# Neural Network and Machine Learning

Midterm Report for Summer of Science - 2020

Name: Sahasra Ranjan
Mentor: Amitrajit Bhattacharjee

Roll no.: 190050102

Machine Learning is undeniably one of the most influential and powerful technologies in today's world. It is a tool for turning information into knowledge. In the past 50 years, there has been an explosion of data. This mass of data is useless unless we analyse it and find the patterns hidden within. Machine learning techniques are used to automatically find the valuable underlying patterns within complex data that we would otherwise struggle to discover. The hidden patterns and knowledge about a problem can be used to predict future events and perform all kinds of complex decision making.

There are multiple forms of Machine Learning; supervised, unsupervised , semi-supervised and reinforcement learning. Each form of Machine Learning has differing approaches, but they all follow the same underlying process and theory. I'll cover supervised and unsupervised learning for my report.

# Supervised and Unsupervised Learning

The most basic thing to remember is that we already know what our correct output should look like in Supervised Learning. But, we have little or no idea about what our results should look like.

**Supervised Learning:**

- Classification: Spam/Not-spam.
- Regression: Predicting age.

**Unsupervised Learning:**

- Clustering: Grouping based on different variables.
- Non Clustering: Finding structue in chaotic environment.

# Linear Regression

Regression being a part of Supervised Learning is used for estimating data with real-valued output.

## Cost Function

This function measures the performance of a Machine Learning model for given data.

**Hypothesis**: $h_\theta(x) = \theta_0 + \theta_1 x$

**Parameters:** $\theta_0, \theta_1$

**Cost Function:**

$$J(\theta_0, \theta_1) = 1/2m \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 \tag{1.1}$$

**Goal:** Minimize cost function with $\theta_0, \theta_1$ as parameters.

## Gradient Descent

**Basic idea:**

- Start with some $\theta_0, \theta_1$
- Keep changing $\theta_0, \theta_1$ to reduce $J(\theta_0, \theta_1)$ until we end up at minima.

**Algorithm:** repeat until convergence:

$$\theta_j := \theta_j - \alpha \frac{\partial J(\theta_0, \theta_1)}{\partial \theta_j} \tag{1.2}$$

(for $j = 0, 1$ ,here).

**Intution:** If $\alpha$ is too small, the descent can be slow and if too large, descent may fail to converge or even diverge. Gradient descent can converge to a local minimum, even with a fixed learning rate $\alpha$. As we approach local minimum, gradient descent will automatically take smaller steps. So, no need to decrease $\alpha$ over time.

### Gradient Descent for linear regression

Combining gradient descent algorithm with linear regression model, we get:

$$j = 0 : \frac{\partial J(\theta_0, \theta_1)}{\partial \theta_0} = 1/2 \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \tag{1.3}$$

$$j = 1 : \frac{\partial J(\theta_0, \theta_1)}{\partial \theta_1} = 1/2 \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}).x^{(i)} \tag{1.4}$$

Now, we can repeat 1.3 and 1.4 until convergence to obtain the minima.

"Batch" gradient descent: Each step of gradient descent uses all the training examples. For eq. "m" batches in equation 1.1.

## Multivariate Linear Regression

Linear regression involving more than one variable. For eq., Predicting the price of a house based on parameters "Plot Area", "No. of Floors", "Connectivity with markets", etc.

### Multiple Features

The multivariable form of the hypothesis is as follows:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + ... + \theta_n x_n. \tag{1.5}$$

This hypothesis funtion can be concisely represented as:

$$h_\theta(x) = \theta^T x \tag{1.6}$$

where, $\theta^T$ is a 1xn matrix consisting of $\theta_0, \theta_1, \theta_2 ... \theta_n$.

### Gradient Descent for Multiple Variables

Gradient descent formula for Multiple variables will be similar to that of a single variable.

$$\theta_j = \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}).x_j^{(i)} \tag{1.7}$$

Repeating this equation until convergence will give the minima. [1]

[1] $x_0 = 1$ in equation 1.7

### Feature Scaling

Feature Scaling is used to reduce the number of iterations in Gradient Descent.The basic idea of feature scaling is to bring all the features on the same scale. (in general we try to approximate every feature in the range $-1 < x_i < 1$)

Reducing the number of iteration doesn't mean making computation of each step easier. And also it does not affect computational efficiency of Normal Equation.

### Mean Normalisation

Mean Normalisation makes features to have approximately zero mean.

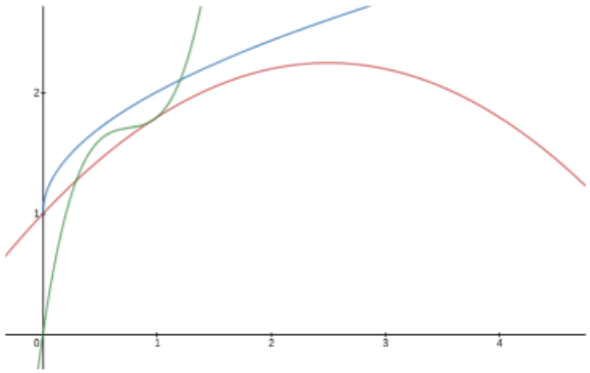
### Learning Rate

If $\alpha$ is too small: slow convergence.

if $\alpha$ is too large: $J(\theta)$ may not decrease on every iteration, or may not converge.

### Polynomial Regression

Selecting proper polynomial for fitting data is very important.




**Red:** Quadratic
**Blue:** Square root funtion $\theta_0 + \theta_1 x + \theta_2 \sqrt{x}$
**Green:** Cubic function

# Normal equation

Normal Equation is a method to solve for $\theta_T$ analytically, by creating a $m \times (n+1)$ matrix $X$ and another $m \times 1$ matrix $Y$.[2]

Mathematically $\theta$ is given as:

$$\theta = (X^T X)^{-1} X^t y \tag{1.8}$$

| Gradient Descent | Normal Equation |
|---|---|
| Need to choose $\alpha$ | No need to choose $\alpha$ |
| Needs many iteration | Don't need to iterate |
| Works well with large n | Slow for large n |

**Reasons for non-invertiblity of $X^T X$**

- Redundant features (linear dependence) [3]
- Too many features (m <= n)

# Classification and Represention

## Classification

The classification problem is just like the regression problem, except that the values we now want to predict take on only a small number of discrete values. For now, we'll discuss the binary classification problem.

## Hypothesis Representation

We may use out old regression algorithm by classifying data on the basis of a threshold. But it will have very poor performance.

We will introduce "Sigmoid Function", also called the "Logistic Function":

$$h_\theta(x) = g(\theta^T x) \tag{1.9}$$

$$z = \theta^T x \tag{1.10}$$

$$g(z) = \frac{1}{1 + e^{-z}} \tag{1.11}$$

This is how the Sigmoid Function looks like:

[2] Every element of first column of matrix $X$ is 1 and other are the feature's coefficient

[3] Eg. Using both $m^2$ & $(feet)^2$ features

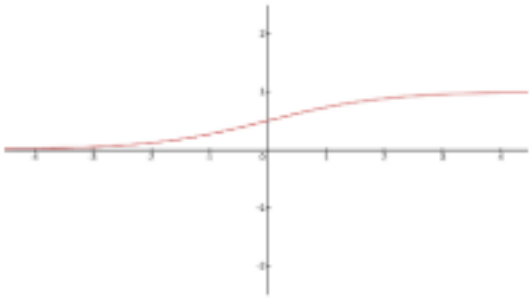


Figure 1.1: Sigmoid Funtion 1.11

## Decision Boundary

The decision boundary is the line that separates the area where y=0 and where y=1. It is similar to the decision boundary for linear regression, the only difference is distribution of values (linear and sigmoid)

## Worked-out Example

```
function J = computeCost(X, y, theta)
  % COMPUTECOST Compute cost for linear regression
  %   J = COMPUTECOST(X, y, theta) computes the cost of using theta as the
  %   parameter for linear regression to fit the data points in X and y

  % Initialize some useful values
  m = length(y); % number of training examples

  J = ((X*theta-y)'*(X*theta-y))/(2*m);
end



function [theta, J_history] = gradientDescent(X, y, theta, alpha, num_iters)
  % GRADIENTDESCENT Performs gradient descent to learn theta
  %   theta = GRADIENTDESCENT(X, y, theta, alpha, num_iters) updates theta by
  %   taking num_iters gradient steps with learning rate alpha

  % Initialize some useful values
  m = length(y); % number of training examples
  J_history = zeros(num_iters, 1);

```

```
  for iter = 1:num_iters,
    theta = theta - alpha*(X'*(X*theta-y))/m;
      % Save the cost J in every iteration
      J_history(iter) = computeCost(X, y, theta);

  end
end
```

Listing 1.1: Octave Implementation for Linear Regression Algorithms

# Logistic Regression

## Cost Function

Cost funtion for logistic regression looks like:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} Cost(h_\theta(x^{(i)}), y^{(i)}) \tag{2.1}$$

$$Cost(h_\theta(x), y) = -\log\left(h_\theta(x)\right) \qquad \text{if } y = 1$$

$$Cost(h_\theta(x), y) = -\log\left(1 - h_\theta(x)\right) \qquad \text{if } y = 0$$


Figure 2.1: Cost Funtion

### Siplified Cost Funtion

This cost funtion can be compressed into a single funtion:

$$Cost(h_\theta(x), y) = -y\log\left(h_\theta(x)\right) - (1-y)\log\left(1 - h_\theta(x)\right) \tag{2.2}$$

A vectorised implementation is:

$h = g(X\theta)$

$J(\theta) = \frac{1}{m}.(-y^T \log h - (1-y)^T \log 1 - h)$

Vectorised implementation for Gradient Descent:

$\theta := \theta - \frac{\alpha}{m} X^T (g(X\theta) - y)$

# Multiclass Classification

## One-vs-all

This approach is when data has more than two categories.We divide our problem into n[1] binary classification problems, in each one, we predict the probability considering one of the category to be +ve and all other to be −ve. Repeating this for all other categories will finally give us all the decision boundaries.

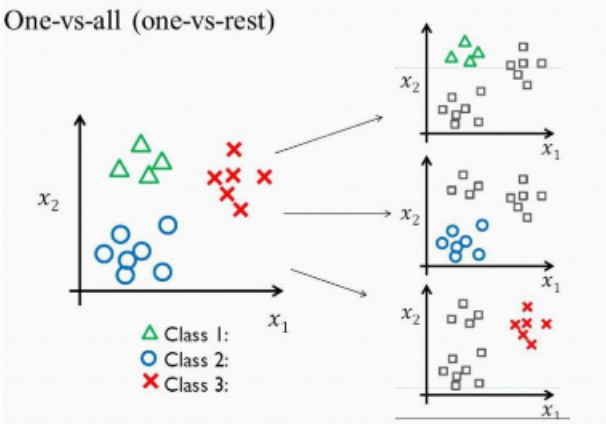



Figure 2.2: One vs all classifiaction method

## Worked-out Example

```
function g = sigmoid(z)
  %SIGMOID Compute sigmoid function
  %   g = SIGMOID(z) computes the sigmoid of z.

  % You need to return the following variables correctly
  g = ones(size(z));
  g = g ./ (1+exp(-z));
end

```

[1] n = no of categories in dataset

```
function [J, grad] = costFunction(theta, X, y)
  %COSTFUNCTION Compute cost and gradient for logistic regression
  %   J = COSTFUNCTION(theta, X, y) computes the cost of using theta as the
  %   parameter for logistic regression and the gradient of the cost
  %   w.r.t. to the parameters.

  % Initialize some useful values
  m = length(y); % number of training examples
  h = sigmoid(X*theta);

  % You need to return the following variables correctly
  J = 1/m * (-y'*log(h)-(1-y)'*log(1-h));

  grad = 1/m * (h-y)'*X;

end

function [J, grad] = costFunctionReg(theta, X, y, lambda)
  %COSTFUNCTIONREG Compute cost and gradient for logistic regression with regularization
  %   J = COSTFUNCTIONREG(theta, X, y, lambda) computes the cost of using
  %   theta as the parameter for regularized logistic regression and the
  %   gradient of the cost w.r.t. to the parameters.

  % Initialize some useful values
  m = length(y); % number of training examples

  [c,g] = costFunction(theta, X, y);

  temp = lambda/(2*m) * theta'*theta;
  temp = temp - lambda/(2*m)*theta(1,1)^2;
  J = c+temp;


  grad = g + lambda/m * theta';
  grad(1,1) = g(1,1);

end

function out = mapFeature(X1, X2)
  % MAPFEATURE Feature mapping function to polynomial features
  %
  %   MAPFEATURE(X1, X2) maps the two input features
  %   to quadratic features used in the regularization exercise.
  %
  %   Returns a new feature array with more features, comprising of
  %   X1, X2, X1.^2, X2.^2, X1*X2, X1*X2.^2, etc..

```

```
  degree = 6;
  out = ones(size(X1(:,1)));
  for i = 1:degree
      for j = 0:i
          out(:, end+1) = (X1.^(i-j)).*(X2.^j);
      end
  end
end

function plotDecisionBoundary(theta, X, y)
  %PLOTDECISIONBOUNDARY Plots the data points X and y into a new figure with
  %the decision boundary defined by theta
  %   PLOTDECISIONBOUNDARY(theta, X,y) plots the data points with + for the
  %   positive examples and o for the negative examples. X is assumed to be
  %   a either
  %   1) Mx3 matrix, where the first column is an all-ones column for the
  %      intercept.
  %   2) MxN, N>3 matrix, where the first column is all-ones

  % Plot Data
  plotData(X(:,2:3), y);
  hold on

  if size(X, 2) <= 3
      % Only need 2 points to define a line, so choose two endpoints
      plot_x = [min(X(:,2))-2,  max(X(:,2))+2];

      % Calculate the decision boundary line
      plot_y = (-1./theta(3)).*(theta(2).*plot_x + theta(1));

      % Plot, and adjust axes for better viewing
      plot(plot_x, plot_y)

      % Legend, specific for the exercise
      legend('Admitted', 'Not admitted', 'Decision Boundary')
      axis([30, 100, 30, 100])
  else
      % Here is the grid range
      u = linspace(-1, 1.5, 50);
      v = linspace(-1, 1.5, 50);

      z = zeros(length(u), length(v));
      % Evaluate z = theta*x over the grid
      for i = 1:length(u)
          for j = 1:length(v)
              z(i,j) = mapFeature(u(i), v(j))*theta;
          end
```

```
      end
      z = z'; % important to transpose z before calling contour

      % Plot z = 0
      % Notice you need to specify the range [0, 0]
      contour(u, v, z, [0, 0], 'LineWidth', 2)
  end
  hold off

end

function p = predict(theta, X)
  %PREDICT Predict whether the label is 0 or 1 using learned logistic
  %regression parameters theta
  %   p = PREDICT(theta, X) computes the predictions for X using a
  %   threshold at 0.5 (i.e., if sigmoid(theta'*x) >= 0.5, predict 1)

  m = size(X, 1); % Number of training examples

  % You need to return the following variables correctly
  p = zeros(m, 1);

  p = sigmoid(X*theta);

  for i = [1:m],
    if p(i,1) >= 0.5,
      p(i,1) = 1;
     else
      p(i,1) = 0;
    end
  end
end
```

Listing 2.1: Octave Implementation for Logistic Regression Algorithms

# Decision Tree

A decision tree is a map of the possible outcomes of a series of a related choices. It allows to weigh possible actions against one another based of various factors.

It uses a tree-like model of decision. It typically starts with a single node, which branches into possible outcomes. Each of those outcomes leads to additional nodes, which branch off into other possiblities.

| Day | *Outlook* | *Temperature* | *Humidity* | *Wind* | *PlayTennis* |
|---|---|---|---|---|---|
| D1 | Sunny | Hot | High | Weak | No |
| D2 | Sunny | Hot | High | Strong | No |
| D3 | Overcast | Hot | High | Weak | Yes |
| D4 | Rain | Mild | High | Weak | Yes |
| D5 | Rain | Cool | Normal | Weak | Yes |
| D6 | Rain | Cool | Normal | Strong | No |
| D7 | Overcast | Cool | Normal | Strong | Yes |
| D8 | Sunny | Mild | High | Weak | No |
| D9 | Sunny | Cool | Normal | Weak | Yes |
| D10 | Rain | Mild | Normal | Weak | Yes |
| D11 | Sunny | Mild | Normal | Strong | Yes |
| D12 | Overcast | Mild | High | Strong | Yes |
| D13 | Overcast | Hot | Normal | Weak | Yes |
| D14 | Rain | Mild | High | Strong | No |

Figure 3.1: Dataset for possiblity of a tennis match

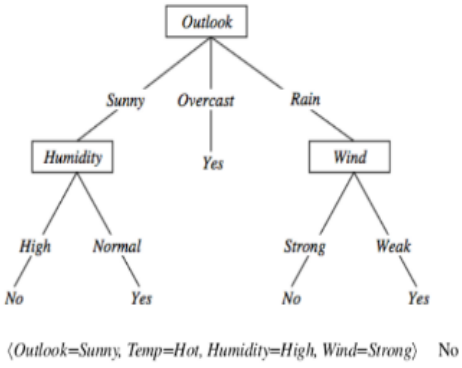



Figure 3.2: Decision Tree for the same

## Advantages and Disadvantages of Decision Trees

### Advantages:

- Performs classification without requiring much computation.
- Provides clear indication of important fields for prediction of classification.

**Disadvantages:**

- Less appropriate for predicting continious attributes.
- Computationally expensive to train

## Creating a Decision Tree

For every node, we have to create subtrees with all the possiblities. and then further repeat for other features.

For eg., In the tennis match problem, for the first node let's check outlook, since having three possiblity (viz. Sunny, Overcast, Rainy), we created three subtrees and then further we keep asking for other features like Humidity & wind to get the final tree.

## Greedy Approach for creating decision tree

Greedy approach is implemented by making an optimal local choice at each node. By making these local optimal choices, we reach the approximate optimal solution globaly.

The algorithm can be summarized as:

1. At each stage (node), pick out the best feature as the test conditiion.
2. Now split the node into possibel outcomes (internal nodes)
3. Repeat the above steps till all the test conditions have been exhausted into leaf nodes.

## Continuous Features

There might be some feature which are not categorical, for these we need to create possibilities on the basis of aprropriate ranges. One such tree is shown below:

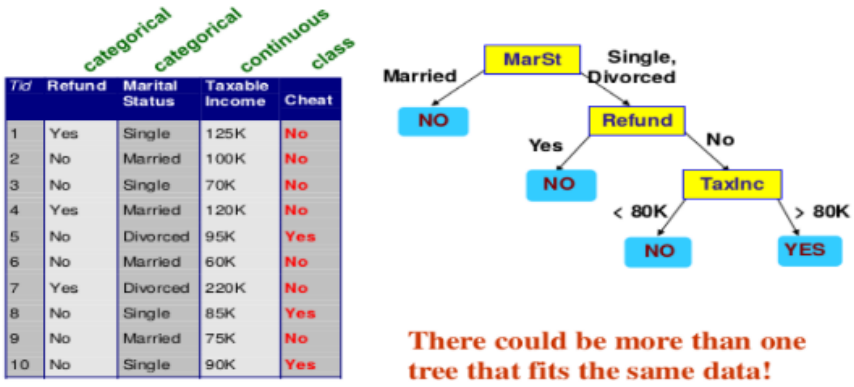



Figure 3.3: Decision tree with continuous feature

## Entropy

In the most layman terms, Entropy is nothing but the **The measure of disorder**. Why is it important to study entropy for machine learning?

Entropy is a measure of disorder or uncertainity and the goal of machine learning models and Data Scientists in general is to reduce uncertainity.

The Mathematical formula for entropy is -

$$E(S) = \sum_{i=1}^{c} -p_i log_2 p_i \tag{3.1}$$

Where $p_i$ is the frequentist probability of an element/class $i$ in out data.

Let's say we have only two classes, a positive and a negative class. Out of 100 data, suppose that 70 belongs to $-$ve class and 30 to $+$ve. Then, P$+$ will be 0.3 and P$-$ will be 0.7. Entropy $E$ will be given by:

$$E = -\frac{3}{10} \times log_2 \frac{3}{10} - \frac{7}{10} \times log_2 \frac{7}{10} \approx \mathbf{0.88} \tag{3.2}$$

## Information Gain

Information gain is basically how much Entropy is removed after training a decision tree.

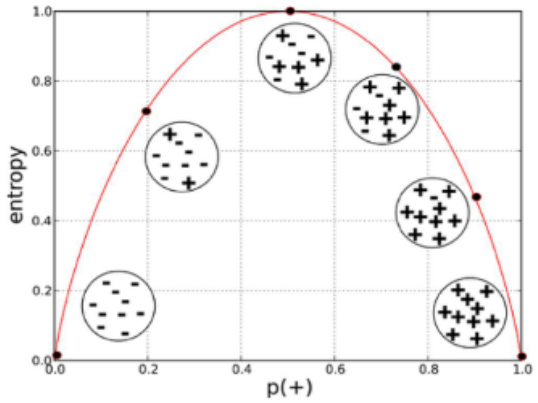



Figure 3.4: Entropy distribution frequentist probability

**Higher information gain = more entropy removed.**
In technical terms, Information Gain from X on Y is defined as:

$$IG(Y, X) = E(Y) - E(Y|X) \tag{3.3}$$

Basics of information gain is well explained here: A Simple Explanation of Information Gain and Entropy

## Example: Decision Tree

Consider an example where we are building a decision tree to predict whether a loan given to a person would result in a write-off or not. Our entire population consists of 30 instances. 16 belong to the write-off class and the other 14 belong to the non-write-off class. We have two features, namely "Balance" that can take on two values: "< 50K" or "> 50K" and "Residence" that can take on three values: "OWN", "RENT" or "OTHER". I'm going to show you how a decision tree algorithm would decide what attribute to split on first and what feature provides more information, or reduces more uncertainty about our target variable out of the two using the concepts of Entropy and Information Gain. The dots are the data points with class right-off and the stars are the non-write-offs. Splitting the parent node on attribute balance gives us 2 child nodes. The left node gets 13 of the total observations with 12/13 ( 0.92 probability) observations from the write-off class and only 1/13( 0.08 probability) observations from the non-write of class. The right node gets 17 of the total observation with 13/17( 0.76 probability) observations from the non-write-off class and 4/17 ( 0.24 probability) from the write-off class.

Let's calculate[1] the entropy for the parent node and see how much uncertainty the

[1]See this for calculations and further refrences: Entropy: How Decision Trees Make Decisions

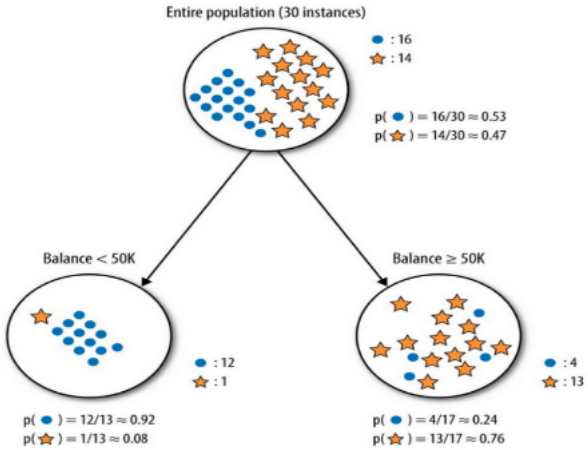



Figure 3.5: Feature 1: Balance

tree can reduce by splitting on Balance. Splitting on feature ,"Balance" leads to an information gain of 0.37 on our target variable. Let's do the same thing for feature, "Residence" to see how it compares.

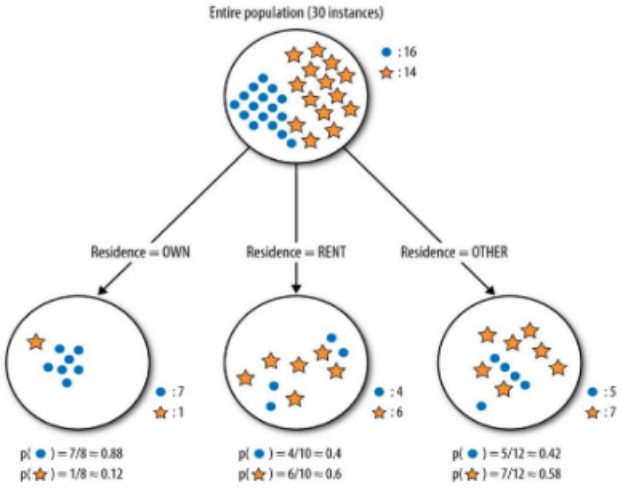



Figure 3.6: Feature 2: Residence

Splitting the tree on Residence gives us 3 child nodes. The left child node gets 8 of the total observations with 7/8 (0.88 probability) observations from the write-off class and only 1/8 (0.12 probability) observations from the non-write-off class. The middle child nodes gets 10 of the total observations with 4/10 (0.4 probability) observations of

the write-off class and 6/10( 0.6 probability) observations from the non-write-off class. The right child node gets 12 of the total observations with 5/12 ( 0.42 probability) observations from the write-off class and 7/12 ( 0.58 ) observations from the non-write-off class. We already know the entropy for the parent node. We simply need to calculate the entropy after the split to compute the information gain from "Residence"

The information gain from feature, Balance is almost 3 times more than the information gain from Residence! If you go back and take a look at the graphs you can see that the child nodes from splitting on Balance do seem purer than those of Residence. However the left most node for residence is also very pure but this is where the weighted averages come in play. Even though that node is very pure, it has the least amount of the total observations and a result contributes a small portion of it's purity when we calculate the total entropy from splitting on Residence. This is important because we're looking for overall informative power of a feature and we don't want our results to be skewed by a rare value in a feature.

By itself the feature, Balance provides more information about our target variable than Residence. It reduces more disorder in our target variable. A decision tree algorithm would use this result to make the first split on our data using Balance. From here on, the decision tree algorithm would use this process at every split to decide what feature it is going to split on next. In a real world scenario , with more than two features the first split is made on the most informative feature and then at every split the information gain for each additional feature needs to be recomputed because it would not be the same as the information gain from each feature by itself. The entropy and information gain would have to be calculated after one or more splits have already been made which would change the results. A decision tree would repeat this process as it grows deeper and deeper till either it reaches a pre-defined depth or no additional split can result in a higher information gain beyond a certain threshold which can also usually be specified as a hyper-parameter!

There you have it! You now know what entropy and information gain are and how they are computed. You understand how a decision tree either by itself or in a tree based ensemble decides on the best order of features to split on and decides when to stop when it trains itself on given data. If you every have to explain the intricacies of how decision trees work to someone, hopefully you won't do too bad.

## Worked-out Implementation

```
1 import pandas as pd
2 import numpy as np
3
4 eps = np.finfo(float).eps
5
```

```
from numpy import log2 as log
dataset = {'Taste':['Salty','Spicy','Spicy','Spicy','Spicy','Sweet','Salty','Sweet','Spicy','Salty'],
    'Temperature':['Hot','Hot','Hot','Cold','Hot','Cold','Cold','Hot','Cold','Hot'],
    'Texture':['Soft','Soft','Hard','Hard','Hard','Soft','Soft','Soft','Soft','Hard'],
    'Eat':['No','No','Yes','No','Yes','Yes','No','Yes','Yes','Yes']}

df = pd.DataFrame(dataset, columns=['Taste','Temperature','Texture','Eat'])

def find_entropy(df):
    '''
    Function to calculate the entropy of the label i.e. Eat.
    '''
    Class = df.keys()[-1]
    entropy = 0
    values = df[Class].unique()
    for value in values:
        fraction = df[Class].value_counts()[value]/len(df[Class])
        entropy += -fraction*np.log2(fraction)
    return entropy


def find_entropy_attribute(df,attribute):
    '''
    Funtion to calculate the entropy of all the features.
    '''
    Class = df.keys()[-1]
    target_variables = df[Class].unique()
    variables = df[attribute].unique()
    entropy2 = 0

    for variable in variables:
        entropy = 0
        for target_variable in target_variables:
            num = len(df[attribute][df[attribute]==variable][df[Class] ==target_variable])
            den = len(df[attribute][df[attribute]==variable])
            fraction = num/(den+eps)
            entropy += -fraction*log(fraction+eps)
        fraction2 = den/len(df)
        entropy2 += -fraction2*entropy
    return abs(entropy2)


def find_winner(df):
    '''
    Function to find the feature with the highest information gain
```

```
    '''
    Entropy_att = []
    IG = []
    for key in df.keys()[:-1]:
        IG.append(find_entropy(df) - find_entropy_attribute(df,key))

    return df.keys()[:-1][np.argmax(IG)]

def get_subtable(df, node, value):
    '''
    Function to get a subtable of met conditions.

    node: Column name
    value: Unique value of the column
    '''
    return df[df[node] == value].reset_index(drop=True)

def buildTree(df,tree=None):
    '''
    Function to build the ID3 Decision Tree.
    '''
    Class = df.keys()[-1]
    #Here we build our decision tree

    #Get attribute with maximum information gain
    node = find_winner(df)

    #Get distinct value of that attribute e.g Salary is node and Low,
   Med and High are values
    attValue = np.unique(df[node])

    #Create an empty dictionary to create tree
    if tree is None:
        tree={}
        tree[node] = {}

   #We make loop to construct a tree by calling this function
   recursively.
    #In this we check if the subset is pure and stops if it is pure.

    for value in attValue:

        subtable = get_subtable(df,node,value)
        clValue,counts = np.unique(subtable['Eat'],return_counts=True
   )

        if len(counts)==1:#Checking purity of subset
            tree[node][value] = clValue[0]
        else:
            tree[node][value] = buildTree(subtable) #Calling the
   function recursively
```

```
98
99      return tree
100
101
102 tree = buildTree(df)
103
104
105 def predict(inst,tree):
106     '''
107     Function to predict for any input variable.
108     '''
109     #Recursively we go through the tree that we built earlier
110
111     for nodes in tree.keys():
112
113         value = inst[nodes]
114         tree = tree[nodes][value]
115         prediction = 0
116
117         if type(tree) is dict:
118             prediction = predict(inst, tree)
119         else:
120             prediction = tree
121             break;
122
123     return prediction
124
125 data = {'Taste':'Salty','Temperature':'Cold','Texture':'Hard'}
126
127 inst = pd.Series(data)
128
129 prediction = predict(inst,tree)
130
131 print(prediction)
132
133 # Prints "Yes"
```

Listing 3.1: Python Implementation for Decision Tree

# Naive Bayes

## Naive Bayes

Naive Bayes methods are a set of supervised learning algorithms based on applying Bayes' theorem with the "naive" assumption of conditional independence between every pair of features given the value of the class variable.

What actually Bayes' theorem is?

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{4.1}$$

$P(A|B)$ is the probability of **A** happening, given that **B** has occured.

Why "Naive"? Because the presence of one particular feature does not affect the other.

Without going to deep, let's see an example:

### The Golf Match Problem

Consider the problem of playing golf, dataset for the same:
Bayes theorem for this example can be rewritten as:

$$P(y|X) = \frac{P(X|y)P(y)}{P(X)} \tag{4.2}$$

The variable **y** is the class variable (play golf), which represents if it is suitable to play golf or not given the conditions. Variable **X** is a matrix representing the parameters/features.

$\mathbf{X} = (x_1, x_2, x_3, ..., x_n)$ $x_1, x_2, ...x_n$ represent the features[1].

[1] temperature, humidity and windy (here)

| | OUTLOOK | TEMPERATURE | HUMIDITY | WINDY | PLAY GOLF |
|---|---|---|---|---|---|
| 0 | Rainy | Hot | High | False | No |
| 1 | Rainy | Hot | High | True | No |
| 2 | Overcast | Hot | High | False | Yes |
| 3 | Sunny | Mild | High | False | Yes |
| 4 | Sunny | Cool | Normal | False | Yes |
| 5 | Sunny | Cool | Normal | True | No |
| 6 | Overcast | Cool | Normal | True | Yes |
| 7 | Rainy | Mild | High | False | No |
| 8 | Rainy | Cool | Normal | False | Yes |
| 9 | Sunny | Mild | Normal | False | Yes |
| 10 | Rainy | Mild | Normal | True | Yes |
| 11 | Overcast | Mild | High | True | Yes |
| 12 | Overcast | Hot | Normal | False | Yes |
| 13 | Sunny | Mild | High | True | No |

Figure 4.1: Dataset for possiblity of a golf match

$$P(y|x_1, ..., x_n) = \frac{P(x_1|y)P(x_2|y)...P(x_n|y)P(y)}{P(x_1)P(x_2)...P(x_n)} \tag{4.3}$$

These values can be obtained by looking at the dataset and substituting them into the equation will give us the result.

In our case, the the class variable($\mathbf{y}$) has only two outcomes, yes or no. Therefore, we need to find class $\mathbf{y}$ with maximum probability.

$$y = argmax_y P(y) \prod_{i=1}^{n} P(x_i|y) \tag{4.4}$$

## Types of Naive Bayes classifier:

### Miltinomial Naive Bayes:

This is mostly used for the document classification problem, i.e whether a document belongs to the category of sports, politics, technology etc. The features/predictors used by the classifier are the frequency of the words present in the document.

### Bernoulli Naive Bayes

This is similar to the multinomial Naive Bayes but the predictors are boolean variables. The parameters that we use to predict the class variable take up only values yes or no, for example, if a word occurs in the text or not.

### Gaussian Naive Bayes

When the predictors take up a continuous value and are not discrete, we assume that these values are sampled from a gaussian distribution.

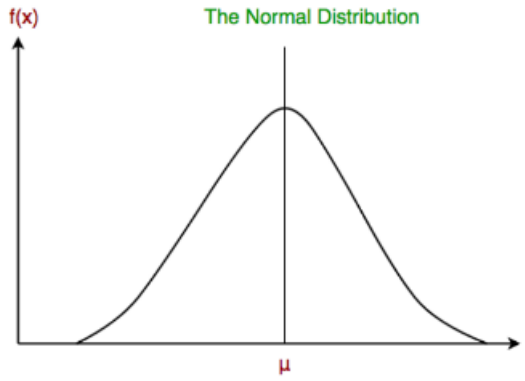



Figure 4.2: Gaussian Distribution(Normal Distribution)

The formula for conditional probability changes to:

$$P(x_i|y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} e^{-\frac{(x_i-\mu_y)^2}{2\sigma_y^2}} \tag{4.5}$$

## Laplace Smoothing

It is problematic when a frequency-based probability is zero because it will wipe out all the information in the other probabilities.

A solution would be **Laplace smoothing**, which is a technique for smoothing categorical data. A small-sample correction, or **pseudo-count**, will be incorporated in every probability estimate. Consequently, no probability will be zero. this is a way of regularizing Naive Bayes, and when the pseudo-count is zero, it is called Laplace smoothing. While in the general case it is often called **Lidstone smoothing.**

$$P_{i,\ \alpha-smoothed} = \frac{x_i + \alpha}{N + \alpha d} \tag{4.6}$$

where, $\alpha > 0$ the "pseudocount" is a smoothing parameter. And, $1/d$ is the Uniform Probability.

### Note

In practice, we use logs to represent probabilities:

$$\log\left(P(x_1|y)P(x_2|y)...P(x_n|y)\right) = \log P(x_1|y) + \log P(x_2|y) + ... + \log P(x_n|y) \tag{4.7}$$

## Applications

Naive Bayes algorithms are mostly used in sentiment analysis, spam filtering, recommendation systems etc. They are fast and easy to implement but their biggest disadvantage is that the requirement of predictors to be independent. In most of the real-life cases, the predictors are dependent, this hinders the performance of the classifier.

## Worked-out Implementation[2]

```
import numpy as np
import pandas as pd

mush = pd.read_csv("./mushrooms.csv")
# This data consist of missing data with '?' as value. All the
    missing values are from single column.

# To remove it.
mush.replace('?',np.nan,inplace=True)
print(len(mush.columns),"columns, after dropping NA,",len(mush.dropna
    (axis=1).columns))

mush.dropna(axis=1,inplace=True)

target = 'class'
features = mush.columns[mush.columns != target]
classes = mush[target].unique()

test = mush.sample(frac=0.3)
mush = mush.drop(test.index)

# Probabilties Calculation
probs = {}
```

[2]**Output:**
23 columns, after dropping NA, 22
Test 1
5671 correct of 5687
Accuracy: 0.997186565851943
Test 2
2433 correct of 2437
Accuracy: 0.9983586376692655

This implementation of Naive Bayes Algorithm has an accuracy of approximately 99.8% which is good in the first go.

```
probcl = {}
for x in classes:
    mushcl = mush[mush[target]==x][features]
    clsp = {}
    tot = len(mushcl)
    for col in mushcl.columns:
        colp = {}
        for val,cnt in mushcl[col].value_counts().iteritems():
            pr = cnt/tot
            colp[val] = pr
        clsp[col] = colp
    probs[x] = clsp
    probcl[x] = len(mushcl)/len(mush)


def probabs(x):
    #X - pandas Series with index as feature
    if not isinstance(x,pd.Series):
        raise IOError("Arg must of type Series")
    probab = {}
    for cl in classes:
        pr = probcl[cl]
        for col,val in x.iteritems():
            try:
                pr *= probs[cl][col][val]
            except KeyError:
                pr = 0
        probab[cl] = pr
    return probab


def classify(x):
    probab = probabs(x)
    mx = 0
    mxcl = ''
    for cl,pr in probab.items():
        if pr > mx:
            mx = pr
            mxcl = cl
    return mxcl

#Train data
b = []
for i in mush.index:
    b.append(classify(mush.loc[i,features]) == mush.loc[i,target])
print("Test 1")
print(sum(b),"correct of",len(mush))
print("Accuracy:", sum(b)/len(mush))


#Test data
```

```
b = []
for i in test.index:
    b.append(classify(test.loc[i,features]) == test.loc[i,target])
print("Test 2")
print(sum(b),"correct of",len(test))
print("Accuracy:",sum(b)/len(test))
```

Listing 4.1: Python Implementation for Naive Bayes

# k-Nearest Neighbours

K-Nearest Neighbours is one of the most basic yet essential algorithms in Machine Learning. It has intense application in pattern recognition, data mining and intrusion detection.

## The Algorithm

1. Load the data
2. Initialize K
3. For each example in data
    - Calculate the distance between the query example and the current example from the data
    - Add the distance and the index of the example to an ordered collection
4. Sort the collection in ascending order.
5. Pick the first K entries from the sorted collection
6. Get the labels of the selected K entries
7. If regression, return the mean else if classification, return the mode of the K labels

## Chosing the right value for K

To select the K that's right for your data, we run the KNN algorithm several times with different values of K and choose the K that reduces the number of errors we encounter while maintaining the algorithm's ability to accurately make predictions when it's given data it hasn't seen before.

Here are some things to keep in mind:

1. As we decrease the value of K to 1, our predictions become less stable.
2. Inversely, as we increase the value of K, our predictions become more stable due to majority voting (up to a certain point). Eventually, increasing K will increase the error.

3. In cases where we are taking majority vote, we usually make "K" an odd number to have a tiebreaker.

### Advantages

- Simple and easy to implement.
- No need to build a model, tune several parameters, or make additional assumptions.
- Versatile, can be used for classification, regresion and search as well.

### Disadvantages

- Slow with increase in number of examples and/or predictors.

## Applications

kNN's main disadvantage of becoming significantly slower as the volume of data increases makes it an impractical choice in environments where predictions need to be made rapidly. Moreover, there are faster algorithms that can produce more accurate classification and regression results.

However, provided you have sufficient computing resources to speedily handle the data you are using to make predictions, KNN can still be useful in solving problems that have solutions that depend on identifying similar objects. An example of this is using the KNN algorithm in recommender systems, an application of KNN-search.

## Worked-out Implementation

```
import numpy as np
from matplotlib import pyplot as plt

class KNearestNeighbor(object):
    def __init__(self, k=3):
        self.k = k

    def fit(self, X, y):
        # Store the original points

        self.X = X
        self.y = y

        return self

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

        # Initialize a zero distance matrix
        dists = np.zeros((X.shape[0], self.X.shape[0]))
```

```
20
21          # Loop through all possible pairs and compute their distances
22          for i in range(dists.shape[0]):
23              for j in range(dists.shape[1]):
24                  dists[i, j] = self.distance(X[i], self.X[j])
25
26
27          # Sort the distance array row-wise, and select the top k
        indexes for each row
28          indexes = np.argsort(dists, axis=1)[:,:self.k]
29
30          # Compute the mean of the values
31          mean = np.mean(self.y[indexes], axis=1)
32
33          return mean
34
35      def distance(self, x, y):
36          return np.sqrt(np.dot(x - y, x - y))
37
38  x = np.linspace(0, 5, 20)
39
40  m = 1.5
41  c = 1
42  y = m * x + c + np.random.normal(size=(20,))
43
44  plt.plot(x, y, 'x')
45
46  model = KNearestNeighbor(k=3)
47
48  model.fit(x,y)
49
50  predicted = model.predict(x.reshape(-1, 1))
51
52  plt.plot(
53      x, y, "x",
54      x, model.predict(x), "-o"
55  )
56  plt.legend(["actual", "prediction"])
57
58  plt.show()
```

Listing 5.1: Python Implementation for K-Nearest Neighbours Algorithm

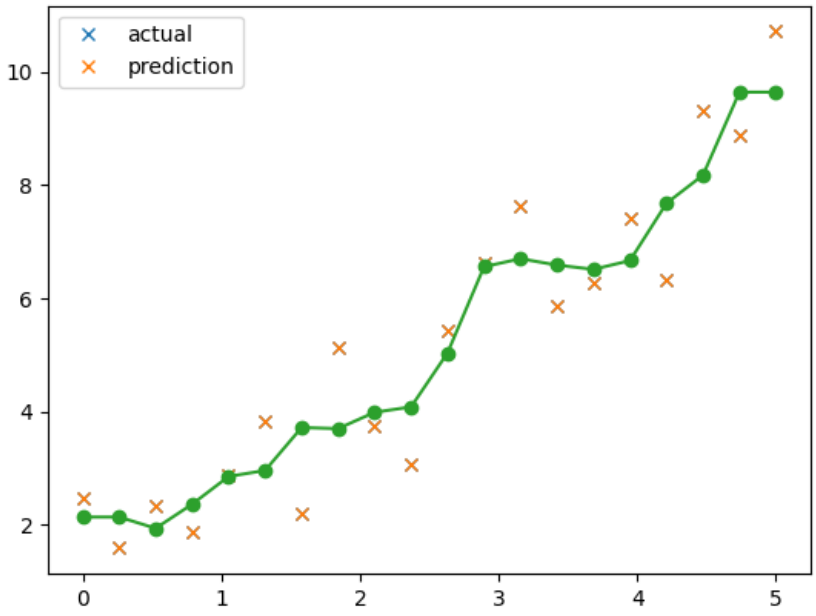



Figure 5.1: Matplotlib plot for the above implementation

# Revised Plan of Action

I covered most of the topics that I planned for the first phase except I covered "Naive Bayes" and not "SVM" algorithm.

## Topics covered so far:

- Linear Regression
- Logistic Regression
- Naive Bayes
- Decesion Algorithm
- K-Nearest Neighbors Algorithm

## Topics still left to be covered:

- Neural Network Representation and Application.
- Deep Learning
  - Backpropagation
  - Convolution Neural Network (CNN)
  - Recurrent Neural Network (RNN)
- Support Vector Mechanism (SVM)
- Unsupervised Learning
  - K-Means Clustering Algorithm

## Revised Timeline:

- **Week 4:** Neural Network and Backpropagation algorithm
- **Week 5:** CNN and RNN

- **Week 6:** SVM and Unsupervised Learning algorithms
- **Week 7:** Applying the algorithms covered.

## Refrences:

As per the guidance of my Mentor, I'll mostly be following:

- Andrew NG's Machine Learning Course on Coursera
- "Deep Learning" book by Ian Goodfellow, Yoshua Bengio and Aaron Courville
- Linear Algebra report by Zico Kolter (Carnegie Mellon University)
- Towards Data Science blogs
- Kaggle discussion forums

Link to worked-out examples: SoS - Neural Network and Machine Learning - Sahasra Ranjan