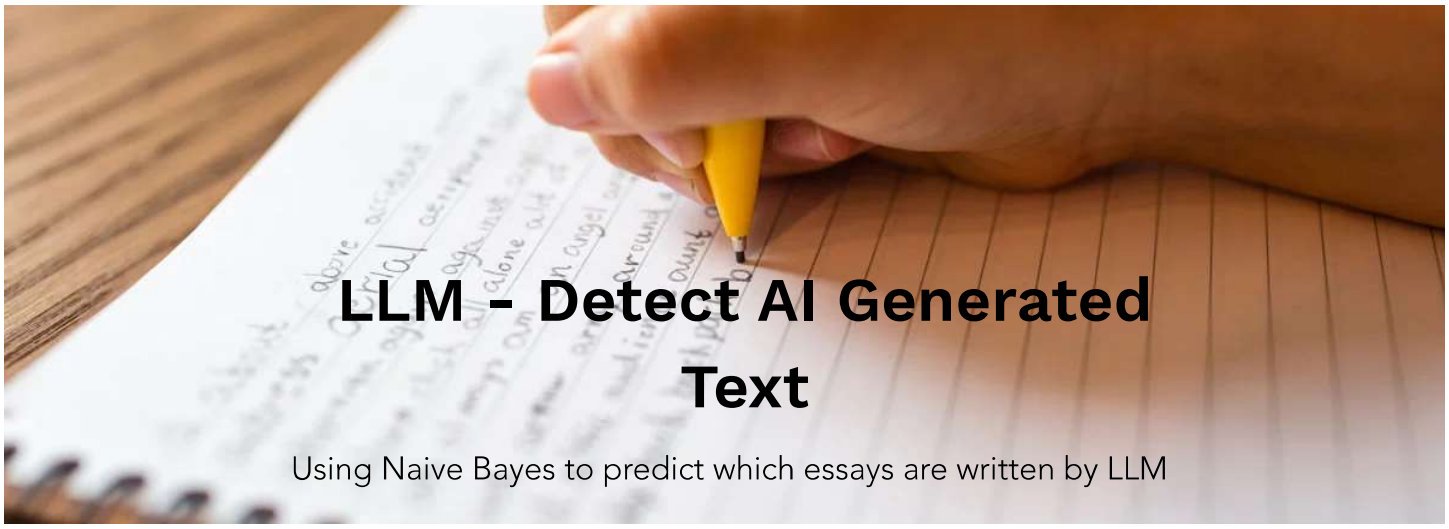


View Notebook

Naive Bayes

Naive Bayes classifiers are probabilistic machine learning models. They are based on the Bayes theorem. Bayes theorem helps us to update the probability of an event based on new evidence or information. Naive Bayes classifiers makes a naive assumption that every feature is independent of one another, although in real life, this is hardly ever true. Another assumption that the model believes is that each feature of the same class makes an equal contribution to the outcome.

Bayes Theorem: $$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Naive Bayes for text classification

Raw text data cannot be fed directly to the model. Most model expect numerical feature vectors with fixed size rather than a raw text blob with variable length. To convert a text data to a matrix with numerical values, we can count the occurrence/frequency of each word/token and provide this information to the model. The model could then find some meaningful pattern between the frequency of the words and it classification.

Building a Naive Bayes classifier for texts generated by LLM(Large language models)

Link to kaggle competition: LLM - Detect AI Generated Text | Kaggle

Data Preprocessing

Before building a naive bayes classifier, we need to preprocess the data. This includes clean the text data. We can remove special characters and numbers as I believe that they do not affect the classification and can be generated by either the

human and LLMs. Next, we can remove the stopwords as they carry very little information. Stopwords are commonly used words in a language. For example, 'a', 'the', 'is' 'are' are stopwords.

```
lef clean_text(string):
    string = re.sub(r"[^a-zA-Z0-9]+", ' ', string)
    text = ''.join([i for i in string if not i.isdigit()])
    text = re.sub('https://.*','',text)
    text = text.lower()
    return text
```

```
stop_words = set(stopwords.words('english'))
def remove_stopwords(texts):
    filtered_texts = []
    for text in texts:
        words = text.split()
        filtered_words = [word for word in words if word not in stop_words]
        filtered_texts.append(' '.join(filtered_words))
    return filtered_texts
```

Lemmatization vs Stemming

Both stemming and lemmatization reduces the number of tokens/features, thus ultimately making the model faster. Stemming works by chopping the end of the tokens. For example 'caring' => 'car'. Stemming is a fast but it is prone to inaccurately chopping the words and changing its context entirely. Lemmatization on the other and looksup the words into a dictionary type database and figures out the root of the word. For example 'caring'=> 'care'. Lemmatization usually takes longer time than stemming but does loose the meaning of the word.
Without understanding whether the word is used as a verb, noun, or adjective, performing the lemmatization with NLTK won't be effective. To perform lemmatization with NLTK in an effective way, the "wordnet" and "pos" parameter should be used.

```
lemmatizer = WordNetLemmatizer()
def nltk_pos_tagger(nltk_tag):
    if nltk_tag.startswith('J'):
        return wordnet.ADJ
    elif nltk_tag.startswith('V'):
        return wordnet.VERB
    elif nltk_tag.startswith('N'):
        return wordnet.NOUN
    elif nltk_tag.startswith('R'):
        return wordnet.ADV
    else:
        return None

def lemmatize_sentence(sentence):
    nltk_tagged = nltk.pos_tag(nltk.word_tokenize(sentence))
    wordnet_tagged = map(lambda x: (x[0], nltk_pos_tagger(x[1])), nltk_tagged)
    lemmatized_sentence = []

    for word, tag in wordnet_tagged:
        if tag is None:
            lemmatized_sentence.append(word)
        else:
            lemmatized_sentence.append(lemmatizer.lemmatize(word, tag))
    return " ".join(lemmatized_sentence)
```

```
porter_stemmer  = PorterStemmer()
def stem_sentence(sentence):
    tokenized = nltk.word_tokenize(sentence)
    stemmed_sentence = []
    for word in tokenized:
            stemmed_sentence.append(porter_stemmer.stem(word))
    return " ".join(stemmed_sentence)
```

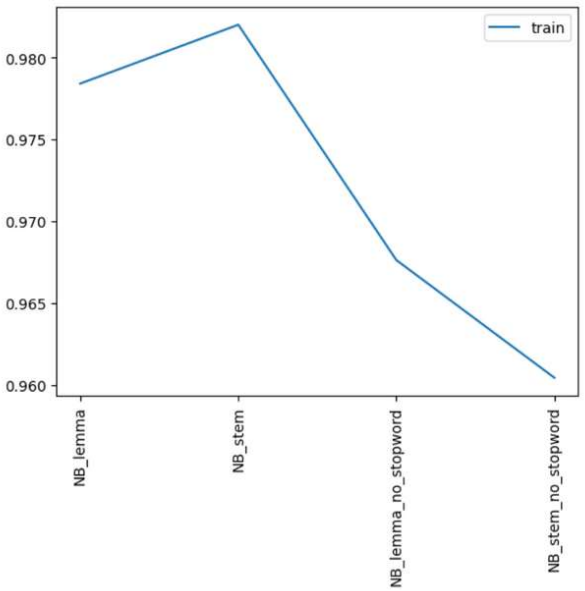
Building the classiffier

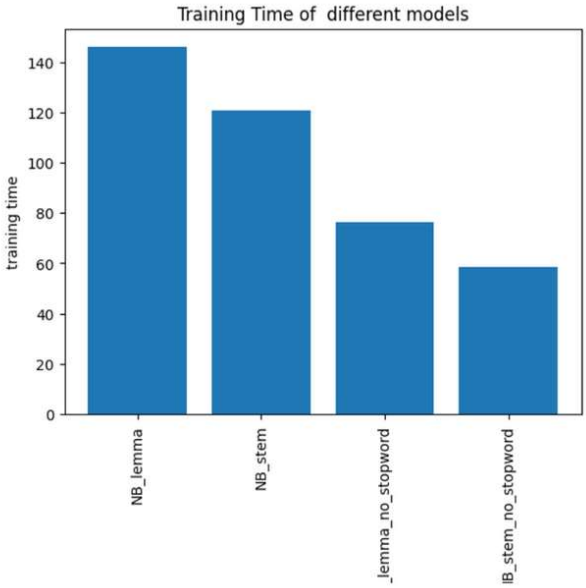
We build a vocabulary which contains all the unique words and the total occurrence of the word in texts. We then create a probability dictionary which stores the tokens and its probability. It also stores the token's probability with respect to the classes. We can then use bayes theorem to find the probability of class when test data is given. Implementation of this is below. Smoothing has to be done because there are token in test data which are not present in the vocabulary built from the training data and without smoothing, we would end up with probability of zero. Working with so many features and multiplying probabilities which are closer to zero might not be handled by simple python operations. After few decimal places, it is possible that the result would be approximated. To overcome this, as we are working with huge number of features, I have used logarithms to calculate the probabilities.

```python
def predict(essay,vocab,probability_dict,llm_texts_count,human_texts_count,llm_prob ):
    missing_token_count = 0
    tokenized = nltk.word_tokenize(essay)

    new_tokenized_list = []
    for token in tokenized:
        if tokenized.count(token) > 5 and token not in new_tokenized_list:
            new_tokenized_list.append(token)

    for token in new_tokenized_list:
        if not vocab.__contains__(token):
            missing_token_count = missing_token_count + 1
        if not vocab.__contains__(token):
            missing_token_count = missing_token_count + 1

    llm_prob_num = 0
    human_prob_num = 0
    for token in new_tokenized_list:
        if not vocab.__contains__(token):
            llm_prob_num = llm_prob_num + math.log10((1)/(llm_texts_count + missing_token_count))
        else:
            llm_prob_num = llm_prob_num + math.log10((probability_dict[token+'__LLM']+llm_texts_count +1)/(llm_texts_count + missing_token_count))
        if not vocab.__contains__(token):
            human_prob_num = human_prob_num + math.log10((1)/(human_texts_count + missing_token_count))
        else:
            human_prob_num = human_prob_num + math.log10((probability_dict[token+'__human']+human_texts_count +1)/(human_texts_count + missing_token_count))

    llm_prob_num = llm_prob_num + math.log10(llm_prob)
    human_prob_num = human_prob_num  + math.log10(1 - llm_prob)

    total_llm_prob = np.divide(10**llm_prob_num,np.add((10**llm_prob_num),(10**human_prob_num)))

    return total_llm_prob
```

Comparison

to compare the accuracy and training time. I trained the above classifier with lemmatized and stemmed input. I also trained the classifier with and without removing the stopwords.





Accuracy of different models

Training time of different models

Naive bayes with lemmatization had the highest accuracy, but it also had the highest training time. Both the accuracy and training time seem to follow the same trend and seem to be directly related. Naive bayes classifier with the lemmatized input has very high development accuracy which possibly indicates that the classifier is overfitted.

My Contribution

To build the naive bayes classifier I developed the code to build vocabulary and find the probabilities. I also built a naive bayes classifier and had to apply smoothing to overcome the zero probability issue. To compute probability from very small numbers I used logarithms for accurate result. Further I compared different preprocessing techniques that can be applied to the text data. I found the lemmatization without removing stopwords gave the highest accuracy on development dataset. But it is possible that the model is overfitted. This method also had the highest training time. Stemming and removing stopwords had the lowest accuracy but was not substantially less than the best accuracy model, but it had

References

1. https://towardsdatascience.com/text-classification-using-naive-bayes-theory-a-working-example-2ef4b7eb7d5a
2. https://www.holisticseo.digital/python-seo/nltk/lemmatize
3. https://www.analyticsvidhya.com/blog/2022/03/building-naive-bayes-classifier-from-scratch-to-perform-sentiment-analysis/
4. https://spotintelligence.com/2022/12/10/stop-words-removal/#:~:text=To%20remove%20stopwords%20with%20Python,your%20own%20list%20of%20stopwords.&text=In%20this%20example%2C%20the%20NLTK,of%20stop%20words%20in%20English.

Email

axc5981@mavs.uta.edu

Follow Me