Project1_505851728_005627440_505297814

January 24, 2022

1 Project1: End-to-End Pipeline to Classify News Articles

- 505851728 Yang-Shan Chen
- 005627440 Chih-En Lin
- 505297814 Rikako Hatoya

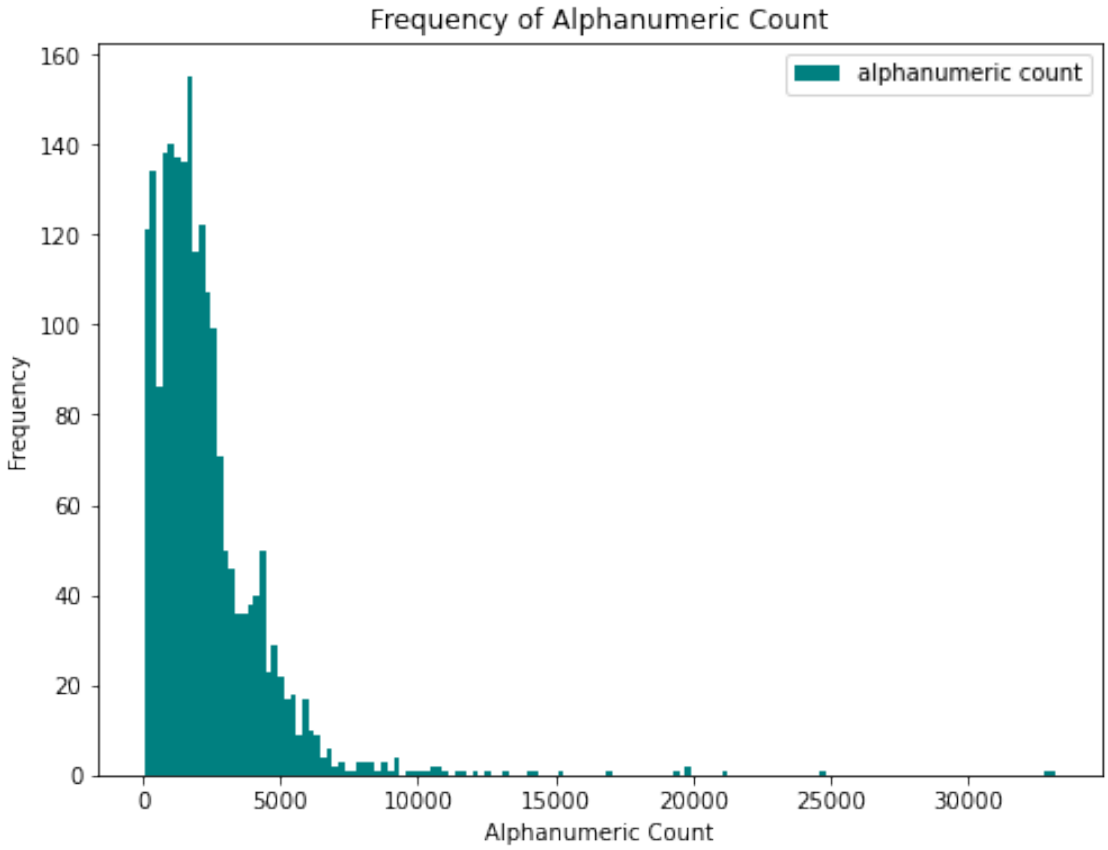
[1]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import random
import re
import nltk
#nltk.download("all")
from matplotlib import pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction import text
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.naive_bayes import MultinomialNB
from sklearn.decomposition import NMF
from sklearn.decomposition import TruncatedSVD
from nltk.tokenize import word_tokenize
from nltk.tag import pos_tag
from nltk.stem import WordNetLemmatizer
from nltk.corpus import wordnet
from nltk.corpus import words
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
```

2 Question 1

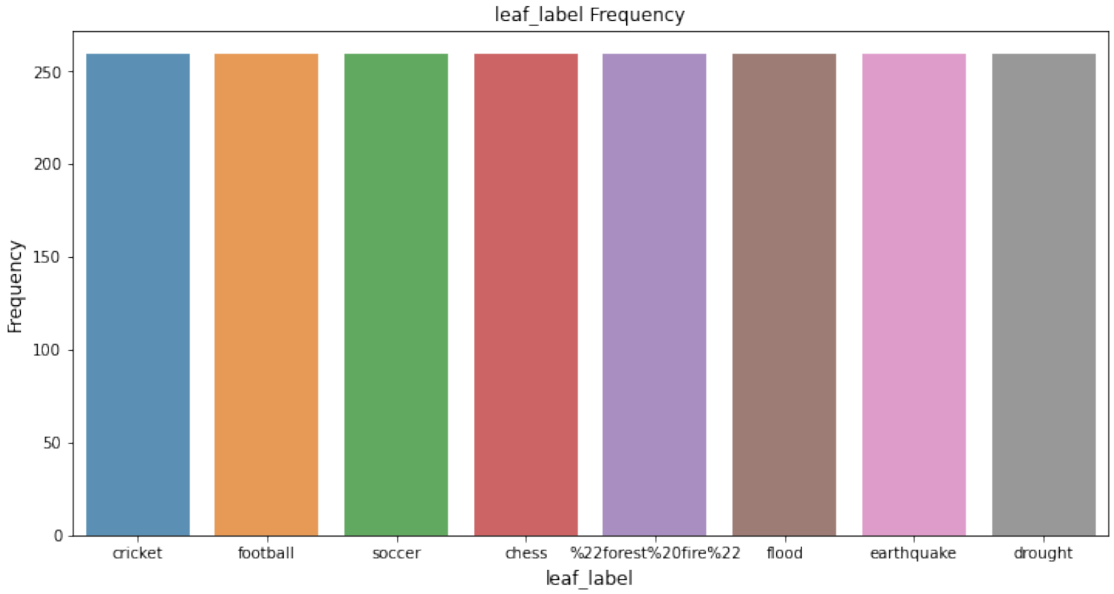
[3]:
```
data = pd.read_csv('Project_1_dataset_01_01_2022.csv')
#Overview
print("rows = "+str(data.shape[0])+"\n"+"columns = "+str(data.shape[1]))
```

```
rows = 2072
columns = 9
```

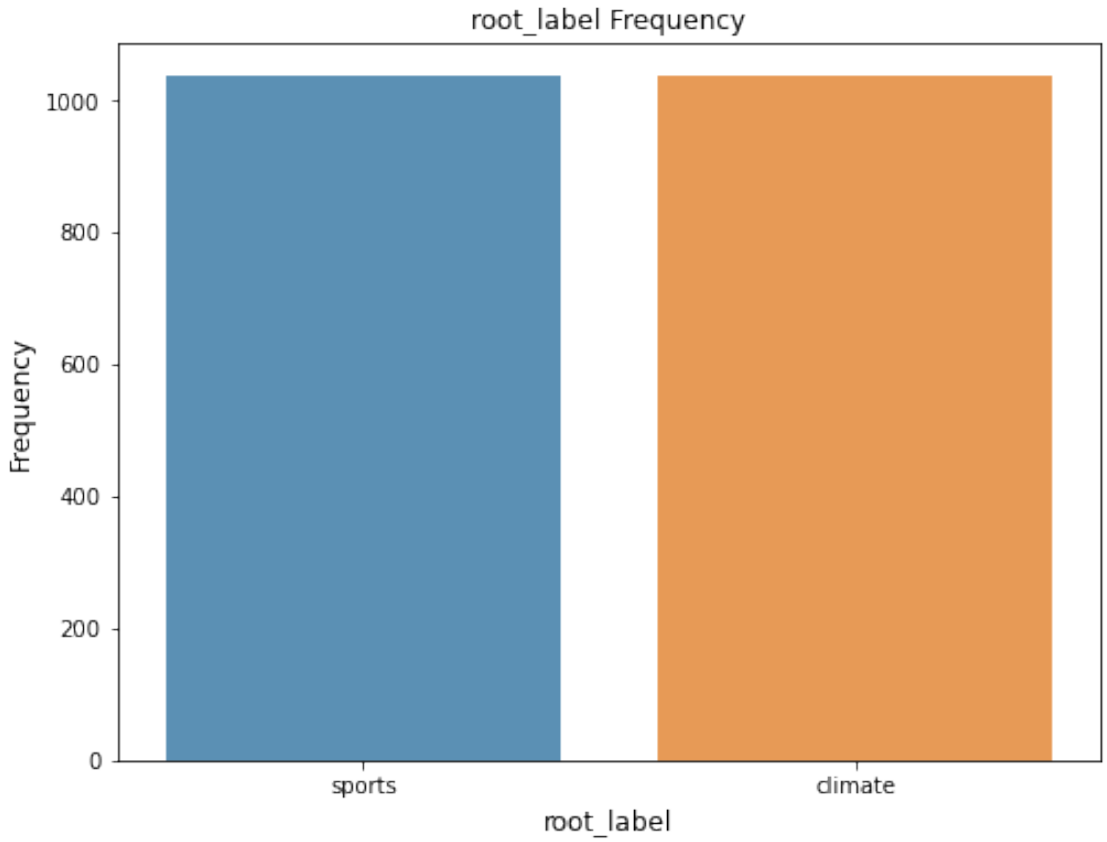
[4]:
```
#Histograms
#(a)
count_alphanumeric = []
for row in range(data.shape[0]):
    count_alphanumeric.append(sum([c.isalnum() for c in data.iat[row,1]]))
data_a=pd.DataFrame({'alphanumeric count':count_alphanumeric})
histogram_alphanumeric=data_a.plot.hist(bins=150, color='teal',figsize=(8,6))
histogram_alphanumeric.set_title("Frequency of Alphanumeric Count")
histogram_alphanumeric.set_xlabel("Alphanumeric Count")
histogram_alphanumeric.set_ylabel("Frequency")
plt.show()
```



[7]:
```
#(b)
plt.figure(figsize=(12,6))
histogram_leaflabel=sns.barplot(x=data['leaf_label'].value_counts().index,␣
 ↪y=data['leaf_label'].value_counts().values, alpha=0.8)
plt.title('leaf_label Frequency')
plt.ylabel('Frequency', fontsize=12)
plt.xlabel('leaf_label', fontsize=12)
plt.show()
```



[8]:
```
#(c)
plt.figure(figsize=(8,6))
histogram_rootlabel=sns.barplot(x=data['root_label'].value_counts().index,␣
 ↪y=data['root_label'].value_counts().values, alpha=0.8)
plt.title('root_label Frequency')
plt.ylabel('Frequency', fontsize=12)
plt.xlabel('root_label', fontsize=12)
plt.show()
```



Histogram interpretations:

For both leaf_label and root_label, frequencies for each category is evenly distributed. As for the alphanumerical lengths, most posts lie under 10,000 characters and the mode lies near 2,500 characters.

3 Question 2

[5]:
```
np.random.seed(42)
random.seed(42)
train, test = train_test_split(data[["full_text","root_label","leaf_label"]],␣
 ↪test_size=0.2)
print("training samples = "+str(len(train))+"\n"+"test samples =␣
 ↪"+str(len(test)))
```

```
training samples = 1657
test samples = 415
```

4 Question 3

Cleaning Data: Each full_string is cleaned using the following processor by removing HTML artefacts as well as removing any URLs, e-mail address, and any words that contain numbers.

[6]:
```
###Cleaning Preprocessor
#Cleans the data, removes raw input other than text
def clean(text):
    text = re.sub(r'^https?:\/\/.*[\r\n]*', '', text, flags=re.MULTILINE)
    text = re.sub(r'https?:\/\/.*[\r\n]*', '', text, flags=re.MULTILINE)
    text = re.sub(r'www.*[\r\n]*', '', text, flags=re.MULTILINE)
    text = re.sub(r'\S*@\S*\s?', '', text, flags=re.MULTILINE) #remove email␣
 ↪address
    text=re.sub(r'\w*\d+\w*', '', text).strip() #remove any word with numbers
    texter = re.sub(r"<br />", " ", text)
    texter = re.sub(r"&quot;", "\"",texter)
    texter = re.sub('&#39;', "\"", texter)
    texter = re.sub('\n', " ", texter)
    texter = re.sub('\__', "", texter)
    texter = re.sub('\|', "", texter)
    texter = re.sub('\/', " ", texter)
    texter = re.sub('--', "", texter)
    texter = re.sub(r'\d+', '', texter) #remove word with just numbers
    texter = re.sub(' u '," you ", texter)
    texter = re.sub(' +', ' ', texter)
    texter = re.sub(r"(!)\1+", r"!", texter)
    texter = re.sub(r"(\?)\1+", r"?", texter)
    texter = re.sub(' &amp; ', ' and ', texter)
    texter = re.sub('\r', ' ',texter)
    clean = re.compile('<.*?>')
    texter = texter.encode('ascii', 'ignore').decode('ascii')
    texter = re.sub(clean, '', texter)
    if texter == "":
        texter = ""
    return texter
def get_wordnet_pos(word):
    tag = nltk.pos_tag([word])[0][1][0].upper()
    tag_dict = {"J": wordnet.ADJ,
                "N": wordnet.NOUN,
                "V": wordnet.VERB,
                "R": wordnet.ADV}

    return tag_dict.get(tag, wordnet.NOUN)

stop_words = text.ENGLISH_STOP_WORDS
english_vocab = set(w.lower() for w in nltk.corpus.words.words())
```

Lemmatizing Data: Each word from full_text corpus is extracted, examined whether they are

included in the NLTK English vocabulary list, if the letters are in English, and whether they are not included in the English stop words list. If the word clears these 3 cases, they are then lemmatized and returned back to its corpus.

[7]:
```
###Lemmatizer (Reference: https://www.machinelearningplus.com/nlp/
 ↪lemmatization-examples-python/)
#Check each word inside each of the full_text corpus, makes words lower-case,␣
 ↪removes non-English words
#converts words to its simplest form i.e plural to singluar, etc using␣
 ↪lemmatizer
#appends that word, joins them into a sentence and adds it to␣
 ↪lemmatized_full_text

cleaned_train_full_text=[]
cleaned_test_full_text=[]
for row in range(train.shape[0]):
    cleaned_train_full_text.append(clean(train.iat[row,0]))
for row in range(test.shape[0]):
    cleaned_test_full_text.append(clean(test.iat[row,0]))

lemmatizer = WordNetLemmatizer()

lemmatized_train_full_text=[]
lemmatized_test_full_text=[]

for full_text in cleaned_train_full_text:
    wordlist=[]
    for word in nltk.word_tokenize(full_text):
        word=word.lower()
        if (word in english_vocab) and word.isalpha() and ((word in stop_words)␣
 ↪== False):
            word=lemmatizer.lemmatize(word, get_wordnet_pos(word))
            wordlist.append(word)
    lemmatized_train_full_text.append(' '.join(wordlist))
for full_text in cleaned_test_full_text:
    wordlist=[]
    for word in nltk.word_tokenize(full_text):
        word=word.lower()
        if (word in english_vocab) and word.isalpha() and ((word in stop_words)␣
 ↪== False):
            word=lemmatizer.lemmatize(word, get_wordnet_pos(word))
            wordlist.append(word)
    lemmatized_test_full_text.append(' '.join(wordlist))
```

[10]:
```
###Initialization
tfidf_transformer=TfidfTransformer()
vectorizer=CountVectorizer(min_df=3)
```

```
###train
train_count=vectorizer.fit_transform(lemmatized_train_full_text) #fitting my␣
 ↪model and vectorizing it
train_tfidf=tfidf_transformer.fit_transform(train_count).toarray()
###test
test_count=vectorizer.transform(lemmatized_test_full_text) #transform doesn't␣
 ↪add new vocabs as where fit_transform trained vocabs
test_tfidf=tfidf_transformer.transform(test_count).toarray()
```

[7]:
```
print("The shape of the training matrix is ({}, {})".format(train_tfidf.
 ↪shape[0],train_tfidf.shape[1]))
print("The shape of the test matrix is ({}, {})".format(test_tfidf.
 ↪shape[0],test_tfidf.shape[1]))
```

```
The shape of the training matrix is (1657, 6759)
The shape of the test matrix is (415, 6759)
```

[11]:
```
###assigning each document category names to category numbers
#making a list of all category types
#0:sports; 1:climate
train_rootlabel_categories = data['root_label'].value_counts().index.tolist()
#making a list of all document categories using category numbers
train_rootlabel_numbers = []

for row in range(train.shape[0]):
    for i, s in enumerate(train_rootlabel_categories):
        if train.iat[row,1] == s:
            train_rootlabel_numbers.append(i)
            break

train_rootlabel_numbers=np.asarray(train_rootlabel_numbers)
########################################
test_rootlabel_numbers=[]

for row in range(test.shape[0]):
    for i, s in enumerate(train_rootlabel_categories):
        if test.iat[row,1] == s:
            test_rootlabel_numbers.append(i)
            break

test_rootlabel_numbers=np.asarray(test_rootlabel_numbers)
```

1. What are the pros and cons of lemmatization versus stemming? How do these processes affect the dictionary size?

Stemming tries to remove the prefixes/suffixes of the word until its stem is reached as where lemmatization would look for the category of a word and uses a full dictionary to conduct a morphological analysis to get its lemma.

2. min_df means minimum document frequency. How does varying min df change the TF-IDF matrix?

Since mindf selects words to store in the dictionary from CountVectorizer if a certain word appears in the file more than the value of mindf, if mindf is increased, the number of vocabularies in the dictionary would decrease and if mindf is decreased, the vice versa occurs.

3. Should I remove stopwords before or after lemmatizing? Should I remove punctuations before or after lemmatizing? Should I remove numbers before or after lemmatizing?

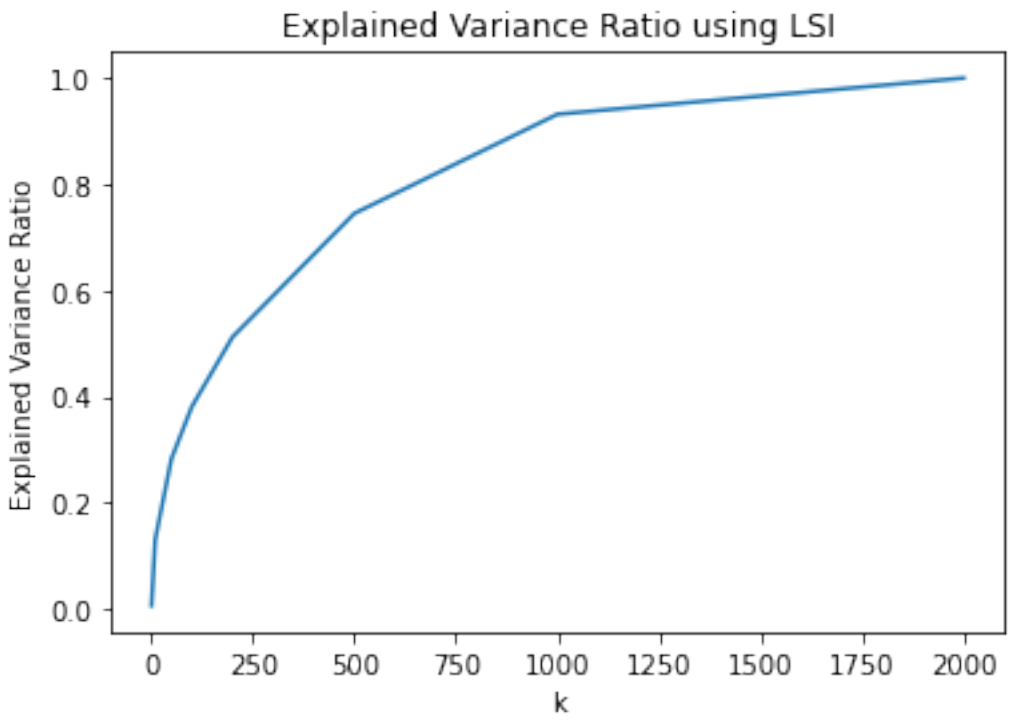
Remove stopwords before lemmatizing because otherwise, stopwords wouldn't be able to remove all the stop words if each word were modified to its lemma. Numbers and punctuations should be removed before lemmatizing since lemmatize only works for words. This is done in my program by only lemmatizing elements in my word list that are english vocabs and checking that they only contain english letters using word.isalpha().

4. Report the shape of the TF-IDF-processed train and test matrices. The number of rows should match the results of Question 2. The number of columns should roughly be in the order of k×10^3. This dimension will vary depending on your exact method of cleaning and lemmatizing and that is okay.

for train tfidf: (1657, 6753) for test tfidf: (415, 6753)

5 Question 4

[15]:
```
###LSI explained variance ratio plot
explained_variance_ratio_sum=[]
k_values=[1,10,50,100,200,500,1000,2000]

for k in k_values:
    lsi = TruncatedSVD(n_components=k, n_iter=10, random_state=42)
    lsi.fit_transform(train_tfidf)
    explained_variance_ratio_sum.append(lsi.explained_variance_ratio_.sum())

plt.plot(k_values, explained_variance_ratio_sum)
plt.xlabel('k')
plt.ylabel('Explained Variance Ratio')
plt.title('Explained Variance Ratio using LSI')
plt.show()
```



This concavity suggests that the explained variance ratio increases rapidly when k is smaller and then decreases change as k-values increase. However, increasing k would also mean that the model gains complexity.

[12]:
```
###LSI, residual error
import scipy.sparse.linalg
from numpy import linalg as LA #frobenius norm function
U, s, Vk_T = scipy.sparse.linalg.svds(train_tfidf, k=50)
Vk=Vk_T.transpose()
reduced_lsi_train=np.dot(train_tfidf, Vk)
reduced_lsi_test=np.dot(test_tfidf, Vk)
print("LSI residual MSE error= "+str((LA.norm(train_tfidf-np.
 ↪dot(reduced_lsi_train, Vk_T), 'fro')**2)))
```

```
LSI residual MSE error= 1146.5211352451013
```

[17]:
```
###NMF, residual error
#(Reference: https://qiita.com/takechanman1228/items/6d1f65f94f7aaa016377)
#W_train and W_test are the dimension reduced matrix, H_train is the keyword␣
 ↪list
model_nmf = NMF(n_components=50, init='random', random_state=42, max_iter=1000)
W_train = model_nmf.fit_transform(train_tfidf)
H_train = model_nmf.components_
W_test=model_nmf.transform(test_tfidf)
```

```
#reconstruction residual MSE error
print("NMF residual MSE error= "+str(model_nmf.reconstruction_err_**2))
```

```
NMF residual MSE error= 1173.0485586513678
```

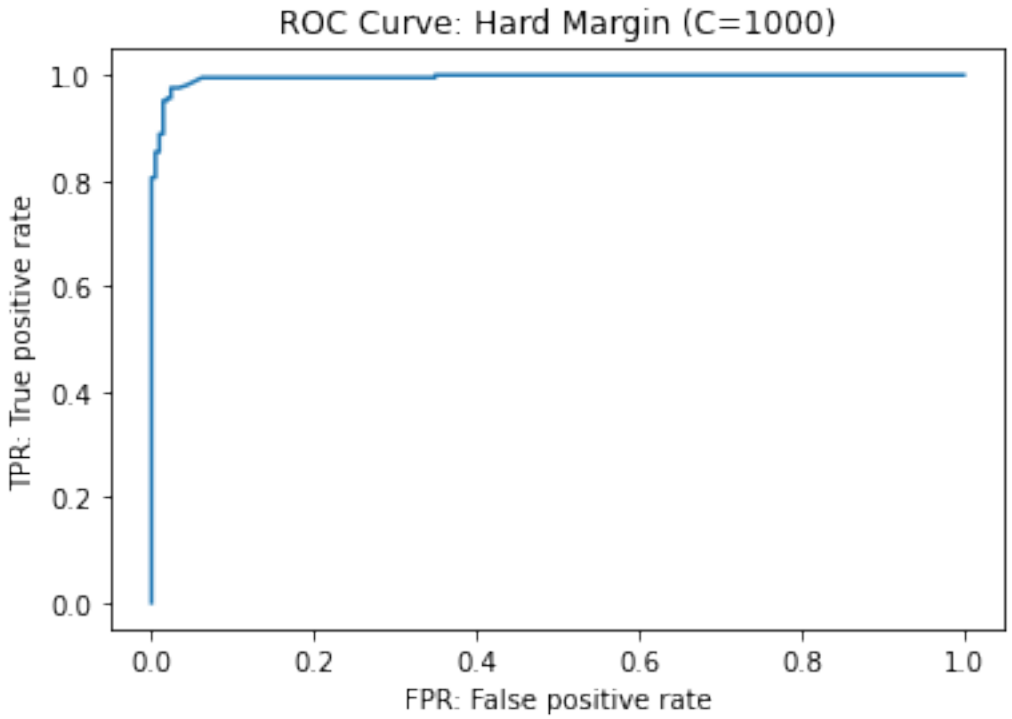
The error is lower for LSI than NMF. Since NMF only allows positive entries after reduction, meanwhile, LSI maintains max variance and can have negative entries. Hence, LSI represents the higher-dimensional matrix better, providing a deeper factorization with lower information loss than NMF.

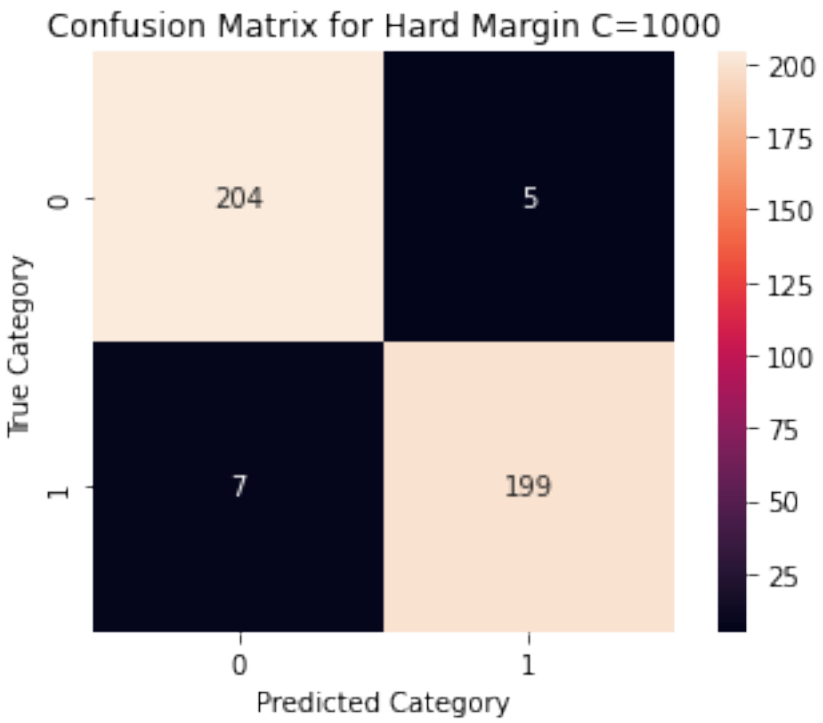
6 Question 5

[8]:
```
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, roc_curve, confusion_matrix,␣
 ↪recall_score, precision_score, f1_score

def plt_roc_curve(model, txt):
    prob = model.predict_proba(reduced_lsi_test)[:,1]
    fpr, tpr, thresholds = roc_curve(test_rootlabel_numbers, prob)
    plt.plot(fpr, tpr)
    plt.title("ROC Curve: "+txt)
    plt.xlabel('FPR: False positive rate')
    plt.ylabel('TPR: True positive rate')
    plt.show()

def plt_confusion_matrix(pred_test, txt):
    cmx_data = confusion_matrix(test_rootlabel_numbers, pred_test)
    df_cmx = pd.DataFrame(cmx_data)
    sns.heatmap(df_cmx, fmt='d', annot=True, square=True)
    plt.title('Confusion Matrix for '+txt)
    plt.xlabel('Predicted Category')
    plt.ylabel('True Category')
    plt.show()

def arpf(pred_test):
    print("accuracy of test data: "+str(accuracy_score(test_rootlabel_numbers,␣
 ↪pred_test)))
    print("recall: "+str(recall_score(test_rootlabel_numbers, pred_test)))
    print("precision: "+str(precision_score(test_rootlabel_numbers, pred_test)))
    print("F-1 Score: "+str(f1_score(test_rootlabel_numbers, pred_test)))
```

[19]:
```
###Hard Margin
#training classifier and predicting

clf_hardmargin = SVC(kernel='linear', random_state=42, C=1000, probability=True)
clf_hardmargin.fit(reduced_lsi_train, train_rootlabel_numbers)
```

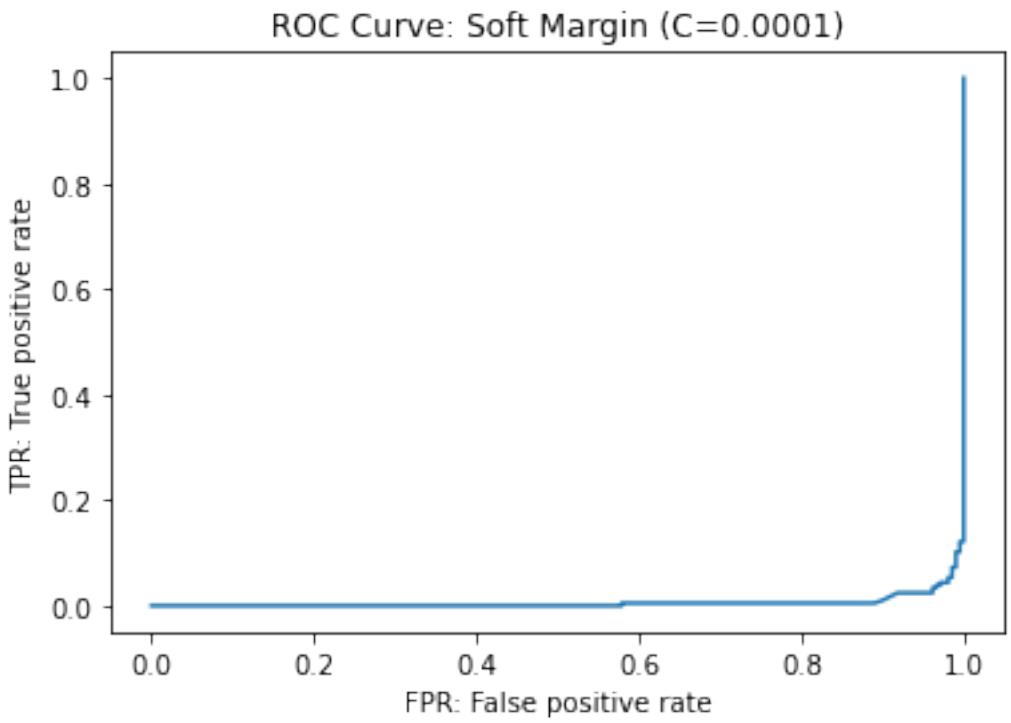
```
pred_test = clf_hardmargin.predict(reduced_lsi_test)

#ROC curve
plt_roc_curve(clf_hardmargin, 'Hard Margin (C=1000)')

#confusion matrix
plt_confusion_matrix(pred_test, 'Hard Margin C=1000')

#accuracy, recall, precision, F-1 Score
arpf(pred_test)
```

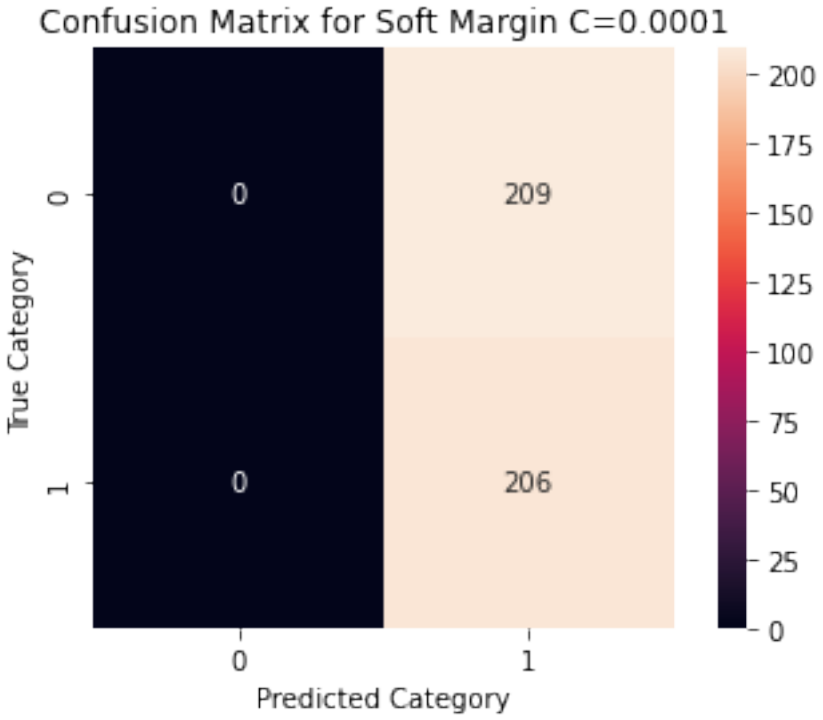





```
accuracy of test data: 0.9710843373493976
recall: 0.9660194174757282
precision: 0.9754901960784313
F-1 Score: 0.9707317073170731
```

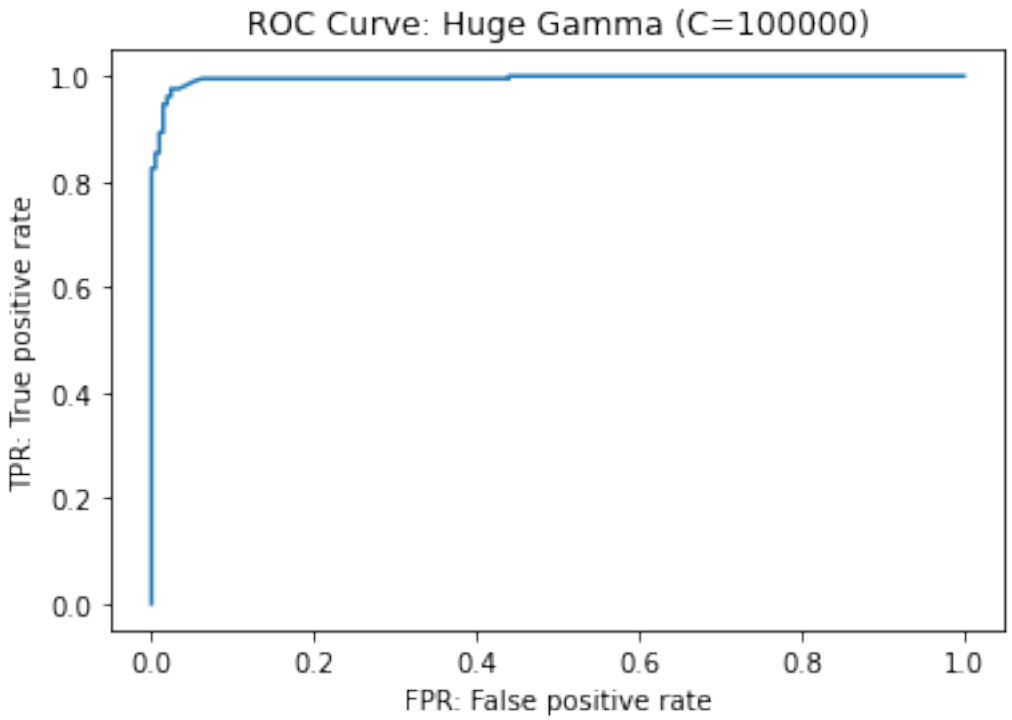
[20]:
```
###Soft Margin
#training classifier and predicting
clf_softmargin = SVC(kernel='linear', random_state=42, C=0.0001,␣
 ↪probability=True)
clf_softmargin.fit(reduced_lsi_train, train_rootlabel_numbers)

pred_test = clf_softmargin.predict(reduced_lsi_test)

#ROC curve
plt_roc_curve(clf_softmargin, 'Soft Margin (C=0.0001)')

#confusion matrix
plt_confusion_matrix(pred_test, 'Soft Margin C=0.0001')

#accuracy, recall, precision, F-1 Score
arpf(pred_test)
```

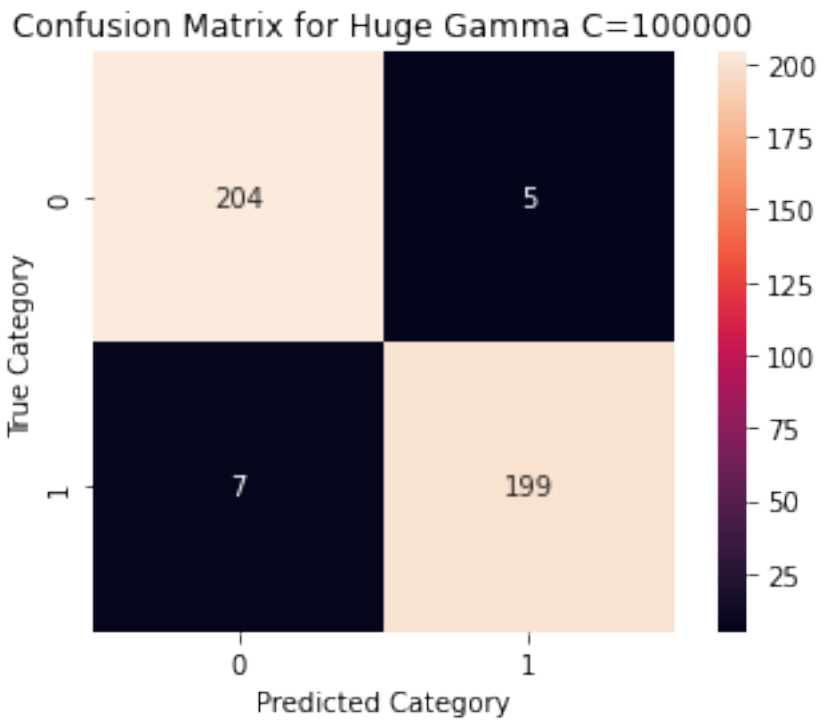

ROC Curve: Soft Margin (C=0.0001)
TPR: True positive rate
FPR: False positive rate
0.0
0.2
0.4
0.6
0.8
1.0


Confusion Matrix for Soft Margin C=0.0001
True Category
Predicted Category
0
209
0
206
0
1
200
175
150
125
100
75
50
25
0

```
accuracy of test data: 0.4963855421686747
recall: 1.0
precision: 0.4963855421686747
F-1 Score: 0.6634460547504025
```

[21]:
```
###huge Gamma
#training classifier and predicting
clf_hugemargin = SVC(kernel='linear', random_state=42, C=100000,␣
 ↪probability=True)
clf_hugemargin.fit(reduced_lsi_train, train_rootlabel_numbers)

pred_test = clf_hugemargin.predict(reduced_lsi_test)

#ROC curve
plt_roc_curve(clf_hugemargin, 'Huge Gamma (C=100000)')

#confusion matrix
plt_confusion_matrix(pred_test, 'Huge Gamma C=100000')

#accuracy, recall, precision, F-1 Score
arpf(pred_test)
```





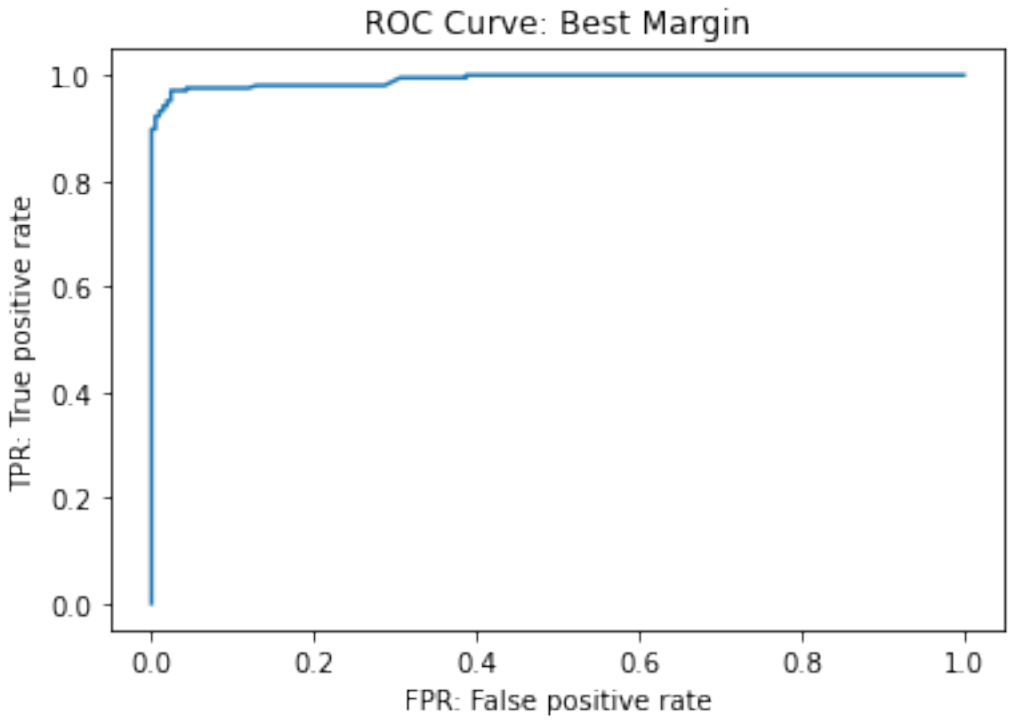

```
accuracy of test data: 0.9710843373493976
recall: 0.9660194174757282
precision: 0.9754901960784313
F-1 Score: 0.9707317073170731
```

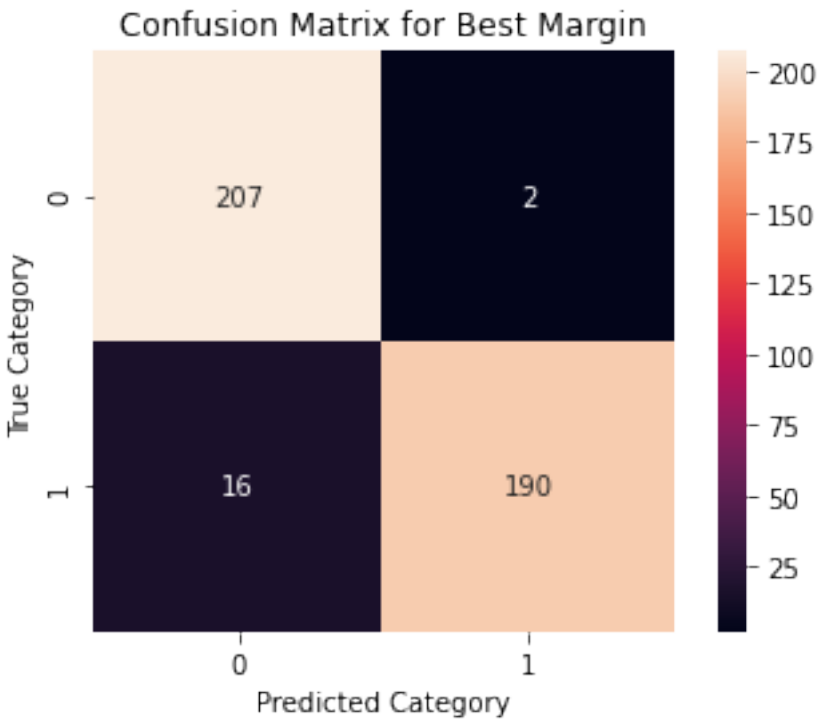
According to the confusion matrices, adopting a huge gamma or a hard margin results to a better classification than using a soft margin. For soft margin, we can see that all predicted labels are the same. This is due to giving the model too much regularization. Hence, even if we have the incorrect predictions, the penalty is subtle compared by the regularization term. Furthermore, the ROC curve for soft margin appears on the right-bottom corner, which indicates as being incompetitive.

[11]:
```
#Cross-Validation (Reference: https://aiacademy.jp/texts/show/?id=299)
from sklearn.model_selection import GridSearchCV

candidate_params = {'C': [10**i for i in range(-3, 7)]}

svc = SVC(kernel='linear', random_state=42, probability=True, max_iter=10000)
clf = GridSearchCV(estimator=svc, param_grid=candidate_params, cv=5, n_jobs=-1)
clf.fit(reduced_lsi_train, train_rootlabel_numbers)
print("Best Parameters: "+str(clf.best_params_))
print("Best Score: "+str(clf.best_score_))

#training classifier and predicting
```

```
clf = SVC(kernel='linear', random_state=42, C=clf.best_params_['C'],␣
 ↪probability=True)
clf.fit(reduced_lsi_train, train_rootlabel_numbers)

pred_test = clf.predict(reduced_lsi_test)

#ROC curve
plt_roc_curve(clf, 'Best Margin')

#confusion matrix
plt_confusion_matrix(pred_test, 'Best Margin')

#accuracy, recall, precision, F-1 Score
arpf(pred_test)
```

```
Best Parameters: {'C': 0.1}
Best Score: 0.9474884431987769
```

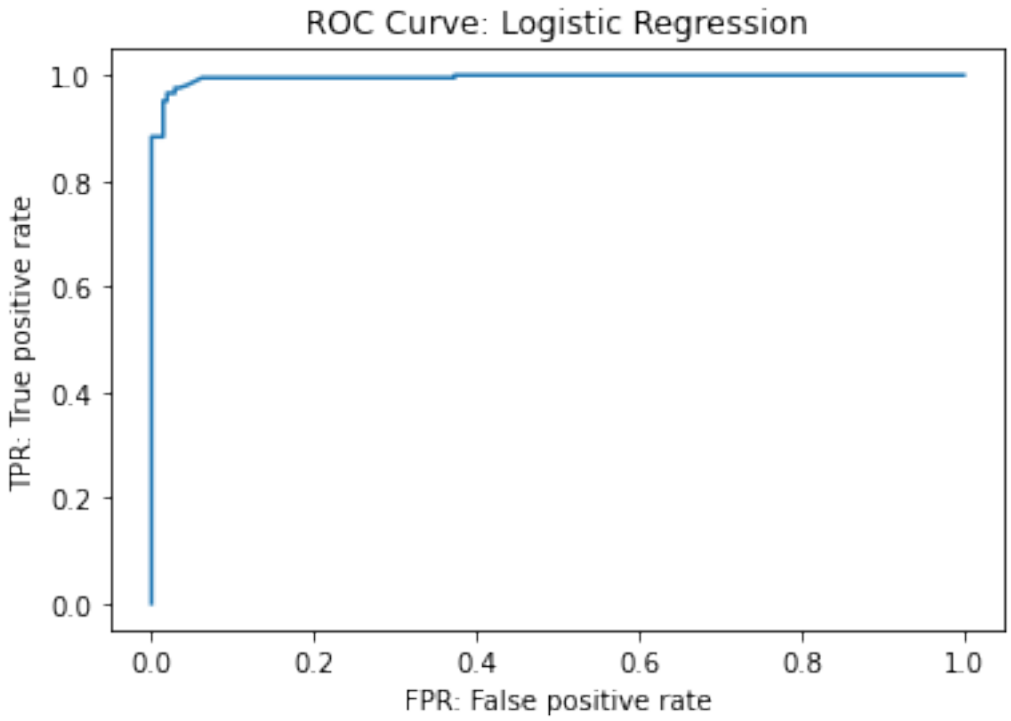


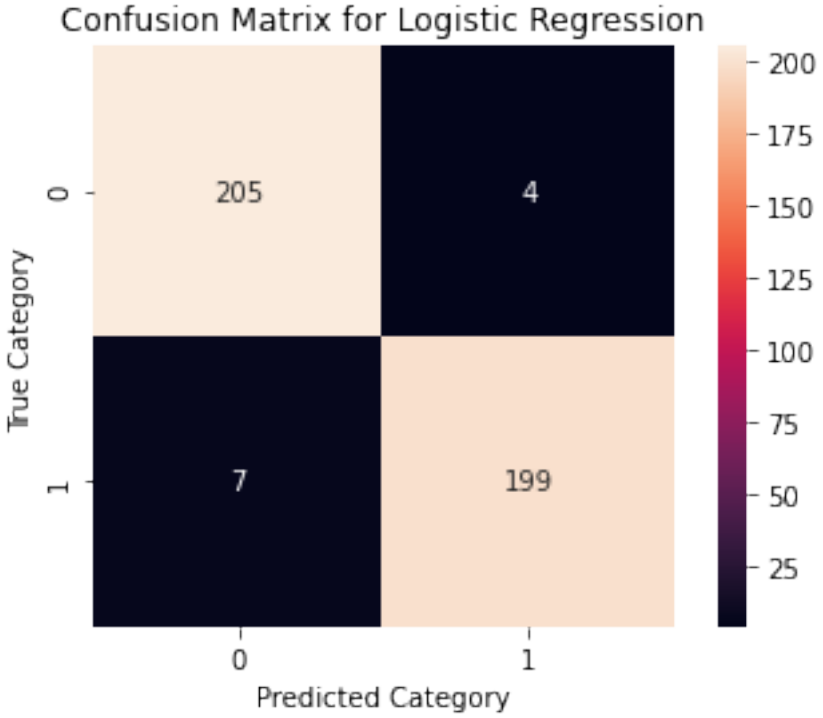



```
accuracy of test data: 0.9566265060240964
recall: 0.9223300970873787
precision: 0.9895833333333334
F-1 Score: 0.9547738693467336
```

7 Question 6

[13]:
```
###No regularization
from sklearn.linear_model import LogisticRegression
clf = LogisticRegression(penalty='none', max_iter=10000, tol=0.0001,␣
 ↪solver='saga', random_state=42)
clf.fit(reduced_lsi_train, train_rootlabel_numbers)
pred_test = clf.predict(reduced_lsi_test)

#ROC curve
plt_roc_curve(clf, 'Logistic Regression')

#confusion matrix
plt_confusion_matrix(pred_test, 'Logistic Regression')

#accuracy, recall, precision, F-1 Score
arpf(pred_test)
```


ROC Curve: Logistic Regression
TPR: True positive rate
FPR: False positive rate
0.0
0.2
0.4
0.6
0.8
1.0


Confusion Matrix for Logistic Regression
True Category
Predicted Category
205
4
7
199
0
1
200
175
150
125
100
75
50
25

```
accuracy of test data: 0.9734939759036144
recall: 0.9660194174757282
precision: 0.9802955665024631
F-1 Score: 0.9731051344743277
```

[59]:
```
###Optimal regularization strength for L1 and L2 using cross-validation
candidate_params = {'C': [10**i for i in range(-4, 5)]}
lgr=LogisticRegression(penalty='l1', solver='saga', max_iter=10000, tol=0.0001,␣
 ↪random_state=42)
clf = GridSearchCV(estimator=lgr, param_grid=candidate_params, cv=5)
clf.fit(reduced_lsi_train, train_rootlabel_numbers)
print('L1 regularization')
print("Optimal C: "+str(clf.best_params_['C']))
print("Best Score: "+str(clf.best_score_))
print('\n')

lgr=LogisticRegression(penalty='l2', solver='saga', max_iter=10000, tol=0.0001,␣
 ↪random_state=42)
clf = GridSearchCV(estimator=lgr, param_grid=candidate_params, cv=5)
clf.fit(reduced_lsi_train, train_rootlabel_numbers)
print('L2 regularization')
print("Optimal C: "+str(clf.best_params_['C']))
print("Best Score: "+str(clf.best_score_))
```

```
L1 regularization
Optimal C: 10
Best Score: 0.94990536162778


L2 regularization
Optimal C: 10
Best Score: 0.9529174098205511
```

[60]:
```
###No regularization
clf = LogisticRegression(penalty='none', solver='saga', max_iter=10000, tol=0.
 ↪0001, random_state=42)
clf.fit(reduced_lsi_train, train_rootlabel_numbers)
pred_test = clf.predict(reduced_lsi_test)
#accuracy, recall, precision, F-1 Score
print('no regularization')
arpf(pred_test)
print('\n')
###L1
clf=LogisticRegression(penalty='l1', solver='saga', C=10, max_iter=10000, tol=0.
 ↪0001, random_state=42)
clf.fit(reduced_lsi_train, train_rootlabel_numbers)
pred_test = clf.predict(reduced_lsi_test)
#accuracy, recall, precision, F-1 Score
```

```
print('L1 regularization')
arpf(pred_test)
print('\n')
###L2
clf=LogisticRegression(penalty='l2', solver='saga', C=10, max_iter=10000, tol=0.
 ↪0001, random_state=42)
clf.fit(reduced_lsi_train, train_rootlabel_numbers)
pred_test = clf.predict(reduced_lsi_test)
#accuracy, recall, precision, F-1 Score
print('L2 regularization')
arpf(pred_test)
```

```
no regularization
accuracy of test data: 0.9734939759036144
recall: 0.9660194174757282
precision: 0.9802955665024631
F-1 Score: 0.9731051344743277


L1 regularization
accuracy of test data: 0.9686746987951808
recall: 0.9660194174757282
precision: 0.9707317073170731
F-1 Score: 0.9683698296836983


L2 regularization
accuracy of test data: 0.9710843373493976
recall: 0.9660194174757282
precision: 0.9754901960784313
F-1 Score: 0.9707317073170731
```

Overall, the performance is no regularization > L2 > L1. However, there is small difference between each.

For these three models, if one wants to follow the training data as closely as possible by avoiding overfitting to occur, "no regularization" is preferred.

L1 regularization is suitable for feature selection, simple and sparse models, and features associated with 0 weights can be discarded. The reason is that L1 regularization encourages all kinds of weights to shrink to 0, regardless of size of w. L1 regularization is more likely to zero out the coefficients than L2 regularization for similar test accuracies because it assumes prior on the weights sampled from an isotropic Laplace distribution.

L2 regularization, which assumes in prior from the weights sampled from a Gaussian distribution, is suitable for reducing the effects of collinear features, which can lead to increased variance (hence instability and sensitivity to outliers) of the model.

For differences between SVM and Logistic regression

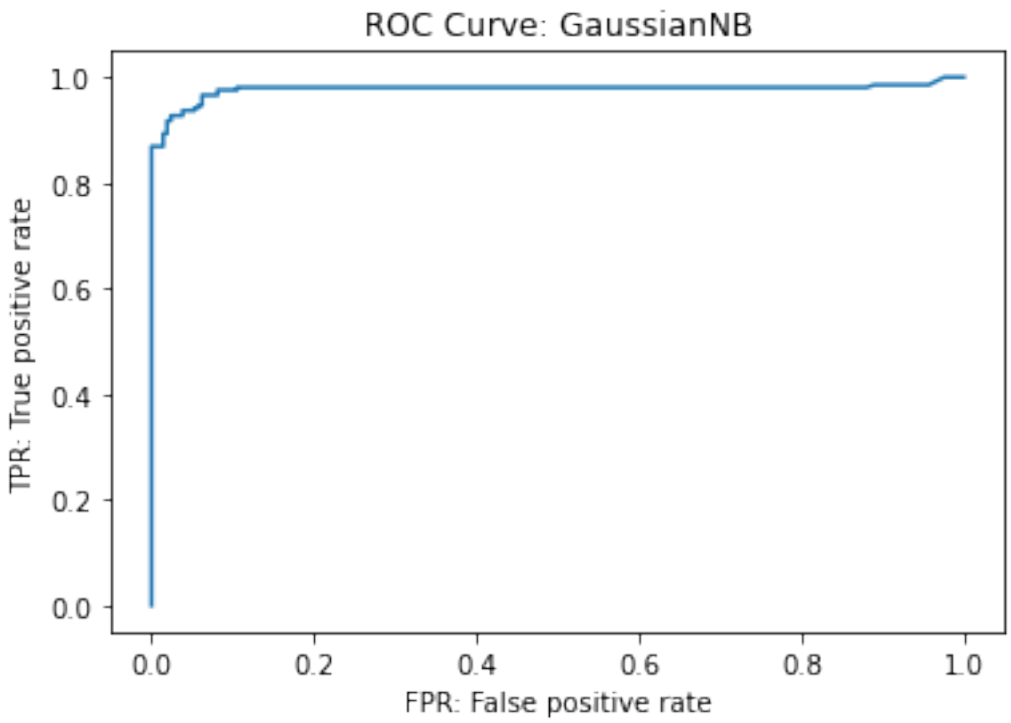
Logistic regression uses classic MLP method to construct its loss function. The goal is to maximize

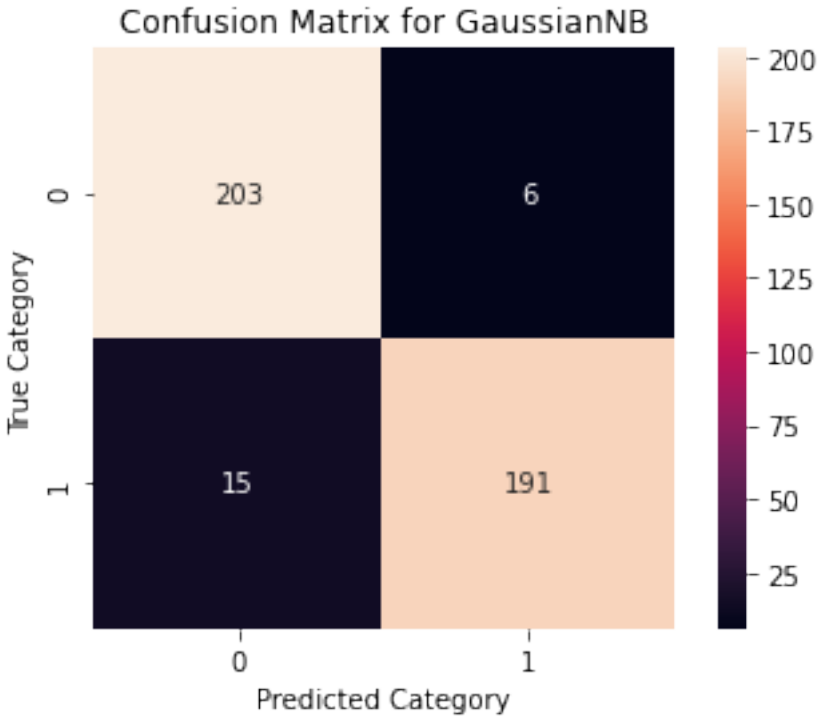
the conditional likelihood of correct classification on the entire training set. SVM is geometric in nature, using only some points closest to the decision boundary, which we called support vectors to find our optimal hyperplane. This enables to maximize the distance of the margin. Therefore, logistic regression does not guarantee it will yield an optimal decision boundary, whereas SVM yields the best separating hyperplane that reduces misclassification rate.

Logistic regression is more likely to cause overfitting since its decision is based on the entire training set, whereas SVM provides better generalization. In addition, logistic loss is sensitive to outliers and does not go to zero. As for hinge loss it can be 0 and is definitely less sensitive to outliers. As discussed earlier, logistic regression provides high confidence in classifying data further away from the hyperplane but fails for samples near the margin. SVM is more efficient and fast at handling complex, high-dimensional and unstructured data through the kernel trick. Logistic regression is more suitable for structured datasets.

8 Question 7

[14]:
```
from sklearn.naive_bayes import GaussianNB
clf = GaussianNB()
clf.fit(reduced_lsi_train, train_rootlabel_numbers)
pred_test = clf.predict(reduced_lsi_test)

#ROC curve
plt_roc_curve(clf, 'GaussianNB')

#confusion matrix
plt_confusion_matrix(pred_test, 'GaussianNB')

#accuracy, recall, precision, F-1 Score
arpf(pred_test)
```


ROC Curve: GaussianNB
TPR: True positive rate
FPR: False positive rate
0.0
0.2
0.4
0.6
0.8
1.0


Confusion Matrix for GaussianNB
True Category
Predicted Category
203
6
15
191
0
1
200
175
150
125
100
75
50
25

```
accuracy of test data: 0.9493975903614458
recall: 0.9271844660194175
precision: 0.9695431472081218
F-1 Score: 0.9478908188585609
```

9 Question 8

[62]:
```
#from nltk.stem import PorterStemmer
import time
from tempfile import mkdtemp
from joblib import Memory
from nltk.stem.lancaster import LancasterStemmer as LS

raw_data_train = [train.iat[row,0] for row in range(train.shape[0])] #text
raw_data_test = [test.iat[row,0] for row in range(test.shape[0])] #text
X_digits, y_digits = raw_data_train, train_rootlabel_numbers

wnl = nltk.wordnet.WordNetLemmatizer()
#ps = PorterStemmer()
ls = LS()

def penn2morphy(penntag):
    """ Converts Penn Treebank tags to WordNet. """
    morphy_tag = {'NN':'n', 'JJ':'a',
                  'VB':'v', 'RB':'r'}
    try:
        return morphy_tag[penntag[:2]]
    except:
        return 'n'

def lemmatize_sent(list_word):
    return [wnl.lemmatize(word.lower(), pos=penn2morphy(tag)) for word, tag in␣
 ↪pos_tag(list_word)]

def stem_sent(list_word):
    return [ls.stem(w) for w in list_word]

def noclean_lemma(doc):
    return (word for word in lemmatize_sent(word_tokenize(doc)) if (word in␣
 ↪english_vocab) and word.isalpha() and (word not in stop_words))

def clean_lemma(doc):
    return (word for word in lemmatize_sent(word_tokenize(clean(doc))) if (word␣
 ↪in english_vocab) and word.isalpha() and (word not in stop_words))

def noclean_stem(doc):
```

```
    return (word for word in stem_sent(word_tokenize(doc)) if (word in␣
 ↪english_vocab) and word.isalpha() and (word not in stop_words))

def clean_stem(doc):
    return (word for word in stem_sent(word_tokenize(clean(doc))) if (word in␣
 ↪english_vocab) and word.isalpha() and (word not in stop_words))

def noclean_nothing(doc):
    return (word for word in word_tokenize(doc) if (word in english_vocab) and␣
 ↪word.isalpha() and (word not in stop_words))

def clean_nothing(doc):
    return (word for word in word_tokenize(clean(doc)) if (word in␣
 ↪english_vocab) and word.isalpha() and (word not in stop_words))
```

[63]:
```
cachedir = mkdtemp()
memory = Memory(location=cachedir, verbose=10)

pipeline = Pipeline([
    ('vect', CountVectorizer(stop_words='english')),
    ('tfidf', TfidfTransformer()),
    ('scaler', StandardScaler(with_mean=False)),
    ('reduce_dim', None),
    ('clf', None),
],
memory=memory
)

param_grid = [{
        "vect__min_df":(3, 5),
        "vect__analyzer":
 ↪(noclean_nothing,clean_nothing,noclean_stem,clean_stem,noclean_lemma,clean_lemma),
        "reduce_dim":(TruncatedSVD(random_state=42), NMF(init='random',␣
 ↪random_state=42)),
        "reduce_dim__n_components": (5, 50, 500),
        "clf":(
            SVC(kernel='linear', random_state=42, probability=True, C=0.1,␣
 ↪max_iter=10000),
            LogisticRegression(penalty='l1', solver='saga', max_iter=10000,␣
 ↪tol=0.0001, random_state=42, C=10),
            LogisticRegression(penalty='l2', solver='saga', max_iter=10000,␣
 ↪tol=0.0001, random_state=42, C=10),
            GaussianNB(),
        ),
    }]
```

[64]:
```
full_time = time.time()
search = GridSearchCV(pipeline,cv=5,param_grid=param_grid,scoring='accuracy')
search.fit(X_digits, y_digits)
print("--- %s seconds ---" % (time.time() - full_time))
```

```
________________________________________________________________________________
[Memory] Calling sklearn.pipeline._fit_transform_one…
_fit_transform_one(CountVectorizer(analyzer=<function noclean_nothing at
0x00000201F32B7550>,
                min_df=3, stop_words='english'),
[ '"Pages of history" features excerpts from The News Journal archives '
  'including the Wilmington Morning News, The Morning News, the Every Evening '
  'and the Evening Journal.\n'
  '\n'
  'Oct. 17, 1987, The News Journal\n'
  '\n'
  'Rescuers free toddler from well after two days\n'
  '\n'
  'MIDLAND, Texas - Eighteen-month-old Jessica McClure was rescued last night '
  'from an abandoned well by workers who spent 2½ days drilling through solid '
  'rock to reach her as the nation waited anxiously to learn her fate.\n'
  '\n'
  'Barefoot, caked with dirt and strapped with gauze to an immobilizing '
  'backboard, Jessica was hoisted by cable out of the shaft just before 9 p.m. '
  'EDT t…,
array([1, …, 0]), None, message_clsname='Pipeline', message=None)
_____________________________________________fit_transform_one - 5.7s, 0.1min
________________________________________________________________________________
[Memory] Calling sklearn.pipeline._fit_transform_one…
_fit_transform_one(TfidfTransformer(), <1325x5227 sparse matrix of type '<class
'numpy.int64'>'
        with 106885 stored elements in Compressed Sparse Row format>,
array([1, …, 0]), None, message_clsname='Pipeline', message=None)
_____________________________________________fit_transform_one - 0.0s, 0.0min
________________________________________________________________________________
[Memory] Calling sklearn.pipeline._fit_transform_one…
_fit_transform_one(StandardScaler(with_mean=False), <1325x5227 sparse matrix of
type '<class 'numpy.float64'>'
        with 106885 stored elements in Compressed Sparse Row format>,
array([1, …, 0]), None, message_clsname='Pipeline', message=None)
_____________________________________________fit_transform_one - 0.0s, 0.0min
________________________________________________________________________________
[Memory] Calling sklearn.pipeline._fit_transform_one…
_fit_transform_one(TruncatedSVD(n_components=5, random_state=42), <1325x5227
sparse matrix of type '<class 'numpy.float64'>'
        with 106885 stored elements in Compressed Sparse Row format>,
array([1, …, 0]), None, message_clsname='Pipeline', message=None)
_____________________________________________fit_transform_one - 0.0s, 0.0min
```

[65]:
```
print("Best parameter (CV score=%0.3f):" % search.best_score_)
print(search.best_params_)
print('--------------------')
par = dict(zip(search.cv_results_['rank_test_score'], search.
 ↪cv_results_['params']))
score = dict(zip(search.cv_results_['rank_test_score'], search.
 ↪cv_results_['mean_test_score']))
for i in range(1, 6):
    print('rank '+str(i)+" (CV score=%0.3f):" % score[i])
    print(par[i])
```

```
Best parameter (CV score=0.945):
{'clf': LogisticRegression(C=10, max_iter=10000, penalty='l1', random_state=42,
                   solver='saga'), 'reduce_dim': TruncatedSVD(n_components=500,
random_state=42), 'reduce_dim__n_components': 500, 'vect__analyzer': <function
noclean_lemma at 0x0000020180933D30>, 'vect__min_df': 5}
--------------------
rank 1 (CV score=0.945):
{'clf': LogisticRegression(C=10, max_iter=10000, penalty='l1', random_state=42,
                   solver='saga'), 'reduce_dim': TruncatedSVD(n_components=500,
random_state=42), 'reduce_dim__n_components': 500, 'vect__analyzer': <function
noclean_lemma at 0x0000020180933D30>, 'vect__min_df': 5}
rank 2 (CV score=0.945):
{'clf': LogisticRegression(C=10, max_iter=10000, random_state=42,
solver='saga'), 'reduce_dim': TruncatedSVD(n_components=500, random_state=42),
'reduce_dim__n_components': 500, 'vect__analyzer': <function noclean_lemma at
0x0000020180933D30>, 'vect__min_df': 5}
rank 3 (CV score=0.945):
{'clf': LogisticRegression(C=10, max_iter=10000, random_state=42,
solver='saga'), 'reduce_dim': TruncatedSVD(n_components=500, random_state=42),
'reduce_dim__n_components': 500, 'vect__analyzer': <function clean_lemma at
0x0000020181851A60>, 'vect__min_df': 5}
rank 4 (CV score=0.945):
{'clf': LogisticRegression(C=10, max_iter=10000, penalty='l1', random_state=42,
                   solver='saga'), 'reduce_dim': TruncatedSVD(n_components=500,
random_state=42), 'reduce_dim__n_components': 500, 'vect__analyzer': <function
clean_lemma at 0x0000020181851A60>, 'vect__min_df': 5}
rank 5 (CV score=0.937):
{'clf': LogisticRegression(C=10, max_iter=10000, random_state=42,
solver='saga'), 'reduce_dim': TruncatedSVD(n_components=500, random_state=42),
'reduce_dim__n_components': 500, 'vect__analyzer': <function noclean_lemma at
0x0000020180933D30>, 'vect__min_df': 3}
```

[75]:
```
###Generate data to see our best 5 models performance on testing sets
cleaned_train_full_text=[]
cleaned_test_full_text=[]
nocleaned_train_full_text=[]
```

```
nocleaned_test_full_text=[]
for row in range(train.shape[0]):
    cleaned_train_full_text.append(clean(train.iat[row,0]))
for row in range(test.shape[0]):
    cleaned_test_full_text.append(clean(test.iat[row,0]))
for row in range(train.shape[0]):
    nocleaned_train_full_text.append(train.iat[row,0])
for row in range(test.shape[0]):
    nocleaned_test_full_text.append(test.iat[row,0])

lemmatizer = WordNetLemmatizer()

cleaned_lemmatized_train_full_text=[]
cleaned_lemmatized_test_full_text=[]
nocleaned_lemmatized_train_full_text=[]
nocleaned_lemmatized_test_full_text=[]

for full_text in cleaned_train_full_text:
    wordlist=[]
    for word in nltk.word_tokenize(full_text):
        word=word.lower()
        if (word in english_vocab) and word.isalpha() and ((word in stop_words)␣
 ↪== False):
            word=lemmatizer.lemmatize(word, get_wordnet_pos(word))
            wordlist.append(word)
    cleaned_lemmatized_train_full_text.append(' '.join(wordlist))
for full_text in cleaned_test_full_text:
    wordlist=[]
    for word in nltk.word_tokenize(full_text):
        word=word.lower()
        if (word in english_vocab) and word.isalpha() and ((word in stop_words)␣
 ↪== False):
            word=lemmatizer.lemmatize(word, get_wordnet_pos(word))
            wordlist.append(word)
    cleaned_lemmatized_test_full_text.append(' '.join(wordlist))
for full_text in nocleaned_train_full_text:
    wordlist=[]
    for word in nltk.word_tokenize(full_text):
        word=word.lower()
        if (word in english_vocab) and word.isalpha() and ((word in stop_words)␣
 ↪== False):
            word=lemmatizer.lemmatize(word, get_wordnet_pos(word))
            wordlist.append(word)
    nocleaned_lemmatized_train_full_text.append(' '.join(wordlist))
for full_text in nocleaned_test_full_text:
    wordlist=[]
    for word in nltk.word_tokenize(full_text):
```

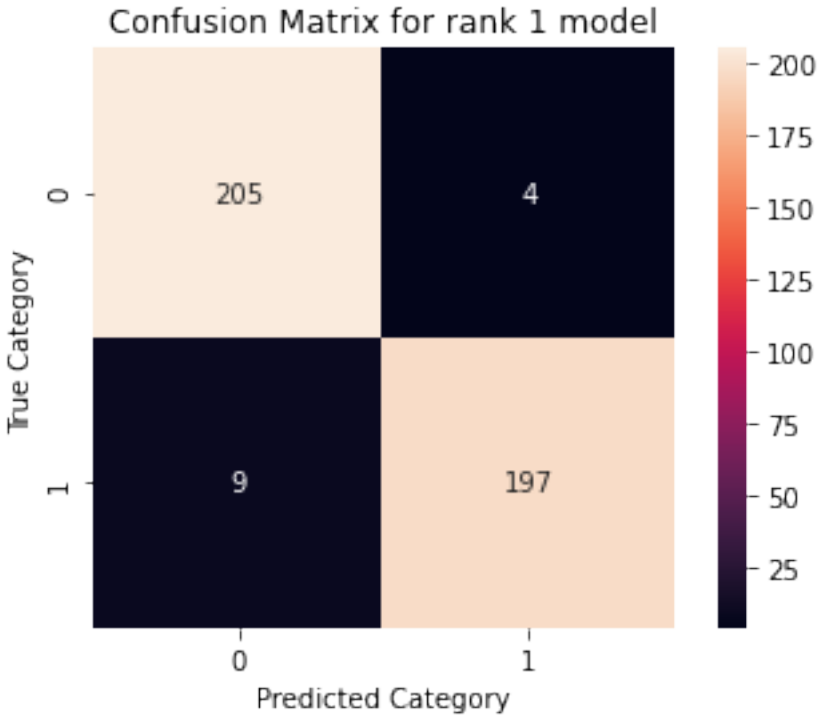
```
        word=word.lower()
        if (word in english_vocab) and word.isalpha() and ((word in stop_words)␣
 ↪== False):
            word=lemmatizer.lemmatize(word, get_wordnet_pos(word))
            wordlist.append(word)
    nocleaned_lemmatized_test_full_text.append(' '.join(wordlist))
```

[79]:
```
###Initialization
tfidf_transformer=TfidfTransformer()
vectorizer3=CountVectorizer(min_df=3)
vectorizer=CountVectorizer(min_df=5)
###cleaning, lemmatized, mindf=5
train_count_clean5=vectorizer5.fit_transform(cleaned_lemmatized_train_full_text)
train_tfidf_clean5=tfidf_transformer.fit_transform(train_count_clean5).toarray()
test_count_clean5=vectorizer5.transform(cleaned_lemmatized_test_full_text)
test_tfidf_clean5=tfidf_transformer.transform(test_count_clean5).toarray()
###not cleaning, lemmatized, mindf=5
train_count_noclean5=vectorizer5.
 ↪fit_transform(nocleaned_lemmatized_train_full_text)
train_tfidf_noclean5=tfidf_transformer.fit_transform(train_count_noclean5).
 ↪toarray()
test_count_noclean5=vectorizer5.transform(nocleaned_lemmatized_test_full_text)
test_tfidf_noclean5=tfidf_transformer.transform(test_count_noclean5).toarray()
###not cleaning, lemmatized, mindf=3
train_count_noclean3=vectorizer3.
 ↪fit_transform(nocleaned_lemmatized_train_full_text)
train_tfidf_noclean3=tfidf_transformer.fit_transform(train_count_noclean3).
 ↪toarray()
test_count_noclean3=vectorizer3.transform(nocleaned_lemmatized_test_full_text)
test_tfidf_noclean3=tfidf_transformer.transform(test_count_noclean3).toarray()
```

[80]:
```
###cleaning, lemmatized, mindf=5, LSI
U, s, Vk_T = scipy.sparse.linalg.svds(train_tfidf_clean5, k=500)
Vk=Vk_T.transpose()
reduced_lsi_train_clean5=np.dot(train_tfidf_clean5, Vk)
reduced_lsi_test_clean5=np.dot(test_tfidf_clean5, Vk)
###not cleaning, lemmatized, mindf=5, LSI
U, s, Vk_T = scipy.sparse.linalg.svds(train_tfidf_noclean5, k=500)
Vk=Vk_T.transpose()
reduced_lsi_train_noclean5=np.dot(train_tfidf_noclean5, Vk)
reduced_lsi_test_noclean5=np.dot(test_tfidf_noclean5, Vk)
###not cleaning, lemmatized, mindf=3, LSI
U, s, Vk_T = scipy.sparse.linalg.svds(train_tfidf_noclean3, k=500)
Vk=Vk_T.transpose()
reduced_lsi_train_noclean3=np.dot(train_tfidf_noclean3, Vk)
reduced_lsi_test_noclean3=np.dot(test_tfidf_noclean3, Vk)
```

rank 1 model: not cleaning data, min_df=5, Lemmatization, LSI(k=500), Logistic regression with L1 regularization

[85]:
```
clf=LogisticRegression(penalty='l1', solver='saga', C=10, max_iter=10000, tol=0.
↪0001, random_state=42)
clf.fit(reduced_lsi_train_noclean5, train_rootlabel_numbers)
pred_test = clf.predict(reduced_lsi_test_noclean5)

#confusion matrix
plt_confusion_matrix(pred_test, 'rank 1 model')

#accuracy, recall, precision, F-1 Score
arpf(pred_test)
```

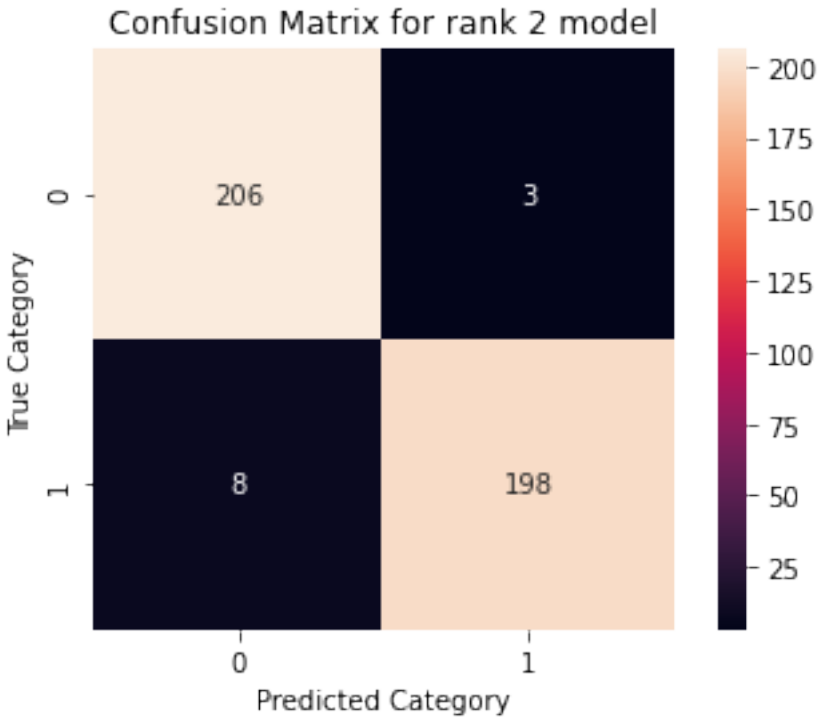



```
accuracy of test data: 0.9686746987951808
recall: 0.9563106796116505
precision: 0.9800995024875622
F-1 Score: 0.9680589680589681
```

rank 2 model: not cleaning data, min_df=5, Lemmatization, LSI(k=500), Logistic regression with L2 regularization

[86]:
```
clf=LogisticRegression(penalty='l2', solver='saga', C=10, max_iter=10000, tol=0.
↪0001, random_state=42)
```

```
clf.fit(reduced_lsi_train_noclean5, train_rootlabel_numbers)
pred_test = clf.predict(reduced_lsi_test_noclean5)

#confusion matrix
plt_confusion_matrix(pred_test, 'rank 2 model')

#accuracy, recall, precision, F-1 Score
arpf(pred_test)
```

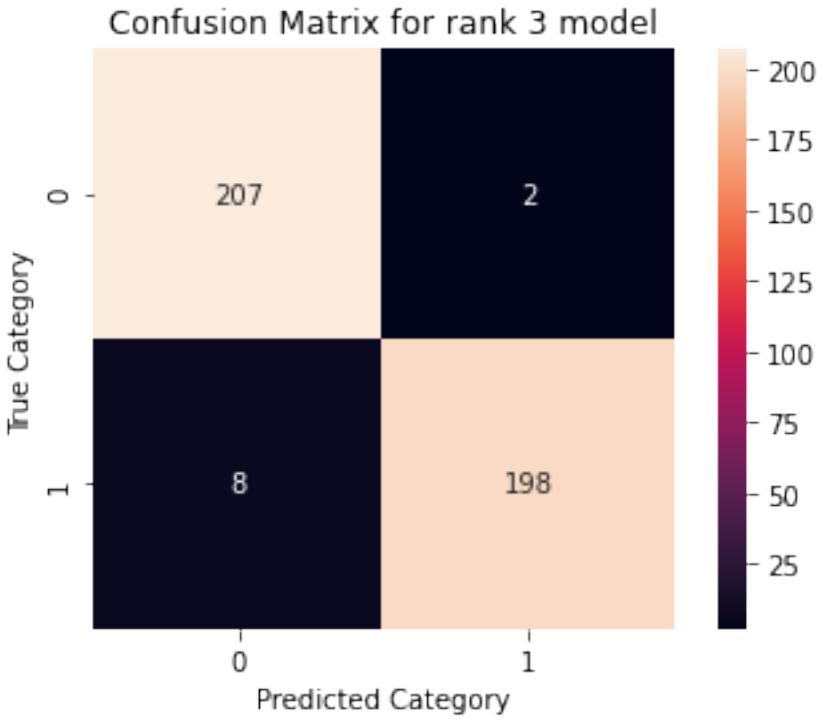



```
accuracy of test data: 0.9734939759036144
recall: 0.9611650485436893
precision: 0.9850746268656716
F-1 Score: 0.9729729729729729
```

rank 3 model: cleaning data, min_df=5, Lemmatization, LSI(k=500), Logistic regression with L2 regularization

[87]:
```
clf=LogisticRegression(penalty='l2', solver='saga', C=10, max_iter=10000, tol=0.
↪0001, random_state=42)
clf.fit(reduced_lsi_train_clean5, train_rootlabel_numbers)
pred_test = clf.predict(reduced_lsi_test_clean5)

#confusion matrix
plt_confusion_matrix(pred_test, 'rank 3 model')
```

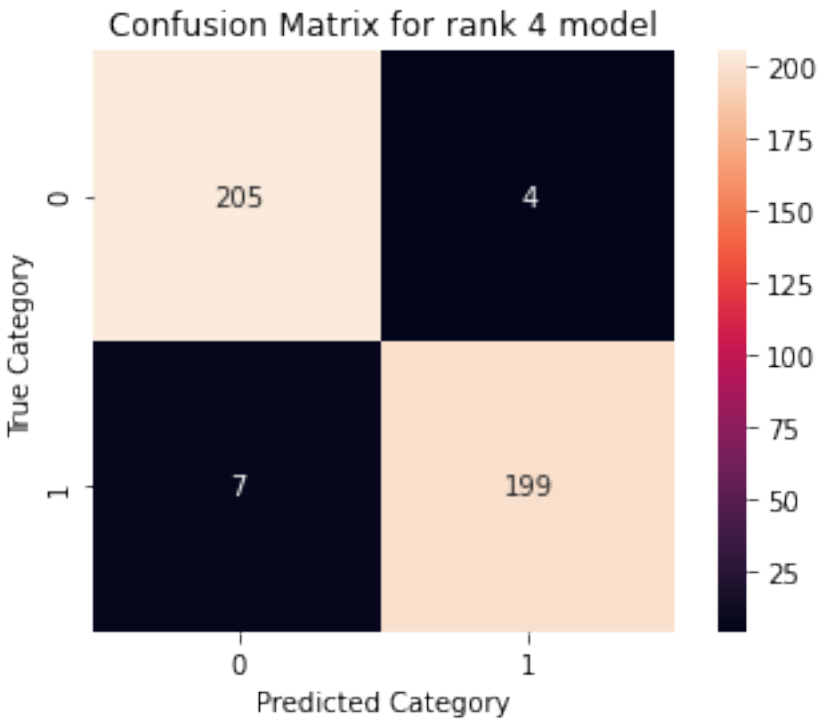
```
#accuracy, recall, precision, F-1 Score
arpf(pred_test)
```




```
accuracy of test data: 0.9759036144578314
recall: 0.9611650485436893
precision: 0.99
F-1 Score: 0.9753694581280788
```

rank 4 model: cleaning data, min_df=5, Lemmatization, LSI(k=500), Logistic regression with L1 regularization

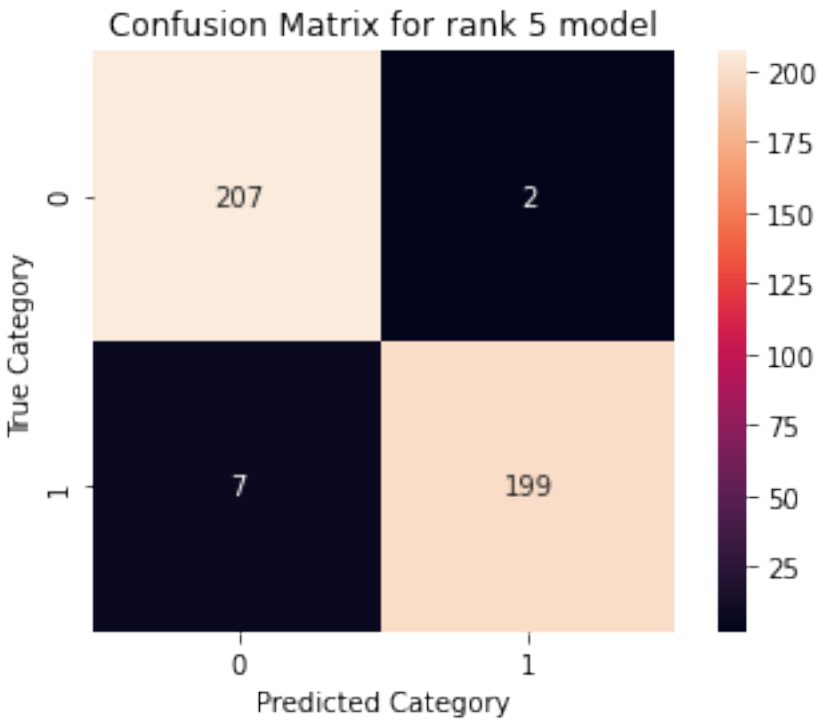
[88]:
```
clf=LogisticRegression(penalty='l1', solver='saga', C=10, max_iter=10000, tol=0.
↪0001, random_state=42)
clf.fit(reduced_lsi_train_clean5, train_rootlabel_numbers)
pred_test = clf.predict(reduced_lsi_test_clean5)

#confusion matrix
plt_confusion_matrix(pred_test, 'rank 4 model')

#accuracy, recall, precision, F-1 Score
arpf(pred_test)
```




```
accuracy of test data: 0.9734939759036144
recall: 0.9660194174757282
precision: 0.9802955665024631
F-1 Score: 0.9731051344743277
```

rank 5 model: not cleaning data, min_df=3, Lemmatization, LSI(k=500), Logistic regression with L2 regularization

[89]:
```
clf=LogisticRegression(penalty='l2', solver='saga', C=10, max_iter=10000, tol=0.
↪0001, random_state=42)
clf.fit(reduced_lsi_train_noclean3, train_rootlabel_numbers)
pred_test = clf.predict(reduced_lsi_test_noclean3)

#confusion matrix
plt_confusion_matrix(pred_test, 'rank 5 model')

#accuracy, recall, precision, F-1 Score
arpf(pred_test)
```




```
accuracy of test data: 0.9783132530120482
recall: 0.9660194174757282
precision: 0.9900497512437811
F-1 Score: 0.977886977886978
```

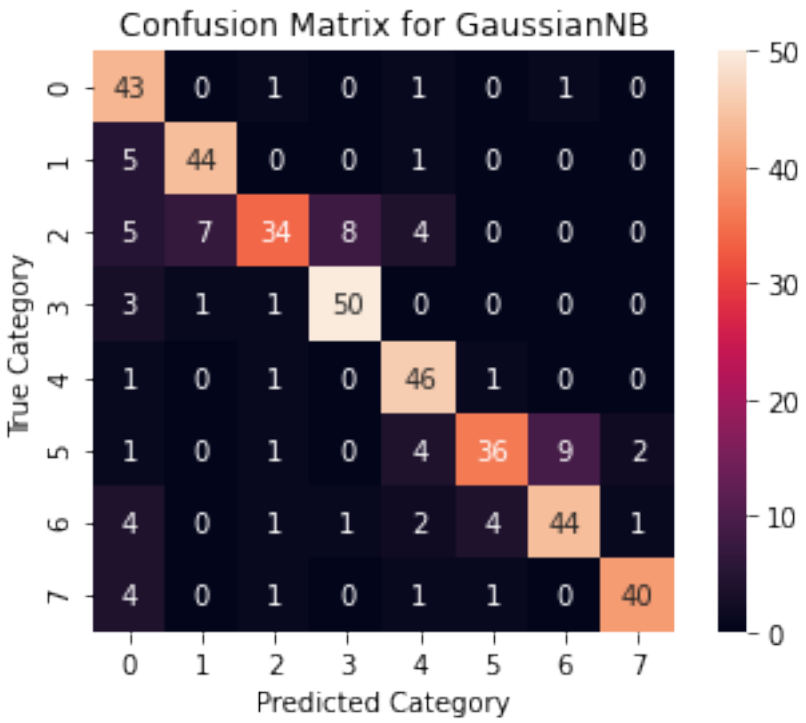
10 Question 9

[19]:
```
###assigning each document leaf label names to numbers
train_leaflabel_categories={0:"chess", 1:"cricket", 2:"soccer", 3:"football", 4:
 ↪"%22forest%20fire%22", 5:"flood", 6:"earthquake", 7:"drought"}
key_list = list(train_leaflabel_categories.keys())
val_list = list(train_leaflabel_categories.values())

#making a list of all document categories using category numbers
train_leaflabel_numbers = []

for row in range(train.shape[0]):
    position = val_list.index(train.iat[row,2])
    train_leaflabel_numbers.append(key_list[position])

train_leaflabel_numbers=np.asarray(train_leaflabel_numbers)
########################################
test_leaflabel_numbers = []
```

```
for row in range(test.shape[0]):
    position = val_list.index(test.iat[row,2])
    test_leaflabel_numbers.append(key_list[position])
```

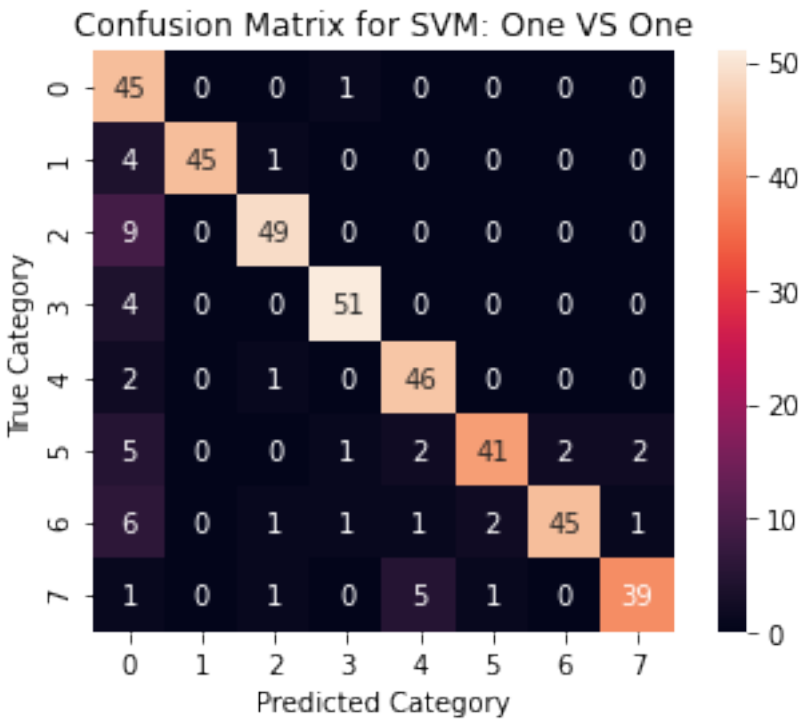
[20]:
```
def plt_confusion_matrix_multiclass(pred_test, txt):
    cmx_data = confusion_matrix(test_leaflabel_numbers, pred_test)
    df_cmx = pd.DataFrame(cmx_data)
    sns.heatmap(df_cmx, fmt='d', annot=True, square=True)
    plt.title('Confusion Matrix for '+txt)
    plt.xlabel('Predicted Category')
    plt.ylabel('True Category')
    plt.show()
def arpf_multiclass(pred_test):
    print("accuracy of test data: "+str(accuracy_score(test_leaflabel_numbers,␣
 ↪pred_test)))
    print("recall: "+str(recall_score(test_leaflabel_numbers, pred_test,␣
 ↪average="macro")))
    print("precision (each category):
 ↪\n"+str(precision_score(test_leaflabel_numbers, pred_test, average="macro")))
    print("F-1 Score: "+str(f1_score(test_leaflabel_numbers, pred_test,␣
 ↪average="macro")))
```

[21]:
```
###Naive-Bayes
clf = GaussianNB()
clf.fit(reduced_lsi_train, train_leaflabel_numbers)
pred_test = clf.predict(reduced_lsi_test)

#confusion matrix
plt_confusion_matrix_multiclass(pred_test, 'GaussianNB')

#accuracy, recall, precision, F-1 Score
arpf_multiclass(pred_test)
```




```
accuracy of test data: 0.8120481927710843
recall: 0.818886857738734
precision (each category):
0.8221222339792502
F-1 Score: 0.8116376305935682
```

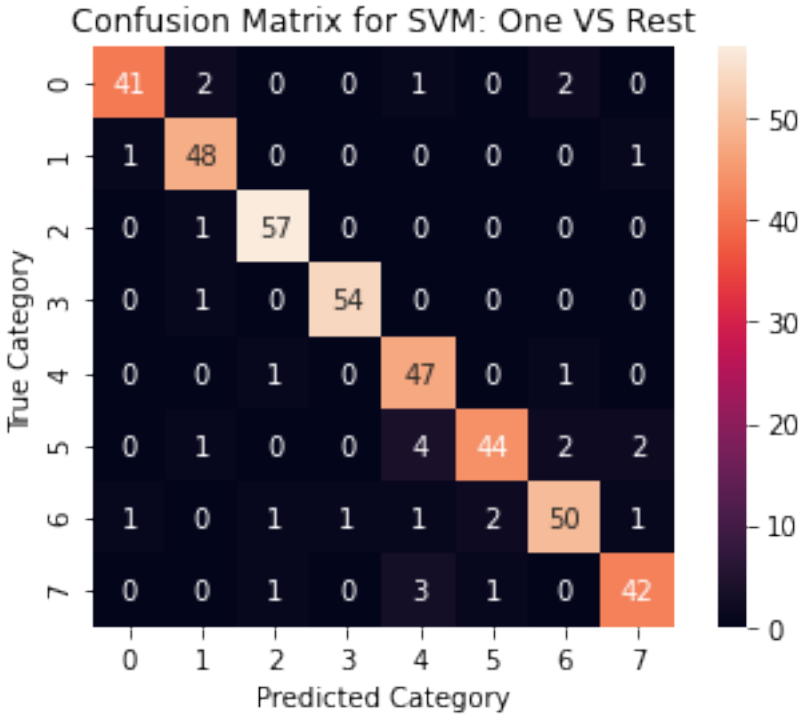
[22]:
```
###SVM: One VS One
clf=SVC(kernel='linear', random_state=42, probability=True,␣
 ↪decision_function_shape='ovo')
clf.fit(reduced_lsi_train, train_leaflabel_numbers)
pred_test = clf.predict(reduced_lsi_test)

#confusion matrix
plt_confusion_matrix_multiclass(pred_test, 'SVM: One VS One')

#accuracy, recall, precision, F-1 Score
arpf_multiclass(pred_test)
```




```
accuracy of test data: 0.8698795180722891
recall: 0.8727478146452975
precision (each category):
0.891345785030154
F-1 Score: 0.8730112320607644
```

[23]:
```
###SVM: One VS Rest using SMOTE
from sklearn.multiclass import OneVsRestClassifier
from imblearn.over_sampling import SMOTE

sm = SMOTE(random_state=42)
reduced_lsi_train_smote, train_leaflabel_numbers_smote = sm.
 ↪fit_resample(reduced_lsi_train, train_leaflabel_numbers)

svc=SVC(kernel='linear', random_state=42, probability=True,␣
 ↪decision_function_shape='ovr')
clf=OneVsRestClassifier(svc)
clf.fit(reduced_lsi_train_smote, train_leaflabel_numbers_smote)
pred_test = clf.predict(reduced_lsi_test)

#confusion matrix
plt_confusion_matrix_multiclass(pred_test, 'SVM: One VS Rest')
```

```
#accuracy, recall, precision, F-1 Score
arpf_multiclass(pred_test)
```




```
accuracy of test data: 0.9228915662650602
recall: 0.9220079383476664
precision (each category):
0.9235696557091433
F-1 Score: 0.9215785088548106
```

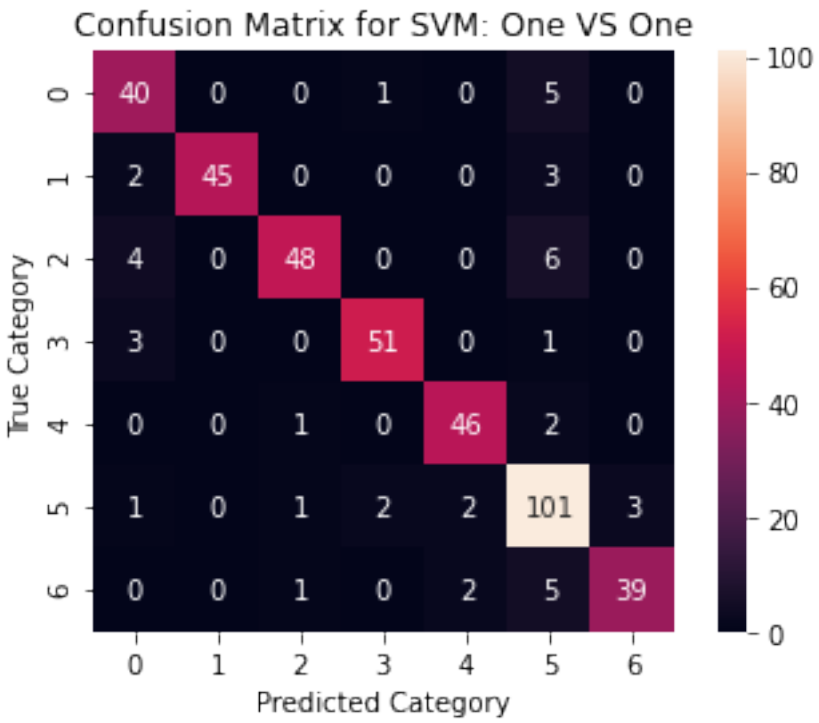
Class imbalance issue was solved using Synthetic Minority Over-sampling Technique (SMOTE.) Since some misclassification is observed between categories 4-7, we attempt to merge them to combined subsets in the following sections. Accuracy increased from 86% (using One VS One) to 93% (using One VS Rest.)

Here, 2 subsets are merged (flood and earthquake):

[24]:
```
###merge subset labels
#merge subset of labels=5:"flood", 6:"earthquake" to 5:"disaster"
train_leaflabel_categories={0:"chess", 1:"cricket", 2:"soccer", 3:"football", 4:
 ↪"%22forest%20fire%22", 5:"disaster", 6:"drought"}
key_list = list(train_leaflabel_categories.keys())
val_list = list(train_leaflabel_categories.values())

#making a list of all document categories using category numbers
```

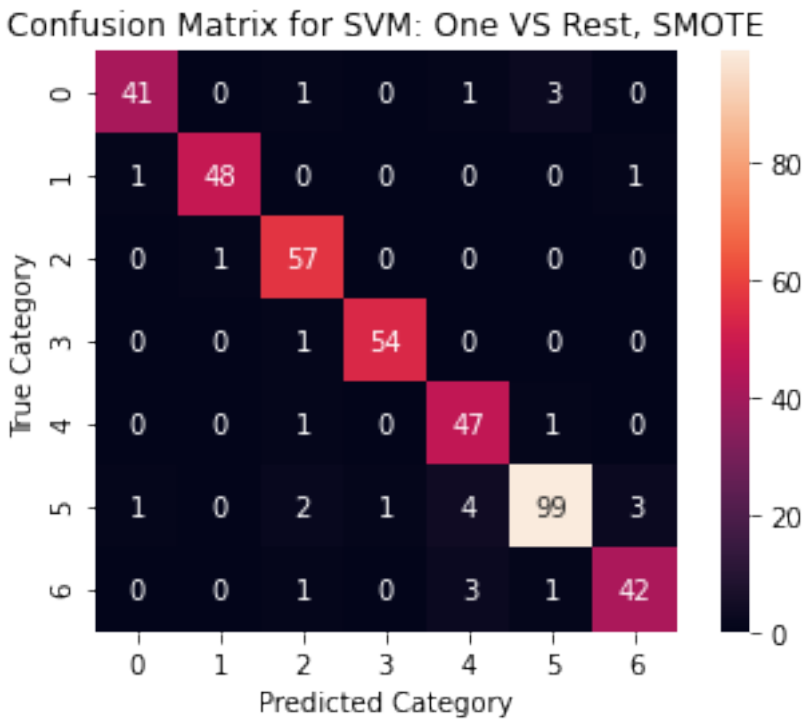
```
train_leaflabel_numbers = []

for row in range(train.shape[0]):
    if train.iat[row,2] == "flood" or train.iat[row,2] == "earthquake":
        position=5
    elif train.iat[row,2] == "drought":
        position=6
    else:
        position = val_list.index(train.iat[row,2])
    train_leaflabel_numbers.append(key_list[position])

train_leaflabel_numbers=np.asarray(train_leaflabel_numbers)
########################################
test_leaflabel_numbers = []

for row in range(test.shape[0]):
    if test.iat[row,2] == "flood" or test.iat[row,2] == "earthquake":
        position=5
    elif test.iat[row,2] == "drought":
        position=6
    else:
        position = val_list.index(test.iat[row,2])
    test_leaflabel_numbers.append(key_list[position])

test_leaflabel_numbers=np.asarray(test_leaflabel_numbers)
```

[25]:
```
###SVM: One VS One
clf=SVC(kernel='linear', random_state=42, probability=True,␣
 ↪decision_function_shape='ovo')
clf.fit(reduced_lsi_train, train_leaflabel_numbers)
pred_test = clf.predict(reduced_lsi_test)

#confusion matrix
plt_confusion_matrix_multiclass(pred_test, 'SVM: One VS One')

#accuracy, recall, precision, F-1 Score
arpf_multiclass(pred_test)
```



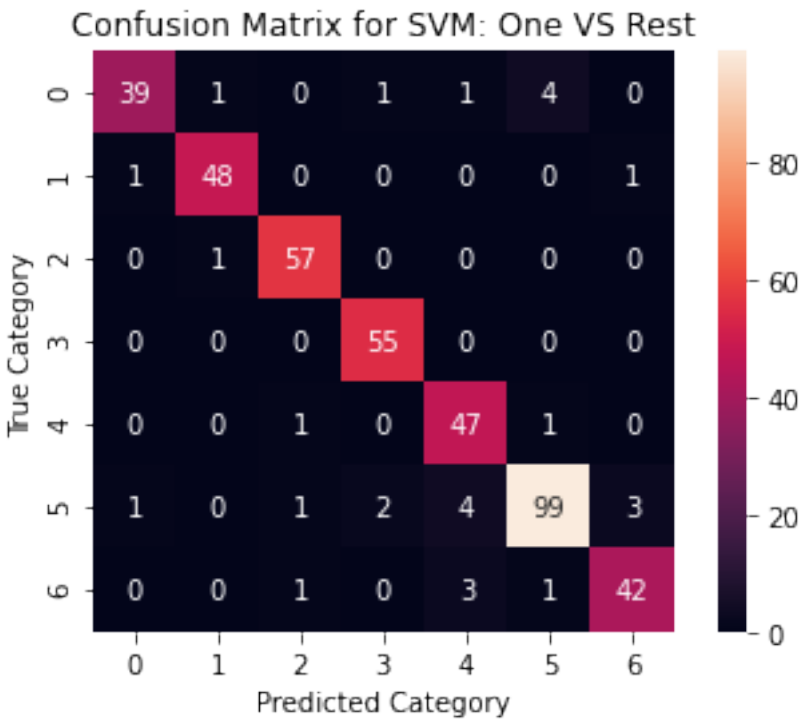

```
accuracy of test data: 0.891566265060241
recall: 0.887309816284148
precision (each category):
0.9079043649980318
F-1 Score: 0.8956951041754825
```

[26]:
```
###SVM: One VS Rest using SMOTE
sm = SMOTE(random_state=42)
reduced_lsi_train_smote, train_leaflabel_numbers_smote = sm.
 ↪fit_resample(reduced_lsi_train, train_leaflabel_numbers)

svc=SVC(kernel='linear', random_state=42, probability=True)
clf=OneVsRestClassifier(svc)
clf.fit(reduced_lsi_train_smote, train_leaflabel_numbers_smote)
pred_test = clf.predict(reduced_lsi_test)

#confusion matrix
plt_confusion_matrix_multiclass(pred_test, 'SVM: One VS Rest, SMOTE')

#accuracy, recall, precision, F-1 Score
arpf_multiclass(pred_test)
```




```
accuracy of test data: 0.9349397590361446
recall: 0.9383831207257012
precision (each category):
0.9341674721624578
F-1 Score: 0.9353311188014359
```

[27]:
```
###SVM: One VS Rest with class imbalance

svc=SVC(kernel='linear', random_state=42, probability=True)
clf=OneVsRestClassifier(svc)
clf.fit(reduced_lsi_train, train_leaflabel_numbers)
pred_test = clf.predict(reduced_lsi_test)

#confusion matrix
plt_confusion_matrix_multiclass(pred_test, 'SVM: One VS Rest')

#accuracy, recall, precision, F-1 Score
arpf_multiclass(pred_test)
```



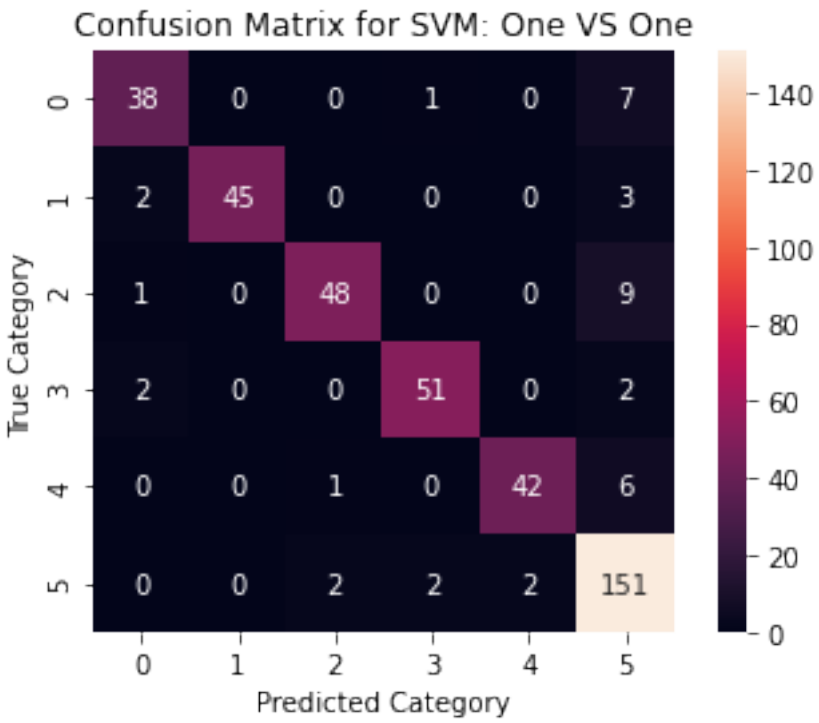

```
accuracy of test data: 0.9325301204819277
recall: 0.93476934319888
precision (each category):
0.9314202071325077
F-1 Score: 0.9320152770489598
```

In the following, 3 subsets are merged (flood, earthquake, drought):

[28]:
```
###merge subset labels
#merge subset of labels=5:"flood", 6:"earthquake", 7:"drought" to 5:"disaster"
train_leaflabel_categories={0:"chess", 1:"cricket", 2:"soccer", 3:"football", 4:
 ↪"%22forest%20fire%22", 5:"disaster"}
key_list = list(train_leaflabel_categories.keys())
val_list = list(train_leaflabel_categories.values())

#making a list of all document categories using category numbers
train_leaflabel_numbers = []

for row in range(train.shape[0]):
    if train.iat[row,2] == "flood" or train.iat[row,2] == "earthquake" or train.
 ↪iat[row,2] =="drought":
        position=5
    else:
        position = val_list.index(train.iat[row,2])
```

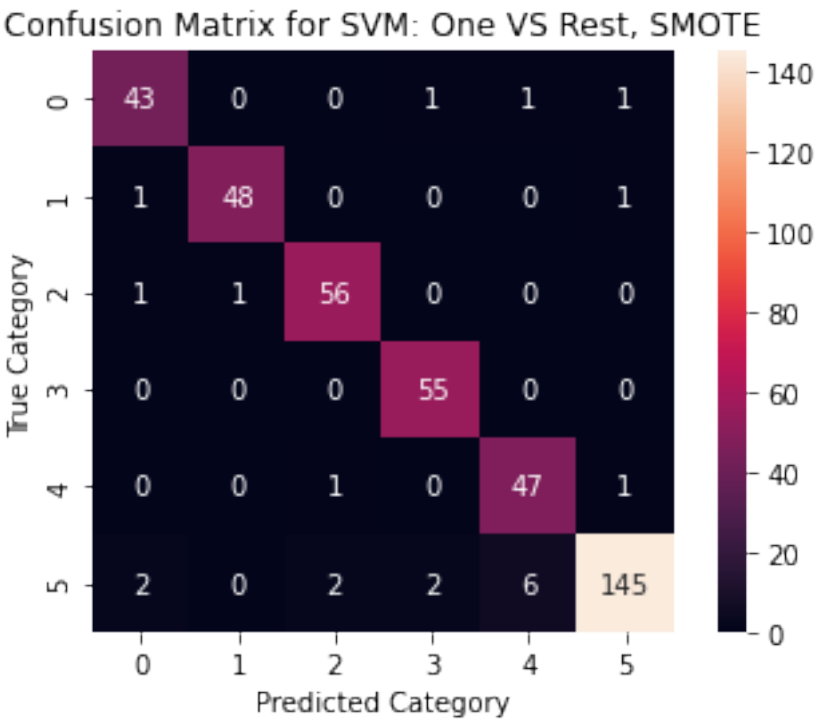
```
    train_leaflabel_numbers.append(key_list[position])

train_leaflabel_numbers=np.asarray(train_leaflabel_numbers)
########################################
test_leaflabel_numbers = []

for row in range(test.shape[0]):
    if test.iat[row,2] == "flood" or test.iat[row,2] == "earthquake" or test.
 ↪iat[row,2] == "drought":
        position=5
    else:
        position = val_list.index(test.iat[row,2])
    test_leaflabel_numbers.append(key_list[position])

test_leaflabel_numbers=np.asarray(test_leaflabel_numbers)
```

[29]:
```
###SVM: One VS One
clf=SVC(kernel='linear', random_state=42, probability=True,␣
 ↪decision_function_shape='ovo')
clf.fit(reduced_lsi_train, train_leaflabel_numbers)
pred_test = clf.predict(reduced_lsi_test)

#confusion matrix
plt_confusion_matrix_multiclass(pred_test, 'SVM: One VS One')

#accuracy, recall, precision, F-1 Score
arpf_multiclass(pred_test)
```



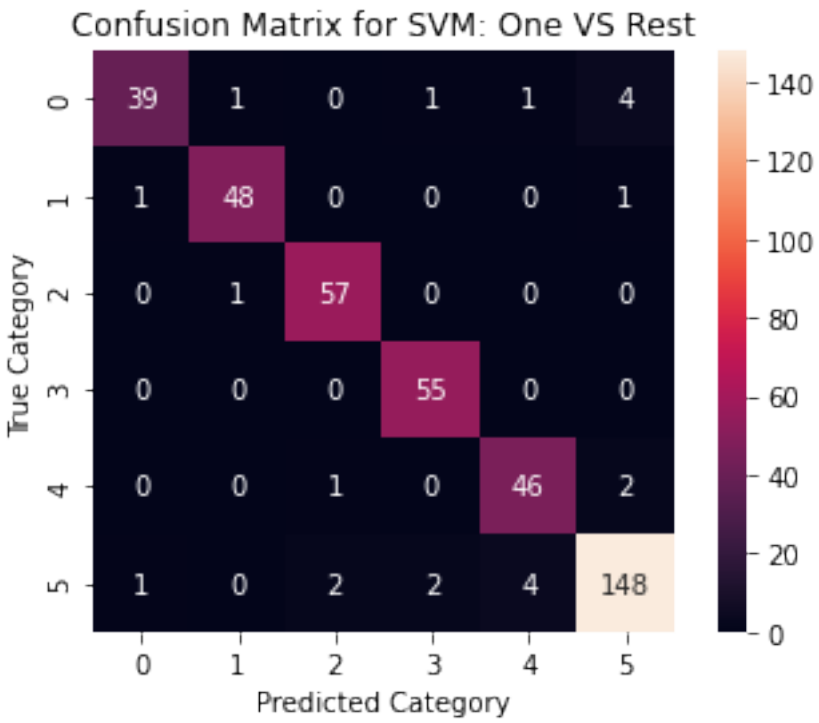

```
accuracy of test data: 0.9036144578313253
recall: 0.88331203122072
precision (each category):
0.928700317758711
F-1 Score: 0.9037555184258718
```

[30]:
```
###SVM: One VS Rest
sm = SMOTE(random_state=42)
reduced_lsi_train_smote, train_leaflabel_numbers_smote = sm.
 ↪fit_resample(reduced_lsi_train, train_leaflabel_numbers)

svc=SVC(kernel='linear', random_state=42, probability=True)
clf=OneVsRestClassifier(svc)
clf.fit(reduced_lsi_train_smote, train_leaflabel_numbers_smote)
pred_test = clf.predict(reduced_lsi_test)

#confusion matrix
plt_confusion_matrix_multiclass(pred_test, 'SVM: One VS Rest, SMOTE')

#accuracy, recall, precision, F-1 Score
arpf_multiclass(pred_test)
```




```
accuracy of test data: 0.9493975903614458
recall: 0.9571750670875404
precision (each category):
0.9403356597163196
F-1 Score: 0.9480975780909504
```

[31]:
```
###SVM: One VS Rest with class imbalance

svc=SVC(kernel='linear', random_state=42, probability=True)
clf=OneVsRestClassifier(svc)
clf.fit(reduced_lsi_train, train_leaflabel_numbers)
pred_test = clf.predict(reduced_lsi_test)

#confusion matrix
plt_confusion_matrix_multiclass(pred_test, 'SVM: One VS Rest')

#accuracy, recall, precision, F-1 Score
arpf_multiclass(pred_test)
```



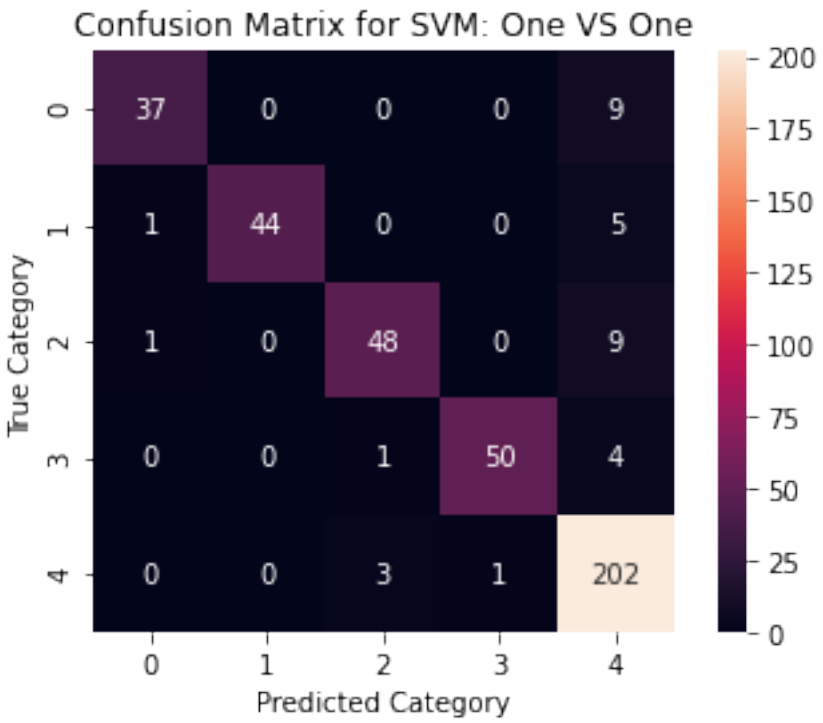

```
accuracy of test data: 0.946987951807229
recall: 0.9453392295143211
precision (each category):
0.9443824780425388
F-1 Score: 0.9441371158674604
```

In the following, 4 subsets are merged (forest fire, flood, earthquake, drought):

[33]:
```
###merge subset labels
#merge subset of labels=5:"flood", 6:"earthquake", 7:"drought" to 5:"disaster"
train_leaflabel_categories={0:"chess", 1:"cricket", 2:"soccer", 3:"football", 4:
 ↪"disaster"}
key_list = list(train_leaflabel_categories.keys())
val_list = list(train_leaflabel_categories.values())

#making a list of all document categories using category numbers
train_leaflabel_numbers = []

for row in range(train.shape[0]):
    if train.iat[row,2] == "%22forest%20fire%22" or train.iat[row,2] == "flood"␣
 ↪or train.iat[row,2] == "earthquake" or train.iat[row,2] =="drought":
        position=4
    else:
        position = val_list.index(train.iat[row,2])
```

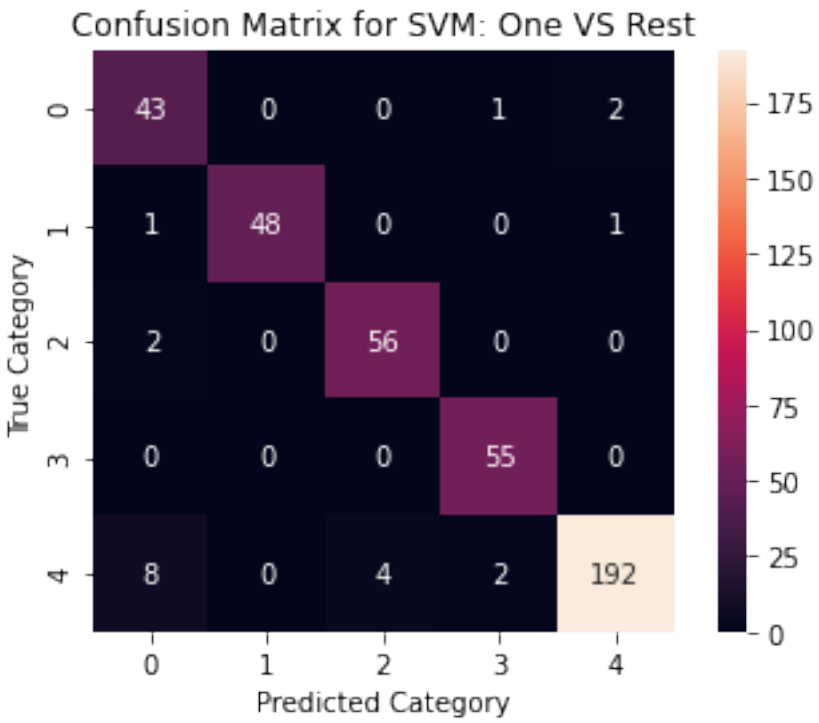
```
    train_leaflabel_numbers.append(key_list[position])

train_leaflabel_numbers=np.asarray(train_leaflabel_numbers)
########################################
test_leaflabel_numbers = []

for row in range(test.shape[0]):
    if test.iat[row,2] == "%22forest%20fire%22" or test.iat[row,2] == "flood"␣
 ↪or test.iat[row,2] == "earthquake" or test.iat[row,2] == "drought":
        position=4
    else:
        position = val_list.index(test.iat[row,2])
    test_leaflabel_numbers.append(key_list[position])

test_leaflabel_numbers=np.asarray(test_leaflabel_numbers)
```

[34]:
```
###SVM: One VS One
clf=SVC(kernel='linear', random_state=42, probability=True,␣
 ↪decision_function_shape='ovo')
clf.fit(reduced_lsi_train, train_leaflabel_numbers)
pred_test = clf.predict(reduced_lsi_test)

#confusion matrix
plt_confusion_matrix_multiclass(pred_test, 'SVM: One VS One')

#accuracy, recall, precision, F-1 Score
arpf_multiclass(pred_test)
```



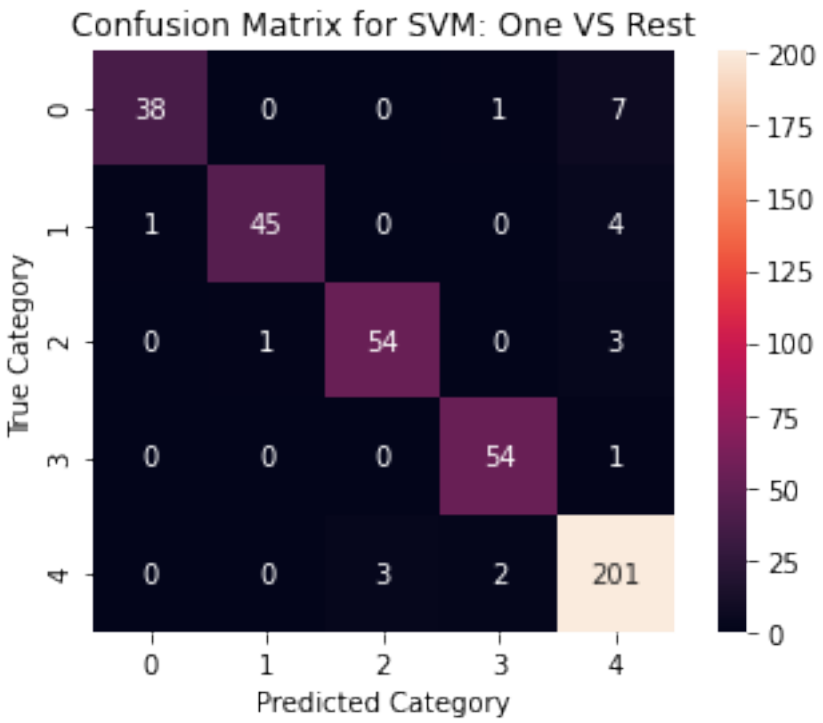

```
accuracy of test data: 0.9180722891566265
recall: 0.8803214932692522
precision (each category):
0.9468566197053224
F-1 Score: 0.9103235158772701
```

[35]:
```
###SVM: One VS Rest
sm = SMOTE(random_state=42)
reduced_lsi_train_smote, train_leaflabel_numbers_smote = sm.
 ↪fit_resample(reduced_lsi_train, train_leaflabel_numbers)

svc=SVC(kernel='linear', random_state=42, probability=True)
clf=OneVsRestClassifier(svc)
clf.fit(reduced_lsi_train_smote, train_leaflabel_numbers_smote)
pred_test = clf.predict(reduced_lsi_test)

#confusion matrix
plt_confusion_matrix_multiclass(pred_test, 'SVM: One VS Rest')

#accuracy, recall, precision, F-1 Score
arpf_multiclass(pred_test)
```



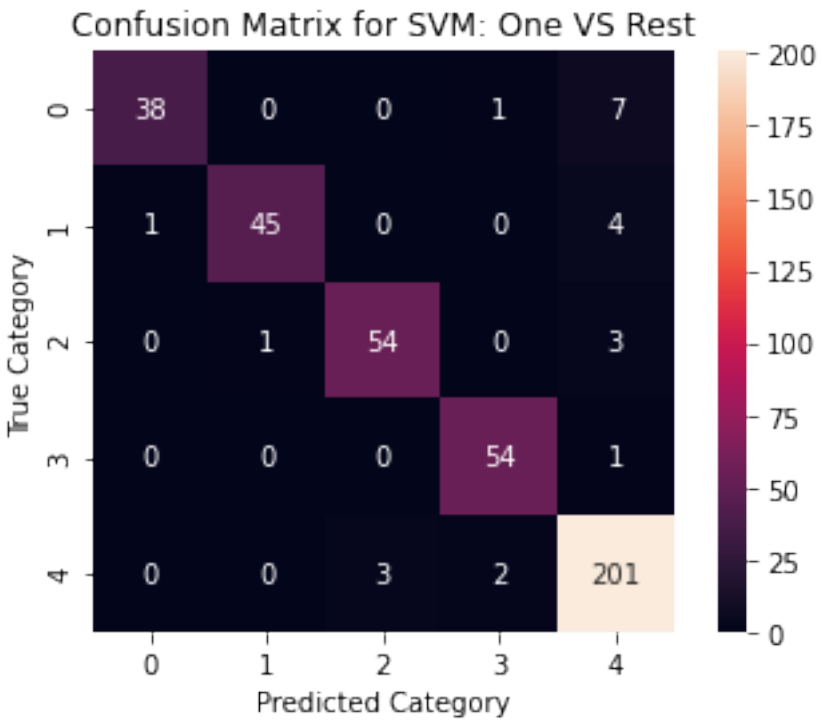

```
accuracy of test data: 0.9493975903614458
recall: 0.9584677370052838
precision (each category):
0.9325041752627958
F-1 Score: 0.943960338315722
```

[30]:
```
###SVM: One VS Rest with class imbalance

svc=SVC(kernel='linear', random_state=42, probability=True)
clf=OneVsRestClassifier(svc)
clf.fit(reduced_lsi_train, train_leaflabel_numbers)
pred_test = clf.predict(reduced_lsi_test)

#confusion matrix
plt_confusion_matrix_multiclass(pred_test, 'SVM: One VS Rest')

#accuracy, recall, precision, F-1 Score
arpf_multiclass(pred_test)
```




```
accuracy of test data: 0.944578313253012
recall: 0.9229335552876694
precision (each category):
0.9555824483170021
F-1 Score: 0.9375280862396472
```

[36]:
```
###SVM: One VS Rest with class imbalance

svc=SVC(kernel='linear', random_state=42, probability=True)
clf=OneVsRestClassifier(svc)
clf.fit(reduced_lsi_train, train_leaflabel_numbers)
pred_test = clf.predict(reduced_lsi_test)

#confusion matrix
plt_confusion_matrix_multiclass(pred_test, 'SVM: One VS Rest')

#accuracy, recall, precision, F-1 Score
arpf_multiclass(pred_test)
```




```
accuracy of test data: 0.944578313253012
recall: 0.9229335552876694
precision (each category):
0.9555824483170021
F-1 Score: 0.9375280862396472
```

Once some classes are merged, there is increased disparity in the accuracy levels between the classifiers that remove class imbalance and the ones that don't. Hence, the resolution is approximately when there is less than 5 categories. With less than 5, apparent difference is observed between class imbalanced data and SMOTE processed data.

11 Question 10

(a) It is because the ratio of co-occurrence probabilities are more effective to seperate the groups of contexual words. For these groups of words which are related to the target word in the co-occurrence ratio, the ratio will be either very large or very small. On the other hand, the probabilities themselves will be affected by the noise and become difficult to find relevant words from corpus.

(b) It will not return the same vector because it uses conditional probability ratios to represent relationships with different words. For example, in these two cases, the relationships with 'presidency' and 'park' are different.

[50]:
```
#Create a folder named 'glove' in the project directorty and put 'glove.6B.300d.
 ↪txt' into it
embeddings_dict = {}
dimension_of_glove = 300
with open("glove/glove.6B.300d.txt", 'r') as f:
    for line in f:
        values = line.split()
        word = values[0]
        vector = np.asarray(values[1:], "float32")
        embeddings_dict[word] = vector
```

[51]:
```
#(c) Practical
from numpy.linalg import norm
print("Value1: {}".format(norm(embeddings_dict['queen'] -␣
 ↪embeddings_dict['king'] - embeddings_dict['wife'] +␣
 ↪embeddings_dict['husband'])))
print("Value2: {}".format(norm(embeddings_dict['queen'] -␣
 ↪embeddings_dict['king'])))
print("Value3: {}".format(norm(embeddings_dict['wife'] -␣
 ↪embeddings_dict['husband'])))
```

```
Value1: 6.165036201477051
Value2: 5.9662580490112305
Value3: 3.1520464420318604
```

(c) Expected (ideal): ||queen - king - wife + husband||2 = ||(queen - king) - (wife - husband)||2 = ||(female) - (female)||2 = 0 It is because a possible contextual word embedding transforms 'king' to 'queen' and 'husband' to 'wife' is the word 'female'.

Practical: ||queen - king - wife + husband||2 = 6.16504 ||queen - king||2 = 5.96626 ||wife - husband||2 = 3.15205

(d) We would choose lemmatization. Even though stemming is faster than lemmatization, lemmatization considers the syntax, context, etc. It usually has the lower error rate than stemming.

12 Question 11

(a) First, we choose the column keywords as our features, split the data into 'keywords' and 'rootlabel', tokenize each keyword in keywords in each sample, generate a matrix mxn (m=the number of samples, n=300 in this case), add each vector of the keywords of one sample in the embedding-dictionary, and normalize each row of the vector.

(b) We select SVM model and use cross-validation to find the best parameter, gamma. Gamma shoud be 1 in this case (n=300).

[52]:
```
np.random.seed(42)
random.seed(42)
```

```
glove_train, glove_test = train_test_split(data[["keywords","root_label"]],␣
 ↪test_size=0.2)
print("training samples = "+str(len(glove_train))+"\n"+"test samples =␣
 ↪"+str(len(glove_test)))
```

```
training samples = 1657
test samples = 415
```

[53]:
```
train_keywords = []
test_keywords = []

for row in range(glove_train.shape[0]):
    train_keywords.append(glove_train.iat[row, 0])
for row in range(glove_test.shape[0]):
    test_keywords.append(glove_test.iat[row, 0])

glove_train_keywords = []
glove_test_keywords = []

for keywords in train_keywords:
    keywords = keywords.replace('\'', '')
    keywords = keywords.replace(',', '')
    keywords = keywords.replace('[', '')
    keywords = keywords.replace(']', '')
    wordlist = []
    for keyword in nltk.word_tokenize(keywords):
        if keyword in embeddings_dict:
            wordlist.append(keyword)
    glove_train_keywords.append(wordlist)

for keywords in test_keywords:
    keywords = keywords.replace('\'', '')
    keywords = keywords.replace(',', '')
    keywords = keywords.replace('[', '')
    keywords = keywords.replace(']', '')
    wordlist = []
    for keyword in nltk.word_tokenize(keywords):
        if keyword in embeddings_dict:
            wordlist.append(keyword)
    glove_test_keywords.append(wordlist)
```

[54]:
```
from sklearn.preprocessing import normalize

rows = glove_train.shape[0]
cols = dimension_of_glove
glove_train_matrix = np.zeros(shape=(rows, cols))
```

```
for row in range(rows):
    for keyword in glove_train_keywords[row]:
        if keyword in embeddings_dict:
            glove_train_matrix[row] += embeddings_dict[keyword]
    glove_train_matrix[row] = normalize(glove_train_matrix[row][:,np.newaxis],␣
 ↪axis=0).ravel()

rows = glove_test.shape[0]
glove_test_matrix = np.zeros(shape=(rows, cols))

for row in range(rows):
    for keyword in glove_test_keywords[row]:
        if keyword in embeddings_dict:
            glove_test_matrix[row] += embeddings_dict[keyword]
    glove_test_matrix[row] = normalize(glove_test_matrix[row][:,np.newaxis],␣
 ↪axis=0).ravel()
```

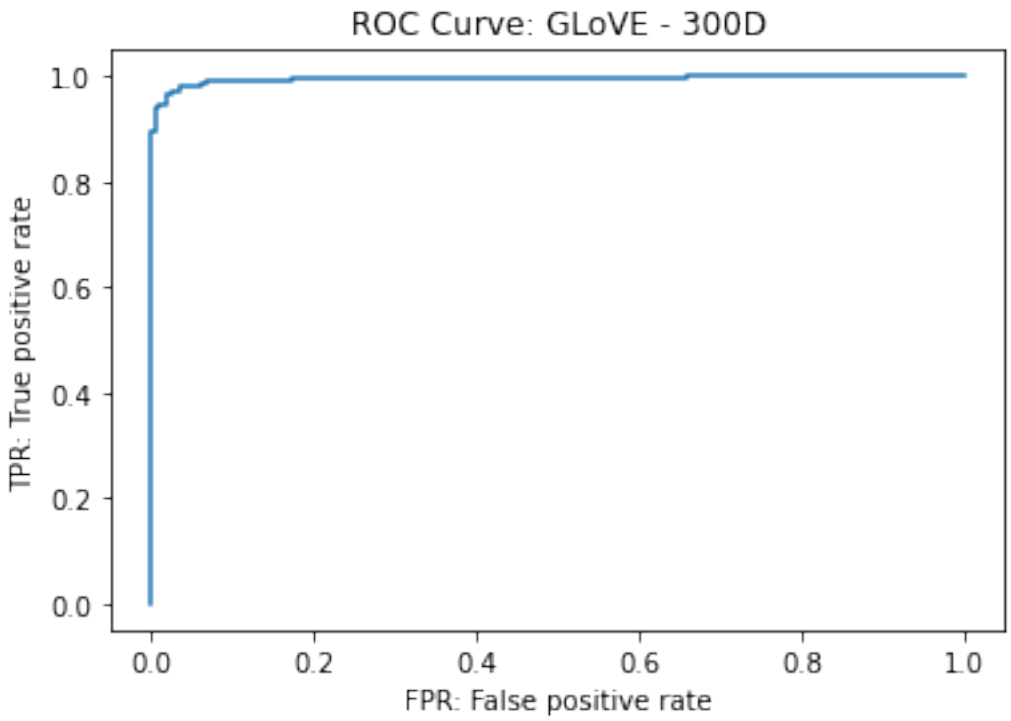
[55]:
```
###assigning each document category names to category numbers
#making a list of all category types
#0:sports; 1:climate
glove_train_rootlabel_categories = data['root_label'].value_counts().index.
 ↪tolist()

#making a list of all document categories using category numbers
glove_train_rootlabel_numbers = []
for row in range(glove_train.shape[0]):
    for i, s in enumerate(glove_train_rootlabel_categories):
        if glove_train.iat[row, 1] == s:
            glove_train_rootlabel_numbers.append(i)
            break
glove_train_rootlabel_numbers = np.asarray(glove_train_rootlabel_numbers)

glove_test_rootlabel_numbers = []
for row in range(glove_test.shape[0]):
    for i, s in enumerate(glove_train_rootlabel_categories):
        if glove_test.iat[row, 1] == s:
            glove_test_rootlabel_numbers.append(i)
            break
glove_test_rootlabel_numbers = np.asarray(glove_test_rootlabel_numbers)
```

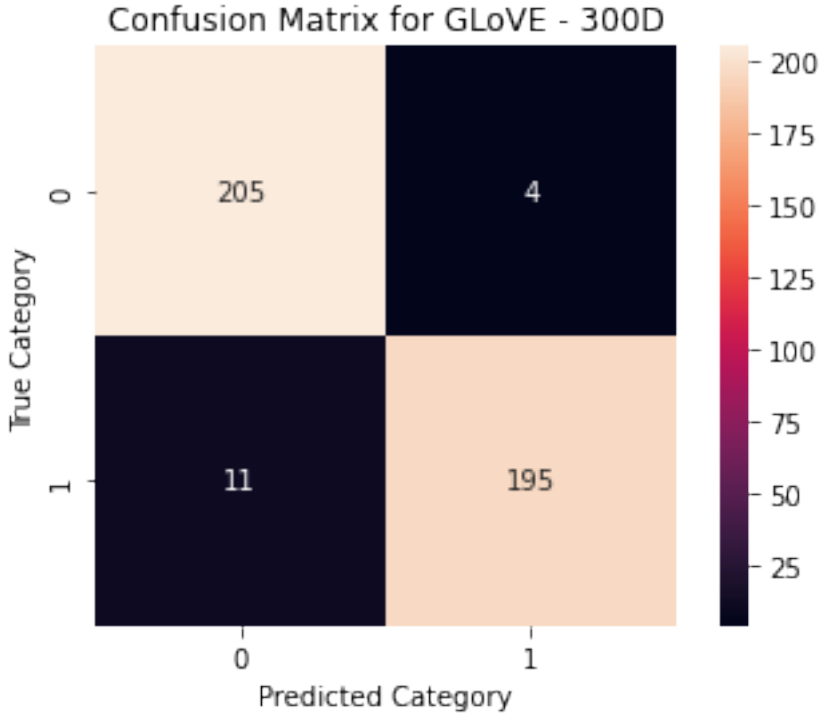
[56]:
```
def plt_roc_curve_glove(model, txt):
    prob = model.predict_proba(glove_test_matrix)[:,1]
    fpr, tpr, thresholds = roc_curve(glove_test_rootlabel_numbers, prob)
    plt.plot(fpr, tpr)
    plt.title("ROC Curve: "+txt)
    plt.xlabel('FPR: False positive rate')
    plt.ylabel('TPR: True positive rate')
```

```
    plt.show()

def plt_confusion_matrix_glove(pred_test, txt):
    cmx_data = confusion_matrix(glove_test_rootlabel_numbers, pred_test)
    df_cmx = pd.DataFrame(cmx_data)
    sns.heatmap(df_cmx, fmt='d', annot=True, square=True)
    plt.title('Confusion Matrix for '+txt)
    plt.xlabel('Predicted Category')
    plt.ylabel('True Category')
    plt.show()

def arpf_glove(pred_test):
    print("accuracy of test data:␣
 ↪"+str(accuracy_score(glove_test_rootlabel_numbers, pred_test)))
    print("recall: "+str(recall_score(glove_test_rootlabel_numbers, pred_test)))
    print("precision: "+str(precision_score(glove_test_rootlabel_numbers,␣
 ↪pred_test)))
    print("F-1 Score: "+str(f1_score(glove_test_rootlabel_numbers, pred_test)))
```

[57]:
```
candidate_params = {'C': [10**i for i in range(-3, 7)]}

svc_glove = SVC(kernel='linear', random_state=42, probability=True)
clf_glove = GridSearchCV(estimator=svc_glove, param_grid=candidate_params,␣
 ↪cv=5, n_jobs=-1)

clf_glove.fit(glove_train_matrix, glove_train_rootlabel_numbers)
print("Best Parameters: "+str(clf_glove.best_params_))
print("Best Score: "+str(clf_glove.best_score_))

#training classifier and predicting
clf_glove = SVC(kernel='linear', random_state=0, C=clf_glove.best_params_['C'],␣
 ↪probability=True)
clf_glove.fit(glove_train_matrix, glove_train_rootlabel_numbers)

pred_test_glove = clf_glove.predict(glove_test_matrix)

#ROC curve
plt_roc_curve_glove(clf_glove, 'GLoVE - 300D')

#confusion matrix
plt_confusion_matrix_glove(pred_test_glove, 'GLoVE - 300D')

#accuracy, recall, precision, F-1 Score
arpf_glove(pred_test_glove)
```

```
Best Parameters: {'C': 1}
Best Score: 0.9444672951625233
```


ROC Curve: GLoVE - 300D
TPR: True positive rate
FPR: False positive rate
0.0
0.2
0.4
0.6
0.8
1.0


Confusion Matrix for GLoVE - 300D
True Category
Predicted Category
205
4
11
195
0
1
200
175
150
125
100
75
50
25

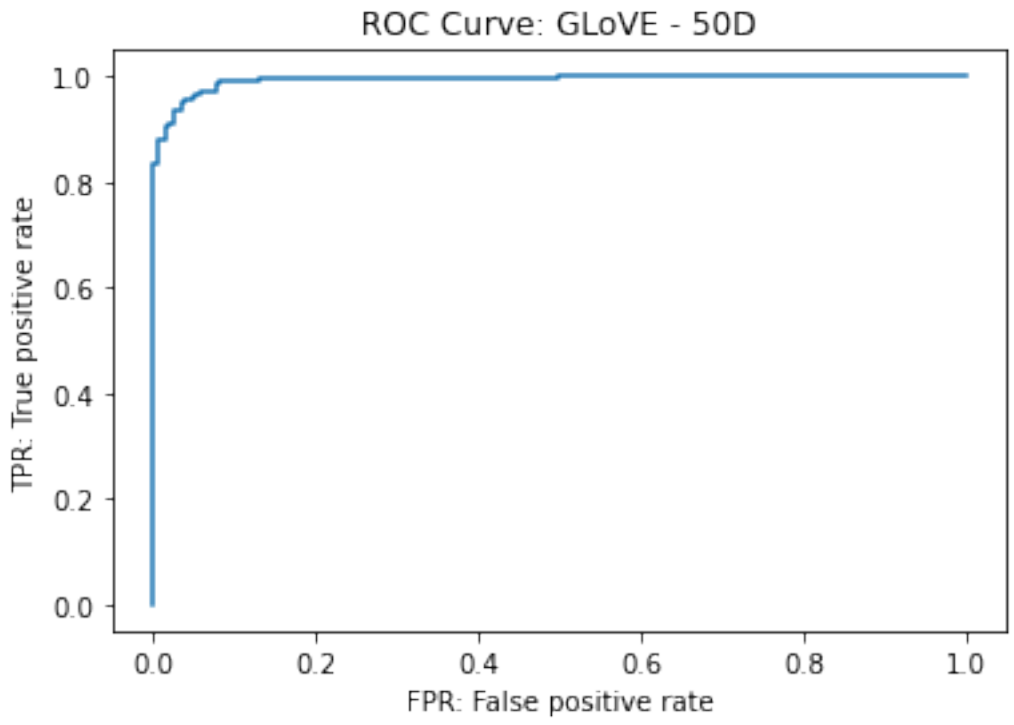
```
accuracy of test data: 0.963855421686747
recall: 0.9466019417475728
precision: 0.9798994974874372
F-1 Score: 0.9629629629629629
```

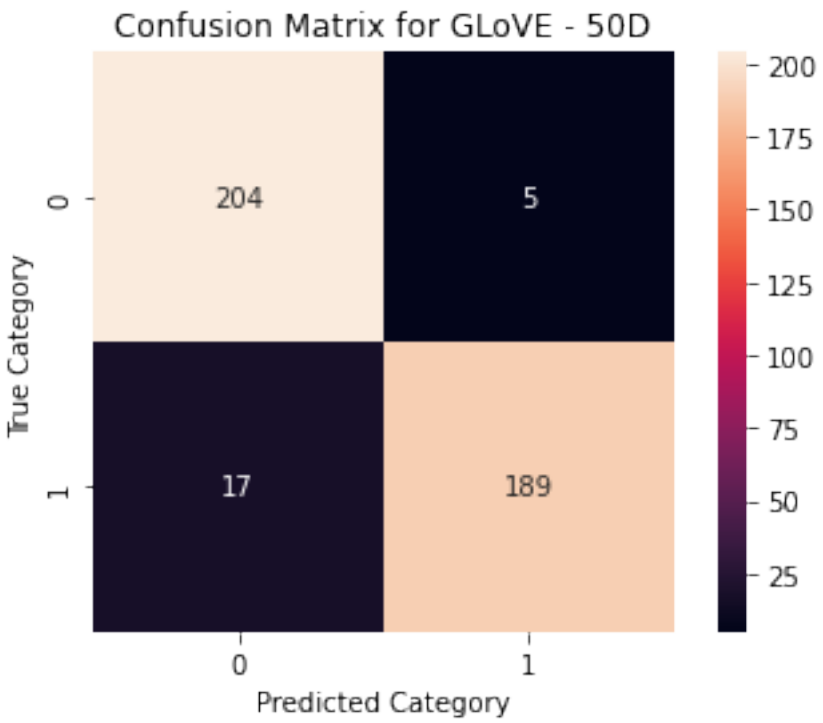
13 Question 12

[58]:
```
filenames = ['glove.6B.50d.txt', 'glove.6B.100d.txt', 'glove.6B.200d.txt',␣
 ↪'glove.6B.300d.txt']
dimensions = [50, 100, 200, 300]
accuracy = []

for i in range(4):
    #Step1: Loading
    print("\nTraining {} ......".format(filenames[i]))
    embeddings_dict = {}
    dimension_of_glove = dimensions[i]
    with open("glove/"+filenames[i], 'r') as f:
        for line in f:
            values = line.split()
            word = values[0]
            vector = np.asarray(values[1:], "float32")
            embeddings_dict[word] = vector

    #Step2: Constructing Matrix
    rows = glove_train.shape[0]
    cols = dimension_of_glove
    glove_train_matrix = np.zeros(shape=(rows, cols))
    for row in range(rows):
        for keyword in glove_train_keywords[row]:
            if keyword in embeddings_dict:
                glove_train_matrix[row] += embeddings_dict[keyword]
        glove_train_matrix[row] = normalize(glove_train_matrix[row][:,np.
 ↪newaxis], axis=0).ravel()

    rows = glove_test.shape[0]
    glove_test_matrix = np.zeros(shape=(rows, cols))
    for row in range(rows):
        for keyword in glove_test_keywords[row]:
            if keyword in embeddings_dict:
                glove_test_matrix[row] += embeddings_dict[keyword]
        glove_test_matrix[row] = normalize(glove_test_matrix[row][:,np.
 ↪newaxis], axis=0).ravel()

    #Step3: Training
    #training classifier and predicting
```

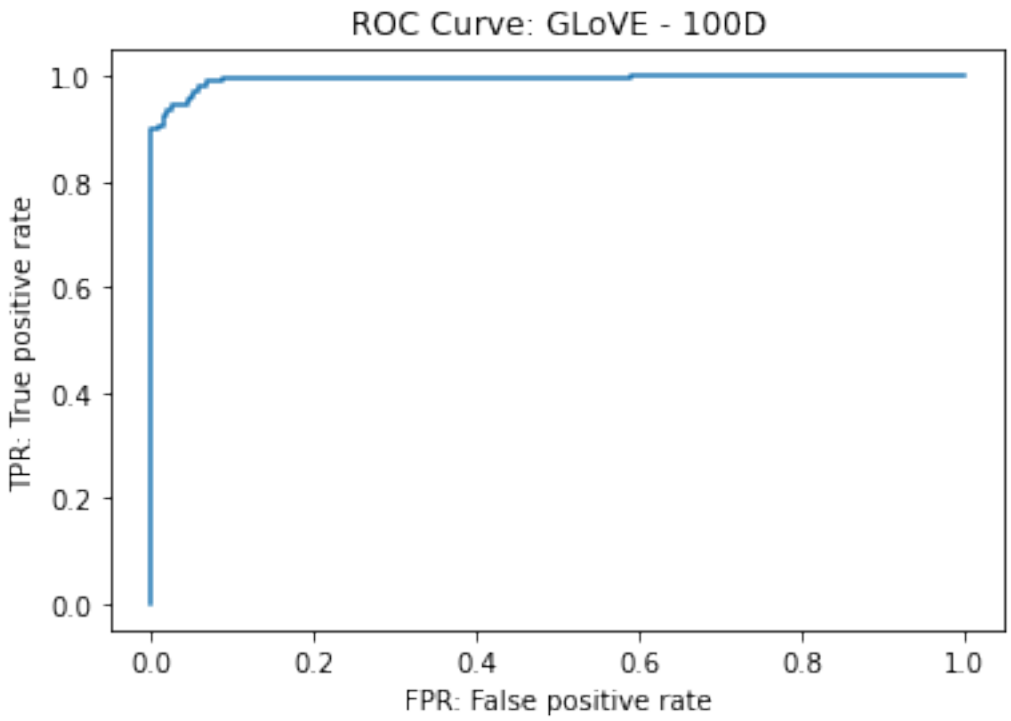
```
    clf_glove = SVC(kernel='linear', random_state=42, C=1, probability=True)
    clf_glove.fit(glove_train_matrix, glove_train_rootlabel_numbers)
    pred_test_glove = clf_glove.predict(glove_test_matrix)

    #ROC curve
    plt_roc_curve_glove(clf_glove, 'GLoVE - '+str(dimension_of_glove)+'D')

    #confusion matrix
    plt_confusion_matrix_glove(pred_test_glove, 'GLoVE -␣
 ↪'+str(dimension_of_glove)+'D')

    #accuracy, recall, precision, F-1 Score
    arpf_glove(pred_test_glove)
    accuracy.append(accuracy_score(glove_test_rootlabel_numbers,␣
 ↪pred_test_glove))

print("\nAccuracy: {}\n".format(accuracy))
```

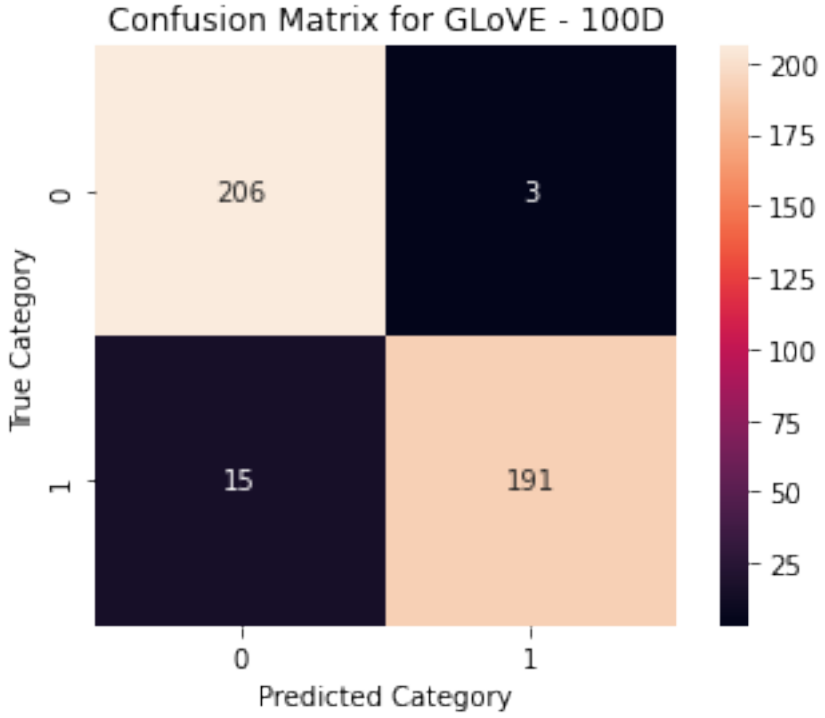
```
Training glove.6B.50d.txt …
```



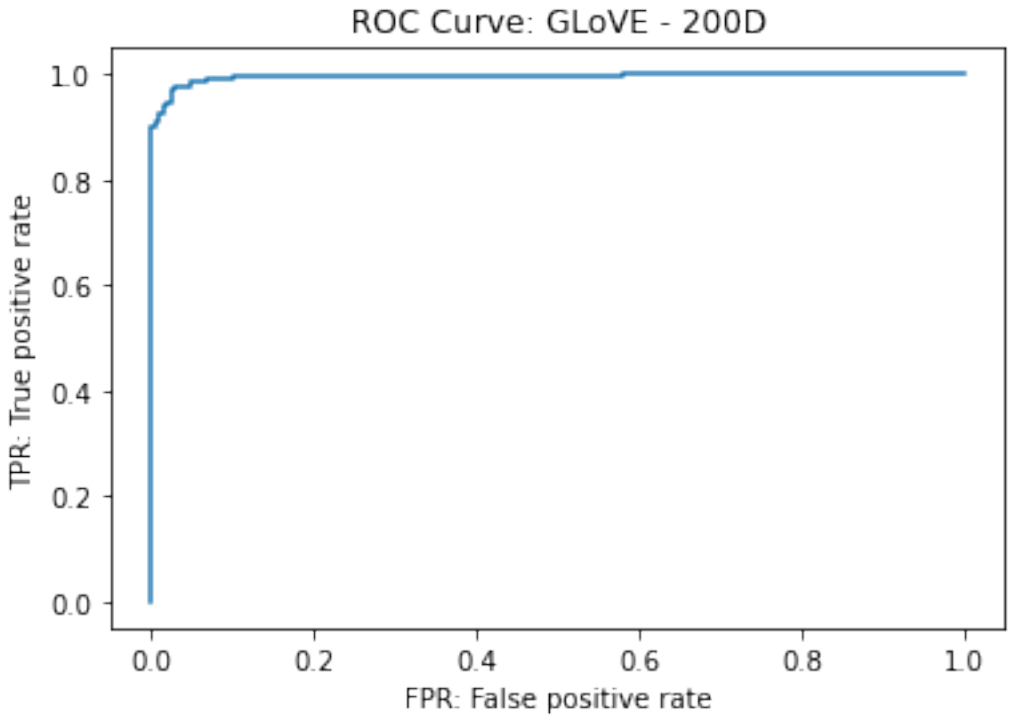


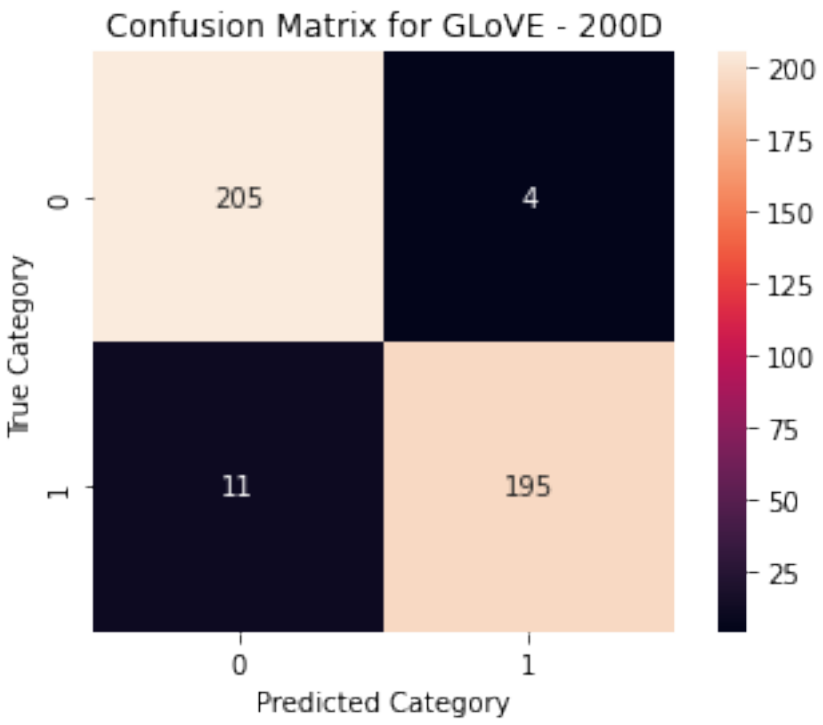

```
accuracy of test data: 0.946987951807229
recall: 0.9174757281553398
precision: 0.9742268041237113
F-1 Score: 0.9450000000000001

Training glove.6B.100d.txt …
```


ROC Curve: GLoVE - 100D
TPR: True positive rate
FPR: False positive rate
0.0
0.2
0.4
0.6
0.8
1.0

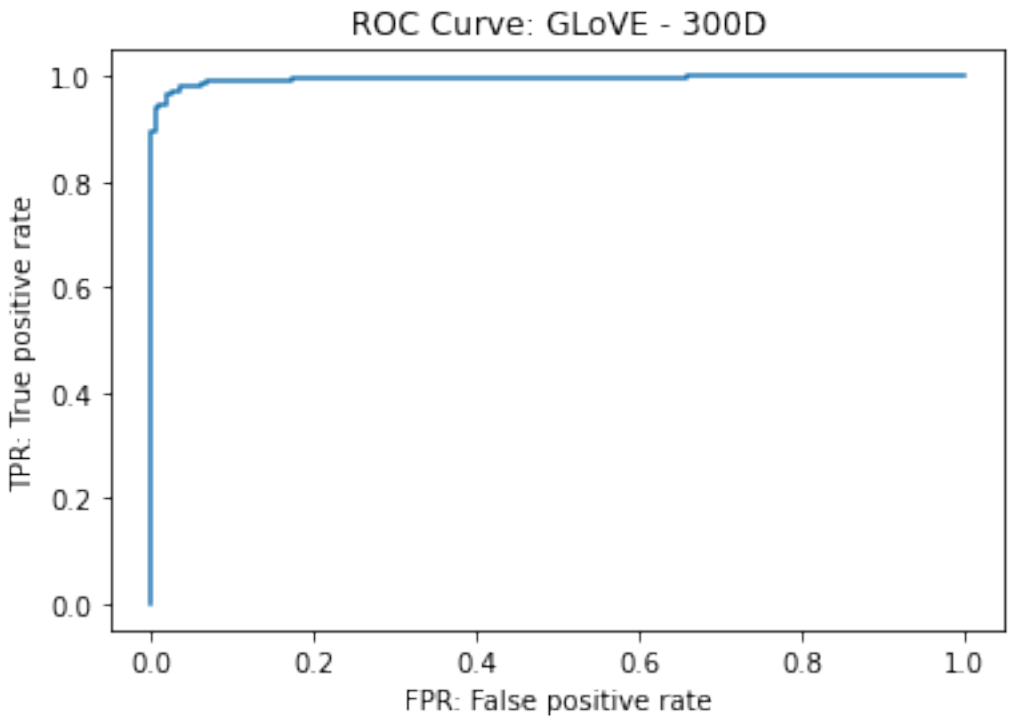

Confusion Matrix for GLoVE - 100D
206
3
15
191
True Category
Predicted Category
0
1
200
175
150
125
100
75
50
25

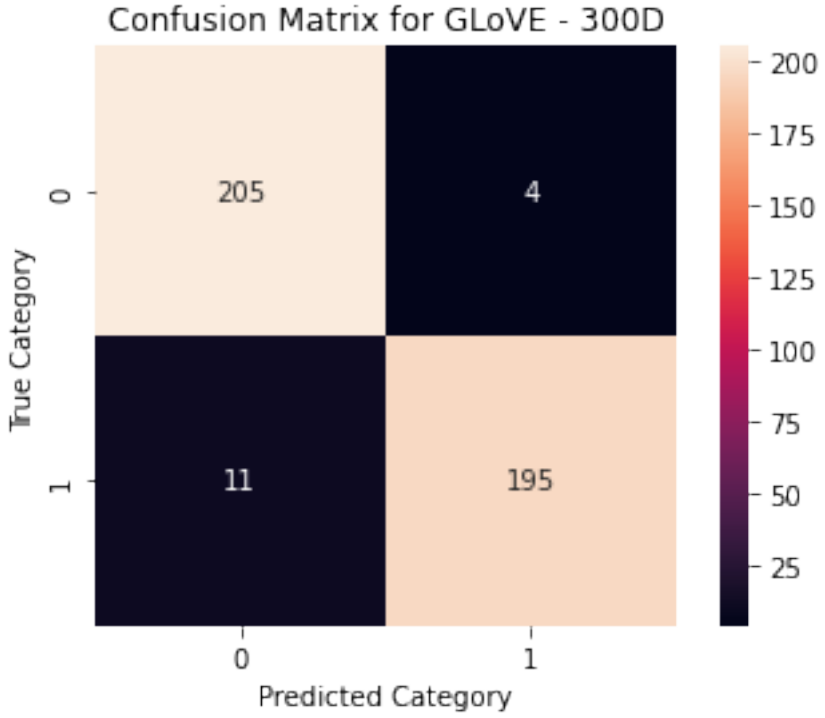
```
accuracy of test data: 0.9566265060240964
recall: 0.9271844660194175
precision: 0.9845360824742269
F-1 Score: 0.9550000000000001

Training glove.6B.200d.txt …
```





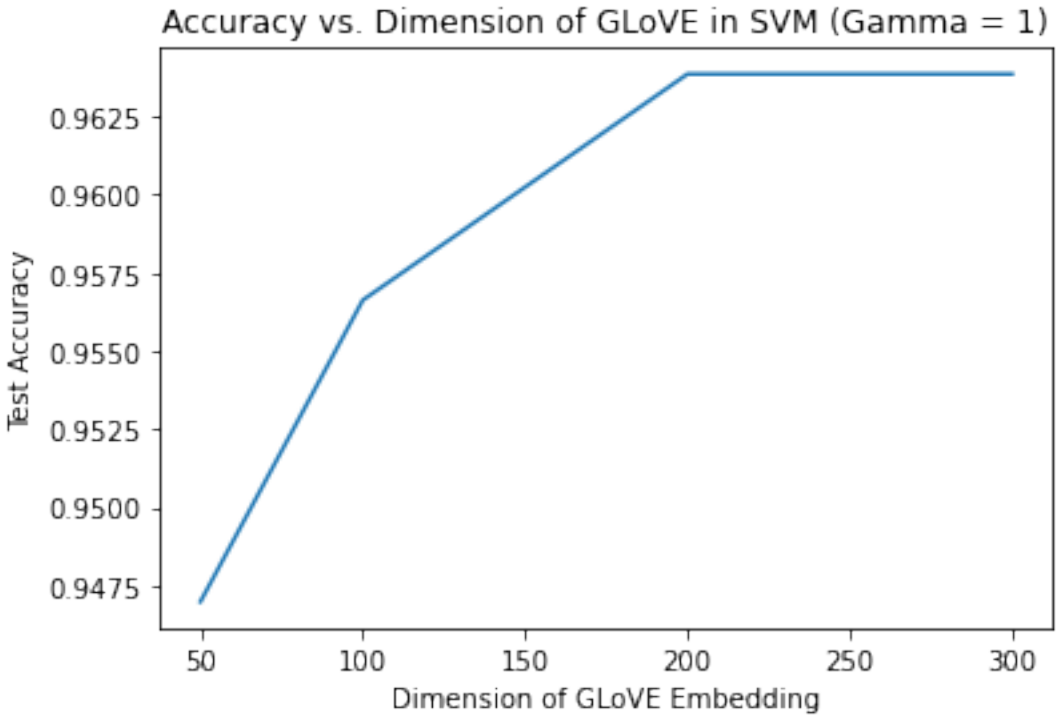

```
accuracy of test data: 0.963855421686747
recall: 0.9466019417475728
precision: 0.9798994974874372
F-1 Score: 0.9629629629629629

Training glove.6B.300d.txt …
```


ROC Curve: GLoVE - 300D
TPR: True positive rate
FPR: False positive rate
0.0
0.2
0.4
0.6
0.8
1.0


Confusion Matrix for GLoVE - 300D
True Category
Predicted Category
205
4
11
195
0
1
200
175
150
125
100
75
50
25

```
accuracy of test data: 0.963855421686747
recall: 0.9466019417475728
precision: 0.9798994974874372
F-1 Score: 0.9629629629629629

Accuracy: [0.946987951807229, 0.9566265060240964, 0.963855421686747,
0.963855421686747]
```

[59]:
```
plt.plot(dimensions, accuracy)
plt.title("Accuracy vs. Dimension of GLoVE in SVM (Gamma = 1)")
plt.xlabel("Dimension of GLoVE Embedding")
plt.ylabel("Test Accuracy")
plt.show()
```



We repeated use SVM classifier on dimension (50, 100, 200, 300) with fixed Gamma=1 because the best parameter we found in Question11 was Gamma=1 and it is better to demonstrate the trend with fixed parameter (control variable).

We can expect the trend looks like this plot. It is because with more dimensions of GLoVE embedding, we have more information about the words and co-occurrence probability ratios between target words and context words. Hence, the accuracy increases. The increases from 50->100 and 100->200 are obvious because the dimensions are doubled.

14 Question 13

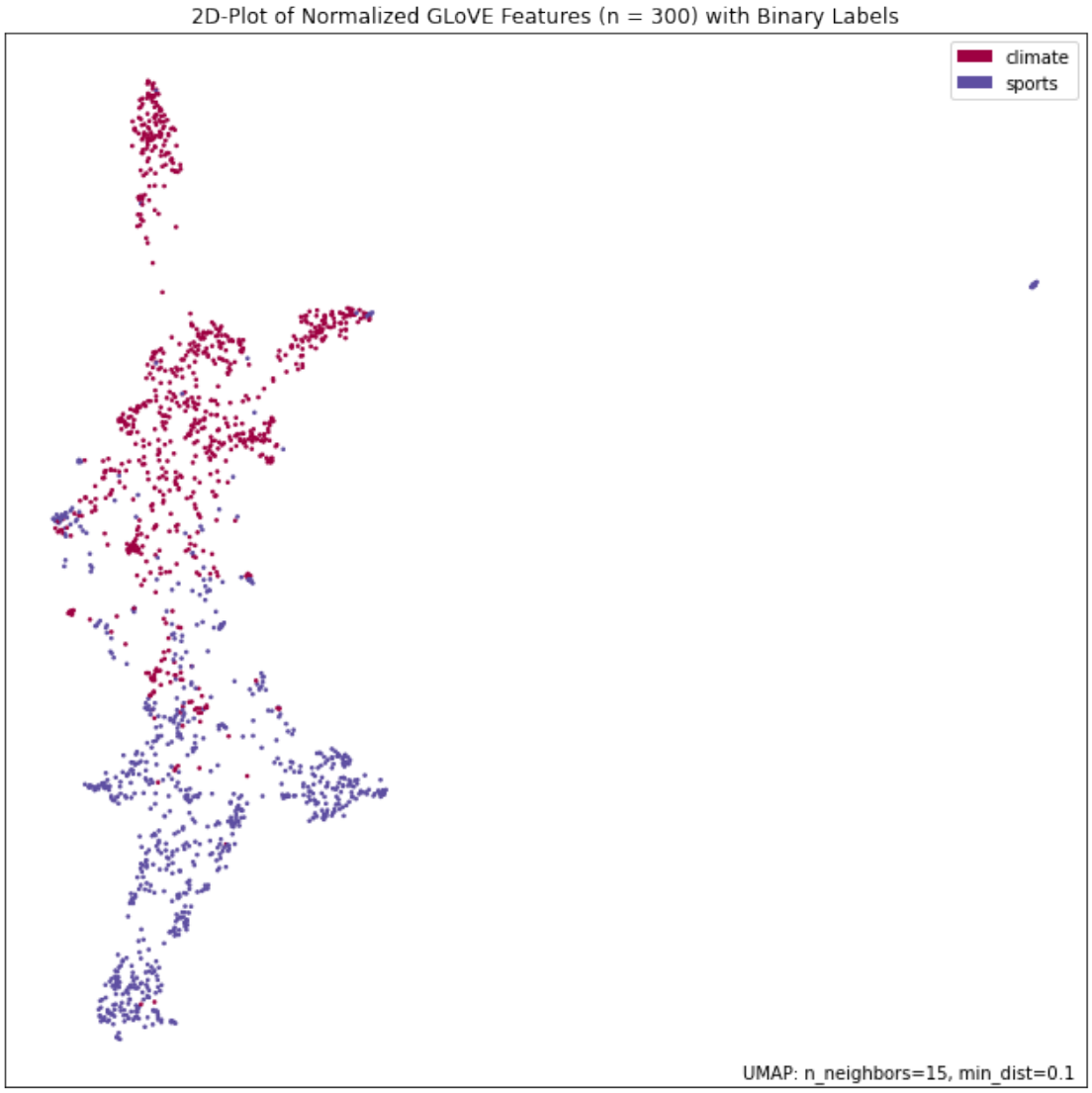
[60]:
```
import umap
import umap.plot

reducer_glove = umap.UMAP(n_components=2, random_state=42)
reducer_glove = reducer_glove.fit(glove_train_matrix)
```

[61]:
```
print("The shape of the reducer_glove is ({}, {})".format(reducer_glove.
 ↪embedding_.shape[0], reducer_glove.embedding_.shape[1]))
```

```
The shape of the reducer_glove is (1657, 2)
```

[62]:
```
#0:sports; 1:climate
glove_train_rootlabel_text = []
for label in glove_train_rootlabel_numbers:
    if label == 0:
        glove_train_rootlabel_text.append("sports")
    else:
        glove_train_rootlabel_text.append("climate")
```

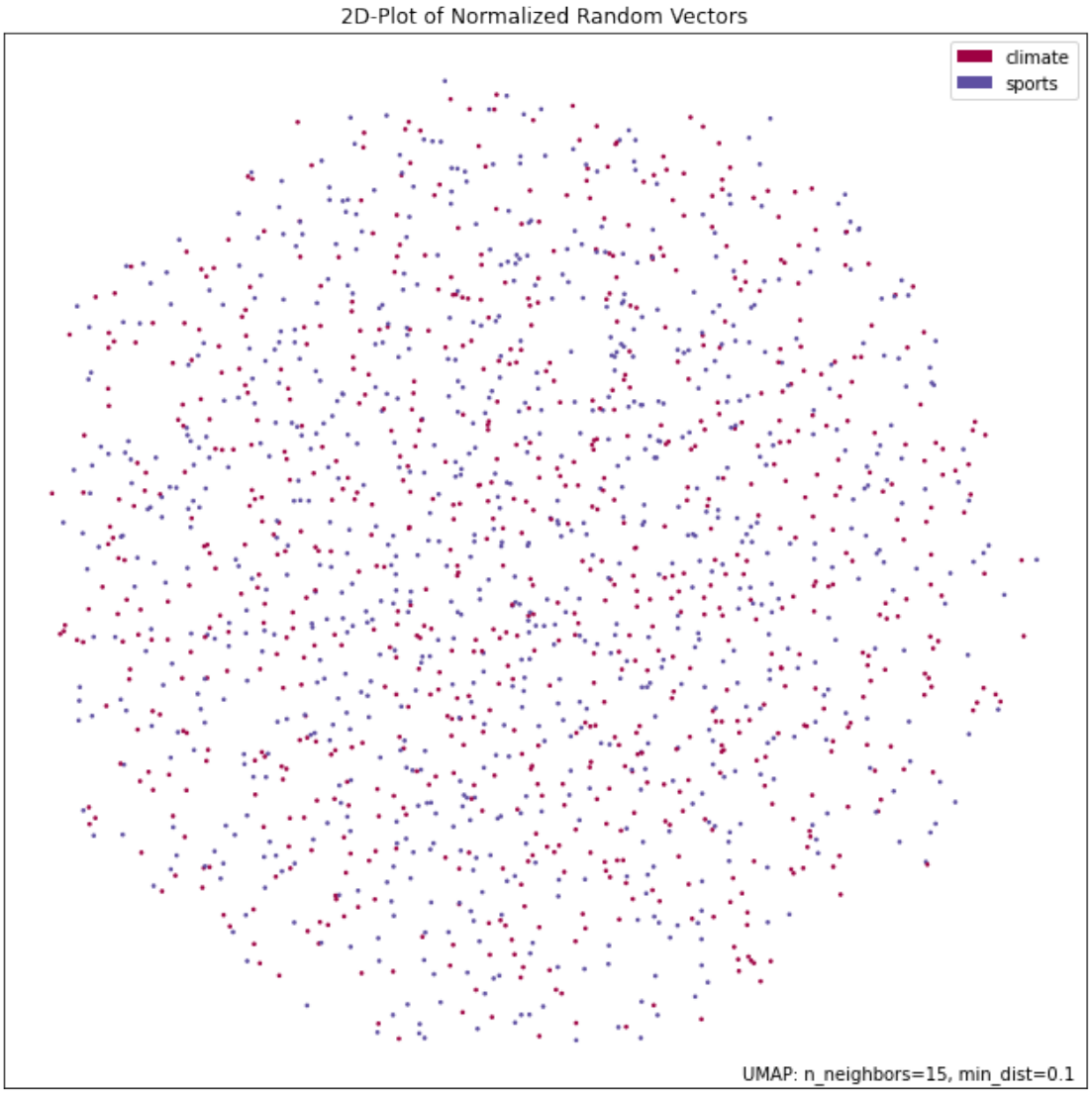
[63]:
```
plot_glove = umap.plot.points(reducer_glove, labels=np.
 ↪array(glove_train_rootlabel_text))
plt.title('2D-Plot of Normalized GLoVE Features (n = 300) with Binary Labels')
plt.show()
```



[64]:
```
np.random.seed(42)
random.seed(42)

rv = np.random.rand(reducer_glove.embedding_.shape[0], 300) #uniform␣
 ↪distribution over [0, 1)
for row in range(rv.shape[0]):
    rv[row] = normalize(rv[row][:,np.newaxis], axis=0).ravel()

reducer_rv = umap.UMAP(n_components=2, random_state=42)
reducer_rv = reducer_rv.fit(rv)
```

[65]:
```
np.random.seed(42)
random.seed(42)

#0:sports; 1:climate
rv_labels = np.random.randint(2, size=reducer_glove.embedding_.shape[0])␣
 ↪#Uniform Random

rv_text = []
for label in rv_labels:
    if label == 0:
        rv_text.append("sports")
    else:
        rv_text.append("climate")
```

[66]:
```
plot_rv = umap.plot.points(reducer_rv, labels=np.array(rv_text))
plt.title('2D-Plot of Normalized Random Vectors')
plt.show()
```



In both cases, we use the dimension=300, and we use uniform distribution over [0, 1) in random vectors, normalize them, and randomly assign them two different labels (sports and climate).

We can only see clusters from the plot of GLoVE features (actual dataset), and it is expected because the GLoVE embeddings truly provide much information to distinguish features. On the other hand, we can't see any clusters from the plot of random vectors because it is uniformly distributed.