YouTube Video Recommendation Application

Shashank Kumar Tekriwal, Saurabh Gupta

Cluster Innovation Centre, University of Delhi

shashank.internet@gmail.com

saurabhgupta3012@gmail.com

***Abstract*— YouTube Video Recommender is a simple command line application that compares and rates YouTube videos in order to determine the best video among the search results. It is mainly targeted for educational videos on YouTube.**

I. Motivation and purpose

. YouTube has become one of the most popular places where students can learn for free. There is no limit to the extent you can learn here be it any topic. From primary grades to even Post Graduate level there are videos for everyone and some being developed by best institutions and professors on earth.

YouTube EDU (1) is YouTube's corpus of 700k+ high quality educational videos from partners like Khan Academy, Stanford and TED-Ed. YouTube created two programs to help schools and teachers utilize YouTube EDU most effectively: YouTube for Schools and YouTube for Teachers.

1. YouTube for Teachers provides tips & tricks for bringing YouTube into the classroom and organizes YouTube EDU videos to align with common core subjects.
2. YouTube for Schools allows schools to access all of the YouTube EDU content while limiting access to non-educational content.

A. *The Problem Statement*

Among such an extensive database of videos for just a single topic it gets somewhat difficult for a user to select a good one. It so happens that the student needs to open and view the videos to decide what is best suited to him and the one that gives a clear explanation of the topic. To quickly analyze the usefulness of a video one can view the comments on a video and get a general overview.

To solve this ambiguity and deciding whether a video tutorial is worth spending the time on, we have tried to make an application that analyzes the comments and other statistical data and give videos a general rating. Using this tool, a user can select the few good ones among the many searches provided by YouTube.

II. The Application

YouTube Video Recommender is a simple command line application that compares and rates YouTube videos in order to determine the best video among the search results. It is mainly targeted for Educational videos on YouTube.

The major objective of the project was to analyze YouTube comments and classify them using various machine learning techniques. Analyzing the comments of the video we could help the user to some extent determine the best video lecture featured on YouTube. Analysis of the comments was combined with other statistical data to help produce a better output. The other statistics of the video include:

- *View Count:* Number of times the video has been viewed
- *Like Count:* Number of likes
- *Dislike Count:* Number of dislikes
- *Total Number of Comments:* All comments
- *Number of positive comments:* all comments having positive sentiment
- *Number of negative comments:* all comments having negative sentiment

A. *Dependencies*

This command line application is tested and written completely in Python 2.7 32 bit version. It needs an active internet connection for downloading comments and following external modules that are not part of the Python core packages:

- *gdata 2.0.8:* for Google's YouTube API v2
- *pickle 3.4.2:* for serialization
- *pyenchant 1.6.6:* for spell checking
- *nltk 3.0:* for natural language processing
- *scikit 0.15:* for support vector machines

B. *Working*

The application simply asks the user to enter a search query. That search query is then used to search videos related to it and the top ten YouTube responses are fetched with their statistics and comments. Then the comments and other statistics are analysed and the fetched videos are racked accordingly.

III. Classification using SVM

The major objective was to fetch YouTube Video Comments and classify them among three classes – positive, negative and neutral.

A. *Support Vector Machines*

In machine learning, support vector machines (2) (SVMs, also support vector networks) are supervised learning models with associated learning algorithms that analyse data and recognize patterns, used for classification and regression analysis. Given a set of training examples, each marked as belonging to one of two categories, an SVM training algorithm builds a model that assigns new examples into one category or the other, making it a non-probabilistic binary linear classifier. An SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall on.

A support vector machine constructs a hyperplane (3) or set of hyperplanes in a high- or infinite-dimensional space, which can be used for classification, regression, or other tasks. Intuitively, a good separation is achieved by the hyperplane that has the largest distance to the nearest training data point of any class (so-called functional margin), since in general the larger the margin the lower the generalization error of the classifier.

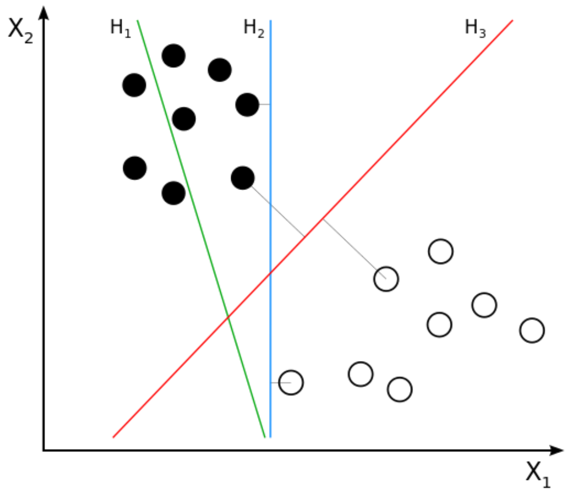

Figure 1: H1 does not separate the classes. H2 does, but only with a small margin. H3 separates them with the maximum margin.

B. Pre-processing Text Data

The raw data, a sequence of symbols cannot be fed directly to the algorithms themselves as most of them expect numerical feature vectors with a fixed size rather than the raw text documents with variable length.

To preprocess the data we used nltk – natural language toolkit to remove all that text from the text data that was not of use or that did not help in deciding the sentiment of the comment like nouns and determinants and verbs. We kept those parts of the comments that depicted sentiment – Adjectives, Adverbs, Gerunds, etc.

All punctuations were also removed on the assumption that the sentiment of the text is entirely based on the words.

Users on internet alter words to large extent. An example of which is writing ‘good’ as ‘goooood’, ‘very’ as ‘verrryy’. All such possible corrections were also made – like converting ‘goooood’ to ‘good’ and then the data was classified for better results.

C. The training data

To train the classifier we needed the appropriate training data. A classifier would return good results when the test data is similar to the training data. This meant that we had to classify a huge number of comments manually into the three classes for the purpose of the training of the training of the classifier.

For this purpose, we picked 200 YouTube Educational Videos and downloaded their comments. This data though quite small was too much for manual classification. Some videos even had above 10,000 comments. So, we decided to use a minimal algorithm to create the training data. We used the *polarity* measurement (4) method to perform sentiment analysis of the data.

This method is based on the database of keywords and each keyword has its own specific polarity. For example: ‘good’ may have a polarity of 0.6 and ‘amazing’ has a polarity of 0.8 even though both of these words show positive sentiment. Likewise, negative words too have polarities. For our purpose we took the priority of every positive keyword to be +1 and every negative keyword -1. We had two text files – one with all positive words and the other with all negative words. If a comment had more positive words in it, it was given a positive class. Comments with 0 polarity were given *neutral* class and similarly, if the overall polarity was less than 0 it was assigned *negative* class.

We then reviewed the 200 files manually and changed conflicting results – i.e. that some comments being negative but marked as positive or vice versa. Thus we now had a proper training data to train the classifier.

D. Naïve Bayes Classifier

Multinomial Naïve Bayes Classifier (5) is the most widely used classifier for text document classification. Naive Bayes methods (6) are a set of supervised learning algorithms based on applying Bayes’ theorem with the “naive” assumption of independence between every pair of features.

Given a class variable y and a dependent feature vector x_1 through x_n, Bayes’ theorem states the following relationship:

$$P(y \mid x_1, \dots, x_n) = \frac{P(y) P(x_1, \dots x_n \mid y)}{P(x_1, \dots, x_n)}$$

Using the naive independence assumption that

$$P(x_i | y, x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n) = P(x_i | y),$$

for all i, this relationship is simplified to

$$P(y \mid x_1, \dots, x_n) = \frac{P(y) \prod_{i=1}^{n} P(x_i \mid y)}{P(x_1, \dots, x_n)}$$

Since $P(x_1, \dots, x_n)$ is constant given the input, we can use the following classification rule:

$$P(y \mid x_1, \dots, x_n) \propto P(y) \prod_{i=1}^{n} P(x_i \mid y)$$

$$\Downarrow$$

$$\hat{y} = \arg\max_y P(y) \prod_{i=1}^{n} P(x_i \mid y),$$

But raw text cannot be given to a classifier. It needs training samples in the form of numerical feature vectors and corresponding target class. To extract numerical features from our text data we have used the *Bag of Words* representation. Among many techniques this is most widely used for textual feature extraction.

E. Bag of Words representation

Text Analysis is a major application field for machine learning algorithms. However the raw data, a sequence of symbols cannot be fed directly to the algorithms themselves as most of them expect numerical feature vectors with a fixed size rather than the raw text documents with variable length.

In order to address this, scikit-learn (the python library through which we are implementing svm) (7) provides utilities for the most common ways to extract numerical features from text content, namely:

- **tokenizing** strings and giving an integer id for each possible token, for instance by using white-spaces and punctuation as token separators.
- **counting** the occurrences of tokens in each document.
- **normalizing** and weighting with diminishing importance tokens that occur in the majority of samples / documents.

In this scheme, features and samples are defined as follows:

- each *individual token occurrence frequency* (normalized or not) is treated as a **feature**.
- the vector of all the token frequencies for a given *document* is considered a *multivariate sample.*

We call **vectorization** the general process of turning a collection of text documents into numerical feature vectors. This specific strategy (tokenization, counting and normalization) is called the **Bag of Words** or “Bag of n-grams” representation.

```
>>> corpus = [
...     'This is the first document.',
...     'This is the second second document.',
...     'And the third one.',
...     'Is this the first document?',
... ]
>>> X = vectorizer.fit_transform(corpus)
>>> X
<4x9 sparse matrix of type '<... 'numpy.int64'>'
   with 19 stored elements in Compressed Sparse ... format>
```

```
>>> analyze = vectorizer.build_analyzer()
>>> analyze("This is a text document to analyze.") == (
...     ['this', 'is', 'text', 'document', 'to', 'analyze'])
True
```

Each term found by the analyzer during the fit is assigned a unique integer index corresponding to a column in the resulting matrix. This interpretation of the columns can be retrieved as follows:

```
>>> vectorizer.get_feature_names() == (
...     ['and', 'document', 'first', 'is', 'one',
...      'second', 'the', 'third', 'this'])
True

>>> X.toarray()
array([[0, 1, 1, 1, 0, 0, 1, 0, 1],
       [0, 1, 0, 1, 0, 2, 1, 0, 1],
       [1, 0, 0, 0, 1, 0, 1, 1, 0],
       [0, 1, 1, 1, 0, 0, 1, 0, 1]]...)
```

The *vectorizer_transform()* method converts then converts the text passed to it into a set of feature vectors.

```
>>> vectorizer.transform(['Something completely new.']).toarray()
...
array([[0, 0, 0, 0, 0, 0, 0, 0, 0]]...)
```

We have used a bi-gram vectorizer for our purpose. Bi-gram vectorizer works by exracting out single as well as pairs of words.

```
>>> bigram_vectorizer = CountVectorizer(ngram_range=(1, 2),
...                                     token_pattern=r'\b\w+\b', min_df=1)
>>> analyze = bigram_vectorizer.build_analyzer()
>>> analyze('Bi-grams are cool!') == (
...     ['bi', 'grams', 'are', 'cool', 'bi grams', 'grams are', 'are cool'])
True
```

After that we have converted our data into vectors we can pass it to the Naïve Bayes Classifier for its training.

```
>>> import numpy as np
>>> X = np.random.randint(5, size=(6, 100))
>>> y = np.array([1, 2, 3, 4, 5, 6])
>>> from sklearn.naive_bayes import MultinomialNB
>>> clf = MultinomialNB()
>>> clf.fit(X, y)
MultinomialNB(alpha=1.0, class_prior=None, fit_prior=True)
>>> print(clf.predict(X[2]))
[3]
```

IV. SERIALIZATION

After that we have trained our classifier we serialized it on to the disk so that every time we use our application it doesn’t have to train the classifier first and then produce the output. Serialization in python can be easily done using the *pickle* module.

Whenever, the application is used, we simply read the classifier data from the disk, saving time, and predict the output of the new data.

V. YOUTUBE STATISTICS

Using the *gdata* module of python and YouTube API v2[1] (8) (9), we are downloading required video comments. Using YouTube API v3 (10) we are fetching other statistics like *view count, like count, dislike count,* and *total comments.*

[1] Fetching comments is right now not possible with latest version of Youtube API – 3.0

After fetching YouTube statistics for the required videos and analysing comments it gets easier to compare videos. We now have a list of features regarding a particular video that include:

- *Number of positive, negative and neutral comments*
- *Number of likes, dislikes*
- *View Count of a video*

Assigning proper weights to each of these attributes of a video one can make their own simple mathematical formula to rank Youtube videos. Some may give more weight to *comments* and less to the *number of likes* or others may do vice versa. This completely depends upon the views of the end user.

We simply first segregated videos on the basis of their *view count* and then ranked them giving equal weights to comments and likes.

VI. Conclusion

This app can be a good tool to classify videos on when the user doesn't have all the time to search for the good ones. During the course of this project we leared a lot about implementing APIs and handling Json data. We learned about various Machine Learning Algorithms, not only the Naïve Bayes Classifier – the one that we implemented here, but also others like SVM with different *kernel functions* and their applications. We got to know how data modeling is implemented through machine learning, classification techniques and clustering models, the overview of the mathematics behind it and its wide range of applications.

Acknowledgment

I express my warm thanks to Mr. Alok Nikhil Jha for assigning me this project. I am thankful for his aspiring guidance, invaluably constructive criticism and friendy advice during the project work.

References

1. About YouTube Edu. *support.google.com.* [Online] https://support.google.com/youtube/answer/2685809?hl=en.
2. Support Vector Machine. *Wikipedia.* [Online] http://en.wikipedia.org/wiki/Support_vector_machine.
3. SVM Kernels. *scikit-learn.* [Online] http://scikit-learn.org/stable/modules/svm.html#svm-kernels.
4. Sentiment Analysis. *Wikipedia.* [Online] http://en.wikipedia.org/wiki/Sentiment_analysis.
5. MultinomialNB. *scikit-learn.* [Online] http://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.MultinomialNB.html#sklearn.naive_bayes.MultinomialNB.
6. Naive Bayes. *Scikit learn.* [Online] http://scikit-learn.org/stable/modules/naive_bayes.html.
7. SK Learn. *scikit-learn.* [Online] http://scikit-learn.org/stable/index.html.
8. YouTube API v2.0. *developers.google.* [Online] https://developers.google.com/youtube/2.0/developers_guide_protocol.
9. Reference Guide YouTube API v2.0. *developers.google.* [Online] https://developers.google.com/youtube/2.0/reference.
10. Youtube API v3.0. *developers.google.* [Online] https://developers.google.com/youtube/v3/docs/videos.