In [ ]:

```python
import os
import sys
import copy
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

import warnings
warnings.filterwarnings("ignore")
```

In [ ]:

```python
data = pd.read_csv("./../../data/fraud_detection_bank_dataset.csv")
col_names = [f"col_{i}" for i in range(111)]
target = "targets"

train_data, test_data = train_test_split(data, train_size=0.8, random_state=123)
X_train, y_train = train_data[col_names].values, train_data[target].values
X_test, y_test = test_data[col_names].values, test_data[target].values
```

Decision Stump

Pseudo-Code

Input: Feature Matrix X and label vector y

```
for each feature j ('Column' of X)
    for each threshold t
        set `y_yes` to most commom label of obejects i satisfying rule (xij > t)
        set `y_no` to most commom label of obejects i not satisfying rule (xij <= t)
        set `y_hat` to be prediction
        compute error
        store the rule (j, t, y_yes, y_no), if it has the lowest error
```

Cost of Decision Stumps

Assume we have:

- 'n' examples (days that we measured).

- 'd' features (foods that we measured).
- 'k' thresholds (>0, >1, >2, ...) for each feature
- final cost is $O(ndk)$, assume k=n, then it is $O(n^2d)$

Improvement

- Accuracy is not a good way to split the feature.
- $O(n^2d)$ can be improved to $O(ndlog(n))$

In [ ]:

```
def accuracy(y, y_hat):

    return (y == y_hat).sum() / len(y)


class Decision_stump():

    def fit(self, X, y):

        accu_max = -np.inf
        model = []

        for idx, feature in enumerate(X.T):

            threshs = set(feature)
            for thresh in threshs:

                y_yes = y[feature > thresh]
                y_no = y[feature <= thresh]

                if y_yes.sum() < int(0.5 * len(y_yes)):
                    y_hat_yes = np.zeros_like(y_yes)
                else:
                    y_hat_yes = np.ones_like(y_yes)

                if y_no.sum() < int(0.5 * len(y_no)):
                    y_hat_no = np.zeros_like(y_no)
                else:
                    y_hat_no = np.ones_like(y_no)

                y_hat_con = np.concatenate([y_hat_yes, y_hat_no])
```

```python
                y_con = np.concatenate([y_yes, y_no])

                accu = accuracy(y_hat_con, y_con)
                if accu > accu_max:
                    model = [idx, thresh, accu, y_hat_yes[0], y_hat_no[0]]
                    accu_max = accu

        self.model = model

    def predict(self, X):

        idx, thresh, _, y_yes_fill, y_no_fill = self.model
        prediction = np.where(X[:, idx]>thresh, y_yes_fill, y_no_fill)

        return prediction
```

In [ ]:

```python
decision_stump = Decision_stump()
decision_stump.fit(X_train, y_train)

y_train_hat = decision_stump.predict(X_train)
y_test_hat = decision_stump.predict(X_test)

print("Train accuracy: ", accuracy(y_train, y_train_hat))
print("Test accuracy: ", accuracy(y_test, y_test_hat))
```

```
Train accuracy:  0.8229510199096128
Test accuracy:  0.8346360527601367
```

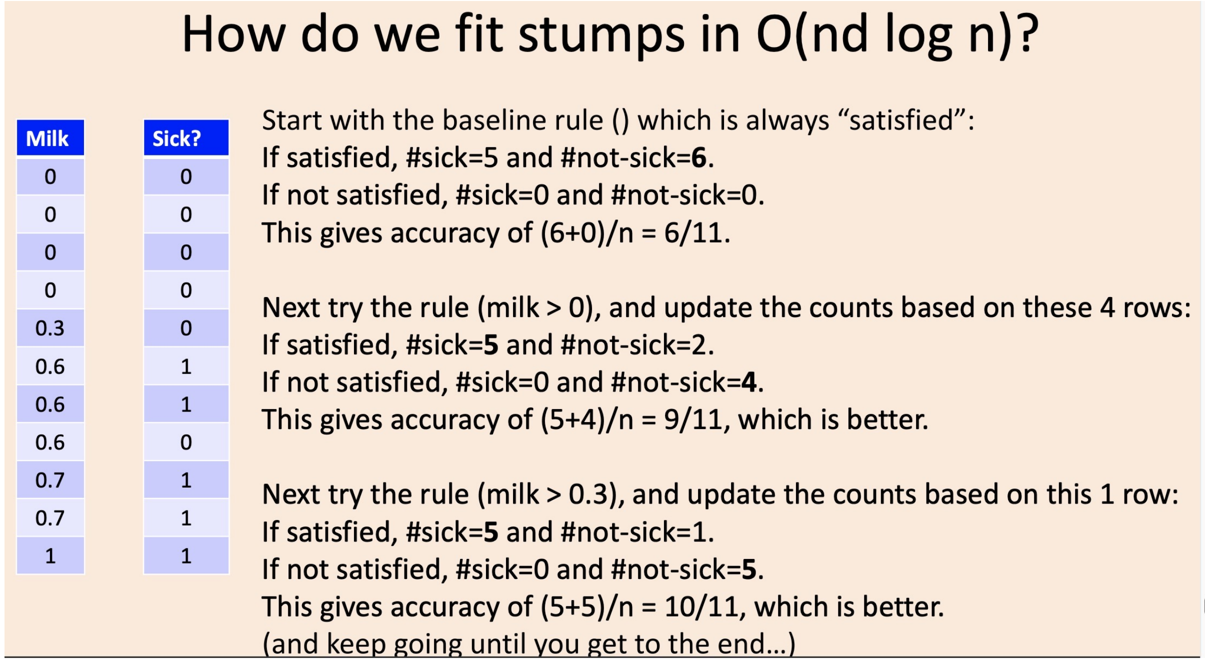
In [ ]:

```python
decision_stump.model
```

Out[ ]:

```
[83, 0.0, 0.8229510199096128, 1, 0]
```

$O(n^2d)$ can be improved to $O(ndlog(n))$



How do we fit stumps in O(nd log n)?

Milk	Sick?
0	0
0	0
0	0
0	0
0.3	0
0.6	1
0.6	1
0.6	0
0.7	1
0.7	1
1	1

Start with the baseline rule () which is always “satisfied”:
If satisfied, #sick=5 and #not-sick=**6**.
If not satisfied, #sick=0 and #not-sick=0.
This gives accuracy of (6+0)/n = 6/11.

Next try the rule (milk > 0), and update the counts based on these 4 rows:
If satisfied, #sick=**5** and #not-sick=2.
If not satisfied, #sick=0 and #not-sick=**4**.
This gives accuracy of (5+4)/n = 9/11, which is better.

Next try the rule (milk > 0.3), and update the counts based on this 1 row:
If satisfied, #sick=**5** and #not-sick=1.
If not satisfied, #sick=0 and #not-sick=**5**.
This gives accuracy of (5+5)/n = 10/11, which is better.
(and keep going until you get to the end…)

- pre-order every feature, it took nlog(n), then repeat for d times
- I tried the algorithm above, but maybe because of numpy's efficiency, I somehow didn't get a faster version

In [ ]:

```
class Decision_stump():

    def fit(self, X, y):

        accu_max = -np.inf
        model = []

        for feat_num, feature in enumerate(X.T):

            sorted_feature = zip(feature, y)
            sorted_feature = sorted(sorted_feature, key=lambda x: x[0], reverse=False)
            _, sorted_y = zip(*sorted_feature)


            p_count_unsatis, f_count_unsatis = 0, 0
            p_count_satis, f_count_satis = sum(sorted_y), len(sorted_y) - sum(sorted_y)

            if p_count_satis < f_count_satis:
                sorted_y_hat = np.zeros_like(sorted_y)
```

```
            else:
                sorted_y_hat = np.ones_like(sorted_y)


            thresh_prev = sorted_feature[0][0]

            for idx, (thresh, label) in enumerate(sorted_feature):

                if label == 1:
                    p_count_unsatis += 1
                    p_count_satis -= 1
                else:
                    f_count_unsatis += 1
                    f_count_satis -= 1

                if thresh != thresh_prev or idx == len(sorted_feature) - 1:

                    if p_count_unsatis < f_count_unsatis:
                        sorted_y_hat[:idx] = 0
                    else:
                        sorted_y_hat[:idx] = 1

                    if p_count_satis < f_count_satis:
                        sorted_y_hat[idx:] = 0
                    else:
                        sorted_y_hat[idx:] = 1


                    accu = accuracy(sorted_y, sorted_y_hat)
                    if accu > accu_max:
                        model = [feat_num, thresh_prev, accu, sorted_y_hat[-1], sorted_y_hat[0]]
                        accu_max = accu

                    thresh_prev = thresh



        self.model = model

    def predict(self, X):

        idx, thresh, _, y_yes_fill, y_no_fill = self.model
        prediction = np.where(X[:, idx]>thresh, y_yes_fill, y_no_fill)

        return prediction
```

In [ ]:

```python
decision_stump_1 = Decision_stump()
decision_stump_1.fit(X_train, y_train)

y_train_hat = decision_stump_1.predict(X_train)
y_test_hat = decision_stump_1.predict(X_test)

print("Train accuracy: ", accuracy(y_train, y_train_hat))
print("Test accuracy: ", accuracy(y_test, y_test_hat))
```

```
Train accuracy:  0.8229510199096128
Test accuracy:  0.8346360527601367
```

Use entropy instead of using accuracy

Pseudo-Code

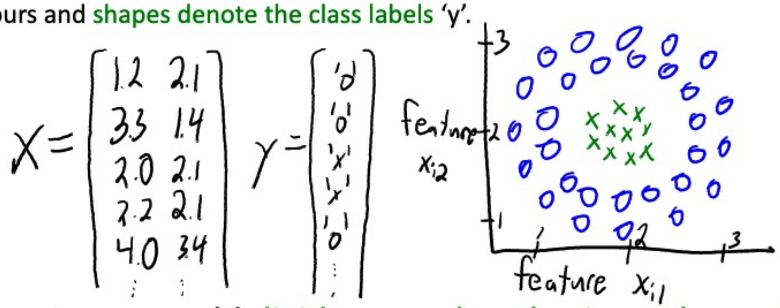
Input: vector y

```
counter_dict = dict
for ele feature y
  dict[ele] += 1

entropy = 0
for i in dict:
  prob = dict[i] / n
  entropy -= prob * log(prob)

return entropy
```

Why not accuracy?

Example Where Accuracy Fails

- Consider a dataset with 2 features and 2 classes ('x' and 'o').
  - Because there are 2 features, we can draw 'X' as a scatterplot.
    - Colours and shapes denote the class labels 'y'.



- A decision stump would divide space by a horizontal or vertical line.
  - Testing whether $x_{i1} > t$ or whether $x_{i2} > t$.
- On this dataset no horizontal/vertical line improves accuracy.
  - Baseline is 'o', but need to get many 'o' wrong to get one 'x' right.

In [ ]:

```python
from collections import Counter

def entropy(y):

    p_dist = np.array(list(Counter(y).values()))
    p_dist = p_dist / p_dist.sum()
    ent = (-1 * np.log(p_dist) * p_dist).sum()

    return ent
```

In [ ]:

```python
class Decision_stump_entropy():

    @staticmethod
    def entropy(y):
        p_dist = np.array(list(Counter(y).values()))
        p_dist = p_dist / p_dist.sum()
        ent = (-1 * np.log(p_dist) * p_dist).sum()
        return ent

    def fit(self, X, y):
```

```python
        gain_max = -np.inf
        model = []
        ent_base = entropy(y)

        for idx, feature in enumerate(X.T):

            threshs = np.linspace(feature.min(), feature.max(), min(len(set(feature)), 100))
            # threshs = set(feature)

            for thresh in threshs:

                y_yes = y[feature > thresh]
                y_no = y[feature <= thresh]

                y_hat_yes = int(y_yes.sum() >= int(0.5 * len(y_yes)))
                y_hat_no = int(y_no.sum() >= int(0.5 * len(y_no)))

                gain = ent_base - (len(y_yes) * entropy(y_yes) + len(y_no) * entropy(y_no)) / len(y)

                if gain > gain_max:
                    model = [idx, thresh, gain, y_hat_yes, y_hat_no]
                    gain_max = gain

        self.model = model

    def predict(self, X):

        idx, thresh, _, y_yes_fill, y_no_fill = self.model
        prediction = np.where(X[:, idx]>thresh, y_yes_fill, y_no_fill)

        return prediction
```

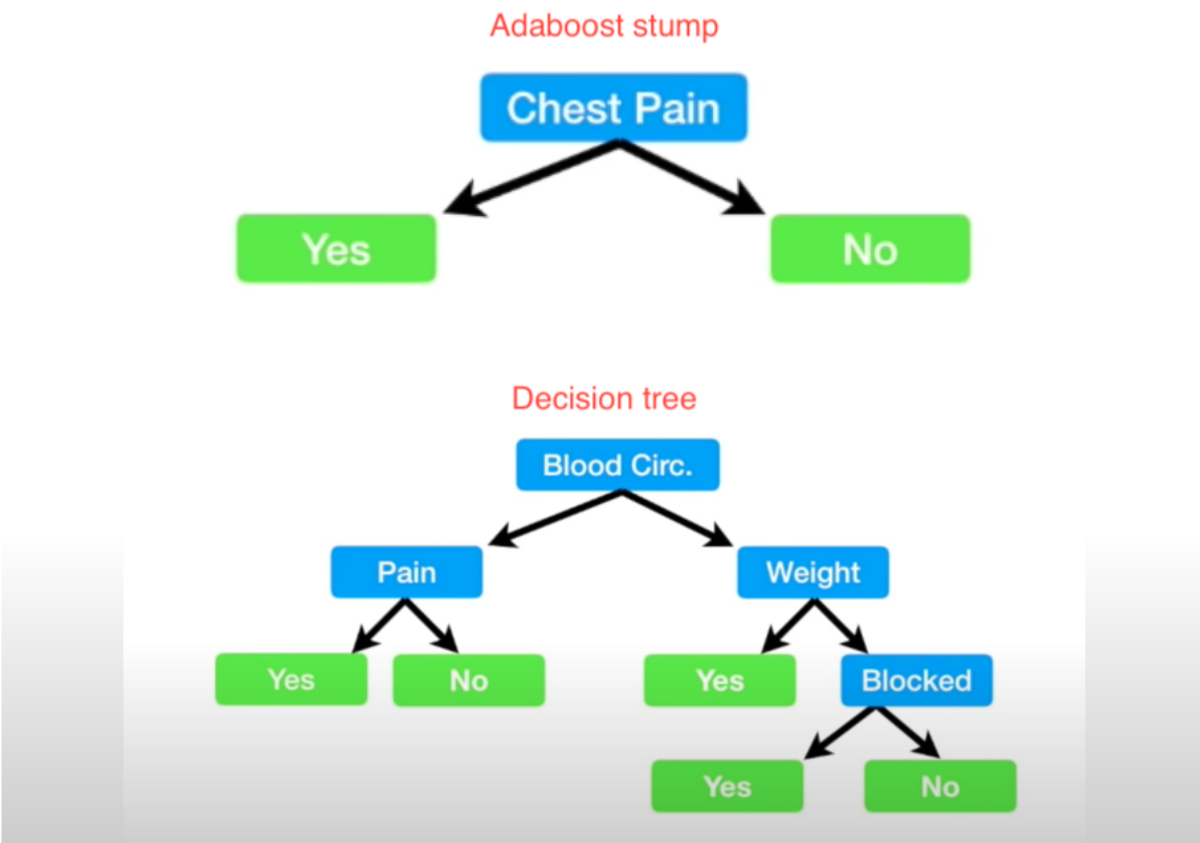
In [ ]:

```python
decision_stump_2 = Decision_stump_entropy()
decision_stump_2.fit(X_train, y_train)

y_train_hat = decision_stump_2.predict(X_train)
y_test_hat = decision_stump_2.predict(X_test)

print("Train accuracy: ", accuracy(y_train, y_train_hat))
print("Test accuracy: ", accuracy(y_test, y_test_hat))
```

```
Train accuracy:  0.7667033101258092
Test accuracy:  0.7657547630679042
```

Decision Tree

- It is a recursive decision stump
- Use tree structure or dict to restore the model. This might cause some difference on the former code.



In [ ]:

```
class Tree(object):

    def __init__(self, var=None, gain=None, thresh=None, left=None, right=None):

        self.var = var
        self.gain = gain
        self.thresh = thresh
        self.left = left
        self.right = right
```

```python
    def inorder(self):

        if self.var is not None:
            print(self.var, end=' ')

        if self.left is not None and isinstance(self.left, Tree):
            self.left.inorder()
        else:
            print('leaf', self.left, end=' ')

        if self.right is not None and isinstance(self.right, Tree):
            self.right.inorder()
        else:
            print('leaf', self.right, end='\r\n')


class Decision_stump(Tree):

    def __init__(self):
        super().__init__()

    @staticmethod
    def entropy(y):
        p_dist = np.array(list(Counter(y).values()))
        p_dist = p_dist / p_dist.sum()
        ent = (-1 * np.log(p_dist) * p_dist).sum()
        return ent

    def fit(self, X, y):

        gain_max = -np.inf
        ent_base = entropy(y)

        for idx, feature in enumerate(X.T):

            threshs = np.linspace(feature.min(), feature.max(), min(len(set(feature)), 100))

            for thresh in threshs:

                y_yes = y[feature > thresh]
                y_no = y[feature <= thresh]

                y_hat_yes = int(y_yes.sum() >= int(0.5 * len(y_yes)))
                y_hat_no = int(y_no.sum() >= int(0.5 * len(y_no)))

                gain = ent_base - (len(y_yes) * entropy(y_yes) + len(y_no) * entropy(y_no)) / len(y)
```

```
                if gain > gain_max:

                    gain_max = gain

                    self.var = idx
                    self.gain = gain
                    self.thresh = thresh
                    self.left = y_hat_no
                    self.right = y_hat_yes

        return self
```

In [ ]:

```
class Decision_Tree(Tree):

    def __init__(self, depth=3):

        super().__init__()
        self.depth = depth
        self.model = None

    def fit(self, X, y, model=Decision_stump(), cur_level=0,):

        model.fit(X, y)

        print(f"""Training  =======>  Current level:{cur_level}, Split_var at: {model.var}, Split_thresh at: {model.thresh}, Left

        if cur_level == self.depth:
            return
        else:
            idx_l, idx_r = X[:, model.var] <= model.thresh, X[:, model.var] > model.thresh
            if len(idx_l) > 0:
                model.left = Decision_stump()
                self.fit(X[idx_l], y[idx_l], model=model.left, cur_level=cur_level+1)

            if len(idx_r) > 0:
                model.right = Decision_stump()
                self.fit(X[idx_r], y[idx_r], model=model.right, cur_level=cur_level+1)

        self.model = model

    def predict(self, X, model=None):

        if isinstance(model, int) or isinstance(model, float):
            return model
```

```python
        if model is None:
            model = self.model

        var = model.var
        thresh = model.thresh

        if X[var] > thresh:
            return self.predict(X, model.right)
        else:
            return self.predict(X, model.left)
```

In [ ]:

```python
decision_tree = Decision_Tree(depth=3)
decision_tree.fit(X_train, y_train)

y_train_hat = np.apply_along_axis(decision_tree.predict, axis=1, arr=X_train)
y_test_hat = np.apply_along_axis(decision_tree.predict, axis=1, arr=X_test)

print("Train accuracy: ", accuracy(y_train, y_train_hat))
print("Test accuracy: ", accuracy(y_test, y_test_hat))
```

```
Training  =======>  Current level:0, Split_var at: 5, Split_thresh at: 0.0, Left leaf value:0, Right leaf value:1.
Training  =======>  Current level:1, Split_var at: 4, Split_thresh at: 0.0, Left leaf value:0, Right leaf value:0.
Training  =======>  Current level:2, Split_var at: 83, Split_thresh at: 0.0, Left leaf value:0, Right leaf value:1.
Training  =======>  Current level:3, Split_var at: 26, Split_thresh at: 0.0, Left leaf value:0, Right leaf value:0.
Training  =======>  Current level:3, Split_var at: 106, Split_thresh at: 0.0, Left leaf value:1, Right leaf value:0.
Training  =======>  Current level:2, Split_var at: 106, Split_thresh at: 0.0, Left leaf value:1, Right leaf value:0.
Training  =======>  Current level:3, Split_var at: 41, Split_thresh at: 0.0, Left leaf value:1, Right leaf value:0.
Training  =======>  Current level:3, Split_var at: 83, Split_thresh at: 0.0, Left leaf value:0, Right leaf value:1.
Training  =======>  Current level:1, Split_var at: 106, Split_thresh at: 0.0, Left leaf value:1, Right leaf value:0.
Training  =======>  Current level:2, Split_var at: 83, Split_thresh at: 0.0, Left leaf value:1, Right leaf value:1.
Training  =======>  Current level:3, Split_var at: 54, Split_thresh at: 0.0, Left leaf value:1, Right leaf value:0.
Training  =======>  Current level:3, Split_var at: 48, Split_thresh at: 0.0, Left leaf value:1, Right leaf value:1.
Training  =======>  Current level:2, Split_var at: 83, Split_thresh at: 0.0, Left leaf value:0, Right leaf value:1.
Training  =======>  Current level:3, Split_var at: 104, Split_thresh at: 0.0, Left leaf value:0, Right leaf value:0.
Training  =======>  Current level:3, Split_var at: 38, Split_thresh at: 168.41666666666666, Left leaf value:1, Right leaf value:
0.
Train accuracy:  0.8861609869304996
Test accuracy:  0.8861748900830484
```