KNN on Amazon Fine Food Reviews

The Amazon Fine Food Reviews dataset consists of reviews of fine foods from Amazon.

Number of reviews: 568,454 Number of users: 256,059 Number of products: 74,258 Timespan: Oct 1999 - Oct 2012 Number of Attributes/Columns in data: 10

Attribute Information:

```
Id - Row Id
ProductId - unique identifier for the product
UserId - unqiue identifier for the user
ProfileName - Profile name of the user
HelpfulnessNumerator - number of users who found the review helpful
HelpfulnessDenominator - number of users who indicated whether they found the revie
w helpful or not
Score - rating between 1 and 5
Time - timestamp for the review
Summary - brief summary of the review
Text - text of the review
```

About KNN

KNN means K-Neighbour neighbours which can store all available cases and classifies new measures based on the similarity measure(e.g., distance functions)

KNN can use different distance functions like Euclidean distance, Hamming distance, Manhattan distance etc...

For a given query we will find out the class label(+ve or -ve) using the K-nearest neighbours based on this we will give majority vote for the class label.

For KNN, K is an hyperparameter so, as K increases then the smoothness of the decision surface will increase.

Build KNN with featurisation techniques like BOW, TFIDF, AVGW2V, TFIDFW2V

Find the optimal K and predict the accuracy score

In [1]:
```
# Import the relevant libraries
import sqlite3
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
#import scikitplot as skplt
import seaborn as sns

import os
import re
import nltk
import string
from nltk.corpus import stopwords
import string
from nltk.stem import PorterStemmer
from nltk.stem import SnowballStemmer

from sklearn.feature_extraction.text import CountVectorizer
from sklearn import preprocessing
from sklearn.manifold import TSNE
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.cross_validation import train_test_split
from sklearn.preprocessing import MaxAbsScaler
import gensim
import warnings
warnings.filterwarnings('ignore')

#KNN imports
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
from sklearn.cross_validation import cross_val_score
from tqdm import tqdm
from sklearn.decomposition import TruncatedSVD
from sklearn.decomposition import PCA
```

```
/usr/local/lib/python3.5/site-packages/sklearn/cross_validation.py:41: Depreca
tionWarning: This module was deprecated in version 0.18 in favor of the model_
selection module into which all the refactored classes and functions are moved
. Also note that the interface of the new CV iterators are different from that
of this module. This module will be removed in 0.20.
  "This module will be removed in 0.20.", DeprecationWarning)
```

In [2]:
```
# Loading the dataset
con = sqlite3.connect('database.sqlite')

Act_data = pd.read_sql_query(""" SELECT * FROM Reviews WHERE Score != 3 LIMIT 10
0000""", con)

print("Actual Shape of data:",Act_data.shape)
#Structure of 5 rows of data
Act_data.head()
```

```
Actual Shape of data: (100000, 10)
```

Out[2]:

	Id	ProductId	UserId	ProfileName	HelpfulnessNumerator	HelpfulnessDenomi
0	1	B001E4KFG0	A3SGXH7AUHU8GW	delmartian	1	1
1	2	B00813GRG4	A1D87F6ZCVE5NK	dll pa	0	0
2	3	B000LQOCH0	ABXLMWJIXXAIN	Natalia Corres "Natalia Corres"	1	1
3	4	B000UA0QIQ	A395BORC6FGVXV	Karl	3	3
4	5	B006K2ZZ7K	A1UQRSCLF8GW1T	Michael D. Bigham "M. Wassir"	0	0

Data Cleaning

In [3]:
```
Act_data.describe()
```

Out[3]:

	Id	HelpfulnessNumerator	HelpfulnessDenominator	Score	T
count	100000.000000	100000.000000	100000.000000	100000.000000	1.000000e
mean	54380.301200	1.645130	2.106580	4.256390	1.296128e
std	31372.425622	6.212261	6.832594	1.329395	4.781070e
min	1.000000	0.000000	0.000000	1.000000	9.486720e
25%	27298.750000	0.000000	0.000000	4.000000	1.270512e
50%	54294.500000	0.000000	1.000000	5.000000	1.311379e
75%	81583.250000	2.000000	2.000000	5.000000	1.332547e
max	108623.000000	559.000000	562.000000	5.000000	1.351210e

In [4]:
```
# Removing the data for which score is not equal to 3
Act_data = Act_data[Act_data['Score'] != 3]
Act_data.shape
```

Out[4]: (100000, 10)

In [5]:
```
#Sampled amazon fine foood reviews filtered data to 30k datapoints for time effi
ecieny
sample_act_data = Act_data.head(50000)
sample_act_data['Score'] = ['Positive' if (int(score)>3) else 'Negative' for sco
re in sample_act_data['Score']]
```

In [6]:
```
sample_act_data.head()
```

Out[6]:

	Id	ProductId	UserId	ProfileName	HelpfulnessNumerator	HelpfulnessDenomi
0	1	B001E4KFG0	A3SGXH7AUHU8GW	delmartian	1	1
1	2	B00813GRG4	A1D87F6ZCVE5NK	dll pa	0	0
2	3	B000LQOCH0	ABXLMWJIXXAIN	Natalia Corres "Natalia Corres"	1	1
3	4	B000UA0QIQ	A395BORC6FGVXV	Karl	3	3
4	5	B006K2ZZ7K	A1UQRSCLF8GW1T	Michael D. Bigham "M. Wassir"	0	0

In [7]:
```
#Sorting the data and deleting the duplicate records
sorted_data = sample_act_data.sort_values('ProductId', ascending=True, inplace=F
alse, kind='quicksort', na_position='last')

final_data = sample_act_data.drop_duplicates(subset = {'UserId','ProfileName','T
ime','Text'}, keep='first', inplace=False)
```

In [8]:
```
final_data['Score'] = [1 if score == 'Positive' else 0 for score in final_data['
Score']]
final_data.sort_values('Time', inplace = True)
```

In [9]:
```
text = final_data['Text']
score = final_data['Score']
```

Data Preprocessing

Methods used in Preprocessing

```
1.Stop words removal
2.Removal of HTML tags
3.Stemming using porter stemming - removing affixes from words
4.Tokenizing - Splitting sentences and words from the body of text
5.Lemmatization - synonym or a different word with the same meaning
```

In [10]:
```
nltk.download('stopwords')
#set of stop words
stop = set(stopwords.words('english'))

#Initializing snowball stemmer
snow_stem = nltk.stem.SnowballStemmer('english')

#function to clean the word of any html-tags
def cleanhtml(sentence):
    cleanr = re.compile('<.*?>')
    cleantext = re.sub(cleanr, ' ', sentence)
    return cleantext

#function to clean the word of any punctuation or special characters
def cleanpunc(sentence):
    cleaned = re.sub(r'[?|!|\'|"|#]',r'',sentence)
    cleaned = re.sub(r'[.|,|)|(|\|/]',r' ',cleaned)
    return  cleaned
```

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

In [11]:
```
#Preprocessing the text like cleaning HTML tags, removal of stopwords, punctuati
ons removal...
Preproc_text = []
cleanedtext = []
for sent in final_data['Text']:
    row_text=[]
    sent = cleanhtml(sent)
    for words in sent.split():
        clean_word = cleanpunc(words)
        if (clean_word.isalpha()) & (len(clean_word)>2):
            if(clean_word.lower() is not stop):
                finalword = (snow_stem.stem(clean_word.lower()).encode('utf8'))
                row_text.append(finalword)
            else:
                continue
        else:
            continue

    Preproc_text.append( b' '.join(row_text).decode('utf8'))
```

In [12]:
```
#Splitting the data into train and test data
text_train, text_test, score_train, score_test = train_test_split(Preproc_text,s
core, test_size=0.3,
                                                                      stratify=None,
random_state=0)
```

In [13]:
```
Preproc_text_kdtree = Preproc_text[0:20000]
score_kdtree = score[0:20000]
text_train_kd, text_test_kd, score_train_kd, score_test_kd = train_test_split(Pr
eproc_text_kdtree,score_kdtree, test_size=0.3,
                                                                    stratify=None,
random_state=0)
```

BOW

In [14]:
```
#Converting the cleaned text to sparse matrix
count_vec = CountVectorizer()
bow_train = count_vec.fit_transform(text_train)
```

In [15]:
```
bow_test = count_vec.transform(text_test)
```

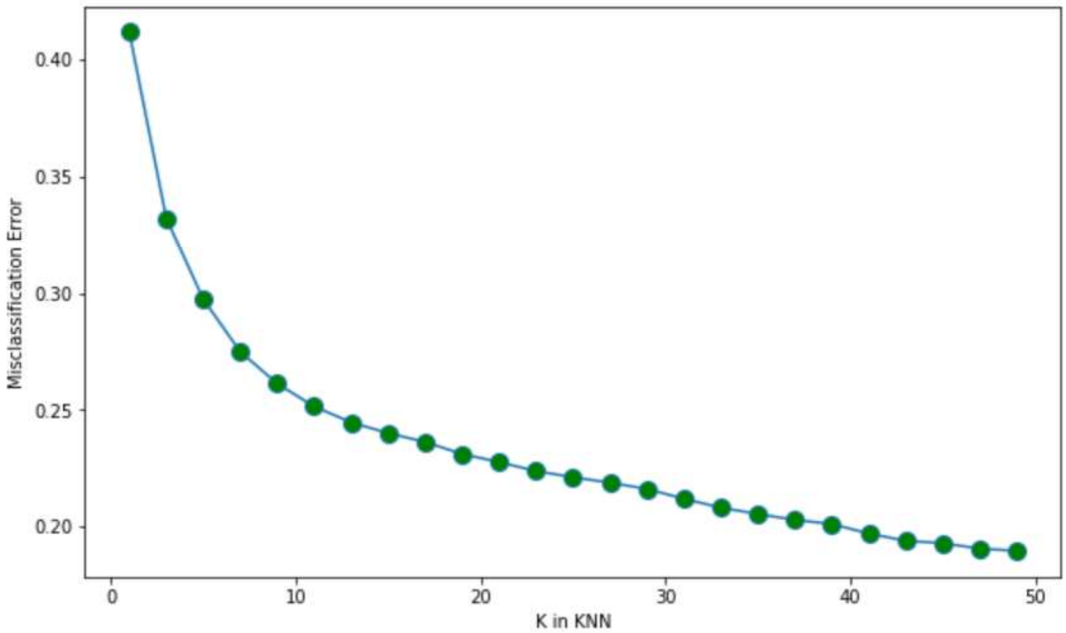
In [16]:
```
#standardization of data
std_data_train = MaxAbsScaler().fit_transform(bow_train)
std_data_test = MaxAbsScaler().fit_transform(bow_test)
print(std_data_train.shape)
print(std_data_test.shape)
```

```
(32250, 18799)
(13822, 18799)
```

BOW(Brute force)

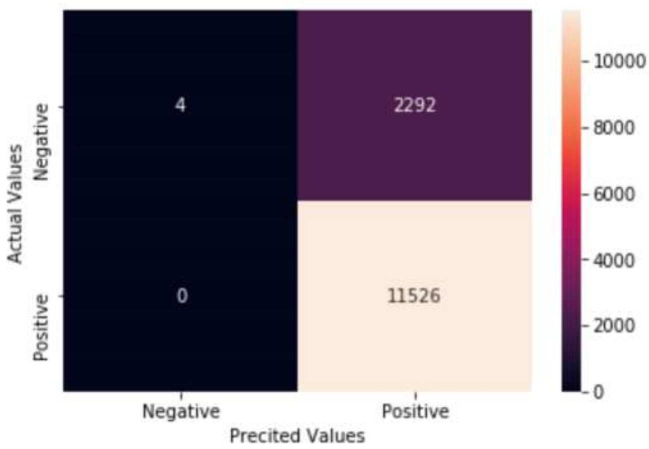
In [17]:
```
# creating odd list of K for KNN
myList = list(range(0,50))
neighbors = list(filter(lambda x: x % 2 != 0, myList))
# empty list that will hold cv scores
cv_scores = []
# perform 10-fold cross validation
for k in tqdm(neighbors):
    knn = KNeighborsClassifier(n_neighbors=k,algorithm='brute', n_jobs=-1)
    scores = cross_val_score(knn, std_data_train, score_train, cv=10, scoring='r
oc_auc')
    cv_scores.append(scores.mean())
# changing to misclassification error
MSE = [1 - x for x in cv_scores]

# determining best k
optimal_k = neighbors[MSE.index(min(MSE))]
plt.figure(figsize=(10,6))
plt.xlabel('K in KNN')
plt.ylabel('Misclassification Error')
plt.plot(neighbors, MSE, marker='o',markerfacecolor='green', markersize=10)
print('\nThe optimal number of neighbors is %d.' % optimal_k)
```

```
100%|██████████| 25/25 [30:58<00:00, 74.76s/it]

The optimal number of neighbors is 49.
```



In [18]:
```
#learning the model k = optimal_k
knn_optimal = KNeighborsClassifier(n_neighbors=optimal_k)

# fitting the model
knn_optimal.fit(std_data_train, score_train)

# predict on the test response
pred = knn_optimal.predict(std_data_test)

# evaluate accuracy on test data
acc = accuracy_score(score_test, pred) * 100

print('\nThe accuracy of the knn classifier for k = %d is %f%%' % (optimal_k, ac
c))

cm = confusion_matrix(score_test, pred)
class_label = ['Negative', 'Positive']
conf_matrix = pd.DataFrame(cm, index=class_label, columns=class_label)
sns.heatmap(conf_matrix, annot=True, fmt='d')
plt.ylabel('Actual Values')
plt.xlabel('Precited Values')
plt.show()
```

```
The accuracy of the knn classifier for k = 49 is 83.417740%
```



BOW(Kd Tree)

In [19]:
```
count_vec = CountVectorizer()
bow_train_kd = count_vec.fit_transform(text_train_kd)
bow_test_kd = count_vec.transform(text_test_kd)
```

In [20]:
```
std_data_train_kd = MaxAbsScaler().fit_transform(bow_train_kd)
std_data_test_kd = MaxAbsScaler().fit_transform(bow_test_kd)
print(std_data_train_kd.shape)
print(std_data_test_kd.shape)
```

```
(14000, 13139)
(6000, 13139)
```

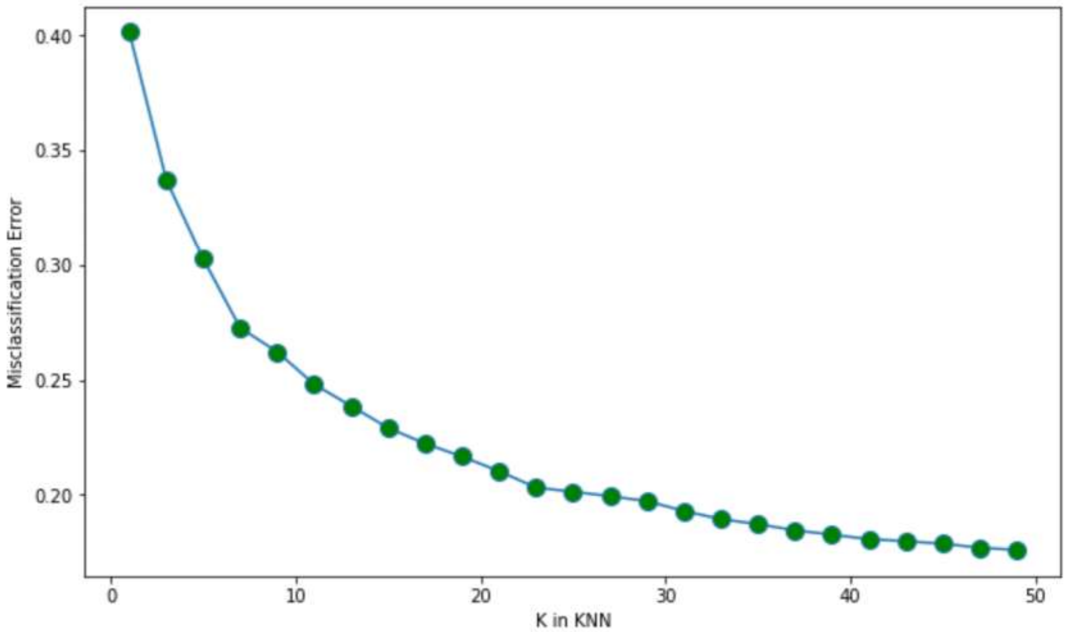
In [21]:
```
# Converting sparse matrix to dense matrix
svd = TruncatedSVD(n_components=400)
std_train_kd = svd.fit_transform(std_data_train_kd)
std_test_kd = svd.transform(std_data_test_kd)
```

In [22]:
```
print(std_train_kd.shape)
print(std_test_kd.shape)
```

```
(14000, 400)
(6000, 400)
```

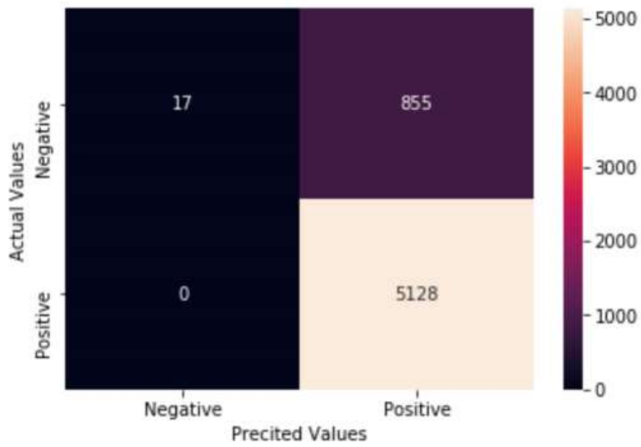
In [23]:
```
myList = list(range(0,50))
neighbors = list(filter(lambda x: x % 2 != 0, myList))
# empty list that will hold cv scores
cv_scores = []
# perform 10-fold cross validation
for k in tqdm(neighbors):
    knn = KNeighborsClassifier(n_neighbors=k,algorithm='kd_tree', n_jobs=-1)
    scores = cross_val_score(knn, std_train_kd, score_train_kd, cv=5, scoring='r
oc_auc')
    cv_scores.append(scores.mean())
# changing to misclassification error
MSE = [1 - x for x in cv_scores]

# determining best k
optimal_k = neighbors[MSE.index(min(MSE))]
plt.figure(figsize=(10,6))
plt.xlabel('K in KNN')
plt.ylabel('Misclassification Error')
plt.plot(neighbors, MSE, marker='o',markerfacecolor='green', markersize=10)
print('\nThe optimal number of neighbors is %d.' % optimal_k)
```

```
100%|██████████| 25/25 [18:04<00:00, 43.19s/it]

The optimal number of neighbors is 49.
```



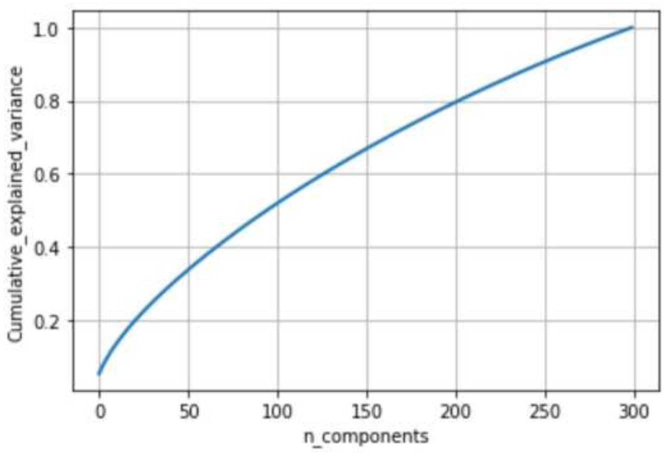
In [24]:
```
#learning the model k = optimal_k
knn_optimal = KNeighborsClassifier(n_neighbors=optimal_k)

# fitting the model
knn_optimal.fit(std_train_kd, score_train_kd)

# predict on the test response
pred = knn_optimal.predict(std_test_kd)

# evaluate accuracy on test data
acc = accuracy_score(score_test_kd, pred) * 100

print('\nThe accuracy of the knn classifier for k = %d is %f%%' % (optimal_k, ac
c))

cm = confusion_matrix(score_test_kd, pred)
class_label = ['Negative', 'Positive']
conf_matrix = pd.DataFrame(cm, index=class_label, columns=class_label)
sns.heatmap(conf_matrix, annot=True, fmt='d')
plt.ylabel('Actual Values')
plt.xlabel('Precited Values')
plt.show()
```

```
The accuracy of the knn classifier for k = 49 is 85.750000%
```



In [26]:
```
from sklearn import decomposition
pca = decomposition.PCA()
# PCA for dimensionality redcution (non-visualization)

pca.n_components = 300
pca_data = pca.fit_transform(std_train_kd)

# Calculating the cummulative sum of variances
percentage_var_explained = pca.explained_variance_ / np.sum(pca.explained_varian
ce_);
cum_var = np.cumsum(percentage_var_explained)
print(cum_var)

# Plot the PCA spectrum
plt.plot(cum_var, linewidth=2)
plt.axis('tight')
plt.grid()
plt.xlabel('n_components')
plt.ylabel('Cumulative_explained_variance')
plt.show()
```

```
[0.05441594 0.06507291 0.07518583 0.08441794 0.09324283 0.10172182
 0.10979264 0.11763786 0.12469015 0.13170914 0.13853149 0.14496861
 0.1513233  0.1576704  0.1639585  0.17006192 0.17597223 0.18173983
 0.18746536 0.19306584 0.19856422 0.20396899 0.20932946 0.21457541
 0.21979407 0.22489887 0.22995916 0.2349943  0.23998368 0.24491287
 0.2497348  0.25450393 0.25923458 0.26392355 0.26857048 0.27317335
 0.27774806 0.28226506 0.28674704 0.29121171 0.2956173  0.29999412
 0.30435442 0.30865666 0.31292847 0.31717706 0.32139928 0.32560881
 0.32980513 0.33395602 0.33808582 0.3421962  0.34626446 0.35031513
 0.35435883 0.35837927 0.36238667 0.36635898 0.37030777 0.37423988
 0.37813264 0.38201759 0.3858726  0.38970016 0.39351577 0.39731451
 0.4010982  0.40484854 0.40858749 0.41230325 0.41601769 0.41970792
 0.42338132 0.42704365 0.43068102 0.43431111 0.43791043 0.44150309
 0.44507779 0.44862754 0.45216215 0.45568474 0.45920405 0.46269053
 0.46615496 0.46961112 0.47304749 0.47648033 0.47989152 0.48329938
 0.48668671 0.49006406 0.49343487 0.49677265 0.50008671 0.50339137
 0.5066893  0.50998006 0.51325954 0.51653458 0.51979557 0.52304656
 0.52628187 0.52950191 0.53271116 0.53590173 0.53908658 0.54225817
 0.54540686 0.54854611 0.55166457 0.55477591 0.55787532 0.56096635
 0.56404817 0.56711773 0.5701779  0.57322679 0.57627085 0.57930273
 0.58232517 0.58533999 0.58834509 0.59134766 0.59433736 0.5973161
 0.60028591 0.60324386 0.6061967  0.60913821 0.6120603  0.61497292
 0.61788302 0.62077876 0.62367219 0.62655969 0.62942809 0.63229082
 0.63512892 0.63796329 0.6407946  0.64361692 0.6464321  0.64924074
 0.6520423  0.65483366 0.65761895 0.66039254 0.66316241 0.66591497
 0.66866077 0.67139724 0.67413293 0.67686064 0.67957447 0.68227452
 0.68496392 0.68765071 0.69032882 0.69299715 0.69565613 0.69830775
 0.70095781 0.70359865 0.70623494 0.70885624 0.71147425 0.71408422
 0.71668152 0.71926193 0.72183714 0.72440277 0.7269646  0.72952006
 0.73207447 0.73461915 0.73715562 0.73969071 0.7422155  0.74473367
 0.74724831 0.74974853 0.7522427  0.75472865 0.75719999 0.75966421
 0.76212627 0.76457824 0.76702794 0.76946917 0.77190916 0.77434102
 0.77676267 0.77918007 0.78159445 0.78399615 0.78639037 0.78877573
 0.79115714 0.79353195 0.79590482 0.79826307 0.80061707 0.80296833
 0.80530529 0.8076368  0.8099632  0.81228257 0.81459472 0.81690151
 0.81919616 0.82148606 0.82377347 0.82605965 0.82833848 0.83061378
 0.83287948 0.83513719 0.83738818 0.83963067 0.84187189 0.84410544
 0.84633316 0.8485519  0.85077037 0.85297918 0.85518035 0.85737506
 0.85956842 0.86175519 0.86393588 0.86611278 0.86828359 0.87044451
 0.87260238 0.87475607 0.87689792 0.87903173 0.88116096 0.88328379
 0.88539525 0.88750288 0.88960722 0.89170753 0.89380704 0.89589365
 0.8979778  0.90006067 0.90213504 0.90420513 0.90626848 0.90832426
 0.91037122 0.91241757 0.91445662 0.91649103 0.91851702 0.92053874
 0.92255781 0.92457499 0.92658716 0.92859083 0.93058437 0.93257128
 0.93455526 0.93653219 0.93850438 0.94046675 0.94242135 0.94437294
 0.94631612 0.94825067 0.95017944 0.95210278 0.95402384 0.95593928
 0.95784996 0.95975741 0.96165682 0.96354962 0.96542962 0.96730308
 0.96917236 0.97103565 0.97289061 0.97474442 0.97659464 0.97843544
 0.98026876 0.98209828 0.98392062 0.98574148 0.98755085 0.98935332
 0.99114701 0.99292884 0.99470806 0.9964802  0.9982416  1.        ]
```



TFIDF

In [27]:
```
tfidf_vect = TfidfVectorizer(ngram_range=(1,2))
```

In [28]:
```
tfidf_train = tfidf_vect.fit_transform(text_train)
tfidf_test = tfidf_vect.transform(text_test)
```

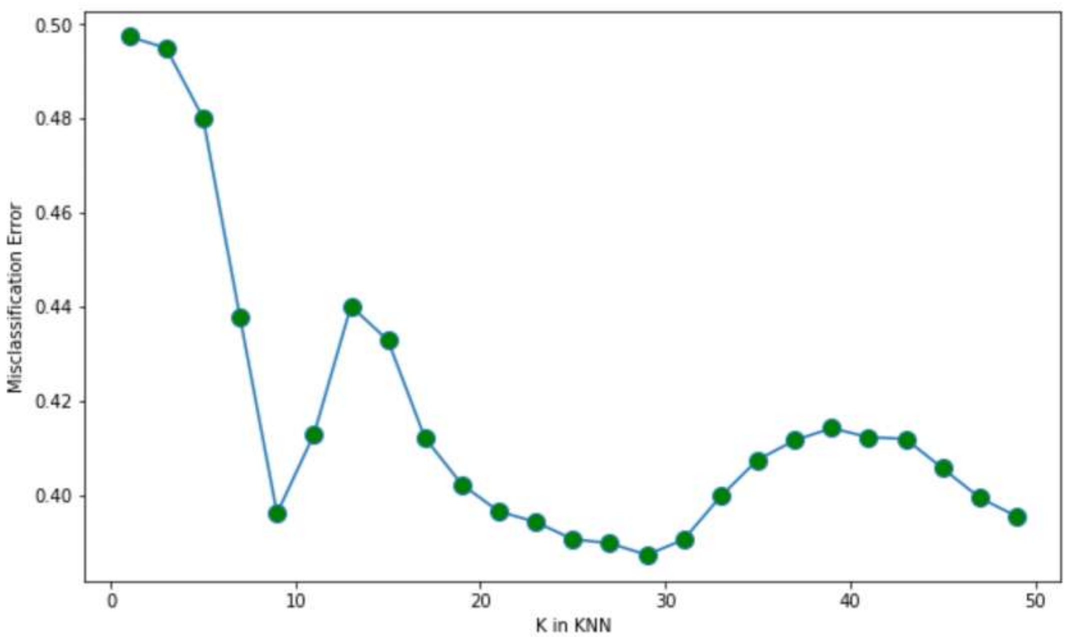
In [29]:
```
#standardization of data
std_data_train = MaxAbsScaler().fit_transform(tfidf_train)
std_data_test = MaxAbsScaler().fit_transform(tfidf_test)
print(std_data_train.shape)
print(std_data_test.shape)
```

```
(32250, 435712)
(13822, 435712)
```

TFIDF(Brute Force)

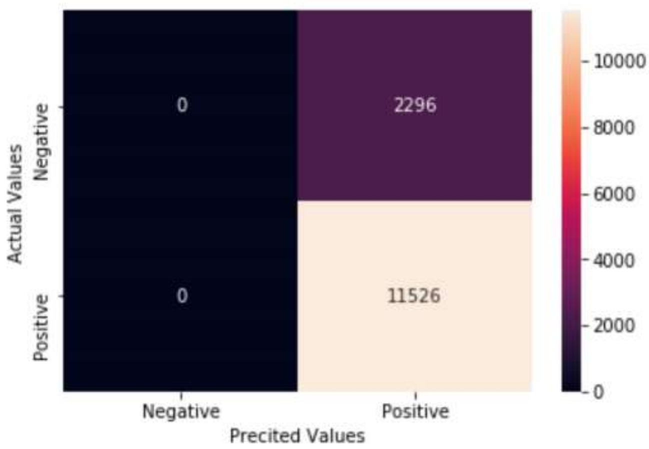
In [30]:
```
myList = list(range(0,50))
neighbors = list(filter(lambda x: x % 2 != 0, myList))
# empty list that will hold cv scores
cv_scores = []
# perform 10-fold cross validation
for k in tqdm(neighbors):
    knn = KNeighborsClassifier(n_neighbors=k,algorithm='brute', n_jobs=-1)
    scores = cross_val_score(knn, std_data_train, score_train, cv=10, scoring='r
oc_auc')
    cv_scores.append(scores.mean())
# changing to misclassification error
MSE = [1 - x for x in cv_scores]

# determining best k
optimal_k = neighbors[MSE.index(min(MSE))]
plt.figure(figsize=(10,6))
plt.xlabel('K in KNN')
plt.ylabel('Misclassification Error')
plt.plot(neighbors, MSE, marker='o',markerfacecolor='green', markersize=10)
print('\nThe optimal number of neighbors is %d.' % optimal_k)
```

```
100%|██████████| 25/25 [31:45<00:00, 76.21s/it]

The optimal number of neighbors is 29.
```



In [31]:
```
#learning the model k = optimal_k
knn_optimal = KNeighborsClassifier(n_neighbors=optimal_k)

# fitting the model
knn_optimal.fit(std_data_train, score_train)

# predict on the test response
pred = knn_optimal.predict(std_data_test)

# evaluate accuracy on test data
acc = accuracy_score(score_test, pred) * 100

print('\nThe accuracy of the knn classifier for k = %d is %f%%' % (optimal_k, ac
c))

cm = confusion_matrix(score_test, pred)
class_label = ['Negative', 'Positive']
conf_matrix = pd.DataFrame(cm, index=class_label, columns=class_label)
sns.heatmap(conf_matrix, annot=True, fmt='d')
plt.ylabel('Actual Values')
plt.xlabel('Precited Values')
plt.show()
```

```
The accuracy of the knn classifier for k = 29 is 83.388800%
```



TFIDF(Kd tree)

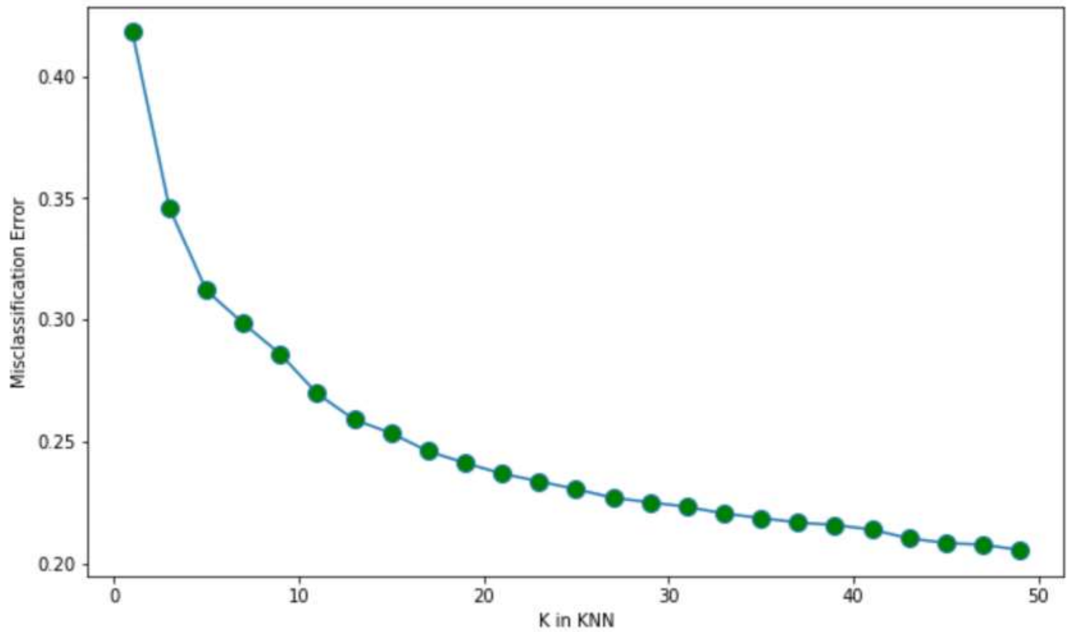
In [32]:
```
tfidf_train_kd = tfidf_vect.fit_transform(text_train_kd)
tfidf_test_kd = tfidf_vect.transform(text_test_kd)
```

In [33]:
```
std_train_kd = MaxAbsScaler().fit_transform(tfidf_train_kd)
std_test_kd = MaxAbsScaler().fit_transform(tfidf_test_kd)
```

In [34]:
```
svd = TruncatedSVD(n_components=400)
std_train_kd = svd.fit_transform(std_train_kd)
std_test_kd = svd.transform(std_test_kd)
```

In [35]:
```
myList = list(range(0,50))
neighbors = list(filter(lambda x: x % 2 != 0, myList))
# empty list that will hold cv scores
cv_scores = []
# perform 10-fold cross validation
for k in tqdm(neighbors):
    knn = KNeighborsClassifier(n_neighbors=k,algorithm='kd_tree', n_jobs=-1)
    scores = cross_val_score(knn, std_train_kd, score_train_kd, cv=5, scoring='r
oc_auc')
    cv_scores.append(scores.mean())
# changing to misclassification error
MSE = [1 - x for x in cv_scores]

# determining best k
optimal_k = neighbors[MSE.index(min(MSE))]
plt.figure(figsize=(10,6))
plt.xlabel('K in KNN')
plt.ylabel('Misclassification Error')
plt.plot(neighbors, MSE, marker='o',markerfacecolor='green', markersize=10)
print('\nThe optimal number of neighbors is %d.' % optimal_k)
```

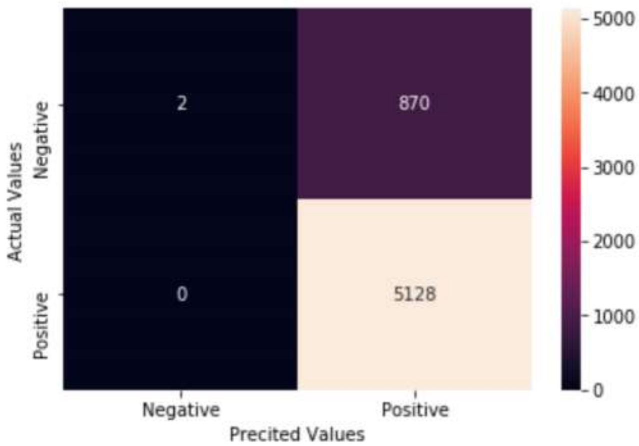
```
100%|██████████| 25/25 [17:42<00:00, 41.70s/it]

The optimal number of neighbors is 49.
```



In [36]:

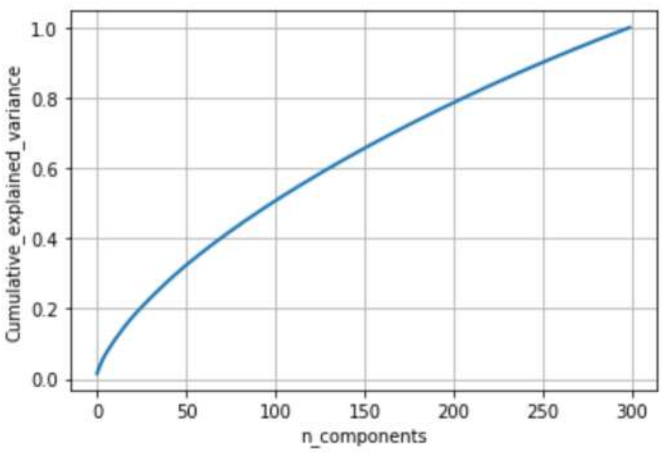
```
#learning the model k = optimal_k
knn_optimal = KNeighborsClassifier(n_neighbors=optimal_k)

# fitting the model
knn_optimal.fit(std_train_kd, score_train_kd)

# predict on the test response
pred = knn_optimal.predict(std_test_kd)

# evaluate accuracy on test data
acc = accuracy_score(score_test_kd, pred) * 100

print('\nThe accuracy of the knn classifier for k = %d is %f%%' % (optimal_k, ac
c))

cm = confusion_matrix(score_test_kd, pred)
class_label = ['Negative', 'Positive']
conf_matrix = pd.DataFrame(cm, index=class_label, columns=class_label)
sns.heatmap(conf_matrix, annot=True, fmt='d')
plt.ylabel('Actual Values')
plt.xlabel('Precited Values')
plt.show()
```

```
The accuracy of the knn classifier for k = 49 is 85.500000%
```



In [37]:
```
from sklearn import decomposition
pca = decomposition.PCA()
# PCA for dimensionality redcution (non-visualization)

pca.n_components = 300
pca_data = pca.fit_transform(std_train_kd)

# Calculating the cummulative sum of variances
percentage_var_explained = pca.explained_variance_ / np.sum(pca.explained_varian
ce_);
cum_var = np.cumsum(percentage_var_explained)
print(cum_var)

# Plot the PCA spectrum
plt.plot(cum_var, linewidth=2)
plt.axis('tight')
plt.grid()
plt.xlabel('n_components')
plt.ylabel('Cumulative_explained_variance')
plt.show()
```

```
[0.01548655 0.03056439 0.04252109 0.05400658 0.06427073 0.07299479
 0.0810848  0.08899008 0.09674921 0.10440736 0.11186325 0.11884
 0.12575818 0.13265203 0.13946356 0.14610043 0.15267345 0.15896809
 0.16521736 0.17130943 0.1771024  0.18280616 0.18844124 0.19402269
 0.19940397 0.20473504 0.20999268 0.21524656 0.2204521  0.22558409
 0.23067073 0.23571205 0.24074291 0.24570124 0.25063497 0.25552839
 0.2604     0.26523971 0.27003277 0.27480283 0.2795398  0.28418974
 0.28874419 0.29328196 0.29780187 0.30226015 0.30665951 0.31101161
 0.31531617 0.31959996 0.32386691 0.32803979 0.3322102  0.33633606
 0.3404378  0.34451297 0.34857022 0.35261796 0.35660332 0.36056558
 0.36450851 0.36842242 0.37230214 0.37617548 0.38003685 0.38388359
 0.38768032 0.39147378 0.39524369 0.39899397 0.40272886 0.40643168
 0.41012852 0.41380557 0.41747211 0.42111755 0.42473508 0.42834227
 0.43194523 0.43554228 0.4390887  0.44262895 0.44616137 0.44965609
 0.45312649 0.45659018 0.46003264 0.46346589 0.46689168 0.47031024
 0.47370835 0.47709495 0.4804784  0.4838218  0.48716041 0.49048985
 0.49379829 0.49709809 0.50037634 0.5036519  0.50691224 0.51015509
 0.51338139 0.51660484 0.51981398 0.52300592 0.52619193 0.52935041
 0.53250206 0.53564657 0.53878054 0.54188789 0.54498812 0.54807165
 0.55115217 0.5542239  0.55727938 0.56032729 0.56336873 0.56640813
 0.56943545 0.57245598 0.5754694  0.57847378 0.58146436 0.58444633
 0.58742542 0.59039686 0.59336156 0.59631789 0.59926258 0.60220285
 0.60513845 0.60806913 0.61098092 0.61388045 0.61676938 0.61965605
 0.62252801 0.62539247 0.62824579 0.63109605 0.63393182 0.6367592
 0.63958284 0.64239602 0.64520384 0.64799043 0.65077137 0.65354652
 0.65631027 0.65906882 0.66182401 0.66456901 0.66730719 0.67003593
 0.67276242 0.67548045 0.67819439 0.68090168 0.68360234 0.68629116
 0.68897235 0.69164909 0.69432334 0.69699094 0.69964963 0.70229675
 0.70493682 0.70756892 0.710192   0.71280765 0.71542088 0.71802692
 0.72062862 0.72322234 0.72581562 0.72840116 0.73097823 0.73354907
 0.73611651 0.7386758  0.74122853 0.74376985 0.74631042 0.74884241
 0.75136786 0.75388162 0.7563908  0.75889067 0.76138742 0.76387676
 0.76636039 0.768836   0.77130522 0.77377312 0.77623561 0.77869106
 0.78114108 0.78358363 0.78601878 0.78844704 0.79087073 0.79328974
 0.79570469 0.79811379 0.80051749 0.80291211 0.80530041 0.80768289
 0.81005867 0.81242971 0.81479809 0.81715918 0.81950998 0.82185744
 0.82419867 0.8265357  0.82886771 0.83119279 0.83351317 0.8358272
 0.83813475 0.84044063 0.84274027 0.84503274 0.84731802 0.8496021
 0.85187747 0.85414685 0.85641062 0.85867174 0.86093096 0.86318619
 0.86542877 0.86766895 0.86990165 0.87212658 0.87434261 0.87655221
 0.87875867 0.88096151 0.88315719 0.88534399 0.88752695 0.88970507
 0.89187979 0.89405176 0.89622018 0.89838346 0.90054298 0.90269301
 0.90483916 0.90698042 0.9091179  0.91125284 0.91337286 0.91548814
 0.91760167 0.91971096 0.92181773 0.92391981 0.92601974 0.928117
 0.93020661 0.93228961 0.9343668  0.93644027 0.93850952 0.94057606
 0.94263459 0.94468679 0.94673672 0.94878083 0.95081994 0.95285811
 0.95488652 0.95691312 0.95893298 0.96094698 0.96295489 0.96495396
 0.96694768 0.96894034 0.9709296  0.97290969 0.97488557 0.97685107
 0.97881397 0.98077075 0.98272058 0.98466263 0.98660205 0.98853889
 0.99046403 0.99237816 0.99429192 0.99620039 0.99810258 1.        ]
```



W2V and AvgW2V

In [38]:
```
#Splitting each sentence in to words
sent_words = []
for sent in Preproc_text:
    sent_words.append(sent.split())
```

In [39]:
```
#Converting each word into vector
from gensim.models import Word2Vec
w2v = Word2Vec(sent_words,min_count=5,size=50,workers=4)
```

In [40]:
```
#fining w2v_words
w2v_words = list(w2v.wv.vocab)
```

In [41]:
```
#Avg_W2V for all Reviews
avg_w2vs = []
for sent in tqdm(Preproc_text):
    #initializing number of words
    n_words = 0
    #initializing vector of size of 50
    sent_vec = np.zeros(50)
    for word in sent.split():
        if word in w2v_words:
            #creating for each word is an vector
            vec = w2v.wv[word]
            sent_vec += vec
            n_words += 1
    if n_words != 0:
        sent_vec /= n_words
        avg_w2vs.append(sent_vec)
```

```
100%|██████████| 46072/46072 [04:26<00:00, 172.96it/s]
```

In [42]:
```
text_train_avg, text_test_avg, score_train_avg, score_test_avg = train_test_spli
t(avg_w2vs, score, test_size=0.3,
                                                                  stratify=None,
random_state=0)
avg_w2vs_kd = avg_w2vs[0:20000]
score_kd = score[0:20000]
text_train_avg_kd, text_test_avg_kd, score_train_avg_kd, score_test_avg_kd = tra
in_test_split(avg_w2vs_kd, score_kd, test_size=0.3,
                                                                  stratify=None,
random_state=0)
```

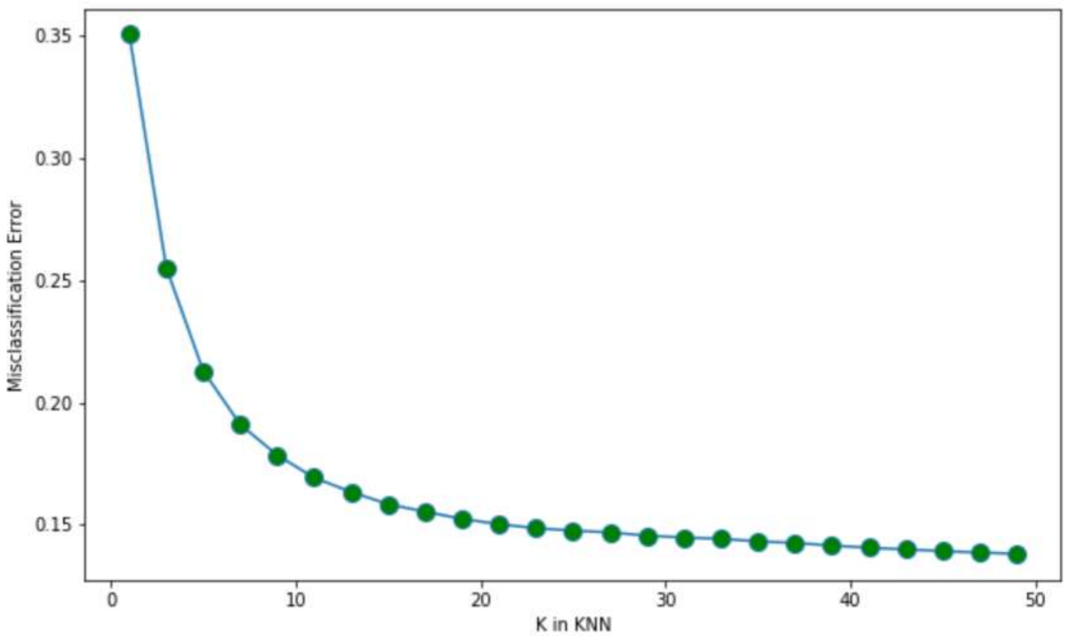
In [43]:
```
#standardization of data for Burte force
std_train_avg = MaxAbsScaler().fit_transform(text_train_avg)
std_test_avg = MaxAbsScaler().fit_transform(text_test_avg)
print(std_train_avg.shape)
print(std_test_avg.shape)
print(score_train_avg.shape)
print(score_test_avg.shape)
#standardization of data for Kdtree
std_train_avg_kd = MaxAbsScaler().fit_transform(text_train_avg_kd)
std_test_avg_kd = MaxAbsScaler().fit_transform(text_test_avg_kd)
print(std_train_avg_kd.shape)
print(std_test_avg_kd.shape)
```

```
(32250, 50)
(13822, 50)
(32250,)
(13822,)
(14000, 50)
(6000, 50)
```

AVGW2V(Brute Force)

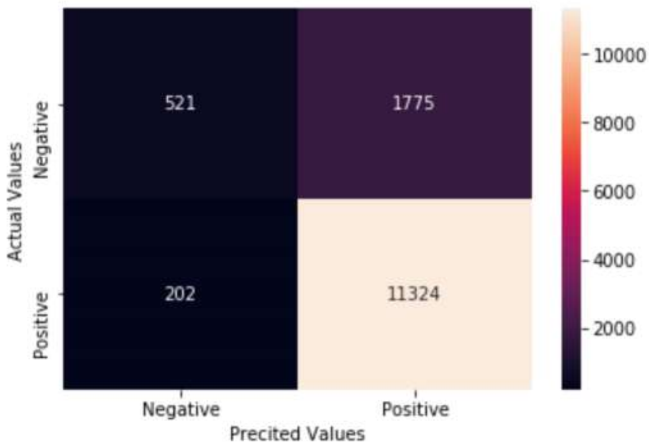
In [44]:
```
myList = list(range(0,50))
neighbors = list(filter(lambda x: x % 2 != 0, myList))
# empty list that will hold cv scores
cv_scores = []
# perform 10-fold cross validation
for k in tqdm(neighbors):
    knn = KNeighborsClassifier(n_neighbors=k,algorithm='brute', n_jobs=-1)
    scores = cross_val_score(knn, std_train_avg, score_train_avg, cv=10, scoring
='roc_auc')
    cv_scores.append(scores.mean())
# changing to misclassification error
MSE = [1 - x for x in cv_scores]

# determining best k
optimal_k = neighbors[MSE.index(min(MSE))]
plt.figure(figsize=(10,6))
plt.xlabel('K in KNN')
plt.ylabel('Misclassification Error')
plt.plot(neighbors, MSE, marker='o',markerfacecolor='green', markersize=10)
print('\nThe optimal number of neighbors is %d.' % optimal_k)
```

```
100%|██████████| 25/25 [26:00<00:00, 62.50s/it]

The optimal number of neighbors is 49.
```



In [45]:
```
#learning the model k = optimal_k
knn_optimal = KNeighborsClassifier(n_neighbors=optimal_k)

# fitting the model
knn_optimal.fit(std_train_avg, score_train_avg)

# predict on the test response
pred = knn_optimal.predict(std_test_avg)

# evaluate accuracy on test data
acc = accuracy_score(score_test_avg, pred) * 100

print('\nThe accuracy of the knn classifier for k = %d is %f%%' % (optimal_k, ac
c))

cm = confusion_matrix(score_test_avg, pred)
class_label = ['Negative', 'Positive']
conf_matrix = pd.DataFrame(cm, index=class_label, columns=class_label)
sns.heatmap(conf_matrix, annot=True, fmt='d')
plt.ylabel('Actual Values')
plt.xlabel('Precited Values')
plt.show()
```

```
The accuracy of the knn classifier for k = 49 is 85.696715%
```

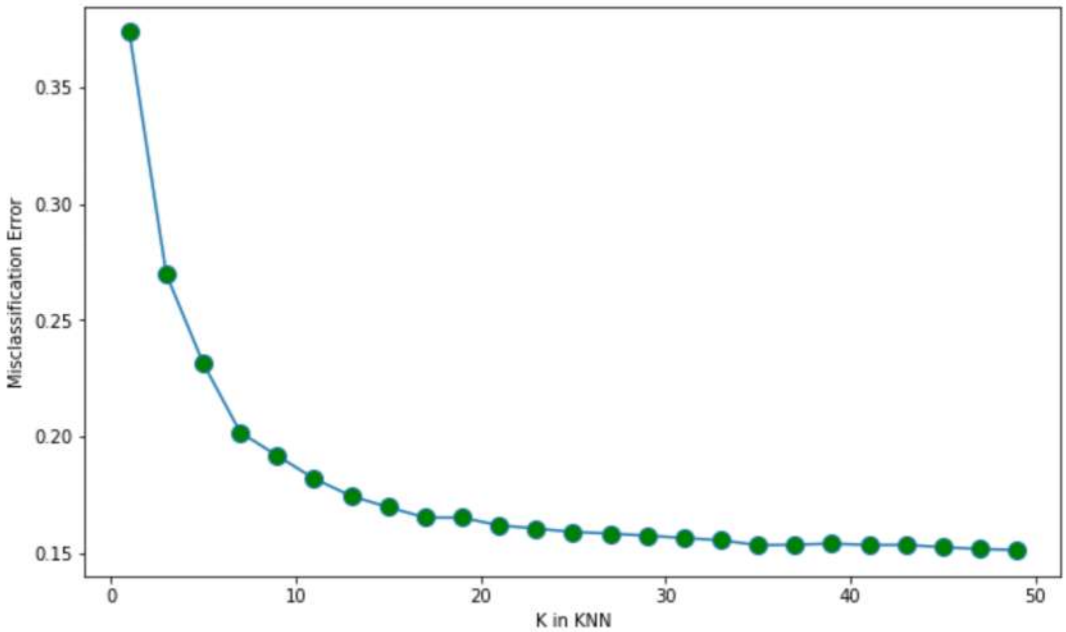


AVGW2V(Kd Tree)

In [46]:
```
#svd = TruncatedSVD(n_components=300)
#std_train_avg_kd = svd.fit_transform(std_train_avg_kd)
#std_test_avg_kd = svd.transform(std_test_avg_kd)
```

In [47]:
```
myList = list(range(0,50))
neighbors = list(filter(lambda x: x % 2 != 0, myList))
# empty list that will hold cv scores
cv_scores = []
# perform 10-fold cross validation
for k in tqdm(neighbors):
    knn = KNeighborsClassifier(n_neighbors=k,algorithm='kd_tree', n_jobs=-1)
    scores = cross_val_score(knn, std_train_avg_kd, score_train_avg_kd, cv=5, sc
oring='roc_auc')
    cv_scores.append(scores.mean())
# changing to misclassification error
MSE = [1 - x for x in cv_scores]

# determining best k
optimal_k = neighbors[MSE.index(min(MSE))]
plt.figure(figsize=(10,6))
plt.xlabel('K in KNN')
plt.ylabel('Misclassification Error')
plt.plot(neighbors, MSE, marker='o',markerfacecolor='green', markersize=10)
print('\nThe optimal number of neighbors is %d.' % optimal_k)
```

```
100%|██████████| 25/25 [03:01<00:00,  7.17s/it]

The optimal number of neighbors is 49.
```



In [48]:
```
#learning the model k = optimal_k
knn_optimal = KNeighborsClassifier(n_neighbors=optimal_k)

# fitting the model
knn_optimal.fit(std_train_avg_kd, score_train_avg_kd)

# predict on the test response
pred = knn_optimal.predict(std_test_avg_kd)

# evaluate accuracy on test data
acc = accuracy_score(score_test_avg_kd, pred) * 100

print('\nThe accuracy of the knn classifier for k = %d is %f%%' % (optimal_k, ac
c))

cm = confusion_matrix(score_test_avg_kd, pred)
class_label = ['Negative', 'Positive']
conf_matrix = pd.DataFrame(cm, index=class_label, columns=class_label)
sns.heatmap(conf_matrix, annot=True, fmt='d')
plt.ylabel('Actual Values')
plt.xlabel('Precited Values')
plt.show()
```

```
The accuracy of the knn classifier for k = 49 is 87.050000%
```

Tfidf and W2V

In [49]:
```
tfidf_vect = TfidfVectorizer(ngram_range=(1,2), min_df = 10, max_features = 500)
tfidf = tfidf_vect.fit_transform(Preproc_text)
print(len(Preproc_text))
```

```
46072
```

In [50]:
```
from IPython.display import clear_output
from tqdm import tqdm
features = tfidf_vect.get_feature_names()
tfidf_w2vs = []
row = 0
for sent in tqdm(Preproc_text):
    clear_output(wait=True)
    sent_vec = np.zeros(50)
    tfidf_sum = 0
    for word in sent.split():
        if(word in w2v_words) and word in features:
            vec = w2v.wv[word]
            tfidf_value = tfidf[row, features.index(word)]
            sent_vec += (vec * tfidf_value)
            tfidf_sum += tfidf_value
#     if(tfidf_sum != 0):
    sent_vec /= tfidf_sum
    tfidf_w2vs.append(sent_vec)
#     row += 1
    pass
```

```
100%|██████████| 46072/46072 [07:53<00:00, 97.23it/s]
```

In [51]:
```
tfidf_w2vs = np.nan_to_num(tfidf_w2vs)
score = np.nan_to_num(score)
text_train_w2v, text_test_w2v, score_train_w2v, score_test_w2v = train_test_spli
t(tfidf_w2vs, score, test_size=0.3,
                                                                  stratify=None,
random_state=0)
tfidf_w2vs_kd = tfidf_w2vs[0:20000]
score_w2v_kd = score[0:20000]
text_train_w2v_kd, text_test_w2v_kd, score_train_w2v_kd, score_test_w2v_kd = tra
in_test_split(tfidf_w2vs_kd, score_w2v_kd, test_size=0.3,
                                                                  stratify=None,
random_state=0)
```

In [52]:
```
#standardization of data
std_train_w2v = MaxAbsScaler().fit_transform(text_train_w2v)
std_test_w2v = MaxAbsScaler().fit_transform(text_test_w2v)
print(std_train_w2v.shape)
print(std_test_w2v.shape)
```

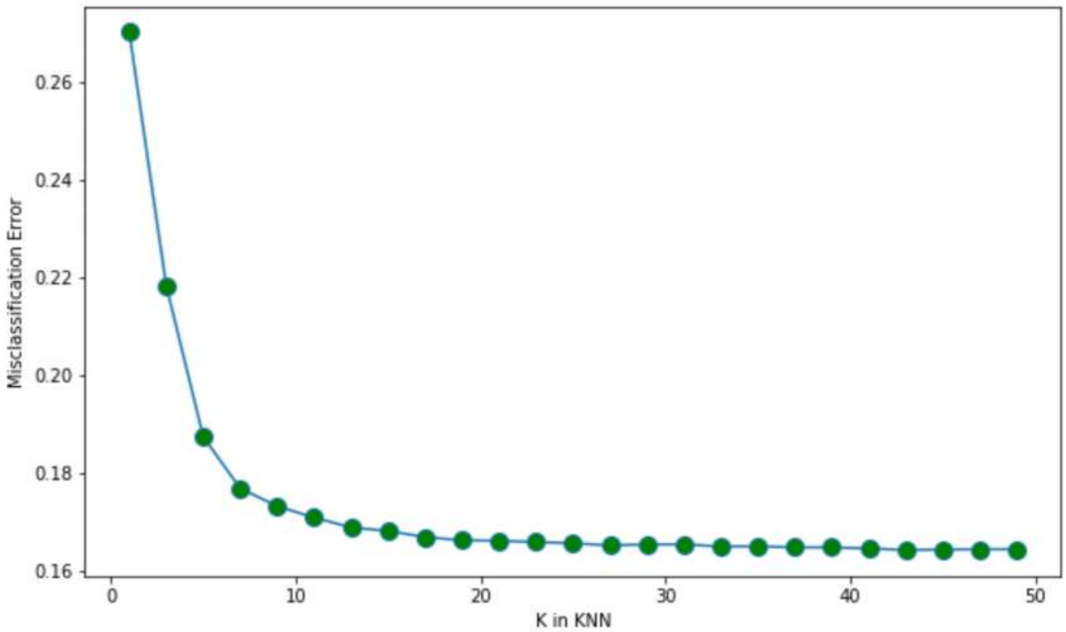
```
(32250, 50)
(13822, 50)
```

TFIDFW2V(Brute Force)

In [53]:

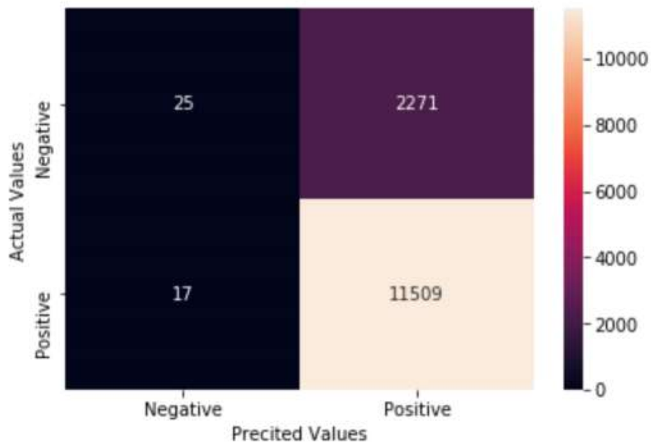
```
myList = list(range(0,50))
neighbors = list(filter(lambda x: x % 2 != 0, myList))
# empty list that will hold cv scores
cv_scores = []
# perform 10-fold cross validation
for k in tqdm(neighbors):
    knn = KNeighborsClassifier(n_neighbors=k,algorithm='brute', n_jobs=-1)
    scores = cross_val_score(knn, std_train_w2v, score_train_w2v, cv=10, scoring
='accuracy')
    cv_scores.append(scores.mean())
# changing to misclassification error
MSE = [1 - x for x in cv_scores]

# determining best k
optimal_k = neighbors[MSE.index(min(MSE))]
plt.figure(figsize=(10,6))
plt.xlabel('K in KNN')
plt.ylabel('Misclassification Error')
plt.plot(neighbors, MSE, marker='o',markerfacecolor='green', markersize=10)
print('\nThe optimal number of neighbors is %d.' % optimal_k)
```

```
100%|██████████| 25/25 [26:26<00:00, 64.19s/it]

The optimal number of neighbors is 43.
```



In [54]:
```
#learning the model k = optimal_k
knn_optimal = KNeighborsClassifier(n_neighbors=optimal_k)

# fitting the model
knn_optimal.fit(std_train_w2v, score_train_w2v)

# predict on the test response
pred = knn_optimal.predict(std_test_w2v)

# evaluate accuracy on test data
acc = accuracy_score(score_test_w2v, pred) * 100

print('\nThe accuracy of the knn classifier for k = %d is %f%%' % (optimal_k, ac
c))

cm = confusion_matrix(score_test_w2v, pred)
class_label = ['Negative', 'Positive']
conf_matrix = pd.DataFrame(cm, index=class_label, columns=class_label)
sns.heatmap(conf_matrix, annot=True, fmt='d')
plt.ylabel('Actual Values')
plt.xlabel('Precited Values')
plt.show()
```

```
The accuracy of the knn classifier for k = 43 is 83.446679%
```

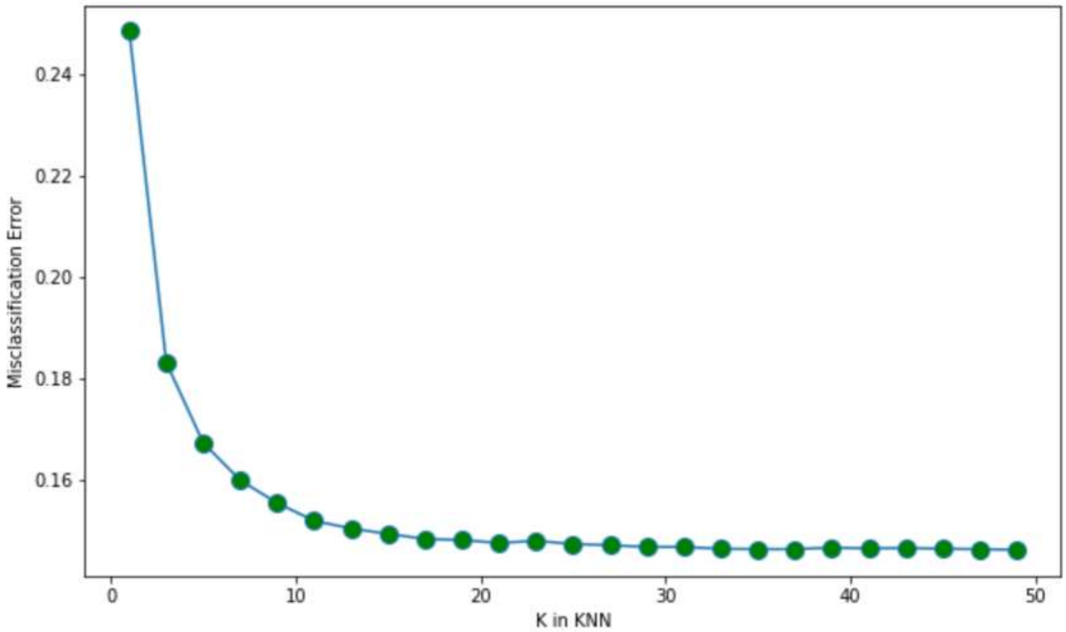


TFIDFW2V(Kd Tree)

In [55]:
```
std_train_w2v_kd = MaxAbsScaler().fit_transform(text_train_w2v_kd)
std_test_w2v_kd = MaxAbsScaler().fit_transform(text_test_w2v_kd)
```

In [56]:
```
myList = list(range(0,50))
neighbors = list(filter(lambda x: x % 2 != 0, myList))
# empty list that will hold cv scores
cv_scores = []
# perform 10-fold cross validation
for k in neighbors:
    knn = KNeighborsClassifier(n_neighbors=k,algorithm='kd_tree', n_jobs=-1)
    scores = cross_val_score(knn, std_train_w2v_kd, score_train_w2v_kd, cv=5, sc
oring='accuracy')
    cv_scores.append(scores.mean())
# changing to misclassification error
MSE = [1 - x for x in cv_scores]

# determining best k
optimal_k = neighbors[MSE.index(min(MSE))]
plt.figure(figsize=(10,6))
plt.xlabel('K in KNN')
plt.ylabel('Misclassification Error')
plt.plot(neighbors, MSE, marker='o',markerfacecolor='green', markersize=10)
print('\nThe optimal number of neighbors is %d.' % optimal_k)
```

```
The optimal number of neighbors is 49.
```



In [57]:
```
#learning the model k = optimal_k
knn_optimal = KNeighborsClassifier(n_neighbors=optimal_k)

# fitting the model
knn_optimal.fit(std_train_w2v_kd, score_train_w2v_kd)

# predict on the test response
pred = knn_optimal.predict(std_test_w2v_kd)

# evaluate accuracy on test data
acc = accuracy_score(score_test_w2v_kd, pred) * 100

print('\nThe accuracy of the knn classifier for k = %d is %f%%' % (optimal_k, ac
c))

cm = confusion_matrix(score_test_w2v_kd, pred)
class_label = ['Negative', 'Positive']
conf_matrix = pd.DataFrame(cm, index=class_label, columns=class_label)
sns.heatmap(conf_matrix, annot=True, fmt='d')
plt.ylabel('Actual Values')
plt.xlabel('Precited Values')
plt.show()
```

```
The accuracy of the knn classifier for k = 49 is 85.450000%
```

Conclusion

In [3]:
```
from IPython.display import HTML, display
import tabulate
table = [["S.NO.","MODEL || KNN ||","Best K","Test Accuracy"],
         ["1","BOW | brute |","49","83.4177"],
         ["2","BOW | kd_tree |","49","85.7500"],
         ["3","TF-IDF | brute |","29","83.3888"],
         ["4","TF-IDF | kd_tree |","49","85.0000"],
         ["5","AVG W2VEC | brute |","49","85.6967"],
         ["6","AVG W2VEC | kd_tree |","49","87.0500"],
         ["7","TF-IDF W2VEC | brute |","43","83.4466"],
         ["8","TF-IDF W2VEC | kd_tree |","49","85.4500"]]
display(HTML(tabulate.tabulate(table, tablefmt='html')))
```

S.NO.	MODEL || KNN ||	Best K	Test Accuracy
1	BOW	brute		49	83.4177
2	BOW	kd_tree		49	85.7500
3	TF-IDF	brute		29	83.3888
4	TF-IDF	kd_tree		49	85.0000
5	AVG W2VEC	brute		49	85.6967
6	AVG W2VEC	kd_tree		49	87.0500
7	TF-IDF W2VEC	brute		43	83.4466
8	TF-IDF W2VEC	kd_tree		49	85.4500