Part 1 Kaggle Competition: BBC News Classification

This is a submission for the Kaggle competition BBC News Classification (https://www.kaggle.com/c/learn-ai-bbc/overview). The subject of the problem statement is to work with text data sourced from the BBC, or the British Broadcasting Corporation which is one of the most well-known and possibly oldest national broadcaster in the world. The text data is of specific articles published by the BBC, though the contents have been pre-processed for ease of use as inputs for machine learning.

Our goal in this project is to take the text of an article, apply matrix factorization, and then classify the article under one of 5 categories: 'business', 'entertainment', 'politics', 'sport', or 'tech'.

Table of Contents:

01 - Exploratory Data Analysis

In [1]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from itertools import permutations

from sklearn.cluster import KMeans
from sklearn.decomposition import TruncatedSVD
from sklearn.decomposition import NMF
from sklearn.ensemble import RandomForestClassifier
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
```

We have access to 3 CSV files, a dataset to be used for training, a dataset to test predict our final result, and a sample solution. In this project, the training dataset will be focused on the most as the labels will be utilized for training and scoring. Utilizing the head function we can see that the train dataset has 3 columns,

one for the article ID, one for the text itself, and lastly the category that the article belongs to. Category will be our target variable to predict.

In [2]:
```
df_tr = pd.read_csv('data/BBC News Train.csv')
df_te = pd.read_csv('data/BBC News Test.csv')
df_so = pd.read_csv('data/BBC News Sample Solution.csv')

df_tr.head()
```

Out[2]:

	ArticleId	Text	Category
0	1833	worldcom ex-boss launches defence lawyers defe...	business
1	154	german business confidence slides german busin...	business
2	1101	bbc poll indicates economic gloom citizens in ...	business
3	1976	lifestyle governs mobile choice faster bett...	tech
4	917	enron bosses in $168m payout eighteen former e...	business

The test dataset is similar to the train dataset just without the category, which we will predict.

In [3]:
```
df_te.head()
```

Out[3]:

	ArticleId	Text
0	1018	qpr keeper day heads for preston queens park r...
1	1319	software watching while you work software that...
2	1138	d arcy injury adds to ireland woe gordon d arc...
3	459	india s reliance family feud heats up the ongo...
4	1020	boro suffer morrison injury blow middlesbrough...

The sample solution dataset displays the desired format the solution should be in.

In [4]:
```
df_so.head()
```

Out[4]:

	ArticleId	Category
0	1018	sport
1	1319	tech
2	1138	business
3	459	entertainment
4	1020	politics

We do some exploratory data analysis by utilizing the info function to get an overview of the dataset. Something we learn right away is that we have 3 columns with 1490 rows. As each row symbolizes an unique document, we thus have 1490 documents to work with. Luckily the dataset is quite clean as well, having zero null values so we do not have to worry about that.

In [5]:
```
df_tr.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1490 entries, 0 to 1489
Data columns (total 3 columns):
 #   Column     Non-Null Count  Dtype
```

```
---  ------     --------------  -----
 0   ArticleId  1490 non-null   int64
 1   Text       1490 non-null   object
 2   Category   1490 non-null   object
dtypes: int64(1), object(2)
memory usage: 35.0+ KB
```

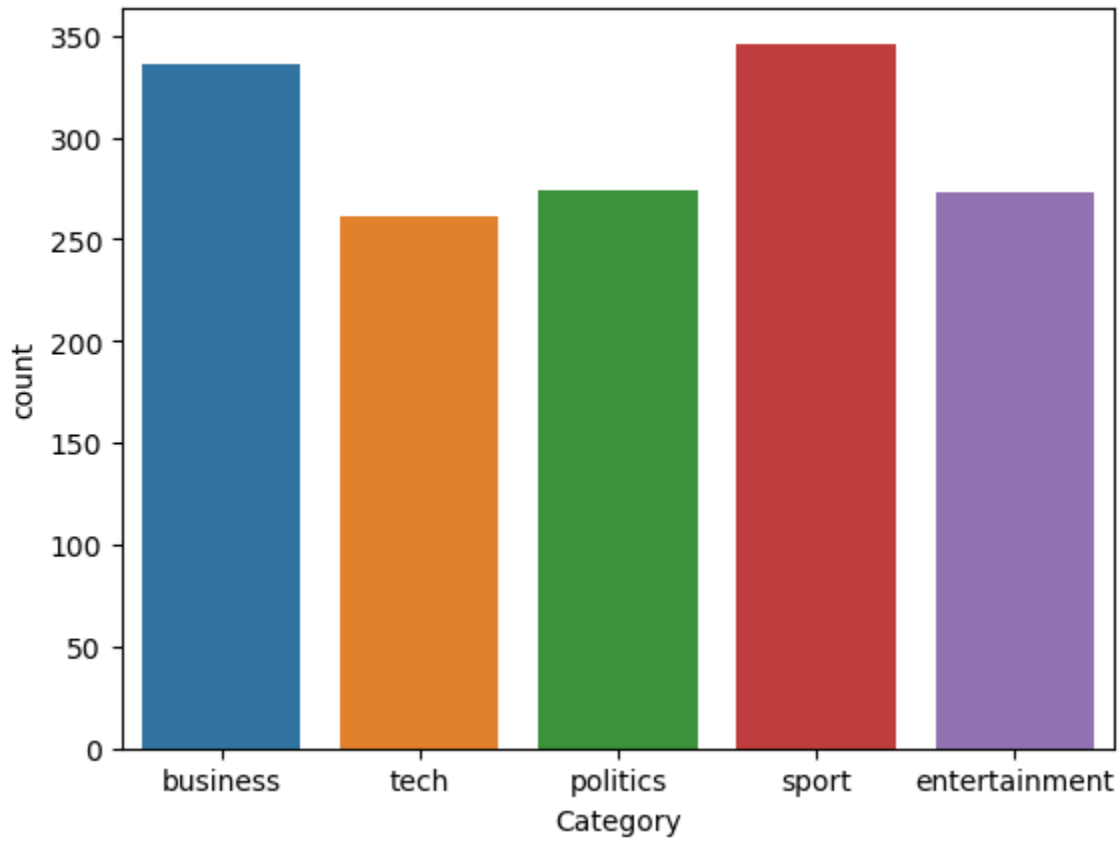
In [6]:
```
df_tr.isna().sum()
```

Out[6]:
```
ArticleId    0
Text         0
Category     0
dtype: int64
```

Thankfully it does not seem that there is any feature imbalance, as according to this countplot we do not have any classes in our target that are either over or under represented, thus we do not have to worry about that as well.

In [7]:
```
sns.countplot(data = df_tr, x = 'Category')
```

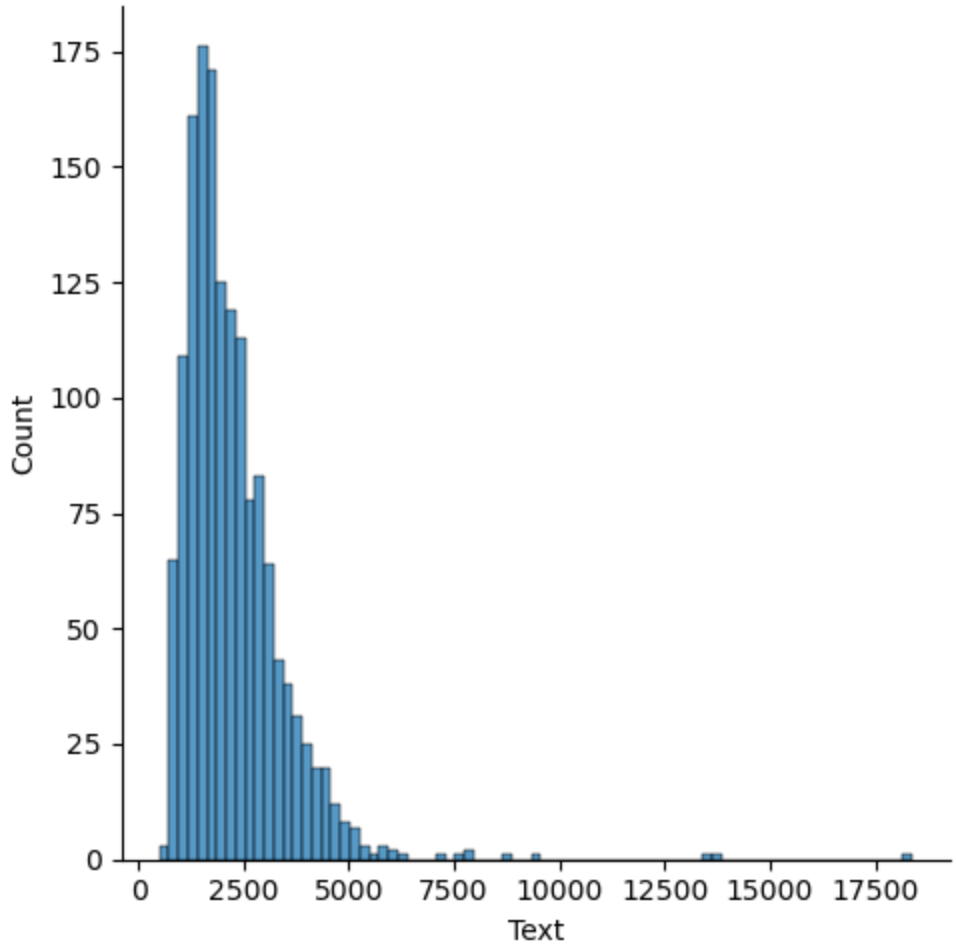
Out[7]:
```
<AxesSubplot:xlabel='Category', ylabel='count'>
```



Finally a visual on text length for each article concludes our EDA. We find the distribution on text length for each article, in case there is a significant portion of articles with vastly greater amount of text length. Here we can see that there is a nice distribution of texts from 500 ~ 6500 words. Further checking shows us that there are 9 articles that have much more than 6500 words, and as these articles only take up a small percentage out of the 1490 articles, it should be safe to ignore as they provide minimal impact.

In [8]:
```
sns.displot(df_tr['Text'].str.len())
```

Out[8]:
```
<seaborn.axisgrid.FacetGrid at 0x14c47961ac0>
```



Based on our EDA findings, our best option would most likely be to predict our labels through the article text. To do this, the best method would most likely to view the text and determine the labels by the contents of it. One such way would be to utilize TFIDF, a popular method used in NLP modeling.

02 - Data Preparation, Cleaning, and Visualization with TFIDF

As our articles are in the form of one large string, in order to turn them into values suitable for a classifier we will utilize the TfidfVectorizer library from sklearn. This library utilizes the TF-IDF algorithm, or the term frequency - inverse document frequency algorithm which is a way to quantify the relevancy a word is to a given document. The general gist of it is that the higher the TF-IDF value, the more significant a word is inside of the overall text.

TFIDF Concepts

- https://medium.com/@cmukesh8688/tf-idf-vectorizer-scikit-learn-dbc0244a911a

Sklearn Library:

- https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html

After creating an instance of the TfidfVectorizer, we fit it to our text data and transform it into the document-term matrix. As a result, we have a matrix with 1481 articles for the rows, and 24312 columns, with each column refering to a unique word in the text.

In [9]:
```
tfidf0 = TfidfVectorizer()
tfidf0_m = tfidf0.fit_transform(df_tr['Text'])
```

```
print(tfidf0_m.shape)
print(tfidf0.get_feature_names_out())
```

```
(1490, 24746)
['00' '000' '0001' ... 'zurich' 'zutons' 'zvonareva']
```

As seen in the preview of the feature names, there are bound to be words that are not actual words, and should be cut out. Thus we modify the hyperparameters a bit for our TfidfVectorizer to trim words that will not provide any real benefit. In particular, we will remove accents from our words (such as ä or è), remove common english stop words (such as if or within), ensure all letters are lowercase, and exclude numbers as well as words containing numbers. With the new hyperparameters, our new document-term matrix has cut down on the number of terms by around ~1.5 thousand.

Regex for excluding numbers as well as words containing numbers referenced from this stackoverflow post:

- https://stackoverflow.com/questions/51643427/how-to-make-tfidfvectorizer-only-learn-alphabetical-characters-as-part-of-the-vo

In [10]:
```
tfidf = TfidfVectorizer(strip_accents = 'ascii', stop_words = 'english', lowercase = Tru
                        token_pattern = r'(?u)\b[A-Za-z]+\b', min_df = 1)
tfidf_m = tfidf.fit_transform(df_tr['Text'])

print(tfidf_m.shape)
```

```
(1490, 23233)
```

To aid in our understanding of the document-term matrix, the matrix can be converted into a Pandas dataframe for a cleaner view.

In [11]:
```
tfidf_df = pd.DataFrame(tfidf_m.toarray(), index=df_tr.ArticleId, columns=tfidf.get_feat

tfidf_df.head()
```

Out[11]:

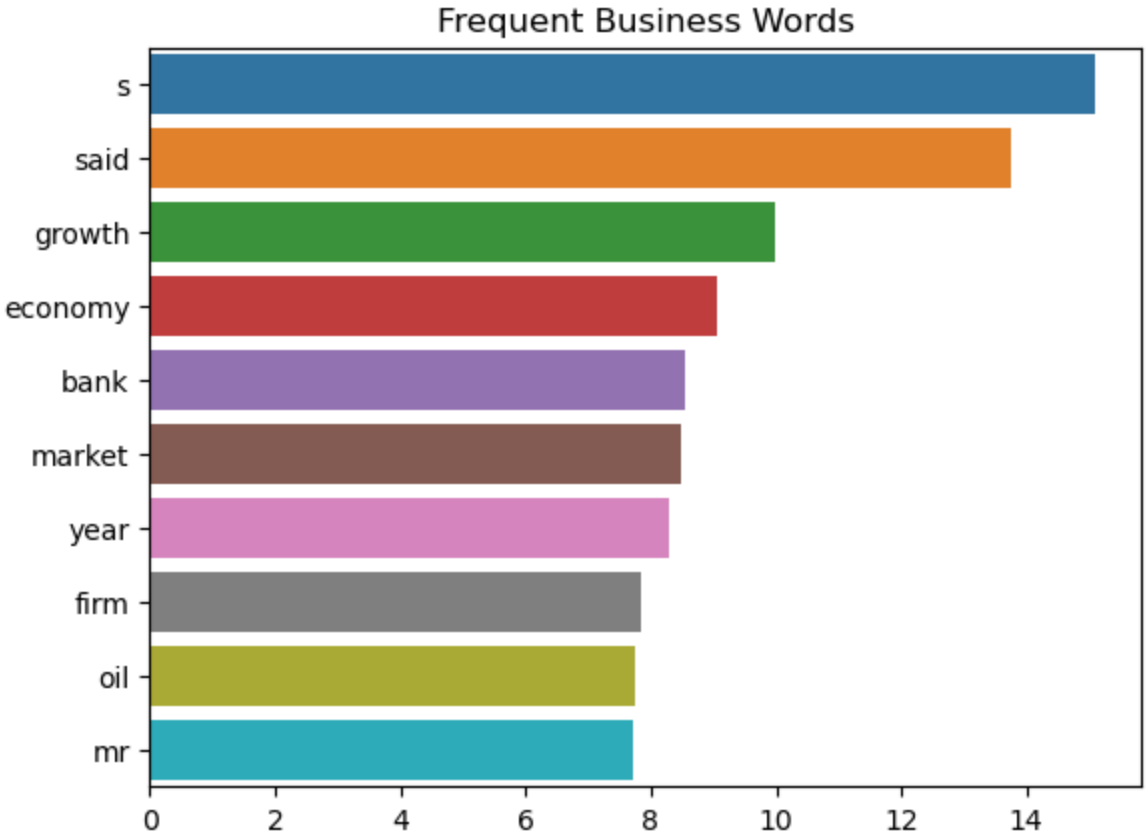
ArticleId	aa	aaa	aaas	aac	aadc	aaliyah	aaltra	aamir	aaron	abacus	...	zonealarm	zones	zoom	zoor
1833	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0
154	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0
1101	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0
1976	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0
917	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0

5 rows × 23233 columns

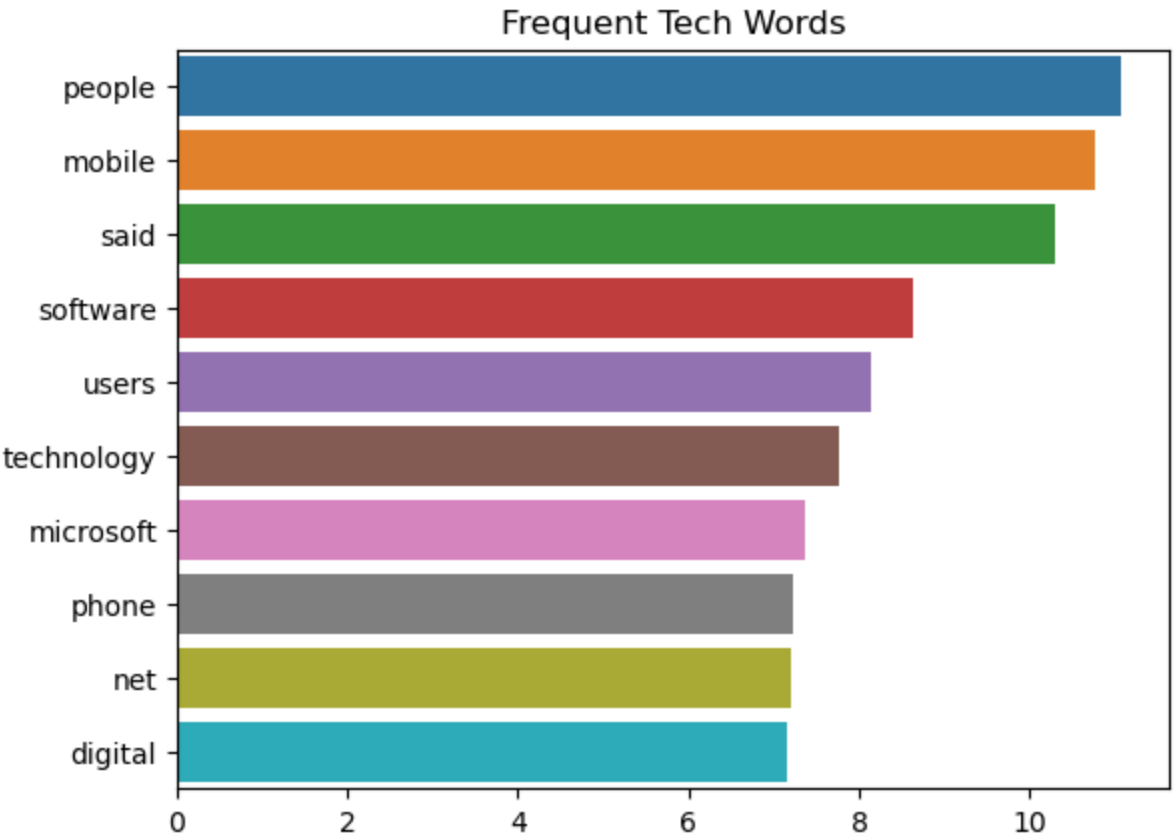
Now we can view the most popular words of each category, which will give us an idea of how an article containing a significant amount of a certain word can help sway our prediction of a category.

In [12]:
```
business = tfidf_df.loc[df_tr[df_tr['Category'] == 'business']['ArticleId'].values].sum(
tech = tfidf_df.loc[df_tr[df_tr['Category'] == 'tech']['ArticleId'].values].sum().sort_v
politics = tfidf_df.loc[df_tr[df_tr['Category'] == 'politics']['ArticleId'].values].sum(
sport = tfidf_df.loc[df_tr[df_tr['Category'] == 'sport']['ArticleId'].values].sum().sort
entertainment = tfidf_df.loc[df_tr[df_tr['Category'] == 'entertainment']['ArticleId'].va
```

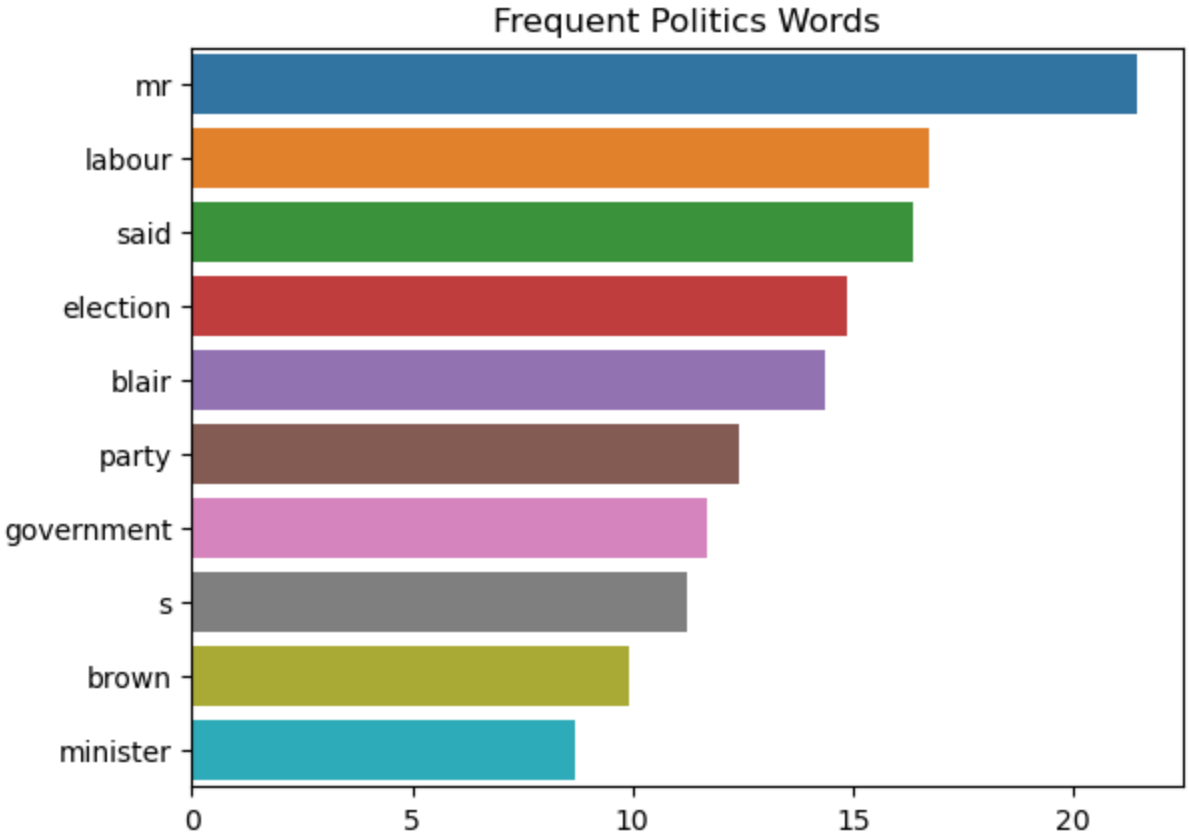
In [13]:
```
sns.barplot(y = business.keys(), x = business.values, ).set(title = 'Frequent Business W

plt.show()
```



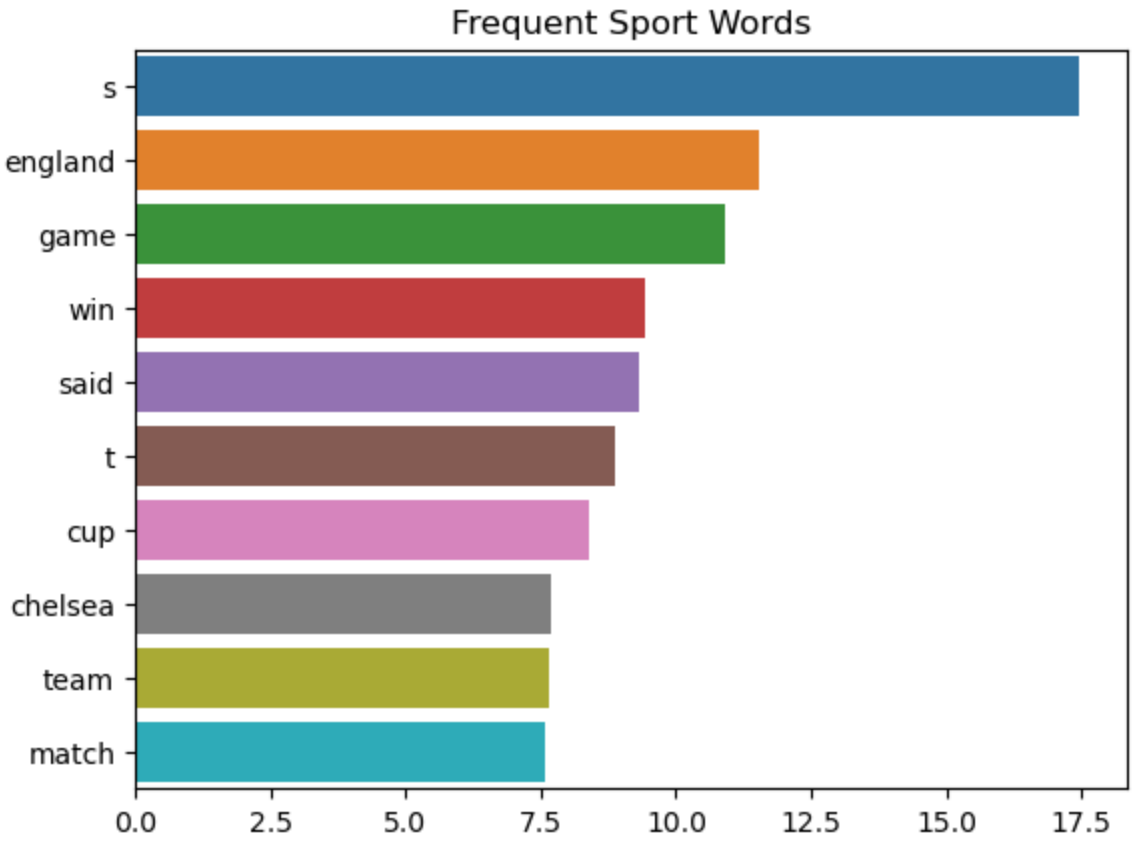
In [14]:
```
sns.barplot(y = tech.keys(), x = tech.values, ).set(title = 'Frequent Tech Words')

plt.show()
```



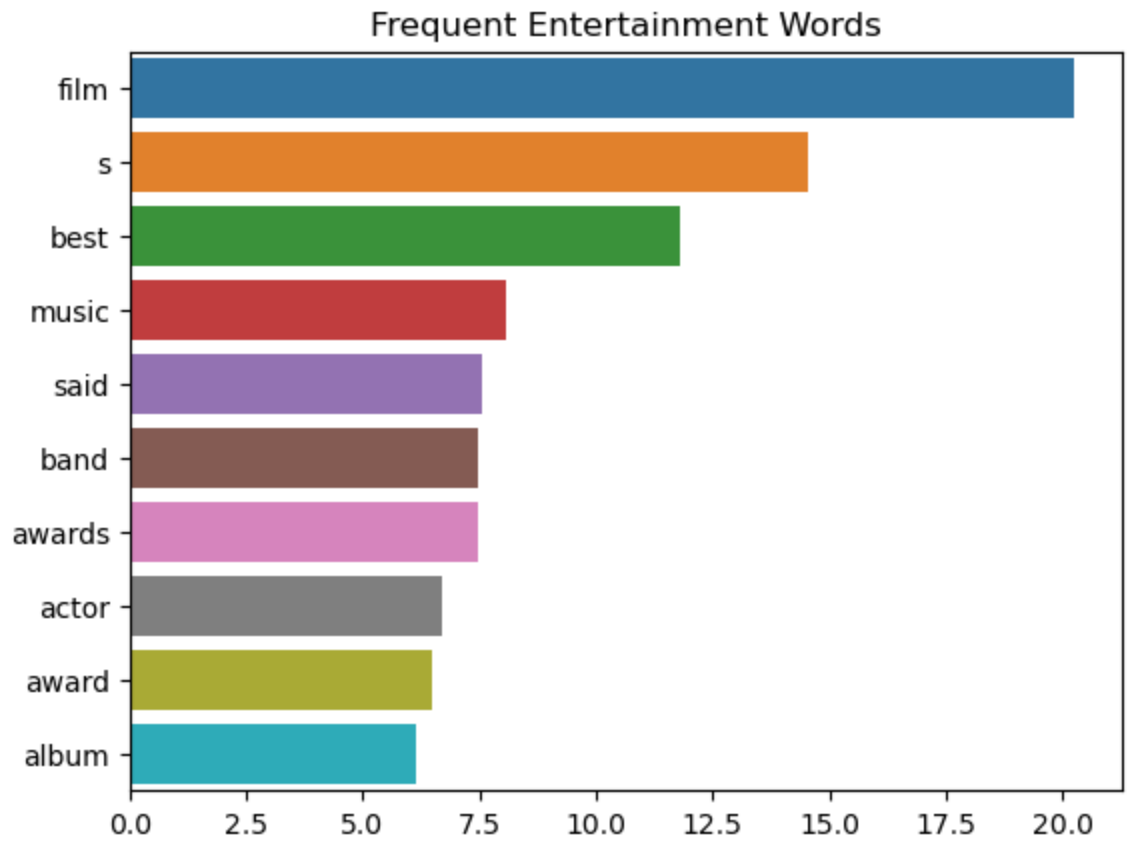
In [15]:
```
sns.barplot(y = politics.keys(), x = politics.values, ).set(title = 'Frequent Politics W

plt.show()
```



In [16]:
```
sns.barplot(y = sport.keys(), x = sport.values, ).set(title = 'Frequent Sport Words')

plt.show()
```



In [17]:
```
sns.barplot(y = entertainment.keys(), x = entertainment.values, ).set(title = 'Frequent

plt.show()
```



03 - SVD: Matrix Factorization

Now it is time for matrix factorization; the main benefit of matrix factorization is to restructure our matrix into one that will improve our model's overall performance. As seen above, our TF-IDF document-term matrix has 1481 rows and 22828 columns, which is particularily disastrous to work with in terms of modeling. To solve this, we will utilize matrix factorization to refactor our matrix into one much more manageable to work with.

For our matrix factorization, we will attempt both SVD and NMF and compare their resulting performance, and we will start with SVD. The sklearn library TruncatedSVD will be used for this portion, and we will start with 5 components as we are attempting to categorize our data into 5 different labels. The result will provide us a matrix with the same 1481 rows, yet have the 22828 columns turn into 5 columns.

Sklearn libray for SVD:

- https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.TruncatedSVD.html

In [18]:
```
svd = TruncatedSVD(n_components=5, n_iter=7, random_state=42)
svd_m = svd.fit_transform(tfidf_m)

print(svd_m.shape)
```

```
(1490, 5)
```

We are also able to view the U, Sigma, and V transposed matrices for our SVD instance.

In [19]:
```
U = np.matmul(tfidf_m.toarray(), np.matmul(svd.components_.T, np.diag(svd.singular_value
Vt = svd.components_
S = svd.singular_values_

print('U:\n', U)
```

```
print('S:\n', S)
print('Vt:\n', Vt)
```

```
U:
 [[ 0.83440459  0.14292701  0.07633619 -0.00314151  0.07904519]
 [ 1.08700573 -0.0029155   0.37545025 -0.09354241  0.84175244]
 [ 1.31824957  0.01203588  0.21845624 -0.10133893  0.37930231]
 ...
 [ 1.06256733 -0.14137185  0.30978934 -0.02764067  0.56504764]
 [ 1.45290768 -0.46688113  1.00680044  0.08914667 -0.4862283 ]
 [ 0.61793955 -0.13251198  0.40329295 -0.07028107 -0.25863027]]
S:
 [6.23390942 4.02253181 3.60347746 3.4287919  3.18674783]
Vt:
 [[ 2.04280722e-04  1.66382858e-03  7.04448689e-04 ...  1.82775352e-03
   3.99438391e-04  1.22355925e-03]
 [ 6.38869682e-05 -2.43744896e-03 -1.04041173e-03 ... -2.99109223e-03
  -5.32988993e-04 -2.35222458e-03]
 [ 1.19198550e-04 -3.03354215e-03 -6.67727677e-04 ... -3.31785483e-03
  -4.17631131e-04 -2.52186959e-03]
 [-3.53119157e-05 -2.36056814e-03 -8.52444896e-04 ... -5.17039023e-03
   2.76704418e-04 -2.06334156e-03]
 [ 1.25011033e-04  5.46934460e-04  8.95606348e-05 ... -8.95572618e-05
  -1.19280197e-05  3.49221613e-06]]
```

04 - SVD: Unsupervised Model

To start, we can answer the question of whether or not we should include texts from the test dataset in the input matrix. The idea is that we should treat it as a general case of training models with test datasets, in which it is not a good idea as we want to train our model so that it can understand patterns in the data. The test dataset is our way of testing how good the model works on unseen data, which is what we use the test data for. Additionally, while adding in test data will let the model learn from a wider range of inputs, it also contributes to overfitting in a way that will be harmful to our performance, and will require new data to test on.

To begin, the model of choice will be Kmeans Clustering and we will be using it on the training dataset. This is used over hierarhical clustering as we know the number of clusters we want beforehand, five, which is reflective to the number of unique labels we want to utilize. For our measurement metric, we will use sklearn's accuracy score metric.

In [20]:
```
X_train, X_test, y_train, y_test = train_test_split(svd_m, df_tr.Category, test_size = 0

kmeans = KMeans(n_clusters = 5, random_state = 42).fit(X_train)
```

In [21]:
```
def find_best_score(categories, labels):
    label_list = [ 0, 1, 2, 3, 4]

    best_labels = {}
    best_score = 0

    for i in permutations(label_list):
        label_dict = {'business':i[0], 'entertainment':i[1], 'politics':i[2], 'sport':i[
        a = accuracy_score([ label_dict[j] for j in categories ], labels)

        if a > best_score:
            best_score = a
            best_labels = label_dict

    return best_score, best_labels
```

In [22]:
```
print('Train Score:', find_best_score(y_train, kmeans.labels_)[0])
print('Test Score: ', find_best_score(y_test, kmeans.predict(X_test))[0])
```

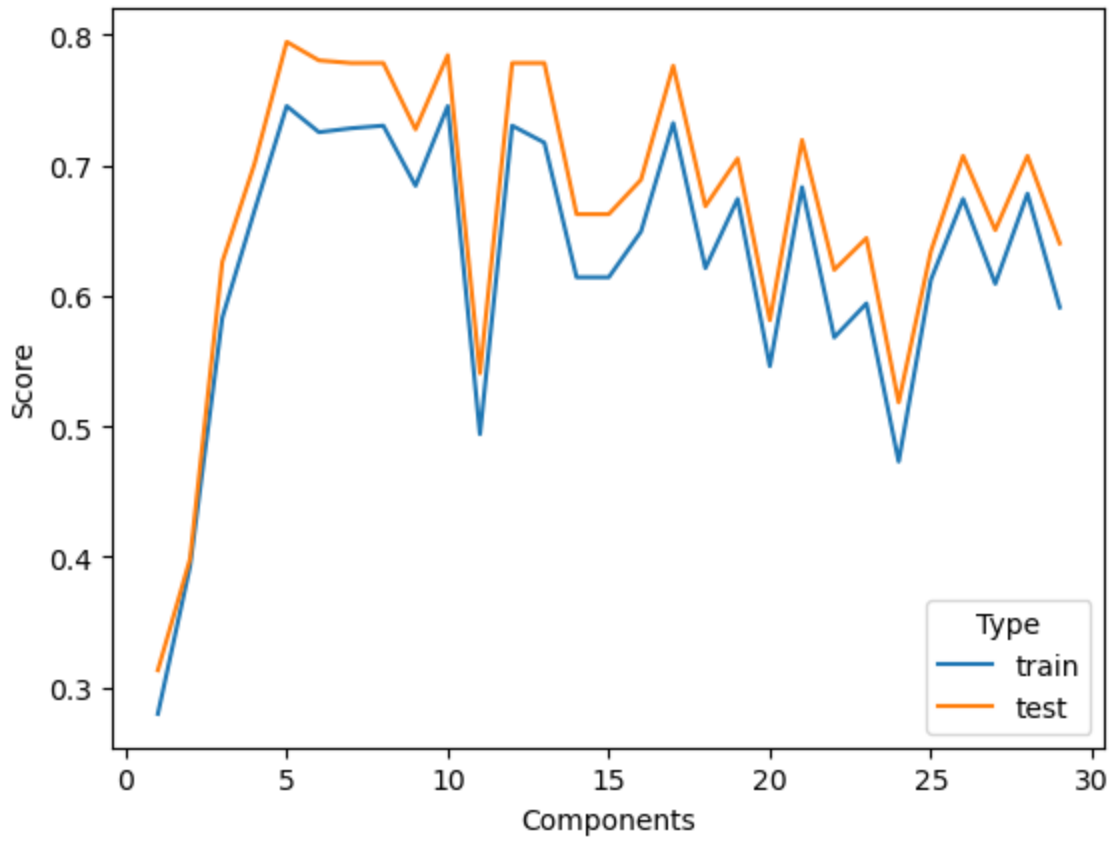
```
Train Score: 0.7454909819639278
Test Score:  0.7947154471544715
```

Now we will attempt this on the testing CSV.

In [23]:
```
tfidf_te = TfidfVectorizer(strip_accents = 'ascii', stop_words = 'english', lowercase =
                           token_pattern = r'(?u)\b[A-Za-z]+\b', min_df = 1).fit_transform(


svd_te = TruncatedSVD(n_components=5, n_iter=7, random_state=42).fit_transform(tfidf_te)


bl = { 0:'business', 2:'entertainment', 4:'politics', 1:'sport', 3:'tech'}

sol_te = [  bl[i] for i in kmeans.predict(svd_te) ]

df_solution = df_te[['ArticleId']]
df_solution['Category'] = sol_te

# df_solution.to_csv('sol.csv', index = False)
```

With our best test score as a 0.79, while it is not the worst score to work with, there is room for improvement so now we will test different hyperparameters. The hyperparameter we will focus on will be the number of components in our SVD.

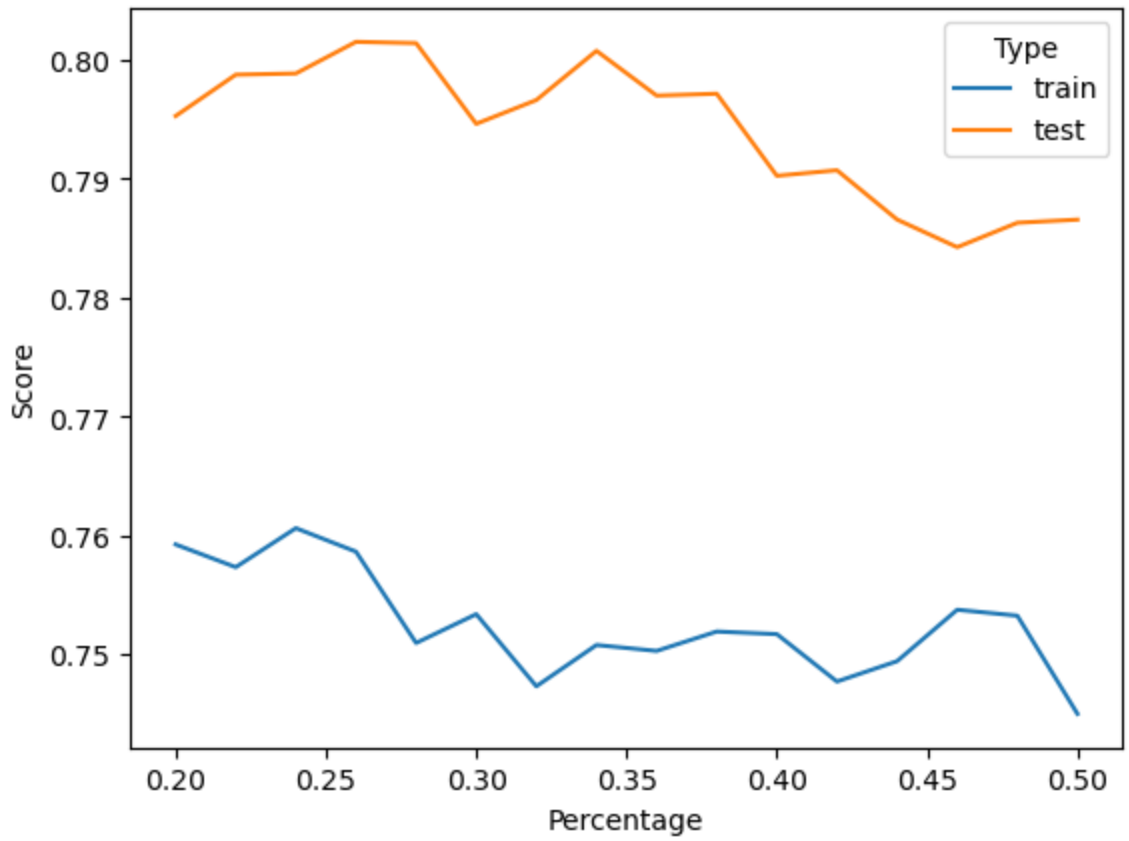
In [24]:
```
tr = []
te = []

for i in range(1,30):
    svd = TruncatedSVD(n_components=i, n_iter=7, random_state=42)
    X = svd.fit_transform(tfidf_m)
    y = df_tr['Category']
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_sta
    kmeans = KMeans(n_clusters = 5, random_state = 42).fit(X_train)

    tr.append(find_best_score(y_train, kmeans.labels_)[0])
    te.append(find_best_score(y_test, kmeans.predict(X_test))[0])

t = tr + te
c = ['train'] * 29 + ['test'] * 29
n = list(range(1,30)) + list(range(1,30))
df_viz = pd.DataFrame(data = list(zip(t, c, n)), columns = ['Score', 'Type', 'Components

sns.lineplot(x = 'Components', y = 'Score', hue = 'Type', data = df_viz)
plt.show()
```



It seems like 5 components is our best choice as afterwards, performance is either equivalent to it or decreases

Next to further imrpove model performance we will test the optimal train test split percentage.

In [25]:
```
tr = []
te = []

# [0.20, 0.22, 0.24, 0.26, 0.28, 0.30, 0.32, 0.34, 0.36, 0.38, 0.40, 0.42, 0.44, 0.46, 0
r = [0.2 + (i * 0.02) for i in range(0,16)]

for i in r:
    svd = TruncatedSVD(n_components=5, n_iter=7, random_state=42)
    X = svd.fit_transform(tfidf_m)
    y = df_tr['Category']
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=i, random_state=
    kmeans = KMeans(n_clusters = 5, random_state = 42).fit(X_train)

    tr.append(find_best_score(y_train, kmeans.labels_)[0])
    te.append(find_best_score(y_test, kmeans.predict(X_test))[0])

t = tr + te
c = ['train'] * 16 + ['test'] * 16
n = r + r
df_viz = pd.DataFrame(data = list(zip(t, c, n)), columns = ['Score', 'Type', 'Percentage

sns.lineplot(x = 'Percentage', y = 'Score', hue = 'Type', data = df_viz)
plt.show()
```



From the table, 0.26 is the best test_size to work with.

In [26]:
```
svd = TruncatedSVD(n_components=5, n_iter=7, random_state=42)
X = svd.fit_transform(tfidf_m)
y = df_tr['Category']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.26, random_state=4
kmeans = KMeans(n_clusters = 5, random_state = 42).fit(X_train)

print('Train Score and Labels:', find_best_score(y_train, kmeans.labels_)[0])
print('Test Score and Labels: ', find_best_score(y_test, kmeans.predict(X_test))[0])
```

```
Train Score and Labels: 0.7586206896551724
Test Score and Labels:  0.8015463917525774
```

05 - NMF: Matrix Factorization

Now we will redo our test but with NMF instead. This is so that we can test another feature extraction method to see how it compares to our SVD results. We will be utilizing sklearn's NMF library for the purposes of this section, and similarily will be starting with 5 components.

Sklearn library for NMF:

- https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.NMF.html

In [27]:
```
nmf = NMF(n_components=5, init='random', solver = 'cd', random_state = 42)
W = nmf.fit_transform(tfidf_m)
H = nmf.components_

print(W.shape)
```

```
(1490, 5)
```

We can view our NMF matrix's W matrix, or transformed data and the H matrix, or the factorization matrix.

In [28]:
```
print('W:\n', W)
print('H:\n', H)
```

```
W:
 [[1.30572729e-03 2.00169356e-02 1.45565027e-02 3.71652253e-03
  2.17661796e-03]
 [0.00000000e+00 8.16935474e-02 0.00000000e+00 0.00000000e+00
  0.00000000e+00]
 [0.00000000e+00 4.95300663e-02 8.28000910e-03 7.75681985e-03
  9.71851382e-03]
 ...
 [1.47107638e-03 6.33482623e-02 0.00000000e+00 1.54327040e-03
  1.97792228e-03]
 [7.28241046e-05 4.38866082e-03 0.00000000e+00 7.96348519e-02
  0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 3.35851521e-02
  1.02616838e-03]]
H:
 [[0.00000000e+00 0.00000000e+00 1.38569430e-04 ... 0.00000000e+00
  4.28586397e-03 9.47839584e-04]
 [1.07403998e-03 5.67841502e-04 1.00016817e-03 ... 5.23321492e-04
  1.66562077e-04 0.00000000e+00]
 [6.67710259e-04 0.00000000e+00 0.00000000e+00 ... 0.00000000e+00
  2.07119596e-05 0.00000000e+00]
 [4.80501542e-04 0.00000000e+00 7.27356045e-04 ... 0.00000000e+00
  1.77448042e-04 0.00000000e+00]
 [1.65060508e-04 2.08273681e-02 7.16305856e-03 ... 2.68657181e-02
  2.23082826e-03 1.73991381e-02]]
```

06 - NMF: Unsupervised Model

Similar to SVD, we will utilize KMeans clustering with our NMF results.

In [29]:
```
X_train, X_test, y_train, y_test = train_test_split(W, df_tr.Category, test_size = 0.33,

kmeans = KMeans(n_clusters = 5, random_state = 42).fit(X_train)
```

In [30]:
```
print('Train Score:', find_best_score(y_train, kmeans.labels_)[0])
print('Test Score: ', find_best_score(y_test, kmeans.predict(X_test))[0])
```

```
Train Score: 0.7885771543086172
Test Score:  0.8272357723577236
```

We will also be testing the results on the testing dataset.

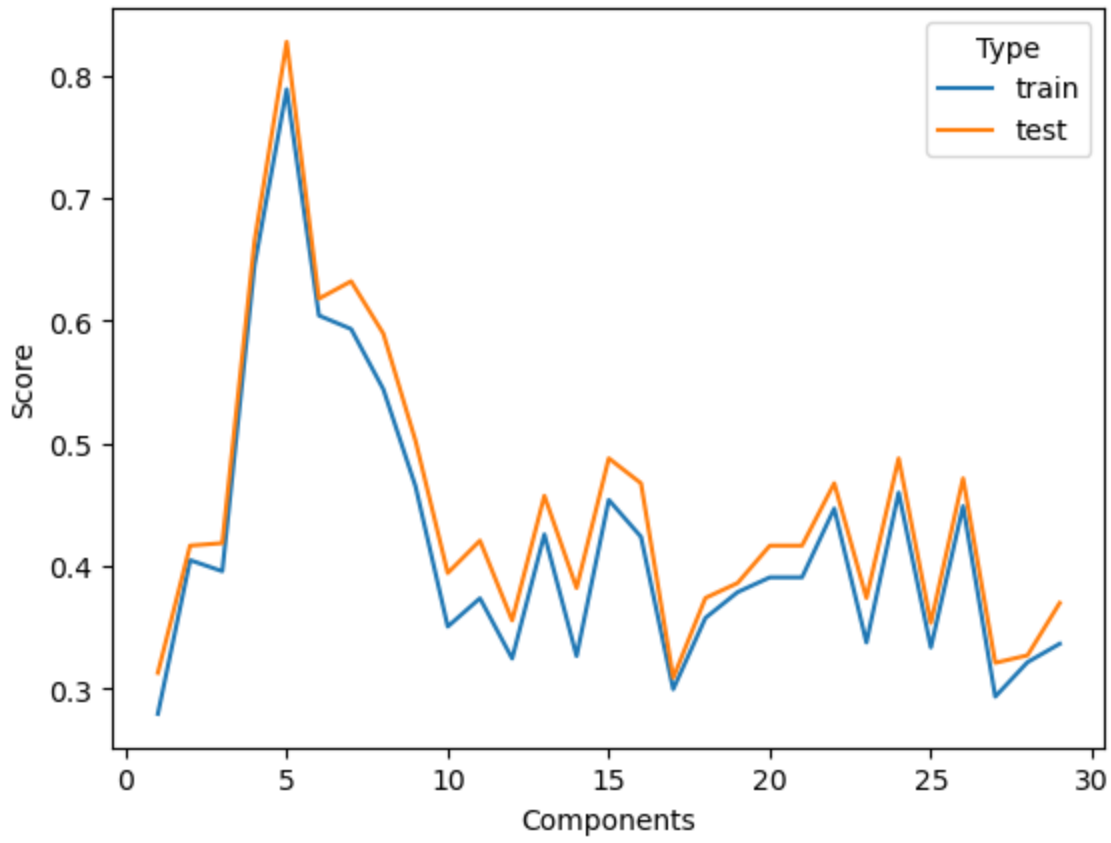
In [31]:
```
tfidf_te = TfidfVectorizer(strip_accents = 'ascii', stop_words = 'english', lowercase =
                           token_pattern = r'(?u)\b[A-Za-z]+\b', min_df = 1).fit_transform(


nmf_te = NMF(n_components=5, init='random', solver = 'cd', random_state = 42).fit_transf


bl = { 0:'business', 2:'entertainment', 4:'politics', 1:'sport', 3:'tech'}

sol_te = [  bl[i] for i in kmeans.predict(nmf_te) ]

df_solution = df_te[['ArticleId']]
df_solution['Category'] = sol_te

# df_solution.to_csv('sol.csv', index = False)
```

Given NMF's attrocious score of 0.64, we will attempt some optimization in an attempt to bump it up to a

reasonable level. Like above, we will first test different number of components in our NMF.

In [32]:
```
%%capture

tr = []
te = []

for i in range(1,30):
    nmf = NMF(n_components=i, init='random', solver = 'cd', random_state = 42)
    X = nmf.fit_transform(tfidf_m)
    y = df_tr['Category']
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_sta
    kmeans = KMeans(n_clusters = 5, random_state = 42).fit(X_train)

    tr.append(find_best_score(y_train, kmeans.labels_)[0])
    te.append(find_best_score(y_test, kmeans.predict(X_test))[0])
```

In [33]:
```
t = tr + te
c = ['train'] * 29 + ['test'] * 29
n = list(range(1,30)) + list(range(1,30))
df_viz = pd.DataFrame(data = list(zip(t, c, n)), columns = ['Score', 'Type', 'Components

sns.lineplot(x = 'Components', y = 'Score', hue = 'Type', data = df_viz)
plt.show()
```

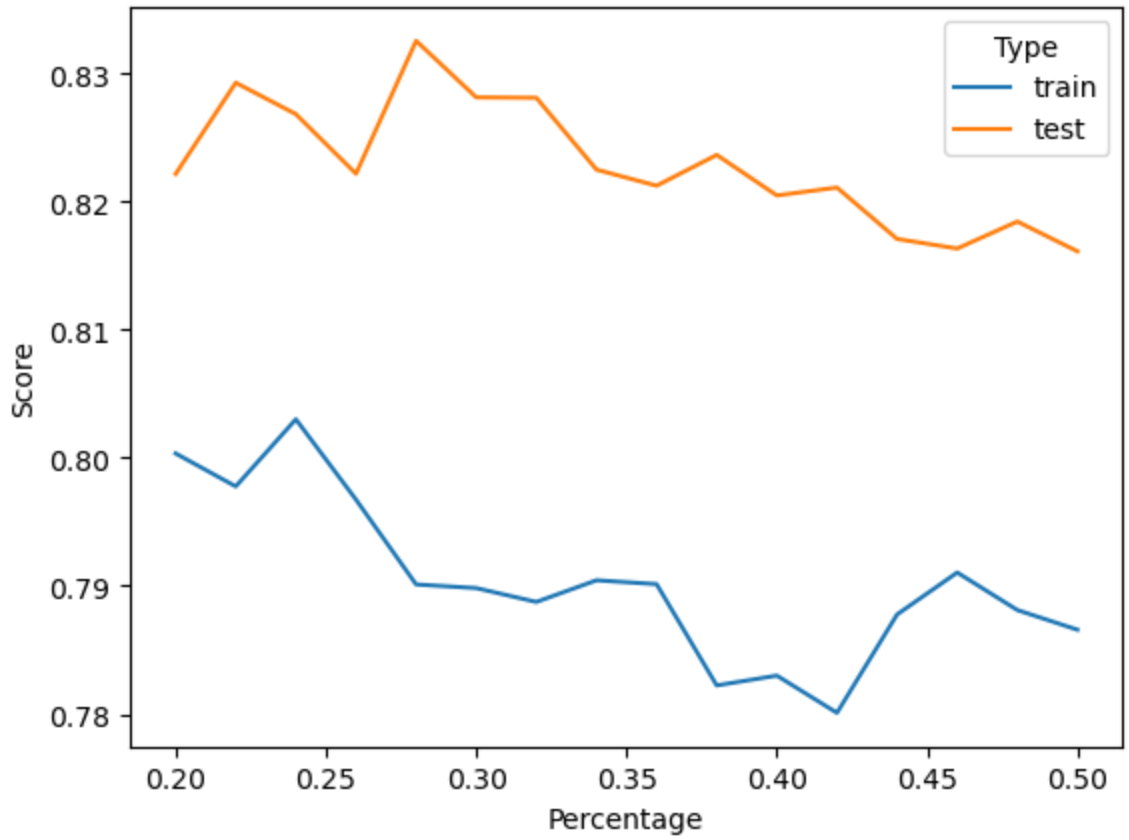


5 components yet again proved to have the best performance, and now we will also test the optimal train test split.

In [34]:
```
%%capture

tr = []
te = []

# [0.20, 0.22, 0.24, 0.26, 0.28, 0.30, 0.32, 0.34, 0.36, 0.38, 0.40, 0.42, 0.44, 0.46, 0
r = [0.2 + (i * 0.02) for i in range(0,16)]
```

```
for i in r:
    nmf = NMF(n_components=5, init='random', solver = 'cd', random_state = 42)
    X = nmf.fit_transform(tfidf_m)
    y = df_tr['Category']
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=i, random_state=
    kmeans = KMeans(n_clusters = 5, random_state = 42).fit(X_train)

    tr.append(find_best_score(y_train, kmeans.labels_)[0])
    te.append(find_best_score(y_test, kmeans.predict(X_test))[0])
```

In [35]:
```
t = tr + te
c = ['train'] * 16 + ['test'] * 16
n = r + r
df_viz = pd.DataFrame(data = list(zip(t, c, n)), columns = ['Score', 'Type', 'Percentage

sns.lineplot(x = 'Percentage', y = 'Score', hue = 'Type', data = df_viz)
plt.show()
```



From the plot, the optimal split percentage is 0.28.

In [36]:
```
nmf = NMF(n_components=5, init='random', solver = 'cd', random_state = 42)
X = nmf.fit_transform(tfidf_m)
y = df_tr['Category']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.28, random_state=4
kmeans = KMeans(n_clusters = 5, random_state = 42).fit(X_train)

print('Train Score and Labels:', find_best_score(y_train, kmeans.labels_)[0])
print('Test Score and Labels: ', find_best_score(y_test, kmeans.predict(X_test))[0])
```

```
Train Score and Labels: 0.7901119402985075
Test Score and Labels:  0.8325358851674641
```

07 - SVD: Supervised Model

Now we can compare our unsupervised model with a supervised variant. To begin, we will use a

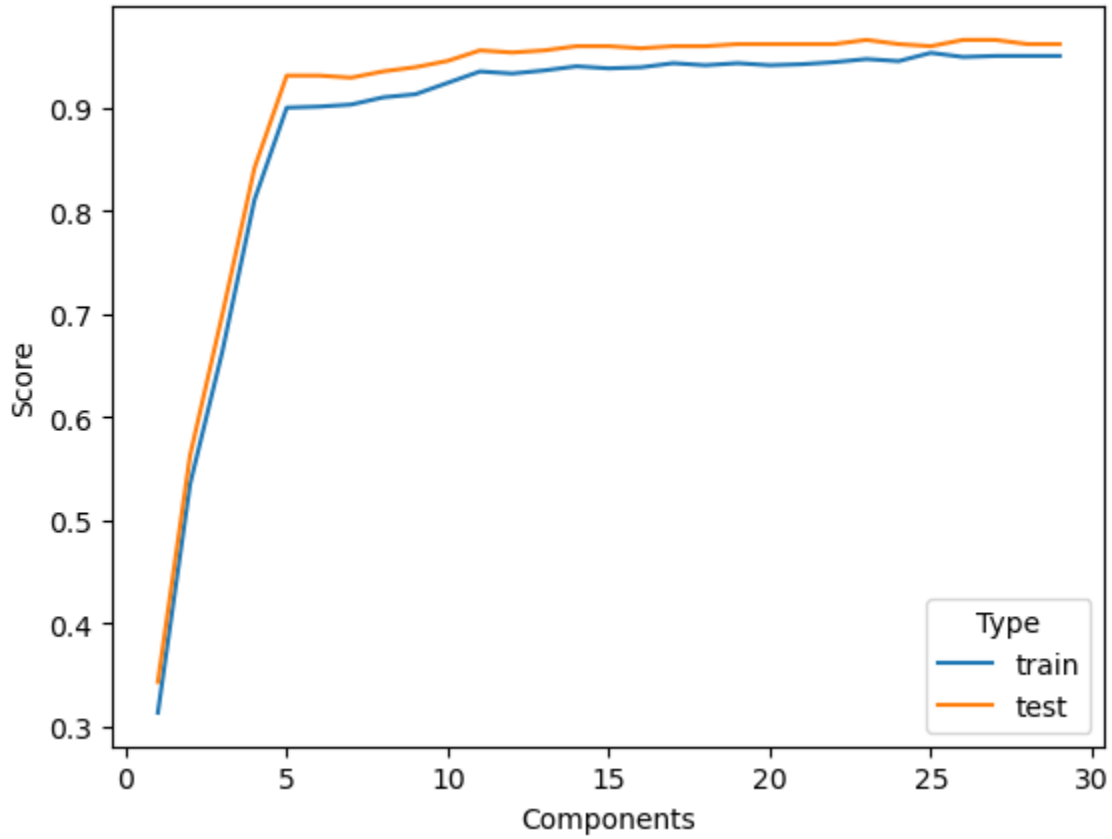
LogisticRegression model. Once we split our SVD matrix and target variable column into train and test portions, we test our results and receive a relatively decent accuracy score of 0.92 for our test score.

In [37]:
```
X = svd_m
y = df_tr['Category']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=4

lr = LogisticRegression(random_state = 42).fit(X_train, y_train)
print('LR Train Score: ', accuracy_score(y_train, lr.predict(X_train)))
print('LR Test Score:  ', accuracy_score(y_test, lr.predict(X_test)))
```

```
LR Train Score:  0.8997995991983968
LR Test Score:   0.9308943089430894
```

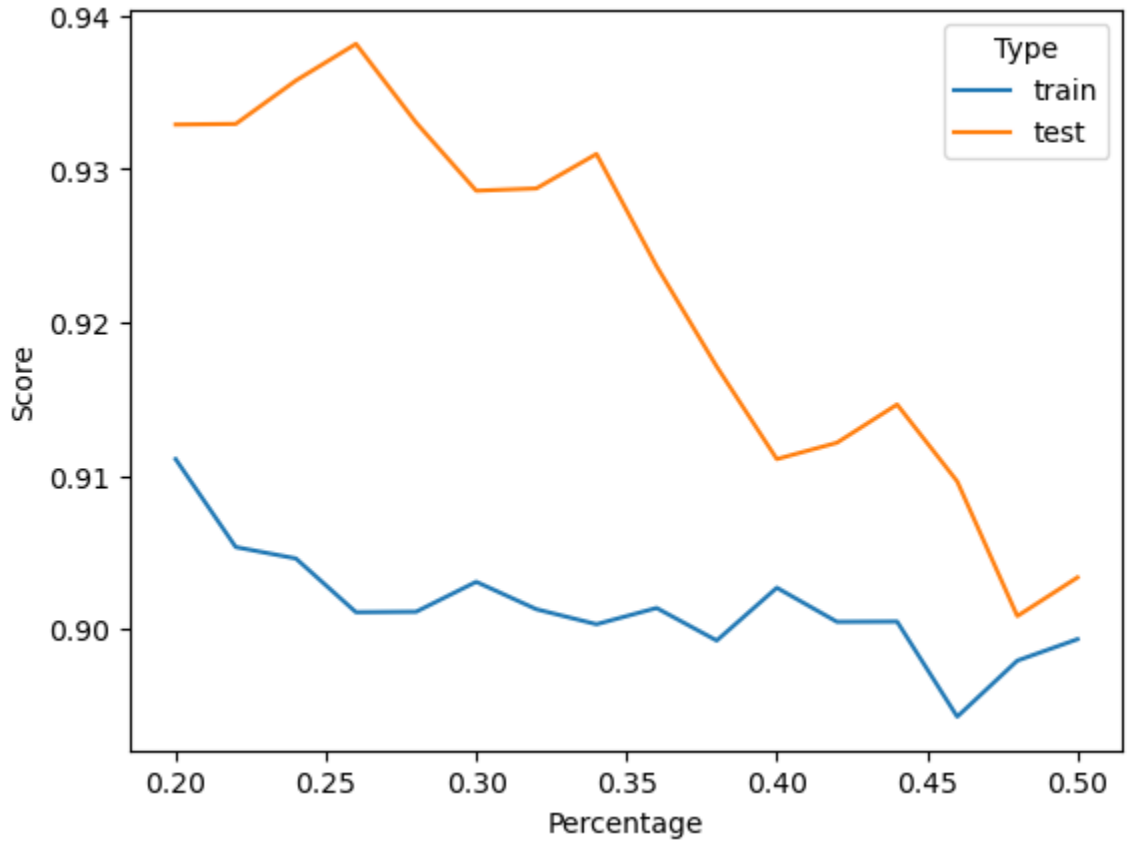
However as this under the default assumption that 5 components was optimal, we can try and optimize our performance by testing other potential component numbers, mainly from 1 component to 30. Here we will attempt the same code but view how our accuracy changes as we change the number of components. Through the plot, it is evident that the performance sharply peaks at 5 components, though dips a bit and steadily increases. It seems that 12 components seems like the best threshold to keep our SVD in regards to number of components versus accuracy.

In [38]:
```
tr = []
te = []

for i in range(1,30):
    svd = TruncatedSVD(n_components=i, n_iter=7, random_state=42)
    X = svd.fit_transform(tfidf_m)
    y = df_tr['Category']
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_sta

    lr = LogisticRegression(random_state = 42).fit(X_train, y_train)
    tr.append(accuracy_score(y_train, lr.predict(X_train)))
    te.append(accuracy_score(y_test, lr.predict(X_test)))


t = tr + te
c = ['train'] * 29 + ['test'] * 29
n = list(range(1,30)) + list(range(1,30))
df_viz = pd.DataFrame(data = list(zip(t, c, n)), columns = ['Score', 'Type', 'Components

sns.lineplot(x = 'Components', y = 'Score', hue = 'Type', data = df_viz)
plt.show()
```



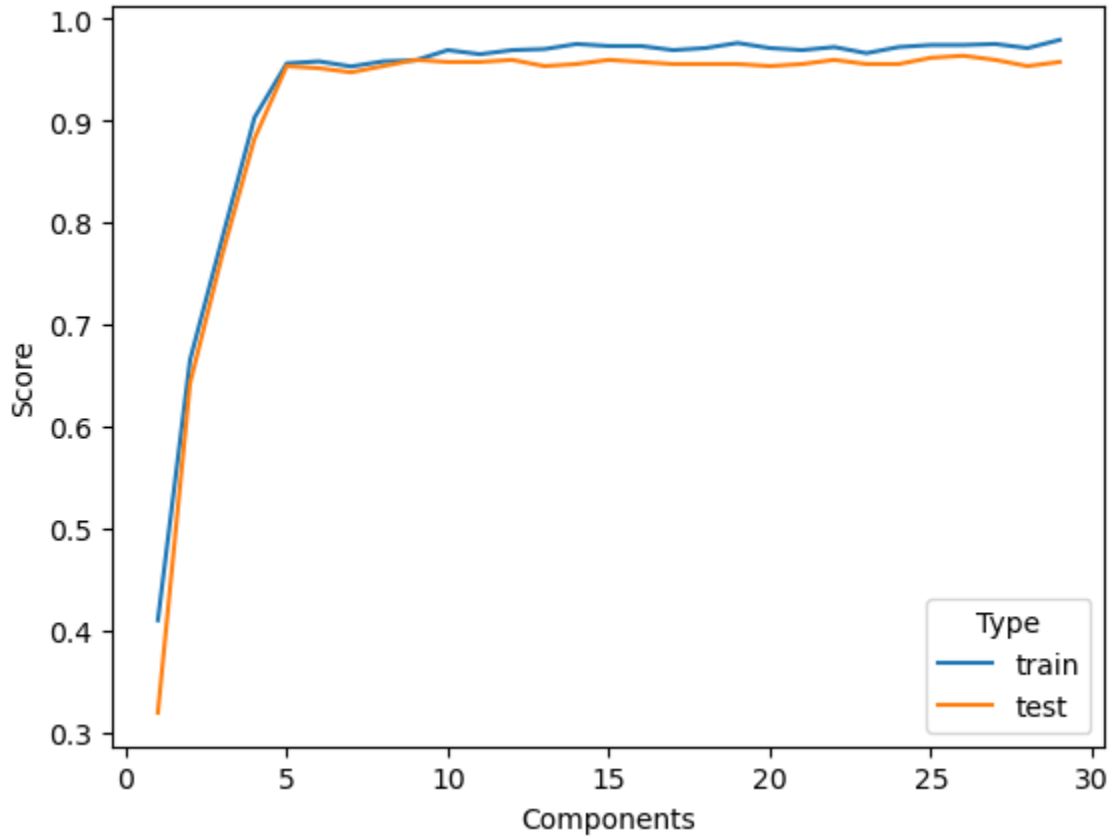
In [39]:

```
tr = []
te = []

# [0.20, 0.22, 0.24, 0.26, 0.28, 0.30, 0.32, 0.34, 0.36, 0.38, 0.40, 0.42, 0.44, 0.46, 0
r = [0.2 + (i * 0.02) for i in range(0,16)]

for i in r:
    svd = TruncatedSVD(n_components=5, n_iter=7, random_state=42)
    X = svd.fit_transform(tfidf_m)
    y = df_tr['Category']
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=i, random_state=

    lr = LogisticRegression(random_state = 42).fit(X_train, y_train)
    tr.append(accuracy_score(y_train, lr.predict(X_train)))
    te.append(accuracy_score(y_test, lr.predict(X_test)))


t = tr + te
c = ['train'] * 16 + ['test'] * 16
n = r + r
df_viz = pd.DataFrame(data = list(zip(t, c, n)), columns = ['Score', 'Type', 'Percentage

sns.lineplot(x = 'Percentage', y = 'Score', hue = 'Type', data = df_viz)
plt.show()
```



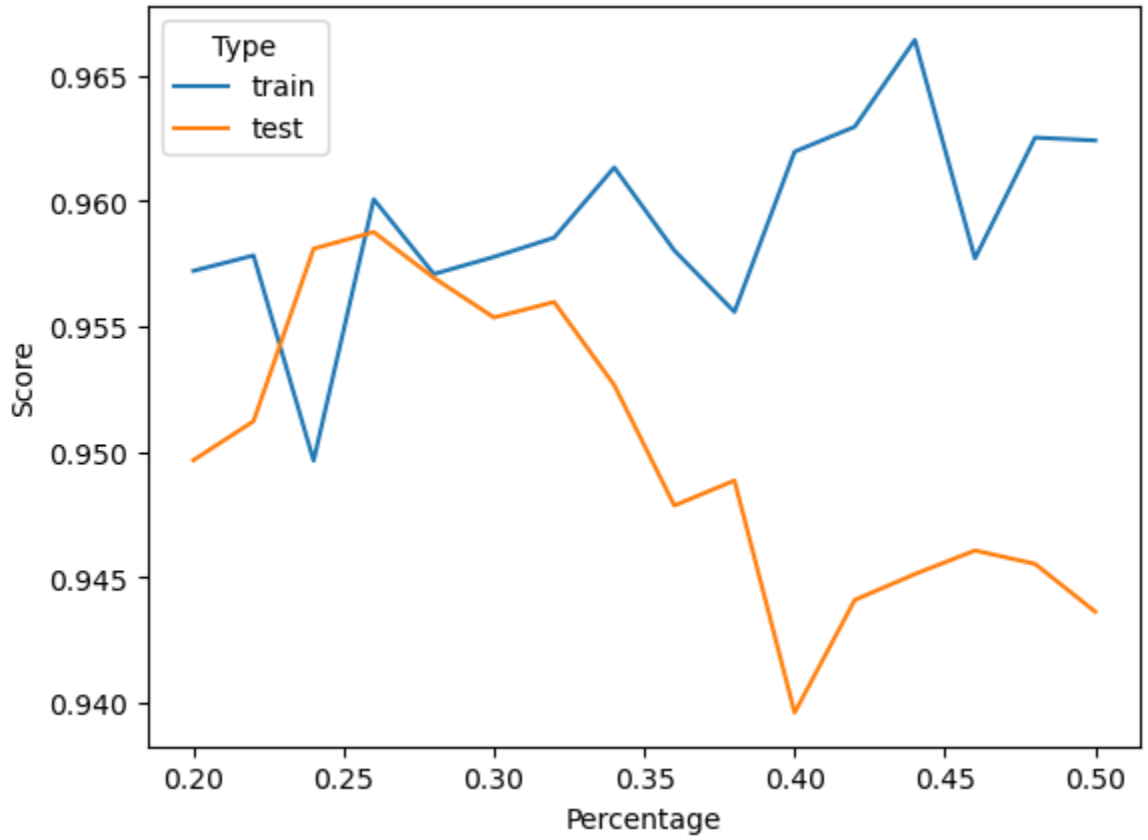
Now we will undergo the same test but with the RandomForestClassifier instead of LogisticRegression. The RandomForestClassifier proves to be a bit more powerful, as at has a relatively similar curve shape but has roughly the same performance at 5 components as LogisticRegression did with 12 components.

In [40]:
```
tr = []
te = []

for i in range(1,30):
    svd = TruncatedSVD(n_components=i, n_iter=7, random_state=42)
    X = svd.fit_transform(tfidf_m)
    y = df_tr['Category']
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_sta

    rf = RandomForestClassifier(max_depth=5, random_state = 42).fit(X_train, y_train)
    tr.append(accuracy_score(y_train, rf.predict(X_train)))
    te.append(accuracy_score(y_test, rf.predict(X_test)))

t = tr + te
c = ['train'] * 29 + ['test'] * 29
n = list(range(1,30)) + list(range(1,30))
df_viz = pd.DataFrame(data = list(zip(t, c, n)), columns = ['Score', 'Type', 'Components

sns.lineplot(x = 'Components', y = 'Score', hue = 'Type', data = df_viz)
plt.show()
```



In [41]:

```
tr = []
te = []

# [0.20, 0.22, 0.24, 0.26, 0.28, 0.30, 0.32, 0.34, 0.36, 0.38, 0.40, 0.42, 0.44, 0.46, 0
r = [0.2 + (i * 0.02) for i in range(0,16)]

for i in r:
    svd = TruncatedSVD(n_components=5, n_iter=7, random_state=42)
    X = svd.fit_transform(tfidf_m)
    y = df_tr['Category']
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=i, random_state=

    rf = RandomForestClassifier(max_depth=5, random_state = 42).fit(X_train, y_train)
    tr.append(accuracy_score(y_train, rf.predict(X_train)))
    te.append(accuracy_score(y_test, rf.predict(X_test)))


t = tr + te
c = ['train'] * 16 + ['test'] * 16
n = r + r
df_viz = pd.DataFrame(data = list(zip(t, c, n)), columns = ['Score', 'Type', 'Percentage

sns.lineplot(x = 'Percentage', y = 'Score', hue = 'Type', data = df_viz)
plt.show()
```



Thus so far, our best performance is with 5 components with 0.26 split when using a random forest classifier.

In [42]:
```
svd = TruncatedSVD(n_components=5, n_iter=7, random_state=42)
X = svd.fit_transform(tfidf_m)
y = df_tr['Category']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.26, random_state=4

rf = RandomForestClassifier(max_depth=5, random_state = 42).fit(X_train, y_train)

print('LR Train Score: ', accuracy_score(y_train, rf.predict(X_train)))
print('LR Test Score:  ', accuracy_score(y_test, rf.predict(X_test)))
```

```
LR Train Score:  0.9600725952813067
LR Test Score:   0.9587628865979382
```

08 - NMF: Supervised Model

With our initial LogisticRegression test, we receive a pitiful 0.66 accuracy score. This is clear that some optimization should be done.

In [43]:
```
X = W
y = df_tr['Category']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=4

lr = LogisticRegression(random_state = 42).fit(X_train, y_train)
print('LR Train Score: ', accuracy_score(y_train, lr.predict(X_train)))
print('LR Test Score:  ', accuracy_score(y_test, lr.predict(X_test)))
```

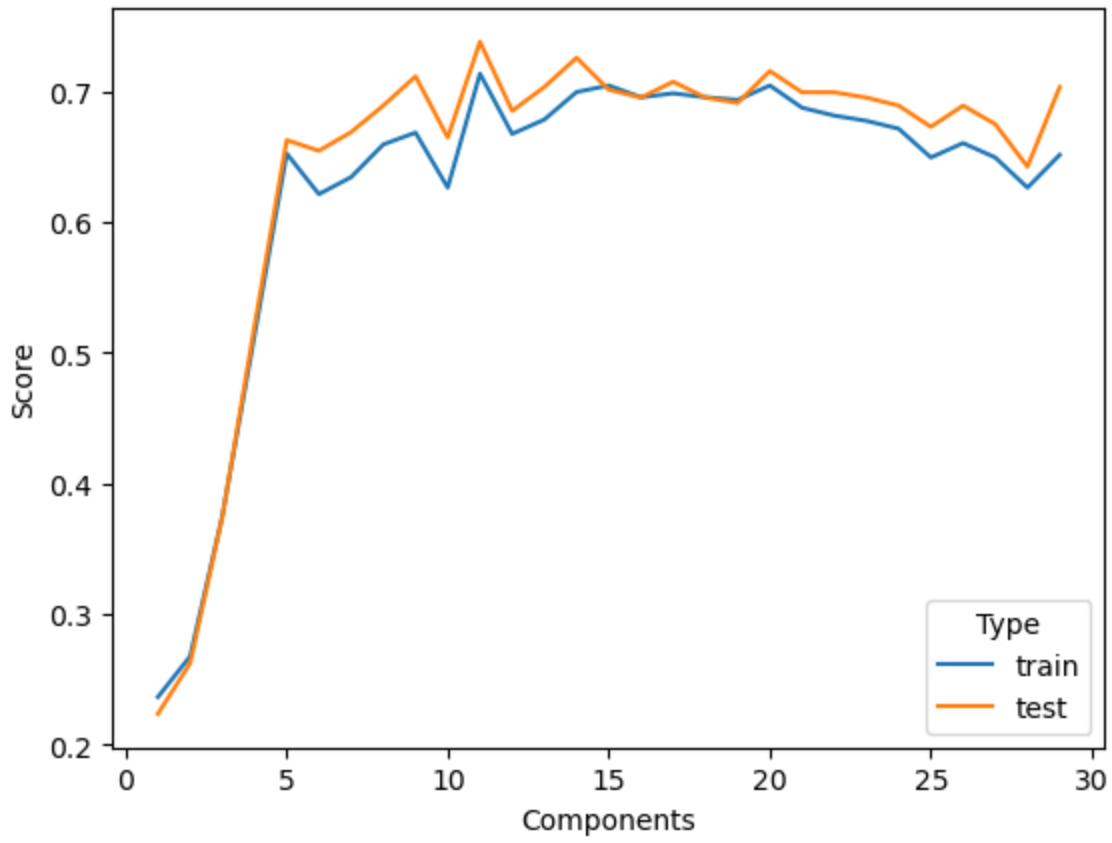
```
LR Train Score:  0.6523046092184369
LR Test Score:   0.6626016260162602
```

Like above, we will test our components from 1 to 30, though our resulting graph is a bit too spiky to be confident of our findings. However the overall result did not improve by much, with the best test score

peaking at around 7.25.

In [44]:
```
%%capture

tr = []
te = []

for i in range(1,30):
    nmf = NMF(n_components=i, init='random', solver = 'cd', random_state = 42)
    X = nmf.fit_transform(tfidf_m)
    y = df_tr['Category']
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_sta

    lr = LogisticRegression(random_state = 42).fit(X_train, y_train)
    tr.append(accuracy_score(y_train, lr.predict(X_train)))
    te.append(accuracy_score(y_test, lr.predict(X_test)))
```

In [45]:
```
t = tr + te
c = ['train'] * 29 + ['test'] * 29
n = list(range(1,30)) + list(range(1,30))
df_viz = pd.DataFrame(data = list(zip(t, c, n)), columns = ['Score', 'Type', 'Components

sns.lineplot(x = 'Components', y = 'Score', hue = 'Type', data = df_viz)
plt.show()
```

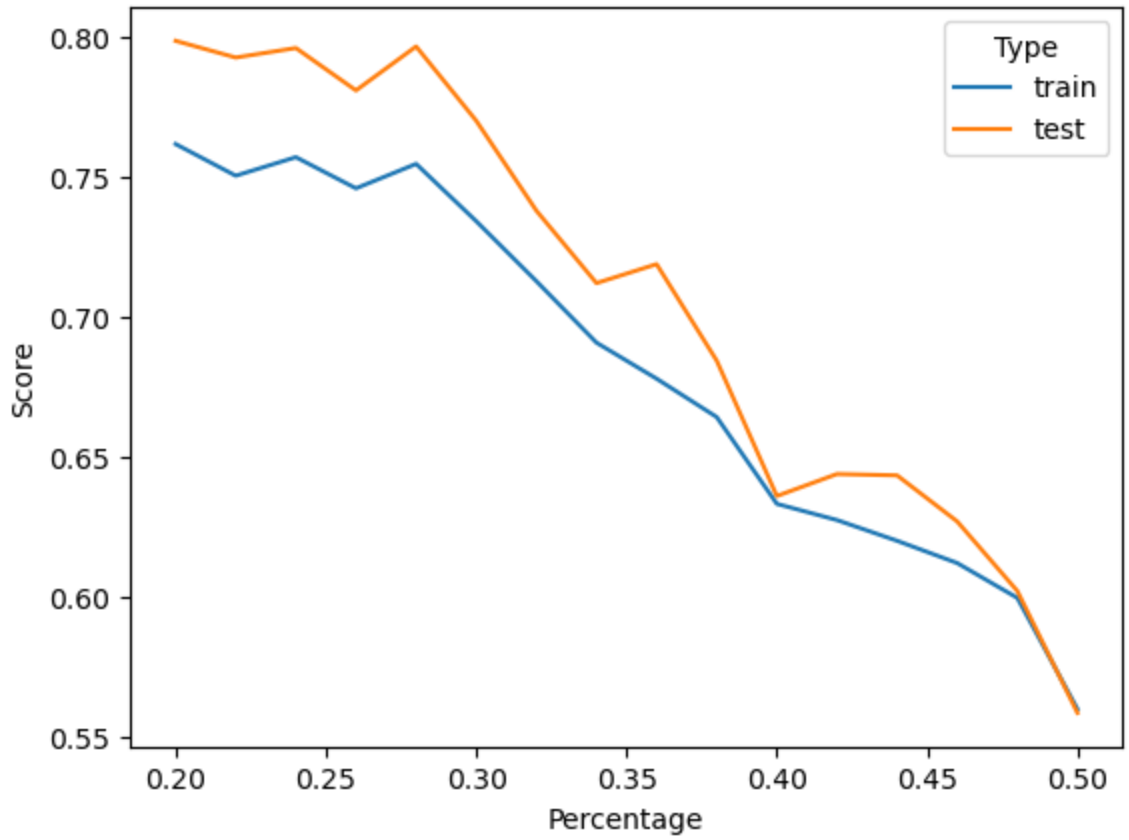


Of note is that 11 components provided us the best score.

In [46]:
```
%%capture

tr = []
te = []

# [0.20, 0.22, 0.24, 0.26, 0.28, 0.30, 0.32, 0.34, 0.36, 0.38, 0.40, 0.42, 0.44, 0.46, 0
r = [0.2 + (i * 0.02) for i in range(0,16)]

for i in r:
```

```
    nmf = NMF(n_components=11, init='random', solver = 'cd', random_state = 42)
    X = nmf.fit_transform(tfidf_m)
    y = df_tr['Category']
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=i, random_state=

    lr = LogisticRegression(random_state = 42).fit(X_train, y_train)
    tr.append(accuracy_score(y_train, lr.predict(X_train)))
    te.append(accuracy_score(y_test, lr.predict(X_test)))
```

In [47]:
```
t = tr + te
c = ['train'] * 16 + ['test'] * 16
n = r + r
df_viz = pd.DataFrame(data = list(zip(t, c, n)), columns = ['Score', 'Type', 'Percentage

sns.lineplot(x = 'Percentage', y = 'Score', hue = 'Type', data = df_viz)
plt.show()
```



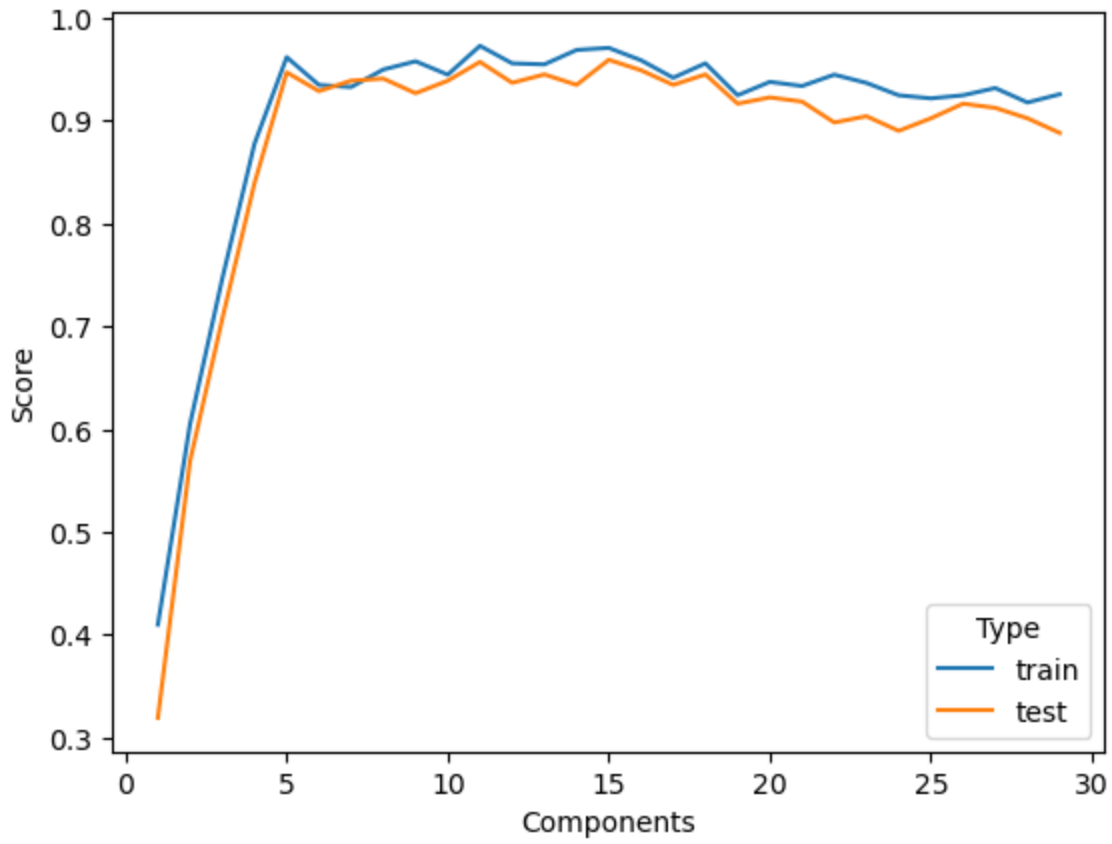
The best split was from a test size of 0.2.

On the other hand, the random forest classifier had vastly superior performance; one thing we can assume is that the NMF data and our target variables do not have much of a linear regression, as this is an area that random forests excels in. However the lineplot is still quite spiky, at least compared to our SVD plot, so the reliability of this is still a bit questionable. However, its best performance at 10 components is comparable to SVD's performance.

In [48]:
```
%%capture

tr = []
te = []

for i in range(1,30):
    nmf = NMF(n_components=i, init='random', solver = 'cd', random_state = 42)
    X = nmf.fit_transform(tfidf_m)
    y = df_tr['Category']
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_sta
```

```
    rf = RandomForestClassifier(max_depth=5, random_state = 42).fit(X_train, y_train)
    tr.append(accuracy_score(y_train, rf.predict(X_train)))
    te.append(accuracy_score(y_test, rf.predict(X_test)))
```

In [49]:
```
t = tr + te
c = ['train'] * 29 + ['test'] * 29
n = list(range(1,30)) + list(range(1,30))
df_viz = pd.DataFrame(data = list(zip(t, c, n)), columns = ['Score', 'Type', 'Components

sns.lineplot(x = 'Components', y = 'Score', hue = 'Type', data = df_viz)
plt.show()
```

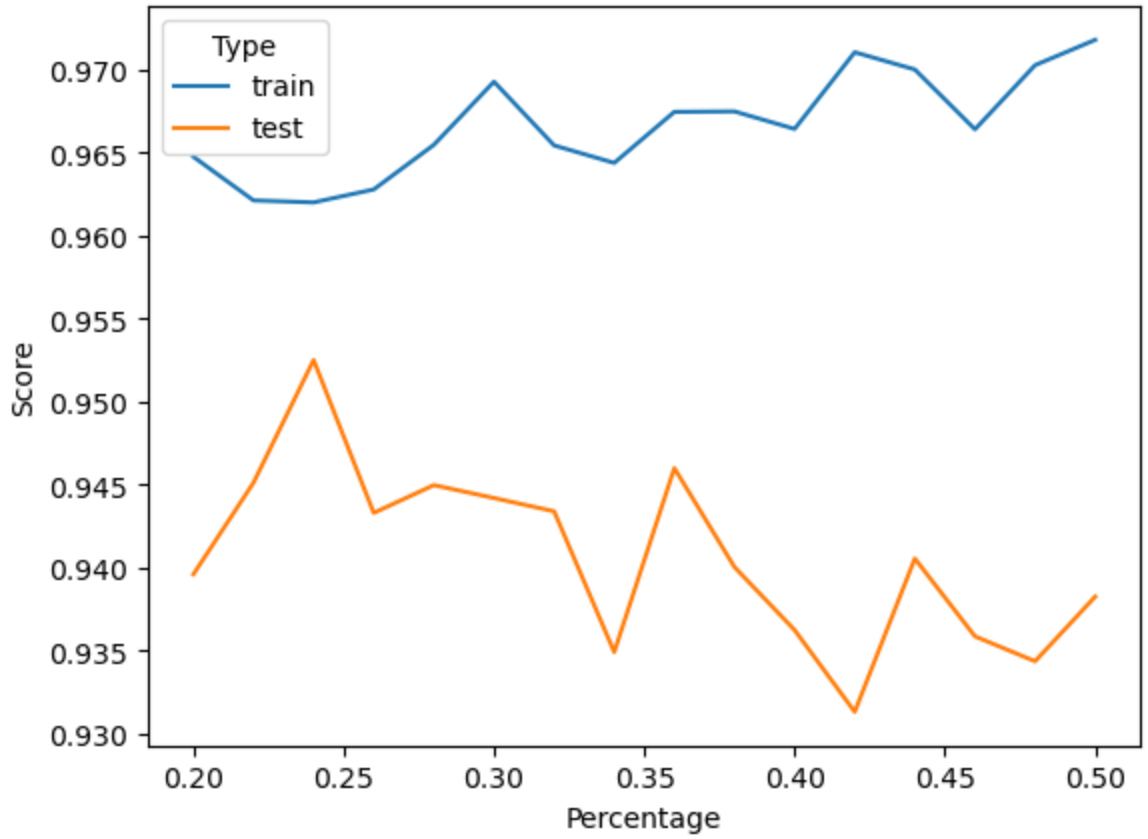


As the performance is rather stable after 5 components, we will choose 5 due to its smallest cost and best performance.

In [50]:
```
%%capture

tr = []
te = []

# [0.20, 0.22, 0.24, 0.26, 0.28, 0.30, 0.32, 0.34, 0.36, 0.38, 0.40, 0.42, 0.44, 0.46, 0
r = [0.2 + (i * 0.02) for i in range(0,16)]

for i in r:
    nmf = NMF(n_components=5, init='random', solver = 'cd', random_state = 42)
    X = nmf.fit_transform(tfidf_m)
    y = df_tr['Category']
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=i, random_state=

    rf = RandomForestClassifier(max_depth=5, random_state = 42).fit(X_train, y_train)
    tr.append(accuracy_score(y_train, rf.predict(X_train)))
    te.append(accuracy_score(y_test, rf.predict(X_test)))
```

In [51]:
```
t = tr + te
c = ['train'] * 16 + ['test'] * 16
```

```
n = r + r
df_viz = pd.DataFrame(data = list(zip(t, c, n)), columns = ['Score', 'Type', 'Percentage
sns.lineplot(x = 'Percentage', y = 'Score', hue = 'Type', data = df_viz)
plt.show()
```



While the results aren't very stable, the best percentage for test split is 0.24.

In [52]:
```
nmf = NMF(n_components=5, init='random', solver = 'cd', random_state = 42)
X = nmf.fit_transform(tfidf_m)
y = df_tr['Category']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.24, random_state=4

rf = RandomForestClassifier(max_depth=5, random_state = 42).fit(X_train, y_train)

print('LR Train Score: ', accuracy_score(y_train, rf.predict(X_train)))
print('LR Test Score:  ', accuracy_score(y_test, rf.predict(X_test)))
```

```
LR Train Score:  0.9620141342756183
LR Test Score:   0.952513966480447
```

09 - BBC News Classification Conclusion

While both supervised and unsupervised learning had good performances, when working with our training data and the labels, the supervised learning had overall better accuracy scores. I believe this is due to the existence of pre-existing labels to train the model with, that supervised learning had a massive advantage. When looking at the train-test split plots as well, it seems that supervised learning methods are also more data efficient as well, although while they start off strong, their performance greatly decreases the bigger the test-size is, which is most likely due to overfitting. On the other hand, unsupervised learning suffers less from having a high test size, assumedly because unsupervised methods can enjoy having more data to learn patterns from.

Additional note is that in this scenario where the problem statement is to assign existing labels to data inputs, unsupervised learning scoring is at a disadvantage as while clustering is very efficient at grouping our articles into groups, it suffers from accurately assigning a specific label to those groups when there is no proper feedback on it.

In the end, while both SVD and NMP had similar performances at their peak, in this scenario I would prefer to use SVD as the resulting performance had a much more stable line on the components versus accuracy plots. Thus my final model will be a SVD with 5 components with a supervised Random Forest Classifier.

10 - Predictions on Test Data

To get the predictions, we will change our test data to fit the same format as our training data, and then predict its results using our trained Random Forest Classifier.

In [53]:
```
df_tr = df_tr[df_tr['Text'].str.len() <= 6500]

tfidf = TfidfVectorizer(strip_accents = 'ascii', stop_words = 'english', lowercase = 'Tr
                        token_pattern = r'(?u)\b[A-Za-z]+\b', min_df = 1)
tfidf_m = tfidf.fit_transform(df_tr['Text'])
svd = TruncatedSVD(n_components=5, n_iter=7, random_state=42)

X = svd.fit_transform(tfidf_m)
y = df_tr['Category']

rf = RandomForestClassifier(max_depth=5, random_state = 42).fit(X, y)
```

In [54]:
```
tfidf_te = TfidfVectorizer(strip_accents = 'ascii', stop_words = 'english', lowercase =
                        token_pattern = r'(?u)\b[A-Za-z]+\b', min_df = 1)
tfidf_te_m = tfidf.fit_transform(df_te['Text'])
svd_te = TruncatedSVD(n_components=5, n_iter=7, random_state=42)

X_te = svd_te.fit_transform(tfidf_te_m)

print(tfidf_te_m.shape)
print(X_te.shape)
print(rf.predict(X_te))
```

```
(735, 17343)
(735, 5)
['entertainment' 'sport' 'tech' 'entertainment' 'tech' 'tech' 'politics'
 'politics' 'business' 'entertainment' 'entertainment' 'sport' 'business'
 'sport' 'business' 'tech' 'politics' 'sport' 'business' 'sport'
 'politics' 'business' 'tech' 'entertainment' 'politics' 'tech'
 'entertainment' 'tech' 'tech' 'entertainment' 'politics' 'sport'
 'entertainment' 'entertainment' 'tech' 'tech' 'tech' 'entertainment'
 'business' 'entertainment' 'politics' 'politics' 'business' 'sport'
 'entertainment' 'sport' 'business' 'entertainment' 'politics'
 'entertainment' 'politics' 'entertainment' 'politics' 'entertainment'
 'sport' 'politics' 'sport' 'business' 'tech' 'sport' 'tech' 'business'
 'sport' 'business' 'entertainment' 'business' 'tech' 'sport' 'tech'
 'tech' 'sport' 'entertainment' 'entertainment' 'politics' 'sport'
 'entertainment' 'sport' 'sport' 'sport' 'business' 'politics' 'tech'
 'business' 'tech' 'entertainment' 'business' 'entertainment' 'business'
 'entertainment' 'sport' 'politics' 'politics' 'tech' 'sport' 'tech'
 'business' 'tech' 'tech' 'tech' 'tech' 'politics' 'tech' 'politics'
 'business' 'entertainment' 'tech' 'business' 'entertainment' 'business'
 'business' 'entertainment' 'entertainment' 'business' 'tech' 'politics'
 'tech' 'politics' 'entertainment' 'politics' 'entertainment'
 'entertainment' 'entertainment' 'business' 'sport' 'tech' 'tech'
 'entertainment' 'business' 'entertainment' 'business' 'entertainment'
```

'politics' 'business' 'sport' 'entertainment' 'entertainment' 'politics'
'tech' 'business' 'tech' 'entertainment' 'sport' 'tech' 'business'
'entertainment' 'entertainment' 'tech' 'business' 'business' 'tech'
'entertainment' 'sport' 'politics' 'business' 'tech' 'tech' 'tech'
'entertainment' 'business' 'tech' 'entertainment' 'sport' 'entertainment'
'sport' 'entertainment' 'tech' 'business' 'sport' 'sport' 'politics'
'entertainment' 'politics' 'entertainment' 'entertainment' 'sport'
'politics' 'sport' 'entertainment' 'entertainment' 'sport' 'business'
'politics' 'business' 'politics' 'entertainment' 'sport' 'business'
'entertainment' 'politics' 'entertainment' 'politics' 'entertainment'
'business' 'tech' 'sport' 'sport' 'entertainment' 'sport' 'politics'
'entertainment' 'tech' 'politics' 'entertainment' 'business'
'entertainment' 'entertainment' 'tech' 'sport' 'entertainment' 'tech'
'business' 'business' 'tech' 'business' 'tech' 'sport' 'politics'
'business' 'tech' 'business' 'tech' 'business' 'tech' 'politics' 'sport'
'business' 'entertainment' 'business' 'politics' 'sport' 'politics'
'tech' 'politics' 'politics' 'politics' 'politics' 'entertainment'
'entertainment' 'business' 'business' 'tech' 'tech' 'entertainment'
'tech' 'sport' 'entertainment' 'politics' 'entertainment' 'politics'
'entertainment' 'entertainment' 'tech' 'sport' 'politics' 'sport' 'sport'
'business' 'sport' 'tech' 'entertainment' 'sport' 'tech' 'tech'
'entertainment' 'business' 'tech' 'politics' 'tech' 'entertainment'
'tech' 'entertainment' 'tech' 'entertainment' 'tech' 'business'
'entertainment' 'entertainment' 'business' 'politics' 'entertainment'
'tech' 'tech' 'sport' 'tech' 'tech' 'business' 'entertainment' 'tech'
'sport' 'politics' 'business' 'entertainment' 'entertainment' 'politics'
'tech' 'business' 'politics' 'entertainment' 'tech' 'tech' 'sport'
'business' 'tech' 'entertainment' 'sport' 'politics' 'tech' 'business'
'entertainment' 'sport' 'politics' 'business' 'politics' 'entertainment'
'business' 'tech' 'entertainment' 'tech' 'sport' 'business' 'politics'
'entertainment' 'business' 'politics' 'entertainment' 'business'
'entertainment' 'sport' 'politics' 'sport' 'sport' 'business'
'entertainment' 'sport' 'business' 'politics' 'politics' 'tech' 'sport'
'business' 'business' 'politics' 'tech' 'sport' 'sport' 'business'
'sport' 'entertainment' 'tech' 'business' 'sport' 'entertainment'
'politics' 'sport' 'sport' 'politics' 'politics' 'entertainment'
'entertainment' 'sport' 'tech' 'politics' 'business' 'entertainment'
'sport' 'entertainment' 'politics' 'politics' 'politics' 'politics'
'sport' 'entertainment' 'entertainment' 'politics' 'tech' 'entertainment'
'tech' 'sport' 'business' 'politics' 'sport' 'sport' 'business'
'entertainment' 'entertainment' 'tech' 'business' 'business' 'tech'
'sport' 'sport' 'tech' 'sport' 'entertainment' 'politics' 'sport'
'entertainment' 'entertainment' 'politics' 'business' 'entertainment'
'sport' 'tech' 'politics' 'entertainment' 'entertainment' 'politics'
'sport' 'tech' 'business' 'politics' 'sport' 'entertainment' 'sport'
'tech' 'business' 'politics' 'entertainment' 'sport' 'tech' 'politics'
'entertainment' 'sport' 'sport' 'politics' 'sport' 'entertainment'
'politics' 'entertainment' 'business' 'entertainment' 'business' 'sport'
'business' 'tech' 'entertainment' 'politics' 'entertainment' 'tech'
'entertainment' 'entertainment' 'business' 'business' 'entertainment'
'sport' 'sport' 'sport' 'entertainment' 'politics' 'business' 'business'
'politics' 'tech' 'entertainment' 'entertainment' 'politics' 'business'
'business' 'politics' 'entertainment' 'tech' 'politics' 'tech'
'entertainment' 'politics' 'politics' 'tech' 'tech' 'politics'
'entertainment' 'sport' 'politics' 'entertainment' 'tech' 'politics'
'sport' 'entertainment' 'business' 'sport' 'business' 'politics'
'entertainment' 'sport' 'tech' 'tech' 'politics' 'entertainment' 'sport'
'politics' 'entertainment' 'entertainment' 'business' 'politics'
'entertainment' 'business' 'entertainment' 'politics' 'tech' 'politics'
'sport' 'sport' 'entertainment' 'politics' 'tech' 'politics' 'tech'
'entertainment' 'business' 'entertainment' 'tech' 'politics' 'sport'
'tech' 'business' 'entertainment' 'sport' 'politics' 'tech' 'politics'
'business' 'tech' 'politics' 'entertainment' 'entertainment' 'politics'
'sport' 'business' 'politics' 'entertainment' 'entertainment' 'sport'
'sport' 'sport' 'entertainment' 'sport' 'business' 'sport'
'entertainment' 'tech' 'sport' 'politics' 'entertainment' 'sport' 'sport'

```
'tech' 'sport' 'entertainment' 'tech' 'entertainment' 'entertainment'
'entertainment' 'politics' 'politics' 'business' 'entertainment'
'business' 'business' 'tech' 'sport' 'politics' 'business'
'entertainment' 'entertainment' 'tech' 'politics' 'entertainment'
'politics' 'entertainment' 'entertainment' 'entertainment'
'entertainment' 'sport' 'sport' 'entertainment' 'entertainment'
'business' 'politics' 'sport' 'politics' 'entertainment' 'politics'
'entertainment' 'sport' 'entertainment' 'entertainment' 'tech' 'sport'
'entertainment' 'business' 'sport' 'business' 'sport' 'tech' 'politics'
'politics' 'sport' 'sport' 'business' 'business' 'politics'
'entertainment' 'entertainment' 'tech' 'politics' 'politics'
'entertainment' 'tech' 'business' 'entertainment' 'tech' 'sport' 'tech'
'sport' 'politics' 'politics' 'tech' 'entertainment' 'tech' 'tech'
'business' 'politics' 'sport' 'tech' 'entertainment' 'tech'
'entertainment' 'entertainment' 'tech' 'politics' 'sport' 'entertainment'
'sport' 'entertainment' 'tech' 'sport' 'entertainment' 'entertainment'
'politics' 'entertainment' 'tech' 'sport' 'entertainment' 'entertainment'
'entertainment' 'business' 'tech' 'sport' 'sport' 'entertainment' 'tech'
'business' 'entertainment' 'business' 'entertainment' 'politics' 'tech'
'tech' 'politics' 'sport' 'tech' 'entertainment' 'entertainment'
'entertainment' 'sport' 'entertainment' 'business' 'sport' 'tech'
'politics' 'sport' 'business' 'sport' 'tech' 'sport' 'business'
'entertainment' 'entertainment' 'business' 'politics' 'tech' 'tech'
'entertainment' 'business' 'sport' 'politics' 'business' 'tech' 'tech'
'politics' 'sport' 'politics' 'business' 'tech' 'business'
'entertainment' 'sport' 'sport' 'tech' 'tech' 'entertainment' 'sport'
'business' 'entertainment' 'sport' 'entertainment' 'entertainment' 'tech'
'business' 'politics' 'business' 'entertainment' 'politics' 'politics'
'politics' 'tech' 'sport' 'sport' 'politics' 'business' 'entertainment'
'sport' 'business' 'business' 'politics' 'politics' 'entertainment'
'business' 'politics' 'sport' 'tech' 'entertainment' 'business'
'entertainment' 'entertainment' 'politics']
```

In [55]:
```
# df_solution.to_csv('sol.csv', index = False)
```

Reference List

TFIDF Concepts

- https://medium.com/@cmukesh8688/tf-idf-vectorizer-scikit-learn-dbc0244a911a

Regex for excluding numbers as well as words containing numbers referenced from this stackoverflow post:

- https://stackoverflow.com/questions/51643427/how-to-make-tfidfvectorizer-only-learn-alphabetical-characters-as-part-of-the-vo

Sklearn Library:

- https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html
- https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html#sklearn.cluster.KMeans
- https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.TruncatedSVD.html
- https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.NMF.html