In [ ]:
```
Name="Muhammad Omer Farooq Bhatti"
ID = "st122498"
```

Multinomial Naive Classification

The Gaussian assumption just described is by no means the only simple assumption that could be used to specify the generative distribution for each label. Another useful example is multinomial naive bayes, where the features are assumed to be generated from a simple multinomial distribution. **The multinomial distribution describes the probability of observing counts among a number of categories, and thus multinomial naive bayes is most appropriate for features that represent counts or count rates.**

The idea is precisely the same as before, except that instead of modeling the data distribution with the best-fit Gaussian, we model the data distribuiton with a best-fit multinomial distribution.

One place where multinomial naive Bayes is often used is in **text classification**, where the features w are related to word counts or frequencies within the documents to be classified and y will be our class. The formation is as follows:

$$P(y|w) = \frac{P(w|y)P(y)}{P(w)}$$

Implementation steps:

1. Prepare your data
   - $\mathbf{X}$ and $\mathbf{y}$ in the right shape
     - $\mathbf{X}$ -> (m, n)
     - $\mathbf{y}$ -> $(m,)$
     - Note that theta is not needed. Why?
   - train-test split
   - feature scale
   - clean out any missing data
   - (optional) feature engineering
2. Using the train documents, calculate the **likelihoods** of each word. Following multinomial distribution, for a given word w_i, we count how many of w_i belong in class k, we then divide this by the count of all the words that belong to k. This gives us the conditional probability for a word w given k:

$$P(w_i \in train \mid y = k) = \frac{count(w_i \in train, k)}{\sum_{i=1}^{n} count(w_i \in train, k)}$$

where

n stands for number of unique vocabulary (i.e., features) and m stands for number of documents (i.e., samples).

Example:

	docID	words in doc	China?
Training set	1	Chinese Beijing Chinese	Yes
	2	Chinese Chinese Shanghai	Yes
	3	Chinese Macao	Yes
	4	Tokyo Japan Chinese	No
Test set	5	Chinese Chinese Chinese Tokyo Japan	?

1. Since nothing in this world has zero probability, similarly, even we never see a particular word in some class should not gaurantee a zero probability, thus we can perform **Laplace smoothing** to account for any words with zero count. Also zero probability is not good when we do a product of probabilities.

$$P(w_i \in train \mid y = k) = \frac{count(w_i \in train, k) + 1}{\sum_{i=1}^{n} count(w_i \in train, k) + n}$$

1. Find **priors** $P(y)$ where is simply number of documents belonging to that class divided by all documents

$$P(y = k) = \frac{\Sigma_{i=1}^{m} 1(y = k)}{m}$$

1. Once we get the **likelihoods** from the train data. If given some test data, we simply use this likelihood to calculate the total likelihood of the test document. Similarly, since we have more than one word in the test document, we need to make a product of all likelihood of each word in the test document.

$$P(w \in test \mid y = k) = \prod_{i=1}^{n} P(w_i \in test \mid y = k)^{\text{freq of } w_i \in test}$$

Then we can multiply $P(y)P(w \in test \mid y)$ for each class which will give us $P(y \mid x)$ (**posteriors**)

$$P(y \mid x) = P(y=k)\prod_{i=1}^{n} P(w_i \in test \mid y=k)^{\text{freq of } w_i \in test}$$

1. Instead of probabilities, we gonna use log (base e) probabiities which have several benefits:

   - **Speed** - Log probabilities become addition, which is faster than multiplication
   - **Stability** - Probabilities can be too small where some significant digits can be lost during calculations. Log probabiities can prevent such underflow. If you don't believe me, try perform $\log_e(0.0000001)$
   - **Simplicity** - Many distributions have exponential form. Taking log cancels out the exp. The reason we can apply $\log$ is because $\log$ is a monotically increasing function, thus will not alter the result
   - **Dot product** - After log, addition can often expressed as dot product of matrix, simplifying the code implementation

   Now that you are convinced,

$$P(y=k)\prod_{i=1}^{n} p(w_i \in test \mid y=k)^{\text{freq of } w_i \in test}$$

   becomes

$$\log\ P(y=k) + (\text{freq of } w_i \in test) * \sum_{i=1}^{n} \log\ p(w_i \in test \mid y=k)$$

   - Note 1: Log of multiplication becomes addition
   - Note 2: Exponent of log becomes multiplicative scalar

   Thus, in implementation we can expressed as

```
np.log(priors) + X_test @ np.log(likelihoods.T)
```

When to Use Naive Bayes

Usually only as baseline! Because naive Bayesian classifiers make such stringent assumptions about data, they will **generally NOT perform as well as a more complicated model.** That said, they have several advantages:

- They are extremely fast for both training and prediction
- They provide straightforward probabilistic prediction
- They are often very easily interpretable
- They have very few (if any) tunable parameters

Naive Bayes classifiers tend to perform well only when your data is clearly separable or has high dimension.

The reason for high dimension is because new dimensions usually add more information, thus data become more separable. Thus, if you have really large dataset, try Naive Bayes and it may surprise you!

TfidVectorizer

Recall that in Naive Multinomial Classification, we want our features to be represented as frequency. Here, we shall go beyond one more step, i.e., after counting the number of words, we shall perform a normalization process called TF-IDF which focuses on **cutting very frequent words which tend to be less meaningful information like "the", "a", "is".**

Here is how it works underhood:

The formula is

$$\text{TF-IDF} = \text{TF} * \text{IDF}$$

where TF is

$$\text{TF}_t = \frac{\text{Count of words t in that document}}{\text{Total count of words in that document}}$$

Thus TF =

	1st word	2nd word	3rd word
doc1	3/4 = 0.75	0	1/4 = 0.25
doc2	2/3 = 0.66	1/3 = 0.33	0
doc3	3/10 = 0.33	2/10 = 0.20	5/10 = 0.5

and

$$\text{IDF} = \log\left(\frac{\text{Number of documents}}{\text{Number of documents containing that word}}\right) + 1$$

Note: We plus one so that super frequent words will not be ignored entirely

Thus IDF =

IDF

	IDF
1st word	log 3/3 + 1 = 1
2nd word	log 3/2 + 1 = 1.4055
3rd word	log 3/2 + 1 = 1.4055

Notice that terms (i.e., 1st word) that appear frequently across documents will get low score. By multiplying this IDF term with the frequency, it will scale the importance down.

Thus TF * IDF =

	1st word	2nd word	3rd word
doc1	0.75 * 1 = 0.75	0 * 1.4055 = 0	0.25 * 1.4055 = 0.3514
doc2	0.66 * 1 = 0.66	0.33 * 1.4055 = 0.4685	0 * 1.4055 = 0
doc3	0.33 * 1 = 0.33	0.20 * 1.4055 = 0.2811	0.5 * 1.4055 =0.7027

We need to further normalize each word using this formula (since each document has unequaled number of words):

$$norm(t_i) = \frac{t_i}{\sqrt{t_1^2 + t_2^2 + ... + t_n^2}}$$

Thus, normalized factor for each document is

doc1 = $\sqrt{0.75^2 + 0^2 + 0.3514^2} = 0.8282$

doc2 = $\sqrt{0.66^2 + 0.4685^2 + 0^2} = 0.8094$

doc3 = $\sqrt{0.33^2 + 0.281^2 + 0.7027^2} = 0.8256$

Thus, normalized(TF * IDF) =

	1st word	2nd word	3rd word
doc1	0.75 / 0.8282 = 0.9056	0	0.3514 / 0.8282 = 0.4243
doc2	0.66 / 0.8094 = 0.8154	0.4685 / 0.8094 = 0.5788	0
doc3	0.33 / 0.8256 = 0.3997	0.2811 / 0.8256 = 0.3405	0.7027 / 0.8256 = 0.8511

Note

- My numbers are not exactly the same due to float precisions

=== Task ===

1) Learn about TFidVectorizer and replace CountVectorizer with TFIDVectorizer (I have provided the explanation below.) 2) Put Multinomial Naive Classification into a class that can transform the data, fit the model and do prediction.

```
- In the class, allow users to choose whether to use CountVectorizer or TFIDVectorizer to transform
the data.
```

In [18]:

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics import confusion_matrix
from sklearn.preprocessing import label_binarize
from sklearn.metrics import average_precision_score, classification_report
from sklearn.feature_extraction.text import CountVectorizer
import seaborn as sns
```

In [55]:

```
class multinomial_NB:
    def __init__(self):
        self.trainData=[]
        self.testData=[]
        self.vectorizer=[]
        self.X_train=[]
        self.X_test=[]
        self.y_train=[]
        self.y_test=[]
        self.k=[]
        self.likelihoods=[]
        self.priors=[]
        self.method=[]

    def fit(self, train, test, method='count'):
        self.trainData=train
        self.testData=test
        self.y_train = self.trainData.target
        self.y_test = self.testData.target

        self.method=method
        if self.method=='count':
            self.count_transform()
        elif self.method=='tfidf':
            self.tfidf_transform()
        else:
            raise ValueError("Invalid method given as argument: ", self.method)
        self.m, self.n = self.X_train.shape

        classes = np.unique(self.y_train)  #list of class
        self.k = len(classes) #number of class
        self.priors = np.zeros(self.k) #prior for each classes
        self.likelihoods = np.zeros((self.k, self.n)) #likehood for each class of each feature

        for idx, label in enumerate(classes):
            X_train_by_class = self.X_train[self.y_train==label]
            self.priors[idx] = self.prior(X_train_by_class, self.m)
            self.likelihoods[idx, :] = self.likelihood(X_train_by_class)


    def predict(self, X_test= None):
        if X_test is None:
            X_test = self.X_test
        yhat = np.log(self.priors) + X_test @ np.log(self.likelihoods.T)
        yhat = np.argmax(yhat, axis=1)
        return yhat

    def count_transform(self):
        #transform our X to frequency data
        self.vectorizer = CountVectorizer()
        self.X_train = self.vectorizer.fit_transform(self.trainData.data)
        self.X_test = self.vectorizer.transform(self.testData.data)
        self.X_test = self.X_test.toarray()  #vectorizer gives us a sparse matrix; convert back to dense mat

    def tfidf_transform(self):
        #transform our X to tf-idf data
        self.vectorizer = TfidfVectorizer()
        self.X_train = self.vectorizer.fit_transform(self.trainData.data)
        self.X_test = self.vectorizer.transform(self.testData.data)
        self.X_test = self.X_test.toarray()  #vectorizer gives us a sparse matrix; convert back to dense mat

    def likelihood(self, X_class, laplace=1):
        return ((X_class.sum(axis=0)) + laplace) / (np.sum(X_class.sum(axis=0) + laplace))

    def prior(self, X_class, m):
        return X_class.shape[0] / m
```

In [9]:

```
from sklearn.datasets import fetch_20newsgroups

data = fetch_20newsgroups()
data.target_names

categories = ['talk.religion.misc', 'soc.religion.christian',
              'sci.space', 'comp.graphics']
train = fetch_20newsgroups(subset='train', categories=categories)
test = fetch_20newsgroups(subset='test', categories=categories)

print(train.data[0]) #first 300 words
print("Target: ", train.target[0])  #start with 1, soc.religion.christian
```

```
From: jono@mac-ak-24.rtsg.mot.com (Jon Ogden)
Subject: Re: Losing your temper is not a Christian trait
Organization: Motorola LPA Development
Lines: 26

In article <Apr.23.02.55.47.1993.3138@geneva.rutgers.edu>, jcj@tellabs.com
(jcj) wrote:

> I'd like to remind people of the withering of the fig tree and Jesus
> driving the money changers et. al. out of the temple.  I think those
```

```
> were two instances of Christ showing anger (as part of His human side).
>
Yes, and what about Paul saying:

26 Be ye angry, and sin not: let not the sun go down upon your wrath:
(Ephesians 4:26).

Obviously then, we can be angry w/o sinning.

Jon

------------------------------------------------
Jon Ogden         - jono@mac-ak-24.rtsg.mot.com
Motorola Cellular - Advanced Products Division
Voice: 708-632-2521      Data: 708-632-6086
------------------------------------------------

They drew a circle and shut him out.
Heretic, Rebel, a thing to flout.
But Love and I had the wit to win;
We drew a circle and took him in.
```

In [46]:

```python
def compute_metrics(y_test, yhat):
    n_classes = len(np.unique(y_test))

    print("Accuracy: ", np.sum(yhat == y_test)/len(y_test))

    print("=========Average precision score=======")
    y_test_binarized = label_binarize(y_test, classes=[0, 1, 2, 3])
    yhat_binarized = label_binarize(yhat, classes=[0, 1, 2, 3])

    for i in range(n_classes):
        class_score = average_precision_score(y_test_binarized[:, i], yhat_binarized[:, i])
        print(f"Class {i} score: ", class_score)

    print("=========Classification report=======")
    print("Report: ", classification_report(y_test, yhat))

    #use confusion matrix
    mat = confusion_matrix(y_test, yhat)


    sns.heatmap(mat.T, annot=True, fmt="d",
                xticklabels=train.target_names, yticklabels=train.target_names)
    plt.xlabel('true')
    plt.ylabel('predicted')
```

In [57]:

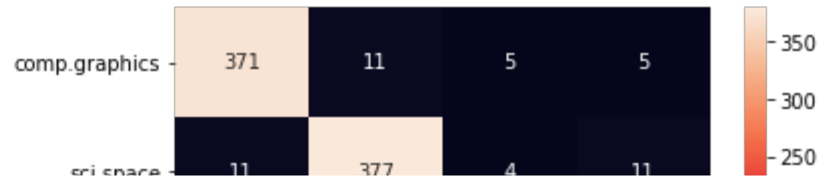
```python
#Instantiating our model class into an object
model = multinomial_NB()

#Using Count Vectorizer
model.fit(train, test, method='count')
yhat = model.predict()
print("Y_predicted from Counts: ",yhat)
compute_metrics(model.y_test, yhat)
```

```
Y_predicted from Counts:  [3 0 1 ... 1 2 1]
Accuracy:  0.9168994413407822
=========Average precision score=======
Class 0 score:  0.9152047938418233
Class 1 score:  0.9069918620723723
Class 2 score:  0.8429395016564877
Class 3 score:  0.7277310085946386
=========Classification report=======
Report:                precision    recall  f1-score   support

           0       0.95      0.95      0.95       389
           1       0.94      0.96      0.95       394
           2       0.87      0.95      0.91       398
           3       0.92      0.74      0.82       251

    accuracy                           0.92      1432
   macro avg       0.92      0.90      0.91      1432
weighted avg       0.92      0.92      0.92      1432
```

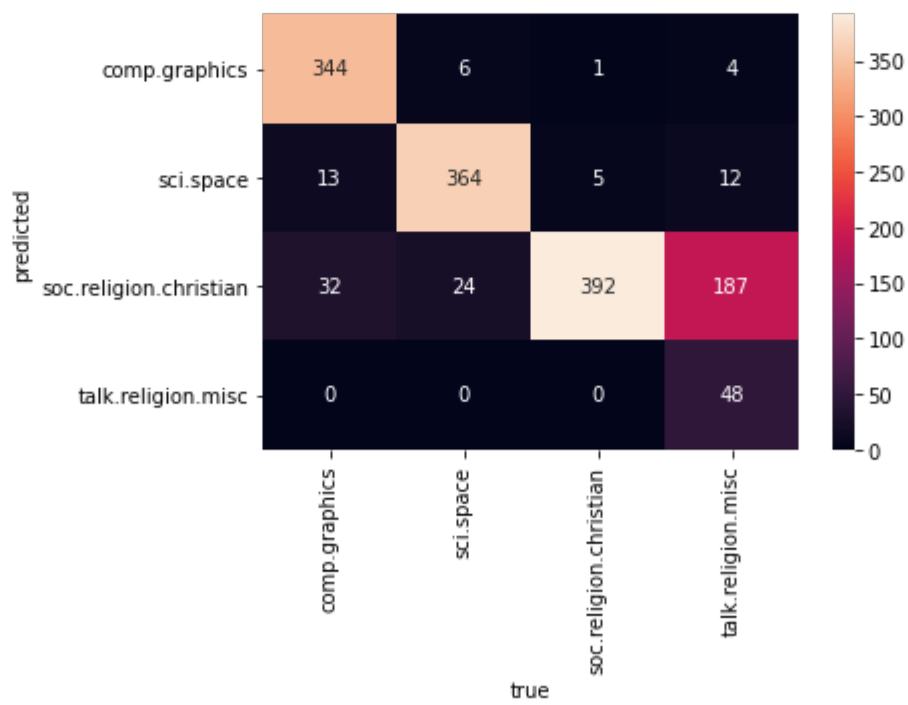


In [64]:
```
#Using TFIDF Vectorizer
model.fit(train, test, method='tfidf')
yhat = model.predict()
print("Y_predicted from TF-IDF: ",yhat)
compute_metrics(model.y_test, yhat)
```

```
Y_predicted from TF-IDF:  [2 0 1 ... 1 2 1]
Accuracy:  0.8016759776536313
=========Average precision score=======
Class 0 score:  0.888341920518241
Class 1 score:  0.8744630809734135
Class 2 score:  0.6122064043881043
Class 3 score:  0.332994836297269
=========Classification report=======
Report:                precision    recall  f1-score   support

           0       0.97      0.88      0.92       389
           1       0.92      0.92      0.92       394
           2       0.62      0.98      0.76       398
           3       1.00      0.19      0.32       251

    accuracy                           0.80      1432
   macro avg       0.88      0.75      0.73      1432
weighted avg       0.86      0.80      0.77      1432
```



In [69]:
```
some_string = ["elon musk is building a rocket",
               "God is good",
               "GUI of this program is very good"]
transformed = model.vectorizer.transform(some_string)
#print(transformed.shape)

prediction = model.predict(transformed)
print(prediction)
print(train.target_names[prediction[0]])
print(train.target_names[prediction[1]])
print(train.target_names[prediction[2]])
```

```
[1 2 0]
sci.space
soc.religion.christian
comp.graphics
```

In [ ]: