Experiment No.2

Title: Implementation of Naïve Bayesian algorithm for classification

Batch: B2 **Roll No.: 16010420117** **Experiment No.:2**

Aim: Implementation of Naïve Bayesian algorithm for classification

---

Resources needed: Any RDBMS, Java

Theory:

A Bayesian classifier is a simple probabilistic classifier. Bayesian classifier can predict membership probabilities such as the probabilities that a sample belongs to a particular class or groupings.

Bayesian classification is based on Bayes theorem and this technique tends to be highly accurate and fast, making it useful on large databases.

Naïve Bayesian Classification Algorithm:

The operation of the Naïve Bayesian is as follows,

1) Let *D* be a training set of tuples and their associated class labels. As usual, each tuple is represented by an *n*-dimensional attribute vector, $\boldsymbol{X} = (x1, x2, :::, xn)$, depicting *n* measurements made on the tuple from *n* attributes, respectively, *A*1, *A*2, : : : , *An*.

2) Suppose that there are m classes C1,C2,……..,Cm, Given a tuple, $\boldsymbol{X}$, the classifier will predict that $\boldsymbol{X}$ belongs to the class having the highest posterior probability, conditioned on $\boldsymbol{X}$. That is, the na¨ıve Bayesian classifier predicts that tuple $\boldsymbol{X}$ belongs to the class *Ci* if and only if,

$$P(C_i|X) > P(C_j|X) \quad \text{for } 1 \leq j \leq m, j \neq i.$$

The class *Ci* for which $P(Ci\,|\boldsymbol{X})$ is maximized is called the *maximum posteriori hypothesis*. 3)

Using Bayes' theorem,

$$P(C_i|\boldsymbol{X}) = \frac{P(\boldsymbol{X}|C_i)P(C_i)}{P(\boldsymbol{X})}.$$

As $P(\boldsymbol{X})$ is constant for all classes, only $P(\boldsymbol{X}/C_i)P(C_i)$ needs to be maximized. If the class prior probabilities are not known, then it is commonly assumed that the classes are equally likely, that is, $P(C_1)= P(C_2)= \ldots\ldots = P(C_m)$, and we would therefore maximize $P(\boldsymbol{X}/C_i)$. Otherwise, we maximize $P(\boldsymbol{X}/C_i)P(C_i)$. Note that the class prior probabilities may be estimated by $P(C_i)= |C_{i,D}| / |D|$, where $|Ci,D|$ is the number of training tuples of class C_i in *D*.

This presumes that the attributes' values are conditionally independent of one another, given the class label of the tuple (i.e., that there are no dependence relationships among the attributes). Thus,

$$P(X|C_i) = \prod_{k=1}^{n} P(x_k|C_i)$$
$$= P(x_1|C_i) \times P(x_2|C_i) \times \cdots \times P(x_n|C_i)$$

We can easily estimate the probabilities $P(x_1/C_i)$, $P(x_2/C_i)$, …… , $P(x_n/C_i)$ from the training tuples. Recall that here x_k refers to the value of attribute A_k for tuple $\boldsymbol{X}$. For each attribute, we look at whether the attribute is categorical or continuous-valued.

4) Sample X is therefore assigned to class Ci if and only if P(X/Ci).P(Ci)>P(X/Cj).P(Cj) for i<=j<=m. y≠1 In other words if it is assigned to the class C for which P(X/Ci).P(Ci) is Max.

Procedure / Approach /Algorithm / Activity Diagram:

1. Identify attributes suitable for applying classification algorithm
2. Implement **Naïve Bayesian** on your dataset.
3. Apply **Naïve Bayesian** to classify unknown tuple.

Results: (Program printout with output / Document printout as per the format)

Program:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import math
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)

def pre_processing(df):
    """ partioning data into features and target """
    X = df.drop([df.columns[-1]], axis = 1)
    y = df[df.columns[-1]]
    return X, y

def train_test_split(x, y, test_size = 0.25, random_state = None):
    """ partioning the data into train and test sets """
    x_test = x.sample(frac = test_size, random_state =
random_state)
```

```
    y_test = y[x_test.index]
    x_train = x.drop(x_test.index)
    y_train = y.drop(y_test.index)
    return x_train, x_test, y_train, y_test
class  NaiveBayes:
    """
        Bayes Theorem:
                                        Likelihood * Class prior probability
                Posterior Probability =    -------------------------------------
                                             Predictor prior probability

                                   P(x|c) * p(c)
                         P(c|x) = ------------------
                                        P(x)
    """

    def __init__(self):

        """
   Attributes:
   likelihoods: Likelihood of each feature per class
   class_priors: Prior probabilities of classes
   pred_priors: Prior probabilities of features
   features: All features of dataset
        """
        self.features = list
        self.likelihoods = {}
        self.class_priors = {}
        self.pred_priors = {}

        self.X_train = np.array
        self.y_train = np.array
        self.train_size = int
        self.num_feats = int

    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()

        # print(self.likelihoods)
        # print(self.class_priors)
        # print(self.pred_priors)

    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

    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_counts().to_dict()

                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


    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)



if __name__ == "__main__":

    #Weather Dataset
    print("\nWeather Dataset:")

    df = pd.read_table("Data\weather.txt")
    #print(df)

    #Split fearures and target
    X,y  = pre_processing(df)

    #Split data into Training and Testing Sets
    X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size = 0.1, random_state = 0)

    #print(X_train, y_train)
    nb_clf = NaiveBayes()
    nb_clf.fit(X_train, y_train)
    #print(X_train, y_train)

    print("Train Accuracy: {}".format(accuracy_score(y_train,
nb_clf.predict(X_train))))
```

```
    print("Test    Accuracy:    {}".format(accuracy_score(y_test,
nb_clf.predict(X_test))))

    #Query 1:
    query = np.array([['Rainy','Mild', 'Normal', 't']])
    print("Query    1:-    {}    --->    {}".format(query,
nb_clf.predict(query)))

    #Query 2:
    query = np.array([['Overcast','Cool', 'Normal', 't']])
    print("Query    2:-    {}    --->    {}".format(query,
nb_clf.predict(query)))

    #Query 3:
    query = np.array([['Sunny','Hot', 'High', 't']])
    print("Query    3:-    {}    --->    {}".format(query,
nb_clf.predict(query)))
```

Output:

```
Weather Dataset:
Train Accuracy: 84.62
Test Accuracy: 100.0
Query 1:- [['Rainy' 'Mild' 'Normal' 't']] ---> ['no']
Query 2:- [['Overcast' 'Cool' 'Normal' 't']] ---> ['yes']
Query 3:- [['Sunny' 'Hot' 'High' 't']] ---> ['no']
```

Post Lab Question- Answers (If Any):

Q.1. What are advantages and disadvantages of Bayesian Classification?

Ans: Advantages

- This algorithm works quickly and can save a lot of time.
- Naive Bayes is suitable for solving multi-class prediction problems.
- If its assumption of the independence of features holds true, it can perform better than other models and requires much less training data.
- Naive Bayes is better suited for categorical input variables than numerical variables.

Disadvantages

- Naive Bayes assumes that all predictors (or features) are independent, rarely happening in real life. This limits the applicability of this algorithm in real-world use cases.
- This algorithm faces the 'zero-frequency problem' where it assigns zero probability to a categorical variable whose category in the test data set wasn't available in the training dataset. It would be best if you used a smoothing technique to overcome this issue.
- Its estimations can be wrong in some cases, so you shouldn't take its probability outputs very seriously.

Q.2. Comment on Laplacian correction.

Ans: Laplace correction is a smoothing technique that helps tackle the problem of zero probability in the Naïve Bayes machine learning algorithm. Using higher alpha values will push the likelihood towards a value of 0.5, i.e., the probability of a word equal to 0.5 for both the positive and negative reviews.

CO: Comprehend basics of ML

Conclusion: In this experiment, I successfully understood and implemented Data preprocessing techniques.

Grade: AA / AB / BB / BC / CC / CD /DD

Signature of faculty in-charge with date

References:

Books/ Journals/ Websites:

1. Han, Kamber, "Data Mining Concepts and Techniques", Morgan Kaufmann 3nd Edition