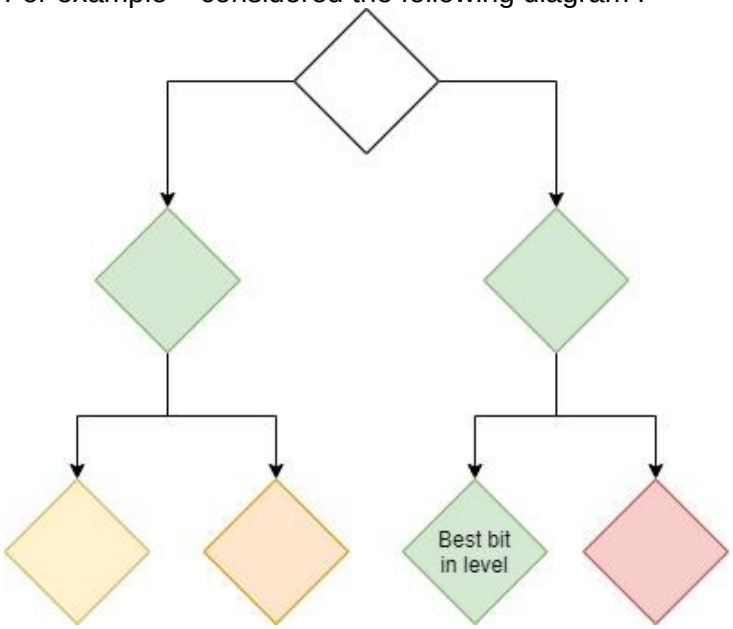
```
def partition(rows, question):
    true_rows, false_rows = [], []
    for row in rows:
        if question.match(row):
            true_rows.append(row)
        else:
            false_rows.append(row)
    return true_rows, false_rows
```

We can now create questions and split the data according to the question .
Next, we should decide which question to ask and when , to get the optimal decision tree. We would find the best partition using the maximum information gain criteria (As we saw in class it is also equivalent to minimum conditional entropy).
For all the versions given below:
1.3.1. For each branch in the tree we take the best bit, so different branches may use different sequences of bits.
For example – considered the following diagram :



Each branch chooses the question to be aske and split the data according to the higher information gain between the two groups of rules we got given the a certain bit. So we get the two branches use different sequences of bits.

1.3.1. At this section , branches at the same level will use the bit that provides the best IG among all the tested bits, so all paths will use the same order of bits.
For example – considered the following diagram :



Each level choose the best bit from the level under it thus all paths will use the same order of bits.

1.3.3 in this section I add a stopping point to the tree. the stopping point may be defined as getting a groups of rules all below a predefined length. We define the sizes of 16, 32, 64, 96 and 128 .This means, that if a group contains up to the given size number of rules, it need not be split more (i.e., becomes a decision node).

Lets look at the result for all versions that were mention above:

```
['criteria', 'IG_best_branch', 'my_tree depth:', 27]
['criteria', 'IG_best_branch', 'my_tree_16 depth:', 26]
['criteria', 'IG_best_branch', 'my_tree_32 depth:', 25]
['criteria', 'IG_best_branch', 'my_tree_64 depth:', 19]
['criteria', 'IG_best_branch', 'my_tree_96 depth:', 19]
['criteria', 'IG_best_branch', 'my_tree_128 depth:', 15]
-----------------------------------------------------------
['criteria', 'IG_best_level', 'my_tree depth:', 19]
['criteria', 'IG_best_level', 'my_tree_16 depth:', 11]
['criteria', 'IG_best_level', 'my_tree_32 depth:', 7]
['criteria', 'IG_best_level', 'my_tree_64 depth:', 5]
['criteria', 'IG_best_level', 'my_tree_96 depth:', 5]
['criteria', 'IG_best_level', 'my_tree_128 depth:', 5]
-----------------------------------------------------------
```

When my_tree symbols the decision tree without stopping pint , and my_tree_16 , my_tree_32 and ect, symbols the tree with the corresponded breaking point.

2. In this section we compare the results in question when the criteria is changed so the bit with the highest entropy is selected (in place of the highest IG).
Meaning right now , instead of taking the max of $\mathrm{I}G(R, b_i) = H(R) - H(R|b_i)$ we just take the max entropy.
Lets look at the he effect of wildcards in this case: Right now we don't need to look at a certain bit and check if it is a wild card, in this case the wild cards add the same number of rules for each case thus it has the same probability to both cases.
We have two option – first , just calculate the entropy with the wildcard but without the part of the conditional entropy .
Second option – because the wild cards gives the same probability to both cases we can just ignore them .The entropy wont be the same as a number but we only interested in the order of which has the highest entropy, and the order remains the same.
In this case, as we explained in the preview section, we consider both cases-the best bit for each branch and the best bit the level .
We ran the same code as in 1.3.3 but now with addition of the entropy as criteria and got:

```
['criteria', 'IG_best_branch', 'my_tree depth:', 27]
['criteria', 'IG_best_branch', 'my_tree_16 depth:', 26]
['criteria', 'IG_best_branch', 'my_tree_32 depth:', 25]
['criteria', 'IG_best_branch', 'my_tree_64 depth:', 19]
['criteria', 'IG_best_branch', 'my_tree_96 depth:', 19]
['criteria', 'IG_best_branch', 'my_tree_128 depth:', 15]
-----------------------------------------------------
['criteria', 'IG_best_level', 'my_tree depth:', 19]
['criteria', 'IG_best_level', 'my_tree_16 depth:', 11]
['criteria', 'IG_best_level', 'my_tree_32 depth:', 7]
['criteria', 'IG_best_level', 'my_tree_64 depth:', 5]
['criteria', 'IG_best_level', 'my_tree_96 depth:', 5]
['criteria', 'IG_best_level', 'my_tree_128 depth:', 5]
-----------------------------------------------------
['criteria', 'entropy_best_branch', 'my_tree depth:', 27]
['criteria', 'entropy_best_branch', 'my_tree_16 depth:', 26]
['criteria', 'entropy_best_branch', 'my_tree_32 depth:', 24]
['criteria', 'entropy_best_branch', 'my_tree_64 depth:', 20]
['criteria', 'entropy_best_branch', 'my_tree_96 depth:', 17]
['criteria', 'entropy_best_branch', 'my_tree_128 depth:', 15]
-----------------------------------------------------
['criteria', 'entropy_best_level', 'my_tree depth:', 25]
['criteria', 'entropy_best_level', 'my_tree_16 depth:', 18]
['criteria', 'entropy_best_level', 'my_tree_32 depth:', 13]
['criteria', 'entropy_best_level', 'my_tree_64 depth:', 8]
['criteria', 'entropy_best_level', 'my_tree_96 depth:', 8]
['criteria', 'entropy_best_level', 'my_tree_128 depth:', 7]
-----------------------------------------------------
```

To success creates all the above I wrote the function:

```
def find_best_split(rows,criterita="IG_best_level"):
    """Find the best question to ask by iterating over every feature / value
    and calculating the information gain."""
    best_gain = 0  # keep track of the best information gain
    best_question = None  # keep train of the feature / value that produced it
    current_uncertainty = compute_HR(rows,-1)
    n_features =64  # number of columns
    values={0,1}
    for bit in range(n_features):  # for each feature
      for val in values:  # for each value
        question = Question(bit,val)
        # print(question,bit,val)
        true_rows, false_rows = partition(rows,question)
        # print (true_rows)
        if len(true_rows) == 0 or len(false_rows) == 0:
            continue

        # 1.3.1 For each branch in the tree you take the best bit, so different branches may
use
        # different sequences of bits
        if(criterita=='IG_best_branch'):
          gain_left = info_gain(true_rows, false_rows, current_uncertainty,bit,'left')
          gain_right = info_gain(true_rows, false_rows, current_uncertainty,bit,'right')
          gain=gain_left if gain_left>gain_right else gain_left

        # 1.3.2.Branches at the same level will use the bit that provides the best IG among a
ll the
        # tested bits, so all paths will use the same order of bits
        elif(criterita=='IG_best_level'):
          gain = info_gain(true_rows, false_rows, current_uncertainty,bit)

        # criteria is changed so the bit with the highest entropy is selected
        # the best bit for each branch
        elif(criterita=='entropy_best_branch'):
          best_bit_true=compute_HR(true_rows,bit)
          best_bit_false=compute_HR(false_rows,bit)
          gain=best_bit_true if best_bit_true>best_bit_false else best_bit_false

        # bit for ll branches
        elif(criterita=='entropy_best_level'):
          gain=compute_HR(rows,bit)


        if gain >= best_gain:
            best_gain, best_question = gain, question

    return best_gain, best_question
```

the function tries all possible questions , split the data according to the question and compute the gain according to the given criteria , the question with the maximum gain will be the best question and that's the next question we will ask, and split the data according to it.

I wrote a functions that predict the rule according to the tree

```
try to predict rule number: 0
{0: '100%'}
try to predict rule number: 92
{92: '100%'}
try to predict rule number: 108
{108: '100%'}
try to predict rule number: 109
{109: '100%'}
try to predict rule number: 200
{200: '100%'}
try to predict rule number: 600
{600: '100%'}
-------------------------------------------
try to predict rule number: 1
{1: '50%', 513: '50%'}
try to predict rule number: 14
{14: '50%', 15: '50%'}
try to predict rule number: 15
{14: '50%', 15: '50%'}
try to predict rule number: 17
{17: '50%', 130: '50%'}
try to predict rule number: 40
{40: '25%', 41: '25%', 300: '25%', 484: '25%'}
try to predict rule number: 45
{45: '33%', 46: '33%', 516: '33%'}
-------------------------------------------
try to predict rule number: 10
{8: '9%', 10: '9%', 85: '9%', 94: '9%', 104: '9%', 249: '9%', 267: '9%', 287: '9%', 297: '9%',
537: '9%', 539: '9%'}
try to predict rule number: 20
{20: '7%', 22: '7%', 23: '7%', 142: '7%', 143: '7%', 144: '7%', 146: '7%', 147: '7%', 148:
'7%', 149: '7%', 256: '7%', 275: '7%', 498: '7%'}
try to predict rule number: 30
{30: '10%', 169: '10%', 171: '10%', 172: '10%', 235: '10%', 259: '10%', 266: '10%', 314:
'10%', 500: '10%', 573: '10%'}
try to predict rule number: 100
{7: '4%', 9: '4%', 86: '4%', 87: '4%', 88: '4%', 90: '4%', 91: '4%', 93: '4%', 95: '4%', 96: '4%',
98: '4%', 99: '4%', 100: '4%', 101: '4%', 102: '4%', 103: '4%', 105: '4%', 250: '4%', 273:
'4%', 298: '4%', 309: '4%', 310: '4%', 311: '4%', 493: '4%', 538: '4%'}
```

We can see that in some rules we get great prediction such as rules 0,92,108 and ect which has 100% prediction , some rules we have pretty good prediction but not so great with 50% prediction such in rules 14,15 or 33% in rule 45 , and some with really bad prediction like rule number 100 with only 4% accuracy.

In addition , I wrote a function that prints the tree, the results are pretty long and messy so I add only one tree for example in the appendix (you can run in the notebook as many examples that you would like.

Task 2- question 3 and 4

In this question you I use a predefined library of DecisionTreeClassifier from sklearn.tree library to classify the rules using the methods of Random Forest.

We were given DATA (packets) in a file with 2M packets. Each row of the file defines a packet with the following format , I uploaded the data and added headers, I got the following result:

	SrcIP	DstIP	Source_Port	Destination_Port	Protocol
0	3031143309	1518948592	46629	7125	6
1	1429492577	3341238196	9403	5632	6
2	1429492561	3169829382	31237	443	6
3	1429492595	3341651338	28616	40058	6
4	1429492549	3341650946	3873	1489	6
...	...	...	...	...	...
1999995	1429492595	3341650946	19446	7590	6
1999996	3031143220	3169020868	39842	1309	6
1999997	3031143235	3031133654	57979	1525	6
1999998	1429492577	3341237214	43434	1705	6
1999999	1429492557	431501441	2655	2110	6

2000000 rows × 5 columns

First, Dou to inefficiency of the python pre-defined functions- I took only the first 400k Packets. Then so first, I wrote a function that takes only 400k packets from the data, and I took add the rule number of the column as the label , meaning X is the data features and y is the labels.

```
def get_Xy(data):
    data_set_size=400000
    # add columns so we can extract the features easily
    data.columns = ["SrcIP", "DstIP", "Source_Port", "Destination_Port", "Protocol",
"Rule_Number"]
    feature_cols = ["SrcIP", "DstIP", "Source_Port", "Destination_Port", "Protocol"]
    X = data[feature_cols]
    X = X[0:data_set_size]
    y = data.Rule_Number  # Target variable
    y = y[0:data_set_size]
    return X,y
```

Next , I split the and use 80% percent of the data for classifications and 20% of the data for testing. I use the predefined method train_test_split from sklearn.model_selection library .

By calling:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1)
```

Now that I split the data I can choose A decision tree a classifier and train it .

I use two methods that were presented in the lectures:

- **First Method: Normal random forest(Bagging aggregation):**

In this case, the classifier based on our training data S, producing a predicted class label at input point x. To bag *C*, we draw bootstrap samples $S_1, \ldots, S_B$ each of size N with replacement from the training data , then $\hat{C}_{Bag} = majority\ vote\{C(S_b, x)\}_{b=1}^{B}$.

I define the function:

```
def run_normal_clf(X_train,y_train,X_test,y_test):
    print("Run normal random forest")
    clf = RandomForestClassifier(max_depth=32)
    clf = clf.fit(X_train, y_train)
    # Predict the response for test dataset
    y_pred = clf.predict(X_test)
    print("Accuracy:", metrics.accuracy_score(y_test, y_pred))
```

- **Second method -Adaboost**

The algorithm :

1. Initialize the observation weights $w_i = 1/N, i = 1, 2, \ldots, N$.
2. For $m = 1$ to M repeat steps (2.1)–(2.4):
   - **2.1.** Fit a classifier $C_m(x)$ to the training data using weights w_i.
   - **2.2.** Compute weighted error of newest tree
   $$\text{err}_m = \left(\sum_{i=1}^{N} w_i\, I(y_i \neq C_m(x_i))\right) \Big/ \left(\sum_{i=1}^{N} w_i\right)$$
   - **2.3.** Compute $\alpha_m = \log[(1 - \text{err}_m)/\text{err}_m]$.
   - **2.4.** Update weights for $i = 1, \ldots, N$:
   $$w_i \leftarrow w_i \cdot \exp(\alpha_m \cdot I(y_i \neq C_m(x_i))$$
   and renormalize w_i to sum to 1.
3. Output $C(x) = \text{sign}\left(\sum_{m=1}^{M} \alpha_m\, C_m(x)\right)$

I used a predefined method of sklearn that get a decision tree classifier and train it according to the Adaboost algorithm .

The code :

```
ef run_AdaBoost(X_train,y_train,X_test,y_test):
    print("Run AdaBoost Algorithm")
    abc = AdaBoostClassifier(RandomForestClassifier(max_depth=32))
    model = abc.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    print("Accuracy:", metrics.accuracy_score(y_test, y_pred))
```

All that is left now is to run both algorithms and look at the results:

```
def main():
    data = pd.read_csv("ScrambledPackets01.tsv", header=None, sep='\t')
    X,y=get_Xy(data)
    X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=1)
    run_normal_clf(X_train,y_train,X_test,y_test)
    run_AdaBoost(X_train,y_train,X_test,y_test)
```

And the results:
For both algorithms I got Accuracy of 0.79 - Not a bad result!

That's all .
The Assignment was very interesting , Thank you very much.

Appendix- example for a decision tree:

```
['criteria', 'IG_best_level', 'my_tree_16:']
Is IP in bit 19 == 0?
--> True:
  Predict: {Rule number,number of accur}  {736: 34359738368}
--> False:
  Is IP in bit 35 == 0?
  --> True:
    Predict: {Rule number,number of accur}  {8: 1, 10: 1, 20: 1, 21: 1, 22: 1, 23: 1, 24: 1,
25: 1, 26: 1, 27: 1, 28: 1, 29: 1, 30: 1, 31: 1, 33: 1, 34: 1, 35: 1, 40: 1, 41: 1, 45: 1, 46: 1,
48: 1, 51: 1, 76: 1, 77: 1, 85: 1, 94: 1, 104: 1, 132: 1, 134: 1, 135: 1, 136: 1, 137: 1, 140:
1, 142: 1, 143: 1, 144: 1, 146: 1, 147: 1, 148: 1, 149: 1, 150: 1, 152: 1, 153: 1, 154: 1,
155: 1, 158: 1, 159: 1, 160: 1, 161: 1, 162: 1, 164: 1, 165: 1, 168: 1, 169: 1, 171: 1, 172:
1, 173: 1, 174: 1, 176: 1, 177: 1, 179: 1, 180: 1, 181: 1, 182: 1, 183: 1, 184: 1, 186: 1,
187: 1, 188: 1, 190: 1, 191: 1, 201: 2, 205: 2, 206: 2, 208: 2, 209: 2, 212: 4, 220: 2, 221:
2, 222: 2, 224: 4, 230: 4, 231: 4, 234: 1, 235: 1, 236: 1, 237: 1, 247: 1, 249: 1, 256: 1,
257: 1, 258: 1, 259: 1, 260: 1, 261: 1, 262: 1, 266: 1, 267: 1, 268: 1, 270: 1, 271: 1, 272:
1, 275: 1, 279: 1, 280: 1, 284: 2, 286: 2, 287: 1, 294: 1, 296: 1, 297: 1, 300: 1, 302: 1,
303: 1, 313: 1, 314: 1, 315: 1, 316: 1, 317: 1, 320: 1, 321: 2, 323: 1, 324: 1, 327: 1, 331:
512, 332: 512, 334: 512, 337: 512, 340: 512, 341: 512, 343: 512, 345: 512, 346: 512,
347: 512, 348: 512, 349: 512, 350: 1024, 355: 1024, 356: 1024, 357: 1024, 358: 1024,
368: 1024, 374: 1024, 375: 1024, 376: 1024, 378: 1024, 379: 1024, 380: 1024, 381:
1024, 383: 1024, 386: 1024, 388: 1024, 389: 1024, 393: 2048, 398: 2048, 399: 2048,
405: 4096, 407: 4096, 408: 4096, 409: 512, 410: 512, 420: 4096, 426: 4096, 427: 4096,
428: 4096, 431: 4096, 432: 4096, 433: 4096, 435: 4096, 439: 512, 443: 8192, 445: 8192,
447: 512, 448: 8192, 450: 8192, 453: 8192, 464: 8192, 469: 8192, 470: 8192, 471: 8192,
472: 8192, 473: 8192, 479: 512, 480: 512, 481: 512, 484: 1, 485: 1, 498: 1, 499: 1, 500:
1, 501: 1, 502: 1, 503: 1, 504: 1, 506: 1, 507: 1, 508: 1, 509: 1, 516: 1, 519: 1, 528: 1,
530: 1, 532: 1, 537: 1, 539: 1, 540: 1, 555: 1, 560: 1, 561: 1, 562: 1, 566: 1, 567: 1, 569:
1, 571: 1, 573: 1, 577: 1, 578: 2, 579: 2, 580: 2, 582: 2, 587: 2, 588: 2, 591: 2, 596: 4,
597: 4, 604: 256, 605: 256, 609: 256, 611: 256, 614: 256, 623: 256, 626: 256, 627: 256,
628: 256, 629: 256, 630: 256, 632: 256, 634: 256, 654: 512, 656: 512, 657: 512, 658:
512, 659: 512, 660: 512, 661: 512, 662: 512, 665: 1024, 666: 1024, 668: 1024, 679:
1024, 680: 1024, 681: 1024, 682: 1024, 688: 2048, 690: 2048, 692: 2048, 693: 4096,
696: 4096, 698: 4096, 701: 65536, 704: 65536, 706: 131072, 709: 33554432, 714:
33554432, 716: 33554432, 720: 33554432, 722: 33554432, 724: 33554432, 725:
33554432, 730: 8589934592, 731: 17179869184, 734: 17179869184, 738:
34359738368, 741: 68719476736, 742: 274877906944, 743: 274877906944, 746:
137438953472, 748: 274877906944, 749: 274877906944, 750: 274877906944, 751:
549755813888, 754: 549755813888, 755: 549755813888, 756: 1099511627776, 757:
1099511627776, 758: 1099511627776, 764: 72057594037927936}
  --> False:
    Is IP in bit 39 == 1?
    --> True:
      Predict: {Rule number,number of accur}  {0: 1, 2: 1, 3: 1, 4: 1, 5: 1, 6: 1, 7: 1, 9: 1, 11:
1, 12: 1, 14: 1, 15: 1, 16: 1, 17: 1, 19: 1, 32: 1, 36: 1, 44: 1, 47: 1, 49: 1, 50: 1, 52: 1, 53:
1, 54: 1, 55: 1, 56: 1, 57: 1, 58: 1, 59: 1, 60: 1, 62: 1, 64: 1, 65: 1, 66: 1, 67: 1, 68: 1, 69:
1, 70: 1, 71: 1, 72: 1, 73: 1, 75: 1, 78: 1, 79: 1, 81: 1, 82: 1, 83: 1, 84: 1, 86: 1, 87: 1, 88:
1, 90: 1, 91: 1, 92: 1, 93: 1, 95: 1, 96: 1, 98: 1, 99: 1, 100: 1, 101: 1, 102: 1, 103: 1, 105:
1, 106: 1, 108: 1, 109: 1, 110: 1, 111: 1, 112: 1, 122: 1, 123: 1, 124: 1, 125: 1, 130: 1,
133: 1, 138: 1, 139: 1, 141: 1, 145: 1, 151: 1, 156: 1, 157: 1, 163: 1, 166: 1, 167: 1, 170:
1, 175: 1, 178: 1, 185: 1, 189: 1, 192: 2, 194: 2, 195: 2, 196: 2, 198: 2, 199: 2, 202: 2,
204: 2, 207: 2, 210: 4, 211: 4, 213: 2, 214: 2, 215: 2, 216: 2, 217: 2, 218: 2, 219: 2, 223:
4, 225: 4, 226: 4, 227: 4, 229: 4, 232: 1, 233: 1, 240: 1, 242: 1, 243: 1, 244: 1, 245: 1,
248: 1, 250: 1, 251: 1, 252: 1, 253: 1, 254: 1, 265: 1, 269: 1, 273: 1, 281: 1, 282: 2, 283:
2, 285: 2, 289: 1, 290: 1, 291: 1, 295: 1, 298: 1, 299: 1, 301: 1, 304: 1, 305: 1, 306: 1,
307: 1, 308: 1, 309: 1, 310: 1, 311: 1, 312: 1, 318: 1, 319: 1, 322: 1, 325: 1, 326: 1, 328:
```

512, 329: 512, 330: 512, 335: 512, 336: 512, 338: 512, 342: 512, 344: 512, 351: 1024, 354: 1024, 359: 1024, 360: 1024, 361: 1024, 362: 1024, 363: 1024, 364: 1024, 366: 1024, 367: 1024, 369: 1024, 371: 1024, 372: 1024, 377: 1024, 382: 1024, 384: 1024, 385: 1024, 387: 1024, 390: 1024, 392: 2048, 394: 2048, 395: 2048, 396: 2048, 397: 2048, 401: 4096, 403: 4096, 404: 4096, 406: 4096, 412: 4096, 413: 4096, 414: 4096, 415: 4096, 417: 4096, 418: 4096, 419: 4096, 421: 4096, 423: 4096, 429: 4096, 430: 4096, 434: 4096, 436: 512, 438: 512, 440: 8192, 441: 8192, 444: 8192, 449: 8192, 451: 8192, 452: 8192, 454: 8192, 455: 8192, 456: 8192, 457: 8192, 458: 8192, 459: 8192, 460: 8192, 461: 8192, 462: 8192, 463: 8192, 467: 8192, 468: 8192, 474: 8192, 475: 8192, 478: 512, 482: 1, 486: 1, 487: 1, 488: 1, 489: 1, 490: 1, 491: 1, 492: 1, 493: 1, 495: 1, 505: 1, 515: 1, 517: 1, 518: 1, 520: 1, 522: 1, 523: 1, 524: 1, 525: 1, 526: 1, 529: 1, 531: 1, 533: 1, 534: 1, 536: 1, 538: 1, 542: 1, 544: 1, 545: 1, 546: 1, 547: 1, 548: 1, 549: 1, 559: 1, 563: 1, 564: 1, 565: 1, 568: 1, 570: 1, 572: 1, 574: 1, 575: 1, 576: 1, 581: 2, 583: 2, 584: 2, 585: 2, 586: 2, 589: 2, 592: 4, 594: 4, 595: 4, 598: 256, 599: 256, 600: 256, 601: 256, 602: 256, 603: 256, 608: 256, 610: 256, 612: 256, 613: 256, 615: 256, 616: 256, 624: 256, 631: 256, 633: 256, 635: 256, 636: 512, 637: 512, 638: 512, 640: 512, 641: 512, 643: 512, 645: 512, 646: 512, 647: 512, 648: 512, 649: 512, 650: 512, 653: 512, 655: 512, 663: 1024, 664: 1024, 667: 512, 671: 1024, 672: 1024, 673: 1024, 674: 1024, 683: 1024, 685: 1024, 686: 1024, 691: 2048, 694: 4096, 697: 4096, 702: 65536, 703: 65536, 705: 131072, 707: 33554432, 715: 33554432, 717: 33554432, 721: 33554432, 723: 33554432, 727: 67108864, 728: 8589934592, 729: 8589934592, 765: 72057594037927936}

--> False:

Is IP in bit 44 == 1?

--> True:

Predict: {Rule number,number of accur} {1: 1, 18: 1, 37: 1, 42: 1, 43: 1, 61: 1, 63: 1, 74: 1, 80: 1, 89: 1, 97: 1, 113: 1, 114: 1, 115: 1, 116: 1, 117: 1, 126: 1, 127: 1, 128: 1, 129: 1, 131: 1, 193: 2, 200: 2, 228: 4, 238: 1, 239: 1, 241: 1, 246: 1, 264: 1, 276: 2, 277: 1, 278: 1, 288: 1, 333: 512, 339: 512, 352: 1024, 365: 1024, 373: 1024, 391: 2048, 400: 4096, 402: 4096, 416: 4096, 424: 4096, 425: 4096, 437: 512, 442: 8192, 446: 512, 465: 8192, 476: 512, 497: 1, 511: 1, 512: 1, 513: 1, 514: 1, 527: 1, 535: 1, 550: 1, 551: 1, 552: 1, 553: 1, 554: 1, 556: 1, 557: 1, 558: 1, 593: 4, 606: 256, 607: 256, 617: 256, 621: 256, 622: 256, 639: 512, 644: 512, 676: 1024, 678: 1024, 684: 1024, 689: 2048, 695: 4096, 699: 32768, 700: 32768, 708: 33554432, 712: 33554432, 713: 33554432, 718: 33554432, 719: 33554432, 726: 67108864, 762: 2251799813685248}

--> False:

Is IP in bit 40 == 1?

--> True:

Is IP in bit 36 == 1?

--> True:

Predict: {Rule number,number of accur} {13: 1, 38: 1, 39: 1, 107: 1, 118: 1, 119: 1, 120: 1, 121: 1, 203: 2, 255: 1, 263: 1, 292: 1, 293: 1, 353: 1024, 370: 1024, 466: 8192, 496: 1, 521: 1, 541: 1, 590: 2, 618: 256, 619: 256, 620: 256, 625: 256, 651: 512, 652: 512, 677: 1024, 687: 1024, 761: 2251799813685248}

--> False:

Predict: {Rule number,number of accur} {543: 1}

--> False:

Is IP in bit 43 == 1?

--> True:

Predict: {Rule number,number of accur} {197: 2, 274: 1, 411: 4096, 477: 512, 483: 1, 494: 1, 510: 1, 642: 512, 669: 1024, 670: 1024, 675: 1024, 711: 33554432, 763: 2251799813685248}

--> False:

Is IP in bit 29 == 1?

--> True:

Predict: {Rule number,number of accur} {422: 4096}

--> False:

```
            Is IP in bit 47 == 1?
            --> True:
              Predict: {Rule number,number of accur}  {710: 33554432}
            --> False:
              Is IP in bit 38 == 1?
              --> True:
                Predict: {Rule number,number of accur}  {733: 17179869184}
              --> False:
                Is IP in bit 7 == 1?
                --> True:
                  Predict: {Rule number,number of accur}  {737: 34359738368, 740:
68719476736, 747: 137438953472, 760: 1099511627776, 766: 36028797018963968}
                --> False:
                  Predict: {Rule number,number of accur}  {732: 17179869184, 735:
34359738368, 739: 68719476736, 744: 68719476736, 745: 137438953472, 752:
549755813888, 753: 274877906944, 759: 1099511627776, 767: 36028797018963968,
768: 36028797018963968, 769: 9223372036854775808, 770: 9223372036854775808}
----------------------------------------------------------
```