NAIVEBAYES CLASSIFIER

Rohith Reddy Mandala
Master's in Computer Science
Texas A & M University – Corpus Christi
Corpus Christi, United States of America
rmandala@islander.tamucc.edu

Gangadhar Attilli
Master's in Computer Science
Texas A & M University – Corpus Christi
Corpus Christi, United States of America
gattilli@islander.tamucc.edu

ABSTRACT:

Machine learning techniques can automatically learn features from a many number of data sets. Text classification using machine learning is used to organize documents or data in a predefined set of classes/groups. So once the data is trained using the machine learning algorithms, the trained model will be able to identify, predict and detect the data for categorizing it in classes/groups/topics. It is very useful in Web content management, Search engines email filtering, spam detection, intent detection, topic labeling, tagging, categorization of data and sentiment analysis, etc. Text classification is the base of many applications. This report proposed a method for text classification using naive Bayes algorithm. The text data set was divided into training data set and test data set. Performed some data preprocessing techniques on both train and test datasets. The model we developed was used to train on the training data set and evaluated on testing data set.

RESPONSIBILITIES:

- **Rohith Reddy Mandala:**
  - Implemented the code portions for Data Pre-Processing Techniques and the Nave Bayes Algorithm.
  - Wrote the REPORT
- **Gangadhar Attilli:**
  - Implemented the code portions for Training and Testing.
  - Wrote the Presentation

1.INTRODUCTION:

Naive Bayes has been one of the popular machine learning methods for many years. Its simplicity makes the framework attractive in various tasks and reasonable performances are obtained in the tasks although this learning is based on an unrealistic independence assumption. For this reason, there also have been many interesting works of investigating naive Bayes. Especially, shows that naive Bayes can perform surprisingly well in the classification tasks where the probability itself calculated by the naive Bayes is not important.

1.1 NAIVE BAYES ALGORITHM:

It is a classification technique based on Bayes' Theorem with an assumption of independence among predictors. In simple terms, a Naive Bayes classifier assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature.

For example, a fruit may be considered to be an apple if it is red, round, and about 3 inches in diameter. Even if these features depend on each other or upon the existence of the other features, all of these properties independently contribute to the probability that this fruit is an apple and that is why it is known as 'Naive'.

Naive Bayes model is easy to build and particularly useful for very large data sets. Along with simplicity, Naive Bayes is known to outperform even highly sophisticated classification methods.

Bayes theorem provides a way of calculating posterior probability P(c|x) from P(c), P(x) and P(x|c). Look at the equation below:

$$\mathbf{P(y|x) \propto P(y)P(x|y) = P(y) \prod_{i=1}^{L} P(x_i|y)}$$

Fig 1

Above in Fig 1,

- $P(y|x)$ is the posterior probability of *class* (y, *target*) given *predictor* (x, *attributes*).
- $P(y)$ is the prior probability of *class*.
- $P(x|y)$ is the likelihood which is the probability of *predictor* given *class*.

Naïve Bayes classifiers are highly scalable, requiring a number of parameters linear in the number of variables (features/predictors) in a learning problem. Maximum-likelihood training can be done by evaluating a closed-form expression, which takes linear time, rather than by expensive iterative approximation as used for many other types of classifiers.

Naive Bayes classifiers are linear classifiers that are known for being simple yet very efficient. In practice, the independence assumption is often violated, but naive Bayes classifiers still tend to perform very well under this unrealistic assumption. Especially for small sample sizes, naive Bayes classifiers can outperform the more powerful alternatives. Being relatively robust, easy to implement, fast, and accurate, naive Bayes classifiers are used in many different fields. Some examples include the diagnosis of diseases and making decisions about treatment processes, the classification of RNA sequences in taxonomic studies, and spam filtering in e-mail clients.

In the following sections, we will take a closer look at the probability model of the naive Bayes classifier in Python that allows us to classify the text data.

2. PROJECT DESCRIPTION:

The naive Bayes classifier can be built in a variety of ways. We can use sklearn's built-in routines, but we are not using this approach for the project. We'll create the classifier without utilizing any built-in functions. In the naïve bayes algorithm, we create classes and methods for each task to develop the classifier. However, for data preprocessing, we use built-in routines.

2.1 PROBLEM STATEMENT:

The two tasks in front of us , we need to preprocess the given training and testing dataset like for each of verbs, you might need to present it as a base form (for example: become ⇐become, became, becomes, becoming); for each of specific numbers, you might need to present it as < Number >; for each of nouns, you might need to present it as singular (for example: text ⇐text, texts; election ⇐ election, elections).

Another task is to build a Naive Bayes classifier, train it on a training dataset, and then test it on a testing dataset.

The classification has two phases, a learning phase, and the evaluation phase. In the learning phase, classifier trains its model on a given dataset and in the evaluation phase, it tests the classifier performance. Performance is evaluated on the basis of various parameters such as accuracy, error, precision, and recall.

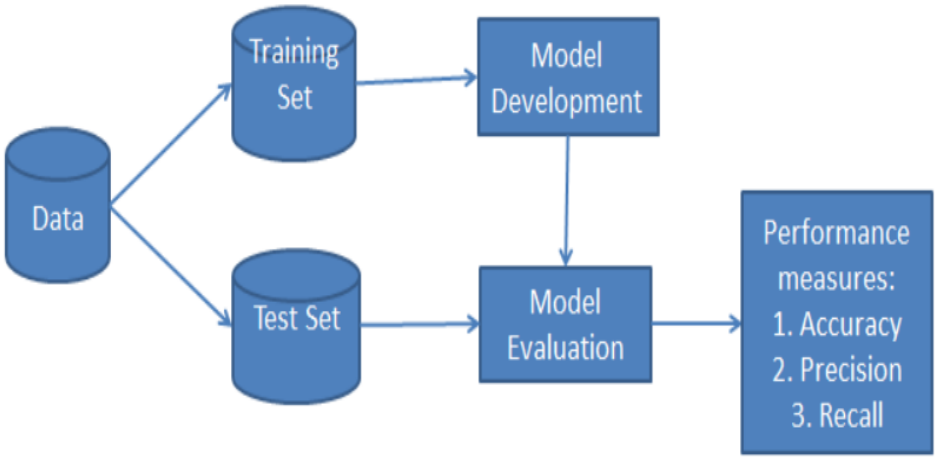

Fig 2

2.2 DATA PREPROCESSING:

The content preprocessing method basically includes word tokenization, replacing numbers with string, converting all letters to lowercase, removing punctuations and lemmatization etc.

2.2.1 TOKENIZATION:

Tokenization describes the general process of breaking down a text corpus into individual elements that serve as input for various natural language processing algorithms. Usually, tokenization is accompanied by other optional processing steps, such as the removal of stop words and punctuation characters, stemming or lemmatizing, and the construction of n-grams. Below Fig 3 is an example of a simple but typical tokenization step that splits a sentence into individual words.

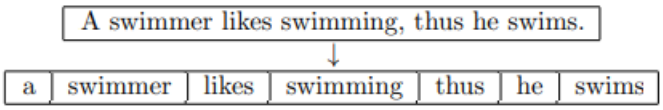

Fig 3: Example of Tokenization

2.2.2 LOWERCASE:

Lowercasing ALL your text data, although commonly overlooked, is one of the simplest and most effective form of text preprocessing. It is applicable to most text mining and NLP problems and can help in cases where your dataset is not very large and significantly helps with consistency of expected output.

2.2.3 REPLACING NUMBERS:

Sometimes it happens that words and digits combine are written in the text which creates a problem for machines to understand. Hence, we need to replace the words and digits which are combined like **game57** or **game5ts7**. This type of word is difficult to process so better to remove them or replace them with a string.

2.2.4 REMOVING PUNCTUATIONS:

One of the other text processing techniques is removing punctuations. There are total 32 main punctuations that need to be taken care of. we can directly use the string module with a regular expression to replace any punctuation in text with an empty string.

2.2.5 LEMMATIZATION:

Lemmatization is similar to stemming, used to stem the words into root word but differs in working. Actually, Lemmatization is a systematic way to reduce the words into their lemma by matching them with a language dictionary.

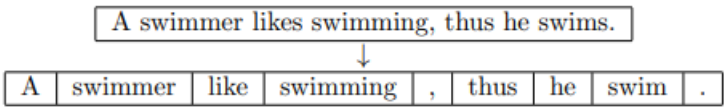

Fig 4: Example of Lemmatization

The Natural Language Toolkit, or more commonly NLTK, is a suite of libraries and programs for symbolic and statistical natural language processing for English written in the Python programming language. It consists of the most common algorithms such as tokenizing, part-of-speech tagging, stemming, sentiment analysis, topic segmentation, and named entity recognition, some of which we will be making use of in this article. All of the tasks listed above were completed using this library.

```
x['Verb'] = x['Verb'].apply(word_tokenize)
x['Noun'] = x['Noun'].apply(word_tokenize)
x['Prep'] = x['Prep'].apply(word_tokenize)
x['Prep_obj'] = x['Prep_obj'].apply(word_tokenize)
x.head()
```

```
x['Verb'] = x['Verb'].apply(lambda x: [word.lower() for word in x])
x['Noun'] = x['Noun'].apply(lambda x: [word.lower() for word in x])
x['Prep'] = x['Prep'].apply(lambda x: [word.lower() for word in x])
x['Prep_obj'] = x['Prep_obj'].apply(lambda x: [word.lower() for word in x])
```

```
wnl = WordNetLemmatizer()
x['Verb'] = x['Verb'].apply(lambda x: [wnl.lemmatize(word, tag) for word, tag in x])
x['Noun'] = x['Noun'].apply(lambda x: [wnl.lemmatize(word, tag) for word, tag in x])
#train['Prep'] = train['Prep'].apply(lambda x: [wnl.lemmatize(word, tag) for word, tag in x])
x['Prep_obj'] = x['Prep_obj'].apply(lambda x: [wnl.lemmatize(word, tag) for word, tag in x])
x.head()
```

```
spec_chars = ["!",'"',"#","%","&","'","(",")",
              "*","+",",","-",".","/",":",";","<",
              "=",">","?","@","[","\\","]","^","_",
              "`","{","|","}","~","–"]
for char in spec_chars:
    train['Verb'] = train['Verb'].str.replace(char, ' ')
    train['Noun'] = train['Noun'].str.replace(char, ' ')
    train['Prep'] = train['Prep'].str.replace(char, ' ')
    train['Prep_obj'] = train['Prep_obj'].str.replace(char, ' ')
```

Fig 5: Python code samples for Data Pre-Processing

2.2.6 NUMBER OF ATTRIBUTES AFTER DATA PRE-PROCESSING:

After data pre-processing the training set contain the number of attributes 783,1191,49 and 1309 in the respective Verb, Noun, Prep and Prep_obj columns.

2.3 IMPLEMENTATION OF NAÏVE BAYES ALGORITHM:

Naive Bayes classifier calculates the probability of an event in the following steps:

Step 1: Calculate the prior probability for given class labels.

Step 2: Find Evidence of Likelihood probability with each attribute for each class.

Step 3: Calculate Posterior Probability for each class using the Naive Bayesian equation.

2.3.1 PRIOR PROBABILITY FOR GIVEN CLASS LABELS:

In the context of pattern classification, the prior probabilities are also called class priors, which describe "the general probability of encountering a particular class." In the case of spam classification, the priors could be formulated as

P(spam) = "the probability that any new message is a spam message" and

P(ham) = 1 − P(spam).

If the priors are following a uniform distribution, the posterior probabilities will be entirely determined by the class-conditional probabilities and the evidence term. And since the evidence term is a constant, the decision rule will entirely depend on the class-conditional probabilities (similar to a frequentist's approach and maximum-likelihood estimate).

Eventually, the a priori knowledge can be obtained, e.g., by consulting a domain expert or by estimation from the training data (assuming that the training data is i.i.d. and a representative sample of the entire population. The maximum-likelihood estimate approach can be formulated as

$$\hat{P}(\omega j) = N\omega j / Nc$$

- Nωj: Count of samples from class ωj.
- Nc: Count of all samples.

And in context of spam classification:

P^(spam) = # of spam messages in training data / # of all messages in training data

```
    def _calc_class_prior(self):

        """ P(c) - Prior Class Probability """

        for outcome in np.unique(self.y_train):
            outcome_count = sum(self.y_train == outcome)
            self.class_priors[outcome] = outcome_count / self.train_size
```

Fig 6: Python code sample for calculating class probabilities

2.3.2 EVIDENCE OF LIKELIHOOD PROBABILITY WITH EACH ATTRIBUTE FOR EACH CLASS:

An additional assumption of naive Bayes classifiers is the conditional independence of features. Under this naive assumption, the class-conditional probabilities or (likelihoods) of the samples can be directly estimated from the training data instead of evaluating all possibilities of x. Thus, given a d-dimensional feature vector x, the class conditional probability can be calculated as follow:

$$P(x \mid \omega j) = P(x1 \mid \omega j) \cdot P(x2 \mid \omega j) \cdot \ldots \cdot P(xd \mid \omega j) = Y\ d\ k=1\ P(xk \mid \omega j)$$

Here, P (x | ωj) simply means: "How likely is it to observe this particular pattern x given that it belongs to class ωj?" The "individual" likelihoods for every feature in the feature vector can be estimated via the maximum-likelihood estimate, which is simply a frequency in the case of categorical data:

$$P\hat{}(xi \mid \omega j) = Nxi,\omega j / N\omega j\ (i = (1, ..., d))$$

- Nxi,ωj: Number of times feature xi appears in samples from class ωj.
- Nωj: Total count of all features in class ωj.

```
    def _calc_likelihoods(self):

        """ P(x|c) - Likelihood """

        for feature in self.features:

            for outcome in np.unique(self.y_train):
                outcome_count = sum(self.y_train == outcome)
                feat_likelihood = self.X_train[feature][self.y_train[self.y_train == outcome].index.values.tolist()].value_co

                for feat_val, count in feat_likelihood.items():
                    self.likelihoods[feature][feat_val + '_' + outcome] = count/outcome_count


    def _calc_predictor_prior(self):

        """ P(x) - Evidence """

        for feature in self.features:
            feat_vals = self.X_train[feature].value_counts().to_dict()

            for feat_val, count in feat_vals.items():
                self.pred_priors[feature][feat_val] = count/self.train_size
```

Fig 7: Python code sample for calculating likelihood probabilities

2.3.3 POSTERIOR PROBABILITY FOR EACH CLASS USING THE NAIVE BAYESIAN EQUATION:

Bayes' theorem forms the core of the whole concept of naive Bayes classification. The posterior probability, in the context of a classification problem, can be interpreted as: "What is the probability that a particular object belongs to class i given its observed feature values?"

The general notation of the posterior probability can be written as:

$$P(\omega j \mid xi) = P(xi \mid \omega j) \cdot P(\omega j) \; / \; P(xi)$$

The objective function in the naive Bayes probability is to maximize the posterior probability given the training data in order to formulate the decision rule.

$$\text{predicted class label} \leftarrow \arg\max_{j=1\ldots,m} P(\omega j \mid xi)$$

```
    def predict(self, X):

        """ Calculates Posterior probability P(c|x) """

        results = []
        X = np.array(X)

        for query in X:
            probs_outcome = {}
            for outcome in np.unique(self.y_train):
                prior = self.class_priors[outcome]
                likelihood = 1
                evidence = 1

                for feat, feat_val in zip(self.features, query):
                    likelihood *= self.likelihoods[feat][feat_val + '_' + outcome]
                    evidence *= self.pred_priors[feat][feat_val]

                posterior = (likelihood * prior) / (evidence)

                probs_outcome[outcome] = posterior

            result = max(probs_outcome, key = lambda x: probs_outcome[x])
            results.append(result)

        return np.array(results)

```

Fig 8: Python code sample for calculating posterior probability

2.3.4 TRAINING MODEL:

We have two datasets, one for training and one for testing. To train the model, we use the training dataset. We divide the dependent and independent variables and assign them to the model's fit technique. We must preserve the model after training. The machine learning model can be saved in a variety of ways, including utilizing the pickle and joblib libraries. We made use of the Pickle Library. The pickle module implements binary protocols for serializing and de-serializing a Python object structure. *"Pickling"* is the process whereby a Python object hierarchy is converted into a byte stream, and *"unpickling"* is the inverse operation, whereby a byte stream (from a binary file or bytes-like object) is converted back into an object hierarchy.

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

    self.features = list(X.columns)
    self.X_train = X
    self.y_train = y
    self.train_size = X.shape[0]
    self.num_feats = X.shape[1]

    for feature in self.features:
        self.likelihoods[feature] = {}
        self.pred_priors[feature] = {}

        for feat_val in np.unique(self.X_train[feature]):
            self.pred_priors[feature].update({feat_val: 0})

            for outcome in np.unique(self.y_train):
                self.likelihoods[feature].update({feat_val+'_'+outcome:0})
                self.class_priors.update({outcome: 0})

    self._calc_class_prior()
    self._calc_likelihoods()
    self._calc_predictor_prior()
```

Fig 9: Python code sample for Fit method

3.RESULT:

After training the model and saving it using pickle library, we have to load the model. Now we have to give testing dataset to the predict method of that model to make predictions. After predicting the values we have to calculate the accuracy. Accuracy is the number of correctly predicted data points out of all the data points. More formally, it is defined as the number of true positives and true negatives divided by the number of true positives, true negatives, false positives, and false negatives. To calculate the accuracy of testing dataset we have to pass the predicted values and true values to the accuracy_score method. And we got the test accuracy as 85.9%

```
def accuracy_score(y_true, y_pred):
    """score = (y_true - y_pred) / len(y_true) """
    return round(float(sum(y_pred == y_true))/float(len(y_true)) * 100 ,2)
```

Fig 10: Python code sample of Accuracy_score method

```
acc=accuracy_score(yt, y_pred)

print("Test Accuracy:",acc)
```

```
Test Accuracy: 85.9
```

```
data_crosstab = pd.crosstab(result['ypred'],
                            result['ytrue'],
                               margins = False)
print(data_crosstab)
```

```
ytrue     N    V
ypred
N      1090  112
V       170  628
```

Fig 11: Output Samples

4.REFERENCES:

[1] Qiong Wang, George M Garrity, James M Tiedje, and James R Cole. Naive bayesian classifier for rapid assignment of rrna sequences into the new bacterial taxonomy. Applied and environmental microbiology, 73(16):5261–5267, 2007.

[2] Irina Rish. An empirical study of the naive bayes classifier. In IJCAI 2001 workshop on empirical methods in artificial intelligence, pages 41–46, 2001.