Multi-Class Text Classification with Scikit-Learn using TF-IDF model

Rohit Batra Follow 12 min read · May 23, 2022

Problem Formulation

In order to design a question set for the teachers in the future, we need to identify a particular question from the pool of questions belonging to a particular subject. These questions are gathered and scrapped through multiple channels and online open sources. Hence, need a classifier model to analyze the text in a question for identifying which subject it belongs to.

Assignment Understanding

1. The column subject is equivalent to the subject names provided under the author table. 'Subject' should be the target variable here with 4 labels/categories: Maths, Biology, Chemistry, and Physics. This subject would be our "Actual Subject" and not the predicted one.
2. There are one too many relationships between the subject and the chapters i.e 1 subject can have many chapters. That is what is included in the chapter column.
3. We need to predict the highest probability of the subject label present under a new column "Predicted Subject" basis on the words available in the question in form of the text.
4. For this project, we need to focus only on two columns — "Subject" and "Question" to train the model.

Objective

Given a new question comes in, we want to assign it to one of the 4 categories of subjects (Maths, Biology, Chemistry, and Physics) i.e the classifier makes the assumption that each new question is assigned to one and only one subject label.

Data Exploration

In [189…

```
# Importing Libraries
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

%matplotlib inline
plt.style.use('seaborn-dark-palette')
from scipy import stats

import datetime as dt

import plotly
import plotly.express as px
```

In [190…

```
import warnings
warnings.filterwarnings("ignore")

pd.set_option('display.max_columns', None)
pd.set_option('display.max_rows', None)
```

In [191…

```
# read the training dataset

sub = pd.read_csv("subjects+classifier_dataset.csv")
sub.head()
```

Out[191…

	id	author	chapter	class_name	creation_time_stamp	edit_time_stamp	folder	question_text	subject
0	11228	ncert12maths	Relations and Functions	12	16-08-2021 4.26	16-08-2021 4.26	NCERT Exemplar	<p>Let <math xmlns="http://www.w3.org/1998/Math/MathML"><mrow><mi mathvariant="bold">N</mi>…	NaN
1	11229	ncert12maths	Relations and Functions	12	16-08-2021 4.26	16-08-2021 4.26	NCERT Exemplar	<p>Set <math xmlns="http://www.w3.org/1998/Math/MathML"><mrow><mi mathvariant="normal">A</m…	NaN
2	11230	ncert12maths	Relations and Functions	12	16-08-2021 4.26	16-08-2021 4.26	NCERT Exemplar	<p>Let <math xmlns="http://www.w3.org/1998/Math/MathML"><mi>f</mi> <mo>:</mo> <mr…	NaN
3	11231	ncert12maths	Relations and Functions	12	16-08-2021 4.26	16-08-2021 4.26	NCERT Exemplar	<p>Let <math xmlns="http://www.w3.org/1998/Math/MathML"><mrow><mi mathvariant="normal">R</m…	NaN
4	11232	ncert12maths	Relations and Functions	12	16-08-2021 4.29	16-08-2021 4.29	NCERT Exemplar	<p>Let <math xmlns="http://www.w3.org/1998/Math/MathML"><mrow><mi mathvariant="bold">N</mi>…	NaN

After describing the first set of raw data we understood that,

In [192…

```
sub.shape
```

Out[192…

```
(2065, 9)
```

In [193…

```
sub.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2065 entries, 0 to 2064
Data columns (total 9 columns):
 #   Column               Non-Null Count  Dtype  
---  ------               --------------  -----  
 0   id                   2065 non-null   int64  
 1   author               2065 non-null   object 
 2   chapter              2065 non-null   object 
 3   class_name           2065 non-null   int64  
 4   creation_time_stamp  2063 non-null   object 
 5   edit_time_stamp      2065 non-null   object 
 6   folder               2065 non-null   object 
 7   question_text        2065 non-null   object 
 8   subject              0 non-null      float64
dtypes: float64(1), int64(2), object(6)
memory usage: 145.3+ KB
```

In [194…

```
sub.describe()
```

Out[194…

	id	class_name	subject
count	2065.000000	2065.000000	0.0
mean	12994.683293	11.484262	NaN
std	1067.054116	0.499873	NaN
min	11228.000000	11.000000	NaN
25%	12186.000000	11.000000	NaN
50%	13253.000000	11.000000	NaN
75%	13866.000000	12.000000	NaN
max	16000.000000	12.000000	NaN

- The above dataset contains variables to predict the subject label. Currently, the subject column is empty (missing values) which has to be filled with the actual subject labels (Maths, Chemistry, Physics, Biology) from the 11th and 12th classes.
- There are 2065 rows and 9 columns affecting the dataset.
- We only need Actual Subject and Question columns for the model building.

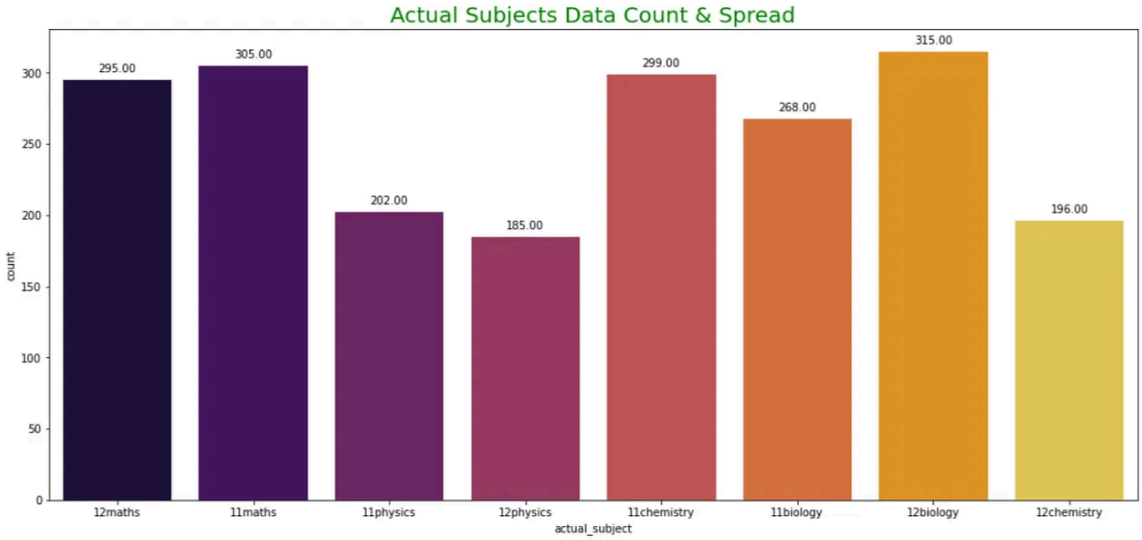
After handling the missing values and using some regex functions to separate the Subject and Questions columns for our further model building we are going to check the imbalance in the dataset.

Imbalanced Classes

```
# Checking the imbalance in the Target Variable

plt.figure(figsize=[18,8])
plot = sns.countplot(subjects.actual_subject, palette = 'inferno')
for p in plot.patches:
    plot.annotate(format(p.get_height(), '.2f'), (p.get_x() +
p.get_width() / 2., p.get_height()), ha = 'center', va = 'center',
xytext = (0, 10), textcoords = 'offset points')

plt.title('Actual Subjects Data Count & Spread', fontdict=
{'fontsize': 20, 'fontweight': 5, 'color': 'Green'})
#plt.xticks(rotation=90)
plt.show()
```

In [201…

```
# Checking the imbalance in the Target Variable

plt.figure(figsize=[18,8])
plot = sns.countplot(subjects.actual_subject, palette = 'inferno')
for p in plot.patches:
    plot.annotate(format(p.get_height(), '.2f'), (p.get_x() + p.get_width() / 2., p.get_height()), ha = 'center', va = 'center', xytext = (0, 10), tex

plt.title('Actual Subjects Data Count & Spread', fontdict={'fontsize': 20, 'fontweight': 5, 'color': 'Green'})
#plt.xticks(rotation=90)
plt.show()
```



- The above data seems to be moderately balanced between the available subject names
- Also, the data above is “Actual Subject” data

Bag of words model

The most common and most popular approach is to create a bag-of-words representation of the text data that you have. The central idea is that any given piece of text, i.e., tweets, articles, messages, emails, etc., can be “represented” by a list of all the words that occur in it (after removing the stopwords), where the sequence of occurrence does not matter. You can visualize it as the “bag” of all “words” that occur in it. For example, consider the messages:

“Gangs of Wasseypur is a great movie”

The bag of words representation for this message would be:

This way, you can create “bags” for representing each of the messages in your training and test data set.

Now, the next question is, how do you get a machine to do all of that? Well, turns out that for doing that, you need to represent all the bags in a matrix format, after which you can use ML algorithms such as naive Bayes, logistic regression, SVM, etc., to do the final classification. The bag-of-words representation is also called the bag-of-words model but this is not to be confused with a machine learning model. A bag-of-words model is just the matrix that you get from text data. Another thing to note is that the values inside any cell can be filled in two ways — 1) you can either fill the cell with the frequency of a word (i.e. a cell can have a value of 0 or more), or 2) fill the cell with either 0, in case the word is not present or 1, in case the word is present (binary format).

Both approaches work fine and don’t usually result in a big difference. The frequency approach is slightly more popular and the NLTK library in Python also fills the bag-of-words model with word frequencies rather than binary 0 or 1 values.

To build a bag-of-words model in Python, you can use the scikit-learn library. Steps to build a Bag of words model are:

- Subetting the dataset
- Plotting word frequencies and removing stopwords
- Tokenization
- Stemming
- Lemmatization

Notes: *You can find the source code for the Bag of Words model on Github. I look forward to hearing any feedback or questions.*

Stemming and lemmatising

Stemming:

It is a rule-based technique that just chops off the suffix of a word to get its root form, which is called the 'stem'. For example, if you use a stemmer to stem the words of the string — "The driver is racing in his boss' car", the words 'driver' and 'racing' will be converted to their root form by just chopping off the suffixes 'er' and 'ing'. So, 'driver' will be converted to 'driv' and 'racing' will be converted to 'rac'.

You might think that the root forms (or stems) don't resemble the root words — 'drive' and 'race'. You don't have to worry about this because the stemmer will convert all the variants of 'drive' and 'racing' to those root forms only. So, it will convert 'drive', 'driving', etc. to 'driv', and 'race', 'racer', etc. to 'rac'. This gives us satisfactory results in most cases.

Lemmatising

This is a more sophisticated technique (and perhaps more 'intelligent') in the sense that it doesn't just chop off the suffix of a word. Instead, it takes an input word and searches for its base word by going recursively through all the variations of dictionary words. The base word, in this case, is called the lemma. Words such as 'feet', 'drove', 'arose', 'bought', etc. can't be reduced to their correct base form using a stemmer. But a lemmatizer can reduce them to their correct base form. The most popular lemmatizer is the WordNet lemmatizer created by a team of researchers at Princeton University.

Nevertheless, you may sometimes find yourself confused about whether to use a stemmer or a lemmatizer in your application. The following points might help you make the decision:

- A stemmer is a rule-based technique, and hence, it is much faster than the lemmatizer (which searches the dictionary to look for the lemma of a word). On the other hand, a stemmer typically gives less accurate results than a lemmatizer.
- A lemmatizer is slower because of the dictionary lookup but gives better results than a stemmer. Now, as a side note, it is important to know that for a lemmatizer to perform accurately, you need to provide the part-of-speech tag of the input word (noun, verb, adjective, etc.). You'll see learn POS tagging in the next session — but it would suffice to know that there are often cases when the POS tagger itself is quite inaccurate on your text, and that will worsen the performance of the lemmatizer as well. In short, you may want to consider a stemmer rather than a lemmatizer if you notice that POS tagging is inaccurate.

In general, you can try both and see if it's worth using a lemmatizer over a stemmer. If a stemmer is giving you almost the same results with increased efficiency then choose a stemmer, otherwise use a lemmatizer.

***Notes**: You can find the source code for Stemming and Lemmatising on Github. I look forward to hearing any feedback or questions.*

TF-IDF model

The bag of words representation, while effective, is a very naive way of representing text. It relies on just the word frequencies of the words of a document. But don't you think word representation shouldn't solely rely on the word frequency? There is another way to represent documents in a matrix format which represents a word in a smarter way. It's called the TF-IDF representation and it is the one that is often preferred by most data scientists.

The term TF stands for term frequency, and the term IDF stands for inverse document frequency. How is this different from bag-of-words representation? Professor Srinath explains the concept of TF-IDF below.

This means the term is relevant to a document if it appears frequently and it is more or less unique to this document (not appearing in all documents). Term Frequency should be high and Document Frequency should below.

The TF-IDF representation, also called the TF-IDF model, takes into account the importance of each word. In the bag-of-words model, each word is assumed to be equally important, which is of course not correct.

The formula to calculate TF-IDF weight of a term in a document is:

- tf t,d = frequency of term 't' in document 'd' / total terms in document 'd'

- idf t = log (total number of documents / total documents that have the term 't')

The log in the above formula is with base 10. Now, the tf-idf score for any term in a document is just the product of these two terms:

- tf–idf = tf t,d ∗ idf t

Higher weights are assigned to terms that are present frequently in a document and which are rare among all documents. On the other hand, a low score is assigned to terms that are common across all documents.

Note that tf-idf is implemented in different ways in different languages and packages. In the tf score representation, some people use only the frequency of the term, i.e. they don't divide the frequency of the term by the total number of terms. In the idf score representation, some people use a natural log instead of the log with base 10. Due to this, you may see a different score for the same terms in the same set of documents. But the goal remains the same — assign a weight according to the word's importance.

```
In [238…
from sklearn.feature_extraction.text import TfidfVectorizer
tfidf = TfidfVectorizer(sublinear_tf=True, min_df=5, norm='l2', encoding='latin-1', ngram_range=(1, 2), stop_words='english')
features = tfidf.fit_transform(subjects.question_text).toarray()
labels = subjects.category_id
features.shape

Out[238…
(2065, 1126)
```

- Now, each of 2065 questions is represented by 1126 features, representing the tf-idf score for different unigrams and bigrams.
- We can use sklearn.feature_selection.chi2 to find the terms that are the most correlated with each of the subjects

In [240…

```
from sklearn.feature_selection import chi2
import numpy as np
N = 2
for actual_subject, category_id in sorted(category_to_id.items()):
    features_chi2 = chi2(features, labels == category_id)
    indices = np.argsort(features_chi2[0])
    feature_names = np.array(tfidf.get_feature_names())[indices]
    unigrams = [v for v in feature_names if len(v.split(' ')) == 1]
    bigrams = [v for v in feature_names if len(v.split(' ')) == 2]
    print("# '{}':".format(actual_subject))
    print("  . Most correlated unigrams:\n. {}".format('\n. '.join(unigrams[-N:])))
    print("  . Most correlated bigrams:\n. {}".format('\n. '.join(bigrams[-N:])))
```

```
# '11biology':
  . Most correlated unigrams:
. belong
. hormone
  . Most correlated bigrams:
. statements incorrect
. living organisms
# '11chemistry':
  . Most correlated unigrams:
. compounds
. iupac
  . Most correlated bigrams:
. correct iupac
. following compounds
# '11maths':
  . Most correlated unigrams:
. nbsp
. mo
  . Most correlated bigrams:
. mo nbsp
. nbsp mo
# '11physics':
  . Most correlated unigrams:
. speed
. motion
  . Most correlated bigrams:
. speed nbsp
. mass nbsp
# '12biology':
  . Most correlated unigrams:
. genes
. dna
  . Most correlated bigrams:
. mo nbsp
. mi nbsp
# '12chemistry':
  . Most correlated unigrams:
. adsorption
. solid
  . Most correlated bigrams:
. nbsp following
. order reaction
# '12maths':
```

```
  . Most correlated unigrams:
. texclass
. mo
  . Most correlated bigrams:
. mo nbsp
. msup mi
# '12physics':
  . Most correlated unigrams:
. current
. fleld
  . Most correlated bigrams:
. magnetic fleld
. electric fleld
```

- After all the above data transformation, now that we have all the features and labels, it is time to train the classifiers. There are a number of algorithms we can use for this type of problem.

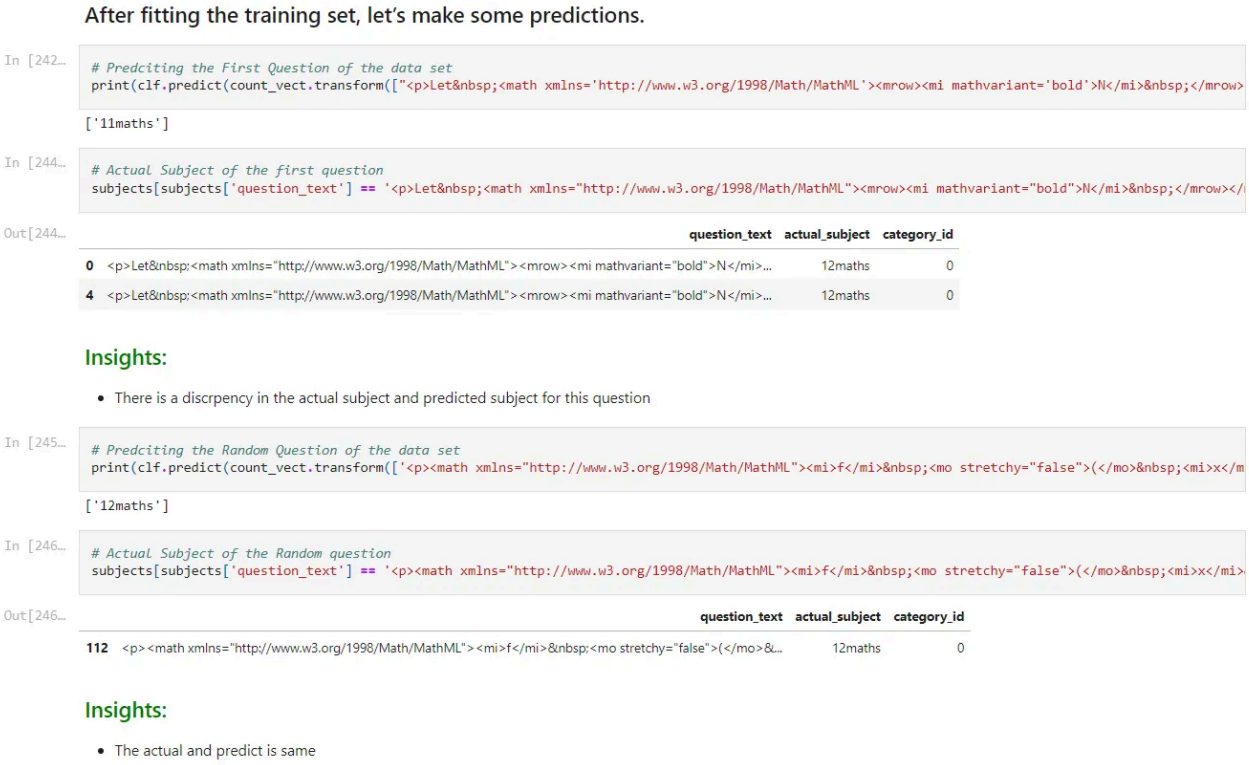
Notes: *You can find the source code for TF-IDF model on Github. I look forward to hearing any feedback or questions.*

Naive Bayes Classifier

the one most suitable for word counts is the multinomial variant:

```
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.naive_bayes import MultinomialNB

X_train, X_test, y_train, y_test =
train_test_split(subjects['question_text'],
subjects['actual_subject'],
                                                    random_state = 0)

count_vect = CountVectorizer()
X_train_counts = count_vect.fit_transform(X_train)

tfidf_transformer = TfidfTransformer()
X_train_tfidf = tfidf_transformer.fit_transform(X_train_counts)

clf = MultinomialNB().fit(X_train_tfidf, y_train)
```

After fitting the training set, let's make some predictions.



After fitting the training set, let's make some predictions.

In [242…
```
# Predciting the First Question of the data set
print(clf.predict(count_vect.transform(["<p>Let <math xmlns='http://www.w3.org/1998/Math/MathML'><mrow><mi mathvariant='bold'>N</mi> </mrow>
```
['11maths']

In [244…
```
# Actual Subject of the first question
subjects[subjects['question_text'] == '<p>Let <math xmlns="http://www.w3.org/1998/Math/MathML"><mrow><mi mathvariant="bold">N</mi> </mrow></
```

Out[244…

	question_text	actual_subject	category_id
0	<p>Let <math xmlns="http://www.w3.org/1998/Math/MathML"><mrow><mi mathvariant="bold">N</mi>...	12maths	0
4	<p>Let <math xmlns="http://www.w3.org/1998/Math/MathML"><mrow><mi mathvariant="bold">N</mi>...	12maths	0

Insights:

- There is a discrpency in the actual subject and predicted subject for this question

In [245…
```
# Predciting the Random Question of the data set
print(clf.predict(count_vect.transform(['<p><math xmlns="http://www.w3.org/1998/Math/MathML"><mi>f</mi> <mo stretchy="false">(</mo> <mi>x</m
```
['12maths']

In [246…
```
# Actual Subject of the Random question
subjects[subjects['question_text'] == '<p><math xmlns="http://www.w3.org/1998/Math/MathML"><mi>f</mi> <mo stretchy="false">(</mo> <mi>x</mi>
```

Out[246…

	question_text	actual_subject	category_id
112	<p><math xmlns="http://www.w3.org/1998/Math/MathML"><mi>f</mi> <mo stretchy="false">(</mo>&...	12maths	0

Insights:

- The actual and predict is same

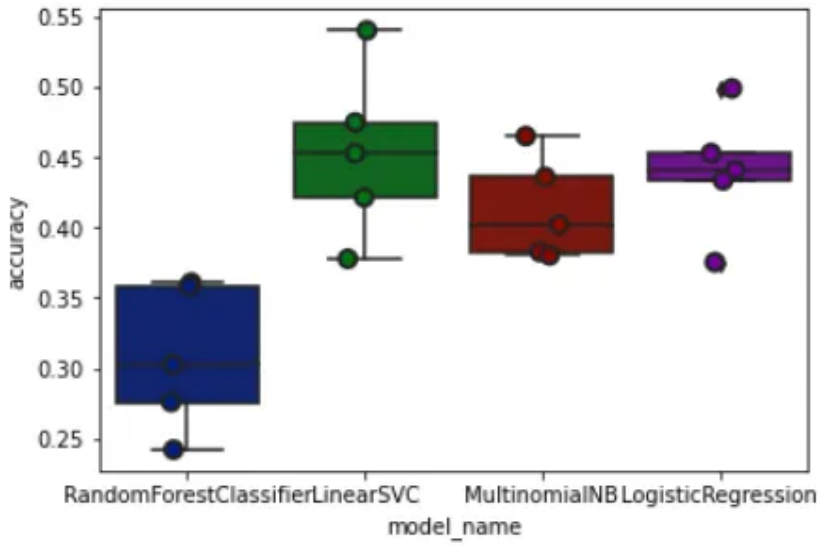
Model Selection

We will benchmark the following four models:

1. Logistic Regression
2. (Multinomial) Naive Bayes
3. Linear Support Vector Machine
4. Random Forest

```
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import LinearSVC
from sklearn.model_selection import cross_val_score

models = [
    RandomForestClassifier(n_estimators=200, max_depth=3,
random_state=0),
```

```
    LinearSVC(),
    MultinomialNB(),
    LogisticRegression(random_state=0),
]

CV = 5
cv_df = pd.DataFrame(index=range(CV * len(models)))

entries = []
for model in models:
    model_name = model.__class__.__name__
    accuracies = cross_val_score(model, features, labels,
scoring='accuracy', cv=CV)
    for fold_idx, accuracy in enumerate(accuracies):
        entries.append((model_name, fold_idx, accuracy))
cv_df = pd.DataFrame(entries, columns=['model_name', 'fold_idx',
'accuracy'])

import seaborn as sns
sns.boxplot(x='model_name', y='accuracy', data=cv_df)
sns.stripplot(x='model_name', y='accuracy', data=cv_df,
              size=8, jitter=True, edgecolor="gray", linewidth=2)
plt.show()

# Accuracy score
cv_df.groupby('model_name').accuracy.mean()
```




```
# Accuracy score
cv_df.groupby('model_name').accuracy.mean()

model_name
LinearSVC                 0.453269
```

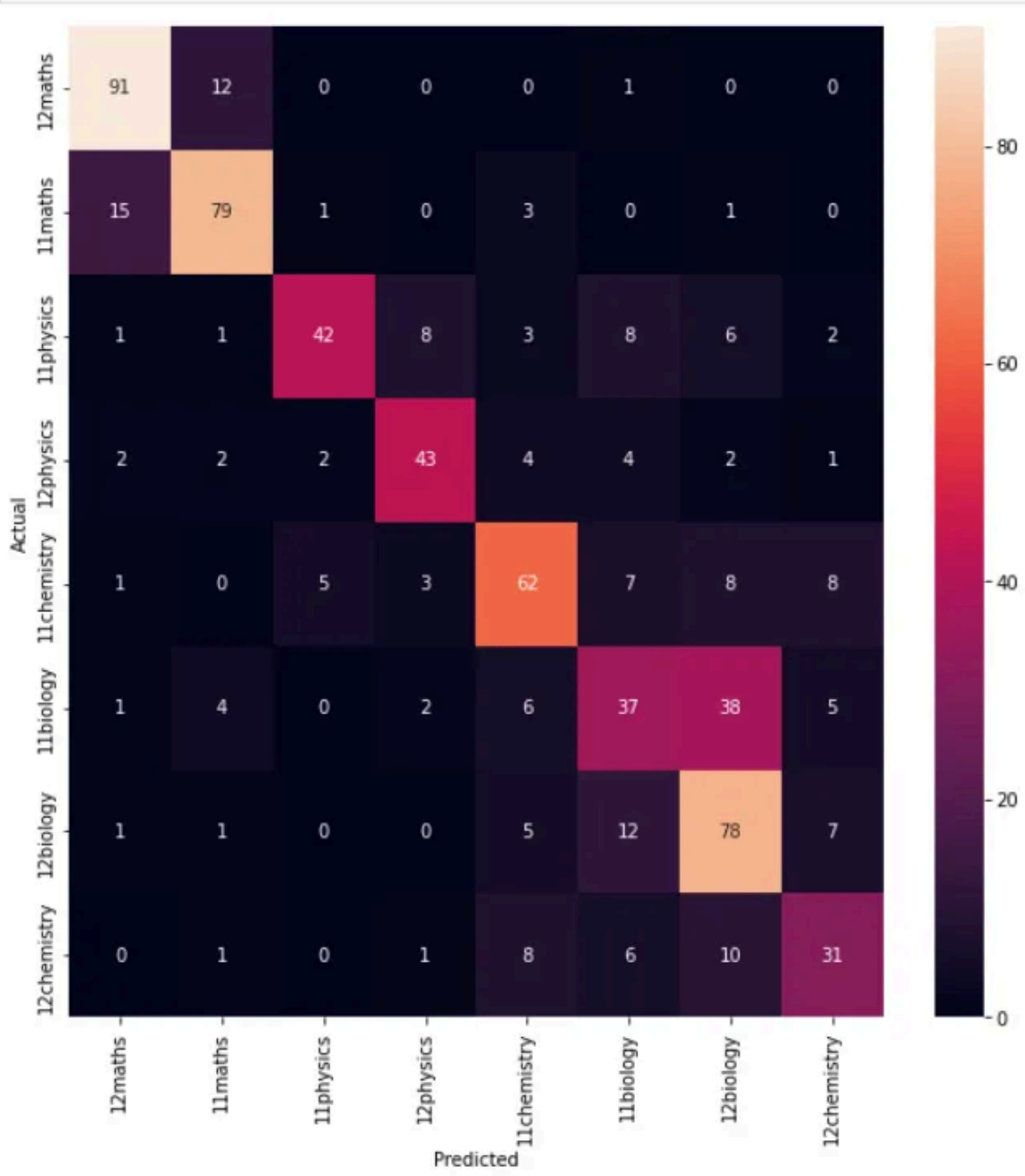
```
LogisticRegression        0.440194
MultinomialNB             0.413075
RandomForestClassifier    0.307990
Name: accuracy, dtype: float64
```

> LinearSVC and Logistic Regression perform better than the other two classifiers, with LinearSVC having a slight advantage with a median accuracy of around 45%.

Model Evaluation

Continuing with our best model (LinearSVC), we are going to look at the confusion matrix, and show the discrepancies between predicted and actual labels.

```
model = LinearSVC()

X_train, X_test, y_train, y_test, indices_train, indices_test =
train_test_split(features, labels, subjects.index, test_size=0.33,
random_state=0)
model.fit(X_train, y_train)
y_pred = model.predict(X_test)

from sklearn.metrics import confusion_matrix
conf_mat = confusion_matrix(y_test, y_pred)

fig, ax = plt.subplots(figsize=(10,10))
sns.heatmap(conf_mat, annot=True, fmt='d',
            xticklabels=category_id_df.actual_subject.values,
yticklabels=category_id_df.actual_subject.values)
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show()
```



The vast majority of the predictions end up on the diagonal (predicted label = actual label), where we want them to be. However, there are a number of misclassifications, and it might be interesting to see what those are caused by:

```
from IPython.display import display
for predicted in category_id_df.category_id:
    for actual in category_id_df.category_id:
        if predicted != actual and conf_mat[actual, predicted] >= 10:
```

```
            print("'{}' predicted as '{}' : {}
examples.".format(id_to_category[actual], id_to_category[predicted],

conf_mat[actual, predicted]))
            display(subjects.loc[indices_test[(y_test == actual) &
(y_pred == predicted)]][['actual_subject',

'question_text']])
            print('')
```

```
'11maths' predicted as '12maths' : 15 examples.

'12maths' predicted as '11maths' : 12 examples.

'12biology' predicted as '11biology' : 12 examples.

'11biology' predicted as '12biology' : 38 examples.

'12chemistry' predicted as '12biology' : 10 examples.
```

As you can see, some of the misclassified subjects are subjects that touch on more than one subject (for example, subjects involving both 11maths and 12maths). This sort of error will always happen.

Again, we use the chi-squared test to find the terms that are the most correlated with each of the categories:

```
model.fit(features, labels)

N = 2
for actual_subject, category_id in sorted(category_to_id.items()):
    features_chi2 = chi2(features, labels == category_id)
    indices = np.argsort(features_chi2[0])
    feature_names = np.array(tfidf.get_feature_names())[indices]
    unigrams = [v for v in feature_names if len(v.split(' ')) == 1]
    bigrams = [v for v in feature_names if len(v.split(' ')) == 2]
    print("# '{}':".format(actual_subject))
    print("  . Most correlated unigrams:\n. {}".format('\n.
'.join(unigrams[-N:])))
    print("  . Most correlated bigrams:\n. {}".format('\n.
'.join(bigrams[-N:])))
```

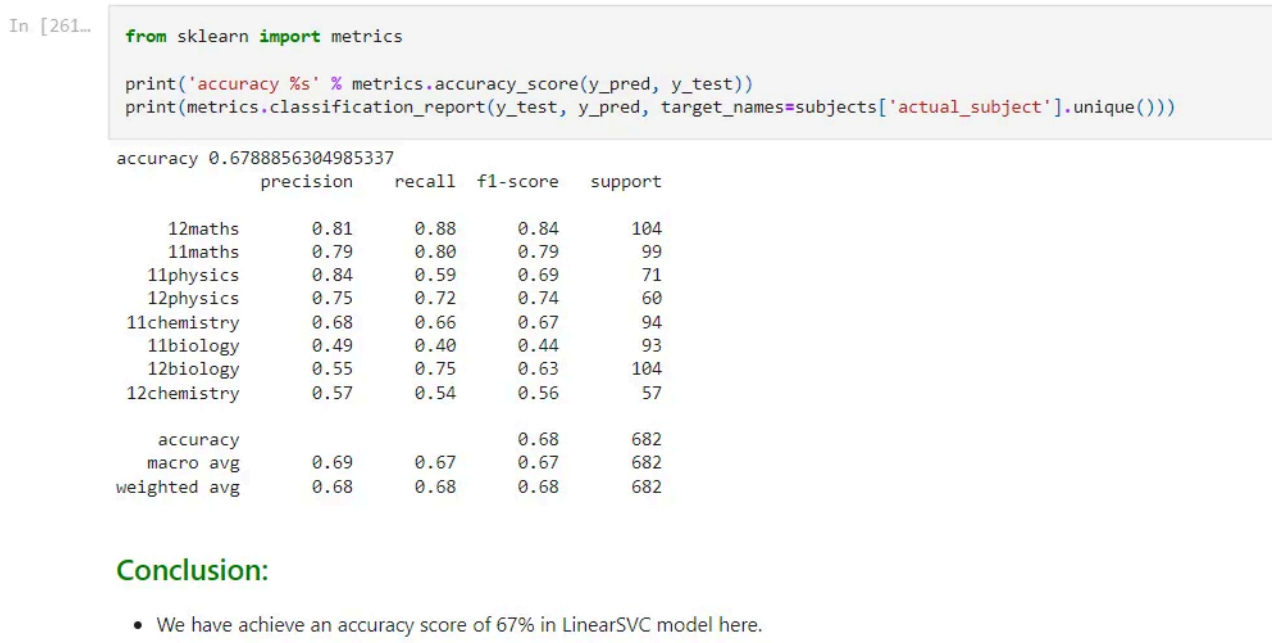
```
# '11biology':
  . Most correlated unigrams:
. belong
. hormone
  . Most correlated bigrams:
. statements incorrect
```

```
. living organisms
# '11chemistry':
  . Most correlated unigrams:
. compounds
. iupac
  . Most correlated bigrams:
. correct iupac
. following compounds
# '11maths':
  . Most correlated unigrams:
. nbsp
. mo
  . Most correlated bigrams:
. mo nbsp
. nbsp mo
# '11physics':
  . Most correlated unigrams:
. speed
. motion
  . Most correlated bigrams:
. speed nbsp
. mass nbsp
# '12biology':
  . Most correlated unigrams:
. genes
. dna
  . Most correlated bigrams:
. mo nbsp
. mi nbsp
# '12chemistry':
  . Most correlated unigrams:
. adsorption
. solid
  . Most correlated bigrams:
. nbsp following
. order reaction
# '12maths':
  . Most correlated unigrams:
. texclass
. mo
  . Most correlated bigrams:
. mo nbsp
. msup mi
# '12physics':
  . Most correlated unigrams:
. current
. fleld
  . Most correlated bigrams:
. magnetic fleld
. electric fleld
```

They are consistent with our expectations.

Finally, we print out the classification report for each class:

```
from sklearn import metrics

print('accuracy %s' % metrics.accuracy_score(y_pred, y_test))
print(metrics.classification_report(y_test, y_pred,
target_names=subjects['actual_subject'].unique()))
```



Conclusion:

- We have achieved an accuracy score of 67% in the LinearSVC model here.

Source code can be found on Github. I look forward to hearing any feedback or questions.

Machine Learning | NLP | Data Science | Scikit Learn | Tf Idf

Written by Rohit Batra

83 followers · 80 following

👋 Hi, I'm Rohit Batra.🌱 I'm pursuing a Master's in Data Science @ International University of Applied Science, Berlin.📫 to reach me...rohitbatra027@gmail.com

Follow

Responses (1)

Write a response

What are your thoughts?

Humberto Cordioli
Oct 18, 2022

It would be nicer if we had the file to play with.

1 reply Reply

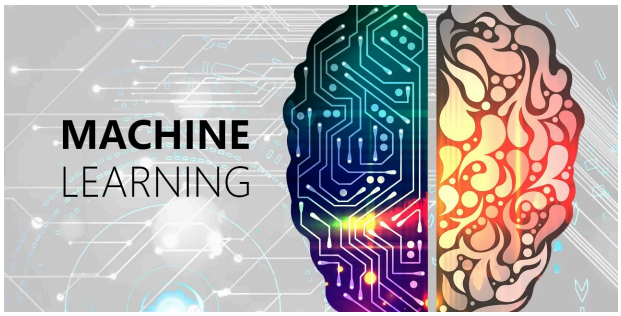
More from Rohit Batra

Rohit Batra

The Math Behind the K-means and Hierarchical Clustering Algorithm!

Understanding Clustering:

Feb 11, 2022 36



Rohit Batra

Linear Regression algorithm using statsmodels and scikit-learn!

Statistics behind Linear Regression model:

Jan 22, 2022 5

Rohit Batra

Logistic Regression Algorithm in just 4 steps!

Multivariate Logistic Regression:

Mar 13, 2022 1

Rohit Batra

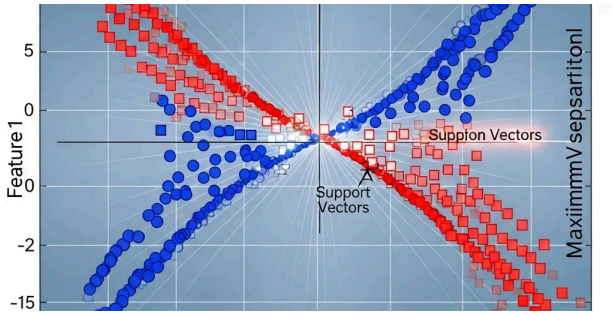
Data Storytelling+Methodology Used-Airbnb, NYC Analysis using...

Purpose: #Methodological Document for Data Analysis on Airbnb, NYC

Mar 17, 2022 7

See all from Rohit Batra

Recommended from Medium

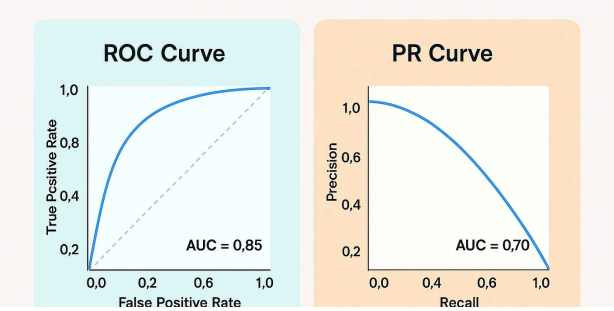


Shankar Angadi

AI: What is Support Vector Kernel in SVM? Part 34

In Support Vector Machines (SVM), kernels are functions that enable SVMs to handle...

Aug 22 52 1

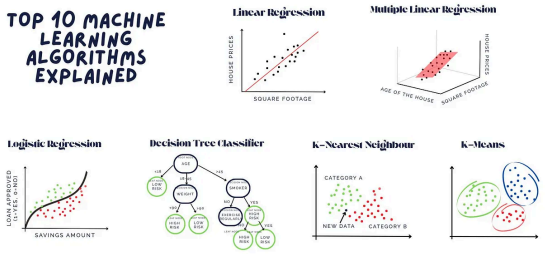


Samriddhi Saxena

How Good Is Your AI? ROC vs PR-AUC Explained Simply

Imagine you're building a smart system that detects fraudulent bank transactions.....

Jun 25 23

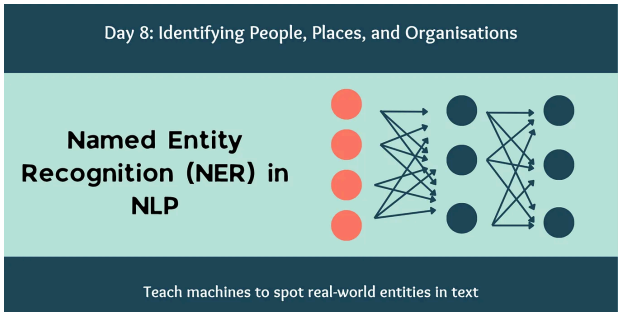


In Learning Data by Rita Angelou

10 ML Algorithms Every Data Scientist Should Know—Part 1

I understand well that machine learning might sound intimidating. But once you break dow...

Jun 10 104 2



Maheera Amjad

Named Entity Recognition (NER): Identifying People, Places, and...

Teach machines to spot real-world entities in text.

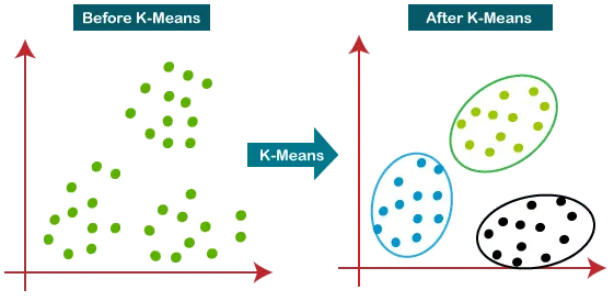
Oct 9 51

PY In Python in Plain English by Vishal Jain

Analyzing Activation Functions: Insights from Training on...

Activation Function Hands-on

Jul 1 51



Abhay singh

K-Means Clustering: A Deep Dive into Unsupervised Learning

K-means clustering is a popular method for grouping data by assigning observations to...

Jul 1 6

See more recommendations