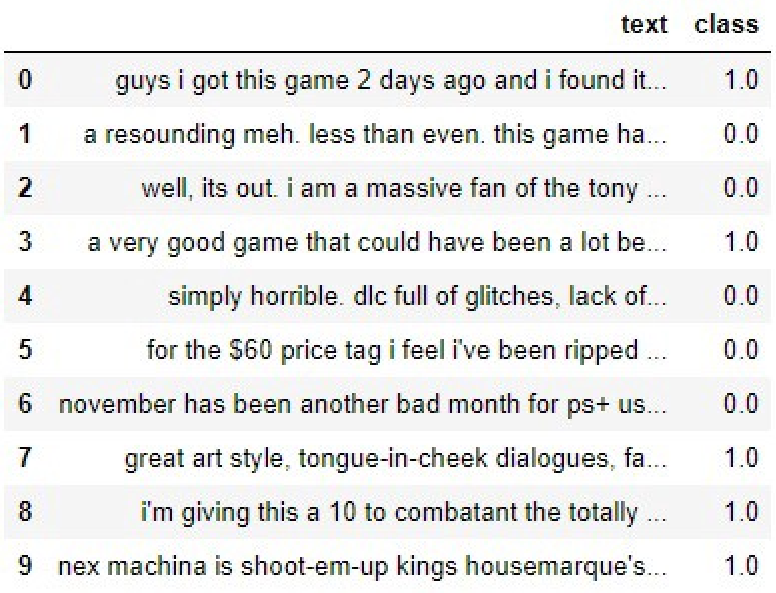
Martín Pellarolo Follow
Feb 21, 2018 · 7 min read

In the Deep Learning era we could be tempted to jump directly to complex architectures since they have proved to yield state of the art results across many domains. However, we should always first create a solid baseline that we'll need to beat to demostrate the value of more advanced models. For that purpose, Naive Bayes is a useful technique to apply in text classification problems.

Text classification aims to assign documents (emails, tweets, posts, news, etc) to one or many categories. Some applications are spam filtering, customer support ticket routing, authorship identification or topic prediction. Another case is sentiment analysis, whose objective is to determine the writer's point of view about a particular topic, product, service, etc.

Data set

Metacritic.com is a review website for movies, videogames, music and tv shows. We will be using a dataset with videogames reviews scraped from the site. Each review contains a text opinion and a numeric score (0 to 100 scale). After keeping just highly-polarized reviews (filtering by scores) and balancing the number of examples in each class we end up with 40838 documents, 50% being positive (class = 1) and the remaining 50% being negative (class = 0).



	text	class
0	guys i got this game 2 days ago and i found it...	1.0
1	a resounding meh. less than even. this game ha...	0.0
2	well, its out. i am a massive fan of the tony ...	0.0
3	a very good game that could have been a lot be...	1.0
4	simply horrible. dlc full of glitches, lack of...	0.0
5	for the $60 price tag i feel i've been ripped ...	0.0
6	november has been another bad month for ps+ us...	0.0
7	great art style, tongue-in-cheek dialogues, fa...	1.0
8	i'm giving this a 10 to combatant the totally ...	1.0
9	nex machina is shoot-em-up kings housemarque's...	1.0

Fig.1 — First 10 rows of the data

Positives examples:

> *"guys i got this game 2 days ago and i found it great in every aspect so why zero everything improved i love the story . i love to be like a commander as commander shepherd in mass effect and guide my army it feels good . maybe some fighting mechanic in the first chapter was better but it is great as a final"*

> *"a very good game that could have been a lot better.\r the graphics are good, the gameplay is smooth and the campaing, besides it is a little short, it is very enjoyable\r :3"*

Negative examples:

> *"genuinely one of the worst, most boring and feeble games i've ever played. thank god i got it on ea access. no campaign. simple and boring customisation. ui problems everywhere. can't understand any of the menus. bad gameplay. bad sound. awful flight mechanics. boring maps with not enough cover. terrible character movement. not actually great graphics. i could go on. but it's boring me."*

> *"simply horrible. dlc full of glitches, lack of information, boring events and short and expensive. don`t buy, run away and play another game, forget destiny and bungie."*

The objective is to train a classifier that given a text review determine if it’s positive or negative.

We split our data into a standard 80 % training set and 20% test set.

```
X_train, X_test, y_train, y_test =
train_test_split(df['text'].values,
                 df['class'].values,
                 test_size=0.2)
```

Preparing the data

Computers do not understand about plain text, so we need to transform our documents into numeric representations.

Tokenization

The first step is tokenization, the process of splitting text into relevant units (characters, words, phrases or others). This units are called tokens, and we will use word-level tokenization. For instance:

“a very good game that could have been a lot better”

results in:

[‘a’, ‘very’, ‘good, ‘game’, ‘that’, ‘could’, ‘have’, ‘been’, ‘a’, ‘lot’, ‘better’]

There are several tonekizers out there that we can use: spaCy, keras, NLTK, fast.ai, just to name a few.

```
re_tok = re.compile(f'([{string.punctuation}“”¨«»®´·º½¾¿¡§£
₤‘’])')
def tokenize(s): return re_tok.sub(r' \1 ', s).split()
```

Bag-of-Words (BoW)

The next step is to create a numerical feature vector for each document. BoW counts the number of times that tokens appear in every document of the collection. It returns a matrix with the next characteristics:

Number of columns = number of unique tokens in the whole collection of documents (vocabulary).

Number of rows = number of documents in the whole collection of documents.

Every cell contains the frequency of a particular token (column) in a particular document (row).

We use the class CountVectorizer from scikit-learn library.

```
vect = CountVectorizer(tokenizer=tokenize)
tf_train = vect.fit_transform(X_train)
tf_test = vect.transform(X_test)
```

Result

```
tf_train
<32670x45094 sparse matrix of type '<class 'numpy.int64'>'
       with 2560062 stored elements in Compressed Sparse Row
format>

tf_test
<8168x45094 sparse matrix of type '<class 'numpy.int64'>'
       with 636636 stored elements in Compressed Sparse Row
format>
```

32670 training documents, 8168 test documents and 45094 tokens in the vocabulary.

Pd: just a small fraction of tokens are present in each document, so most part of cells are filled with zeros. For efficient storage, sparse matrices are used.

Building a classifier

Naive Bayes is a probabilistic learning method based on applying Bayes' theorem. There are some variations of the algorithm but here we will work with Multinomial.

Multinomial Naive Bayes

Let each row of our term-document training matrix be the feature count vector for training case i.

```
tf_train[i] # feature count vector for training case i
y_train[i] # label for training case i
```

The count vectors are defined as:

p = sum of all feature count vectors with label 1

```
p = tf_train[y_train==1].sum(0) + 1
```

q = sum of all feature count vectors with label 0

```
q = tf_train[y_train==0].sum(0) + 1
```

Notice that we add 1 to both count vectors to ensure that every token appear at least one time in each class.

The log-count ratio **r** is:

```
r = np.log((p/p.sum()) / (q/q.sum()))
```

And *b:*

```
b = np.log(len(p) / len(q))
```

Just the ratio of number of positive and negative training cases.

Predictions

With calculated coefficients we can now generate predictions on test set. Since we are trying to fit a linear classifier, the form of the linear equation is:

$y = mx + b$

```
pre_preds = tf_test @ r.T + b
preds = pre_preds.T > 0
accuracy = (preds == y_test).mean()
```

We get a **90.38 %** accuracy.

The method is called "naive" because assumes independence between features, i.e. they do not interact. Additionally, there's an assumption by Bag-of-Words in which the order / position of tokens does not matter . Although in real data both assumptions are unlikely to be true, the method achieves good results.

Logistic Regression

Rather than calculating **r** and *b* coefficients trough theoretical model , we can train a model to learn them using training data. Let's fit a Logistic Regression:

```
model = LogisticRegression(C=0.2, dual=True)
model.fit(tf_train, y_train)
preds = m.predict(tf_test)
accuracy = (preds == y_test).mean()
```

We get a **92.75 %** accuracy.

Model inspection

A confusion matrix is a table that allows us to visualize the performance of a classification algorithm. There are two kind of errors:

False positive: negative opinion classified as positive (305).

False negative: positive opinion classified as negative (287).

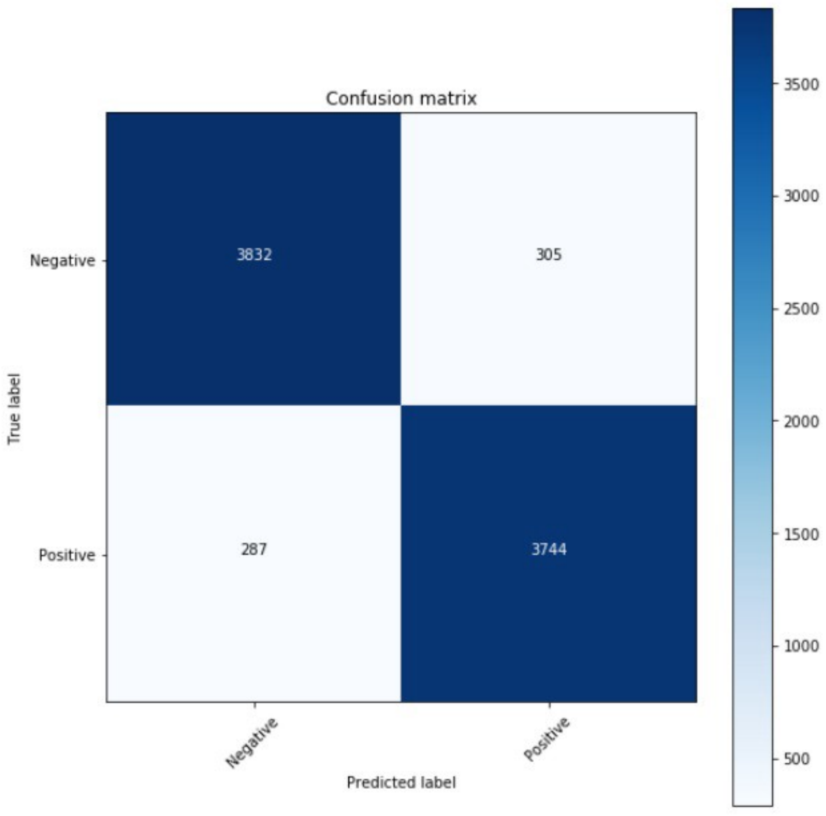


Fig. 2 — Confusion matrix

This great post shows a visualization about the most relevant tokens that the model is using to classify documents. We just need to map Logistic Regression coefficients with their corresponding tokens and finally sort them by importance.

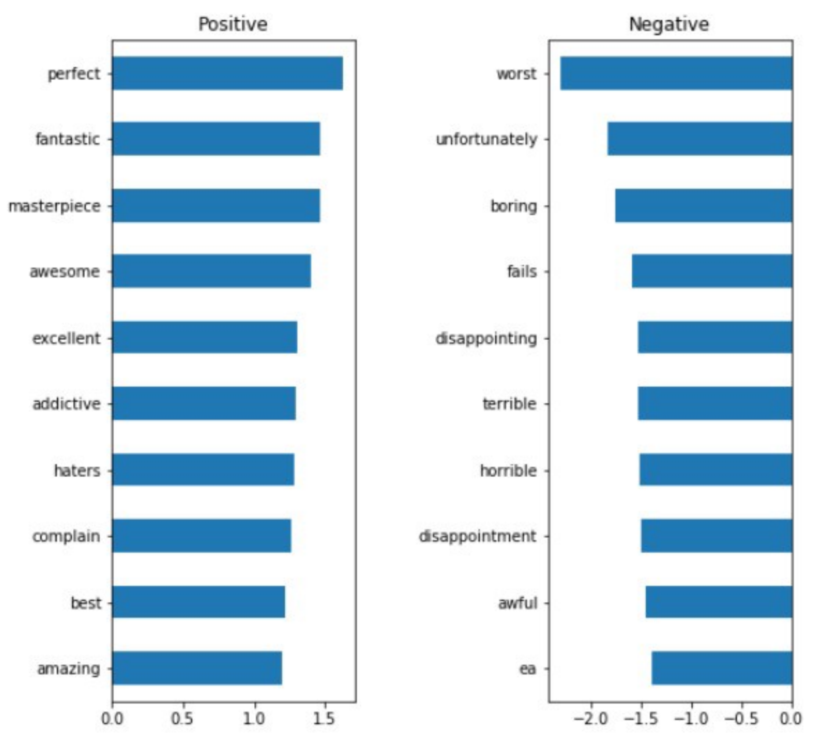


Fig.3 — Most relevant words

We can see that in most cases the model is using generalizable words (“perfect”, “fantastic”, “worst”, “unfortunately”), with the exception of “ea”. It seems like there are a lot of negative reviews about EA (Electronic Arts, a video game company) so the classifier is overfitting with this term. If we remove it from the documents we get a slightly better accuracy: **92.8 %**. Maybe looking for more than 10 tokens to discover similar cases and repeating the same step could lead to a better score.

Improvements

There are words that take place in several documents from both classes, so they do not give relevant information. To overcome this problem there is a useful technique called **term frequency-inverse document frequency (td-idf)**. It contemplates not just frequency but also how unique the word is.

Furthermore, in the BoW model that we created, each token represents a single word. That’s called **unigram** model. We can also try adding **bigrams**, where tokens represents pairs of consecutive words.

Scikit-learn implements td-idf with the TfidfVectorizer class.

```
vect = TfidfVectorizer(strip_accents='unicode',
tokenizer=tokenize, ngram_range=(1, 2), max_df=0.9,
min_df=3, sublinear_tf=True)
tfidf_train = vect.fit_transform(X_train)
tfidf_test = vect.transform(X_test)
```

We fit a new model

```
model = LogisticRegression(C=30, dual=True)
model.fit(tfidf_train, y_train)
preds = m.predict(tfidf_test)
accuracy = (preds==y_test).mean()
```

And we reach a **94.83 %** accuracy.

Conclussion

Naive Bayes is a simple but useful technique for text classification tasks. We can create solid baselines with little effort and depending on business needs explore more complex solutions.

There are also some modifications that can be applied to our classifier in order to get a better accuracy, but are out of the scope of this post. You can learn more on “Baselines and Bigrams: Simple, Good Sentiment and Topic Classification” paper.

A fantastic explanation by Jeremy Howard can be found on lesson 10 of fast ai Machine Learning course (I also highly recommend his Deep Learning set of lectures).

You can download data and code here.

References

https://nlp.stanford.edu/IR-book/html/htmledition/naive-bayes-text-classification-1.html

Manning, Cristopher D. and Wang, Sida. 2012. Baselines and Bigrams: Simple, Good Sentiment and Topic Classification

Fast.ai Machine Learning course—Lesson 10

A Gentle Introduction to the Bag-of-Words Model—Machine Learning Mastery

https://blog.insightdatascience.com/how-to-solve-90-of-nlp-problems-a-step-by-step-guide-fda605278e4e