Popel Emil-Bogdan

Aliens Language Classifier

The Naïve-Bayes classifier is based on Bayes's theorem. The classifier assumes that the features of a dataset are mutually independent, meaning that the occurrence/number of occurences of one feature does not affect the probability of another feature's occurrence. I chose this model because it is well-suited for the task at hand (text classification).

Steps:

1)Preprocess the data:

-In this particular case, the text is saved as it is, removing stop-words not being a good ideea, as we don't know wich ones should be removed. Also, the punctuation isn't removed because the words are separated by spaces and the characters that make up a word can be a form of punctuation.

-The text is not saved in lowercase, as there are uppercase letters inside words that may have a meaning in the alien languages

2)Collect the data sets in DataFrames

- I managed this by making two functions that read (encoding = "UTF-8") normal .txt files or .txt files inside .zip -> they return datasets based on the content of the files,

With the proper columns (id, label) for data_labels and (id, text) for data_samples.

The Dataframes should look like this: [10000 rows x 2 columns]

Row number/ columns	ID	TEXT
0	108345	"Uq$%y gkuKDuZ*KmH quf& qKf Du*& Du*ZX; Kf DY*f..."

1	101973	YcîT » ? *Y*qK# Yuț';Du*ăuH H ènSè ènSè KièâT ...
...	...	...
9998	107985	“sample text..”
9999	106543	"Sample text..”

2) Vectorize -> construct the vocabulary from the training dataset

```
vectorizer = CountVectorizer(analyzer='char',ngram_range=(2,8), binary=True,
encoding="UTF-8", lowercase=False, strip_accents = “unicode”)
```

- I chose to compose the vocabulary (for the final solution) of all the char ngrams (substrings of words) of length ∈ [2, 8).

- When using word ngrams, the predictions were worse.

- For normalization I chose the to remove accents (ènSè -> enSe)

strip_accents = “unicode” -> accepts all characters and normalizes the data using the nfkd normalization.

Essentially, it breaks down the character into two code points, later on filtering the code point of the letter and the code point of the accent:

NFC character	A	m	é		l	i	e
NFC code point	0041	006d	00e9		006c	0069	0065
NFD code point	0041	006d	0065	0301	006c	0069	0065
NFD character	A	m	e	◌́	l	i	e

Source: https://en.wikipedia.org/wiki/Unicode_equivalence

vectorizer.fit(training_data[‘text’])

-construct matrix of ngram occurences for each data entry (row)

-Instead of storing the number of occurences, I chose to store if the ngram is present in the entry or not, seeing that the results improve this way.

Training_data = vec.transform(training_data['sentence'])

-> shape: (10000, 2751595)

	ngram1	ngram2	...	ngramN
0	1	1	1	1
1	1	0	0	1

2	0	0	1	1
3	1	0	1	1

if counvectorizer(binary = True) => ngrams will only be marked as present(1) or not present(0). With binary = False => occurences of ngrams will be counted

3.1) Model creation: MultinomialNB

classifier = MultinomialNB(alpha=0.2,fit_prior=True)

Alpha = 0.2 => Laplace smoothing:

Bayes: P(A | B) = P(A) * P(B | A) / P(B)

Let's say we're trying to classify "UăX; ZYHmH ,ènSè ènSèc" into a language, so we calculate the

(1) P("UăX; ZYHmH ,ènSè ènSèc" | 1) = P(UăX; | 1) x P(ZYHmH | 1) x P(,ènSè | 1) x P(ènSèc | 1)

(2) P("UăX; ZYHmH ,ènSè ènSèc" | 2) = P(UăX; | 2) x P(ZYHmH | 2) x P(,ènSè | 2) x P(ènSèc | 2)

Now let's suppose that our training data doesn't contain "ènSèc":

=> P(ènSèc) = 0 => (1) and (2) become 0

To solve this, we use the alpha parameter to increase the probabilities of these occurences by 'alpha' so that we don't end up with a probability of 0.

Fit_prior = True:

It tells the model to learn prior probabilities.

Models created and the f1_scores:

4 — F1-score: This is the harmonic mean of Precision and Recall and gives a better measure of the incorrectly classified cases than the Accuracy Metric.

$$\text{F1-score} = \left(\frac{\text{Recall}^{-1} + \text{Precision}^{-1}}{2}\right)^{-1} = 2 * \frac{(\text{Precision} * \text{Recall})}{(\text{Precision} + \text{Recall})}$$

F1-Score

We use the Harmonic Mean since it penalizes the extreme values.

Source: https://medium.com/analytics-vidhya/accuracy-vs-f1-score-6258237beca2

#0.7134069228790301

vectorizer = CountVectorizer(analyzer='word',ngram_range=(1,4), binary=False, encoding="UTF-8", lowercase=False, strip_accents = "unicode")
classifier = MultinomialNB(alpha=0.2,fit_prior=True)

#0.7143629995610228

vectorizer = CountVectorizer(analyzer='word',ngram_range=(1,4), binary=True,

encoding="UTF-8", lowercase=False, strip_accents = "unicode")
# classifier = MultinomialNB(alpha=0.2,fit_prior=True)

#0.6940176180490499

vectorizer = CountVectorizer(analyzer='word',ngram_range=(1,4), binary=True, encoding="UTF-8", lowercase=False, strip_accents = "unicode")
classifier = MultinomialNB(alpha=0.001,fit_prior=True)
0.7651684359697467
vectorizer = CountVectorizer(analyzer='char',ngram_range=(2,8), binary=True, encoding="UTF-8", lowercase=False, strip_accents = "unicode")
classifier = MultinomialNB(alpha=0.2,fit_prior=True)

->best one => confusion matrix:

1564	296	264
281	1093	76
155	111	1160

```
# 0.70335082868369
# classifier = MultinomialNB(alpha=0.1,fit_prior=True)
# vectorizer = CountVectorizer(analyzer='word',ngram_range=(1,10), binary=True,
encoding="UTF-8")
# 0.759488074162559
# vectorizer = CountVectorizer(analyzer='char',ngram_range=(1,10), binary=True,
encoding="UTF-8", strip_accents="unicode")
# classifier = MultinomialNB(alpha=0.3,fit_prior=True)
# 0.7644664447357888
# vectorizer = CountVectorizer(analyzer='char',ngram_range=(1,7), binary=True,
encoding="UTF-8")
# classifier = MultinomialNB(alpha=0.1,fit_prior=True)
# 0.7474784963959363
# classifier = MultinomialNB(alpha=0.1,fit_prior=True)
# vectorizer = CountVectorizer(analyzer='char',ngram_range=(1,7), binary=True,
encoding="UTF-8", strip_accents="ascii")
# 0.7650995741960854
# vectorizer = CountVectorizer(analyzer='char',ngram_range=(1,7), binary=True,
encoding="UTF-8", strip_accents="unicode")
# classifier = MultinomialNB(alpha=0.1,fit_prior=True)
# # 0.7607096885228519
# classifier = MultinomialNB(alpha=0.5,fit_prior=True)
# vectorizer = CountVectorizer(analyzer='char',ngram_range=(1,8), binary=True,
encoding="UTF-8")
# # 0.7640544257626006
# vectorizer = CountVectorizer(analyzer='char',ngram_range=(2,8), binary=True,
encoding="UTF-8")
# classifier = MultinomialNB(alpha=0.2,fit_prior=True)
```

3.2)Model Creation: Suport Vector Classification

I chose a SVC for the final version of the code with a linear kernel because it’s best suited for text classification, given the high number of features.

In our case, our data is linearly separable, which is exactly what a linear SVC needs. Also, it is less prone to overfitting.

C parameter:

It tells the SVM how much error is acceptable. With a higher C, the classifier will try to be as strict as possible when classifying. If the C is too high, then the classifier will be oferfitted and it will not be precise. Same goes for a C that's too small.

It's the only parameter that's relevant when using a linear kernel.

Gamma parameter (for rbf, poly, sigmoid kernels):

Intuitively, the gamma parameter defines how much a single training example in has influence

->small gamma=> support vectors are too big, they can encapsulate the whole training set

->big gamma => support vectors are small, they only consider data points that lie close to the hyperplane of separation.

Models created and the f1_scores:

```
classifier = svm.SVC(C=10,kernel= "linear", gamma=100)

# 0.7253103356973932
# classifier = svm.SVC(C=1,kernel= "rbf", gamma=100)
# vec = CountVectorizer(analyzer="char", binary=True, encoding="UTF-8",
ngram_range=(2,6))

# 0.6843664065346209
# vec = CountVectorizer(analyzer="char", binary=True, encoding="UTF-8",
ngram_range=(1,2))
# classifier= svm.SVC(C=10,kernel= "poly", gamma=100)
```

0.1920443274321253
vec = CountVectorizer(analyzer="char", binary=True, encoding="UTF-8", ngram_range=(1,2))
classifier= svm.SVC(C=100,kernel= "linear", gamma=100)

=> OVERFITTING

#0.7641317247587457

#classifier = svm.SVC(C=5, kernel= "linear", gamma="scale")

#vectorizer = CountVectorizer(analyzer='char',ngram_range=(2,8), binary=True, encoding="UTF-8", lowercase=False, strip_accents = "unicode")

=>Confusion matrix:

1564	296	246
281	1093	76
155	111	1160