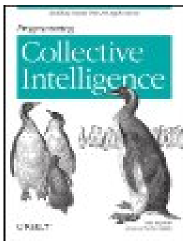
StreamHacker

Tag Archives: bayes

PYTHON

PROGRAMMING COLLECTIVE INTELLIGENCE REVIEW

JULY 11, 2011 | 3 COMMENTS



Programming Collective Intelligence is a great conceptual introduction to many common machine learning algorithms and techniques. It covers classification algorithms such as Naive Bayes and Neural Networks, and algorithmic optimization approaches like Genetic Programming. The book also manages to pick interesting example applications, such as stock price prediction and topic identification.

There are two chapters in particular that stand out to me. First is Chapter 6, which covers **Naive Bayes classification**. What stood out was that the algorithm presented is an online learner, which means it can be updated as data comes in, unlike the NLTK NaiveBayesClassifier, which can be trained only once. Another thing that caught my attention was Fisher's method, which is not implemented in NLTK, but could be with a little work. Apparently **Fisher's method** is great for spam filtering, and is used by the SpamBayes Outlook plugin (which is also written in Python).

Second, I found Chapter 9, which covers Support Vector Machines and Kernel Methods, to be quite intuitive. It explains the idea by starting with examples of linear classification and its shortfalls. But then the examples show that by scaling the data in a particular way first, linear classification suddenly becomes possible. And the kernel trick is simply a neat and efficient way to reduce the amount of calculation necessary to train a classifier on scaled data.

The final chapter summarizes all the key algorithms, and for many it includes commentary on their strengths and weaknesses. This seems like valuable reference material, especially for when you have a new data set to learn from, and you're not sure which algorithms will help get the results you're looking for. Overall, I found Programming Collective Intelligence to be an enjoyable read on my Kindle 3, and highly recommend it to anyone getting started with machine learning and Python, as well as anyone interested in a general survey of machine learning algorithms.

SHARE THIS:

Twitter Reddit Facebook 1

BAYES MACHINELEARNING NLTK PYTHON SPAM SVM

PYTHON

TEXT CLASSIFICATION FOR SENTIMENT ANALYSIS – STOPWORDS AND COLLOCATIONS

MAY 24, 2010 | 55 COMMENTS

Improving feature extraction can often have a significant positive impact on classifier accuracy (and precision and recall). In this article, I'll be evaluating two modifications of the `word_feats` feature extraction method:

1. filter out stopwords
2. include bigram collocations

To do this effectively, we'll modify the previous code so that we can use an arbitrary feature extractor function that takes the words in a file and returns the feature dictionary. As before, we'll use these features to train a Naive Bayes Classifier.

```
import collections
import nltk.classify.util, nltk.metrics
from nltk.classify import NaiveBayesClassifier
from nltk.corpus import movie_reviews

def evaluate_classifier(featx):
    negids = movie_reviews.fileids('neg')
    posids = movie_reviews.fileids('pos')

    negfeats = [(featx(movie_reviews.words(fileids=[f])), 'neg
    posfeats = [(featx(movie_reviews.words(fileids=[f])), 'pos

    negcutoff = len(negfeats)*3/4
    poscutoff = len(posfeats)*3/4

    trainfeats = negfeats[:negcutoff] + posfeats[:poscutoff]
    testfeats = negfeats[negcutoff:] + posfeats[poscutoff:]

    classifier = NaiveBayesClassifier.train(trainfeats)
    refsets = collections.defaultdict(set)
    testsets = collections.defaultdict(set)

    for i, (feats, label) in enumerate(testfeats):
            refsets[label].add(i)
            observed = classifier.classify(feats)
            testsets[observed].add(i)

    print 'accuracy:', nltk.classify.util.accuracy(classifier,
    print 'pos precision:', nltk.metrics.precision(refsets['po
    print 'pos recall:', nltk.metrics.recall(refsets['pos'], t
    print 'neg precision:', nltk.metrics.precision(refsets['ne
    print 'neg recall:', nltk.metrics.recall(refsets['neg'], t
    classifier.show_most_informative_features()
```

Baseline Bag of Words Feature Extraction

Here's the baseline feature extractor for bag of words feature selection.

```
def word_feats(words):
    return dict([(word, True) for word in words])

evaluate_classifier(word_feats)
```

The results are the same as in the previous articles, but I've included them here for reference:

```
accuracy: 0.728
pos precision: 0.651595744681
pos recall: 0.98
neg precision: 0.959677419355
neg recall: 0.476
Most Informative Features
         magnificent = True              pos : neg     =     15.0 :
1.0
         outstanding = True              pos : neg     =     13.6 :
1.0
           insulting = True              neg : pos     =     13.0 :
```

```
1.0
         vulnerable = True              pos : neg    =     12.3 :
1.0
          ludicrous = True              neg : pos    =     11.8 :
1.0
             avoids = True              pos : neg    =     11.7 :
1.0
        uninvolving = True              neg : pos    =     11.7 :
1.0
         astounding = True              pos : neg    =     10.3 :
1.0
        fascination = True              pos : neg    =     10.3 :
1.0
            idiotic = True              neg : pos    =      9.8 :
1.0
```

Stopword Filtering

Stopwords are words that are generally considered *useless*. Most search engines ignore these words because they are so common that including them would greatly increase the size of the index without improving precision or recall. NLTK comes with a stopwords corpus that includes a list of 128 english stopwords. Let's see what happens when we filter out these words.

```
from nltk.corpus import stopwords
stopset = set(stopwords.words('english'))

def stopword_filtered_word_feats(words):
    return dict([(word, True) for word in words if word not in

evaluate_classifier(stopword_filtered_word_feats)
```

And the results for a stopword filtered bag of words are:

```
accuracy: 0.726
pos precision: 0.649867374005
pos recall: 0.98
neg precision: 0.959349593496
neg recall: 0.472
```

Accuracy went down .2%, and *pos precision* and *neg recall* dropped as well! Apparently **stopwords add information to sentiment analysis classification**. I did not include the most informative features since they did not change.

Bigram Collocations

As mentioned at the end of the article on precision and recall, it's possible that including bigrams will improve classification accuracy. The hypothesis is that people say things like "not great", which is a negative expression that the bag of words model could interpret as positive since it sees "great" as a separate word.

To find significant bigrams, we can use nltk.collocations.BigramCollocationFinder along with nltk.metrics.BigramAssocMeasures. The BigramCollocationFinder maintains 2 internal FreqDists, one for individual word frequencies, another for bigram frequencies. Once it has these frequency distributions, it can score individual bigrams using a scoring function provided by BigramAssocMeasures, such chi-square. These scoring functions measure the collocation correlation of 2 words, basically whether the bigram occurs about as frequently as each individual word.

```
import itertools
from nltk.collocations import BigramCollocationFinder
from nltk.metrics import BigramAssocMeasures

def bigram_word_feats(words, score_fn=BigramAssocMeasures.chi_
    bigram_finder = BigramCollocationFinder.from_words(words)
    bigrams = bigram_finder.nbest(score_fn, n)
    return dict([(ngram, True) for ngram in itertools.chain(wo

evaluate_classifier(bigram_word_feats)
```

After some experimentation, I found that using the 200 best bigrams from each file produced great results:

```
accuracy: 0.816
pos precision: 0.753205128205
pos recall: 0.94
neg precision: 0.920212765957
neg recall: 0.692
Most Informative Features
         magnificent = True              pos : neg    =     15.0 :
1.0
         outstanding = True              pos : neg    =     13.6 :
```

```
1.0
           insulting = True              neg : pos    =     13.0 :
1.0
          vulnerable = True              pos : neg    =     12.3 :
1.0
    ('matt', 'damon') = True             pos : neg    =     12.3 :
1.0
       ('give', 'us') = True             neg : pos    =     12.3 :
1.0
           ludicrous = True              neg : pos    =     11.8 :
1.0
         uninvolving = True              neg : pos    =     11.7 :
1.0
              avoids = True              pos : neg    =     11.7 :
1.0
('absolutely', 'no') = True              neg : pos    =     10.6 :
1.0
```

Yes, you read that right, Matt Damon is apparently one of the best predictors for positive sentiment in movie reviews. But despite this chuckle-worthy result

- accuracy is up almost 9%
- `pos` precision has increased over 10% with only 4% drop in recall
- `neg` recall has increased over 21% with just under 4% drop in precision

So it appears that the bigram hypothesis is correct, and **including significant bigrams can increase classifier effectiveness**. Note that it's *significant bigrams* that enhance effectiveness. I tried using nltk.util.bigrams to include all bigrams, and the results were only a few points above baseline. This points to the idea that including only significant features can improve accuracy compared to using all features. In a future article, I'll try trimming down the single word features to only include significant words.

BAYES BIGRAMS CLASSIFICATION COLLOCATION CORRELATION FEATURE EXTRACTION NLP NLTK PYTHON SENTIMENT STATISTICS STOPWORDS

PYTHON

TEXT CLASSIFICATION FOR SENTIMENT ANALYSIS – PRECISION AND RECALL

MAY 17, 2010 | 43 COMMENTS

Accuracy is not the only metric for evaluating the effectiveness of a classifier. Two other useful metrics are precision and recall. These two metrics can provide much greater insight into the performance characteristics of a binary classifier.

Classifier Precision

Precision measures the exactness of a classifier. A higher precision means less false positives, while a lower precision means more false positives. This is often at odds with recall, as an easy way to improve precision is to decrease recall.

Classifier Recall

Recall measures the completeness, or sensitivity, of a classifier. Higher recall means less false negatives, while lower recall means more false negatives. Improving recall can often decrease precision because it gets increasingly harder to be precise as the sample space increases.

F-measure Metric

Precision and recall can be combined to produce a single metric known as *F-measure*, which is the weighted harmonic mean of precision and recall. I find F-measure to be about as useful as accuracy. Or in other words, compared to precision & recall, F-measure is mostly useless, as you'll see below.

Measuring Precision and Recall of a Naive Bayes Classifier

The NLTK metrics module provides functions for calculating all three metrics mentioned above. But to do so, you need to build 2 sets for each classification label: a *reference set* of correct values, and a *test set* of observed values. Below is a modified version of the code from the previous article, where we trained a Naive Bayes Classifier. This time, instead of measuring accuracy, we'll collect reference values and observed values for each label (pos or neg), then use those sets to calculate the precision, recall, and F-measure of the naive bayes classifier.

The actual values collected are simply the index of each featureset using enumerate.

```
import collections
import nltk.metrics
from nltk.classify import NaiveBayesClassifier
from nltk.corpus import movie_reviews

def word_feats(words):
    return dict([(word, True) for word in words])

negids = movie_reviews.fileids('neg')
posids = movie_reviews.fileids('pos')

negfeats = [(word_feats(movie_reviews.words(fileids=[f])), 'ne
posfeats = [(word_feats(movie_reviews.words(fileids=[f])), 'po

negcutoff = len(negfeats)*3/4
poscutoff = len(posfeats)*3/4

trainfeats = negfeats[:negcutoff] + posfeats[:poscutoff]
testfeats = negfeats[negcutoff:] + posfeats[poscutoff:]
print 'train on %d instances, test on %d instances' % (len(tra

classifier = NaiveBayesClassifier.train(trainfeats)
refsets = collections.defaultdict(set)
testsets = collections.defaultdict(set)

for i, (feats, label) in enumerate(testfeats):
    refsets[label].add(i)
    observed = classifier.classify(feats)
    testsets[observed].add(i)

print 'pos precision:', nltk.metrics.precision(refsets['pos'],
print 'pos recall:', nltk.metrics.recall(refsets['pos'], tests
print 'pos F-measure:', nltk.metrics.f_measure(refsets['pos'],
print 'neg precision:', nltk.metrics.precision(refsets['neg'],
print 'neg recall:', nltk.metrics.recall(refsets['neg'], tests
print 'neg F-measure:', nltk.metrics.f_measure(refsets['neg'],
```

Precision and Recall for Positive and Negative Reviews

I found the results quite interesting:

```
pos precision: 0.651595744681
pos recall: 0.98
pos F-measure: 0.782747603834
neg precision: 0.959677419355
neg recall: 0.476
neg F-measure: 0.636363636364
```

So what does this mean?

1. Nearly every file that is pos is correctly identified as such, with 98% recall. This means very few *false negatives* in the pos class.
2. But, a file given a pos classification is only 65% likely to be correct. Not so good precision leads to **35% false positives** for the pos label.
3. Any file that is identified as neg is 96% likely to be correct (high precision). This means very few *false positives* for the neg class.
4. But many files that are neg are incorrectly classified. Low recall causes **52% false negatives** for the neg label.
5. **F-measure provides no useful information**. There's no insight to be gained from having it, and we wouldn't lose any knowledge if it was taken away.

Improving Results with Better Feature Selection

One possible explanation for the above results is that people use normally positives words in negative reviews, but the word is preceded by "not" (or some other negative word), such as "not great". And since the classifier uses the bag of words model, which assumes every word is independent, it cannot learn that "not great" is a negative. If this is the case, then these metrics should improve if we also train on multiple words, a topic I'll explore in a future article.

Another possibility is the abundance of naturally neutral words, the kind of words that are devoid of sentiment. But the classifier treats all words the same, and has to assign each word to either pos or neg. So maybe otherwise neutral or meaningless words are being placed in the pos class because the classifier doesn't know what else to do. If this is the case, then the metrics should improve if we eliminate the neutral or meaningless words from the featuresets, and only classify using *sentiment rich* words. This is usually done using the concept of information gain, aka mutual information, to improve feature selection, which I'll also explore in a future article.

If you have your own theories to explain the results, or ideas on how to improve precision and recall, please share in the comments.

SHARE THIS:

Twitter Reddit Facebook 5

BAYES CLASSIFICATION FEATURE EXTRACTION PERFORMANCE PRECISION PYTHON RECALL SENTIMENT

PYTHON

TEXT CLASSIFICATION FOR SENTIMENT ANALYSIS – NAIVE BAYES CLASSIFIER

MAY 10, 2010 | 140 COMMENTS

Sentiment analysis is becoming a popular area of research and social media analysis, especially around user reviews and tweets. It is a special case of text mining generally focused on identifying opinion polarity, and while it's often not very accurate, it can still be useful. For simplicity (and because the training data is easily accessible) I'll focus on 2 possible sentiment classifications: *positive* and *negative*.

NLTK Naive Bayes Classification

NLTK comes with all the pieces you need to get started on sentiment analysis: a movie reviews corpus with reviews categorized into *pos* and *neg* categories, and a number of trainable classifiers. We'll start with a simple NaiveBayesClassifier as a baseline, using boolean word feature extraction.

Bag of Words Feature Extraction

All of the NLTK classifiers work with featstructs, which can be simple dictionaries mapping a *feature name* to a *feature value*. For text, we'll use a simplified bag of words model where every word is feature name with a value of True. Here's the feature extraction method:

```
def word_feats(words):
        return dict([(word, True) for word in words])
```

Training Set vs Test Set and Accuracy

The movie reviews corpus has 1000 positive files and 1000 negative files. We'll use 3/4 of them as the training set, and the rest as the test set. This gives us 1500 training instances and 500 test instances. The classifier training method expects to be given a list of tokens in the form of [(feats, label)] where feats is a feature dictionary and label is the classification label. In our case, feats will be of the form {word: True} and label will be one of 'pos' or 'neg'. For accuracy evaluation, we can use nltk.classify.util.accuracy with the test set as the gold standard.

Training and Testing the Naive Bayes Classifier

Here's the complete python code for training and testing a Naive Bayes Classifier on the movie review corpus.

```
import nltk.classify.util
from nltk.classify import NaiveBayesClassifier
from nltk.corpus import movie_reviews

def word_feats(words):
    return dict([(word, True) for word in words])

negids = movie_reviews.fileids('neg')
posids = movie_reviews.fileids('pos')

negfeats = [(word_feats(movie_reviews.words(fileids=[f])), 'ne
posfeats = [(word_feats(movie_reviews.words(fileids=[f])), 'po

negcutoff = len(negfeats)*3/4
poscutoff = len(posfeats)*3/4

trainfeats = negfeats[:negcutoff] + posfeats[:poscutoff]
testfeats = negfeats[negcutoff:] + posfeats[poscutoff:]
print 'train on %d instances, test on %d instances' % (len(tra

classifier = NaiveBayesClassifier.train(trainfeats)
print 'accuracy:', nltk.classify.util.accuracy(classifier, tes
classifier.show_most_informative_features()
```

And the output is:

```
train on 1500 instances, test on 500 instances
accuracy: 0.728
Most Informative Features
         magnificent = True              pos : neg    =     15.0 :
1.0
         outstanding = True              pos : neg    =     13.6 :
1.0
           insulting = True              neg : pos    =     13.0 :
1.0
          vulnerable = True              pos : neg    =     12.3 :
1.0
           ludicrous = True              neg : pos    =     11.8 :
1.0
              avoids = True              pos : neg    =     11.7 :
1.0
         uninvolving = True              neg : pos    =     11.7 :
```

```
1.0
          astounding = True              pos : neg    =     10.3 :
1.0
         fascination = True              pos : neg    =     10.3 :
1.0
             idiotic = True              neg : pos    =      9.8 :
1.0
```

As you can see, the 10 most informative features are, for the most part, highly descriptive adjectives. The only 2 words that seem a bit odd are "vulnerable" and "avoids". Perhaps these words refer to important plot points or character development that signify a good movie. Whatever the case, with simple assumptions and very little code we're able to get almost 73% accuracy. This is somewhat near human accuracy, as apparently people agree on sentiment only around 80% of the time. Future articles in this series will cover precision & recall metrics, alternative classifiers, and techniques for improving accuracy.

BAYES CLASSIFICATION NLP NLTK PYTHON SENTIMENT STATISTICS