Natural Language Processing

The ability of a computer program to understand, interpret and manipulate human language is what we call Natural Language Processing.

Where is it used?

→ To perform tasks like –

- Translation
- Summarization
- Named entity recognition (NER)
- Relationship extraction
- Speech Recognition
- Topic Segmentation

NLP needs a NLTK or a Natural Language Toolkit library which should be installed.

→ pip : pip install nltk

→ conda : conda install scikit-learn
conda install nltk

TOKENIZATION

- **Tokenisation refers to breaking a sentence into words.**
- Tokens of any number of consecutive written words are known as Ngram, where - N = 1,2,3,4...
- Tokens of 2 consecutive written words are known as Bigram
- Tokens of 3 consecutive written words are known as Trigram
- To do tokenisation, first we should download and use 'punkt' package that is useful to divide text into a group of sentences

In [2]:
```
import nltk
nltk.download('punkt')
```

```
[nltk_data] Downloading package punkt to /home/shreyanbr/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt.zip.
```

Out[2]: True

In [3]:
```
#take a string and divide it into tokens
string = 'Hold fast to dreams, for if dreams die, life is a broken-winged bi
tokens = nltk.word_tokenize(string)
tokens
```

Out[3]:
```
['Hold',
 'fast',
 'to',
 'dreams',
 ',',
 'for',
 'if',
 'dreams',
 'die',
 ',',
 'life',
 'is',
 'a',
 'broken-winged',
 'bird',
 'that',
 'can',
 'not',
 'fly']
```

In [4]:
```
#divide it into bigrams
bigram = nltk.bigrams(tokens)
str_bigrams = list(bigram)
str_bigrams
```

Out[4]:
```
[('Hold', 'fast'),
 ('fast', 'to'),
 ('to', 'dreams'),
 ('dreams', ','),
 (',', 'for'),
 ('for', 'if'),
 ('if', 'dreams'),
 ('dreams', 'die'),
 ('die', ','),
 (',', 'life'),
 ('life', 'is'),
 ('is', 'a'),
 ('a', 'broken-winged'),
 ('broken-winged', 'bird'),
 ('bird', 'that'),
 ('that', 'can'),
 ('can', 'not'),
 ('not', 'fly')]
```

In [5]:
```
#divide it into trigrams
trigram = nltk.trigrams(tokens)
```

```
str_trigrams = list(trigram)
str_trigrams
```

Out[5]:
```
[('Hold', 'fast', 'to'),
 ('fast', 'to', 'dreams'),
 ('to', 'dreams', ','),
 ('dreams', ',', 'for'),
 (',', 'for', 'if'),
 ('for', 'if', 'dreams'),
 ('if', 'dreams', 'die'),
 ('dreams', 'die', ','),
 ('die', ',', 'life'),
 (',', 'life', 'is'),
 ('life', 'is', 'a'),
 ('is', 'a', 'broken-winged'),
 ('a', 'broken-winged', 'bird'),
 ('broken-winged', 'bird', 'that'),
 ('bird', 'that', 'can'),
 ('that', 'can', 'not'),
 ('can', 'not', 'fly')]
```

In [7]:
```
#divide it into n-grams where n = 1,2,3,4....
str_ngrams = list(nltk.ngrams(tokens,4))
str_ngrams
```

Out[7]:
```
[('Hold', 'fast', 'to', 'dreams'),
 ('fast', 'to', 'dreams', ','),
 ('to', 'dreams', ',', 'for'),
 ('dreams', ',', 'for', 'if'),
 (',', 'for', 'if', 'dreams'),
 ('for', 'if', 'dreams', 'die'),
 ('if', 'dreams', 'die', ','),
 ('dreams', 'die', ',', 'life'),
 ('die', ',', 'life', 'is'),
 (',', 'life', 'is', 'a'),
 ('life', 'is', 'a', 'broken-winged'),
 ('is', 'a', 'broken-winged', 'bird'),
 ('a', 'broken-winged', 'bird', 'that'),
 ('broken-winged', 'bird', 'that', 'can'),
 ('bird', 'that', 'can', 'not'),
 ('that', 'can', 'not', 'fly')]
```

Stemming

- Normalising the words into their base form or root form is called 'stemming'
- eg: affection, affects, affected, affectation have the same root words as 'affect'. So affect is the stem here.
- There are various types of stemmers eg - Porter Stemmer and Lancaster Stemmer

Let's see if we get what we are thinking through this stemming process.

In [1]:
```
#using Porter Stemmer here
from nltk.stem import PorterStemmer
pst = PorterStemmer()
```

In [2]:
```
#find root word for 'having'
pst.stem('having')
```

Out[2]: 'have'

In [3]:
```
#finding the stem for a list of words
words = ['give', 'giving', 'given', 'gave']
for w in words:
    print(w, ':', pst.stem(w))
```

```
give : give
giving : give
given : given
gave : gave
```

This should've showed 'give' as a root word for all but didn't. Hmmm... let's use a Lancaster Stemmer to check another way if it shows the root word as we are thinking it to be.

In [4]:
```
#using LancasterStemmer here
from nltk.stem import LancasterStemmer
lst = LancasterStemmer()
```

In [5]:
```
for w in words:
    print(w,':', lst.stem(w))
```

```
give : giv
giving : giv
given : giv
gave : gav
```

This stemmer is simply chopping off words and giving 'giv' or 'gav' as the stem word which are also not the ones we are looking for.

We have to use a different approach now...

LEMMATIZATION

- Stemming cuts the word to give the root word but it does not do so in many cases. This is where lemmatization comes in.
- **'Lemmatization' groups together different inflected forms of the same word called 'lemma'.**
- eg: 'gone, going, went' can be lemmatized into the same word - 'go'. Here 'go' is the lemma.
- To do lemmatization, we need the total English dictionary to be available.

In [1]:
```
#download the dictionary 'wordnet'
import nltk
nltk.download('wordnet')
```

```
[nltk_data] Downloading package wordnet to
[nltk_data]     /home/shreyanbr/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
```

Out[1]: True

In [2]:
```
from nltk.stem import wordnet
from nltk.stem import WordNetLemmatizer
wnl = WordNetLemmatizer()
```

In [3]:
```
#lemmatize the word
wnl.lemmatize('corpora')
```

Out[3]: 'corpus'

In [4]:
```
#lemmatize a list of words
words = ['give', 'giving', 'given', 'gave']
for w in words:
    print(w, ':', wnl.lemmatize(w))
```

```
give : give
giving : giving
given : given
gave : gave
```

Hmm... it didn't work as expected again ! It is considering all the above words as unrelated and hence giving same word as root. But if it knows that all the above words are different verb forms of the root 'give', it would have been the correct output. Let's add an attribute here for this -

In [5]:
```
for w in words:
    print(w, ':', wnl.lemmatize(w, pos='v'))
#here pos(parts of speech) is being specified as a 'verb'. Hence the lemmati
```

```
give : give
giving : give
given : give
gave : give
```

Viola !

Stop Words

Let me give you an example first - " I am a good person"

- Here 'I', 'am', 'a' are not significant. Only 'good' and 'person' are.
- **These non-significant words are called stop words.**
- These stop words are generally removed before doing any kind of analysis.
- NLTK has its own list of stop words.

In [1]:
```
import nltk
nltk.download('stopwords')
```

```
[nltk_data] Downloading package stopwords to
[nltk_data]     /home/shreyanbr/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

Out[1]:
```
True
```

In [2]:
```
from nltk.corpus import stopwords
stopwords.words('english')[:20] #showing only first 20
```

Out[2]:
```
['i',
 'me',
 'my',
 'myself',
 'we',
 'our',
 'ours',
 'ourselves',
 'you',
 "you're",
 "you've",
 "you'll",
 "you'd",
 'your',
 'yours',
 'yourself',
 'yourselves',
 'he',
 'him',
 'his']
```

In [3]:
```
len(stopwords.words('english')) #out of 179 stop words
```

Out[3]:
```
179
```

In [4]:
```
test = "Hey! This is a test sentence"
```

Note that the test sentence has 2 punctuation marks, (!) and (.). We need to get rid of all punctuation to find the stop words. We'll use the punctuation library of the string module.

In [5]:
```
import string
#str = []
#for char in test:
#    if char not in string.punctuation:
#        str.append(char)

#let us use a list comprehension method instead of using this above. Better
str = [char for char in test if char not in string.punctuation]
str
```

Out[5]:
```
['H',
 'e',
 'y',
 ' ',
 'T',
 'h',
 'i',
 's',
 ' ',
 'i',
 's',
 ' ',
 'a',
 ' ',
 't',
 'e',
 's',
 't',
 ' ',
 's',
 'e',
 'n',
 't',
 'e',
 'n',
 'c',
 'e']
```

In [6]:
```
str = ''.join(str)
str
```

Out[6]:
```
'Hey This is a test sentence'
```

In [7]:
```
#let us split this string into pieces where a space is found
x = str.split()
x
```

Out[7]:
```
['Hey', 'This', 'is', 'a', 'test', 'sentence']
```

In [8]:
```
cleaned = [word for word in x if word.lower() not in stopwords.words('englis
cleaned
```

Out[8]:
```
['Hey', 'test', 'sentence']
```

Parts Of Speech

In [1]:
```
import nltk
```

In [2]:
```
#break the sentence into words or tokenize the sentence
sentence = "The cat caught the mouse"
words = nltk.word_tokenize(sentence)
words
```

Out[2]:
```
['The', 'cat', 'caught', 'the', 'mouse']
```

In [3]:
```
#download averaged_perceptron_tagger
nltk.download('averaged_perceptron_tagger')
```

```
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     /home/shreyanbr/nltk_data...
[nltk_data]   Package averaged_perceptron_tagger is already up-to-
[nltk_data]       date!
```

Out[3]:
```
True
```

In [4]:
```
#find the POS for given sentence
nltk.pos_tag(words)
```

Out[4]:
```
[('The', 'DT'),
 ('cat', 'NN'),
 ('caught', 'VBD'),
 ('the', 'DT'),
 ('mouse', 'NN')]
```

There are various abbreviations/tags used in NLP to represent the parts of speech.

Tag	Description
CC	Coordinating conjunction
CD	Cardinal number
DT	Determiner
EX	Existential there
FW	Foreign word
IN	Preposition or subordinating conjunction
JJ	Adjective
JJR	Adjective, comparative
JJS	Adjective, superlative
LS	List item marker
MD	Modal
NN	Noun, singular or mass
NNS	Noun, plural
NNP	Proper noun, singular
NNPS	Proper noun, plural
PDT	Predeterminer
POS	Possessive ending
PRP	Personal pronoun

Tag	Description
PRP$	Possessive pronoun
RB	Adverb
RBR	Adverb, comparative
RBS	Adverb, superlative
RP	Particle
SYM	Symbol
TO	to
UH	Interjection
VB	Verb, base form
VBD	Verb, past tense
VBG	Verb, gerund or present participle
VBN	Verb, past participle
VBP	Verb, non3rd person singular present
VBZ	Verb, 3rd person singular present
WDT	Whdeterminer
WP	Whpronoun
WP$	Possessive whpronoun
WRB	Whadverb

Feature Extraction

- NLP is used on textual data. To understand the textual data, first of all, it should be converted into numeric data.
- This is **Feature Extraction**. A technique to convert the content into numerical vectors and understand the importance of words or tokens in the given text.
- **Vectorizer** is a class in NLP that is useful to perform feature extraction.
- There are **2 types of vectorizers**. They are -
  - Count Vectorizer
  - Tfidf vectorizer

Count Vectorizer

- Separates each string into tokens and then counts the number of times each word occurs in the sentence.
- This approach is also known as 'bag of words' approach.

In [1]:
```
sentences = ['Hey hey hey lets go get lunch today', 'Did you go home?', 'Hey
print(sentences)
```

```
['Hey hey hey lets go get lunch today', 'Did you go home?', 'Hey!!! I need a
favour']
```

In [2]:
```
#create an object to COuntVectorizer class
from sklearn.feature_extraction.text import CountVectorizer
cv = CountVectorizer()
```

In [3]:
```
s = cv.fit_transform(sentences)
print(s)
```

```
  (0, 4)        3
  (0, 6)        1
  (0, 3)        1
  (0, 2)        1
  (0, 7)        1
  (0, 9)        1
  (1, 3)        1
  (1, 0)        1
  (1, 10)       1
  (1, 5)        1
  (2, 4)        1
  (2, 8)        1
  (2, 1)        1
```

[(x, y) n], where in xth sentence, x = 0,1,2,3... yth word is being repeated for n times. Let me make this more clearer on how it produced this output -

After processing with CountVectorizer, the feature names (unique words) extracted from these sentences, excluding common stop words and punctuation, are:

In [4]:
```
print("Feature Names:", cv.get_feature_names_out())
```

```
Feature Names: ['did' 'favour' 'get' 'go' 'hey' 'home' 'lets' 'lunch' 'need'
'today'
 'you']
```

The extracted feature names are organised alphabetically automatically. The sparse matrix representation of the sentences, where each row corresponds to a sentence and each column to a feature (word) with its count in the sentence, is:

In [5]:
```
print("Sparse Matrix:\n", s.toarray())
```

```
Sparse Matrix:
 [[0 0 1 1 3 0 1 1 0 1 0]
 [1 0 0 1 0 1 0 0 0 0 1]
 [0 1 0 0 1 0 0 0 1 0 0]]
```

Considering the first row in this matrix, i.e. for the first sentence - did, favour,home, need, you = 0, which indicates it does not occur in the first sentence. get, go, lets, lunch, today = 1 and hey=3.

Tfidf Vectorizer

- Alternative to Count Vectorizer
- Creates a table from our text or sentences
- Does not count how many times a word is repeated.
- It calculates term-frequency (TF) inverse document frequency (IDF) value for each word. This TF-IDF is the product of 2 weights : the term frequency and the inverse document frequency.

TF or Term Frequency represents how often a word occurs **in a single sentence**.

- TF value increases when we have several occurences of the same word in the same sentence.
- Eg - if a word is repeated many times in a sentence and not found in other sentences then that word is more meaningful and important. Such a word will be given more importance. TF-IDF will allocate a high score for that word.

IDF or Inverse Document Frequency is another weight representing how common a word is **across many sentences**.

- If a word is used in many sentences, then its TF-IDF will decrease.
- When several sentences are there, some common words may be repeated in each sentences. Such common words should not be given importance. TF-IDF will allocate low score for such words.

In [6]:
```
sentences = ['Hey lets get lunch', 'Hey!!! i need a favour']
print(sentences)
```

```
['Hey lets get lunch', 'Hey!!! i need a favour']
```

In [7]:
```
#lets create an object to the TfidfVectorizer class
from sklearn.feature_extraction.text import TfidfVectorizer
tv = TfidfVectorizer()
```

In [8]:
```
s = tv.fit_transform(sentences)
print(s)
```

```
  (0, 4)        0.534046329052269
  (0, 1)        0.534046329052269
  (0, 3)        0.534046329052269
  (0, 2)        0.37997836159100784
  (1, 0)        0.6316672017376245
  (1, 5)        0.6316672017376245
  (1, 2)        0.4494364165239821
```

← note these values here

In [9]:
```
# Printing the feature (word) names and the sparse matrix in a dense form
print("Feature Names:", tv.get_feature_names_out())
```

```
Feature Names: ['favour' 'get' 'hey' 'lets' 'lunch' 'need']
```

In [10]:
```
print("TF-IDF Matrix:\n", s.toarray())
```

```
TF-IDF Matrix:
[[0.         0.53404633 0.37997836 0.53404633 0.53404633 0.        ]
 [0.6316672  0.         0.44943642 0.         0.         0.6316672 ]]
```

In this matrix:

- The first row represents the TF-IDF values for each word in the first sentence. Words not present in the sentence have a TF-IDF score of 0.
- The second row represents the TF-IDF values for the second sentence.

So, count vectorizer counts how many times a word repeats and stores those figures in a table. This is called **bag of words.**

But this doesn't work for multiple sentences. This is where **Tf-Idf** comes in.

As discussed, Tf-Idf vectorizer works by the term frequency of the word by idf.

$$Tf = \frac{\text{No. of occurences of a word}}{\text{Total words in sentence}}$$

$$Idf = \log\left(\frac{\text{Total no. of sentences}}{\text{No. of sentences containing the word}}\right)$$

Or,

$$Idf = \log\left(\frac{\text{Total no. of documents}}{\text{No. of documents with the term in them}}\right)$$

Sentiment Analysis :

Sentiment Analysis is a subfield of natural language processing (NLP) that focuses on identifying and categorizing opinions expressed in text, especially to determine whether the writer's attitude towards a particular topic, product, or service is positive, negative, or neutral.

→ Lets do sentiment analysis using Natural Language Processing and classify the movie reviews into good or bad.

In [1]:
```
#download stop words list and import it
import nltk
nltk.download('stopwords')
from nltk.corpus import stopwords
```

```
[nltk_data] Downloading package stopwords to
[nltk_data]     /home/shreyanbr/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

In [2]:
```
#load data files from txt_sentoken folder
from sklearn.datasets import load_files
movie_data = load_files("review_polarity/txt_sentoken/")
```

In [3]:
```
#take input data 'x' and target data 'y'
x, y = movie_data.data, movie_data.target
print(len(x), len(y))
```

```
2000 2000
```

In [4]:
```
#know the target label
movie_data.target[0]
#0- negative, 1- positive
```

Out[4]:
```
0
```

Each review contains unnecessary spaces, special characters, punctuation marks, single characters etc. that are not useful for our analysis. These spaces must be removed by applying regular expressions or **regex**.

Use the 're' module for using regular expressions

In [5]:
```
import re
#preprocessing of data typ
lst = []
```

In [6]:
```
#use word net lemmatizer for getting the root words
from nltk.stem import WordNetLemmatizer
stemmer = WordNetLemmatizer()
```

In [7]:
```
#take sentences from the input data - x
for sen in range(0, len(x)):
    #remove all the special characters by a single space
    # \W - Matches any non-alphanumeric character
    txt = re.sub(r'\W', ' ', str(x[sen]))

    #remove all single characters
    # \s - any whitespace charcaters
    txt = re.sub(r'\s+[a-zA-z]\s+',' ', txt)

    txt = re.sub(r'^[a-zA-z]\s+',' ', txt)

    txt = re.sub(r'\s+',' ', txt)

    #converting to lower case
    txt = txt.lower()

    #Lemmatization
    txt = txt.split()
    txt = [stemmer.lemmatize(word) for word in txt]
```

```
    txt = ' '.join(txt)

    #add the final text to the list
    lst.append(txt)
```

The cleaned data is now available in the list 'lst'. This data now can be converted into numeric and score can be allocated to each word depending on its frequency.

In [8]:
```
len(lst)
```

Out[8]:
```
2000
```

In [9]:
```
#convert text to numbers using TfidVectorizer

from sklearn.feature_extraction.text import TfidfVectorizer
converter = TfidfVectorizer(stop_words=stopwords.words('english'))
x = converter.fit_transform(lst).toarray()
```

In [10]:
```
#split the data as training and testing sets
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(x,y, test_size=0.2, rand
```

We can now use any classifier model like Logistic Regression, Decision Tree, RandomForest or SVM to understand the review text and classify it into 0 or 1.

Let us use **RandomForest** classifier model with **200 trees**.

In [11]:
```
from sklearn.ensemble import RandomForestClassifier
classifier = RandomForestClassifier(n_estimators=200, random_state=0)
classifier.fit(x_train, y_train)
```

Out[11]:
```
RandomForestClassifier(n_estimators=200, random_state=0)
```

In [12]:
```
#find accuracy score
classifier.score(x_test, y_test)
```

Out[12]:
```
0.795
```

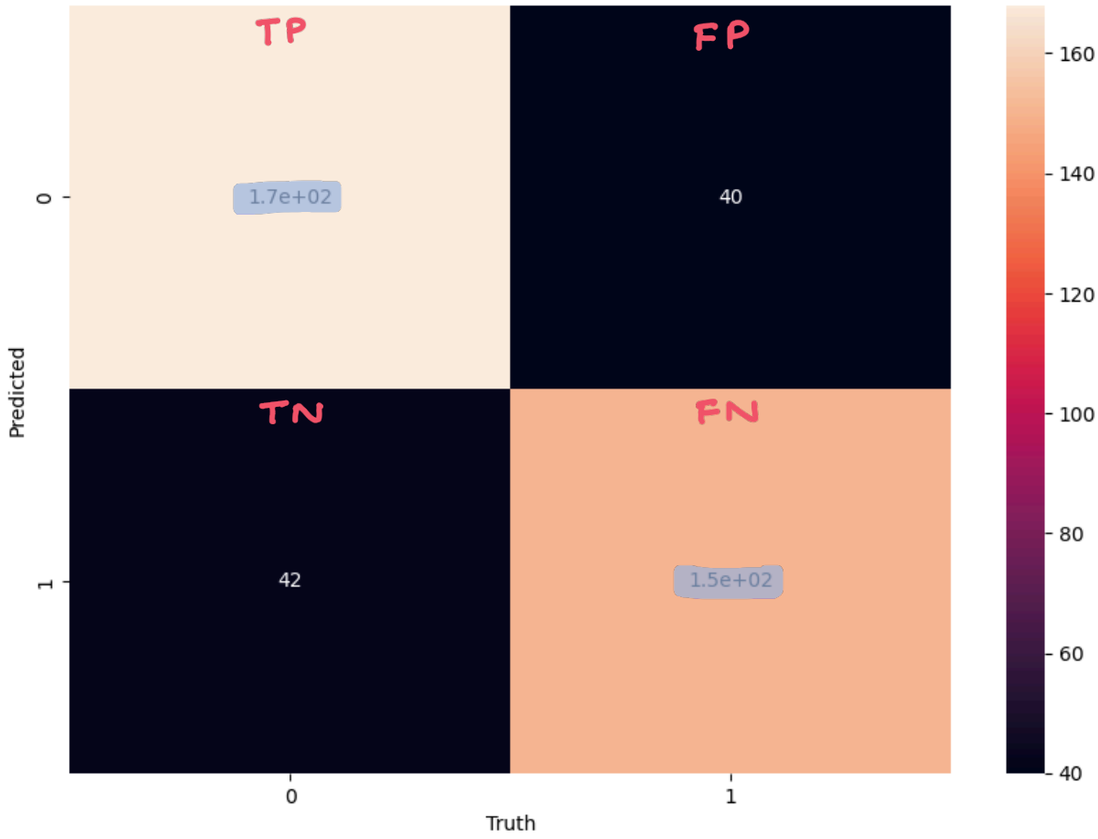
In [13]:
```
#Create confusion matrix to know where our model failed
y_predicted = classifier.predict(x_test)
```

In [14]:
```
#here y_test is truth and y_predicted is predicted
from sklearn.metrics import confusion_matrix
conmat = confusion_matrix(y_test, y_predicted)
conmat
```

Out[14]:
```
array([[168,  40],
       [ 42, 150]])
```

In [15]:
```
#the above matrix can be shown as plot in seaborn
import seaborn as sns
import matplotlib.pyplot as plt
plt.figure(figsize = (10,7))
sns.heatmap(conmat, annot=True)
plt.xlabel('Truth')
plt.ylabel('Predicted')
```

Out[15]:
```
Text(95.72222222222221, 0.5, 'Predicted')
```



In [16]:
```
#make predictions for our own review
review1 = "It was so bad" ← negative/bad
review2 = "I really enjoyed the climax scene of the movie" ← good review.
```

In [17]:
```
vx = converter.transform([review1,review2]).toarray()
classifier.predict(vx)
```

Out[17]: `array([0, 1])`

The output shows that the first review is bad[0] and the 2nd review is a good review [1].

and to all who are just curious how the 'lst' looks like after cleaning —

[18]: ['arnold schwarzenegger ha been an icon for action enthusiast since the late 80 but lately his film have been very sloppy and the one liner are getting worse nit hard seeing arnold a mr freeze in batman and robin especially when he say ton of ice joke but hey he got 15 million what it matter to him nonce again arnold ha signed to do another expensive blockbuster that can compare with the like of the terminator series true lie and even eraser nin this so called dark thriller the devil gabriel byrne ha come upon earth to impregnate woman robin tunney which happens every 1000 year and basically destroy the world but apparently god ha chosen one man and that one man is jericho cane arnold himself nwith the help of trusty sidekick kevin pollack they will stop at nothing to let the devil take over the world nparts of this are actually so absurd that they would fit right in with dogma nyes the film is that weak but it better than the other blockbuster right now sleepy hollow but it make the world is not enough look like 4 star film nanyway this definitely doesn seem like an arnold movie nit just wasn the type of film you can see him doing nsure he gave u few chuckle with his well known one liner but he seemed confused a to where his character and the film wa going nit understandable especially when the ending had to be changed according to some source naside form that he still walked through it much like he ha in the past few film ni sorry to say this arnold but maybe these are the end of your action day nspeaking of action where wa it in this film nthere wa hardly any explosion or fight nthe devil made few place explode but arnold wasn kicking some devil butt nthe ending wa changed to make it more spiritual which undoubtedly ruined the film ni wa at least hoping for cool ending if nothing else occurred but once again wa let down ni also don know why the film took so long and cost so much nthere wa really no super affect at all unless you consider an invisible devil who wa in it for 5 minute top worth the overpriced budget nthe budget should have gone into better script where at least audience could be somewhat entertained instead of facing boredom nit pitiful to see how script like these get bought and made into movie ndo they even read these thing anymore nit sure doesn seem like it nthankfully gabriel performance gave some light to this poor film nwhen he walk down the street searching for robin tunney you can help but feel that he looked like devil nthe guy is creepy looking anyway nwhen it all over you re just glad it the end of the movie ndon bother to see this if you re expecting solid action flick because it neither solid nor doe it have action nit just another movie that we are suckered in to seeing due to strategic marketing campaign nsave your money and see the world is not enough for an entertaining experience',
'good film are hard to find these day ngreat film are beyond rare nproof of life russell crowe one two punch of deft kidnap and rescue thriller is one of those rare gem na taut drama laced with strong and subtle acting an intelligent script and masterful directing together it delivers something virtually unheard of in the film industry these day genuine motivation in story that ring true nconsider the strange coincidence of russell crowe character in proof of life making the move on distraught wife played by meg ryan character in the film all while the real russell crowe wa hitching up with married woman meg ryan in the outside world ni haven seen t his much chemistry between actor since mcqueen and mcgraw teamed up in peckinpah masterpiece the getaway nbut enough with the gossip let get to the review nthe film revolves around the kidnapping of peter bowman david n

its huge!

—Shreyan.