Lecture 4

Predictive Analytics I

Dr. Lydia Wahid

Agenda

- Introduction
- K-Nearest Neighbor
- Applying KNN on Big Data
- Decision Trees
- Applying Decision Trees on Big Data
- Naïve Bayes classifier
- Applying Naïve Bayes classifier on Big Data
- Performance Evaluation of classifiers

Introduction

Introduction

- **Predictive Analytics** use **Predictive Data Mining** techniques which use **Supervised Machine Learning** techniques.

Introduction

- *Classification* is an instance of Supervised Machine Learning and is widely used for prediction purposes.

- In classification, a classifier is given a set of examples that are already classified (i.e. given a class label), and from these examples, the classifier learns to assign a label to unseen examples.

- Examples of classification problems include:
  - Given an email, classify if it is spam or not.
  - Given a handwritten character, classify it as one of the known characters.

Introduction

- **Examples of Classification Techniques:**
  - **K-Nearest Neighbor (KNN)**
  - **Decision Trees (DT)**
  - **Naïve Bayes**
  - **Support Vector Machines (SVM)**
  - **Neural Networks**

K-Nearest Neighbor

K-Nearest Neighbor

- K-nearest neighbors is an algorithm that stores all available cases and classifies new cases based on a **similarity measure** (e.g., distance functions).

- KNN algorithm at the training phase just stores the dataset and when it gets new data, then it classifies that data into a category that is much similar to the new data.

K-Nearest Neighbor

- The K-NN working can be explained on the basis of the below algorithm:

Step 1 – For implementing any algorithm, we need dataset. So during the first step of KNN, we must load the training as well as test data.

Step 2 – Next, we need to choose the value of K i.e. the nearest data points. K can be any integer.

Step 3 – For each point in the test data do the following –

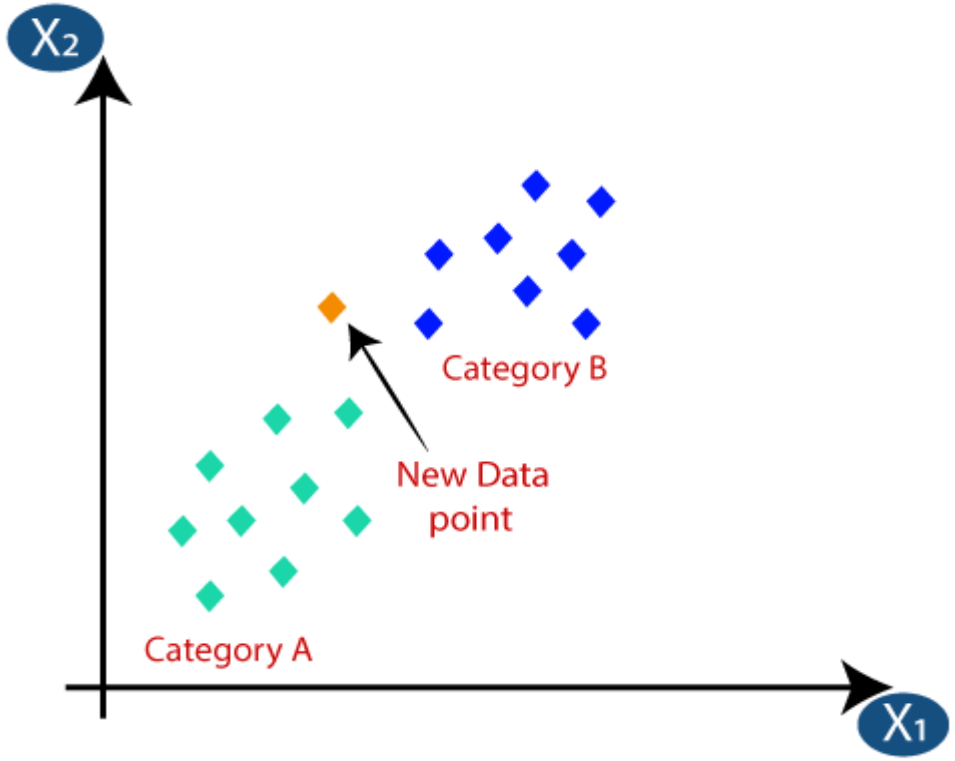
3.1 – Calculate the distance between test data and each row of training data with the help of any of the method namely: Euclidean, Manhattan or Hamming distance.

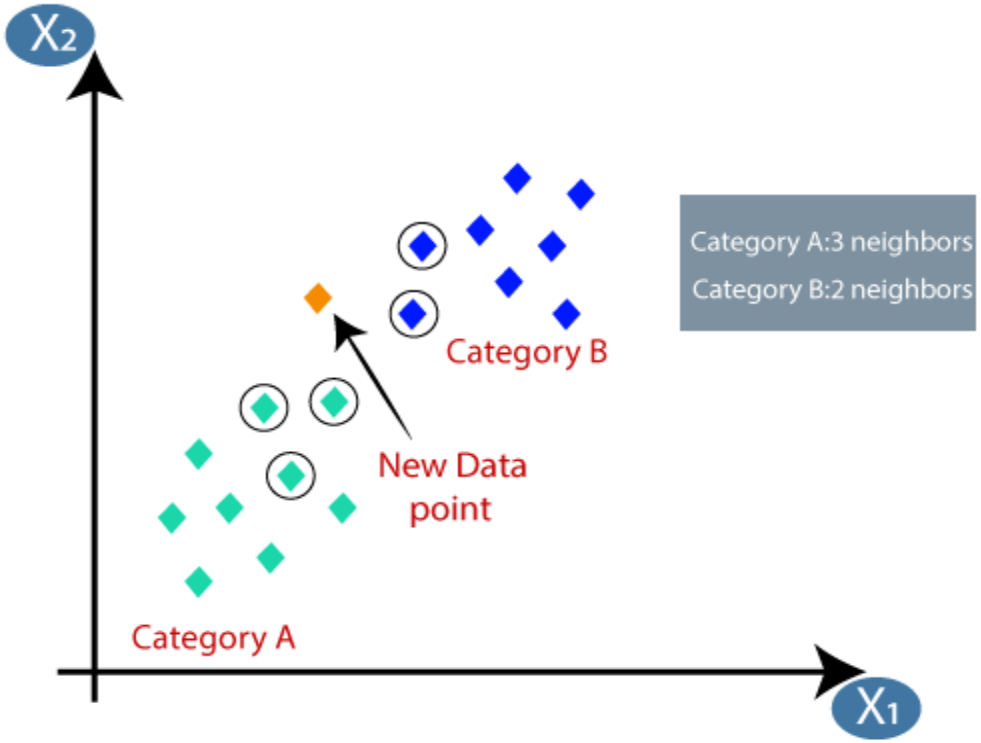
3.2 – Now, based on the distance value, sort them in ascending order.

3.3 – Next, it will choose the top K rows from the sorted array.

3.4 – Now, it will assign a class to the test point based on most frequent class of these rows.

K-Nearest Neighbor





K-Nearest Neighbor

- **Distance functions:**

1. Euclidean Distance:

$$D = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

n is the number of features

2. Manhattan Distance:

$$D = \sum_{i=1}^{n} |x_i - y_i|$$

n is the number of features

K-Nearest Neighbor

- **Distance functions:**

3. Hamming Distance:
   - It is a measure of the number of instances in which corresponding symbols are different in two strings of equal length. It is suitable for categorial features.

$$D_H = \sum_{i=1}^{n} |x_i - y_i|$$

n is the number of features

$$x = y \Rightarrow D = 0$$

$$x \neq y \Rightarrow D = 1$$

K-Nearest Neighbor

- Example:

Height (in cms)	Weight (in kgs)	T Shirt Size
158	58	M
158	59	M
158	63	M
160	59	M
160	60	M
163	60	M
163	61	M
160	64	L
163	64	L
165	61	L
165	62	L
165	65	L

New customer has height 161cm and weight 61kg.

What is his T Shirt Size?

K-Nearest Neighbor

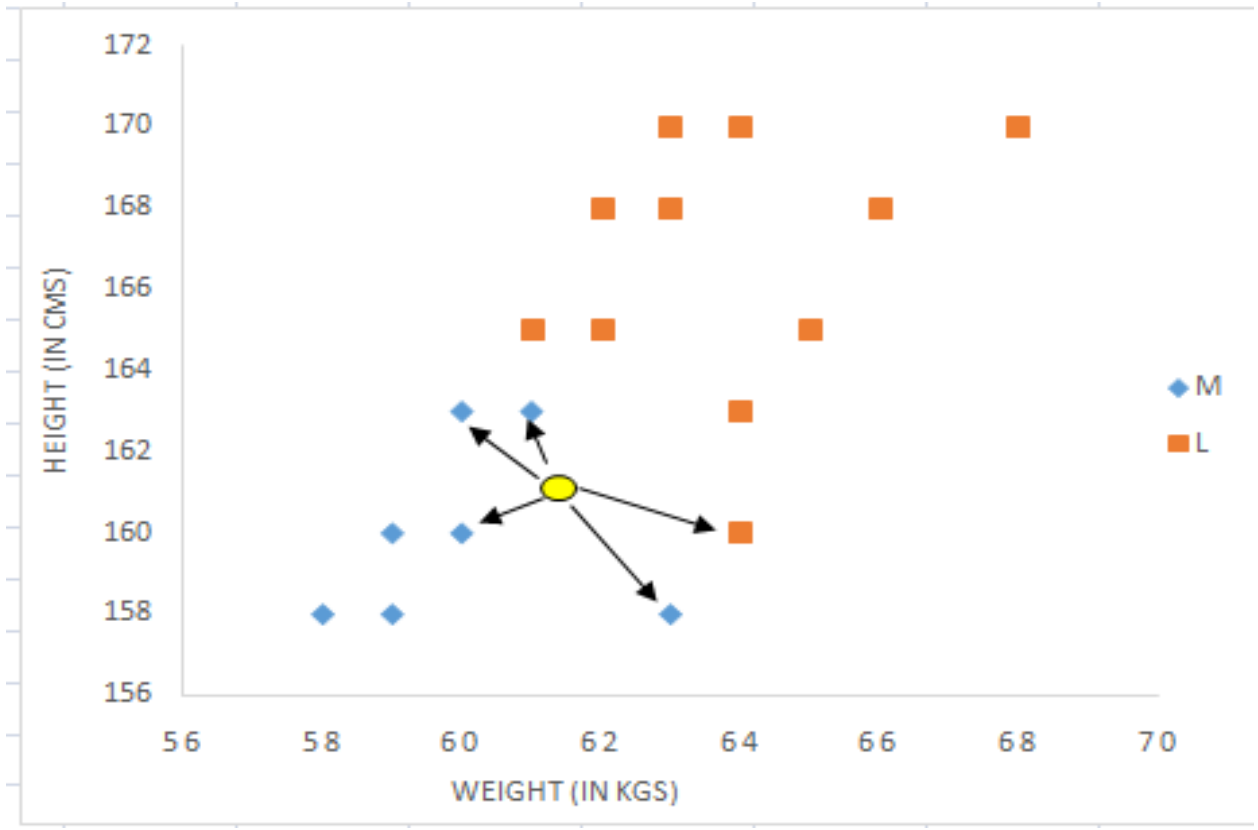
- **Example:**

Height (in cms)	Weight (in kgs)	T Shirt Size	Distance	
158	58	M	4.2	
158	59	M	3.6	
158	63	M	3.6	
160	59	**M**	2.2	**3**
160	60	**M**	1.4	**1**
163	60	**M**	2.2	**3**
163	61	**M**	2.0	**2**
160	64	**L**	3.2	**5**
163	64	L	3.6	
165	61	L	4.0	
165	62	L	4.1	
165	65	L	5.7	

Euclidean Distance is used.
For K=5,
T shirt Size=M

K-Nearest Neighbor

- **Example:**



K-Nearest Neighbor

What are the limitations of KNN?

K-Nearest Neighbor

- **Normalization and Standardization:**
  - When independent variables in training data are measured in different units, it is important to **scale the variables** before calculating distance.
  - For example, if one variable is based on height in cms, and the other is based on weight in kgs then **height will influence more** on the distance calculation.
  - Scaling the variables can be done by any of the following methods:

$$Xs = \frac{X - min}{\max - min}$$

Normalization

$$Xs = \frac{X - mean}{s.d.}$$

Standardization

K-Nearest Neighbor

- **Handling categorial features:**
  - Hamming Distance can be used
  - Can assign a number to each category

K-Nearest Neighbor

- **How to find best K value?**
  - **Cross-validation** is a way to find out the **optimal K value**. It estimates the validation error rate by holding out a subset of the training set from the model building process.
  - Cross-validation (let's say 10 fold validation) involves randomly dividing the **training set** into 10 groups, or folds, of approximately equal size. 90% data is used to train the model and remaining 10% to validate it. The error rate is then computed on the 10% validation data. This procedure repeats 10 times each time with a different fold. It results to 10 estimates of the validation error which are then averaged out.
  - The process is repeated for different values of K. The value of K that yields the smallest average error is selected.

Applying KNN on Big Data

Applying KNN on Big Data

- Despite the promising results shown by the k-NN in a wide variety of problems, it lacks scalability to address Big datasets.
- The main problems found to deal with large-scale data are:
  - **Runtime:** The complexity of the traditional k-NN algorithm is $O((n \cdot D))$, where n is the number of instances and D the number of features.
  - **Memory consumption:** For a rapid computation of the distances, the k-NN model may normally require to store the training data in memory. When TR is too big, it could easily exceed the available RAM memory.

Applying KNN on Big Data

- These drawbacks motivate the use of Big Data techniques to distribute the processing of KNN over a cluster a nodes.
- A MapReduce-based approach for k-Nearest neighbor classification can be applied.
- This allows us to simultaneously classify large amounts of unseen cases (test examples) against a big (training) dataset.

Applying KNN on Big Data

- First, the training data will be divided into multiple splits.
- The **map phase** will determine the k-nearest neighbors in the different splits of the data.
- As a result of each map, the k nearest neighbors together with their computed distance values will be emitted to the reduce phase.

Applying KNN on Big Data

- Afterwards, the **reduce phase** will compute the definitive neighbors from the list obtained in the map phase.
- The reduce phase will determine which are the final k nearest neighbors from the list provided by the maps.
- This parallel implementation provides the exact classification rate as the original k-NN model.

Decision Trees

Decision Trees

- A **decision tree** (also called **prediction tree**) uses a tree structure to specify sequences of decisions and consequences.
- Given input $X = \{ x1, x2,...,xn \}$, the goal is to predict a response or output variable Y.
- The prediction can be achieved by constructing a decision tree with test points (nodes) and branches. At each test point, a decision is made to pick a specific branch and traverse down the tree.
- Eventually, a final point is reached, and a prediction can be made.

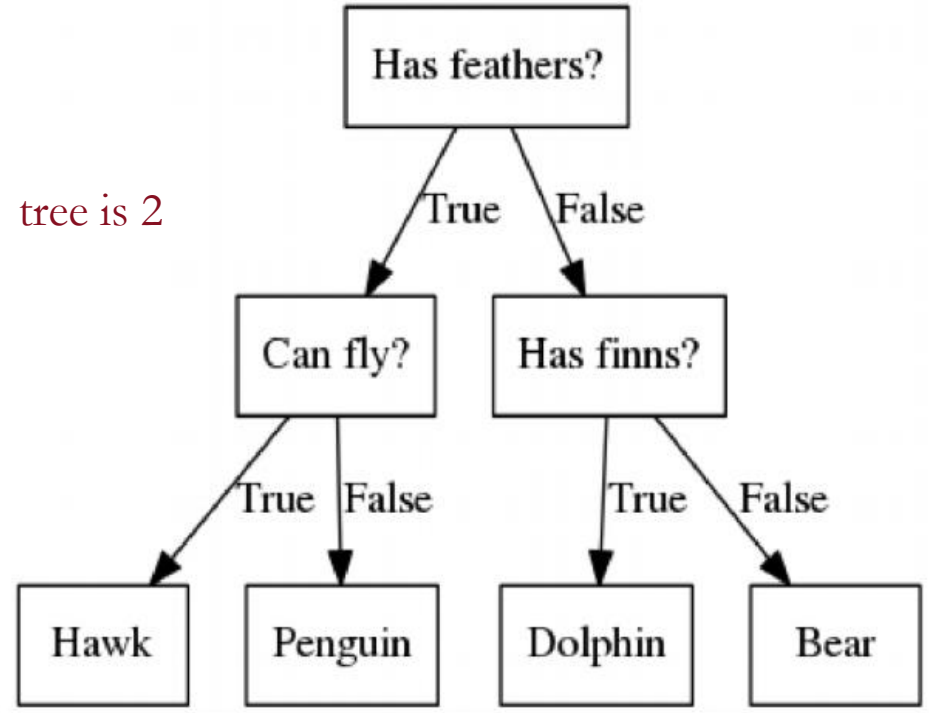
Decision Trees

- Each test point in a decision tree involves testing a particular input variable (or attribute), and each branch represents the decision being made.
- Due to its flexibility and easy visualization, decision trees are commonly deployed in data mining applications for classification purposes.
- The input values of a decision tree can be categorical or continuous.
- The leaf nodes return class labels and, in some implementations, they return the probability scores.
- A decision tree can be converted into a set of decision rules.

Decision Trees

- The following figure shows an example of using a decision tree to predict whether customers will buy a product.

Height of the tree is 2



---- **Root** – Top internal node

----- **Branch** – Outcome of test

----- **Internal Node** – Decision on variable

<---- **Leaf Node** – Class label

Decision Trees

- The term ***branch*** refers to the outcome of a decision and is visualized as a line connecting two nodes.
- If a decision is **numerical**, the "greater than" branch is usually placed on the right, and the "less than" branch is placed on the left.
- ***Internal nodes*** are the decision or test points.
- Each internal node refers to an input variable or an attribute. The top internal node is called the ***root***.
- The decision tree in the figure is a binary tree in that each internal node has no more than two branches. The branching of a node is referred to as a ***split***.

Decision Trees

- Sometimes decision trees may **have more than two branches** stemming from a node. For example, if an input variable *Weather* is categorical and has three choices—Sunny, Rainy, and Snow.
- The ***depth*** of a node is the minimum number of steps (edges) required to reach the node from the root. In the figure, for example, nodes *Income* and *Age* have a depth of one, and the four nodes on the bottom of the tree have a depth of two.
- ***Leaf nodes*** are at the end of the last branches on the tree. They represent class labels—the outcome of all the prior decisions.

Decision Trees: The General Algorithm

- The first step in constructing a decision tree is to choose **the most informative attribute**.
- A common way to identify the most informative attribute is to use **Entropy** and **Information Gain** methods.
- After the first attribute test splits up the examples, each outcome is a new decision tree learning problem in itself, with fewer examples and one less attribute.

Decision Trees: The General Algorithm

- There are four cases to consider for these recursive problems:

1. If the remaining examples are **all positive** (or **all negative**), then we are done.
2. If there are **some positive and some negative** examples, then choose the best attribute to split them.
3. If there are **no examples left**, it means that no example has been observed for this combination of attribute values, and we return a default value calculated from the plurality classification (most frequent class) of all the examples that were used in constructing the node's parent.

Decision Trees: The General Algorithm

- There are four cases to consider for these recursive problems (cont.):

4. If there are no attributes left, but both positive and negative examples, it means that these examples have exactly the same description, but different classifications. This can happen because there is an **error** or **noise** in the data; The best we can do is return the plurality classification of the remaining examples.

Decision Trees: The General Algorithm

- **Example:**

Example	Input Attributes										Goal
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*WillWait*
$\mathbf{x}_1$	*Yes*	*No*	*No*	*Yes*	*Some*	*$$$*	*No*	*Yes*	*French*	*0–10*	$y_1 = Yes$
$\mathbf{x}_2$	*Yes*	*No*	*No*	*Yes*	*Full*	*$*	*No*	*No*	*Thai*	*30–60*	$y_2 = No$
$\mathbf{x}_3$	*No*	*Yes*	*No*	*No*	*Some*	*$*	*No*	*No*	*Burger*	*0–10*	$y_3 = Yes$
$\mathbf{x}_4$	*Yes*	*No*	*Yes*	*Yes*	*Full*	*$*	*Yes*	*No*	*Thai*	*10–30*	$y_4 = Yes$
$\mathbf{x}_5$	*Yes*	*No*	*Yes*	*No*	*Full*	*$$$*	*No*	*Yes*	*French*	*>60*	$y_5 = No$
$\mathbf{x}_6$	*No*	*Yes*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Italian*	*0–10*	$y_6 = Yes$
$\mathbf{x}_7$	*No*	*Yes*	*No*	*No*	*None*	*$*	*Yes*	*No*	*Burger*	*0–10*	$y_7 = No$
$\mathbf{x}_8$	*No*	*No*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Thai*	*0–10*	$y_8 = Yes$
$\mathbf{x}_9$	*No*	*Yes*	*Yes*	*No*	*Full*	*$*	*Yes*	*No*	*Burger*	*>60*	$y_9 = No$
$\mathbf{x}_{10}$	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$$$*	*No*	*Yes*	*Italian*	*10–30*	$y_{10} = No$
$\mathbf{x}_{11}$	*No*	*No*	*No*	*No*	*None*	*$*	*No*	*No*	*Thai*	*0–10*	$y_{11} = No$
$\mathbf{x}_{12}$	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$*	*No*	*No*	*Burger*	*30–60*	$y_{12} = Yes$

Decision Trees: The General Algorithm

- Given a class X and its label $x \in X$, let $P(x)$ be the probability of x. Hx, the entropy of X, is defined as

$$H_X = -\sum_{\forall x \in X} P(x) \log_2 P(x)$$

- The first step is to compute the base entropy which is the entropy of the output variable (i.e Goal), in our example the Goal attribute is WillWait:

H(Goal)=H(WillWait)=-{6/12 log2(6/12) + 6/12 log2(6/12)}=1

Decision Trees: The General Algorithm

- The next step is to identify the conditional entropy for each attribute.
- Given an attribute X, its value x, its outcome Y, and its value y, conditional entropy $H_{Y|X}$ is the remaining entropy of Y given X, formally defined as:

$$H_{Y|X} = \sum_{x} P(x) H(Y|X = x)$$

$$= -\sum_{\forall x \in X} P(x) \sum_{\forall y \in Y} P(y|x) \log_2 P(y|x)$$

Decision Trees: The General Algorithm

- For attribute Patrons X={Some, Full, None} and Y(i.e Goal)={Yes, No}
- **$H_{Y|X} = H_{WillWait|Patrons} = P(Some)*H(Y|Some) + P(Full)*H(Y|Full) + P(None)*H(Y|None)$**
- Computations:
  - **P(Some)=4/12**
  - **H(Y|Some)=-4/4 log2(4/4) (from the 4 having x=some how many have goal=Yes and how many have goal=No)**
  - **P(Full)=6/12; H(Y|Full)=-(4/6 log2(4/6) + 2/6 log2(2/6))**
  - **P(None)=2/12; H(Y|None)=-2/2 log2(2/2)**
- Therefore, **$H_{Y|X} = H_{WillWait|Patrons} = 0.459$**

Decision Trees: The General Algorithm

- Then compute the Information Gain of attribute *Patrons*:

$$\textbf{InformationGain(Patrons)} = H(\text{WillWait}) - H_{\text{WillWait}|\text{Patron}}$$

- After computing the Information Gain of each attribute, Choose the attribute having the highest value.
- Repeat the process for each subtree generated.
- The algorithm indicated is called ID3.

Decision Trees: ID3 Algorithm

- Let A be a set of categorical input variables, P be the output variable (or the predicted class), and T be the training set. The ID3 algorithm is shown here.

```
1   ID3 (A, P, T)
2    if T ∈ φ
3      return φ
4    if all records in T have the same value for P
5      return a single node with that value
6    if A ∈ φ
7      return a single node with the most frequent value of P in T
8    Compute information gain for each attribute in A relative to T
9    Pick attribute D with the largest gain
10   Let { d1,d2...dm } be the values of attribute D
11   Partition T into { T1,T2...Tm } according to the values of D
12   return a tree with root D and branches labeled d1,d2...dm
           going respectively to trees ID3(A-{D}, P, T1),
           ID3(A-{D}, P, T2), . . . ID3(A-{D}, P, Tm)
```

Applying Decision Trees on Big Data

Applying Decision Trees on Big Data

- Classification tree learning on massive datasets is a common data mining task at Google.
- Google Inc. published an approach called **PLANET** (**P**arallel **L**earner for **A**ssembling **N**umerous **E**nsemble Trees.).
- PLANET is a scalable distributed framework for learning tree models over large datasets.
- PLANET defines tree learning as a series of distributed computations, and implements each one using the **MapReduce** model.

Naïve Bayes classifier

Naïve Bayes

- Naïve Bayes is a probabilistic classification method based on Bayes' theorem (or Bayes' law).
- Bayes' theorem gives the **relationship between the probabilities of two events and their conditional probabilities**.
- A naïve Bayes classifier assumes that the presence (or absence) of a particular feature of a class is unrelated to the presence (or absence) of other features.

Naïve Bayes

- The input variables are generally **categorical**, but variations of the algorithm can accept continuous variables.
- There are also ways to convert continuous variables into categorical ones. This process is often referred to as the ***discretization of continuous variables***.
- The output typically includes a class label and its corresponding probability score.

Naïve Bayes: Bayes' Theorem

- The ***conditional probability*** of event C occurring, given that event A has already occurred, is denoted as $P(C|A)$, which can be found using the following equation:

$$P(C|A) = \frac{P(A \cap C)}{P(A)}$$

$$P(C|A) = \frac{P(A|C) \cdot P(C)}{P(A)}$$

where C is the class label C $\in$ {c1, c2,…cn } and A is the observed attributes A = {a1, a2, …am}

Naïve Bayes: Bayes' Theorem

- An example to illustrate the use of Bayes' theorem: compute the probability that a patient carries a disease based on the result of a lab test:

Assume that a patient named M took a lab test for a certain disease and the result came back positive. The test returns a positive result in 95% of the cases in which the disease is actually present, and it returns a positive result in 6% of the cases in which the disease is not present. Furthermore, 1% of the entire population has this disease. What is the probability that M actually has the disease, given that the test is positive?

Naïve Bayes: Bayes' Theorem

- Let C = {having the disease} and A = {testing positive}.
- The goal is to solve the probability of having the disease, given that M has a positive test result, $P(C|A)$.
- From the problem description, $P(C)=0.01$, $P(\neg C)=0.99$, $P(A|C)=0.95$ and $P(A|\neg C)=0.06$.
- First, we need to compute $P(A)$:

$$\begin{aligned} P(A) &= P(A \cap C) + P(A \cap \neg C) \\ &= P(C) \cdot P(A|C) + P(\neg C) \cdot P(A|\neg C) \\ &= 0.01 \times 0.95 + 0.99 \times 0.06 = 0.0689 \end{aligned}$$

Naïve Bayes: Bayes' Theorem

- Then $P(C|A)$ can be computed as follows:

$$P(C|A) = \frac{P(A|C)P(C)}{P(A)} = \frac{0.95 \times 0.01}{0.0689} \approx 0.1379$$

- That means that the probability of M actually having the disease given a positive test result is only 13.79%.

Naïve Bayes: Bayes' Theorem

- A more general form of Bayes' theorem assigns a classified label to an object with multiple attributes $A = \{a1, a2, \ldots am\}$ such that the label corresponds to the largest value of $P(ci|A)$.

$$P(c_i|A) = \frac{P(a_1, a_2, \ldots, a_m|c_i) \cdot P(c_i)}{P(a_1, a_2, \ldots, a_m)}, i = 1, 2, \ldots n$$

Naïve Bayes classifier

- With two simplifications, Bayes' theorem can be extended to become a naïve Bayes classifier.
- **The first simplification** is to use the conditional independence assumption. That is, each attribute is conditionally independent of every other attribute given a class label c*i*.
- **The second simplification** is to ignore the denominator *P* (a1, a2,..am) because it appears in the denominator for all values of *i*, removing the denominator will have no impact on the relative probability scores and will simplify calculations.

Naïve Bayes classifier

- Naïve Bayes classification applies the two simplifications mentioned earlier and, as a result, P (ci | a1, a2,..am) is proportional to the product of P (aj | ci) times P (ci).

$$P(c_i|A) \propto P(c_i) \cdot \prod_{J=1}^{m} P(a_j|c_i) \qquad i = 1,2,\ldots n$$

Naïve Bayes classifier

- Building a naïve Bayes classifier requires knowing certain statistics, all calculated from the training set.
- The first requirement is to collect the probabilities of all class labels, $P(c_i)$.
- The second thing the naïve Bayes classifier needs to know is the conditional probabilities of each attribute a_j given each class label c_i, namely $P(a_j \mid c_i)$. For each attribute and its possible values, computing the conditional probabilities given each class label is required.

Naïve Bayes classifier

- For a given attribute assume it can have the following values {x,y,z} and assume that we have two class labels {c1 and c2}.
- Then the following probabilities need to be computed:
  - P(x | c1)
  - P(x | c2)
  - P(y | c1)
  - P(y | c2)
  - P(z | c1)
  - P(z | c2)

Naïve Bayes classifier

- After that, the naïve Bayes classifier can be tested over the testing set.
- For each record in the testing set, the naïve Bayes classifier assigns the classifier label c*i* that maximizes:

$$P(c_i) \cdot \prod_{j=1}^{m} P(a_j|c_i)$$

where:

- i is the number of class labels
- j is the number of features (dimensions)
- a1 is the value of the first feature in the test record
- a2 is the value of the second feature in the test record…….

Applying Naïve Bayes classifier on Big Data

Applying Naïve Bayes classifier on Big Data

- Applying MapReduce with Naïve Bayes classifier significantly decreases computation times allowing its application on Big Data problems.
- First, the training data will be divided into multiple splits.
- During the **map phase**, each map processes a single split, and computes statistics of the input data.
- For each attribute, the map outputs a <Key, Value> pair, where
  - Key is the class label,
  - Value is AttributeValue → the frequency of attribute value within that class label

Applying Naïve Bayes classifier on Big Data

- In the **reduce phase**, the reduce function aggregates the number of each attribute value within each class label.
- For each attribute, the reduce function outputs a <Key, Value> pair, where:
  - Key is the class label,
  - Value is AttributeValue → Σi the frequency of attribute value within that class label,

    where i is the number of mappers.

Performance Evaluation of classifiers

Performance Evaluation of classifiers

- A ***confusion matrix*** is a specific table layout that allows visualization of the performance of a classifier.
- The following figure shows the confusion matrix for a two-class classifier:

		Predicted Class	
		Positive	Negative
Actual Class	Positive	True Positives (TP)	False Negatives (FN)
	Negative	False Positives (FP)	True Negatives (TN)

Performance Evaluation of classifiers

- ***True positives*** (TP) are the number of positive instances the classifier correctly identified as positive.
- ***False positives*** (FP) are the number of instances in which the classifier identified as positive but in reality are negative.
- ***True negatives*** (TN) are the number of negative instances the classifier correctly identified as negative.
- ***False negatives*** (FN) are the number of instances classified as negative but in reality are positive.

Performance Evaluation of classifiers

- The ***accuracy*** (or the ***overall success rate***) is a metric defining the rate at which a model has classified the records correctly.
- It is defined as the sum of TP and TN divided by the total number of instances, as shown in the following equation:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \times 100\%$$

Performance Evaluation of classifiers

- The ***false positive rate*** (FPR) shows what percent of negatives the classifier marked as positive.
- The FPR is also called the ***false alarm rate*** or the ***type I error rate .***
- FPR is computed as follows:

$$FPR = \frac{FP}{FP + TN}$$

Performance Evaluation of classifiers

- ***Precision*** is the percentage of instances marked positive that really are positive. It is computed as follows:

$$Precision = \frac{TP}{TP + FP}$$

- ***Recall*** is the percentage of positive instances that were correctly identified. It is also called ***true positive rate*** (TPR). It is computed as follows:

$$TPR\ (or\ Recall) = \frac{TP}{TP + FN}$$

Performance Evaluation of classifiers

- ***f1-score*** is the harmonic mean of the precision and recall:

$$F_1 = 2 \cdot \frac{\mathrm{precision} \cdot \mathrm{recall}}{\mathrm{precision} + \mathrm{recall}}$$

Thank You