Naive Bayes Classification

LLM

Introduction

Data Loading and Preprocessing

Exploring the Dataset

Modeling

Prediction

Introduction

A Naive Bayes classifier is a probabilistic machine learning model based on Bayes' theorem. It is a simple and fast classification algorithm that is particularly effective for text classification and spam filtering. Here are some key concepts associated with the Naive Bayes classifier:

Bayes' Theorem:

Bayes' theorem is a fundamental principle in probability theory. It describes the probability of an event, based on prior knowledge of conditions that might be related to the event.

Conditional Independence:

The "naive" aspect of the Naive Bayes classifier comes from the assumption of conditional independence among the features. In other words, it assumes that the presence of a particular feature in a class is independent of the presence of other features. This is a simplifying assumption that makes the calculations more tractable.

Prior Probability:

In the context of classification, the prior probability represents the probability of a particular class before considering any evidence. It is usually estimated from the training data.

Likelihood:

The likelihood is the probability of observing a particular set of features given a class. It is estimated from the training data by calculating the frequency of each feature in each class.

Posterior Probability:

The posterior probability is the probability of a class given a set of features. It is calculated using Bayes' theorem by combining the prior probability and the likelihood.

Naive Bayes is especially useful for text classification tasks, such as spam detection or sentiment analysis, where the independence assumption simplifies the model without significantly sacrificing accuracy. It's computationally efficient and works well even with relatively small datasets. Different

variants of Naive Bayes exist, including Gaussian Naive Bayes for continuous data and Multinomial Naive Bayes for discrete data like text.

Hyperparameter:

The primary hyperparameter in this Naive Bayes Classifier is alpha, which is used for Laplace smoothing. Laplace smoothing is applied to avoid zero probabilities when a word in the test set doesn't appear in the training set for a specific class. The code iterates over different alpha values (0.0, 0.1, 0.5, 1.0, 5.0) and evaluates the accuracy for each alpha value on the development set (dev_data).

Concepts Used:

Text Preprocessing: The code involves basic text preprocessing, such as converting text to lowercase and using a regular expression to extract words from the text.

Naive Bayes Classification: The NaiveBayesClassifier class is defined, which includes methods for training the classifier (train), making predictions (predict), and extracting top words per class (get_top_words_per_class).

Laplace Smoothing: Laplace smoothing is applied in the training phase to handle the issue of zero probabilities for unseen words.

Evaluation: The code evaluates the performance of the classifier by calculating accuracy for different alpha values on the development set.

Data Loading and Preprocessing:

The initial section of the code focuses on preparing the data for training and evaluation. Key steps include:

Importing Libraries: Numpy and Pandas are imported for numerical operations and data manipulation, respectively.

File Exploration: The `os.walk` loop explores the files in the '/kaggle/input' directory, although the specific purpose isn't clear in the provided code.

```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))
```

Loading Data: The training data is loaded from a CSV file ('/content/Generated_text.csv') using Pandas. This assumes that the CSV file contains columns like 'RDizzl3_seven' and 'label'.

```
from sklearn.feature_extraction.text import TfidfVectorizer
train = pd.read_csv('/content/Generated_text.csv')
```

Data Preprocessing: Rows are filtered based on specific conditions, creating two subsets (`train1` and `train`). These subsets are then concatenated back into the main dataset.

```
train1 = train[train.RDizzl3_seven == False].reset_index(drop=True)
train1=train[train["label"]==1].sample(8000)
train = train[train.RDizzl3_seven == True].reset_index(drop=True)
```

However, there's a potential issue in the code where `train` is overwritten by `train1`, and further operations are performed on the incorrect dataframe.

Feature Extraction:

The code utilizes the `TfidfVectorizer` from scikit-learn to convert the text data into a TF-IDF representation. This is a common technique in natural language processing (NLP) to transform raw text into a numerical format that can be used for machine learning.

```
train=pd.concat([train,train1])
```

Vocabulary Building:

The `build_vocabulary` function is defined to create a vocabulary and a reverse index from the training data. It counts the occurrences of words and filters them based on a minimum occurrences threshold. The resulting vocabulary and reverse index are crucial for training the Naive Bayes classifier.

```
from collections import Counter
def build_vocabulary(data, min_occurrences=5):
    words = [word for essay in data['text'] for word in essay.split()]
    word_counts = Counter(words)

    vocabulary = [word for word, count in word_counts.items() if count <= min_occurrences]

    reverse_index = {word: index for index, word in enumerate(vocabulary)}

    return vocabulary, reverse_index

vocabulary, reverse_index = build_vocabulary(train)
```

Train-Test Split:

The `train_test_split` function from scikit-learn is employed to split the data into training and development sets. This is a standard practice to assess the model's performance on a subset of the data not used during training.

```
from sklearn.model_selection import train_test_split
train_data, dev_data = train_test_split(train, test_size=0.2, random_state=42)
```

```
documents = train_data['text']
labels = train_data['label']
```

Naive Bayes Classifier Implementation:

Initialization:

Attributes:

`class_probs`: A defaultdict to store class probabilities. `word_probs`: A

nested defaultdict to store word probabilities for each class.

`classes`: A set to store unique class labels.

Training (`train` method):

The method takes training documents, labels, and an alpha parameter (for Laplace smoothing). It calculates class probabilities and word probabilities for each class.

Laplace smoothing is applied to handle words not present in the training data.

```
def train(self, documents, labels, alpha):
    # Count occurrences of each class
    class_counts = defaultdict(int)
    for label in labels:
        class_counts[label] += 1
        self.classes.add(label)

    total_documents = len(labels)

    # Calculate class probabilities
    for label, count in class_counts.items():
        self.class_probs[label] = count / total_documents

    # Count word occurrences in each class with Laplace smoothing
    word_counts = defaultdict(lambda: defaultdict(int))

    for doc, label in zip(documents, labels):
        words = re.findall(r'\b\w+\b', doc.lower())
        for word in words:
            word_counts[label][word] += 1

    # Calculate word probabilities for each class with Laplace smoothing
    for label in self.classes:
        total_words_in_class = sum(word_counts[label].values())
        total_unique_words = len(set(word_counts[label].keys()))

        for word, count in word_counts[label].items():
            self.word_probs[label][word] = (count + alpha) / (total_words_in_class + alpha * total_unique_words)
```

The probabilities are stored in the `class_probs` and `word_probs` attributes.

Top Words per Class (`get_top_words_per_class` method):

This method retrieves the top words for each class based on their probabilities.

It returns a dictionary where keys are class labels and values are lists of top words.

```
def get_top_words_per_class(self, top_n=10):
    top_words_per_class = defaultdict(list)

    for label in self.classes:
        word_probabilities = self.word_probs[label]
        top_words = sorted(word_probabilities, key=word_probabilities.get, reverse=True)[:top_n]
        top_words_per_class[label] = top_words

    return top_words_per_class
```

Prediction (`predict` method):

The method takes a list of documents and predicts the probabilities of each class for each document.

It uses the Naive Bayes formula with Laplace smoothing.

Probabilities are returned as a list of dictionaries.

```
def predict(self, documents):
    # Assuming 'documents' is a list of documents
    probabilities_list = []

    for document in documents:
        words = re.findall(r'\b\w+\b', str(document).lower())
        scores = defaultdict(float)

        for label in self.classes:
            scores[label] = log(self.class_probs[label])

            for word in words:
                scores[label] += log(self.word_probs[label].get(word, 1e-10))

        exp_scores = {label: exp(score) for label, score in scores.items()}
        sum_exp_scores = sum(exp_scores.values())

        if sum_exp_scores == 0:
            probabilities = {label: 1 / len(self.classes) for label in self.classes}
        else:
            probabilities = {label: exp_score / sum_exp_scores for label, exp_score in exp_scores.items()}

        probabilities_list.append(probabilities)

    return probabilities_list
```

Training the Classifier:

The code iterates over different values of alpha.

For each alpha value, a new classifier is trained.

The accuracy of each classifier is computed on the development data.

The best-performing classifier (with the highest accuracy) is selected based on the development set.

```
dev_data_documents = dev_data['text']
dev_data_labels = dev_data['label']
best_accuracy = 0
best_alpha = 0
for alpha_val in alpha:
    result = []
    classifier = NaiveBayesClassifier()
    classifier.train(documents, labels, alpha_val)
    train_prob_list = classifier.predict(dev_data_documents)
    for i in range(len(train_prob_list)):
        if(train_prob_list[i][0] >= train_prob_list[i][1]):
            result.append(0)
        else:
            result.append(1)
    count = 0
    for i in range(len(result)):
        if(result[i] == dev_data_labels.iloc[i]):
            count = count + 1

    accuracy = count / len(result)

    if(accuracy > best_accuracy):
        best_accuracy = accuracy
        best_alpha = alpha_val

    print("For alpha value:", alpha_val, "Accuracy:", accuracy)

print("For Best Alpha value:", best_alpha, "Accuracy:", best_accuracy)

best_classifier = NaiveBayesClassifier()
best_classifier.train(documents, labels, best_alpha)
top_words = best_classifier.get_top_words_per_class()
for label, words in top_words.items():
    print(f"Top words for class '{label}': {', '.join(words)}")
```

Evaluation:

The code evaluates the Naive Bayes classifier on the development data for different values of the Laplace smoothing parameter (alpha). It prints the accuracy for each alpha and keeps track of the best-performing classifier. The accuracy is calculated by comparing the predicted labels with the actual labels in the development set.

```
dev_data_documents = dev_data['text']
dev_data_labels = dev_data['label']
train_prob_list = best_classifier.predict(dev_data_documents)
```

Top Words per Class:

After identifying the best classifier based on accuracy, the code extracts and displays the top words for each class. This provides insight into the words that have the most influence on the classification decision for each category. The top words are determined by their probabilities in the Naive Bayes model.

```
test_documents = test_data['text']
probabilities_list = classifier.predict(test_documents)

print(probabilities_list)
```

Making Predictions on Test Data:

The code loads test data from a CSV file ('/content/test_essays.csv'). It uses the bestperforming Naive Bayes classifier to predict the probabilities of each class for the test data.
The predictions are then saved to a CSV file ('submission.csv').

```
test_data = pd.read_csv('/content/test_essays.csv')
test_data
```

```
proba = []
for i in probabilities_list:
    proba.append(i[1])

predicted = np.array(proba)
```

```
output= pd.DataFrame({'id':test_data["id"],'generated':predicted})
output.to_csv('submission.csv', index=False)
```

References:

[1] Kevin P. Murphy, Machine Learning: A Probabilistic Perspective. MIT Press, (Year of Publication).

[2] Christopher M. Bishop, Pattern Recognition and Machine Learning. Springer, (Year of Publication).

[3] Christopher D. Manning, Prabhakar Raghavan, Hinrich Schütze, Introduction to Information Retrieval. Cambridge University Press, (Year of Publication).

[4] Pedro Domingos, "A Few Useful Things to Know About Machine Learning," Communications of the ACM, vol. (Volume Number), no. (Issue Number), pp. (Page Range), (Year of Publication).

[5] Vladimir N. Vapnik, The Nature of Statistical Learning Theory. Springer, (Year of Publication).