17 MARCH 2015 / NLP

Naive bayes: Predicting movie review sentiment

Sentiment analysis is a field dedicated to extracting subjective emotions and feelings from text. One common use of sentiment analysis is to figure out if a text expresses negative or positive feelings. Written reviews are great datasets for doing sentiment analysis, because they often come with a score that can be used to train an algorithm.

Naive bayes is a popular algorithm for classifying text. Although it is fairly simple, it often performs as well as much more complicated solutions.

In this post, we'll use the naive bayes algorithm to predict the sentiment of movie reviews. We'll also do some natural language processing to extract features to train the algorithm from the text of the reviews.

Before we classify

We have a csv file containing movie reviews. Each row in the dataset contains the text of the review, and whether the tone of the review was classified as positive(`1`), or negative(`-1`).

We want to predict whether a review is negative or positive given only the text. In order to do this, we'll train an algorithm using the reviews and classifications in `train.csv`, and then make predictions on the reviews in `test.csv`. We'll then be able to calculate our error using the actual classifications in `test.csv`, and see how good our predictions were.

For our classification algorithm, we're going to use naive bayes. A naive bayes classifier works by figuring out the probability of different attributes of the data being associated with a certain class. This is based on bayes' theorem. The theorem is $P(A \mid B) = \frac{P(B|A),P(A)}{P(B)}$. This basically states "the probability of A given that B is true equals the probability of B given that A is true times the probability of A being true, divided by the probability of B being true."

Let's do a quick exercise to understand this rule better.

```
# Here's a running history for the past week.
# For each day, it contains whether or not the person ran, and whether or not they were tired.
days = [["ran", "was tired"], ["ran", "was not tired"], ["didn't run", "was tired"], ["ran", "was

# Let's say we want to calculate the odds that someone was tired given that they ran, using bayes'
# This is P(A).
prob_tired = len([d for d in days if d[1] == "was tired"]) / len(days)
# This is P(B).
prob_ran = len([d for d in days if d[0] == "ran"]) / len(days)
# This is P(B|A).
prob_ran_given_tired = len([d for d in days if d[0] == "ran" and d[1] == "was tired"]) / len([d fo

# Now we can calculate P(A|B).
prob_tired_given_ran = (prob_ran_given_tired * prob_tired) / prob_ran

print("Probability of being tired given that you ran: {0}".format(prob_tired_given_ran))
```

```
Probability of being tired given that you ran: 0.6
```

Naive bayes intro

Let's try a slightly different example. Let's say we still had one classification -- whether or not you were tired. And let's say we had two data points -- whether or not you ran, and whether or not you woke up early. Bayes' theorem doesn't work in this case, because we have two data points, not just one.

This is where naive bayes can help. Naive bayes extends bayes' theorem to handle this case by assuming that each data point is independent.

The formula looks like this: $P(y \mid x_1, \dots, x_n) = \frac{P(y) \prod_{i=1}^{n} P(x_i|y)}{P(x_1, \dots, x_n)}$. This is saying "the probability that classification y is correct given the features x_1, x_2, and so on equals the probability of y times the product of each x feature given y, divided by the probability of the x features".

To find the "right" classification, we just find out which classification ($P(y \mid x_1, \dots, x_n)$) has the highest probability with the formula.

```
# Here's our data, but with "woke up early" or "didn't wake up early" added.
days = [["ran", "was tired", "woke up early"], ["ran", "was not tired", "didn't wake up early"],

# We're trying to predict whether or not the person was tired on this day.
new_day = ["ran", "didn't wake up early"]

def calc_y_probability(y_label, days):
  return len([d for d in days if d[1] == y_label]) / len(days)

def calc_ran_probability_given_y(ran_label, y_label, days):
  return len([d for d in days if d[1] == y_label and d[0] == ran_label]) / len(days)

def calc_woke_early_probability_given_y(woke_label, y_label, days):
  return len([d for d in days if d[1] == y_label and d[2] == woke_label]) / len(days)

denominator = len([d for d in days if d[0] == new_day[0] and d[2] == new_day[1]]) / len(days)
# Plug all the values into our formula.  Multiply the class (y) probability, and the probability o
prob_tired = (calc_y_probability("was tired", days) * calc_ran_probability_given_y(new_day[0], "wa

prob_not_tired = (calc_y_probability("was not tired", days) * calc_ran_probability_given_y(new_day

# Make a classification decision based on the probabilities.
classification = "was tired"
if prob_not_tired > prob_tired:
  classification = "was not tired"
print("Final classification for new day: {0}. Tired probability: {1}. Not tired probability: {2}."
```

```
Final classification for new day: was tired. Tired probability: 0.10204081632653061. Not tired probability
```

Finding word counts

We're trying to determine if a data row should be classified as negative or positive. Because of this, we can ignore the denominator. As you saw in the last code example, it will be a constant in each of the possible classes, thus affecting each probability equally, so it won't change which one is greatest (dividing 5 by 2 and 10 by 2 doesn't change the fact that the second number is bigger).

So we have to calculate the probabilities of each classification, and the probabilities of each feature falling into each classification.

We were working with several discrete features in the last example. Here, all we have is one long string. The easiest way to generate features from text is to split the text up into words. Each word in a review will then be a feature that we can then work with. In order to do this, we'll split the reviews based on whitespace.

We'll then count up how many times each word occurs in the negative reviews, and how many times each word occurs in the positive reviews. This will allow us to eventually compute the probabilities of a new review belonging to each class.

```
# A nice python class that lets you count how many times items occur in a list
from collections import Counter
import csv
import re

# Read in the training data.
with open("train.csv", 'r') as file:
  reviews = list(csv.reader(file))

def get_text(reviews, score):
  # Join together the text in the reviews for a particular tone.
  # We lowercase to avoid "Not" and "not" being seen as different words, for example.
  return " ".join([r[0].lower() for r in reviews if r[1] == str(score)])

def count_text(text):
  # Split text into words based on whitespace.  Simple but effective.
  words = re.split("\s+", text)
  # Count up the occurence of each word.
  return Counter(words)

negative_text = get_text(reviews, -1)
positive_text = get_text(reviews, 1)
# Generate word counts for negative tone.
negative_counts = count_text(negative_text)
# Generate word counts for positive tone.
positive_counts = count_text(positive_text)

print("Negative text sample: {0}".format(negative_text[:100]))
print("Positive text sample: {0}".format(positive_text[:100]))
```

```
Negative text sample: plot : two teen couples go to a church party drink and then drive . they get into an
Positive text sample: films adapted from comic books have had plenty of success whether they're about super
```

Making predictions

Now that we have the word counts, we just have to convert them to probabilities and multiply them out to get the predicted classification. Let's say we wanted to find the probability that the review `didn't like it` expresses a negative sentiment. We would find the total number of times the word `didn't` occured in the negative reviews, and divide it by the total number of words in the negative reviews to get the probability of x given y. We would then do the same for `like` and `it`. We would multiply all three probabilities, and then multiply by the probability of any document expressing a negative sentiment to get our final probability that the sentence expresses negative sentiment.

We would do the same for positive sentiment, and then whichever probability is greater would be the class that the review is assigned to.

To do all this, we'll need to compute the probabilities of each class occuring in the data, and then make a function to compute the classification.

```
import re
from collections import Counter

def get_y_count(score):
  # Compute the count of each classification occuring in the data.
  return len([r for r in reviews if r[1] == str(score)])

# We need these counts to use for smoothing when computing the prediction.
positive_review_count = get_y_count(1)
negative_review_count = get_y_count(-1)

# These are the class probabilities (we saw them in the formula as P(y)).
prob_positive = positive_review_count / len(reviews)
prob_negative = negative_review_count / len(reviews)

def make_class_prediction(text, counts, class_prob, class_count):
  prediction = 1
  text_counts = Counter(re.split("\s+", text))
  for word in text_counts:
      # For every word in the text, we get the number of times that word occured in the reviews fo
      # Smoothing ensures that we don't multiply the prediction by 0 if the word didn't exist in t
      # We also smooth the denominator counts to keep things even.
      prediction *=  text_counts.get(word) * ((counts.get(word, 0) + 1) / (sum(counts.values()) +
  # Now we multiply by the probability of the class existing in the documents.
  return prediction * class_prob

# As you can see, we can now generate probabilities for which class a given review is part of.
# The probabilities themselves aren't very useful -- we make our classification decision based on
print("Review: {0}".format(reviews[0][0]))
print("Negative prediction: {0}".format(make_class_prediction(reviews[0][0], negative_counts, prob
print("Positive prediction: {0}".format(make_class_prediction(reviews[0][0], positive_counts, prob
```

```
Review: plot : two teen couples go to a church party drink and then drive . they get into an accident . one
Negative prediction: 3.0050530362356505e-221
Positive prediction: 1.3071705466906793e-226
```

Predicting the test set

Now that we can make predictions, let's predict the probabilities on the reviews in `test.csv`. You'll get misleadingly good results if you predict on the reviews in `train.csv`, because the probabilities were generated from it (and this, the algorithm has prior knowledge about the data it's predicting on).

Getting good results on the training set could mean that your model is overfit, and is just picking up random noise. Only testing on a set that the model wasn't trained with can tell you if it's performing properly.

```
import csv

def make_decision(text, make_class_prediction):
    # Compute the negative and positive probabilities.
    negative_prediction = make_class_prediction(text, negative_counts, prob_negative, negative_rev
    positive_prediction = make_class_prediction(text, positive_counts, prob_positive, positive_rev

    # We assign a classification based on which probability is greater.
    if negative_prediction > positive_prediction:
      return -1
    return 1

with open("test.csv", 'r') as file:
    test = list(csv.reader(file))

predictions = [make_decision(r[0], make_class_prediction) for r in test]
```

Computing error

Now that we know the predictions, we'll compute error using the area under the ROC curve. This will tell us how "good" the model is -- closer to 1 means that the model is better.

Computing error is very important to knowing when your model is "good", and when it is getting better or worse.

```
actual = [int(r[1]) for r in test]

from sklearn import metrics

# Generate the roc curve using scikits-learn.
fpr, tpr, thresholds = metrics.roc_curve(actual, predictions, pos_label=1)

# Measure the area under the curve.  The closer to 1, the "better" the predictions.
print("AUC of the predictions: {0}".format(metrics.auc(fpr, tpr)))
```

```
AUC of the predictions: 0.680701754385965
```

A faster way to predict

There are a lot of extensions that we could make to this algorithm to make it perform better. We could look at n-grams instead of unigrams. We could remove punctuation and other non-characters. We could remove stopwords. We could also perform stemming or lemmatization.

We don't want to have to code the whole algorithm out every time, though. An easier way to use naive bayes is to use the implementation in scikit-learn. Scikit-learn is a python machine learning library that contains implementations of all the common machine learning algorithms.

```
from sklearn.naive_bayes import MultinomialNB
from sklearn.feature_extraction.text import CountVectorizer
from sklearn import metrics

# Generate counts from text using a vectorizer.  There are other vectorizers available, and lots o
# This performs our step of computing word counts.
vectorizer = CountVectorizer(stop_words='english')
train_features = vectorizer.fit_transform([r[0] for r in reviews])
test_features = vectorizer.transform([r[0] for r in test])

# Fit a naive bayes model to the training data.
# This will train the model using the word counts we computer, and the existing classifications in
nb = MultinomialNB()
nb.fit(train_features, [int(r[1]) for r in reviews])

# Now we can use the model to predict classifications for our test features.
predictions = nb.predict(test_features)

# Compute the error.  It is slightly different from our model because the internals of this proces
fpr, tpr, thresholds = metrics.roc_curve(actual, predictions, pos_label=1)
print("Multinomial naive bayes AUC: {0}".format(metrics.auc(fpr, tpr)))
```

```
Multinomial naive bayes AUC: 0.6509287925696594
```

Next Steps

To learn more, checkout our Dataquest mission on naive bayes.

Vik Paruchuri
Read more posts by this author.

Read More

— Dataquest Data Science Blog —
NLP
Natural Language Processing with Python Tutorial
1 post →

LEARN PYTHON
Apr 06, 2015
Python Counter Class and Probability Mass Functions
This post walks you through a few applications of the Python Counter Class.
VIK PARUCHURI

LEARN PYTHON
Feb 15, 2015
k-means clustering US Senators
In this post, we'll explore cluster US Senators using an interactive Python environment.
VIK PARUCHURI