Perfoming Decision Trees on Amazon Fine Food Reviews

Data Source: https://www.kaggle.com/snap/amazon-fine-food-reviews (https://www.kaggle.com/snap/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 products : 74,258

Timespan : Oct 1999 - Oct 2012

Number of Attributes/Columns in data : 10

Attribute Information:

1. Id
2. ProductId - unique identifier for the product
3. UserId - unqiue identifier for the user
4. ProfileName
5. HelpfulnessNumerator - number of users who found the review helpful
6. HelpfulnessDenominator - number of users who indicated whether they found the review helpful or not
7. Score - rating between 1 and 5
8. Time - timestamp for the review
9. Summary - brief summary of the review
10. Text - text of the review

Objective:

Given a review, determine whether the review is positive (rating of 4 or 5) or negative (rating of 1 or 2).

1. Reading Data

1.1. Loading Data

The dataset is available in two forms

1. csv file
2. SQLite Database

In order to load the data, We have used the SQLITE dataset as it is easier to query the data and visualise the data efficiently. Here as we only want to get the global sentiment of the recommendations (positive or negative), we will purposefully ignore all Scores equal to 3. If the score is above 3, then the recommendation wil be set to "positive". Otherwise, it will be set to "negative".

In [2]:

```
# Importing libraries
import warnings
warnings.filterwarnings("ignore")

import sqlite3
import pandas as pd
import numpy as np
import nltk
import string
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.feature_extraction.text import TfidfVectorizer

from sklearn.feature_extraction.text import CountVectorizer
from nltk.stem.porter import PorterStemmer

import re

import string
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
from nltk.stem.wordnet import WordNetLemmatizer

from gensim.models import Word2Vec
from gensim.models import KeyedVectors
import pickle
```

In [3]:

```
# using the SQLite Table to read data.
con1 = sqlite3.connect('database.sqlite')

# Eliminating neutral reviews i.e. those reviews with Score = 3
filtered_data = pd.read_sql_query(" SELECT * FROM Reviews WHERE Score != 3 ", con1)

# Give reviews with Score>3 a positive rating, and reviews with a score<3 a negative rating
def polarity(x):
    if x < 3:
        return 'negative'
    return 'positive'

# Applying polarity function on Score column of filtered_data
filtered_data['Score'] = filtered_data['Score'].map(polarity)

print(filtered_data.shape)
filtered_data.head()
```

```
(525814, 10)
```

Out[3]:

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

2. Exploratory Data Analysis

2.1. Data Cleaning: Deduplication

It is observed (as shown in the table below) that the reviews data had many duplicate entries. Hence it was necessary to remove duplicates in order to get unbiased results for the analysis of the data. Following is an example:

In [4]:

```
#Sorting data according to ProductId in ascending order
sorted_data=filtered_data.sort_values('ProductId', axis=0, ascending=True, inplace=False, k

#Deduplication of entries
final=sorted_data.drop_duplicates(subset={"UserId","ProfileName","Time","Text"}, keep='firs
print(final.shape)

#Checking to see how much % of data still remains
((final.shape[0]*1.0)/(filtered_data.shape[0]*1.0)*100)
```

```
(364173, 10)
```

Out[4]:

```
69.25890143662969
```

In [5]:

```
# Removing rows where HelpfulnessNumerator is greater than HelpfulnessDenominator
final = final[final.HelpfulnessNumerator <= final.HelpfulnessDenominator]

print(final.shape)
final[30:50]
```

```
(364171, 10)
```

Out[5]:

	Id	ProductId	UserId	ProfileName	HelpfulnessNumerator	Helpfuln
138683	150501	0006641040	AJ46FKXOVC7NR	Nicholas A Mesiano	2	
138676	150493	0006641040	AMX0PJKV4PPNJ	E. R. Bird "Ramseelbird"	71	
138682	150500	0006641040	A1IJKK6Q1GTEAY	A Customer	2	
138681	150499	0006641040	A3E7R866M94L0C	L. Barker "simienwolf"	2	
476617	515426	141278509X	AB1A5EGHHVA9M	CHelmic	1	
22621	24751	2734888454	A1C298ITT645B6	Hugh G. Pritchard	0	
22620	24750	2734888454	A13ISQV0U9GZIC	Sandikaye	1	
284375	308077	2841233731	A3QD68O22M2XHQ	LABRNTH	0	
157850	171161	7310172001	AFXMWPNS1BLU4	H. Sandler	0	

	Id	ProductId	UserId	ProfileName	HelpfulnessNumerator	Helpfuln
157849	171160	7310172001	A74C7IARQEM1R	stucker	0	
157833	171144	7310172001	A1V5MY8V9AWUQB	Cheryl Sapper "champagne girl"	0	
157832	171143	7310172001	A2SWO60IW01VPX	Sam	0	
157837	171148	7310172001	A3TFTWTG2CC1GA	J. Umphress	0	
157831	171142	7310172001	A2ZO1AYFVQYG44	Cindy Rellie "Rellie"	0	
157830	171141	7310172001	AZ40270J4JBZN	Zhinka Chunmee "gamer from way back in the 70's"	0	
157829	171140	7310172001	ADXXVGRCGQQUO	Richard Pearlstein	0	
157828	171139	7310172001	A13MS1JQG2ADOJ	C. Perrone	0	
157827	171138	7310172001	A13LAE0YTXA11B	Dita Vyslouzilova "dita"	0	
157848	171159	7310172001	A16GY2RCF410DT	LB	0	
157834	171145	7310172001	A1L8DNQYY69L2Z	R. Flores	0	

OBSERVATION :- Here books with ProductId - 0006641040 and 2841233731 are also there so we have to remove all these rows with these ProductIds from the data

In [6]:

```
final = final[final['ProductId'] != '2841233731']
final = final[final['ProductId'] != '0006641040']
final.shape
```

Out[6]:

```
(364136, 10)
```

3. Preprocessing

3.1. Preprocessing Review Text

Now that we have finished deduplication our data requires some preprocessing before we go on further with analysis and making the prediction model.

Hence in the Preprocessing phase we do the following in the order below:-

1. Begin by removing the html tags
2. Remove any punctuations or limited set of special characters like , or . or # etc.
3. Check if the word is made up of english letters and is not alpha-numeric
4. Check to see if the length of the word is greater than 2 (as it was researched that there is no adjective in 2-letters)
5. Convert the word to lowercase
6. Remove Stopwords
7. Finally Snowball Stemming the word (it was obsereved to be better than Porter Stemming)

After which we collect the words used to describe positive and negative reviews

In [7]:

```
#set of stopwords in English
from nltk.corpus import stopwords
stop = set(stopwords.words('english'))
words_to_keep = set(('not'))
stop -= words_to_keep
#initialising the snowball stemmer
sno = 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
```

In [8]:

```
#Code for removing HTML tags , punctuations . Code for removing stopwords . Code for checki
# also greater than 2 . Code for stemmimg and also to convert them to lowercase letters
i=0
str1=' '
final_string=[]
all_positive_words=[] # store words from +ve reviews here
all_negative_words=[] # store words from -ve reviews here.
s=''
for sent in final['Text'].values:
    filtered_sentence=[]
    #print(sent);
    sent=cleanhtml(sent) # remove HTML tags
    for w in sent.split():
        for cleaned_words in cleanpunc(w).split():
            if((cleaned_words.isalpha()) & (len(cleaned_words)>2)):
                if(cleaned_words.lower() not in stop):
                    s=(sno.stem(cleaned_words.lower())).encode('utf8')
                    filtered_sentence.append(s)
                    if (final['Score'].values)[i] == 'positive':
                        all_positive_words.append(s) #list of all words used to describe po
                    if(final['Score'].values)[i] == 'negative':
                        all_negative_words.append(s) #list of all words used to describe ne
                else:
                    continue
            else:
                continue

    str1 = b" ".join(filtered_sentence) #final string of cleaned words


    final_string.append(str1)
    i+=1
```

In [9]:

```
#adding a column of CleanedText which displays the data after pre-processing of the review
final['CleanedText']=final_string
final['CleanedText']=final['CleanedText'].str.decode("utf-8")
#below the processed review can be seen in the CleanedText Column
print('Shape of final',final.shape)
final.head()
```

```
Shape of final (364136, 11)
```

Out[9]:

	Id	ProductId	UserId	ProfileName	HelpfulnessNumerator	Helpfulnes
476617	515426	141278509X	AB1A5EGHHVA9M	CHelmic	1	
22621	24751	2734888454	A1C298ITT645B6	Hugh G. Pritchard	0	
22620	24750	2734888454	A13ISQV0U9GZIC	Sandikaye	1	
157850	171161	7310172001	AFXMWPNS1BLU4	H. Sandler	0	
157849	171160	7310172001	A74C7IARQEM1R	stucker	0	

TIME BASED SPLITTING OF SAMPLE DATASET

In [10]:

```
from sklearn.model_selection import train_test_split
##Sorting data according to Time in ascending order for Time Based Splitting
time_sorted_data = final.sort_values('Time', axis=0, ascending=True, inplace=False, kind='q

x = time_sorted_data['CleanedText'].values
y = time_sorted_data['Score']

# split the data set into train and test
X_train, X_test, Y_train, Y_test = train_test_split(x, y, test_size=0.3, random_state=0)
```

Word2Vec

In [11]:

```
# List of sentence in X_train text
sent_of_train=[]
for sent in X_train:
    sent_of_train.append(sent.split())

# List of sentence in X_est text
sent_of_test=[]
for sent in X_test:
    sent_of_test.append(sent.split())

# Train your own Word2Vec model using your own train text corpus
# min_count = 5 considers only words that occured atleast 5 times
w2v_model=Word2Vec(sent_of_train,min_count=5,size=50, workers=4)

w2v_words = list(w2v_model.wv.vocab)
print("number of words that occured minimum 5 times ",len(w2v_words))
```

```
number of words that occured minimum 5 times  18825
```

(1). Avg Word2Vec

In [12]:

```
# compute average word2vec for each review for X_train .
train_vectors = [];
for sent in sent_of_train:
    sent_vec = np.zeros(50)
    cnt_words =0;
    for word in sent: #
        if word in w2v_words:
            vec = w2v_model.wv[word]
            sent_vec += vec
            cnt_words += 1
    if cnt_words != 0:
        sent_vec /= cnt_words
    train_vectors.append(sent_vec)

# compute average word2vec for each review for X_test .
test_vectors = [];
for sent in sent_of_test:
    sent_vec = np.zeros(50)
    cnt_words =0;
    for word in sent: #
        if word in w2v_words:
            vec = w2v_model.wv[word]
            sent_vec += vec
            cnt_words += 1
    if cnt_words != 0:
        sent_vec /= cnt_words
    test_vectors.append(sent_vec)

import warnings
warnings.filterwarnings('ignore')
# Data-preprocessing: Standardizing the data
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train_vec_standardized = sc.fit_transform(train_vectors)
X_test_vec_standardized = sc.transform(test_vectors)
```

GridSearchCV Implementation (Decision Tree)

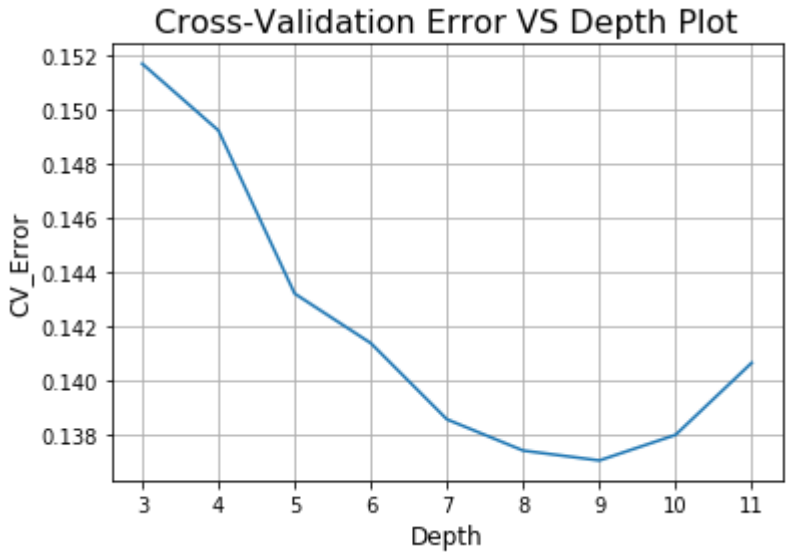
In [13]:

```
# Importing libraries
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score,confusion_matrix,f1_score,precision_score,recall

Depths = [3,4,5,6,7,8,9,10,11]

param_grid = {'max_depth': Depths}
model = GridSearchCV(DecisionTreeClassifier(), param_grid, scoring = 'accuracy', cv=3 , n_j
model.fit(X_train_vec_standardized, Y_train)
print("Model with best parameters :\n",model.best_estimator_)
print("Accuracy of the model : ",model.score(X_test_vec_standardized, Y_test))

# Cross-Validation errors
cv_errors = [1-i for i in model.cv_results_['mean_test_score']]

# Optimal value of depth
optimal_depth = model.best_estimator_.max_depth
print("The optimal value of depth is : ",optimal_depth)

# DecisionTreeClassifier with Optimal value of depth
dt = DecisionTreeClassifier(max_depth=optimal_depth)
dt.fit(X_train_vec_standardized,Y_train)
predictions = dt.predict(X_test_vec_standardized)

# Variables that will be used for  making table in Conclusion part of this assignment
avg_w2v_depth = optimal_depth
avg_w2v_train_acc = model.score(X_test_vec_standardized, Y_test)*100
avg_w2v_test_acc = accuracy_score(Y_test, predictions) * 100
```

```
Model with best parameters :
 DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=9,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, presort=False, random_state=None,
            splitter='best')
Accuracy of the model :  0.8662773134628939
The optimal value of depth is :  9
```

In [14]:

```
# plotting Cross-Validation Error vs Depth graph
plt.plot(Depths, cv_errors)
plt.xlabel('Depth',size=12)
plt.ylabel('CV_Error',size=12)
plt.title('Cross-Validation Error VS Depth Plot',size=16)
plt.grid()
plt.show()
```



In [15]:

```
# evaluate accuracy
acc = accuracy_score(Y_test, predictions) * 100
print('\nThe Test Accuracy of the DecisionTreeClassifier for depth = %d is %f%%' % (optimal

# evaluate precision
acc = precision_score(Y_test, predictions, pos_label = 'positive')
print('\nThe Test Precision of the DecisionTreeClassifier for depth = %d is %f' % (optimal_

# evaluate recall
acc = recall_score(Y_test, predictions, pos_label = 'positive')
print('\nThe Test Recall of the DecisionTreeClassifier for depth = %d is %f' % (optimal_dep

# evaluate f1-score
acc = f1_score(Y_test, predictions, pos_label = 'positive')
print('\nThe Test F1-Score of the DecisionTreeClassifier for depth = %d is %f' % (optimal_c
```

```
The Test Accuracy of the DecisionTreeClassifier for depth = 9 is 86.622239%

The Test Precision of the DecisionTreeClassifier for depth = 9 is 0.886291

The Test Recall of the DecisionTreeClassifier for depth = 9 is 0.965320

The Test F1-Score of the DecisionTreeClassifier for depth = 9 is 0.924119
```

Visualize Decision Tree

In [16]:

```
# Importing libraries
from sklearn import tree
import pydotplus
from IPython.display import Image
from IPython.display import SVG
from graphviz import Source
from IPython.display import display

target = ['negative','positive']
# Create DOT data
data = tree.export_graphviz(dt,out_file=None,class_names=target,filled=True,rounded=True,sp

# Draw graph
graph = pydotplus.graph_from_dot_data(data)
#graph = Source(data)

# Show graph
Image(graph.create_png())
#display(SVG(graph.pipe(format='svg')))
```

```
dot: graph is too large for cairo-renderer bitmaps. Scaling by 0.35064 to fi
t
```

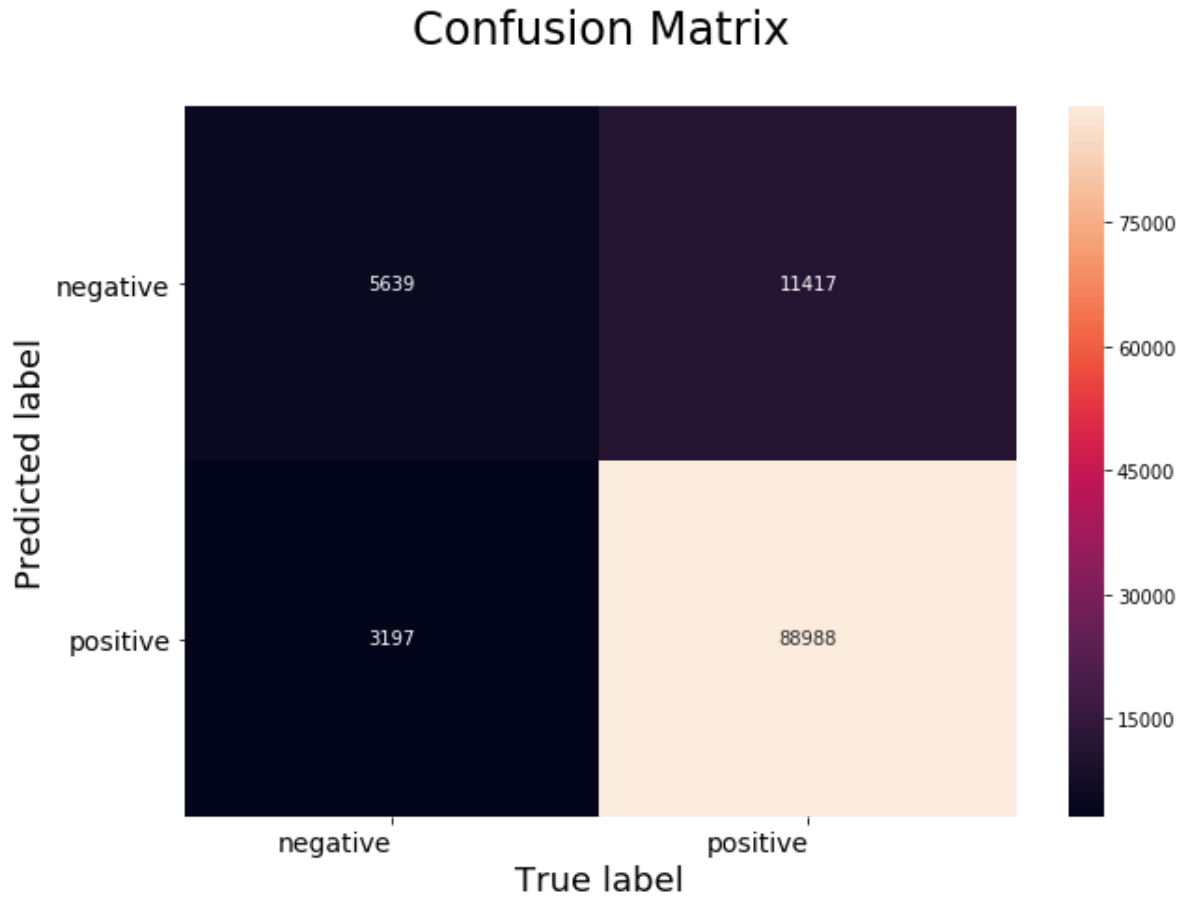
Out[16]:

NOTE :- Decision Tree is very large to visualize . Later in the end we will visualize it with small sample dataset

SEABORN HEATMAP FOR REPRESENTATION OF CONFUSION MATRIX :

In [17]:

```
# Code for drawing seaborn heatmaps
class_names = ['negative','positive']
df_heatmap = pd.DataFrame(confusion_matrix(Y_test, predictions), index=class_names, columns
fig = plt.figure(figsize=(10,7))
heatmap = sns.heatmap(df_heatmap, annot=True, fmt="d")

# Setting tick labels for heatmap
heatmap.yaxis.set_ticklabels(heatmap.yaxis.get_ticklabels(), rotation=0, ha='right', fontsi
heatmap.xaxis.set_ticklabels(heatmap.xaxis.get_ticklabels(), rotation=0, ha='right', fontsi
plt.ylabel('Predicted label',size=18)
plt.xlabel('True label',size=18)
plt.title("Confusion Matrix\n",size=24)
plt.show()
```



(2). TFIDF-Word2Vec

NOTE :- It is taking a lot off time to perform TFIDF-Word2Vec on whole 364K rows of data . So , I am performing it with only 100K rows . Please don't mind because I am unable to perform it with whole data due to poor condition of my laptop . But I am completing all the steps as was asked .

RANDOMLY SAMPLING 100K POINTS OUT OF WHOLE DATASET

In [18]:

```
# We will collect different 100K rows without repetition from time_sorted_data dataframe
my_final = time_sorted_data.take(np.random.permutation(len(final))[:100000])
print(my_final.shape)

x = my_final['CleanedText'].values
y = my_final['Score']

# split the data set into train and test
X_train, X_test, Y_train, Y_test = train_test_split(x, y, test_size=0.3, random_state=0)

# List of sentence in X_train text
sent_of_train=[]
for sent in X_train:
    sent_of_train.append(sent.split())

# List of sentence in X_est text
sent_of_test=[]
for sent in X_test:
    sent_of_test.append(sent.split())

w2v_model=Word2Vec(sent_of_train,min_count=5,size=50, workers=4)
w2v_words = list(w2v_model.wv.vocab)
```

```
(100000, 11)
```

In [19]:

```
# TF-IDF weighted Word2Vec
tf_idf_vect = TfidfVectorizer()

# final_tf_idf1 is the sparse matrix with row= sentence, col=word and cell_val = tfidf
final_tf_idf1 = tf_idf_vect.fit_transform(X_train)

# tfidf words/col-names
tfidf_feat = tf_idf_vect.get_feature_names()

# compute TFIDF Weighted Word2Vec for each review for X_test .
tfidf_test_vectors = [];
row=0;
for sent in sent_of_test:
    sent_vec = np.zeros(50)
    weight_sum =0;
    for word in sent:
        if word in w2v_words:
            vec = w2v_model.wv[word]
            # obtain the tf_idfidf of a word in a sentence/review
            tf_idf = final_tf_idf1[row, tfidf_feat.index(word)]
            sent_vec += (vec * tf_idf)
            weight_sum += tf_idf
    if weight_sum != 0:
        sent_vec /= weight_sum
    tfidf_test_vectors.append(sent_vec)
    row += 1
```

In [20]:

```
# compute TFIDF Weighted Word2Vec for each review for X_train .
tfidf_train_vectors = [];
row=0;
for sent in sent_of_train:
    sent_vec = np.zeros(50)
    weight_sum =0;
    for word in sent:
        if word in w2v_words:
            vec = w2v_model.wv[word]
            # obtain the tf_idfidf of a word in a sentence/review
            tf_idf = final_tf_idf1[row, tfidf_feat.index(word)]
            sent_vec += (vec * tf_idf)
            weight_sum += tf_idf
    if weight_sum != 0:
        sent_vec /= weight_sum
    tfidf_train_vectors.append(sent_vec)
    row += 1

# Data-preprocessing: Standardizing the data
sc = StandardScaler()
X_train_vec_standardized = sc.fit_transform(tfidf_train_vectors)
X_test_vec_standardized = sc.transform(tfidf_test_vectors)
```

GridSearchCV Implementation (Decision Tree)

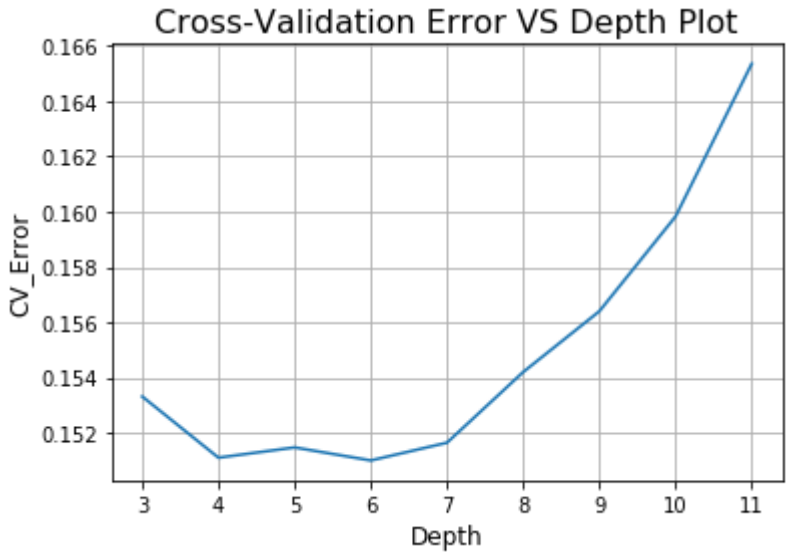
In [21]:

```
Depths = [3,4,5,6,7,8,9,10,11]

param_grid = {'max_depth': Depths}
model = GridSearchCV(DecisionTreeClassifier(), param_grid, scoring = 'accuracy', cv=3 , n_j
model.fit(X_train_vec_standardized, Y_train)
print("Model with best parameters :\n",model.best_estimator_)
print("Accuracy of the model : ",model.score(X_test_vec_standardized, Y_test))

# Cross-Validation errors
cv_errors = [1-i for i in model.cv_results_['mean_test_score']]

# Optimal value of depth
optimal_depth = model.best_estimator_.max_depth
print("The optimal value of depth is : ",optimal_depth)

# DecisionTreeClassifier with Optimal value of depth
dt = DecisionTreeClassifier(max_depth=optimal_depth)
dt.fit(X_train_vec_standardized,Y_train)
predictions = dt.predict(X_test_vec_standardized)

# Variables that will be used for  making table in Conclusion part of this assignment
tfidf_w2v_depth = optimal_depth
tfidf_w2v_train_acc = model.score(X_test_vec_standardized, Y_test)*100
tfidf_w2v_test_acc = accuracy_score(Y_test, predictions) * 100
```

```
Model with best parameters :
 DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=6,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, presort=False, random_state=None,
            splitter='best')
Accuracy of the model :  0.8101333333333334
The optimal value of depth is :  6
```

In [22]:

```
# plotting Cross-Validation Error vs Depth graph
plt.plot(Depths, cv_errors)
plt.xlabel('Depth',size=12)
plt.ylabel('CV_Error',size=12)
plt.title('Cross-Validation Error VS Depth Plot',size=16)
plt.grid()
plt.show()
```



In [23]:

```
# evaluate accuracy
acc = accuracy_score(Y_test, predictions) * 100
print('\nThe Test Accuracy of the DecisionTreeClassifier for depth = %d is %f%%' % (optimal

# evaluate precision
acc = precision_score(Y_test, predictions, pos_label = 'positive')
print('\nThe Test Precision of the DecisionTreeClassifier for depth = %d is %f' % (optimal_

# evaluate recall
acc = recall_score(Y_test, predictions, pos_label = 'positive')
print('\nThe Test Recall of the DecisionTreeClassifier for depth = %d is %f' % (optimal_dep

# evaluate f1-score
acc = f1_score(Y_test, predictions, pos_label = 'positive')
print('\nThe Test F1-Score of the DecisionTreeClassifier for depth = %d is %f' % (optimal_c
```

```
The Test Accuracy of the DecisionTreeClassifier for depth = 6 is 81.013333%

The Test Precision of the DecisionTreeClassifier for depth = 6 is 0.845783

The Test Recall of the DecisionTreeClassifier for depth = 6 is 0.946889

The Test F1-Score of the DecisionTreeClassifier for depth = 6 is 0.893485
```

Visualize Decision Tree

In [24]:

```
target = ['negative','positive']
# Create DOT data
data = tree.export_graphviz(dt,out_file=None,class_names=target,filled=True,rounded=True,sp

# Draw graph
graph = pydotplus.graph_from_dot_data(data)

# Show graph
Image(graph.create_png())
```

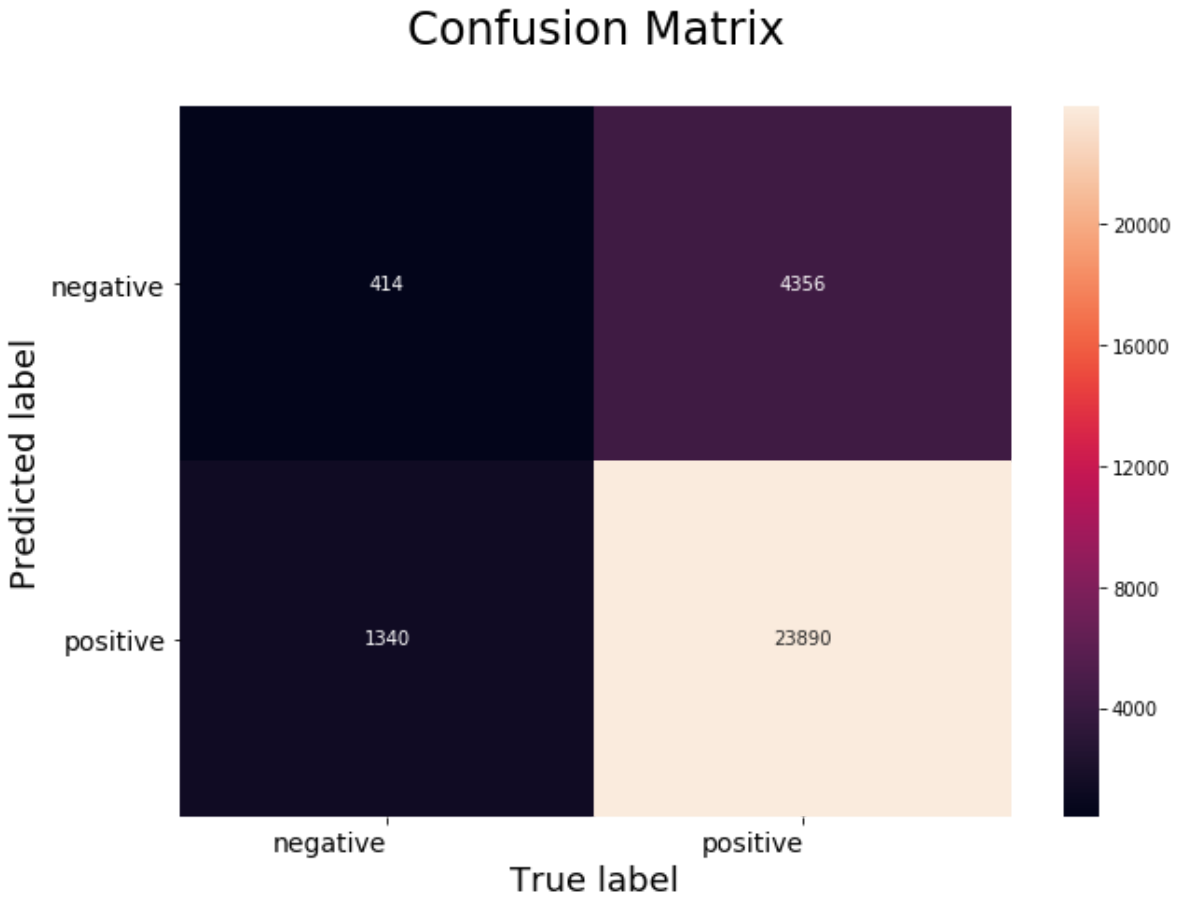
Out[24]:

NOTE :- Decision Tree is very large to visualize . Later in the end we will visualize it with small sample dataset

SEABORN HEATMAP FOR REPRESENTATION OF CONFUSION MATRIX :

In [25]:

```
# Code for drawing seaborn heatmaps
class_names = ['negative','positive']
df_heatmap = pd.DataFrame(confusion_matrix(Y_test, predictions), index=class_names, columns
fig = plt.figure(figsize=(10,7))
heatmap = sns.heatmap(df_heatmap, annot=True, fmt="d")

# Setting tick labels for heatmap
heatmap.yaxis.set_ticklabels(heatmap.yaxis.get_ticklabels(), rotation=0, ha='right', fontsi
heatmap.xaxis.set_ticklabels(heatmap.xaxis.get_ticklabels(), rotation=0, ha='right', fontsi
plt.ylabel('Predicted label',size=18)
plt.xlabel('True label',size=18)
plt.title("Confusion Matrix\n",size=24)
plt.show()
```



Implementing Decision Tree on Small Sample for BoW and TFIDF

RANDOMLY SAMPLING 40K POINTS OUT OF WHOLE DATASET

In [26]:

```
# We will collect different 40K rows without repetition from time_sorted_data dataframe
my_final = time_sorted_data.take(np.random.permutation(len(final))[:40000])
print(my_final.shape)

x = my_final['CleanedText'].values
y = my_final['Score']

# split the data set into train and test
X_train, X_test, Y_train, Y_test = train_test_split(x, y, test_size=0.3, random_state=0)
```

```
(40000, 11)
```

(3). Bag of Words (BoG)

In [27]:

```
#BoW
count_vect = CountVectorizer(min_df = 1000)
X_train_vec = count_vect.fit_transform(X_train)
X_test_vec = count_vect.transform(X_test)
print("the type of count vectorizer :",type(X_train_vec))
print("the shape of out text BOW vectorizer : ",X_train_vec.get_shape())
print("the number of unique words :", X_train_vec.get_shape()[1])
```

```
the type of count vectorizer : <class 'scipy.sparse.csr.csr_matrix'>
the shape of out text BOW vectorizer :  (28000, 174)
the number of unique words : 174
```

In [28]:

```
sc = StandardScaler(with_mean=False)
X_train_vec_standardized = sc.fit_transform(X_train_vec)
X_test_vec_standardized = sc.transform(X_test_vec)
```

GridSearchCV Implementation (Decision Tree)

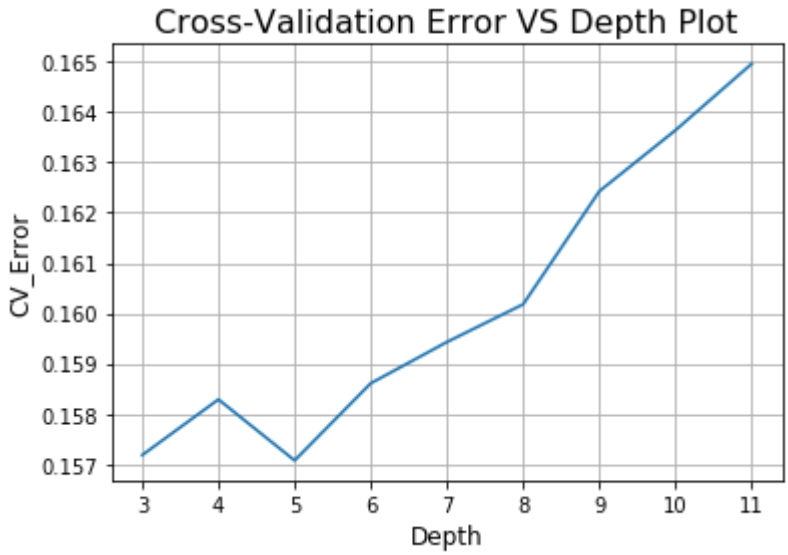
In [29]:

```
Depths = [3,4,5,6,7,8,9,10,11]

param_grid = {'max_depth': Depths}
model = GridSearchCV(DecisionTreeClassifier(), param_grid, scoring = 'accuracy', cv=3 , n_j
model.fit(X_train_vec_standardized, Y_train)
print("Model with best parameters :\n",model.best_estimator_)
print("Accuracy of the model : ",model.score(X_test_vec_standardized, Y_test))

# Cross-Validation errors
cv_errors = [1-i for i in model.cv_results_['mean_test_score']]

# Optimal value of depth
optimal_depth = model.best_estimator_.max_depth
print("The optimal value of depth is : ",optimal_depth)

# DecisionTreeClassifier with Optimal value of depth
dt = DecisionTreeClassifier(max_depth=optimal_depth)
dt.fit(X_train_vec_standardized,Y_train)
predictions = dt.predict(X_test_vec_standardized)

# Variables that will be used for  making table in Conclusion part of this assignment
bow_depth = optimal_depth
bow_train_acc = model.score(X_test_vec_standardized, Y_test)*100
bow_test_acc = accuracy_score(Y_test, predictions) * 100
```

```
Model with best parameters :
 DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=5,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, presort=False, random_state=None,
            splitter='best')
Accuracy of the model :  0.8413333333333334
The optimal value of depth is :  5
```

In [30]:

```
# plotting Cross-Validation Error vs Depth graph
plt.plot(Depths, cv_errors)
plt.xlabel('Depth',size=12)
plt.ylabel('CV_Error',size=12)
plt.title('Cross-Validation Error VS Depth Plot',size=16)
plt.grid()
plt.show()
```



In [31]:

```
# evaluate accuracy
acc = accuracy_score(Y_test, predictions) * 100
print('\nThe Test Accuracy of the DecisionTreeClassifier for depth = %d is %f%%' % (optimal

# evaluate precision
acc = precision_score(Y_test, predictions, pos_label = 'positive')
print('\nThe Test Precision of the DecisionTreeClassifier for depth = %d is %f' % (optimal_

# evaluate recall
acc = recall_score(Y_test, predictions, pos_label = 'positive')
print('\nThe Test Recall of the DecisionTreeClassifier for depth = %d is %f' % (optimal_dep

# evaluate f1-score
acc = f1_score(Y_test, predictions, pos_label = 'positive')
print('\nThe Test F1-Score of the DecisionTreeClassifier for depth = %d is %f' % (optimal_c
```

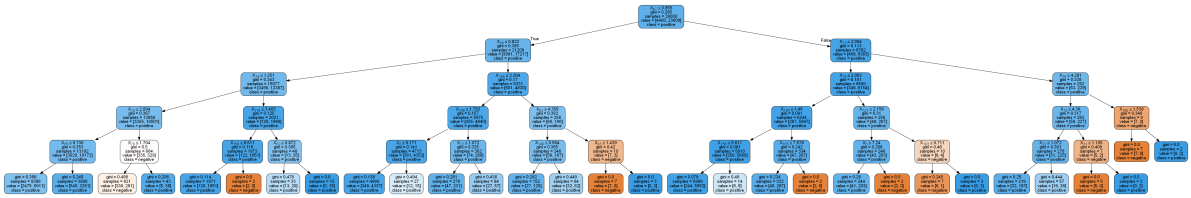
```
The Test Accuracy of the DecisionTreeClassifier for depth = 5 is 84.166667%

The Test Precision of the DecisionTreeClassifier for depth = 5 is 0.849957

The Test Recall of the DecisionTreeClassifier for depth = 5 is 0.985836

The Test F1-Score of the DecisionTreeClassifier for depth = 5 is 0.912868
```

Visualize Decision Tree

In [32]:

```
target = ['negative','positive']
# Create DOT data
data = tree.export_graphviz(dt,out_file=None,class_names=target,filled=True,rounded=True,sp

# Draw graph
graph = pydotplus.graph_from_dot_data(data)

# Show graph
Image(graph.create_png())
```

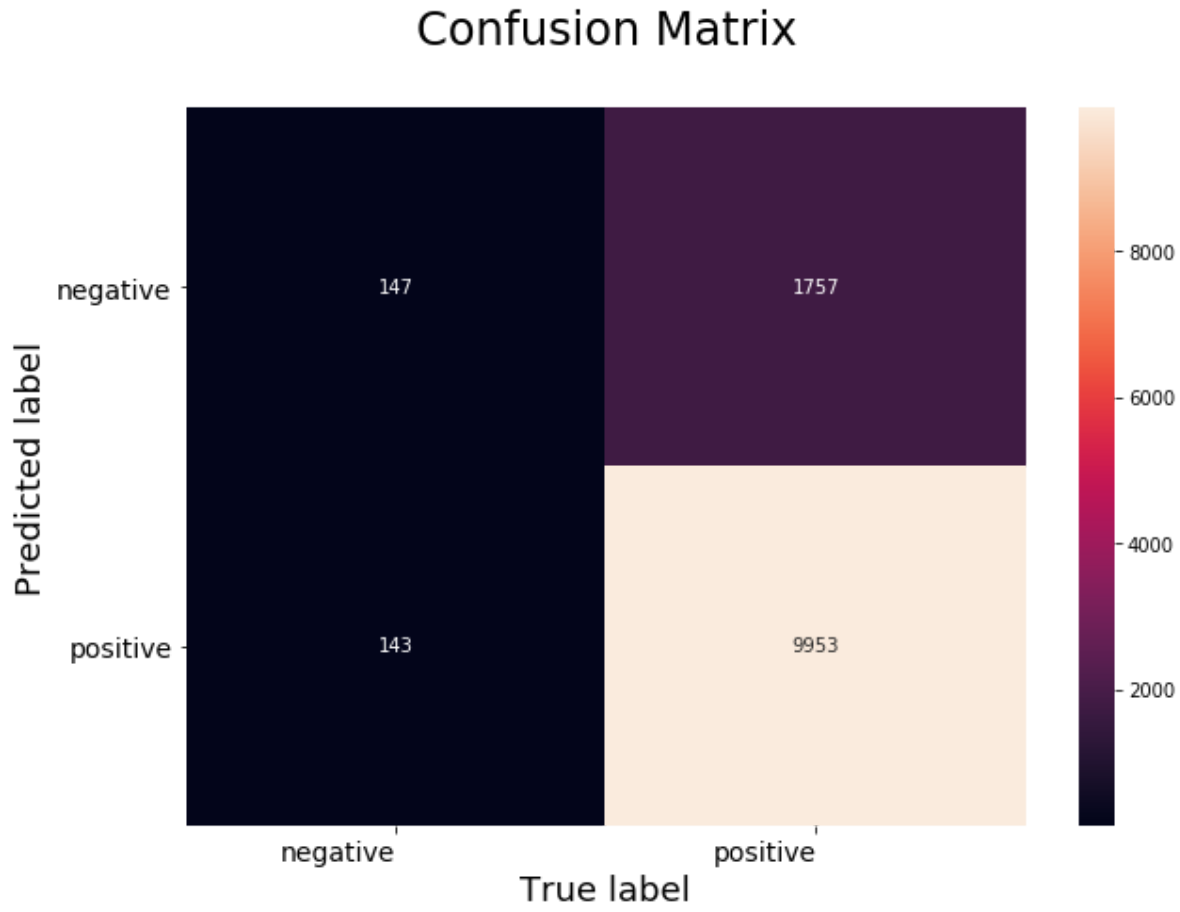
Out[32]:



SEABORN HEATMAP FOR REPRESENTATION OF CONFUSION MATRIX :

In [33]:

```
# Code for drawing seaborn heatmaps
class_names = ['negative','positive']
df_heatmap = pd.DataFrame(confusion_matrix(Y_test, predictions), index=class_names, columns
fig = plt.figure(figsize=(10,7))
heatmap = sns.heatmap(df_heatmap, annot=True, fmt="d")

# Setting tick labels for heatmap
heatmap.yaxis.set_ticklabels(heatmap.yaxis.get_ticklabels(), rotation=0, ha='right', fontsi
heatmap.xaxis.set_ticklabels(heatmap.xaxis.get_ticklabels(), rotation=0, ha='right', fontsi
plt.ylabel('Predicted label',size=18)
plt.xlabel('True label',size=18)
plt.title("Confusion Matrix\n",size=24)
plt.show()
```



(4). TFIDF

In [34]:

```
tf_idf_vect = TfidfVectorizer(min_df=1000)
X_train_vec = tf_idf_vect.fit_transform(X_train)
X_test_vec = tf_idf_vect.transform(X_test)
print("the type of count vectorizer :",type(X_train_vec))
print("the shape of out text TFIDF vectorizer : ",X_train_vec.get_shape())
print("the number of unique words :", X_train_vec.get_shape()[1])

# Data-preprocessing: Standardizing the data
sc = StandardScaler(with_mean=False)
X_train_vec_standardized = sc.fit_transform(X_train_vec)
X_test_vec_standardized = sc.transform(X_test_vec)
```

```
the type of count vectorizer : <class 'scipy.sparse.csr.csr_matrix'>
the shape of out text TFIDF vectorizer :  (28000, 174)
the number of unique words : 174
```

GridSearchCV Implementation (Decision Tree)

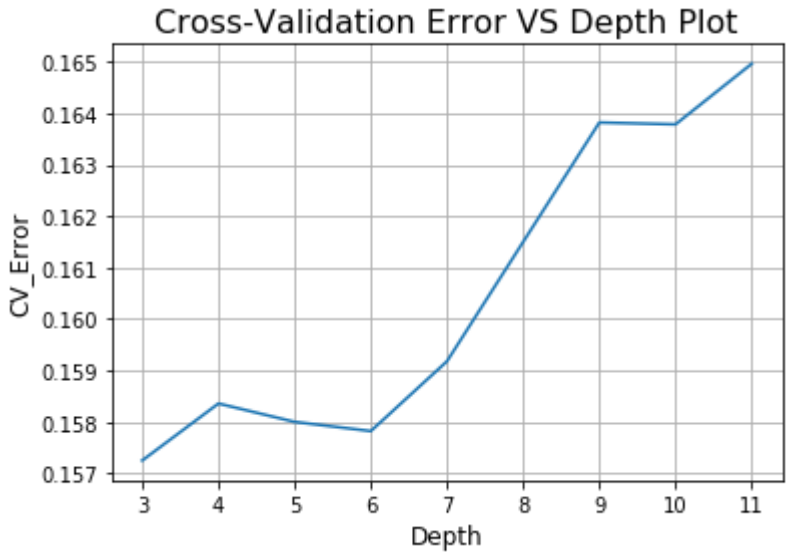
In [35]:

```
Depths = [3,4,5,6,7,8,9,10,11]

param_grid = {'max_depth': Depths}
model = GridSearchCV(DecisionTreeClassifier(), param_grid, scoring = 'accuracy', cv=3 , n_j
model.fit(X_train_vec_standardized, Y_train)
print("Model with best parameters :\n",model.best_estimator_)
print("Accuracy of the model : ",model.score(X_test_vec_standardized, Y_test))

# Cross-Validation errors
cv_errors = [1-i for i in model.cv_results_['mean_test_score']]

# Optimal value of depth
optimal_depth = model.best_estimator_.max_depth
print("The optimal value of depth is : ",optimal_depth)

# DecisionTreeClassifier with Optimal value of depth
dt = DecisionTreeClassifier(max_depth=optimal_depth)
dt.fit(X_train_vec_standardized,Y_train)
predictions = dt.predict(X_test_vec_standardized)

# Variables that will be used for  making table in Conclusion part of this assignment
tfidf_depth = optimal_depth
tfidf_train_acc = model.score(X_test_vec_standardized, Y_test)*100
tfidf_test_acc = accuracy_score(Y_test, predictions) * 100
```

```
Model with best parameters :
 DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=3,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, presort=False, random_state=None,
            splitter='best')
Accuracy of the model :  0.8410833333333333
The optimal value of depth is :  3
```

In [36]:

```
# plotting Cross-Validation Error vs Depth graph
plt.plot(Depths, cv_errors)
plt.xlabel('Depth',size=12)
plt.ylabel('CV_Error',size=12)
plt.title('Cross-Validation Error VS Depth Plot',size=16)
plt.grid()
plt.show()
```



In [37]:

```
# evaluate accuracy
acc = accuracy_score(Y_test, predictions) * 100
print('\nThe Test Accuracy of the DecisionTreeClassifier for depth = %d is %f%%' % (optimal

# evaluate precision
acc = precision_score(Y_test, predictions, pos_label = 'positive')
print('\nThe Test Precision of the DecisionTreeClassifier for depth = %d is %f' % (optimal_

# evaluate recall
acc = recall_score(Y_test, predictions, pos_label = 'positive')
print('\nThe Test Recall of the DecisionTreeClassifier for depth = %d is %f' % (optimal_dep

# evaluate f1-score
acc = f1_score(Y_test, predictions, pos_label = 'positive')
print('\nThe Test F1-Score of the DecisionTreeClassifier for depth = %d is %f' % (optimal_c
```

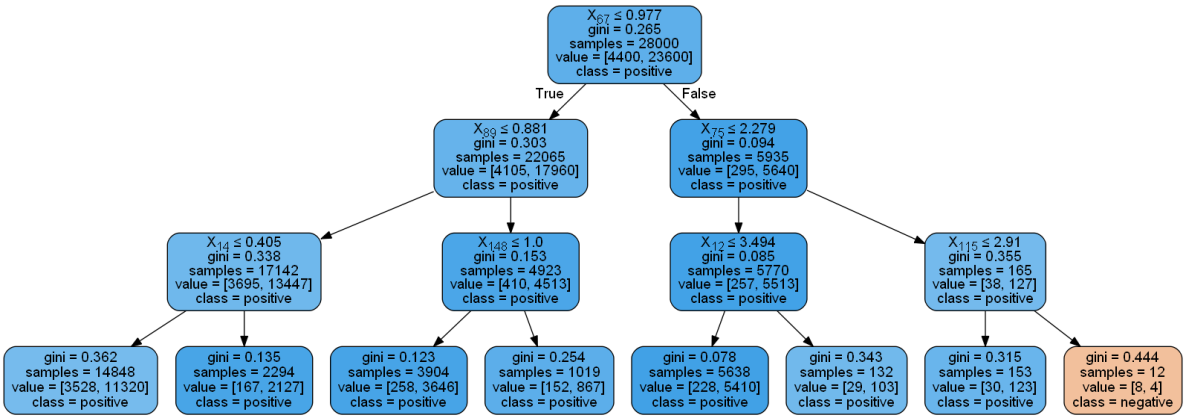
```
The Test Accuracy of the DecisionTreeClassifier for depth = 3 is 84.108333%

The Test Precision of the DecisionTreeClassifier for depth = 3 is 0.841351

The Test Recall of the DecisionTreeClassifier for depth = 3 is 0.999604

The Test F1-Score of the DecisionTreeClassifier for depth = 3 is 0.913675
```

Visualize Decision Tree

In [38]:

```
target = ['negative','positive']
# Create DOT data
data = tree.export_graphviz(dt,out_file=None,class_names=target,filled=True,rounded=True,sp

# Draw graph
graph = pydotplus.graph_from_dot_data(data)

# Show graph
Image(graph.create_png())
```

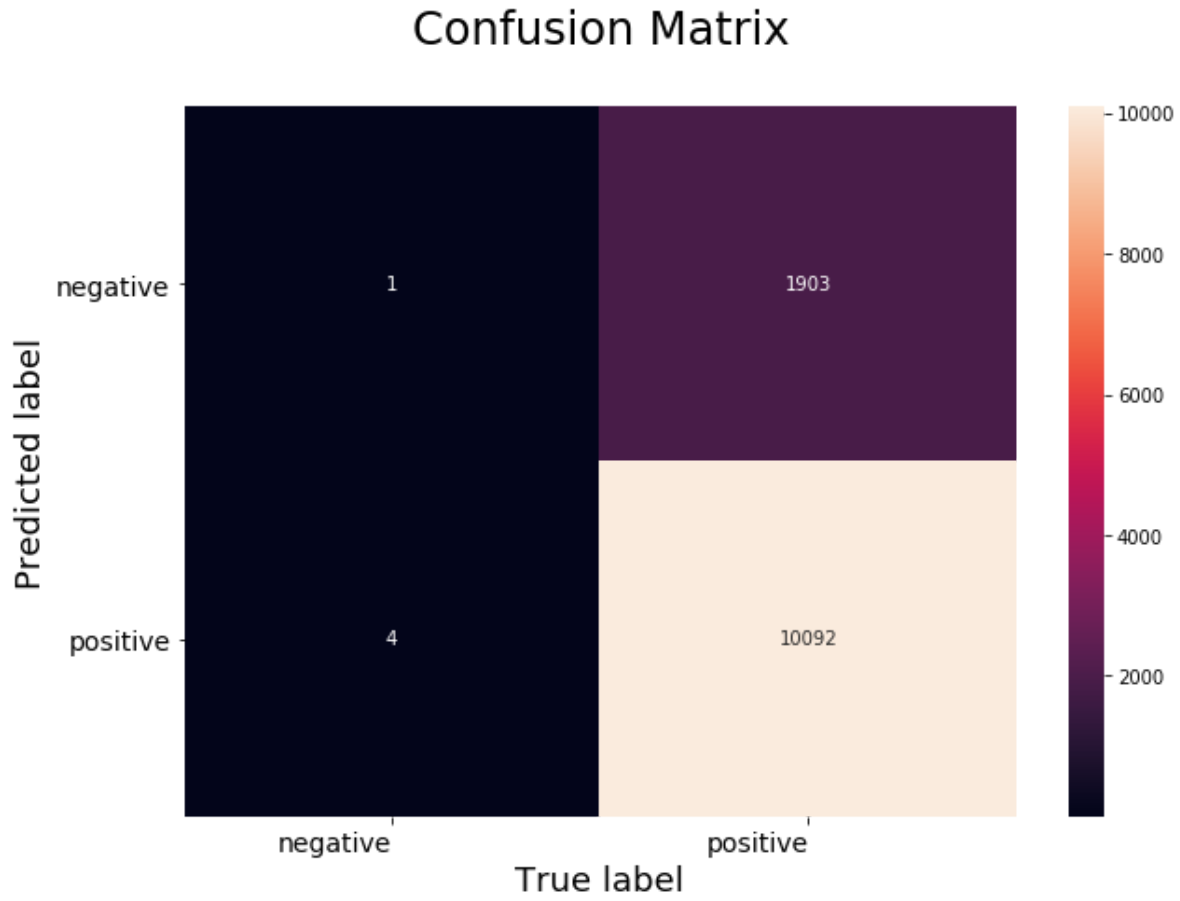
Out[38]:



SEABORN HEATMAP FOR REPRESENTATION OF CONFUSION MATRIX :

In [39]:

```
# Code for drawing seaborn heatmaps
class_names = ['negative','positive']
df_heatmap = pd.DataFrame(confusion_matrix(Y_test, predictions), index=class_names, columns
fig = plt.figure(figsize=(10,7))
heatmap = sns.heatmap(df_heatmap, annot=True, fmt="d")

# Setting tick labels for heatmap
heatmap.yaxis.set_ticklabels(heatmap.yaxis.get_ticklabels(), rotation=0, ha='right', fontsi
heatmap.xaxis.set_ticklabels(heatmap.xaxis.get_ticklabels(), rotation=0, ha='right', fontsi
plt.ylabel('Predicted label',size=18)
plt.xlabel('True label',size=18)
plt.title("Confusion Matrix\n",size=24)
plt.show()
```



CONCLUSION :-

(a). Procedure followed :

STEP 1 :- Text Preprocessing

STEP 2:- Time-based splitting of whole dataset into train_data and test_data

STEP 3:- Training the vectorizer on train_data and later applying same vectorizer on both train_data and test_data to transform them into vectors

STEP 4:- Using Decision_Tree as an estimator in GridSearchCV in order to find optimal value of depth of the tree

STEP 5:- Once , we get optimal value of depth then train Decision_Tree again with this optimal depth and make predictions on test_data

STEP 6:- Draw Cross-Validation Error vs Depth graph

STEP 7 :- Evaluate : Accuracy , F1-Score , Precision , Recall

STEP 8:- Visualizing the Decision Tree using Graphviz

STEP 9:- Draw Seaborn Heatmap for Confusion Matrix .

Repeat from STEP 3 to STEP 9 for each of these four vectorizers : BoW, TFIDF, Avg Word2Vec and TFIDF Word2Vec

(b). Table (Model Performances with their hyperparameters :

In [41]:

```
# Creating table using PrettyTable library
from prettytable import PrettyTable

# Names of the models
names =['Decision_Tree for BoW','Decision_Tree for TFIDF','Decision_Tree for Avg_Word2Vec',

# Values of optimal depth
optimal_depth = [bow_depth, tfidf_depth,avg_w2v_depth, tfidf_w2v_depth]

# Training Accuracies
train_acc = [bow_train_acc,tfidf_train_acc,avg_w2v_train_acc, tfidf_w2v_train_acc]

# Test Accuracies
test_acc = [bow_test_acc,tfidf_test_acc,avg_w2v_test_acc, tfidf_w2v_test_acc]
numbering = [1,2,3,4]

# Initializing prettytable
ptable = PrettyTable()

# Adding columns
ptable.add_column("S.NO.",numbering)
ptable.add_column("MODEL",names)
ptable.add_column("Optimal Depth",optimal_depth)
ptable.add_column("Training Accuracy",train_acc)
ptable.add_column("Test Accuracy",test_acc)

# Printing the Table
print(ptable)
```

```
+-------+---------------------------------+---------------+----------------
---+-------------------+
| S.NO. |              MODEL              | Optimal Depth | Training Accura
cy |   Test Accuracy   |
+-------+---------------------------------+---------------+----------------
---+-------------------+
|   1   |      Decision_Tree for BoW      |       5       | 84.133333333333
34 | 84.16666666666667 |
|   2   |     Decision_Tree for TFIDF     |       3       | 84.108333333333
33 | 84.10833333333333 |
|   3   |  Decision_Tree for Avg_Word2Vec  |       9       |  86.62773134628
94 |  86.622238902976  |
|   4   | Decision_Tree for tfidf_Word2Vec |       6       | 81.013333333333
34 | 81.01333333333334 |
+-------+---------------------------------+---------------+----------------
---+-------------------+
```

In [ ]: