How To Implement Naive Bayes From Scratch in Python

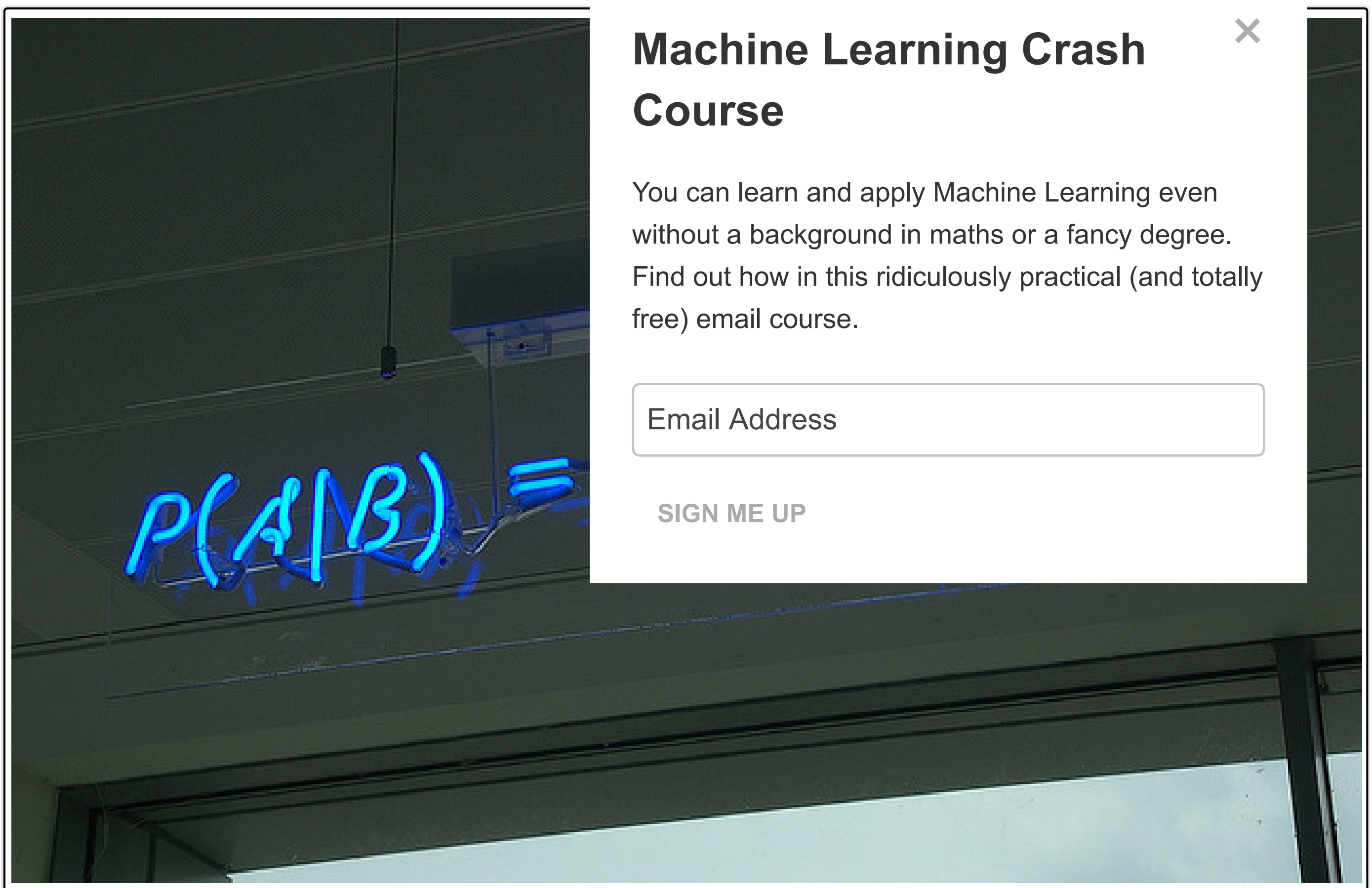
by **Jason Brownlee** on December 8, 2014 in **Uncategorized**

87 29 87

The Naive Bayes algorithm is simple and effective and should be one of the first methods you try on a classification problem.

In this tutorial you are going to learn about the Naive Bayes algorithm including how it works and how to implement it from scratch in Python.

Update: Check out the follow-up on tips for using the naive bayes algorithm titled: "Better Naive Bayes: 12 Tips To Get The Most From The Naive Bayes Algorithm"



Naive Bayes Classifier
Photo by Matt Buck, some rights reserved

About Naive Bayes

The Naive Bayes algorithm is an intuitive method that uses the probabilities of each attribute belonging to each class to make a prediction. It is the supervised learning approach you would come up with if you wanted to model a predictive modeling problem probabilistically.

Naive bayes simplifies the calculation of probabilities by assuming that the probability of each attribute belonging to a given class value is independent of all other attributes. This is a strong assumption but results in a fast and effective method.

The probability of a class value given a value of an attribute is called the conditional probability. By multiplying the conditional probabilities together for each attribute for a given class value, we have a probability of a data instance belonging to that class.

To make a prediction we can calculate probabilities of the instance belonging to each class and select the class value with the highest probability.

Naive bases is often described using categorical data because it is easy to describe and calculate using ratios. A more useful version of the algorithm for our purposes supports numeric attributes and assumes the values of each numerical attribute are normally distributed (fall somewhere on a bell curve). Again, this is a strong assumption, but still gives robust results.

Predict the Onset of Diabetes

The test problem we will use in this tutorial is the Pima Indians Diabetes problem.

This problem is comprised of 768 observations of medical details for Pima indians patents. The records describe instantaneous measurements taken from the patient such as their age, the number of times pregnant and blood workup. All patients are women aged 21 or older. All attributes are numeric, and their units vary from attribute to attribute.

Each record has a class value that indicates whether the patient suffered an onset of diabetes within 5 years of when the measurements were taken (1) or not (0).

This is a standard dataset that has been studied a lot in machine learning literature. A good prediction accuracy is 70%-76%.

Below is a sample from the pima-indians.data.csv file to get a sense of the data we will be working with.

NOTE: Download this file and save it with a .csv extension (e.g. **pima-indians-diabetes.data.csv**). See this file for a description of all the attributes.

```
6,148,72,35,0,33.6,0.627,50,1
1,85,66,29,0,26.6,0.351,31,0
8,183,64,0,0,23.3,0.672,32,1
1,89,66,23,94,28.1,0.167,21,0
0,137,40,35,168,43.1,2.288,33,1
```

Naive Bayes Algorithm Tutorial

This tutorial is broken down into the following steps:

1. **Handle Data**: Load the data from CSV file and split it into training and test datasets.
2. **Summarize Data**: summarize the properties in the training dataset so that we can calculate probabilities and make predictions.
3. **Make a Prediction**: Use the summaries of the dataset to generate a single prediction.
4. **Make Predictions**: Generate predictions given a test dataset and a summarized training dataset.
5. **Evaluate Accuracy**: Evaluate the accuracy of predictions made for a test dataset as the percentage correct out of all predictions made.
6. **Tie it Together**: Use all of the code elements to present a complete and standalone implementation of the Naive Bayes algorithm.

1. Handle Data

The first thing we need to do is load our data file. The data is in CSV format without a header line or any quotes. We can open the file with the open function and read the data lines using the reader function in the csv module.

We also need to convert the attributes that were loaded as strings into numbers that we can work with them. Below is the **loadCsv()** function for loading the Pima indians dataset.

```
import csv
def loadCsv(filename):
    lines = csv.reader(open(filename, "rb"))
    dataset = list(lines)
    for   in range(len(dataset)):
        dataset[i] = [float(x) for x in dataset[i]]
    return dataset
```

We can test this function by loading the pima indians dataset and printing the number of data instances that were loaded.

```
filename = 'pima-indians-diabetes.data.csv'
dataset = loadCsv(filename)
print('Loaded data file {0} with {1} rows').format(filename, len(dataset))
```

Running this test, you should see something like:

```
Loaded data file pima-indians-diabetes.data.csv rows
```

Next we need to split the data into a training dataset that Naive Bayes can use to make predictions and a test dataset that we can use to evaluate the accuracy of the model. We need to split the data set randomly into train and datasets with a ratio of 67% train and 33% test (this is a common ratio for testing an algorithm on a dataset).

Below is the **splitDataset()** function that will split a given dataset into a given split ratio.

```
import random
def splitDataset(dataset, splitRatio):
    trainSize = int(len(dataset) * splitRatio)
    trainSet = []
    copy = list(dataset)
    while len(trainSet) < trainSize:
        index = random.randrange(len(copy))
        trainSet.append(copy.pop(index))
    return [trainSet, copy]
```

We can test this out by defining a mock dataset with 5 instances, split it into training and testing datasets and print them out to see which data instances ended up where.

```
dataset = [[1], [2], [3], [4], [5]]
splitRatio = 0.67
train, test = splitDataset(dataset, splitRatio)
print('Split {0} rows into train with {1} and test with {2}').format(len(dataset), tra
```

Running this test, you should see something like:

```
      5 rows into train       [[4], [3], [5]] and test       [[1], [2]]
```

2. Summarize Data

The naive bayes model is comprised of a summary of the data in the training dataset. This summary is then used when making predictions.

The summary of the training data collected involves the mean and the standard deviation for each attribute, by class value. For example, if there are two class values and 7 numerical attributes, then we need a mean and standard deviation for each attribute (7) and class value (2) combination, that is 14 attribute summaries.

These are required when making predictions to calculate the probability of specific attribute values belonging to each class value.

We can break the preparation of this summary data down into the following sub-tasks:

1. Separate Data By Class
2. Calculate Mean
3. Calculate Standard Deviation
4. Summarize Dataset
5. Summarize Attributes By Class

Separate Data By Class

The first task is to separate the training dataset instances by class value so that we can calculate statistics for each class. We can do that by creating a map of each class value to a list of instances that belong to that class and sort the entire dataset of instances into the appropriate lists.

The **separateByClass()** function below does just this.

```
def separateByClass(dataset):
    separated = {}
    for i in range(len(dataset)):
        vector = dataset[i]
        if (vector[-1] not in separated):
            separated[vector[-1]] = []
        separated[vector[-1]].append(vector)
    return separated
```

You can see that the function assumes that the last attribute (-1) is the class value. The function returns a map of class values to lists of data instances.

We can test this function with some sample data, as follows:

```
dataset = [[1,20,1], [2,21,0], [3,22,1]]
separated = separateByClass(dataset)
print('Separated instances: {0}').format(separated)
```

Running this test, you should see something like:

```
Separated instances: {0: [[2, 21, 0]], 1: [[1, 20, 1], [3, 22, 1]]}
```

Calculate Mean

We need to calculate the mean of each attribute for a class value. The mean is the central middle or central tendency of the data, and we will use it as the middle of our gaussian distribution when calculating probabilities.

We also need to calculate the standard deviation of each attribute for a class value. The standard deviation describes the variation of spread of the data, and we will use it to characterize the expected spread of each attribute in our Gaussian distribution when calculating probabilities.

The standard deviation is calculated as the square root of the variance. The variance is calculated as the average of the squared differences for each attribute value from the mean. Note we are using the N-1 method, which subtracts 1 from the number of attribute values when calculating the variance.

```
import math
def mean(numbers):
    return sum(numbers)/float(len(numbers))

def stdev(numbers):
    avg = mean(numbers)
    variance = sum([pow(x-avg,2) for x in numbers])/float(len(numbers)-1)
    return math.sqrt(variance)
```

We can test this by taking the mean of the numbers from 1 to 5.

```
numbers = [1,2,3,4,5]
print('Summary of {0}: mean={1}, stdev={2}').format(numbers, mean(numbers), stdev(numb
```

Running this test, you should see something like:

```
Summary of [1, 2, 3, 4, 5]: mean=3.0, stdev=1.58113883008
```

Summarize Dataset

Now we have the tools to summarize a dataset. For a given list of instances (for a class value) we can calculate the mean and the standard deviation for each attribute.

The zip function groups the values for each attribute across our data instances into their own lists so that we can compute the mean and standard deviation values for the attribute.

```
def summarize(dataset):
    summaries = [(mean(attribute), stdev(attribute)) for attribute in zip(*dataset)]
    del summaries[-1]
    return summaries
```

We can test this **summarize()** function with some test data that shows markedly different mean and standard deviation values for the first and second data attributes.

```
dataset = [[1,20,0], [2,21,1], [3,22,0]]
summary = summarize(dataset)
print('Attribute summaries: {0}').format(summary)
```

Running this test, you should see something like:

```
Attribute summaries: [(2.0, 1.0), (21.0, 1.0)]
```

Summarize Attributes By Class

We can pull it all together by first separating our training dataset into instances grouped by class.

Then calculate the summaries for each attribute.

```
def summarizeByClass(dataset):
    separated = separateByClass(dataset)
    summaries = {}
    for classValue, instances in separated.iteritems():
        summaries[classValue] = summarize(instances)
    return summaries
```

We can test this **summarizeByClass()** function with a small test dataset.

```
dataset = [[1,20,1], [2,21,0], [3,22,1], [4,22,0]]
summary = summarizeByClass(dataset)
print('Summary by class value: {0}').format(summary)
```

Running this test, you should see something like:

```
Summary by class value:
{0: [(3.0, 1.4142135623730951), (21.5, 0.7071067811865476)],
1: [(2.0, 1.4142135623730951), (21.0, 1.4142135623730951)]}
```

3. Make Prediction

We are now ready to make predictions using the summaries prepared from our training data. Making predictions involves calculating the probability that a given data instance belongs to each class, then selecting the class with the largest probability as the prediction.

We can divide this part into the following tasks:

1. Calculate Gaussian Probability Density Function
2. Calculate Class Probabilities
3. Make a Prediction
4. Estimate Accuracy

Calculate Gaussian Probability Density Function

We can use a Gaussian function to estimate the probability of a given attribute value, given the known mean and standard deviation for the attribute estimated from the training data.

Given that the attribute summaries where prepared for each attribute and class value, the result is the conditional probability of a given attribute value given a class value.

See the references for the details of this equation for the Gaussian probability density function. In summary we are plugging our known details into the Gaussian (attribute value, mean and standard deviation) and reading off the likelihood that our attribute value belongs to the class.

In the **calculateProbability()** function we calculate the exponent first, then calculate the main division. This lets us fit the equation nicely on two lines.

```
import math
def calculateProbability(x, mean, stdev):
    exponent = math.exp(-(math.pow(x-mean,2)/(2*math.pow(stdev,2))))
    return (1 / (math.sqrt(2*math.pi) * stdev)) * exponent
```

We can test this with some sample data, as follows.

```
x = 71.5
mean = 73
stdev = 6.2
probability = calculateProbability(x, mean, stdev)
print('Probability of belonging to this class: {0}').format(probability)
```

Running this test, you should see something like:

```
Probability of belonging to this class: 0.0624896575937
```

Calculate Class Probabilities

Now that we can calculate the probability of an attribute belonging to a class, we can combine the probabilities of all of the attribute values for a data instance and come up with a probability of the entire data instance belonging to the class.

We combine probabilities together by multiplying them. In the **calculateClassProbabilities()** below, the probability of a given data instance is calculated by multiplying together the attribute probabilities for each class. the result is a map of class values to probabilities.

```
def calculateClassProbabilities(summaries, inputVector):
    probabilities = {}
    for classValue, classSummaries in summaries.iteritems():
        probabilities[classValue] = 1
        for i in range(len(classSummaries)):
            mean, stdev = classSummaries[i]
            x = inputVector[i]
            probabilities[classValue] *= calculateProbability(x, mean, stdev)
    return probabilities
```

We can test the **calculateClassProbabilities()** function.

```
summaries = {0:[(1, 0.5)], 1:[(20, 5.0)]}
inputVector = [1.1, '?']
probabilities = calculateClassProbabilities(summaries, inputVector)
print('Probabilities for each class: {0}').format(probabilities)
```

Running this test, you should see something like:

```
Probabilities for each class: {0: 0.7820853879509118, 1: 6.298736258150442e-05}
```

Make a Prediction

Now that can calculate the probability of a data instance belonging to each class value, we can look for the largest probability and return the associated class.

The **predict()** function belong does just that.

```
def predict(summaries, inputVector):
    probabilities = calculateClassProbabilities(summaries, inputVector)
    bestLabel, bestProb = None, -1
    for classValue, probability in probabilities.iteritems():
        if bestLabel is None or probability > bestProb:
            bestProb = probability
            bestLabel = classValue
    return bestLabel
```

We can test the **predict()** function as follows:

```
summaries = {'A':[(1, 0.5)], 'B':[(20, 5.0)]}
inputVector = [1.1, '?']
result = predict(summaries, inputVector)
print('Prediction: {0}').format(result)
```

Running this test, you should see something like:

```
Prediction: A
```

4. Make Predictions

Finally, we can estimate the accuracy of the model by making predictions for each data instance in our test dataset. The **getPredictions()** will do this and return a list of predictions for each test instance.

```
def getPredictions(summaries, testSet):
    predictions = []
    for   in range(len(testSet)):
        result = predict(summaries, testSet[i])
        predictions.append(result)
    return predictions
```

We can test the **getPredictions()** function.

```
summaries = {'A':[(1, 0.5)], 'B':[(20, 5.0)]}
testSet = [[1.1, '?'], [19.1, '?']]
predictions = getPredictions(summaries, testSet)
print('Predictions: {0}').format(predictions)
```

Running this test, you should see something like:

```
Predictions: ['A', 'B']
```

5. Get Accuracy

The predictions can be compared to the class values in the test dataset and a classification accuracy can be calculated as an accuracy ratio between 0& and 100%. The **getAccuracy()** will calculate this accuracy ratio.

```
def getAccuracy(testSet, predictions):
    correct = 0
    for   in range(len(testSet)):
        if testSet[x][-1] == predictions[x]:
            correct += 1
    return (correct/float(len(testSet))) * 100.0
```

We can test the **getAccuracy()** function using the sample code below.

```
testSet = [[1,1,1,'a'], [2,2,2,'a'], [3,3,3,'b']]
predictions = ['a', 'a', 'a']
accuracy = getAccuracy(testSet, predictions)
print('Accuracy: {0}').format(accuracy)
```

Running this test, you should see something like:

```
Accuracy: 66.6666666667
```

6. Tie it Together

Finally, we need to tie it all together.

Below provides the full code listing for Naive Bayes implemented from scratch in Python.

```
# Example of Naive Bayes implemented from Scratch in Python
import csv
import random
import math

def loadCsv(filename):
    lines = csv.reader(open(filename, "rb"))
    dataset = list(lines)
    for i in range(len(dataset)):
        dataset[i] = [float(x) for x in dataset[i]]
    return dataset

def splitDataset(dataset, splitRatio):
    trainSize = int(len(dataset) * splitRatio)
    trainSet = []
    copy = list(dataset)
    while len(trainSet) < trainSize:
        index = random.randrange(len(copy))
        trainSet.append(copy.pop(index))
    return [trainSet, copy]

def separateByClass(dataset):
    separated = {}
    for i in range(len(dataset)):
        vector = dataset[i]
        if (vector[-1] not in separated):
            separated[vector[-1]] = []
        separated[vector[-1]].append(vector)
    return separated

def mean(numbers):
    return sum(numbers)/float(len(numbers))

def stdev(numbers):
    avg = mean(numbers)
    variance = sum([pow(x-avg,2) for x in numbers])/float(len(numbers)-1)
    return math.sqrt(variance)

def summarize(dataset):
    summaries = [(mean(attribute), stdev(attribute)) for attribute in zip(*dataset)]
    del summaries[-1]
    return summaries

def summarizeByClass(dataset):
    separated = separateByClass(dataset)
    summaries = {}
    for classValue, instances in separated.iteritems():
        summaries[classValue] = summarize(instances)
    return summaries

def calculateProbability(x, mean, stdev):
    exponent = math.exp(-(math.pow(x-mean,2)/(2*math.pow(stdev,2))))
```

```python
    return (1 / (math.sqrt(2*math.pi) * stdev)) * exponent

def calculateClassProbabilities(summaries, inputVector):
    probabilities = {}
    for classValue, classSummaries in summaries.iteritems():
        probabilities[classValue] = 1
        for i in range(len(classSummaries)):
            mean, stdev = classSummaries[i]
            x = inputVector[i]
            probabilities[classValue] *= calculateProbability(x, mean, stdev)
    return probabilities

def predict(summaries, inputVector):
    probabilities = calculateClassProbabilities(summaries, inputVector)
    bestLabel, bestProb = None, -1
    for classValue, probability in probabilities.iteritems():
        if bestLabel is None or probability > bestProb:
            bestProb = probability
            bestLabel = classValue
    return bestLabel

def getPredictions(summaries, testSet):
    predictions = []
    for i in range(len(testSet)):
        result = predict(summaries, testSet[i])
        predictions.append(result)
    return predictions

def getAccuracy(testSet, predictions):
    correct = 0
    for i in range(len(testSet)):
        if testSet[i][-1] == predictions[i]:
            correct += 1
    return (correct/float(len(testSet))) * 100.0

def main():
    filename = 'pima-indians-diabetes.data.csv'
    splitRatio = 0.67
    dataset = loadCsv(filename)
    trainingSet, testSet = splitDataset(dataset, splitRatio)
    print('Split {0} rows into train={1} and test={2} rows').format(len(dataset), le
    # prepare model
    summaries = summarizeByClass(trainingSet)
    # test model
    predictions = getPredictions(summaries, testSet)
    accuracy = getAccuracy(testSet, predictions)
    print('Accuracy: {0}%').format(accuracy)

main()
```

Running the example provides output like the following:

```
       768 rows into train=514 and test=254 rows
Accuracy: 76.3779527559%
```

Implementation Extensions

This section provides you with ideas for extensions that you could apply and investigate with the Python code you have implemented as part of this tutorial.

You have implemented your own version of Gaussian Naive Bayes in python from scratch.

You can extend the implementation further.

- **Calculate Class Probabilities**: Update the example to summarize the probabilities of a data instance belonging to each class as a ratio. This can be calculated as the probability of a data instance belonging to one class, divided by the sum of the probabilities of the data instance belonging to each class. For example an instance had the probability of 0.02 for class A and 0.001 for class B, the likelihood of the instance belonging to class A is (0.02/(0.02+0.001))*100 which is about 95.23%.
- **Log Probabilities**: The conditional probabilities for each class given an attribute value are small. When they are multiplied together they result in very small values, which can lead to floating point underflow (numbers too small to represent in Python). A common fix for this is to combine the log of the probabilities together. Research and implement this improvement.
- **Nominal Attributes**: Update the implementation to support nominal attributes. This is much similar and the summary information you can collect for each attribute is the ratio of category values for each class. Dive into the references for more information.
- **Different Density Function** (*bernoulli* or *multinomial*): We have looked at Gaussian Naive Bayes, but you can also look at other distributions. Implement a different distribution such as multinomial, bernoulli or kernel naive bayes that make different assumptions about the distribution of attribute values and/or their relationship with the class value.

Resources and Further Reading

This section will provide some resources that you can use to learn more about the Naive Bayes algorithm in terms of both theory of how and why it works and practical concerns for implementing it in code.

Problem

More resources for learning about the problem of predicting the onset of diabetes.

- Pima Indians Diabetes Data Set: This page provides access to the dataset files, describes the attributes and lists papers that use the dataset.
- Dataset File: The dataset file.
- Dataset Summary: Description of the dataset attributes.
- Diabetes Dataset Results: The accuracy of many standard algorithms on this dataset.

Code

This section links to open source implementations of Naive Bayes in popular machine learning libraries. Review these if you are considering implementing your own version of the method for operational use.

- Naive Bayes in Scikit-Learn: Implementation of naive bayes in the scikit-learn library.
- Naive Bayes documentation: Scikit-Learn documentation and sample code for Naive Bayes
- Simple Naive Bayes in Weka: Weka implementation of naive bayes

Books

You may have one or more books on applied machine learning. This section highlights the sections or chapters in common applied books on machine learning that refer to Naive Bayes.

- Applied Predictive Modeling, page 353
- Data Mining: Practical Machine Learning Tools and Techniques, page 94
- Machine Learning for Hackers, page 78
- An Introduction to Statistical Learning: with Applications in R, page 138
- Machine Learning: An Algorithmic Perspective, page 171
- Machine Learning in Action, page 61 (Chapter 4)
- Machine Learning, page 177 (chapter 6)

Next Step

Take action.

Follow the tutorial and implement Naive Bayes from scratch. Adapt the example to another problem. Follow the extensions and improve upon the implementation.

Leave a comment and share your experiences.

Update: Check out the follow-up on tips for using the naive bayes algorithm titled: "Better Naive Bayes: 12 Tips To Get The Most From The Naive Bayes Algorithm"