Personalized cancer diagnosis

1. Business Problem

1.1. Description

Source: https://www.kaggle.com/c/msk-redefining-cancer-treatment/

Data: Memorial Sloan Kettering Cancer Center (MSKCC)

Download training_variants.zip and training_text.zip from Kaggle.

Context:

Source: https://www.kaggle.com/c/msk-redefining-cancer-treatment/discussion/35336#198462

Problem statement :

Classify the given genetic variations/mutations based on evidence from text-based clinical literature.

1.2. Source/Useful Links

Some articles and reference blogs about the problem statement

1. https://www.forbes.com/sites/matthewherper/2017/06/03/a-new-cancer-drug-helped-almost-everyone-who-took-it-almost-heres-what-it-teaches-us/#2a44ee2f6b25
2. https://www.youtube.com/watch?v=UwbuW7oK8rk
3. https://www.youtube.com/watch?v=qxXRKVompI8

1.3. Real-world/Business objectives and constraints.

- No low-latency requirement.
- Interpretability is important.
- Errors can be very costly.
- Probability of a data-point belonging to each class is needed.

2. Machine Learning Problem Formulation

2.1. Data

2.1.1. Data Overview

- Source: https://www.kaggle.com/c/msk-redefining-cancer-treatment/data
- We have two data files: one conatins the information about the genetic mutations and the other contains the clinical evidence (text) that human experts/pathologists use to classify the genetic mutations.
- Both these data files are have a common column called ID
- Data file's information:
  - training_variants (ID , Gene, Variations, Class)
  - training_text (ID, Text)

2.1.2. Example Data Point

training_variants

ID,Gene,Variation,Class
0,FAM58A,Truncating Mutations,1
1,CBL,W802*,2
2,CBL,Q249E,2
...

training_text

ID,Text
0||Cyclin-dependent kinases (CDKs) regulate a variety of fundamental cellular processes. CDK10 stands out as one of the last orphan CDKs for which no activating cyclin has been identified and no kinase activity revealed. Previous work has shown that CDK10 silencing increases ETS2 (v-ets erythroblastosis virus E26 oncogene homolog 2)-driven activation of the MAPK pathway, which confers tamoxifen resistance to breast cancer cells. The precise mechanisms by which CDK10 modulates ETS2 activity, and more generally the functions of CDK10, remain elusive. Here we demonstrate that CDK10 is a cyclin-dependent kinase by identifying cyclin M as an activating cyclin. Cyclin M, an orphan cyclin, is the product of FAM58A, whose mutations cause STAR syndrome, a human developmental anomaly whose features include toe syndactyly, telecanthus, and anogenital and renal malformations. We show that STAR syndrome-associated cyclin M mutants are unable to interact with CDK10. Cyclin M silencing phenocopies CDK10 silencing in increasing c-Raf and in conferring tamoxifen resistance to breast cancer cells. CDK10/cyclin M phosphorylates ETS2 in vitro, and in cells it positively controls ETS2 degradation by the proteasome. ETS2 protein levels are increased in cells derived from a STAR patient, and this increase is attributable to decreased cyclin M levels. Altogether, our results reveal an additional regulatory mechanism for ETS2, which plays key roles in cancer and development. They also shed light on the molecular mechanisms underlying STAR syndrome.Cyclin-dependent kinases (CDKs) play a pivotal role in the control of a number of fundamental cellular processes (1). The human genome contains 21 genes encoding proteins that can be considered as members of the CDK family owing to their sequence similarity with bona fide CDKs, those known to be activated by cyclins (2). Although discovered almost 20 y ago (3, 4), CDK10 remains one of the two CDKs without an identified cyclin partner. This knowledge gap has largely impeded the exploration of its biological functions. CDK10 can act as a positive cell cycle regulator in some cells (5, 6) or as a tumor suppressor in others (7, 8). CDK10 interacts with the ETS2 (v-ets erythroblastosis virus E26 oncogene homolog 2) transcription factor and inhibits its transcriptional activity through an unknown mechanism (9). CDK10 knockdown derepresses ETS2, which increases the expression of the c-Raf protein kinase, activates the MAPK pathway, and induces resistance of MCF7 cells to tamoxifen (6). ...

2.2. Mapping the real-world problem to an ML problem

2.2.1. Type of Machine Learning Problem

There are nine different classes a genetic mutation can be classified into => Multi class classification problem

2.2.2. Performance Metric

Source: https://www.kaggle.com/c/msk-redefining-cancer-treatment#evaluation

Metric(s):

- Multi class log-loss
- Confusion matrix

2.2.3. Machine Learing Objectives and Constraints

Objective: Predict the probability of each data-point belonging to each of the nine classes.

Constraints:

- Interpretability
- Class probabilities are needed.
- Penalize the errors in class probabilites => Metric is Log-loss.
- No Latency constraints.

2.3. Train, CV and Test Datasets

Split the dataset randomly into three parts train, cross validation and test with 64%,16%, 20% of data respectively

3. Exploratory Data Analysis

In [19]:
```
import pandas as pd
import matplotlib.pyplot as plt
import re
import time
import warnings
import numpy as np
from nltk.corpus import stopwords
from sklearn.decomposition import TruncatedSVD
from sklearn.preprocessing import normalize
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.manifold import TSNE
```

```
import seaborn as sns
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import confusion_matrix
from sklearn.metrics.classification import accuracy_score, log_loss
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import SGDClassifier
# from imblearn.over_sampling import SMOTE
from collections import Counter
from scipy.sparse import hstack
# from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import SVC
from sklearn.model_selection import StratifiedKFold
from collections import Counter, defaultdict
from sklearn.calibration import CalibratedClassifierCV
from sklearn.naive_bayes import MultinomialNB
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.feature_extraction.text import TfidfVectorizer
import math
from sklearn.metrics import normalized_mutual_info_score
from sklearn.ensemble import RandomForestClassifier
warnings.filterwarnings("ignore")

from mlxtend.classifier import StackingClassifier

from sklearn import model_selection
from sklearn.linear_model import LogisticRegression
```

3.1. Reading Data

3.1.1. Reading Gene and Variation Data

In [20]:
```
data = pd.read_csv('training_variants')
print('Number of data points : ', data.shape[0])
print('Number of features : ', data.shape[1])
print('Features : ', data.columns.values)
data.head()
```

```
Number of data points :  3321
Number of features :  4
Features :  ['ID' 'Gene' 'Variation' 'Class']
```

Out[20]:

	ID	Gene	Variation	Class
0	0	FAM58A	Truncating Mutations	1
1	1	CBL	W802*	2
2	2	CBL	Q249E	2
3	3	CBL	N454D	3
4	4	CBL	L399V	4

training/training_variants is a comma separated file containing the description of the genetic mutations used for training.
Fields are

- **ID :** the id of the row used to link the mutation to the clinical evidence
- **Gene :** the gene where this genetic mutation is located
- **Variation :** the aminoacid change for this mutations
- **Class :** 1-9 the class this genetic mutation has been classified on

3.1.2. Reading Text Data

In [21]:
```
# note the seprator in this file
data_text =pd.read_csv("training_text",sep="\|\|",engine="python",names=["ID",
"TEXT"],skiprows=1)
print('Number of data points : ', data_text.shape[0])
print('Number of features : ', data_text.shape[1])
print('Features : ', data_text.columns.values)
data_text.head()
```

```
Number of data points :  3321
Number of features :  2
Features :  ['ID' 'TEXT']
```

Out[21]:

	ID	TEXT
0	0	Cyclin-dependent kinases (CDKs) regulate a var...
1	1	Abstract Background Non-small cell lung canc...
2	2	Abstract Background Non-small cell lung canc...
3	3	Recent evidence has demonstrated that acquired...
4	4	Oncogenic mutations in the monomeric Casitas B...

3.1.3. Preprocessing of text

In [22]:
```
# loading stop words from nltk library
stop_words = set(stopwords.words('english'))


def nlp_preprocessing(total_text, index, column):
    if type(total_text) is not int:
        string = ""
        # replace every special char with space
        total_text = re.sub('[^a-zA-Z0-9\n]', ' ', total_text)
        # replace multiple spaces with single space
        total_text = re.sub('\s+',' ', total_text)
        # converting all the chars into lower-case.
        total_text = total_text.lower()

        for word in total_text.split():
        # if the word is a not a stop word then retain that word from the data
            if not word in stop_words:
                string += word + " "

        data_text[column][index] = string
```

In [23]:
```
#text processing stage.
start_time = time.clock()
for index, row in data_text.iterrows():
```

```
    if type(row['TEXT']) is str:
        nlp_preprocessing(row['TEXT'], index, 'TEXT')
    else:
        print("there is no text description for id:",index)
print('Time took for preprocessing the text :',time.clock() - start_time, "seconds")
```

```
there is no text description for id: 1109
there is no text description for id: 1277
there is no text description for id: 1407
there is no text description for id: 1639
there is no text description for id: 2755
Time took for preprocessing the text : 252.40018529999998 seconds
```

In [24]:
```
#merging both gene_variations and text data based on ID
result = pd.merge(data, data_text,on='ID', how='left')
result.head()
```

Out[24]:

	ID	Gene	Variation	Class	TEXT
0	0	FAM58A	Truncating Mutations	1	cyclin dependent kinases cdks regulate variety...
1	1	CBL	W802*	2	abstract background non small cell lung cancer...
2	2	CBL	Q249E	2	abstract background non small cell lung cancer...
3	3	CBL	N454D	3	recent evidence demonstrated acquired uniparen...
4	4	CBL	L399V	4	oncogenic mutations monomeric casitas b lineag...

In [25]:
```
result[result.isnull().any(axis=1)]
```

Out[25]:

	ID	Gene	Variation	Class	TEXT
1109	1109	FANCA	S1088F	1	NaN
1277	1277	ARID5B	Truncating Mutations	1	NaN
1407	1407	FGFR3	K508M	6	NaN
1639	1639	FLT1	Amplification	6	NaN
2755	2755	BRAF	G596C	7	NaN

In [26]:
```
result.loc[result['TEXT'].isnull(),'TEXT'] = result['Gene'] +' '+result['Variation']
```

In [27]:
```
result[result['ID']==1109]
```

Out[27]:

	ID	Gene	Variation	Class	TEXT
1109	1109	FANCA	S1088F	1	FANCA S1088F

3.1.4. Test, Train and Cross Validation Split

3.1.4.1. Splitting data into train, test and cross validation (64:20:16)

In [28]:
```
y_true = result['Class'].values
result.Gene      = result.Gene.str.replace('\s+', '_')
```

```
result.Variation = result.Variation.str.replace('\s+', '_')

# split the data into test and train by maintaining same distribution of outpu
t varaible 'y_true' [stratify=y_true]
X_train, test_df, y_train, y_test = train_test_split(result, y_true, stratify=
y_true, test_size=0.2)
# split the train data into train and cross validation by maintaining same dis
tribution of output varaible 'y_train' [stratify=y_train]
train_df, cv_df, y_train, y_cv = train_test_split(X_train, y_train, stratify=y
_train, test_size=0.2)
```

We split the data into train, test and cross validation data sets, preserving the ratio of class distribution in the original data set

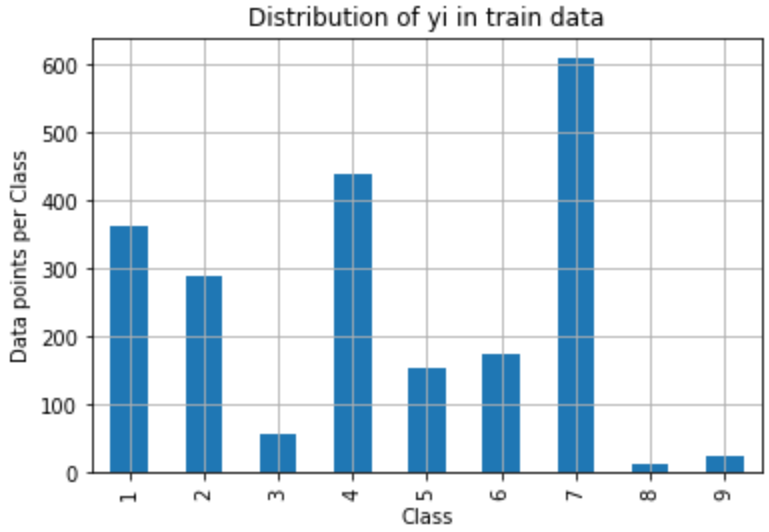
In [32]:
```
print('Number of data points in train data:', train_df.shape[0])
print('Number of data points in test data:', test_df.shape[0])
print('Number of data points in cross validation data:', cv_df.shape[0])
```

```
Number of data points in train data: 2124
Number of data points in test data: 665
Number of data points in cross validation data: 532
```

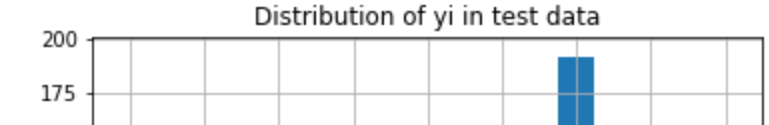
3.1.4.2. Distribution of y_i's in Train, Test and Cross Validation datasets

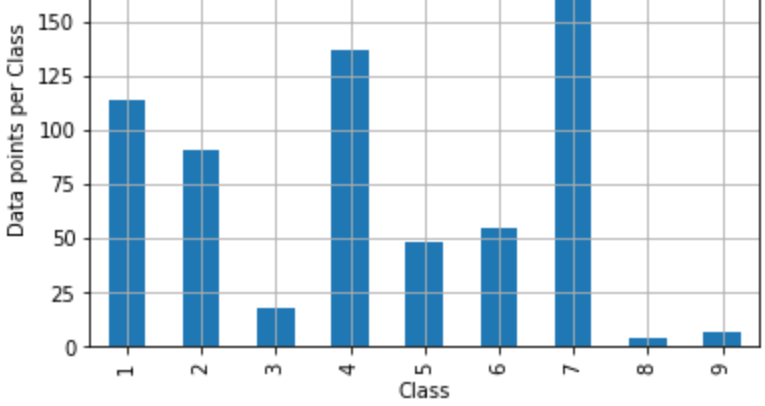
In [13]:
```
# it returns a dict, keys as class labels and values as the number of data poi
nts in that class
train_class_distribution = train_df['Class'].value_counts().sort_index()
test_class_distribution = test_df['Class'].value_counts().sort_index()
cv_class_distribution = cv_df['Class'].value_counts().sort_index()

my_colors = 'rgbkymc'
train_class_distribution.plot(kind='bar')
plt.xlabel('Class')
plt.ylabel('Data points per Class')

plt.title('Distribution of yi in train data')
plt.grid()
plt.show()

# ref: argsort https://docs.scipy.org/doc/numpy/reference/generated/numpy.args
ort.html
# -(train_class_distribution.values): the minus sign will give us in decreasin
g order
sorted_yi = np.argsort(-train_class_distribution.values)
for i in sorted_yi:
    print('Number of data points in class', i+1, ':',train_class_distribution.
values[i], '(', np.round((train_class_distribution.values[i]/train_df.shape[0]
*100), 3), '%)')


print('-'*80)
my_colors = 'rgbkymc'
test_class_distribution.plot(kind='bar')
plt.xlabel('Class')
plt.ylabel('Data points per Class')
plt.title('Distribution of yi in test data')
plt.grid()
plt.show()
```

```
# ref: argsort https://docs.scipy.org/doc/numpy/reference/generated/numpy.argsort.html
# -(train_class_distribution.values): the minus sign will give us in decreasing order
sorted_yi = np.argsort(-test_class_distribution.values)
for i in sorted_yi:
    print('Number of data points in class', i+1, ':',test_class_distribution.values[i], '(', np.round((test_class_distribution.values[i]/test_df.shape[0]*100), 3), '%)')

print('-'*80)
my_colors = 'rgbkymc'
cv_class_distribution.plot(kind='bar')
plt.xlabel('Class')
plt.ylabel('Data points per Class')
plt.title('Distribution of yi in cross validation data')
plt.grid()
plt.show()

# ref: argsort https://docs.scipy.org/doc/numpy/reference/generated/numpy.argsort.html
# -(train_class_distribution.values): the minus sign will give us in decreasing order
sorted_yi = np.argsort(-train_class_distribution.values)
for i in sorted_yi:
    print('Number of data points in class', i+1, ':',cv_class_distribution.values[i], '(', np.round((cv_class_distribution.values[i]/cv_df.shape[0]*100), 3), '%)')
```

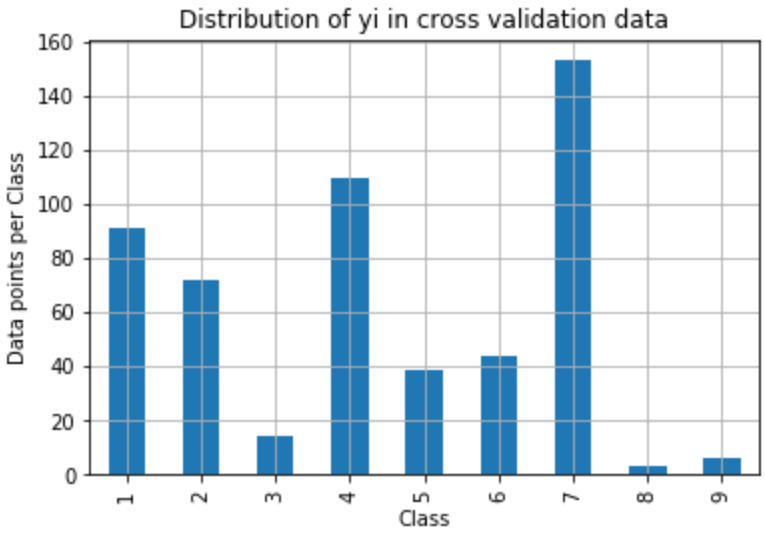



```
Number of data points in class 7 : 609 ( 28.672 %)
Number of data points in class 4 : 439 ( 20.669 %)
Number of data points in class 1 : 363 ( 17.09 %)
Number of data points in class 2 : 289 ( 13.606 %)
Number of data points in class 6 : 176 ( 8.286 %)
Number of data points in class 5 : 155 ( 7.298 %)

Number of data points in class 3 : 57 ( 2.684 %)
Number of data points in class 9 : 24 ( 1.13 %)
Number of data points in class 8 : 12 ( 0.565 %)
--------------------------------------------------------------------------------
```






```
Number of data points in class 7 : 191 ( 28.722 %)
Number of data points in class 4 : 137 ( 20.602 %)
Number of data points in class 1 : 114 ( 17.143 %)
Number of data points in class 2 : 91 ( 13.684 %)
Number of data points in class 6 : 55 ( 8.271 %)
Number of data points in class 5 : 48 ( 7.218 %)
Number of data points in class 3 : 18 ( 2.707 %)
Number of data points in class 9 : 7 ( 1.053 %)
Number of data points in class 8 : 4 ( 0.602 %)
------------------------------------------------------------------------
----
```




```
Number of data points in class 7 : 153 ( 28.759 %)
Number of data points in class 4 : 110 ( 20.677 %)
Number of data points in class 1 : 91 ( 17.105 %)
Number of data points in class 2 : 72 ( 13.534 %)
Number of data points in class 6 : 44 ( 8.271 %)
Number of data points in class 5 : 39 ( 7.331 %)
Number of data points in class 3 : 14 ( 2.632 %)
Number of data points in class 9 : 6 ( 1.128 %)
Number of data points in class 8 : 3 ( 0.564 %)
```

3.2 Prediction using a 'Random' Model

In a 'Random' Model, we generate the NINE class probabilites randomly such that they sum to 1.

In [14]:
```
# This function plots the confusion matrices given y_i, y_i_hat.
def plot_confusion_matrix(test_y, predict_y):
```

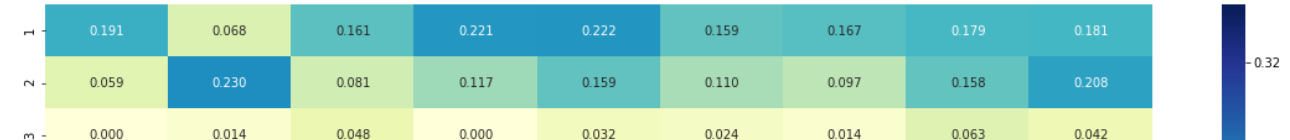
```
    C = confusion_matrix(test_y, predict_y)
    # C = 9,9 matrix, each cell (i,j) represents number of points of class i a
re predicted class j

    A =(((C.T)/(C.sum(axis=1))).T)
    #divid each element of the confusion matrix with the sum of elements in th
at column

    # C = [[1, 2],
    #     [3, 4]]
    # C.T = [[1, 3],
    #        [2, 4]]
    # C.sum(axis = 1)  axis=0 corresonds to columns and axis=1 corresponds to
 rows in two diamensional array
    # C.sum(axix =1) = [[3, 7]]
    # ((C.T)/(C.sum(axis=1))) = [[1/3, 3/7]
    #                            [2/3, 4/7]]

    # ((C.T)/(C.sum(axis=1))).T = [[1/3, 2/3]
    #                              [3/7, 4/7]]
    # sum of row elements = 1

    B =(C/C.sum(axis=0))
    #divid each element of the confusion matrix with the sum of elements in th
at row
    # C = [[1, 2],
    #     [3, 4]]
    # C.sum(axis = 0)  axis=0 corresonds to columns and axis=1 corresponds to
 rows in two diamensional array
    # C.sum(axix =0) = [[4, 6]]
    # (C/C.sum(axis=0)) = [[1/4, 2/6],
    #                      [3/4, 4/6]]

    labels = [1,2,3,4,5,6,7,8,9]
    # representing A in heatmap format
    print("-"*20, "Confusion matrix", "-"*20)
    plt.figure(figsize=(20,7))
    sns.heatmap(C, annot=True, cmap="YlGnBu", fmt=".3f", xticklabels=labels, y
ticklabels=labels)
    plt.xlabel('Predicted Class')
    plt.ylabel('Original Class')
    plt.show()

    print("-"*20, "Precision matrix (Columm Sum=1)", "-"*20)
    plt.figure(figsize=(20,7))
    sns.heatmap(B, annot=True, cmap="YlGnBu", fmt=".3f", xticklabels=labels, y
ticklabels=labels)
    plt.xlabel('Predicted Class')
    plt.ylabel('Original Class')
    plt.show()

    # representing B in heatmap format
    print("-"*20, "Recall matrix (Row sum=1)", "-"*20)
    plt.figure(figsize=(20,7))
    sns.heatmap(A, annot=True, cmap="YlGnBu", fmt=".3f", xticklabels=labels, y
ticklabels=labels)
    plt.xlabel('Predicted Class')
    plt.ylabel('Original Class')
    plt.show()
```

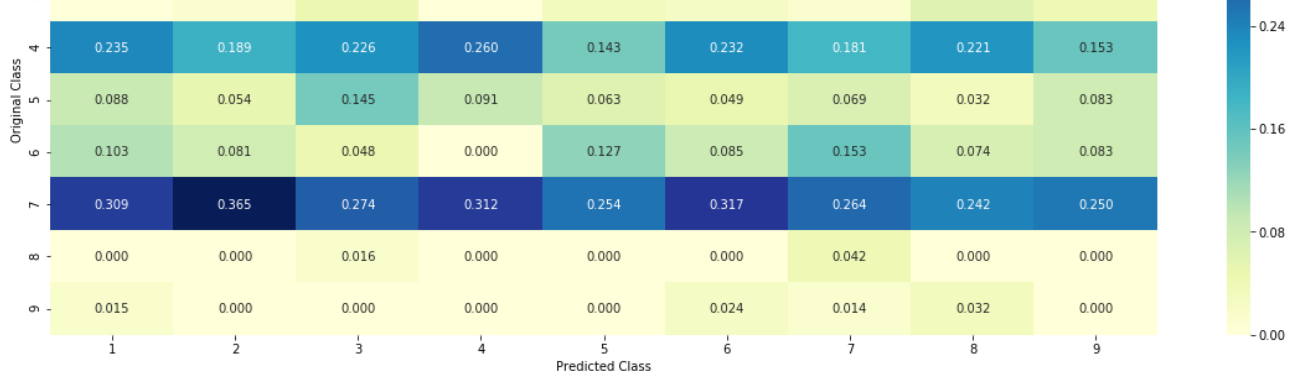
In [0]:
```
# we need to generate 9 numbers and the sum of numbers should be 1
```

```
# one solution is to genarate 9 numbers and divide each of the numbers by thei
r sum
# ref: https://stackoverflow.com/a/18662466/4084039
test_data_len = test_df.shape[0]
cv_data_len = cv_df.shape[0]

# we create a output array that has exactly same size as the CV data
cv_predicted_y = np.zeros((cv_data_len,9))
for i in range(cv_data_len):
    rand_probs = np.random.rand(1,9)
    cv_predicted_y[i] = ((rand_probs/sum(sum(rand_probs)))[0])
print("Log loss on Cross Validation Data using Random Model",log_loss(y_cv,cv_
predicted_y, eps=1e-15))


# Test-Set error.
#we create a output array that has exactly same as the test data
test_predicted_y = np.zeros((test_data_len,9))
for i in range(test_data_len):
    rand_probs = np.random.rand(1,9)
    test_predicted_y[i] = ((rand_probs/sum(sum(rand_probs)))[0])
print("Log loss on Test Data using Random Model",log_loss(y_test,test_predicte
d_y, eps=1e-15))

predicted_y =np.argmax(test_predicted_y, axis=1)
plot_confusion_matrix(y_test, predicted_y+1)
```

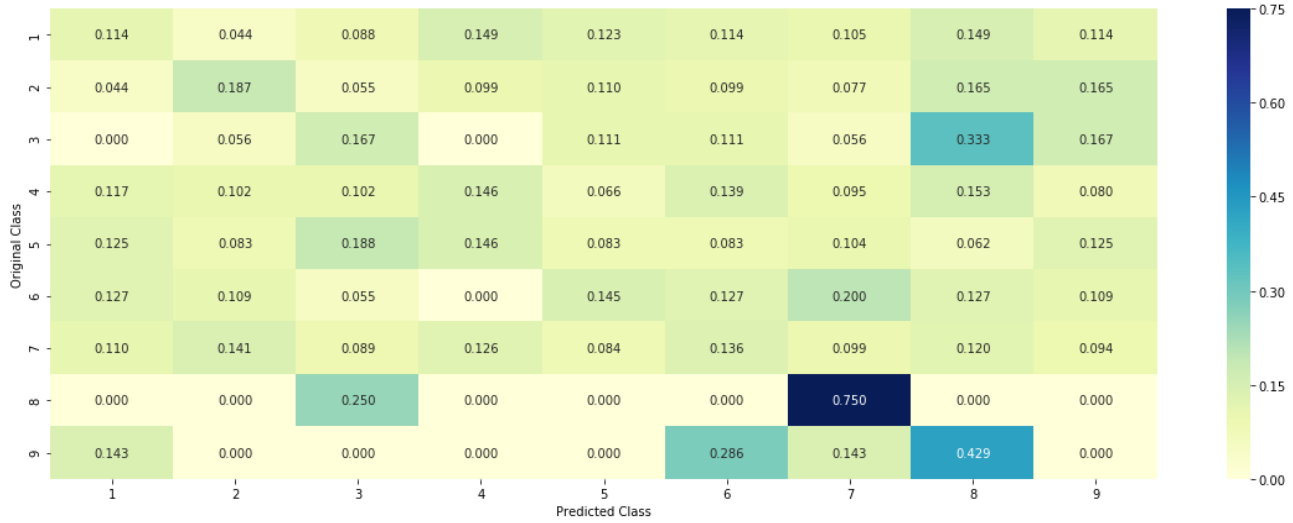
```
Log loss on Cross Validation Data using Random Model 2.536598785706848
Log loss on Test Data using Random Model 2.501572555849742
-------------------- Confusion matrix --------------------
```


```
-------------------- Precision matrix (Columm Sum=1) --------------------
```






```
-------------------- Recall matrix (Row sum=1) --------------------
```



3.3 Univariate Analysis

In [14]:

```
# code for response coding with Laplace smoothing.
# alpha : used for laplace smoothing
# feature: ['gene', 'variation']
# df: ['train_df', 'test_df', 'cv_df']
# algorithm
# ----------
# Consider all unique values and the number of occurances of given feature in
 train data dataframe
# build a vector (1*9) , the first element = (number of times it occured in cl
ass1 + 10*alpha / number of time it occurred in total data+90*alpha)
# gv_dict is like a look up table, for every gene it store a (1*9) representat
ion of it
# for a value of feature in df:
# if it is in train data:
# we add the vector that was stored in 'gv_dict' look up table to 'gv_fea'
# if it is not there is train:
# we add [1/9, 1/9, 1/9, 1/9,1/9, 1/9, 1/9, 1/9, 1/9] to 'gv_fea'
# return 'gv_fea'
# ----------------------

# get_gv_fea_dict: Get Gene varaition Feature Dict
def get_gv_fea_dict(alpha, feature, df):
    # value_count: it contains a dict like
    # print(train_df['Gene'].value_counts())
    # output:
    #        {BRCA1      174
```

```
    #         TP53       106
    #         EGFR        86
    #         BRCA2       75
    #         PTEN        69
    #         KIT         61
    #         BRAF        60
    #         ERBB2       47
    #         PDGFRA      46
    #         ...}
    # print(train_df['Variation'].value_counts())
    # output:
    # {
    # Truncating_Mutations                     63
    # Deletion                                 43
    # Amplification                            43
    # Fusions                                  22
    # Overexpression                            3
    # E17K                                      3
    # Q61L                                      3
    # S222D                                     2
    # P130S                                     2
    # ...
    # }
    value_count = train_df[feature].value_counts()

    # gv_dict : Gene Variation Dict, which contains the probability array for
 each gene/variation
    gv_dict = dict()

    # denominator will contain the number of time that particular feature occu
red in whole data
    for i, denominator in value_count.items():
        # vec will contain (p(yi==1/Gi) probability of gene/variation belongs
 to perticular class
        # vec is 9 diamensional vector
        vec = []
        for k in range(1,10):
            # print(train_df.loc[(train_df['Class']==1) & (train_df['Gene']=
='BRCA1')])
            #         ID   Gene             Variation  Class
            # 2470  2470  BRCA1               S1715C      1
            # 2486  2486  BRCA1               S1841R      1
            # 2614  2614  BRCA1                  M1R      1
            # 2432  2432  BRCA1               L1657P      1
            # 2567  2567  BRCA1               T1685A      1
            # 2583  2583  BRCA1               E1660G      1
            # 2634  2634  BRCA1               W1718L      1
            # cls_cnt.shape[0] will return the number of rows

            cls_cnt = train_df.loc[(train_df['Class']==k) & (train_df[feature]
==i)]

            # cls_cnt.shape[0](numerator) will contain the number of time that
particular feature occured in whole data
            vec.append((cls_cnt.shape[0] + alpha*10)/ (denominator + 90*alpha
))

        # we are adding the gene/variation to the dict as key and vec as value
        gv_dict[i]=vec
    return gv_dict

# Get Gene variation feature
```

```
def get_gv_feature(alpha, feature, df):
    # print(gv_dict)
    #     {'BRCA1': [0.20075757575757575, 0.03787878787878788, 0.0681818181818
18177, 0.13636363636363635, 0.25, 0.19318181818181818, 0.03787878787878788, 0.
03787878787878788, 0.03787878787878788],
    #      'TP53': [0.32142857142857145, 0.061224489795918366, 0.0612244897959
18366, 0.27040816326530615, 0.061224489795918366, 0.066326530612244902, 0.0510
20408163265307, 0.051020408163265307, 0.056122448979591837],
    #      'EGFR': [0.056818181818181816, 0.21590909090909091, 0.0625, 0.06818
1818181818177, 0.068181818181818177, 0.0625, 0.34659090909090912, 0.0625, 0.05
6818181818181816],
    #      'BRCA2': [0.13333333333333333, 0.060606060606060608, 0.060606060606
060608, 0.078787878787878782, 0.1393939393939394, 0.34545454545454546, 0.06060
6060606060608, 0.060606060606060608, 0.060606060606060608],
    #      'PTEN': [0.069182389937106917, 0.062893081761006289, 0.069182389937
106917, 0.46540880503144655, 0.075471698113207544, 0.062893081761006289, 0.069
182389937106917, 0.062893081761006289, 0.062893081761006289],
    #      'KIT': [0.066225165562913912, 0.25165562913907286, 0.07284768211920
5295, 0.072847682119205295, 0.066225165562913912, 0.066225165562913912, 0.2715
2317880794702, 0.066225165562913912, 0.066225165562913912],
    #      'BRAF': [0.066666666666666666, 0.17999999999999999, 0.0733333333333
33334, 0.073333333333333334, 0.093333333333333338, 0.080000000000000002, 0.299
99999999999999, 0.066666666666666666, 0.066666666666666666],
    #      ...
    #     }
    gv_dict = get_gv_fea_dict(alpha, feature, df)
    # value_count is similar in get_gv_fea_dict
    value_count = train_df[feature].value_counts()

    # gv_fea: Gene_variation feature, it will contain the feature for each fea
ture value in the data
    gv_fea = []
    # for every feature values in the given data frame we will check if it is
 there in the train data then we will add the feature to gv_fea
    # if not we will add [1/9,1/9,1/9,1/9,1/9,1/9,1/9,1/9,1/9] to gv_fea
    for index, row in df.iterrows():
        if row[feature] in dict(value_count).keys():
            gv_fea.append(gv_dict[row[feature]])
        else:
            gv_fea.append([1/9,1/9,1/9,1/9,1/9,1/9,1/9,1/9,1/9])
#             gv_fea.append([-1,-1,-1,-1,-1,-1,-1,-1,-1])
    return gv_fea
```

when we caculate the probability of a feature belongs to any particular class, we apply laplace smoothing

- (numerator + 10*alpha) / (denominator + 90*alpha)

3.2.1 Univariate Analysis on Gene Feature

Q1. Gene, What type of feature it is ?

Ans. Gene is a categorical variable

Q2. How many categories are there and How they are distributed?

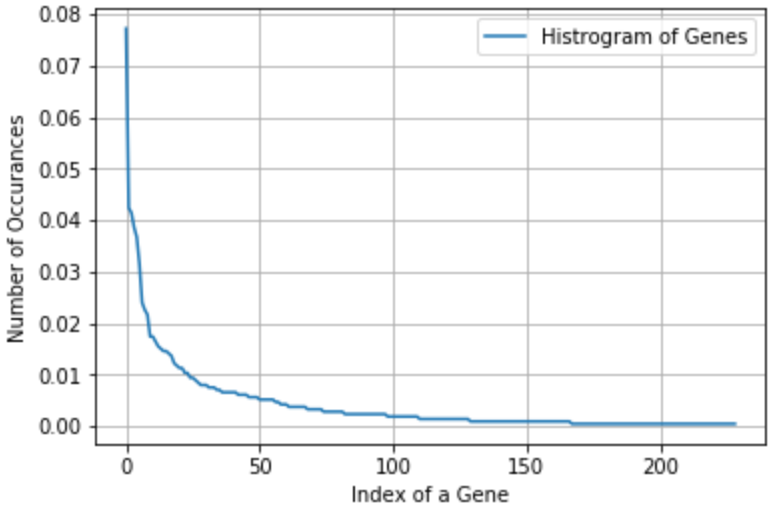
In [31]:
```
unique_genes = train_df['Gene'].value_counts()
print('Number of Unique Genes :', unique_genes.shape[0])
# the top 10 genes that occured most
print(unique_genes.head(10))
```

```
Number of Unique Genes : 238
BRCA1     163
EGFR      103
TP53      102
PTEN       81
BRCA2      79
KIT        68
BRAF       53
ALK        51
ERBB2      38
PDGFRA     38
Name: Gene, dtype: int64
```

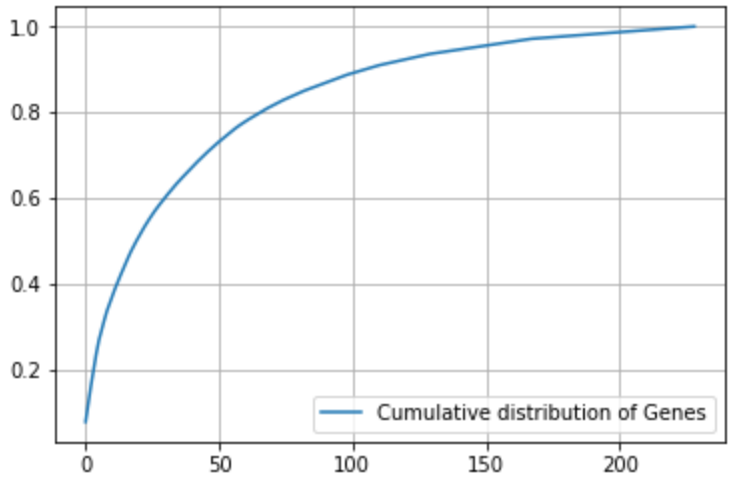
In [49]:
```
print("Ans: There are", unique_genes.shape[0] ,"different categories of genes
 in the train data, and they are distibuted as follows",)
```

```
Ans: There are 238 different categories of genes in the train data, and they
are distibuted as follows
```

In [0]:
```
s = sum(unique_genes.values);
h = unique_genes.values/s;
plt.plot(h, label="Histrogram of Genes")
plt.xlabel('Index of a Gene')
plt.ylabel('Number of Occurances')
plt.legend()
plt.grid()
plt.show()
```



In [0]:
```
c = np.cumsum(h)
plt.plot(c,label='Cumulative distribution of Genes')
plt.grid()
plt.legend()
plt.show()
```



Q3. How to featurize this Gene feature ?

Ans.there are two ways we can featurize this variable check out this video: https://www.appliedaicourse.com/course/applied-ai-course-online/lessons/handling-categorical-and-numerical-features/

1. One hot Encoding
2. Response coding

We will choose the appropriate featurization based on the ML model we use. For this problem of multi-class classification with categorical features, one-hot encoding is better for Logistic regression while response coding is better for Random Forests.

In [74]:
```
#response-coding of the Gene feature
# alpha is used for laplace smoothing
alpha = 1
# train gene feature
train_gene_feature_responseCoding = np.array(get_gv_feature(alpha, "Gene", tra
in_df))
# test gene feature
test_gene_feature_responseCoding = np.array(get_gv_feature(alpha, "Gene", test
_df))
# cross validation gene feature
cv_gene_feature_responseCoding = np.array(get_gv_feature(alpha, "Gene", cv_df
))
```

In [29]:
```
print("train_gene_feature_responseCoding is converted feature using respone co
ding method. The shape of gene feature:", train_gene_feature_responseCoding.sh
ape)
```

```
train_gene_feature_responseCoding is converted feature using respone coding
method. The shape of gene feature: (2124, 9)
```

In [24]:
```
# Onehotencoding of Gene feature.
gene_vectorizer = CountVectorizer()
train_gene_feature_onehotCoding = gene_vectorizer.fit_transform(train_df['Gen
e'])
test_gene_feature_onehotCoding = gene_vectorizer.transform(test_df['Gene'])
cv_gene_feature_onehotCoding = gene_vectorizer.transform(cv_df['Gene'])
```

In [25]:
```
train_df['Gene'].head()
```

Out[25]:
```
2769     BRAF
```

```
2600    BRCA1
2462    BRCA1
1487    FGFR2
2185     PTEN
Name: Gene, dtype: object
```

In [26]:
```
gene_vectorizer.get_feature_names()
```

Out[26]:
```
['abl1',
 'acvr1',
 'ago2',
 'akt1',
 'akt2',
 'akt3',
 'alk',
 'apc',
 'ar',
 'araf',
 'arid1b',
 'arid2',
 'arid5b',
 'asxl1',
 'atm',
 'atr',
 'atrx',
 'aurka',
 'aurkb',
 'axl',
 'b2m',
 'bap1',
 'bard1',
 'bcl10',
 'bcl2l11',
 'bcor',
 'braf',
 'brca1',
 'brca2',
 'brd4',
 'brip1',
 'btk',
 'card11',
 'carm1',
 'casp8',
 'cbl',
 'ccnd1',
 'ccnd2',
 'ccnd3',
 'ccne1',
 'cdh1',
 'cdk12',
 'cdk4',
 'cdk6',
 'cdk8',
 'cdkn1a',
 'cdkn1b',
 'cdkn2a',
 'cdkn2b',
 'chek2',
 'cic',
 'crebbp',
 'ctcf',
```

```
'ctla4',
'ctnnb1',
'ddr2',
'dicer1',
'dnmt3a',
'dnmt3b',
'egfr',
'eif1ax',
'elf3',
'ep300',
'epas1',
'epcam',
'erbb2',
'erbb3',
'erbb4',
'ercc2',
'ercc4',
'erg',
'errfi1',
'esr1',
'etv1',
'etv6',
'ewsr1',
'ezh2',
'fam58a',
'fanca',
'fat1',
'fbxw7',
'fgf3',
'fgf4',
'fgfr1',
'fgfr2',
'fgfr3',
'fgfr4',
'flt1',
'flt3',
'foxa1',
'foxl2',
'foxo1',
'foxp1',
'gata3',
'gli1',
'gnaq',
'gnas',
'h3f3a',
'hist1h1c',
'hla',
'hnf1a',
'hras',
'idh1',
'idh2',
'igf1r',
'ikzf1',
'jak1',
'jak2',
'jun',
'kdm5a',
'kdm5c',
'kdm6a',
'kdr',
'keap1',
'kit',
```

```
'klf4',
'kmt2a',
'kmt2b',
'kmt2c',
'kmt2d',
'knstrn',
'kras',
'lats1',
'lats2',
'map2k1',
'map2k2',
'map2k4',
'map3k1',
'mapk1',
'med12',
'mef2b',
'men1',
'met',
'mga',
'mlh1',
'mpl',
'msh2',
'msh6',
'mtor',
'myc',
'mycn',
'myd88',
'myod1',
'ncor1',
'nf1',
'nf2',
'nfe2l2',
'nfkbia',
'nkx2',
'notch1',
'notch2',
'npm1',
'nras',
'nsd1',
'ntrk1',
'ntrk2',
'ntrk3',
'nup93',
'pak1',
'pbrm1',
'pdgfra',
'pdgfrb',
'pik3ca',
'pik3cb',
'pik3cd',
'pik3r1',
'pik3r2',
'pik3r3',
'pim1',
'pms2',
'pole',
'ppp2r1a',
'ppp6c',
'prdm1',
'ptch1',
'pten',
'ptpn11',
```

```
'ptprd',
'ptprt',
'rab35',
'rac1',
'rad21',
'rad50',
'rad51b',
'rad51c',
'rad51d',
'rad54l',
'raf1',
'rara',
'rasa1',
'rb1',
'ret',
'rheb',
'rhoa',
'rit1',
'ros1',
'rras2',
'runx1',
'rxra',
'rybp',
'setd2',
'sf3b1',
'shoc2',
'shq1',
'smad2',
'smad3',
'smad4',
'smarca4',
'smarcb1',
'smo',
'sos1',
'sox9',
'spop',
'src',
'srsf2',
'stag2',
'stat3',
'stk11',
'tcf3',
'tcf7l2',
'tert',
'tet1',
'tet2',
'tgfbr1',
'tgfbr2',
'tmprss2',
'tp53',
'tp53bp1',
'tsc1',
'tsc2',
'u2af1',
'vegfa',
'vhl',
'whsc1l1',
'xpo1',
'xrcc2',
'yap1']
```

In [27]:
```
print("train_gene_feature_onehotCoding is converted feature using one-hot enco
ding method. The shape of gene feature:", train_gene_feature_onehotCoding.shap
e)
```

```
train_gene_feature_onehotCoding is converted feature using one-hot encoding
method. The shape of gene feature: (2124, 237)
```

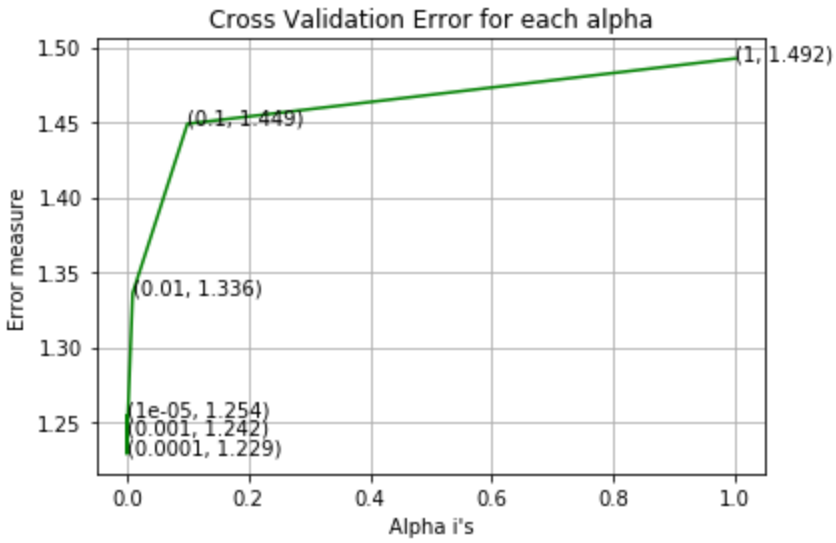
Q4. How good is this gene feature in predicting y_i?

There are many ways to estimate how good a feature is, in predicting y_i. One of the good methods is to build a proper ML model using just this feature. In this case, we will build a logistic regression model using only Gene feature (one hot encoded) to predict y_i.

In [29]:
```
alpha = [10 ** x for x in range(-5, 1)] # hyperparam for SGD classifier.


cv_log_error_array=[]
for i in alpha:
    clf = SGDClassifier(alpha=i, penalty='l2', loss='log', random_state=42)
    clf.fit(train_gene_feature_onehotCoding, y_train)
    sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
    sig_clf.fit(train_gene_feature_onehotCoding, y_train)
    predict_y = sig_clf.predict_proba(cv_gene_feature_onehotCoding)
    cv_log_error_array.append(log_loss(y_cv, predict_y, labels=clf.classes_, e
ps=1e-15))
    print('For values of alpha = ', i, "The log loss is:",log_loss(y_cv, predi
ct_y, labels=clf.classes_, eps=1e-15))

fig, ax = plt.subplots()
ax.plot(alpha, cv_log_error_array,c='g')
for i, txt in enumerate(np.round(cv_log_error_array,3)):
    ax.annotate((alpha[i],np.round(txt,3)), (alpha[i],cv_log_error_array[i]))
plt.grid()
plt.title("Cross Validation Error for each alpha")
plt.xlabel("Alpha i's")
plt.ylabel("Error measure")
plt.show()


best_alpha = np.argmin(cv_log_error_array)
clf = SGDClassifier(alpha=alpha[best_alpha], penalty='l2', loss='log', random_
state=42)
clf.fit(train_gene_feature_onehotCoding, y_train)
sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
sig_clf.fit(train_gene_feature_onehotCoding, y_train)

predict_y = sig_clf.predict_proba(train_gene_feature_onehotCoding)
print('For values of best alpha = ', alpha[best_alpha], "The train log loss i
s:",log_loss(y_train, predict_y, labels=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(cv_gene_feature_onehotCoding)
print('For values of best alpha = ', alpha[best_alpha], "The cross validation
 log loss is:",log_loss(y_cv, predict_y, labels=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(test_gene_feature_onehotCoding)
print('For values of best alpha = ', alpha[best_alpha], "The test log loss i
s:",log_loss(y_test, predict_y, labels=clf.classes_, eps=1e-15))
```

```
For values of alpha =  1e-05 The log loss is: 1.2540316787614125
For values of alpha =  0.0001 The log loss is: 1.2287339616034
For values of alpha =  0.001 The log loss is: 1.242053039388555
For values of alpha =  0.01 The log loss is: 1.3359713400934339
```

```
For values of alpha =  0.1 The log loss is: 1.4489347118298848
For values of alpha =  1 The log loss is: 1.492493210833703
```




```
For values of best alpha =  0.0001 The train log loss is: 0.9918925688762129
For values of best alpha =  0.0001 The cross validation log loss is: 1.22873
39616034
For values of best alpha =  0.0001 The test log loss is: 1.1558085530692999
```

Q5. Is the Gene feature stable across all the data sets (Test, Train, Cross validation)?

Ans. Yes, it is. Otherwise, the CV and Test errors would be significantly more than train error.

In [32]:
```
print("Q6. How many data points in Test and CV datasets are covered by the ",
unique_genes.shape[0], " genes in train dataset?")

test_coverage=test_df[test_df['Gene'].isin(list(set(train_df['Gene'])))].shape
[0]
cv_coverage=cv_df[cv_df['Gene'].isin(list(set(train_df['Gene'])))].shape[0]

print('Ans\n1. In test data',test_coverage, 'out of',test_df.shape[0], ":",(te
st_coverage/test_df.shape[0])*100)
print('2. In cross validation data',cv_coverage, 'out of ',cv_df.shape[0],":"
,(cv_coverage/cv_df.shape[0])*100)
```

```
Q6. How many data points in Test and CV datasets are covered by the  238  ge
nes in train dataset?
Ans
1. In test data 652 out of 665 : 98.04511278195488
2. In cross validation data 516 out of  532 : 96.99248120300751
```

3.2.2 Univariate Analysis on Variation Feature

Q7. Variation, What type of feature is it ?

Ans. Variation is a categorical variable

Q8. How many categories are there?

In [33]:
```
unique_variations = train_df['Variation'].value_counts()
print('Number of Unique Variations :', unique_variations.shape[0])
```

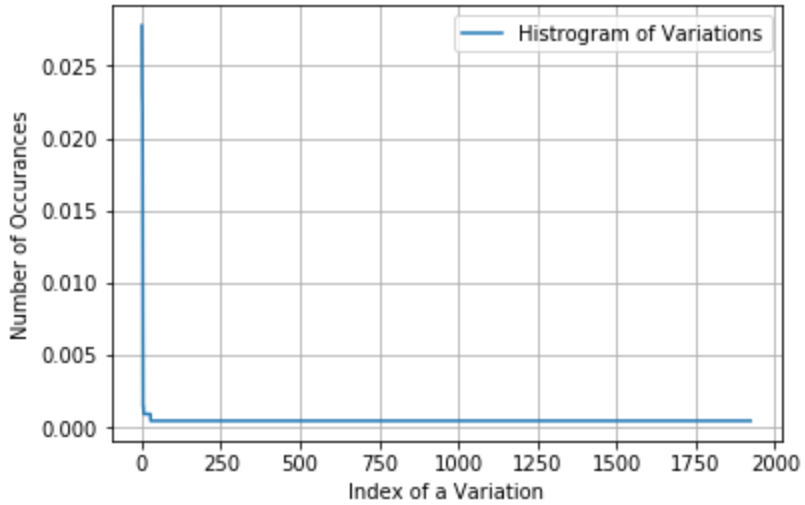
```
# the top 10 variations that occured most
print(unique_variations.head(10))
```

```
Number of Unique Variations : 1928
Deletion                  54
Truncating_Mutations      53
Amplification             47
Fusions                   22
Overexpression             4
Q61L                       3
S308A                      2
R170W                      2
E330K                      2
EWSR1-ETV1_Fusion          2
Name: Variation, dtype: int64
```

In [0]:
```
print("Ans: There are", unique_variations.shape[0] ,"different categories of v
ariations in the train data, and they are distibuted as follows",)
```

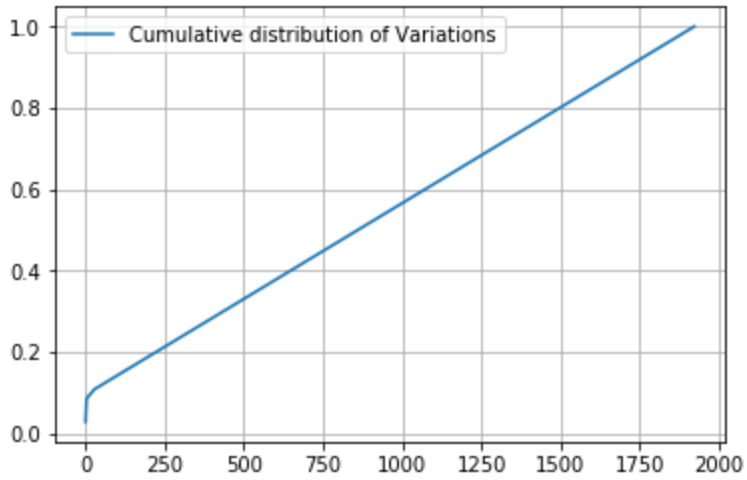
```
Ans: There are 1924 different categories of variations in the train data, an
d they are distibuted as follows
```

In [0]:
```
s = sum(unique_variations.values);
h = unique_variations.values/s;
plt.plot(h, label="Histrogram of Variations")
plt.xlabel('Index of a Variation')
plt.ylabel('Number of Occurances')
plt.legend()
plt.grid()
plt.show()
```



In [0]:
```
c = np.cumsum(h)
print(c)
plt.plot(c,label='Cumulative distribution of Variations')
plt.grid()
plt.legend()
plt.show()
```

```
[0.02777778 0.05084746 0.07297552 ... 0.99905838 0.99952919 1.        ]
```



Q9. How to featurize this Variation feature ?

Ans. There are two ways we can featurize this variable check out this video: https://www.appliedaicourse.com/course/applied-ai-course-online/lessons/handling-categorical-and-numerical-features/

1. One hot Encoding
2. Response coding

We will be using both these methods to featurize the Variation Feature

In [34]:
```
# alpha is used for laplace smoothing
alpha = 1
# train gene feature
train_variation_feature_responseCoding = np.array(get_gv_feature(alpha, "Variation", train_df))
# test gene feature
test_variation_feature_responseCoding = np.array(get_gv_feature(alpha, "Variation", test_df))
# cross validation gene feature
cv_variation_feature_responseCoding = np.array(get_gv_feature(alpha, "Variation", cv_df))
```

In [35]:
```
print("train_variation_feature_responseCoding is a converted feature using the response coding method. The shape of Variation feature:", train_variation_feature_responseCoding.shape)
```

```
train_variation_feature_responseCoding is a converted feature using the response coding method. The shape of Variation feature: (2124, 9)
```

In [40]:
```
# onehotCoding of variation feature.
variation_vectorizer = CountVectorizer()
train_variation_feature_onehotCoding = variation_vectorizer.fit_transform(train_df['Variation'])
test_variation_feature_onehotCoding = variation_vectorizer.transform(test_df['Variation'])
cv_variation_feature_onehotCoding = variation_vectorizer.transform(cv_df['Variation'])
```

In [41]:
```
print("train_variation_feature_onehotEncoded is converted feature using the onehotcoding method. The shape of Variation feature:", train_variation_feature_onehotCoding.shape)
```

```
train_variation_feature_onehotEncoded is converted feature using the onehotc
```

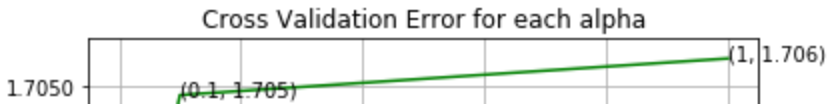
```
oding method. The shape of Variation feature: (2124, 1959)
```

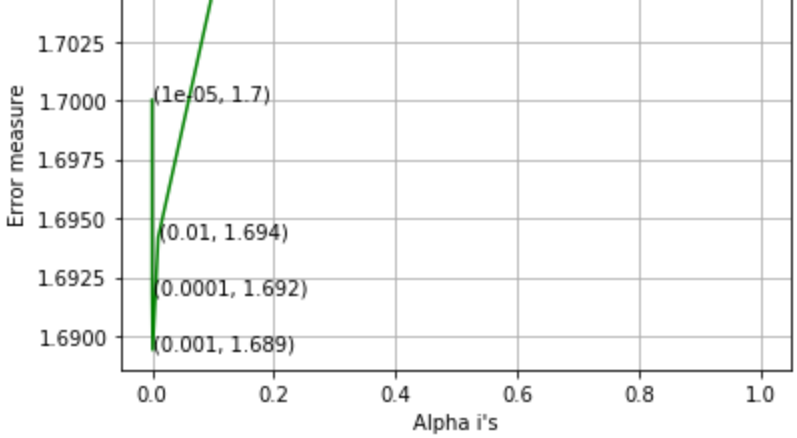
Q10. How good is this Variation feature in predicting y_i?

Let's build a model just like the earlier!

In [42]:
```
alpha = [10 ** x for x in range(-5, 1)]

cv_log_error_array=[]
for i in alpha:
    clf = SGDClassifier(alpha=i, penalty='l2', loss='log', random_state=42)
    clf.fit(train_variation_feature_onehotCoding, y_train)

    sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
    sig_clf.fit(train_variation_feature_onehotCoding, y_train)
    predict_y = sig_clf.predict_proba(cv_variation_feature_onehotCoding)

    cv_log_error_array.append(log_loss(y_cv, predict_y, labels=clf.classes_, e
ps=1e-15))
    print('For values of alpha = ', i, "The log loss is:",log_loss(y_cv, predi
ct_y, labels=clf.classes_, eps=1e-15))

fig, ax = plt.subplots()
ax.plot(alpha, cv_log_error_array,c='g')
for i, txt in enumerate(np.round(cv_log_error_array,3)):
    ax.annotate((alpha[i],np.round(txt,3)), (alpha[i],cv_log_error_array[i]))
plt.grid()
plt.title("Cross Validation Error for each alpha")
plt.xlabel("Alpha i's")
plt.ylabel("Error measure")
plt.show()


best_alpha = np.argmin(cv_log_error_array)
clf = SGDClassifier(alpha=alpha[best_alpha], penalty='l2', loss='log', random_
state=42)
clf.fit(train_variation_feature_onehotCoding, y_train)
sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
sig_clf.fit(train_variation_feature_onehotCoding, y_train)

predict_y = sig_clf.predict_proba(train_variation_feature_onehotCoding)
print('For values of best alpha = ', alpha[best_alpha], "The train log loss i
s:",log_loss(y_train, predict_y, labels=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(cv_variation_feature_onehotCoding)
print('For values of best alpha = ', alpha[best_alpha], "The cross validation
 log loss is:",log_loss(y_cv, predict_y, labels=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(test_variation_feature_onehotCoding)
print('For values of best alpha = ', alpha[best_alpha], "The test log loss i
s:",log_loss(y_test, predict_y, labels=clf.classes_, eps=1e-15))
```

```
For values of alpha =  1e-05 The log loss is: 1.7000133005735867
For values of alpha =  0.0001 The log loss is: 1.6918603581254255
For values of alpha =  0.001 The log loss is: 1.6894234451303654
For values of alpha =  0.01 The log loss is: 1.6941842225458308
For values of alpha =  0.1 The log loss is: 1.7046313189693698
For values of alpha =  1 The log loss is: 1.7061806021084311
```






```
For values of best alpha =  0.001 The train log loss is: 1.0842380718332265
For values of best alpha =  0.001 The cross validation log loss is: 1.689423
4451303654
For values of best alpha =  0.001 The test log loss is: 1.711430663226359
```

Q11. Is the Variation feature stable across all the data sets (Test, Train, Cross validation)?

Ans. Not sure! But lets be very sure using the below analysis.

In [43]:
```
print("Q12. How many data points are covered by total ", unique_variations.sha
pe[0], " genes in test and cross validation data sets?")
test_coverage=test_df[test_df['Variation'].isin(list(set(train_df['Variation'
]))))].shape[0]
cv_coverage=cv_df[cv_df['Variation'].isin(list(set(train_df['Variation'])))].s
hape[0]
print('Ans\n1. In test data',test_coverage, 'out of',test_df.shape[0], ":",(te
st_coverage/test_df.shape[0])*100)
print('2. In cross validation data',cv_coverage, 'out of ',cv_df.shape[0],":"
,(cv_coverage/cv_df.shape[0])*100)
```

```
Q12. How many data points are covered by total  1928  genes in test and cros
s validation data sets?
Ans
1. In test data 73 out of 665 : 10.977443609022556
2. In cross validation data 53 out of  532 : 9.962406015037594
```

Assignments

1. Apply All the models with tf-idf features (Replace CountVectorizer with tfidfVectorizer and run the same cells)
2. Instead of using all the words in the dataset, use only the top 1000 words based of tf-idf values
3. Apply Logistic regression with CountVectorizer Features, including both unigrams and bigrams
4. Try any of the feature engineering techniques discussed in the course to reduce the CV and test log-loss to a value less than 1.0

1. Apply All the models with tf-idf features (Replace CountVectorizer with tfidfVectorizer and run the same cells)

2. Instead of using all the words in the dataset, use only the top 1000 words based of tf-idf values

3.2.3 Univariate Analysis on Text Feature

1. How many unique words are present in train data?
2. How are word frequencies distributed?
3. How to featurize text field?
4. Is the text feature useful in predicitng y_i?
5. Is the text feature stable across train, test and CV datasets?

In [44]:
```
# cls_text is a data frame
# for every row in data fram consider the 'TEXT'
# split the words by space
# make a dict with those words
# increment its count whenever we see that word

def extract_dictionary_paddle(cls_text):
    dictionary = defaultdict(int)
    for index, row in cls_text.iterrows():
        for word in row['TEXT'].split():
            dictionary[word] +=1
    return dictionary
```

In [45]:
```
import math
#https://stackoverflow.com/a/1602964
def get_text_responsecoding(df):
    text_feature_responseCoding = np.zeros((df.shape[0],9))
    for i in range(0,9):
        row_index = 0
        for index, row in df.iterrows():
            sum_prob = 0
            for word in row['TEXT'].split():
                sum_prob += math.log(((dict_list[i].get(word,0)+10 )/(total_di
ct.get(word,0)+90)))
            text_feature_responseCoding[row_index][i] = math.exp(sum_prob/len(
row['TEXT'].split()))
            row_index += 1
    return text_feature_responseCoding
```

Using Tfidf Encoding for 'Text' Feature with only top 1000 feature

In [46]:
```
# building a CountVectorizer with all the words that occured minimum 3 times i
n train data
text_vectorizer = TfidfVectorizer(min_df=3,max_features=1000)
train_text_feature_tfidf = text_vectorizer.fit_transform(train_df['TEXT'])
# getting all the feature names (words)
train_text_features= text_vectorizer.get_feature_names()

# train_text_feature_onehotCoding.sum(axis=0).A1 will sum every row and return
s (1*number of features) vector
train_text_fea_counts = train_text_feature_tfidf.sum(axis=0).A1

# zip(list(text_features),text_fea_counts) will zip a word with its number of
 times it occured
text_fea_dict = dict(zip(list(train_text_features),train_text_fea_counts))
```

```
print("Total number of unique words in train data :", len(train_text_features
))
```

```
Total number of unique words in train data : 1000
```

In [61]:
```
dict_list = []
# dict_list =[] contains 9 dictoinaries each corresponds to a class
for i in range(1,10):
    cls_text = train_df[train_df['Class']==i]
    # build a word dict based on the words in that class
    dict_list.append(extract_dictionary_paddle(cls_text))
    # append it to dict_list

# dict_list[i] is build on i'th  class text data
# total_dict is buid on whole training text data
total_dict = extract_dictionary_paddle(train_df)


confuse_array = []
for i in train_text_features:
    ratios = []
    max_val = -1
    for j in range(0,9):
        ratios.append((dict_list[j][i]+10 )/(total_dict[i]+90))
    confuse_array.append(ratios)
confuse_array = np.array(confuse_array)
```

In [62]:
```
#response coding of text features
train_text_feature_responseCoding  = get_text_responsecoding(train_df)
test_text_feature_responseCoding  = get_text_responsecoding(test_df)
cv_text_feature_responseCoding  = get_text_responsecoding(cv_df)
```

In [72]:
```
# https://stackoverflow.com/a/16202486
# we convert each row values such that they sum to 1
train_text_feature_responseCoding = (train_text_feature_responseCoding.T/train
_text_feature_responseCoding.sum(axis=1)).T
test_text_feature_responseCoding = (test_text_feature_responseCoding.T/test_te
xt_feature_responseCoding.sum(axis=1)).T
cv_text_feature_responseCoding = (cv_text_feature_responseCoding.T/cv_text_fea
ture_responseCoding.sum(axis=1)).T
```

In [64]:
```
# don't forget to normalize every feature
train_text_feature_tfidf = normalize(train_text_feature_tfidf, axis=0)

# we use the same vectorizer that was trained on train data
test_text_feature_tfidf = text_vectorizer.transform(test_df['TEXT'])
# don't forget to normalize every feature
test_text_feature_tfidf = normalize(test_text_feature_tfidf, axis=0)

# we use the same vectorizer that was trained on train data
cv_text_feature_tfidf = text_vectorizer.transform(cv_df['TEXT'])
# don't forget to normalize every feature
cv_text_feature_tfidf = normalize(cv_text_feature_tfidf, axis=0)
```

In [65]:
```
#https://stackoverflow.com/a/2258273/4084039
sorted_text_fea_dict = dict(sorted(text_fea_dict.items(), key=lambda x: x[1] ,
reverse=True))
sorted_text_occur = np.array(list(sorted_text_fea_dict.values()))
```

In [66]:
```
# Number of words for a given frequency.
print(Counter(sorted_text_occur))
```

```
Counter({252.24558113347163: 1, 179.50253571968764: 1, 131.60652683247108:
1, 129.32613044475715: 1, 129.041499869281: 1, 121.77471219056166: 1, 121.41
717002411431: 1, 116.67933955594809: 1, 110.88926875087527: 1, 105.808383510
11354: 1, 105.51796648976011: 1, 92.32679292307229: 1, 88.07434994225082: 1,
87.8984301253517: 1, 84.36109644547733: 1, 81.23955127663716: 1, 80.60883129
634384: 1, 80.3762860110055: 1, 79.5318126080093: 1, 78.03079386895551: 1, 7
5.77099680143053: 1, 73.53408326596441: 1, 72.51894693546737: 1, 70.95252420
482961: 1, 68.38117715231388: 1, 66.18323706027483: 1, 65.51415106214614: 1,
65.1238324367345: 1, 64.62019942196817: 1, 64.41681391150941: 1, 63.96865382
295219: 1, 62.08170119657515: 1, 61.19042253379302: 1, 59.81210383497463: 1,
58.83249041541081: 1, 57.51655807231967: 1, 56.648708277473325: 1, 56.343246
99650718: 1, 55.4439708363273: 1, 53.31172523722199: 1, 51.40572834611365:
1, 49.73405807726486: 1, 49.2447114494261: 1, 48.60793947161777: 1, 48.19370
8036228756: 1, 46.519424886615276: 1, 46.51681178597405: 1, 46.0099487561817
36: 1, 45.633730011638924: 1, 44.934708653762065: 1, 44.64909895488913: 1, 4
3.576305376906504: 1, 43.2773505008485: 1, 43.26500695586847: 1, 42.71348359
4308435: 1, 42.55296331850166: 1, 42.31433057948067: 1, 42.03133716368801:
1, 41.83012692744308: 1, 41.47212261761407: 1, 41.35033529361198: 1, 41.2408
167595891: 1, 40.904535382309724: 1, 40.33069553361391: 1, 40.2945408171514
6: 1, 39.955999214703475: 1, 39.880145848858085: 1, 39.58845117853332: 1, 3
9.25670175703626: 1, 39.22150141083673: 1, 39.173689208398365: 1, 39.0485091
93999436: 1, 38.74220525256794: 1, 37.81713578436664: 1, 37.8107909621742:
1, 37.291932280885796: 1, 36.91366562321106: 1, 36.80996927813149: 1, 36.419
40971659223: 1, 35.984156603204134: 1, 35.74145362540688: 1, 35.636149594831
16: 1, 35.20802536644373: 1, 35.18358967562891: 1, 34.88067692503665: 1, 34.
76515713095325: 1, 34.68572056078687: 1, 34.534411219277914: 1, 34.092748130
03534: 1, 33.89728669865751: 1, 33.78607020631177: 1, 33.47503520874564: 1,
32.942381356001626: 1, 32.71997959335806: 1, 32.65373479846036: 1, 32.524850
86181903: 1, 32.3670630628631: 1, 32.280932449742544: 1, 32.19854083408961:
1, 31.845563460102632: 1, 31.68426652630336: 1, 31.634140671428398: 1, 31.58
5762101415305: 1, 31.499348939784177: 1, 31.448239701676027: 1, 31.409161978
924644: 1, 31.390514319641042: 1, 31.163815965137154: 1, 31.122722997430003:
1, 30.977425401009484: 1, 30.880320496760486: 1, 30.831770504762034: 1, 30.6
63333336871432: 1, 30.40355377326771: 1, 30.283966455455246: 1, 30.133337037
325514: 1, 30.119807416265214: 1, 29.81532288013134: 1, 29.766028955397445:
1, 29.75585054485098: 1, 29.440480233265298: 1, 29.421017546049722: 1, 29.24
759289346898: 1, 29.142142285520983: 1, 29.066158979432938: 1, 28.9341294510
9949: 1, 28.853147460858477: 1, 28.85140687458416: 1, 28.806849993944443: 1,
28.655316911103707: 1, 28.614441975330504: 1, 28.53322834341001: 1, 28.46495
2605175878: 1, 27.97727740797792: 1, 27.792400792431952: 1, 27.6694611812526
8: 1, 27.596553939897515: 1, 27.579050725850276: 1, 27.47157022059362: 1, 2
7.369516557871584: 1, 27.099645736583348: 1, 26.968628505974223: 1, 26.90491
26648423: 1, 26.797299815535435: 1, 26.668915370525493: 1, 26.63723833095422
2: 1, 26.287404455157724: 1, 26.186744754113608: 1, 25.657539337310943: 1, 2
5.574911417371755: 1, 25.54590616094108: 1, 25.3255158510789: 1, 25.31351629
140165: 1, 25.264732590485412: 1, 25.185169158715027: 1, 25.172132765363365:
1, 25.07055803637308: 1, 25.027688744971062: 1, 25.014055131432535: 1, 24.85
4420410040152: 1, 24.753743515788557: 1, 24.730901104076814: 1, 24.695496774
340047: 1, 24.63438958698489: 1, 24.534225603931063: 1, 24.45341009513999:
1, 24.30106482722985: 1, 24.29613578366838: 1, 24.24122140938613: 1, 24.1187
5811356399: 1, 24.064264647505446: 1, 23.990437090753254: 1, 23.793451129568
673: 1, 23.78328094414504: 1, 23.751296910602367: 1, 23.47584580986328: 1, 2
3.33198896996858: 1, 23.260511958756666: 1, 23.196864759275822: 1, 23.162787
340498824: 1, 23.09785012594055: 1, 23.025962233622916: 1, 22.9912616689513
6: 1, 22.975889571843005: 1, 22.963902729104404: 1, 22.873854168338365: 1, 2
2.844848676545915: 1, 22.786997259405958: 1, 22.785656316582426: 1, 22.75337
724172211: 1, 22.677247573062683: 1, 22.62601846861442: 1, 22.5567716758296:
1, 22.52318380234471: 1, 22.49386423897323: 1, 22.46823123770825: 1, 22.3446
```

1112259909: 1, 22.3006047870531: 1, 22.187283192350474: 1, 22.13447556443026
4: 1, 22.0598324029164: 1, 22.056464454398753: 1, 21.990413959501307: 1, 21.
833997745970578: 1, 21.817824469021726: 1, 21.81170504104259: 1, 21.78374157
215888: 1, 21.709830263378716: 1, 21.69056421952413: 1, 21.653859428260382:
1, 21.637018109920746: 1, 21.530742780612197: 1, 21.488947737373145: 1, 21.4
77699138954677: 1, 21.454260528114915: 1, 21.40324801416008: 1, 21.382753710
019582: 1, 21.32814656276291: 1, 21.31514446819115: 1, 21.279083178490342:
1, 21.278431567871216: 1, 21.19059459396275: 1, 21.083728565085053: 1, 21.06
469444481715: 1, 21.0492586110356: 1, 21.045905253202147: 1, 20.998637282485
19: 1, 20.7874723694752: 1, 20.669745644662: 1, 20.643153919887475: 1, 20.62
3361922818262: 1, 20.621738103453513: 1, 20.621687802670344: 1, 20.589809502
675763: 1, 20.545407889557314: 1, 20.49418769798847: 1, 20.469254518284885:
1, 20.437386152802688: 1, 20.344801149755092: 1, 20.33796848825306: 1, 20.28
697931881244: 1, 20.196587874616462: 1, 20.1694811066706: 1, 20.072087610757
038: 1, 20.042905982533224: 1, 19.974049549822357: 1, 19.923390571864832: 1,
19.910321459020825: 1, 19.829127198721476: 1, 19.752738443152023: 1, 19.7439
24107503304: 1, 19.690627665941857: 1, 19.66441207322572: 1, 19.655868006207
11: 1, 19.625429251744006: 1, 19.570348787313165: 1, 19.53034946495481: 1, 1
9.52197654147918: 1, 19.518831693505412: 1, 19.517016552462618: 1, 19.510608
495594973: 1, 19.48409189160275: 1, 19.377096902351823: 1, 19.37476184461753
4: 1, 19.319004340414015: 1, 19.24170336673515: 1, 19.22699372094281: 1, 19.
215318174909733: 1, 19.156020975006903: 1, 19.12288977178748: 1, 19.09691113
1369318: 1, 19.05754643929046: 1, 19.0398693345467: 1, 19.024213638816782:
1, 19.016565259964082: 1, 19.010771728564784: 1, 18.948603500774887: 1, 18.9
16806849187346: 1, 18.841873356997667: 1, 18.742986832120454: 1, 18.68010793
4092355: 1, 18.605526654087893: 1, 18.595635893978844: 1, 18.53871405169660
5: 1, 18.517876942091334: 1, 18.51608988502727: 1, 18.462170457973038: 1, 1
8.45626539668234: 1, 18.420991810006605: 1, 18.416792893619885: 1, 18.389618
652932587: 1, 18.37673201062899: 1, 18.34636816353954: 1, 18.30593582839519
2: 1, 18.224328299932512: 1, 18.204960734637417: 1, 18.20385367058838: 1, 1
8.169853247937457: 1, 18.127836710079198: 1, 18.07282568979856: 1, 18.025955
13203812: 1, 17.96127250182758: 1, 17.875009431334803: 1, 17.86754906207972
5: 1, 17.82648643115842: 1, 17.771530122615474: 1, 17.760825518256535: 1, 1
7.69413648144131: 1, 17.684283355489935: 1, 17.682652263572074: 1, 17.672982
91843512: 1, 17.6663449057669: 1, 17.65438991224089: 1, 17.63050712011063:
1, 17.614808729986855: 1, 17.613059927764514: 1, 17.591292523214374: 1, 17.5
70471786667703: 1, 17.539478485574396: 1, 17.47359088428282: 1, 17.455120956
593603: 1, 17.44614989378608: 1, 17.415821125855818: 1, 17.414475693866436:
1, 17.410250582039723: 1, 17.406691579697892: 1, 17.37956927255081: 1, 17.33
5742346837655: 1, 17.332101448843456: 1, 17.33153819210575: 1, 17.3207999745
03895: 1, 17.285317036028882: 1, 17.282254956429846: 1, 17.27260061383772:
1, 17.236313148765465: 1, 17.225728509134896: 1, 17.209831545186997: 1, 17.2
03360372018025: 1, 17.141816212629134: 1, 17.10694332080226: 1, 17.089303118
398075: 1, 17.071486057802492: 1, 17.05232231753967: 1, 17.036461814164774:
1, 17.00527208059051: 1, 16.993133206678987: 1, 16.980389917598217: 1, 16.94
2417874588003: 1, 16.931300586827447: 1, 16.918250541132924: 1, 16.906167305
90375: 1, 16.85884770383122: 1, 16.836993146572137: 1, 16.661185299049674:
1, 16.649717242134916: 1, 16.618914862356235: 1, 16.61354092165688: 1, 16.52
380066017648: 1, 16.519487426794257: 1, 16.4808117793882: 1, 16.462118473795
748: 1, 16.44439038122967: 1, 16.43122037605204: 1, 16.40290439199319: 1, 1
6.38040930936793: 1, 16.361819625211513: 1, 16.346861167404818: 1, 16.346648
978335317: 1, 16.27890321662599: 1, 16.260414334067654: 1, 16.23074965235330
3: 1, 16.210047798303542: 1, 16.188984956700402: 1, 16.054466154721986: 1, 1
6.028799148609547: 1, 16.024015311482735: 1, 15.985372601707489: 1, 15.94365
238367767: 1, 15.941939436991499: 1, 15.900444135365976: 1, 15.8912390931769
53: 1, 15.87003091209682: 1, 15.832396591365367: 1, 15.808937398807243: 1, 1
5.808163466297461: 1, 15.741724851758415: 1, 15.720385956390714: 1, 15.70819
1042651853: 1, 15.652643028114635: 1, 15.651078900352413: 1, 15.585502255525
155: 1, 15.574628644549014: 1, 15.556180723558155: 1, 15.548345723184621: 1,
15.511733160528797: 1, 15.501692010928798: 1, 15.492211378100144: 1, 15.4839
62446788803: 1, 15.480664415190034: 1, 15.455055459198903: 1, 15.41349892139
056: 1, 15.359018309312289: 1, 15.350866829215265: 1, 15.328477201637783: 1,

15.241108909064561: 1, 15.224063008962958: 1, 15.202040677659361: 1, 15.19355
9593149105: 1, 15.191455462088593: 1, 15.123991675389417: 1, 15.079109343586
795: 1, 15.078520883827391: 1, 15.0514474243044: 1, 15.022825419536824: 1, 1
4.9933318041171: 1, 14.984815121746003: 1, 14.976881931844956: 1, 14.9739411
33523944: 1, 14.947156703363731: 1, 14.935959045552806: 1, 14.89589923323687
2: 1, 14.865412170821484: 1, 14.863005140101984: 1, 14.85306607835619: 1, 1
4.806087213259241: 1, 14.805935028123482: 1, 14.799792740576954: 1, 14.79833
3825762485: 1, 14.785874474137588: 1, 14.774817529093747: 1, 14.742099522611
827: 1, 14.741980692449179: 1, 14.714052295242125: 1, 14.689136301582849: 1,
14.637376850204573: 1, 14.630884900841611: 1, 14.609897390702018: 1, 14.5993
90761722297: 1, 14.599360217796498: 1, 14.569474407828945: 1, 14.54657124126
7692: 1, 14.538660118064989: 1, 14.515946232725824: 1, 14.46081574990699: 1,
14.451735092245567: 1, 14.391402299113519: 1, 14.377489935812925: 1, 14.3686
57237345946: 1, 14.356378128188723: 1, 14.326802270522606: 1, 14.26749283447
6581: 1, 14.231046280942497: 1, 14.224894779058237: 1, 14.220750343303267:
1, 14.190064244714817: 1, 14.18995666780601: 1, 14.169880541415186: 1, 14.16
809278529473: 1, 14.130791483858678: 1, 14.117268877125047: 1, 14.1104982366
26393: 1, 14.110141514072488: 1, 14.08096664007051: 1, 14.072287604716417:
1, 14.060347197419965: 1, 14.050771779128484: 1, 14.011145121806292: 1, 14.0
06550892252301: 1, 14.003892579403702: 1, 13.991188976742077: 1, 13.99097397
503489: 1, 13.98363513380971: 1, 13.967924154542207: 1, 13.910965547931387:
1, 13.88051742641749: 1, 13.880496564386966: 1, 13.841114823847258: 1, 13.81
573121516196: 1, 13.805860172499036: 1, 13.804149883726913: 1, 13.7908402879
03338: 1, 13.73946811652391: 1, 13.684283568857294: 1, 13.65020172232069: 1,
13.556926174978823: 1, 13.50777609063992: 1, 13.476735614326758: 1, 13.45776
8153329454: 1, 13.414817389160403: 1, 13.414565391990152: 1, 13.410616153928
569: 1, 13.379179788959707: 1, 13.374256758092397: 1, 13.299908218171447: 1,
13.295734240456767: 1, 13.294963080537803: 1, 13.230247667717583: 1, 13.2279
7936861032: 1, 13.218305089099117: 1, 13.217036764555223: 1, 13.212530563197
609: 1, 13.197786247734069: 1, 13.1616045588501: 1, 13.144281222889294: 1, 1
3.130206825183942: 1, 13.126634585062854: 1, 13.081962874425168: 1, 13.07813
5250228856: 1, 13.067896125579779: 1, 13.03617396197838: 1, 13.0276076718364
2: 1, 13.020116317257935: 1, 12.978928878234184: 1, 12.970694684357845: 1, 1
2.925327214737564: 1, 12.896359156666096: 1, 12.88784066764076: 1, 12.883639
82987181: 1, 12.83971717984472: 1, 12.830229546023968: 1, 12.80849890867291
8: 1, 12.805704067811783: 1, 12.799609321340727: 1, 12.783900293287658: 1, 1
2.772907789130342: 1, 12.761537325136041: 1, 12.713836837366172: 1, 12.70279
3128544284: 1, 12.700342667007737: 1, 12.694318788571033: 1, 12.686833720373
588: 1, 12.677707952770646: 1, 12.634589654658562: 1, 12.624681412644101: 1,
12.613110326306797: 1, 12.607783946718873: 1, 12.564335287367232: 1, 12.5398
26750722348: 1, 12.534020080421763: 1, 12.50285717698172: 1, 12.491967126181
283: 1, 12.491713711099207: 1, 12.480888822605877: 1, 12.44453028758694: 1,
12.402904550783092: 1, 12.392885311964724: 1, 12.376108317786192: 1, 12.3492
34066153203: 1, 12.33777287812063: 1, 12.333880203621627: 1, 12.294114354839
806: 1, 12.293866152810338: 1, 12.275321802964886: 1, 12.26057912503649: 1,
12.248605829137352: 1, 12.245066225282033: 1, 12.239306588480398: 1, 12.2357
5842155398: 1, 12.235664193616765: 1, 12.23081333755198: 1, 12.2177462654430
9: 1, 12.185229815667489: 1, 12.120878291805889: 1, 12.112122767339422: 1, 1
2.061163176767494: 1, 12.031546551609575: 1, 12.03033282508958: 1, 12.025382
445725022: 1, 12.024989380246522: 1, 11.998770099153212: 1, 11.9947057585667
38: 1, 11.988160689357375: 1, 11.960243297719028: 1, 11.91996894929523: 1, 1
1.884997850433304: 1, 11.856011273934252: 1, 11.847900772563413: 1, 11.84545
7874294075: 1, 11.799150410623499: 1, 11.741679942497445: 1, 11.733470921593
005: 1, 11.701720450685132: 1, 11.663860309220835: 1, 11.648659083278048: 1,
11.64851149156517: 1, 11.640189163917372: 1, 11.638313257395113: 1, 11.62193
5609643796: 1, 11.608123073632267: 1, 11.60744198671405: 1, 11.6034294924593
38: 1, 11.595113103854949: 1, 11.586013124433785: 1, 11.581154205210343: 1,
11.539764820147163: 1, 11.526973869086868: 1, 11.490945359451915: 1, 11.4683
97232597885: 1, 11.45726037887031: 1, 11.435169854419623: 1, 11.429612293015
342: 1, 11.415655883295248: 1, 11.414004658533019: 1, 11.403253363412137: 1,
11.400383537421511: 1, 11.398989638110736: 1, 11.387538502493111: 1, 11.3875
15526666919: 1, 11.387176284117533: 1, 11.38037448285336: 1, 11.306907603907

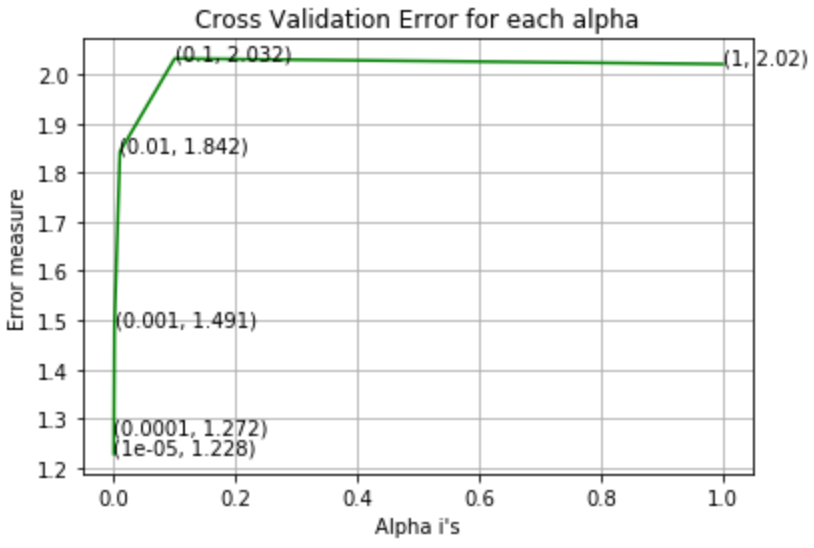
675: 1, 11.262818339802426: 1, 11.25640489521316: 1, 11.252189525330369: 1,
11.242591942307513: 1, 11.239412182728438: 1, 11.21024920091681: 1, 11.20882
2669129608: 1, 11.199066304256887: 1, 11.166963766114275: 1, 11.160050020012
054: 1, 11.152267679471251: 1, 11.146853588194833: 1, 11.12919379488357: 1,
11.11334132995865: 1, 11.107236332924982: 1, 11.077332398437662: 1, 11.05574
707171497: 1, 11.02661415534901: 1, 10.99763527166814: 1, 10.97522573323203
5: 1, 10.964683595020146: 1, 10.962922205906136: 1, 10.955753253070075: 1, 1
0.955235223562866: 1, 10.91089001215484: 1, 10.908278940682079: 1, 10.895433
084528683: 1, 10.895275128712706: 1, 10.894217630577065: 1, 10.8856065554136
02: 1, 10.881195115447817: 1, 10.870167680402362: 1, 10.844735709162764: 1,
10.844415477654065: 1, 10.837084715169757: 1, 10.82860732721304: 1, 10.82638
5409355117: 1, 10.796989916502415: 1, 10.793381850808842: 1, 10.783388899432
02: 1, 10.776945630036673: 1, 10.75002908521235: 1, 10.71328307695489: 1, 1
0.698419279209563: 1, 10.692985324700336: 1, 10.683200605626585: 1, 10.67854
4062024296: 1, 10.676263405887445: 1, 10.655590152471559: 1, 10.653680106042
973: 1, 10.634352502040896: 1, 10.630970339831197: 1, 10.616653692109386: 1,
10.61191446116134: 1, 10.587736493098761: 1, 10.57527301720664: 1, 10.562454
022873654: 1, 10.559005046857315: 1, 10.541106479241023: 1, 10.5354124913315
13: 1, 10.522567852317334: 1, 10.51222181067435: 1, 10.511400608539622: 1, 1
0.511300421196754: 1, 10.495164890982434: 1, 10.484143730616898: 1, 10.47329
9285397951: 1, 10.471535318385461: 1, 10.45751700692113: 1, 10.4507100602410
57: 1, 10.419865960958786: 1, 10.417725035560885: 1, 10.415082478567394: 1,
10.394000940947928: 1, 10.382394357184527: 1, 10.354336408596696: 1, 10.3530
11402645446: 1, 10.328461073484139: 1, 10.30104937570316: 1, 10.269637698261
21: 1, 10.250948734500941: 1, 10.24045992601376: 1, 10.22552954867233: 1, 1
0.220748977867217: 1, 10.217593118894445: 1, 10.217411520431433: 1, 10.21333
1274052852: 1, 10.203862510124138: 1, 10.202724109735437: 1, 10.197968394837
542: 1, 10.195769671730403: 1, 10.185152396120811: 1, 10.183295121297522: 1,
10.172208563953243: 1, 10.158806349393203: 1, 10.139773349422482: 1, 10.1122
3926829064: 1, 10.106302761474362: 1, 10.105442527721479: 1, 10.102842940540
08: 1, 10.097106738492084: 1, 10.06856564745287: 1, 10.059577630327977: 1, 1
0.04781533441906: 1, 10.046047991665272: 1, 10.007946004061942: 1, 9.9975471
37247446: 1, 9.971884031012772: 1, 9.955832836649114: 1, 9.926692424355148:
1, 9.920002431004656: 1, 9.91852239703198: 1, 9.90089787800352: 1, 9.8932167
38392168: 1, 9.885333012519974: 1, 9.874676956411175: 1, 9.874061961895805:
1, 9.84667928002406: 1, 9.835288083287809: 1, 9.832829421271226: 1, 9.831580
936780163: 1, 9.818624138988303: 1, 9.784842514673379: 1, 9.776466224038296:
1, 9.770695171540876: 1, 9.768920884772928: 1, 9.767102595102289: 1, 9.76342
0953313895: 1, 9.754559764094399: 1, 9.742008981025345: 1, 9.73845463671605
8: 1, 9.736950091438745: 1, 9.733111662069234: 1, 9.728406266115892: 1, 9.69
0239011531409: 1, 9.685184390662217: 1, 9.671816754094088: 1, 9.668480298408
555: 1, 9.66304184399181: 1, 9.657615008731916: 1, 9.65679674470232: 1, 9.64
6001127474738: 1, 9.623310486316475: 1, 9.618733823238795: 1, 9.606577159289
12: 1, 9.600635716904518: 1, 9.599761808910786: 1, 9.587939257160043: 1, 9.5
79576487216407: 1, 9.562254501434984: 1, 9.559510899810412: 1, 9.52955121354
9066: 1, 9.518565012637774: 1, 9.513343901625857: 1, 9.499360992030354: 1,
9.485815002444493: 1, 9.481347816911176: 1, 9.478359804837105: 1, 9.46916625
8433804: 1, 9.46508484485039: 1, 9.450591500806885: 1, 9.447703665334487: 1,
9.447190811441898: 1, 9.422614698978116: 1, 9.412876278372934: 1, 9.41149605
844427: 1, 9.40091880122847: 1, 9.398389930846221: 1, 9.391418621390224: 1,
9.39030513538749: 1, 9.375922661661031: 1, 9.338596824756493: 1, 9.325197907
625247: 1, 9.316352252218131: 1, 9.312329947072262: 1, 9.304994962355396: 1,
9.301970809528234: 1, 9.298863960853868: 1, 9.281162204865065: 1, 9.27977764
81808: 1, 9.241106885510188: 1, 9.239728152908096: 1, 9.23529810794901: 1,
9.196669602545994: 1, 9.178422341541769: 1, 9.17208184361428: 1, 9.170626786
062492: 1, 9.165067100851287: 1, 9.164496746584808: 1, 9.1601669524143: 1,
9.153942619315236: 1, 9.151379197589103: 1, 9.145221524245374: 1, 9.12982134
7250735: 1, 9.126418347460525: 1, 9.126196739186016: 1, 9.105701363017513:
1, 9.097930863593374: 1, 9.088241346379911: 1, 9.079862981540659: 1, 9.05978
94959831: 1, 9.023898845220476: 1, 9.02372450600261: 1, 9.016308059184526:
1, 9.007591410252267: 1, 9.005502323637268: 1, 9.003371796903753: 1, 9.00070
1018401683: 1, 8.997316108528633: 1, 8.99312779975333: 1, 8.987879089548747:

1, 8.970158584929539: 1, 8.94880753090148: 1, 8.946823268091492: 1, 8.936593
489614797: 1, 8.93519010397895: 1, 8.923076489761305: 1, 8.918275083063753:
1, 8.886933401451541: 1, 8.881347706051425: 1, 8.877092950494692: 1, 8.86372
6480251904: 1, 8.86024215133172: 1, 8.858829580418432: 1, 8.837840275652418:
1, 8.83590744464127: 1, 8.831762252290956: 1, 8.824207527161885: 1, 8.819074
778325582: 1, 8.801934780285988: 1, 8.78725091462289: 1, 8.78528054851851:
1, 8.78491499655734: 1, 8.78125820114658: 1, 8.779489454010722: 1, 8.7748560
19385885: 1, 8.764980454584315: 1, 8.763130896017993: 1, 8.762983168244567:
1, 8.758506076139835: 1, 8.758141028182589: 1, 8.75708645828349: 1, 8.754867
939650191: 1, 8.750865397515152: 1, 8.748348613445767: 1, 8.74694134403829:
1, 8.724384518359217: 1, 8.672787059479061: 1, 8.657668907841545: 1, 8.63469
6587009875: 1, 8.633919404034176: 1, 8.606034075026844: 1, 8.60432674094121
8: 1, 8.598394929689173: 1, 8.596844475174699: 1, 8.591583598426126: 1, 8.58
4039371534166: 1, 8.568192009082802: 1, 8.564685208978837: 1, 8.543792713279
084: 1, 8.520069895673075: 1, 8.506075913377865: 1, 8.492399998146517: 1, 8.
491167429184044: 1, 8.484574607171005: 1, 8.454879168156934: 1, 8.4431288292
85984: 1, 8.442167915215894: 1, 8.433072184133218: 1, 8.422535529390359: 1,
8.406319205012426: 1, 8.405429074020415: 1, 8.398149910064179: 1, 8.39693082
2968208: 1, 8.370378062563464: 1, 8.369649788857474: 1, 8.361309707507653:
1, 8.351666017057848: 1, 8.349331296724937: 1, 8.34781149067474: 1, 8.337785
70050384: 1, 8.31748068967022: 1, 8.310284360725593: 1, 8.304668844764956:
1, 8.286586574339465: 1, 8.272538030697591: 1, 8.267280400128053: 1, 8.19138
3185699696: 1, 8.136625080577078: 1, 8.131832274802196: 1, 8.11850416999759
4: 1, 8.08539852919023: 1, 8.08401812807047: 1, 8.077677062229798: 1, 8.0594
23098264604: 1, 8.05567080909689: 1, 7.9931809009024235: 1, 7.98860256280530
4: 1, 7.984419665769326: 1, 7.970380256702725: 1, 7.962587195856419: 1, 7.93
9620352818787: 1, 7.9360915899793065: 1, 7.932662122414827: 1, 7.93122849542
092: 1, 7.923630365874301: 1, 7.915043628869993: 1, 7.905157742188584: 1, 7.
881898515772249: 1, 7.880882955351807: 1, 7.877559678495513: 1, 7.8707842689
32044: 1, 7.8423946487768825: 1, 7.82774337864381: 1, 7.822134676257455: 1,
7.8193049414058855: 1, 7.817570533796929: 1, 7.805890024865479: 1, 7.8054241
83954712: 1, 7.805110249027472: 1, 7.780839653852427: 1, 7.771755703815484:
1, 7.758102726452453: 1, 7.755093249240273: 1, 7.7522024749053156: 1, 7.7500
46033110902: 1, 7.747860691766928: 1, 7.729327443098042: 1, 7.72929179245904
6: 1, 7.728282064303092: 1, 7.714912996315197: 1, 7.672742489028083: 1, 7.67
2444755688531: 1, 7.656678608254542: 1, 7.644220157836531: 1, 7.635627029534
249: 1, 7.6329030022813535: 1, 7.617854834995611: 1, 7.617591074725545: 1,
7.603285164142247: 1, 7.596599840426935: 1, 7.588727940600021: 1, 7.5749348
921287165: 1, 7.565567559280216: 1, 7.56121542833702: 1, 7.556585232918044:
1, 7.517814823360422: 1, 7.4300442478647994: 1, 7.400026174399118: 1, 7.3897
45326139708: 1, 7.351807430340666: 1, 7.351207066691874: 1, 7.321995558180874: 1, 7.31016509961918: 1, 7.3020631084783645: 1, 7.28441214615859: 1, 7.256
510833126594: 1, 7.218173395300184: 1, 7.2126686942038525: 1, 7.202369637341
593: 1, 7.197865331419381: 1, 7.1974488080594865: 1, 7.183652908950468: 1,
7.1582171899858125: 1, 7.124353683906635: 1, 7.104294339855586: 1, 7.1031499
1039213: 1, 7.096520037042779: 1, 7.09614404612092: 1, 7.093536565648409: 1,
7.081818136108998: 1, 7.04740157426142: 1, 7.036005408287328: 1, 6.984438030
045087: 1, 6.945479371822156: 1, 6.911526096269914: 1, 6.881460178214983: 1,
6.8733338482021455: 1, 6.868759646759431: 1, 6.837198697398309: 1, 6.8229341
61300998: 1, 6.81073749534474: 1, 6.784822662580922: 1, 6.76474482385993: 1,
6.752163061528158: 1, 6.733597879586418: 1, 6.730992363143084: 1, 6.57530753
1278251: 1, 6.25101758951101: 1})

In [67]:
```
# Train a Logistic regression+Calibration model using text features whicha re
 on-hot encoded
alpha = [10 ** x for x in range(-5, 1)]


cv_log_error_array=[]
for i in alpha:
    clf = SGDClassifier(alpha=i, penalty='l2', loss='log', random_state=42)
```

```
    clf.fit(train_text_feature_tfidf, y_train)

    sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
    sig_clf.fit(train_text_feature_tfidf, y_train)
    predict_y = sig_clf.predict_proba(cv_text_feature_tfidf)
    cv_log_error_array.append(log_loss(y_cv, predict_y, labels=clf.classes_, e
ps=1e-15))
    print('For values of alpha = ', i, "The log loss is:",log_loss(y_cv, predi
ct_y, labels=clf.classes_, eps=1e-15))

fig, ax = plt.subplots()
ax.plot(alpha, cv_log_error_array,c='g')
for i, txt in enumerate(np.round(cv_log_error_array,3)):
    ax.annotate((alpha[i],np.round(txt,3)), (alpha[i],cv_log_error_array[i]))
plt.grid()
plt.title("Cross Validation Error for each alpha")
plt.xlabel("Alpha i's")
plt.ylabel("Error measure")
plt.show()


best_alpha = np.argmin(cv_log_error_array)
clf = SGDClassifier(alpha=alpha[best_alpha], penalty='l2', loss='log', random_
state=42)
clf.fit(train_text_feature_tfidf, y_train)
sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
sig_clf.fit(train_text_feature_tfidf, y_train)

predict_y = sig_clf.predict_proba(train_text_feature_tfidf)
print('For values of best alpha = ', alpha[best_alpha], "The train log loss i
s:",log_loss(y_train, predict_y, labels=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(cv_text_feature_tfidf)
print('For values of best alpha = ', alpha[best_alpha], "The cross validation
 log loss is:",log_loss(y_cv, predict_y, labels=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(test_text_feature_tfidf)
print('For values of best alpha = ', alpha[best_alpha], "The test log loss i
s:",log_loss(y_test, predict_y, labels=clf.classes_, eps=1e-15))
```

```
For values of alpha =  1e-05 The log loss is: 1.228419508554544
For values of alpha =  0.0001 The log loss is: 1.271669328280693
For values of alpha =  0.001 The log loss is: 1.4906726519093054
For values of alpha =  0.01 The log loss is: 1.8423886300971135
For values of alpha =  0.1 The log loss is: 2.031818831038479
For values of alpha =  1 The log loss is: 2.0204377235244677
```



```
For values of best alpha =  1e-05 The train log loss is: 0.7054744357184605
For values of best alpha =  1e-05 The cross validation log loss is: 1.228419
508554544
For values of best alpha =  1e-05 The test log loss is: 1.181010621919759
```

Q. Is the Text feature stable across all the data sets (Test, Train, Cross validation)?

Ans. Yes, it seems like!

In [54]:
```
def get_intersec_text(df):
    df_text_vec = TfidfVectorizer(min_df=3)
    df_text_fea = df_text_vec.fit_transform(df['TEXT'])
    df_text_features = df_text_vec.get_feature_names()

    df_text_fea_counts = df_text_fea.sum(axis=0).A1
    df_text_fea_dict = dict(zip(list(df_text_features),df_text_fea_counts))
    len1 = len(set(df_text_features))
    len2 = len(set(train_text_features) & set(df_text_features))
    return len1,len2
```

In [55]:
```
len1,len2 = get_intersec_text(test_df)
print(np.round((len2/len1)*100, 3), "% of word of test data appeared in train
 data")
len1,len2 = get_intersec_text(cv_df)
print(np.round((len2/len1)*100, 3), "% of word of Cross Validation appeared in
train data")
```

```
3.398 % of word of test data appeared in train data
3.875 % of word of Cross Validation appeared in train data
```

4. Machine Learning Models

In [24]:
```
#Data preparation for ML models.

#Misc. functionns for ML models


def predict_and_plot_confusion_matrix(train_x, train_y,test_x, test_y, clf):
    clf.fit(train_x, train_y)
    sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
    sig_clf.fit(train_x, train_y)
    pred_y = sig_clf.predict(test_x)

    # for calculating log_loss we willl provide the array of probabilities bel
ongs to each class
    print("Log loss :",log_loss(test_y, sig_clf.predict_proba(test_x)))
    # calculating the number of data points that are misclassified
    print("Number of mis-classified points :", np.count_nonzero((pred_y- test_
y))/test_y.shape[0])
    plot_confusion_matrix(test_y, pred_y)
```

In [25]:
```
def report_log_loss(train_x, train_y, test_x, test_y,  clf):
    clf.fit(train_x, train_y)
    sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
    sig_clf.fit(train_x, train_y)
    sig_clf_probs = sig_clf.predict_proba(test_x)
    return log_loss(test_y, sig_clf_probs, eps=1e-15)
```

In [26]:
```
# this function will be used just for naive bayes
# for the given indices, we will print the name of the features
# and we will check whether the feature present in the test point text or not
def get_impfeature_names(indices, text, gene, var, no_features):
    gene_count_vec = CountVectorizer()
    var_count_vec = CountVectorizer()
    text_count_vec = TfidfVectorizer(min_df=3)

    gene_vec = gene_count_vec.fit(train_df['Gene'])
    var_vec  = var_count_vec.fit(train_df['Variation'])
    text_vec = text_count_vec.fit(train_df['TEXT'])

    fea1_len = len(gene_vec.get_feature_names())
    fea2_len = len(var_count_vec.get_feature_names())

    word_present = 0
    for i,v in enumerate(indices):
        if (v < fea1_len):
            word = gene_vec.get_feature_names()[v]
            yes_no = True if word == gene else False
            if yes_no:
                word_present += 1
                print(i, "Gene feature [{}] present in test data point [{}]".f
ormat(word,yes_no))
        elif (v < fea1_len+fea2_len):
            word = var_vec.get_feature_names()[v-(fea1_len)]
            yes_no = True if word == var else False
            if yes_no:
                word_present += 1
                print(i, "variation feature [{}] present in test data point [
{}]".format(word,yes_no))
        else:
            word = text_vec.get_feature_names()[v-(fea1_len+fea2_len)]
            yes_no = True if word in text.split() else False
            if yes_no:
                word_present += 1
                print(i, "Text feature [{}] present in test data point [{}]".f
ormat(word,yes_no))

    print("Out of the top ",no_features," features ", word_present, "are prese
nt in query point")
```

Stacking the three types of features

In [75]:
```
# merging gene, variance and text features

# building train, test and cross validation data sets
# a = [[1, 2],
#      [3, 4]]
# b = [[4, 5],
#      [6, 7]]
# hstack(a, b) = [[1, 2, 4, 5],
#                 [ 3, 4, 6, 7]]

train_gene_var_tfidf = hstack((train_gene_feature_onehotCoding,train_variation
_feature_onehotCoding))
test_gene_var_tfidf = hstack((test_gene_feature_tf_idf,test_variation_feature_
onehotCoding))
```

```
cv_gene_var_tfidf = hstack((cv_gene_feature_onehotCoding,cv_variation_feature_
onehotCoding))

train_x_tfidf = hstack((train_gene_var_tfidf, train_text_feature_tfidf)).tocsr
()
train_y = np.array(list(train_df['Class']))

test_x_tfidf = hstack((test_gene_var_tfidf, test_text_feature_tfidf)).tocsr()
test_y = np.array(list(test_df['Class']))

cv_x_tfidf = hstack((cv_gene_var_tfidf, cv_text_feature_tfidf)).tocsr()
cv_y = np.array(list(cv_df['Class']))


train_gene_var_responseCoding = np.hstack((train_gene_feature_responseCoding,t
rain_variation_feature_responseCoding))
test_gene_var_responseCoding = np.hstack((test_gene_feature_responseCoding,tes
t_variation_feature_responseCoding))
cv_gene_var_responseCoding = np.hstack((cv_gene_feature_responseCoding,cv_vari
ation_feature_responseCoding))

train_x_responseCoding = np.hstack((train_gene_var_responseCoding, train_text_
feature_responseCoding))
test_x_responseCoding = np.hstack((test_gene_var_responseCoding, test_text_fea
ture_responseCoding))
cv_x_responseCoding = np.hstack((cv_gene_var_responseCoding, cv_text_feature_r
esponseCoding))
```

In [76]:
```
print("One tfidf encoding features :")
print("(number of data points * number of features) in train data = ", train_x
_tfidf.shape)
print("(number of data points * number of features) in test data = ", test_x_t
fidf.shape)
print("(number of data points * number of features) in cross validation data
 =", cv_x_tfidf.shape)
```

```
One tfidf encoding features :
(number of data points * number of features) in train data =  (2124, 3196)
(number of data points * number of features) in test data =  (665, 3196)
(number of data points * number of features) in cross validation data = (53
2, 3196)
```

In [77]:
```
print(" Response encoding features :")
print("(number of data points * number of features) in train data = ", train_x
_responseCoding.shape)
print("(number of data points * number of features) in test data = ", test_x_r
esponseCoding.shape)
print("(number of data points * number of features) in cross validation data
 =", cv_x_responseCoding.shape)
```

```
 Response encoding features :
(number of data points * number of features) in train data =  (2124, 27)
(number of data points * number of features) in test data =  (665, 27)
(number of data points * number of features) in cross validation data = (53
2, 27)
```

4.1. Base Line Model

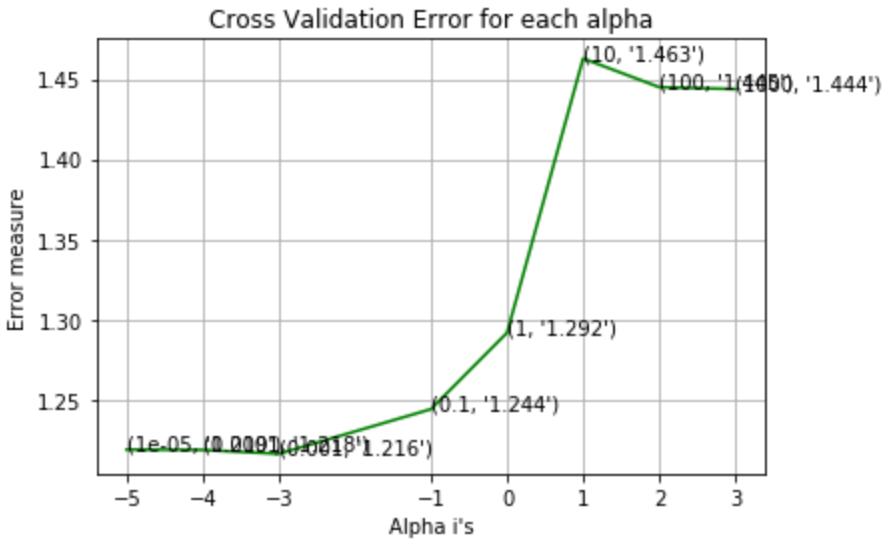
4.1.1. Naive Bayes

4.1.1.1. Hyper parameter tuning

In [78]:
```
alpha = [0.00001, 0.0001, 0.001, 0.1, 1, 10, 100,1000]
cv_log_error_array = []
for i in alpha:
    print("for alpha =", i)
    clf = MultinomialNB(alpha=i)
    clf.fit(train_x_tfidf, train_y)
    sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
    sig_clf.fit(train_x_tfidf, train_y)
    sig_clf_probs = sig_clf.predict_proba(cv_x_tfidf)
    cv_log_error_array.append(log_loss(cv_y, sig_clf_probs, labels=clf.classes
_, eps=1e-15))
    # to avoid rounding error while multiplying probabilites we use log-probab
ility estimates
    print("Log Loss :",log_loss(cv_y, sig_clf_probs))

fig, ax = plt.subplots()
ax.plot(np.log10(alpha), cv_log_error_array,c='g')
for i, txt in enumerate(np.round(cv_log_error_array,3)):
    ax.annotate((alpha[i],str(txt)), (np.log10(alpha[i]),cv_log_error_array[i
]))
plt.grid()
plt.xticks(np.log10(alpha))
plt.title("Cross Validation Error for each alpha")
plt.xlabel("Alpha i's")
plt.ylabel("Error measure")
plt.show()


best_alpha = np.argmin(cv_log_error_array)
clf = MultinomialNB(alpha=alpha[best_alpha])
clf.fit(train_x_tfidf, train_y)
sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
sig_clf.fit(train_x_tfidf, train_y)


predict_y = sig_clf.predict_proba(train_x_tfidf)
print('For values of best alpha = ', alpha[best_alpha], "The train log loss i
s:",log_loss(y_train, predict_y, labels=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(cv_x_tfidf)
print('For values of best alpha = ', alpha[best_alpha], "The cross validation
 log loss is:",log_loss(y_cv, predict_y, labels=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(test_x_tfidf)
print('For values of best alpha = ', alpha[best_alpha], "The test log loss i
s:",log_loss(y_test, predict_y, labels=clf.classes_, eps=1e-15))
```

```
for alpha = 1e-05
Log Loss : 1.2186582860153912

for alpha = 0.0001
Log Loss : 1.2184427441925039
for alpha = 0.001
Log Loss : 1.216009029569923
for alpha = 0.1
Log Loss : 1.243908860854463
for alpha = 1
Log Loss : 1.2916492430710935
for alpha = 10
```

```
Log Loss : 1.4629920433992019
for alpha = 100
Log Loss : 1.445079088627846
for alpha = 1000
Log Loss : 1.4438886838109255
```



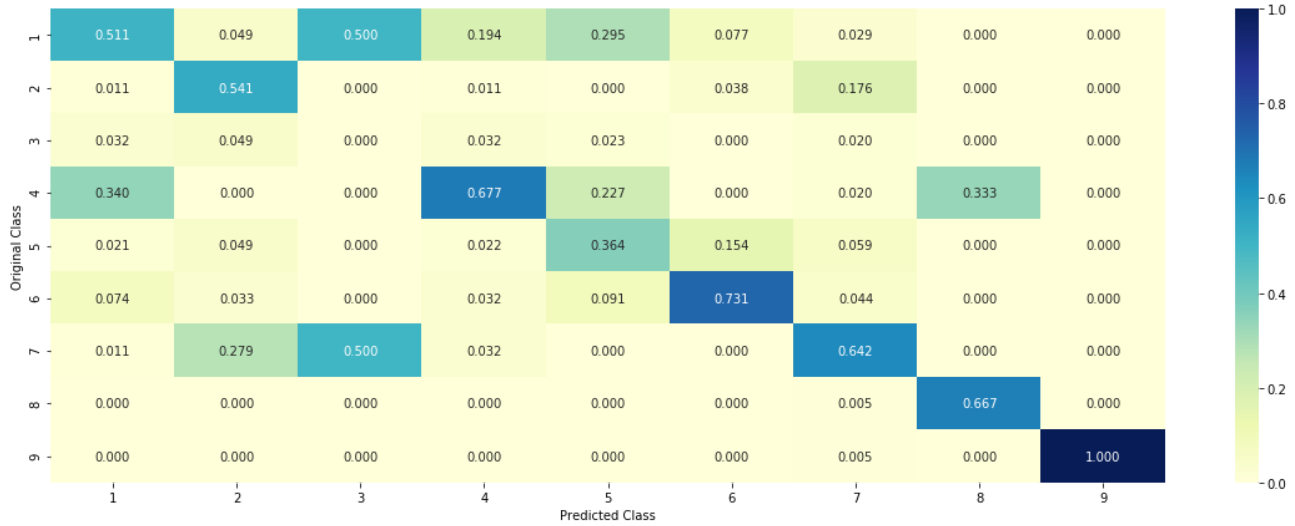

```
For values of best alpha =  0.001 The train log loss is: 0.5225733310345027
For values of best alpha =  0.001 The cross validation log loss is: 1.216009
029569923
For values of best alpha =  0.001 The test log loss is: 1.167277216771183
```

4.1.1.2. Testing the model with best hyper paramters

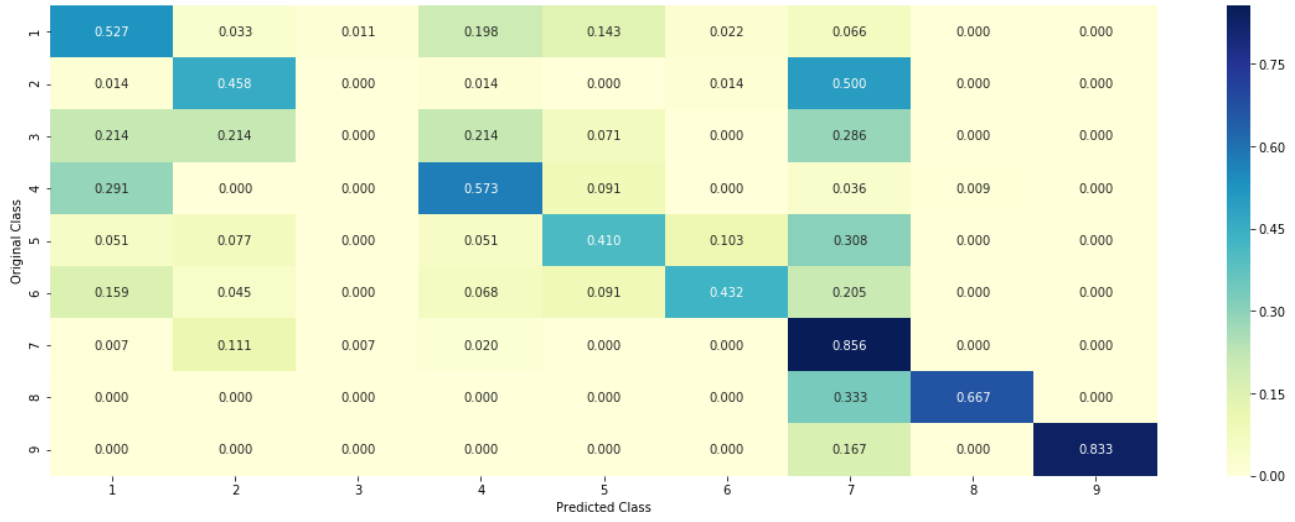
In [85]:
```
clf = MultinomialNB(alpha=alpha[best_alpha])
clf.fit(train_x_tfidf, train_y)
sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
sig_clf.fit(train_x_tfidf, train_y)
sig_clf_probs = sig_clf.predict_proba(cv_x_tfidf)
# to avoid rounding error while multiplying probabilites we use log-probabilit
y estimates
print("Log Loss :",log_loss(cv_y, sig_clf_probs))
print("Number of missclassified point :", np.count_nonzero((sig_clf.predict(cv
_x_tfidf)- cv_y))/cv_y.shape[0])
plot_confusion_matrix(cv_y, sig_clf.predict(cv_x_tfidf.toarray()))
```

```
Log Loss : 1.216009029569923
Number of missclassified point : 0.4041353383458647
-------------------- Confusion matrix --------------------
```

---------- Precision matrix (Columm Sum=1) ----------



---------- Recall matrix (Row sum=1) ----------



4.1.1.3. Feature Importance, Correctly classified point

In [89]:
```
test_point_index = 15
no_feature = 100
predicted_cls = sig_clf.predict(test_x_tfidf[test_point_index])
print("Predicted Class :", predicted_cls[0])
print("Predicted Class Probabilities:", np.round(sig_clf.predict_proba(test_x_
tfidf[test_point_index]),4))
```

```
print("Actual Class :", test_y[test_point_index])
indices = np.argsort(-clf.coef_)[predicted_cls-1][:,:no_feature]
print("-"*50)
get_impfeature_names(indices[0], test_df['TEXT'].iloc[test_point_index],test_df['Gene'].iloc[test_point_index],test_df['Variation'].iloc[test_point_index], no_feature)
```

```
Predicted Class : 4
Predicted Class Probabilities: [[0.1035 0.0455 0.0076 0.6872 0.0327 0.0315
0.0862 0.0031 0.0027]]
Actual Class : 4
--------------------------------------------------
26 Text feature [004] present in test data point [True]
42 Text feature [105k] present in test data point [True]
45 Text feature [105] present in test data point [True]
51 Text feature [12] present in test data point [True]
53 Text feature [0006] present in test data point [True]
62 Text feature [0h] present in test data point [True]
95 Text feature [083] present in test data point [True]
Out of the top  100  features  7 are present in query point
```

4.1.1.4. Feature Importance, Incorrectly classified point

In [91]:
```
test_point_index = 12
no_feature = 100
predicted_cls = sig_clf.predict(test_x_tfidf[test_point_index])
print("Predicted Class :", predicted_cls[0])
print("Predicted Class Probabilities:", np.round(sig_clf.predict_proba(test_x_tfidf[test_point_index]),4))
print("Actual Class :", test_y[test_point_index])
indices = np.argsort(-clf.coef_)[predicted_cls-1][:,:no_feature]
print("-"*50)
get_impfeature_names(indices[0], test_df['TEXT'].iloc[test_point_index],test_df['Gene'].iloc[test_point_index],test_df['Variation'].iloc[test_point_index], no_feature)
```

```
Predicted Class : 6
Predicted Class Probabilities: [[0.0613 0.0454 0.0081 0.081  0.0372 0.6752
0.086  0.0031 0.0026]]
Actual Class : 6
--------------------------------------------------
Out of the top  100  features  0 are present in query point
```

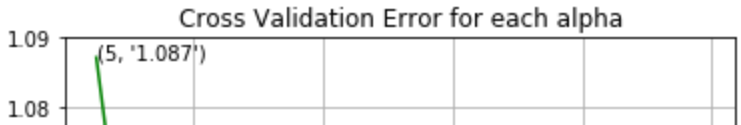
4.2. K Nearest Neighbour Classification

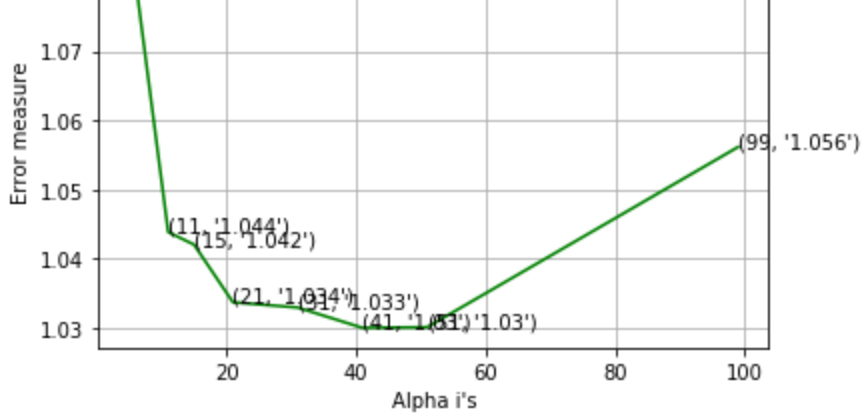
4.2.1. Hyper parameter tuning

In [87]:
```
# find more about KNeighborsClassifier() here http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html
# -------------------------
# default parameter
# KNeighborsClassifier(n_neighbors=5, weights='uniform', algorithm='auto', leaf_size=30, p=2,
# metric='minkowski', metric_params=None, n_jobs=1, **kwargs)


alpha = [5, 11, 15, 21, 31, 41, 51, 99]
```

```
cv_log_error_array = []
for i in alpha:
    print("for alpha =", i)
    clf = KNeighborsClassifier(n_neighbors=i)
    clf.fit(train_x_responseCoding, train_y)
    sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
    sig_clf.fit(train_x_responseCoding, train_y)
    sig_clf_probs = sig_clf.predict_proba(cv_x_responseCoding)
    cv_log_error_array.append(log_loss(cv_y, sig_clf_probs, labels=clf.classes
_, eps=1e-15))
    # to avoid rounding error while multiplying probabilites we use log-probab
ility estimates
    print("Log Loss :",log_loss(cv_y, sig_clf_probs))

fig, ax = plt.subplots()
ax.plot(alpha, cv_log_error_array,c='g')
for i, txt in enumerate(np.round(cv_log_error_array,3)):
    ax.annotate((alpha[i],str(txt)), (alpha[i],cv_log_error_array[i]))
plt.grid()
plt.title("Cross Validation Error for each alpha")
plt.xlabel("Alpha i's")
plt.ylabel("Error measure")
plt.show()


best_alpha = np.argmin(cv_log_error_array)
clf = KNeighborsClassifier(n_neighbors=alpha[best_alpha])
clf.fit(train_x_responseCoding, train_y)
sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
sig_clf.fit(train_x_responseCoding, train_y)

predict_y = sig_clf.predict_proba(train_x_responseCoding)
print('For values of best alpha = ', alpha[best_alpha], "The train log loss i
s:",log_loss(y_train, predict_y, labels=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(cv_x_responseCoding)
print('For values of best alpha = ', alpha[best_alpha], "The cross validation
 log loss is:",log_loss(y_cv, predict_y, labels=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(test_x_responseCoding)
print('For values of best alpha = ', alpha[best_alpha], "The test log loss i
s:",log_loss(y_test, predict_y, labels=clf.classes_, eps=1e-15))
```

```
for alpha = 5
Log Loss : 1.087179276607684
for alpha = 11
Log Loss : 1.043843366639075
for alpha = 15
Log Loss : 1.0420186802873235

for alpha = 21
Log Loss : 1.033679711584495
for alpha = 31
Log Loss : 1.032920998349714
for alpha = 41
Log Loss : 1.0299833043448472
for alpha = 51
Log Loss : 1.0300790747990303
for alpha = 99
Log Loss : 1.0561420020809233
```



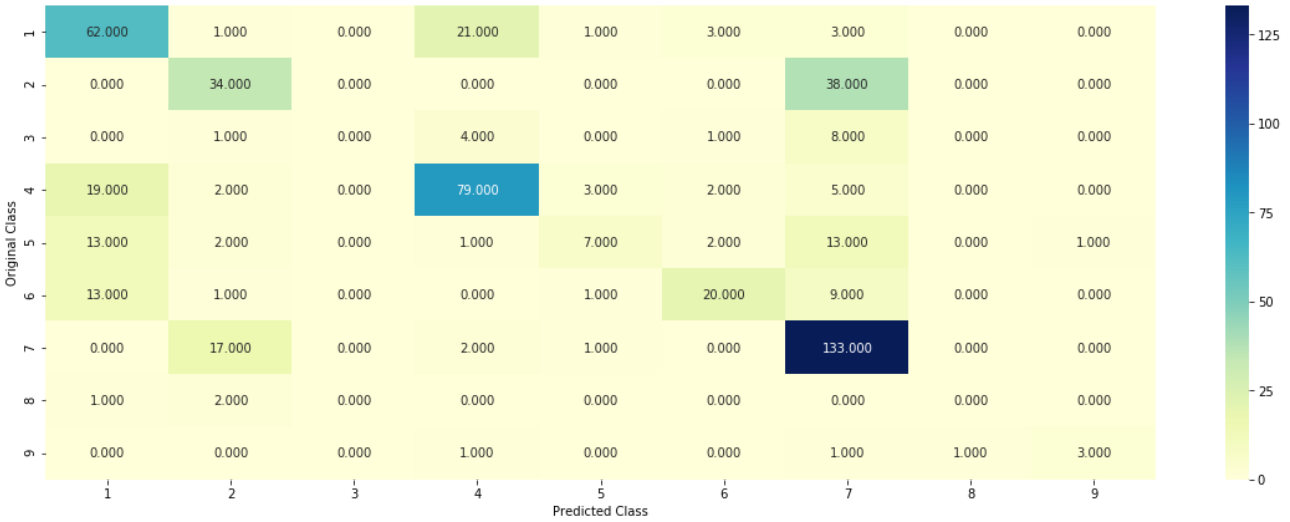



```
For values of best alpha =  41 The train log loss is: 0.859694004887734
For values of best alpha =  41 The cross validation log loss is: 1.029983304
3448472
For values of best alpha =  41 The test log loss is: 1.1429068858152711
```

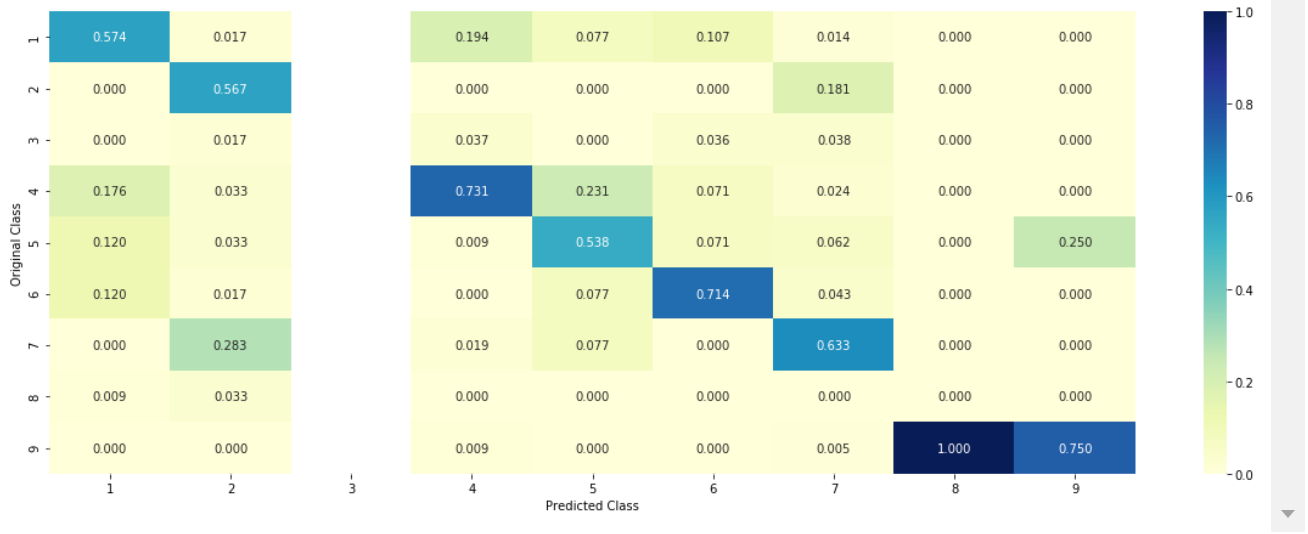
4.2.2. Testing the model with best hyper paramters

In [89]:
```
clf = KNeighborsClassifier(n_neighbors=alpha[best_alpha])
predict_and_plot_confusion_matrix(train_x_responseCoding, train_y, cv_x_respon
seCoding, cv_y, clf)
```

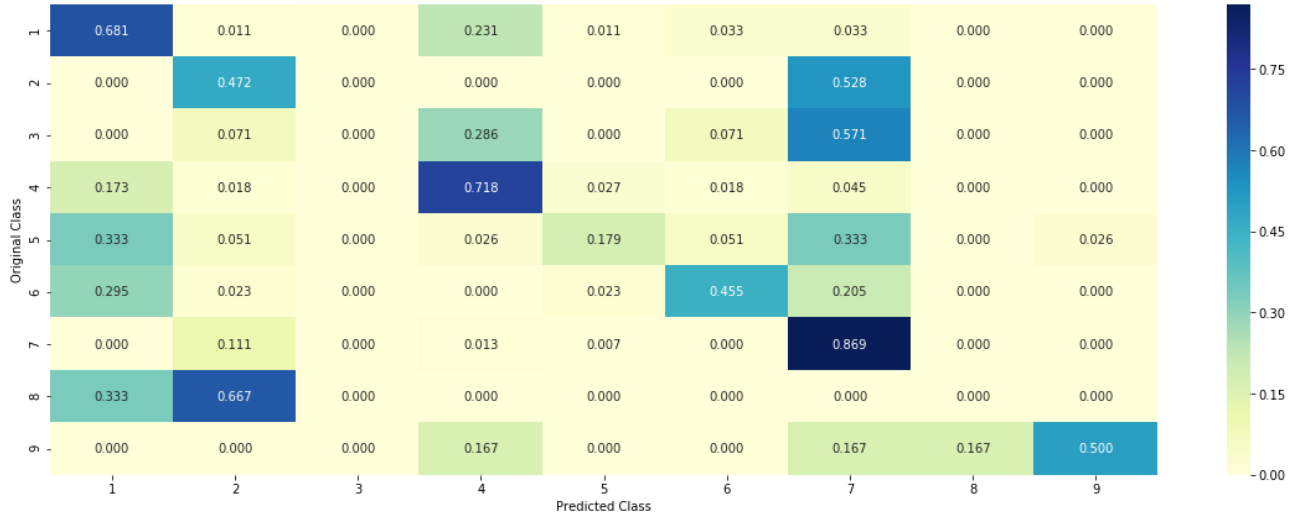
```
Log loss : 1.0299833043448472
Number of mis-classified points : 0.36466165413533835
-------------------- Confusion matrix --------------------
```




```
-------------------- Precision matrix (Columm Sum=1) --------------------
```



---------- Recall matrix (Row sum=1) ----------



4.2.3.Sample Query point -1

In [91]:

```
clf = KNeighborsClassifier(n_neighbors=alpha[best_alpha])
clf.fit(train_x_responseCoding, train_y)
sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
sig_clf.fit(train_x_responseCoding, train_y)

test_point_index = 100
predicted_cls = sig_clf.predict(test_x_responseCoding[0].reshape(1,-1))
print("Predicted Class :", predicted_cls[0])
print("Actual Class :", test_y[test_point_index])
neighbors = clf.kneighbors(test_x_responseCoding[test_point_index].reshape(1,
-1), alpha[best_alpha])
print("The ",alpha[best_alpha]," nearest neighbours of the test points belongs
to classes",train_y[neighbors[1][0]])
print("Fequency of nearest points :",Counter(train_y[neighbors[1][0]]))
```

```
Predicted Class : 4
Actual Class : 4
The  41  nearest neighbours of the test points belongs to classes [4 1 4 4 4
4 4 1 1 4 4 4 4 4 7 4 2 4 4 4 4 4 4 4 4 4 4 4 4 4 4 5 1 4 4 1 4
 4 4 4 4]
Fequency of nearest points : Counter({4: 33, 1: 5, 7: 1, 2: 1, 5: 1})
```

4.2.4. Sample Query Point-2

In [95]:
```
test_point_index = 600
predicted_cls = sig_clf.predict(test_x_responseCoding[0].reshape(1,-1))
print("Predicted Class :", predicted_cls[0])
print("Actual Class :", test_y[test_point_index])
neighbors = clf.kneighbors(test_x_responseCoding[test_point_index].reshape(1,
-1), alpha[best_alpha])
print("The ",alpha[best_alpha]," nearest neighbours of the test points belongs
to classes",train_y[neighbors[1][0]])
print("Fequency of nearest points :",Counter(train_y[neighbors[1][0]]))
```

```
Predicted Class : 4
Actual Class : 6
The  41  nearest neighbours of the test points belongs to classes [6 6 6 6 6
6 6 6 6 6 6 6 6 6 6 6 4 4 4 3 4 4 4 5 3 3 6 3 6 5 5 4 3 4 3 4
 6 6 4 4]
Fequency of nearest points : Counter({6: 21, 4: 11, 3: 6, 5: 3})
```

4.3. Logistic Regression

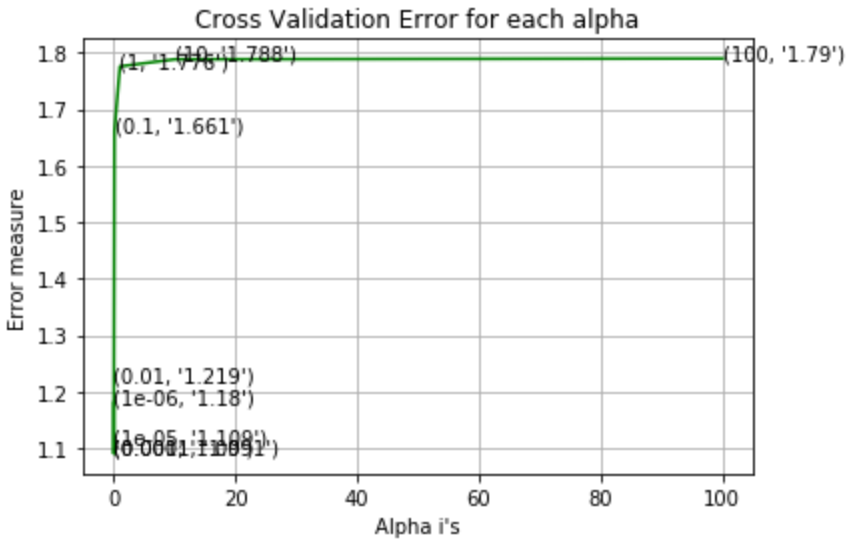
4.3.1. With Class balancing

4.3.1.1. Hyper paramter tuning

In [92]:
```
alpha = [10 ** x for x in range(-6, 3)]
cv_log_error_array = []
for i in alpha:
    print("for alpha =", i)
    clf = SGDClassifier(class_weight='balanced', alpha=i, penalty='l2', loss=
'log', random_state=42)
    clf.fit(train_x_tfidf, train_y)
    sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
    sig_clf.fit(train_x_tfidf, train_y)
    sig_clf_probs = sig_clf.predict_proba(cv_x_tfidf)
    cv_log_error_array.append(log_loss(cv_y, sig_clf_probs, labels=clf.classes
_, eps=1e-15))
    # to avoid rounding error while multiplying probabilites we use log-probab
ility estimates
    print("Log Loss :",log_loss(cv_y, sig_clf_probs))

fig, ax = plt.subplots()
ax.plot(alpha, cv_log_error_array,c='g')
for i, txt in enumerate(np.round(cv_log_error_array,3)):
    ax.annotate((alpha[i],str(txt)), (alpha[i],cv_log_error_array[i]))
plt.grid()
plt.title("Cross Validation Error for each alpha")
plt.xlabel("Alpha i's")
plt.ylabel("Error measure")
plt.show()


best_alpha = np.argmin(cv_log_error_array)
clf = SGDClassifier(class_weight='balanced', alpha=alpha[best_alpha], penalty=
'l2', loss='log', random_state=42)
```

```
clf.fit(train_x_tfidf, train_y)
sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
sig_clf.fit(train_x_tfidf, train_y)

predict_y = sig_clf.predict_proba(train_x_tfidf)
print('For values of best alpha = ', alpha[best_alpha], "The train log loss i
s:",log_loss(y_train, predict_y, labels=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(cv_x_tfidf)
print('For values of best alpha = ', alpha[best_alpha], "The cross validation
 log loss is:",log_loss(y_cv, predict_y, labels=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(test_x_tfidf)
print('For values of best alpha = ', alpha[best_alpha], "The test log loss i
s:",log_loss(y_test, predict_y, labels=clf.classes_, eps=1e-15))
```

```
for alpha = 1e-06
Log Loss : 1.1804064362204576
for alpha = 1e-05
Log Loss : 1.1085281919164307
for alpha = 0.0001
Log Loss : 1.0910632744198592
for alpha = 0.001
Log Loss : 1.08995108020113
for alpha = 0.01
Log Loss : 1.218874173768796
for alpha = 0.1
Log Loss : 1.6614405426210481
for alpha = 1
Log Loss : 1.7755433249415864
for alpha = 10
Log Loss : 1.788147389341592
for alpha = 100
Log Loss : 1.7895941611279524
```




```
For values of best alpha =  0.001 The train log loss is: 0.7180617442371529
For values of best alpha =  0.001 The cross validation log loss is: 1.089951

08020113
For values of best alpha =  0.001 The test log loss is: 1.0111152336383311
```

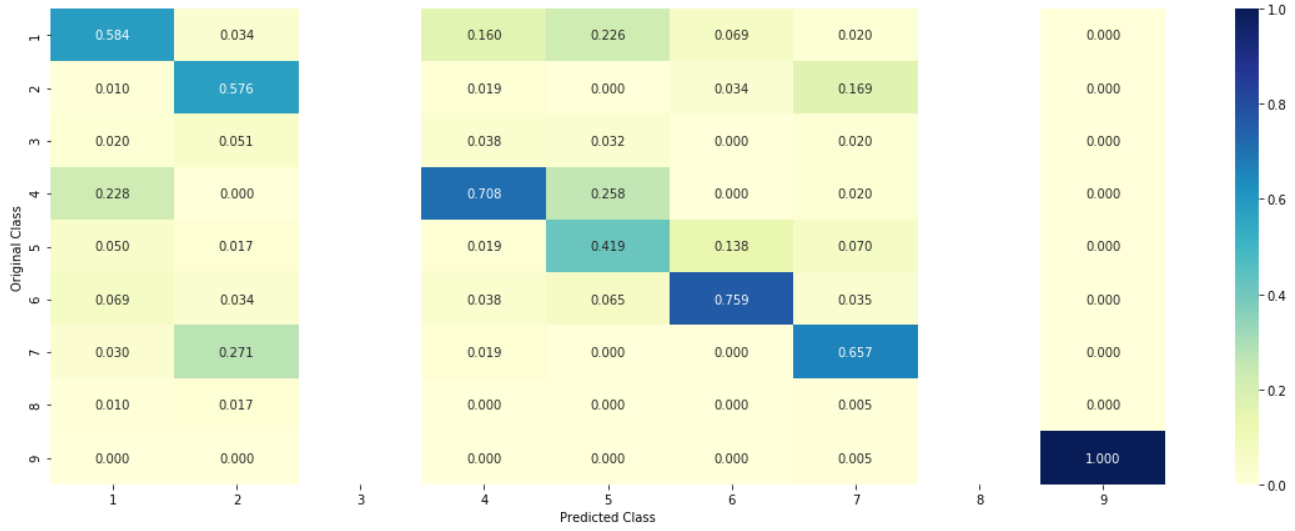
4.3.1.2. Testing the model with best hyper paramters

In [93]:
```
clf = SGDClassifier(class_weight='balanced', alpha=alpha[best_alpha], penalty=
'l2', loss='log', random_state=42)
```

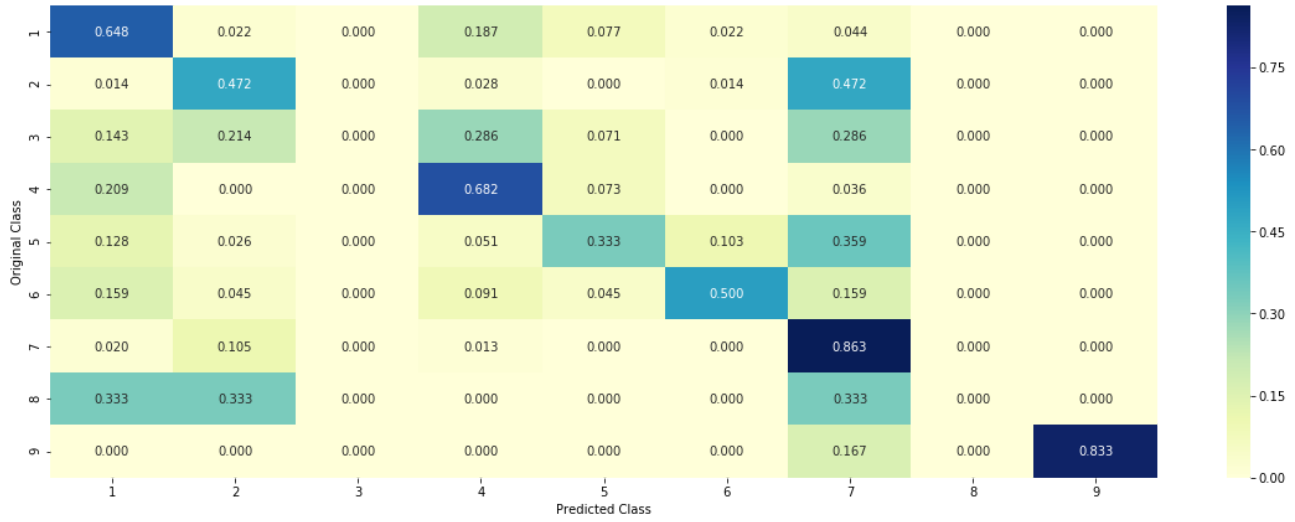
```
predict_and_plot_confusion_matrix(train_x_tfidf, train_y, cv_x_tfidf, cv_y, cl
f)
```

```
Log loss : 1.08995108020113
Number of mis-classified points : 0.3609022556390977
-------------------- Confusion matrix --------------------
```


```
-------------------- Precision matrix (Columm Sum=1) --------------------
```




```
-------------------- Recall matrix (Row sum=1) --------------------
```



4.3.1.3. Feature Importance

In [27]:
```
def get_imp_feature_names(text, indices, removed_ind = []):
    word_present = 0
    tabulte_list = []
    incresingorder_ind = 0
    for i in indices:
        if i < train_gene_feature_tf_idf.shape[1]:
            tabulte_list.append([incresingorder_ind, "Gene", "Yes"])
        elif i< 18:
            tabulte_list.append([incresingorder_ind,"Variation", "Yes"])
        if ((i > 17) & (i not in removed_ind)) :
            word = train_text_features[i]
            yes_no = True if word in text.split() else False
            if yes_no:
                word_present += 1
            tabulte_list.append([incresingorder_ind,train_text_features[i], ye
s_no])
        incresingorder_ind += 1
    print(word_present, "most importent features are present in our query poin
t")
    print("-"*50)
    print("The features that are most importent of the ",predicted_cls[0]," cl
ass:")
    print (tabulate(tabulte_list, headers=["Index",'Feature name', 'Present or
Not']))
```

4.3.1.3.1. Correctly Classified point

In [95]:
```
# from tabulate import tabulate
clf = SGDClassifier(class_weight='balanced', alpha=alpha[best_alpha], penalty=
'l2', loss='log', random_state=42)
clf.fit(train_x_tfidf,train_y)
test_point_index = 1
no_feature = 500
predicted_cls = sig_clf.predict(test_x_tfidf[test_point_index])
print("Predicted Class :", predicted_cls[0])
print("Predicted Class Probabilities:", np.round(sig_clf.predict_proba(test_x_
tfidf[test_point_index]),4))
print("Actual Class :", test_y[test_point_index])
indices = np.argsort(-clf.coef_)[predicted_cls-1][:,:no_feature]
print("-"*50)
get_impfeature_names(indices[0], test_df['TEXT'].iloc[test_point_index],test_d
f['Gene'].iloc[test_point_index],test_df['Variation'].iloc[test_point_index],
no_feature)
```

```
Predicted Class : 7
Predicted Class Probabilities: [[3.370e-02 2.315e-01 1.600e-03 3.730e-02 4.9
90e-02 6.350e-02 5.731e-01
  8.900e-03 4.000e-04]]
Actual Class : 7
--------------------------------------------------
68 Text feature [112] present in test data point [True]
98 Text feature [049] present in test data point [True]
Out of the top  500  features  2 are present in query point
```

4.3.1.3.2. Incorrectly Classified point

In [96]:
```
test_point_index = 15
no_feature = 500
```

```
predicted_cls = sig_clf.predict(test_x_tfidf[test_point_index])
print("Predicted Class :", predicted_cls[0])
print("Predicted Class Probabilities:", np.round(sig_clf.predict_proba(test_x_
tfidf[test_point_index]),4))
print("Actual Class :", test_y[test_point_index])
indices = np.argsort(-clf.coef_)[predicted_cls-1][:,:no_feature]
print("-"*50)
get_impfeature_names(indices[0], test_df['TEXT'].iloc[test_point_index],test_d
f['Gene'].iloc[test_point_index],test_df['Variation'].iloc[test_point_index],
no_feature)
```

```
Predicted Class : 1
Predicted Class Probabilities: [[5.566e-01 6.100e-03 4.000e-04 4.196e-01 3.6
00e-03 2.900e-03 9.100e-03
  1.300e-03 2.000e-04]]
Actual Class : 4
--------------------------------------------------
28 Text feature [0cl] present in test data point [True]
68 Text feature [105] present in test data point [True]
168 Text feature [100] present in test data point [True]
353 Text feature [113] present in test data point [True]
384 Text feature [124] present in test data point [True]
422 Text feature [122] present in test data point [True]
425 Text feature [04] present in test data point [True]
428 Text feature [1040] present in test data point [True]
466 Text feature [015355] present in test data point [True]
492 Text feature [11] present in test data point [True]
Out of the top  500  features  10 are present in query point
```

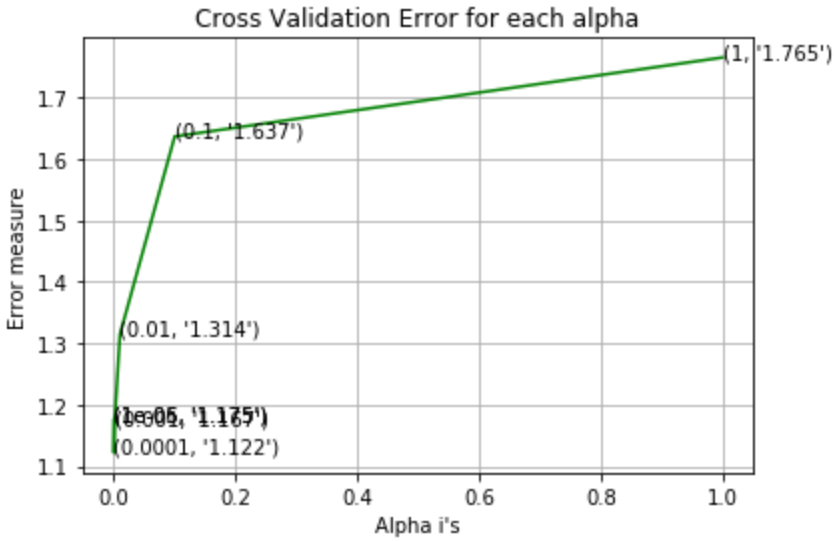
4.3.2. Without Class balancing

4.3.2.1. Hyper paramter tuning

In [97]:
```
alpha = [10 ** x for x in range(-6, 1)]
cv_log_error_array = []
for i in alpha:
    print("for alpha =", i)
    clf = SGDClassifier(alpha=i, penalty='l2', loss='log', random_state=42)
    clf.fit(train_x_tfidf, train_y)
    sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
    sig_clf.fit(train_x_tfidf, train_y)
    sig_clf_probs = sig_clf.predict_proba(cv_x_tfidf)
    cv_log_error_array.append(log_loss(cv_y, sig_clf_probs, labels=clf.classes
_, eps=1e-15))
    print("Log Loss :",log_loss(cv_y, sig_clf_probs))

fig, ax = plt.subplots()
ax.plot(alpha, cv_log_error_array,c='g')
for i, txt in enumerate(np.round(cv_log_error_array,3)):
    ax.annotate((alpha[i],str(txt)), (alpha[i],cv_log_error_array[i]))
plt.grid()
plt.title("Cross Validation Error for each alpha")
plt.xlabel("Alpha i's")
plt.ylabel("Error measure")
plt.show()


best_alpha = np.argmin(cv_log_error_array)
clf = SGDClassifier(alpha=alpha[best_alpha], penalty='l2', loss='log', random_
```

```
state=42)
clf.fit(train_x_tfidf, train_y)
sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
sig_clf.fit(train_x_tfidf, train_y)

predict_y = sig_clf.predict_proba(train_x_tfidf)
print('For values of best alpha = ', alpha[best_alpha], "The train log loss i
s:",log_loss(y_train, predict_y, labels=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(cv_x_tfidf)
print('For values of best alpha = ', alpha[best_alpha], "The cross validation
 log loss is:",log_loss(y_cv, predict_y, labels=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(test_x_tfidf)
print('For values of best alpha = ', alpha[best_alpha], "The test log loss i
s:",log_loss(y_test, predict_y, labels=clf.classes_, eps=1e-15))
```

```
for alpha = 1e-06
Log Loss : 1.1751134304000133
for alpha = 1e-05
Log Loss : 1.1750241099273344
for alpha = 0.0001
Log Loss : 1.1217885230636226
for alpha = 0.001
Log Loss : 1.1669900464282108
for alpha = 0.01
Log Loss : 1.3143267629633155
for alpha = 0.1
Log Loss : 1.636531336093196
for alpha = 1
Log Loss : 1.7649963808816782
```



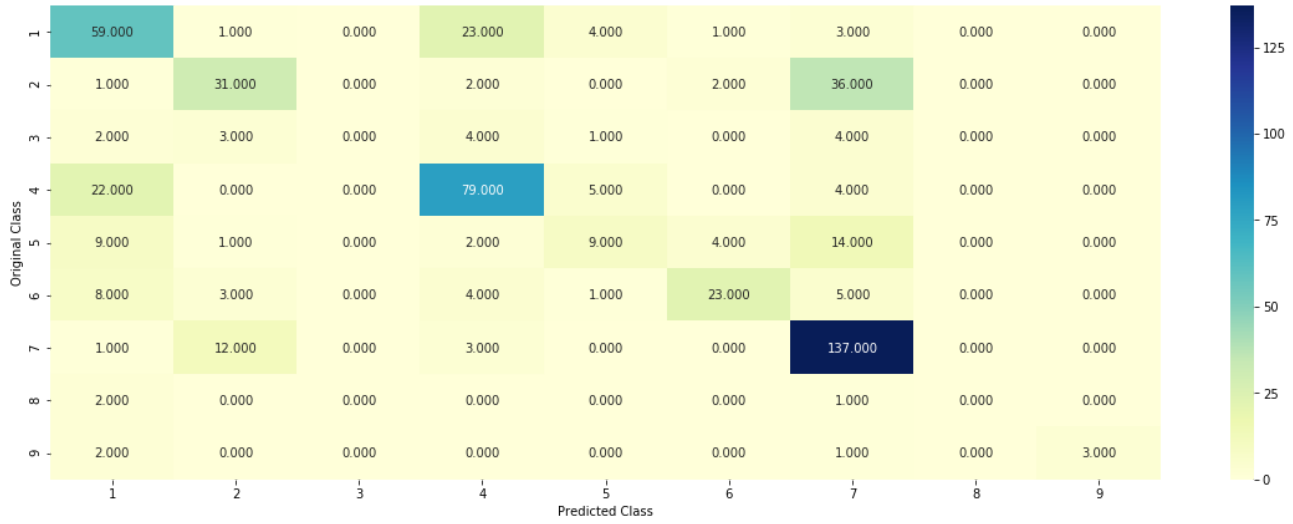

```
For values of best alpha =  0.0001 The train log loss is: 0.4314867971982387
4
For values of best alpha =  0.0001 The cross validation log loss is: 1.12178
85230636226
For values of best alpha =  0.0001 The test log loss is: 1.0153082927903025
```

4.3.2.2. Testing model with best hyper parameters

In [98]:
```
clf = SGDClassifier(alpha=alpha[best_alpha], penalty='l2', loss='log', random_
state=42)
predict_and_plot_confusion_matrix(train_x_tfidf, train_y, cv_x_tfidf, cv_y, cl
f)
```

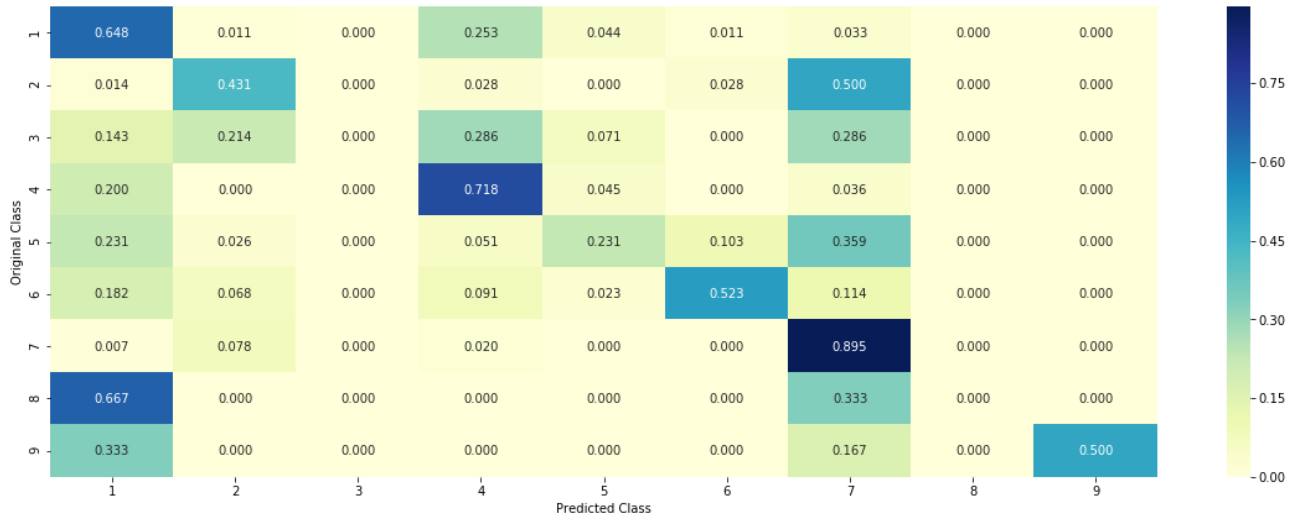
```
Log loss : 1.1217885230636226
```

```
Number of mis-classified points : 0.35902255639097747
-------------------- Confusion matrix --------------------
```




```
-------------------- Precision matrix (Columm Sum=1) --------------------
```


```
-------------------- Recall matrix (Row sum=1) --------------------
```



4.3.2.3. Feature Importance, Correctly Classified point

In [99]:
```
clf = SGDClassifier(alpha=alpha[best_alpha], penalty='l2', loss='log', random_
state=42)
```

```
clf.fit(train_x_tfidf,train_y)
test_point_index = 44
no_feature = 500
predicted_cls = sig_clf.predict(test_x_tfidf[test_point_index])
print("Predicted Class :", predicted_cls[0])
print("Predicted Class Probabilities:", np.round(sig_clf.predict_proba(test_x_
tfidf[test_point_index]),4))
print("Actual Class :", test_y[test_point_index])
indices = np.argsort(-clf.coef_)[predicted_cls-1][:,:no_feature]
print("-"*50)
get_impfeature_names(indices[0], test_df['TEXT'].iloc[test_point_index],test_d
f['Gene'].iloc[test_point_index],test_df['Variation'].iloc[test_point_index],
no_feature)
```

```
Predicted Class : 1
Predicted Class Probabilities: [[7.767e-01 1.240e-02 1.000e-04 1.989e-01 1.2
00e-03 5.800e-03 3.100e-03
  1.800e-03 1.000e-04]]
Actual Class : 1
--------------------------------------------------
163 Text feature [0cl] present in test data point [True]
217 Text feature [100] present in test data point [True]
259 Text feature [105] present in test data point [True]
301 Text feature [04] present in test data point [True]
326 Text feature [12b] present in test data point [True]
345 Text feature [113] present in test data point [True]
371 Text feature [016] present in test data point [True]
418 Text feature [094] present in test data point [True]
465 Text feature [03] present in test data point [True]
471 Text feature [12a] present in test data point [True]
483 Text feature [114] present in test data point [True]
Out of the top  500  features  11 are present in query point
```

4.3.2.4. Feature Importance, Inorrectly Classified point

In [100]:
```
test_point_index = 52
no_feature = 500
predicted_cls = sig_clf.predict(test_x_tfidf[test_point_index])
print("Predicted Class :", predicted_cls[0])
print("Predicted Class Probabilities:", np.round(sig_clf.predict_proba(test_x_
tfidf[test_point_index]),4))
print("Actual Class :", test_y[test_point_index])
indices = np.argsort(-clf.coef_)[predicted_cls-1][:,:no_feature]
print("-"*50)
get_impfeature_names(indices[0], test_df['TEXT'].iloc[test_point_index],test_d
f['Gene'].iloc[test_point_index],test_df['Variation'].iloc[test_point_index],
no_feature)
```

```
Predicted Class : 4
Predicted Class Probabilities: [[0.3745 0.0064 0.0061 0.4514 0.1082 0.0434
0.0073 0.0028 0.    ]]
Actual Class : 5
--------------------------------------------------
166 Text feature [12] present in test data point [True]
258 Text feature [1118a4g] present in test data point [True]
295 Text feature [129] present in test data point [True]

394 Text feature [000] present in test data point [True]
423 Text feature [128] present in test data point [True]
Out of the top  500  features  5 are present in query point
```

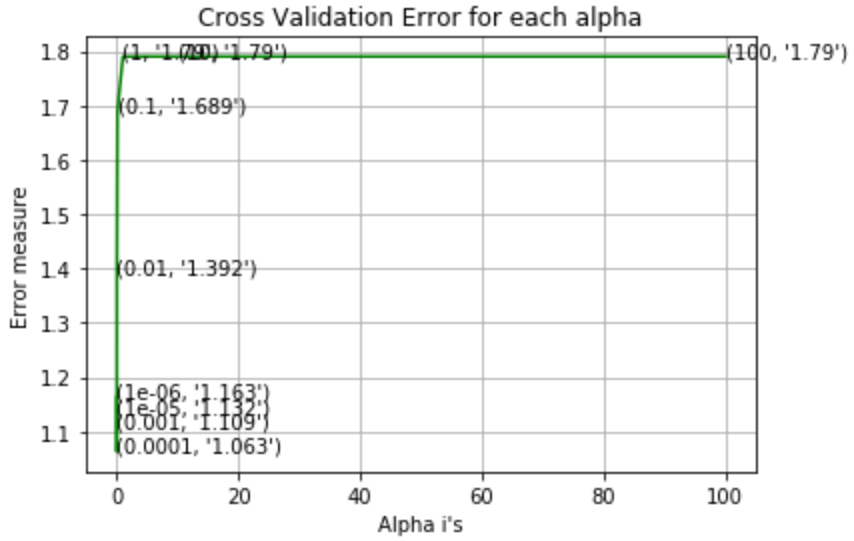
4.4. Linear Support Vector Machines

4.4.1. Hyper paramter tuning

In [101]:
```
alpha = [10 ** x for x in range(-6, 3)]
cv_log_error_array = []
for i in alpha:
    print("for alpha =", i)
    clf = SGDClassifier(class_weight='balanced', alpha=i, penalty='l2', loss=
'hinge', random_state=42)
    clf.fit(train_x_tfidf, train_y)
    sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
    sig_clf.fit(train_x_tfidf, train_y)
    sig_clf_probs = sig_clf.predict_proba(cv_x_tfidf)
    cv_log_error_array.append(log_loss(cv_y, sig_clf_probs, labels=clf.classes
_, eps=1e-15))
    # to avoid rounding error while multiplying probabilites we use log-probab
ility estimates
    print("Log Loss :",log_loss(cv_y, sig_clf_probs))

fig, ax = plt.subplots()
ax.plot(alpha, cv_log_error_array,c='g')
for i, txt in enumerate(np.round(cv_log_error_array,3)):
    ax.annotate((alpha[i],str(txt)), (alpha[i],cv_log_error_array[i]))
plt.grid()
plt.title("Cross Validation Error for each alpha")
plt.xlabel("Alpha i's")
plt.ylabel("Error measure")
plt.show()


best_alpha = np.argmin(cv_log_error_array)
clf = SGDClassifier(class_weight='balanced', alpha=alpha[best_alpha], penalty=
'l2', loss='log', random_state=42)
clf.fit(train_x_tfidf, train_y)
sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
sig_clf.fit(train_x_tfidf, train_y)

predict_y = sig_clf.predict_proba(train_x_tfidf)
print('For values of best alpha = ', alpha[best_alpha], "The train log loss i
s:",log_loss(y_train, predict_y, labels=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(cv_x_tfidf)
print('For values of best alpha = ', alpha[best_alpha], "The cross validation
 log loss is:",log_loss(y_cv, predict_y, labels=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(test_x_tfidf)
print('For values of best alpha = ', alpha[best_alpha], "The test log loss i
s:",log_loss(y_test, predict_y, labels=clf.classes_, eps=1e-15))
```

```
for alpha = 1e-06
Log Loss : 1.163387465505117
for alpha = 1e-05
Log Loss : 1.132454926401396
for alpha = 0.0001
Log Loss : 1.0625529361271555
for alpha = 0.001
Log Loss : 1.1092274702735214
for alpha = 0.01
Log Loss : 1.3916864418321546
```

```
for alpha = 0.1
Log Loss : 1.6892584761892835
for alpha = 1
Log Loss : 1.7899545713592326
for alpha = 10
Log Loss : 1.7899545172792628
for alpha = 100
Log Loss : 1.7899585860738598
```

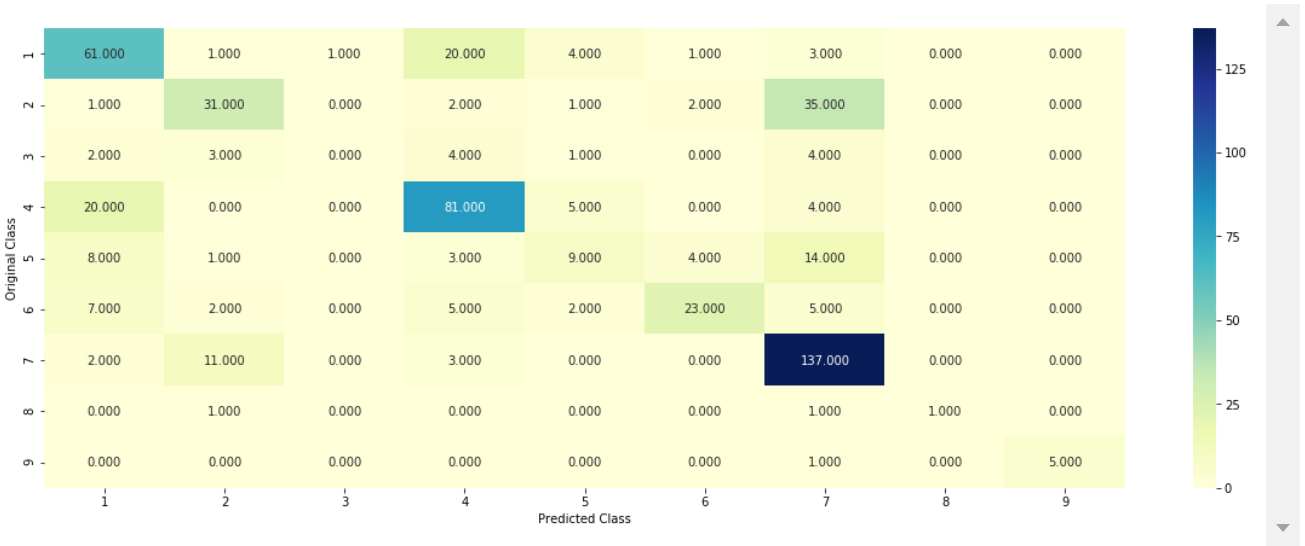


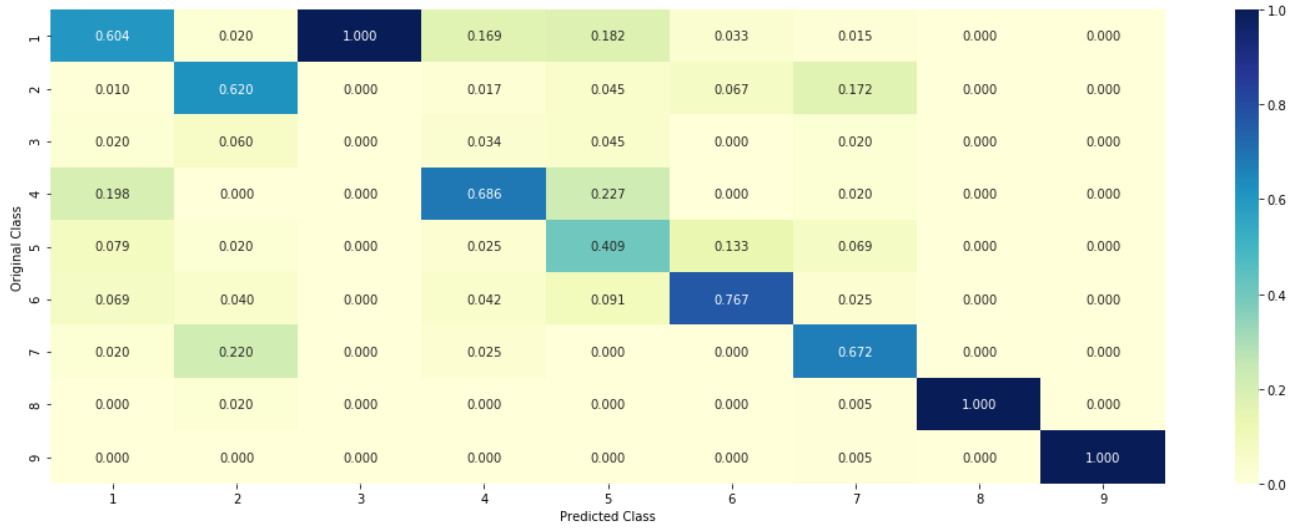

```
For values of best alpha =  0.0001 The train log loss is: 0.4376904723619819
4
For values of best alpha =  0.0001 The cross validation log loss is: 1.09106
32744198592
For values of best alpha =  0.0001 The test log loss is: 0.9883099124820276
```

4.4.2. Testing model with best hyper parameters

In [102]:
```
# clf = SVC(C=alpha[best_alpha],kernel='linear',probability=True, class_weight
='balanced')
clf = SGDClassifier(alpha=alpha[best_alpha], penalty='l2', loss='hinge', rando
m_state=42,class_weight='balanced')
predict_and_plot_confusion_matrix(train_x_tfidf, train_y,cv_x_tfidf,cv_y, clf)
```

```
Log loss : 1.0625529361271555
Number of mis-classified points : 0.3458646616541353
-------------------- Confusion matrix --------------------
```





---------- Recall matrix (Row sum=1) ----------

4.3.3. Feature Importance

4.3.3.1. For Correctly classified point

In [103]:
```
clf = SGDClassifier(alpha=alpha[best_alpha], penalty='l2', loss='hinge', rando
m_state=42)
clf.fit(train_x_tfidf,train_y)
test_point_index = 1
# test_point_index = 100
no_feature = 500
```

```
predicted_cls = sig_clf.predict(test_x_tfidf[test_point_index])
print("Predicted Class :", predicted_cls[0])
print("Predicted Class Probabilities:", np.round(sig_clf.predict_proba(test_x_
tfidf[test_point_index]),4))
print("Actual Class :", test_y[test_point_index])
indices = np.argsort(-clf.coef_)[predicted_cls-1][:,:no_feature]
print("-"*50)
get_impfeature_names(indices[0], test_df['TEXT'].iloc[test_point_index],test_d
f['Gene'].iloc[test_point_index],test_df['Variation'].iloc[test_point_index],
no_feature)
```

```
Predicted Class : 7
Predicted Class Probabilities: [[2.470e-02 1.545e-01 5.000e-04 3.860e-02 3.3
00e-02 2.240e-02 7.228e-01
  3.200e-03 4.000e-04]]
Actual Class : 7
--------------------------------------------------
Out of the top  500  features  0 are present in query point
```

4.3.3.2. For Incorrectly classified point

In [104]:
```
test_point_index = 200
# test_point_index = 100
no_feature = 500
predicted_cls = sig_clf.predict(test_x_tfidf[test_point_index])
print("Predicted Class :", predicted_cls[0])
print("Predicted Class Probabilities:", np.round(sig_clf.predict_proba(test_x_
tfidf[test_point_index]),4))
print("Actual Class :", test_y[test_point_index])
indices = np.argsort(-clf.coef_)[predicted_cls-1][:,:no_feature]
print("-"*50)
get_impfeature_names(indices[0], test_df['TEXT'].iloc[test_point_index],test_d
f['Gene'].iloc[test_point_index],test_df['Variation'].iloc[test_point_index],
no_feature)
```

```
Predicted Class : 5
Predicted Class Probabilities: [[0.0415 0.0082 0.0414 0.1895 0.6444 0.0662
0.003  0.0041 0.0016]]
Actual Class : 5
--------------------------------------------------
246 Text feature [1142] present in test data point [True]
317 Text feature [120] present in test data point [True]
394 Text feature [11] present in test data point [True]
418 Text feature [1158] present in test data point [True]
Out of the top  500  features  4 are present in query point
```

4.5 Random Forest Classifier

4.5.1. Hyper paramter tuning (With One hot Encoding)

In [105]:
```
alpha = [100,200,500,1000,2000]
max_depth = [5, 10]
cv_log_error_array = []
for i in alpha:
    for j in max_depth:
        print("for n_estimators =", i,"and max depth = ", j)
        clf = RandomForestClassifier(n_estimators=i, criterion='gini', max_dep
```

```
th=j, random_state=42, n_jobs=-1)
        clf.fit(train_x_tfidf, train_y)
        sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
        sig_clf.fit(train_x_tfidf, train_y)
        sig_clf_probs = sig_clf.predict_proba(cv_x_tfidf)
        cv_log_error_array.append(log_loss(cv_y, sig_clf_probs, labels=clf.cla
sses_, eps=1e-15))
        print("Log Loss :",log_loss(cv_y, sig_clf_probs))

'''fig, ax = plt.subplots()
features = np.dot(np.array(alpha)[:,None],np.array(max_depth)[None]).ravel()
ax.plot(features, cv_log_error_array,c='g')
for i, txt in enumerate(np.round(cv_log_error_array,3)):
    ax.annotate((alpha[int(i/2)],max_depth[int(i%2)],str(txt)), (features[i],c
v_log_error_array[i]))
plt.grid()
plt.title("Cross Validation Error for each alpha")
plt.xlabel("Alpha i's")
plt.ylabel("Error measure")
plt.show()
'''

best_alpha = np.argmin(cv_log_error_array)
clf = RandomForestClassifier(n_estimators=alpha[int(best_alpha/2)], criterion=
'gini', max_depth=max_depth[int(best_alpha%2)], random_state=42, n_jobs=-1)
clf.fit(train_x_tfidf, train_y)
sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
sig_clf.fit(train_x_tfidf, train_y)

predict_y = sig_clf.predict_proba(train_x_tfidf)
print('For values of best estimator = ', alpha[int(best_alpha/2)], "The train
 log loss is:",log_loss(y_train, predict_y, labels=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(cv_x_tfidf)
print('For values of best estimator = ', alpha[int(best_alpha/2)], "The cross
 validation log loss is:",log_loss(y_cv, predict_y, labels=clf.classes_, eps=1
e-15))
predict_y = sig_clf.predict_proba(test_x_tfidf)
print('For values of best estimator = ', alpha[int(best_alpha/2)], "The test l
og loss is:",log_loss(y_test, predict_y, labels=clf.classes_, eps=1e-15))
```

```
for n_estimators = 100 and max depth =  5
Log Loss : 1.2728169642628346
for n_estimators = 100 and max depth =  10
Log Loss : 1.280984431455046
for n_estimators = 200 and max depth =  5
Log Loss : 1.25590926045607
for n_estimators = 200 and max depth =  10
Log Loss : 1.2726325543868626
for n_estimators = 500 and max depth =  5
Log Loss : 1.248748312713173

for n_estimators = 500 and max depth =  10
Log Loss : 1.2674566273109473
for n_estimators = 1000 and max depth =  5
Log Loss : 1.2409521762479696
for n_estimators = 1000 and max depth =  10
Log Loss : 1.2675195206842271
for n_estimators = 2000 and max depth =  5
Log Loss : 1.2367586809369744
for n_estimators = 2000 and max depth =  10
Log Loss : 1.2643144603227454
For values of best estimator =  2000 The train log loss is: 0.82490524497155
3
```

```
For values of best estimator =  2000 The cross validation log loss is: 1.236
7586809369746
For values of best estimator =  2000 The test log loss is: 1.146367482696518
```

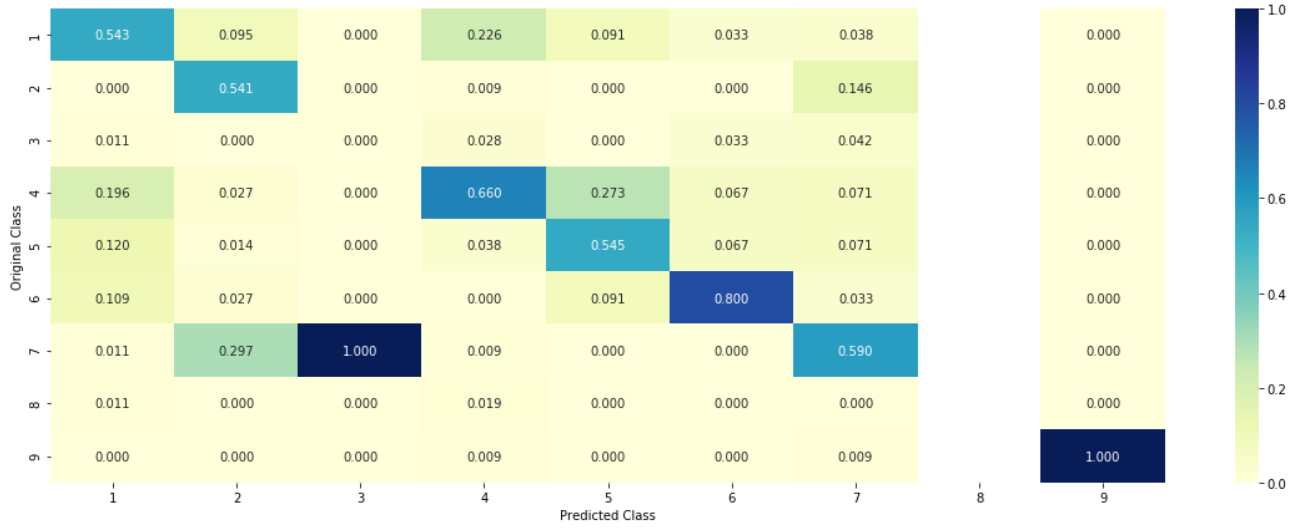
4.5.2. Testing model with best hyper parameters (One Hot Encoding)

In [ ]:
```
# best estimator
# n_estimators = 2000 and max depth =  5
```

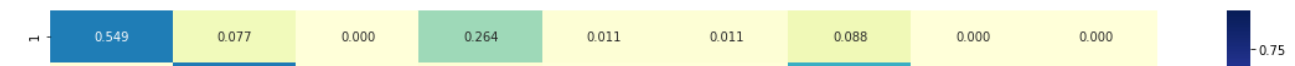
In [124]:
```
clf = RandomForestClassifier(n_estimators=alpha[int(best_alpha/2)], criterion=
'gini', max_depth=max_depth[int(best_alpha%2)], random_state=42, n_jobs=-1)
predict_and_plot_confusion_matrix(train_x_tfidf, train_y,cv_x_tfidf,cv_y, clf)
```

```
Log loss : 1.1590757162450653
Number of mis-classified points : 0.40225563909774437
-------------------- Confusion matrix --------------------
```


```
-------------------- Precision matrix (Columm Sum=1) --------------------
```




```
-------------------- Recall matrix (Row sum=1) --------------------
```



4.5.3. Feature Importance

4.5.3.1. Correctly Classified point

In [128]:
```
# test_point_index = 10
clf = RandomForestClassifier(n_estimators=alpha[int(best_alpha/2)], criterion=
'gini', max_depth=max_depth[int(best_alpha%2)], random_state=42, n_jobs=-1)
clf.fit(train_x_tfidf, train_y)
sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
sig_clf.fit(train_x_tfidf, train_y)

test_point_index = 1
no_feature = 100
predicted_cls = sig_clf.predict(test_x_tfidf[test_point_index])
print("Predicted Class :", predicted_cls[0])
print("Predicted Class Probabilities:", np.round(sig_clf.predict_proba(test_x_
tfidf[test_point_index]),4))
print("Actual Class :", test_y[test_point_index])
indices = np.argsort(-clf.feature_importances_)
print("-"*50)
get_impfeature_names(indices[:no_feature], test_df['TEXT'].iloc[test_point_ind
ex],test_df['Gene'].iloc[test_point_index],test_df['Variation'].iloc[test_poin
t_index], no_feature)
```

```
Predicted Class : 4
Predicted Class Probabilities: [[0.2905 0.0447 0.0177 0.4896 0.0476 0.0426
0.05   0.0079 0.0094]]
Actual Class : 6
--------------------------------------------------
0 Text feature [kinase] present in test data point [True]
3 Text feature [phosphorylation] present in test data point [True]
4 Text feature [inhibitors] present in test data point [True]

7 Text feature [activation] present in test data point [True]
8 Text feature [suppressor] present in test data point [True]
10 Text feature [function] present in test data point [True]
11 Text feature [loss] present in test data point [True]
15 Text feature [defective] present in test data point [True]
```

```
20 Text feature [missense] present in test data point [True]
22 Text feature [downstream] present in test data point [True]
24 Text feature [treated] present in test data point [True]
26 Text feature [repair] present in test data point [True]
27 Text feature [protein] present in test data point [True]
28 Text feature [cells] present in test data point [True]
31 Text feature [stability] present in test data point [True]
32 Text feature [functional] present in test data point [True]
34 Text feature [yeast] present in test data point [True]
40 Text feature [expressing] present in test data point [True]
41 Text feature [variants] present in test data point [True]
43 Text feature [brca1] present in test data point [True]
47 Text feature [dose] present in test data point [True]
50 Text feature [patients] present in test data point [True]
54 Text feature [cell] present in test data point [True]
55 Text feature [inhibited] present in test data point [True]
58 Text feature [clinical] present in test data point [True]
61 Text feature [serum] present in test data point [True]
62 Text feature [inhibition] present in test data point [True]
63 Text feature [atp] present in test data point [True]
69 Text feature [dna] present in test data point [True]
73 Text feature [proteins] present in test data point [True]
74 Text feature [harboring] present in test data point [True]
76 Text feature [unstable] present in test data point [True]
78 Text feature [response] present in test data point [True]
81 Text feature [potential] present in test data point [True]
84 Text feature [null] present in test data point [True]
86 Text feature [pathway] present in test data point [True]
91 Text feature [functions] present in test data point [True]
93 Text feature [defect] present in test data point [True]
94 Text feature [patient] present in test data point [True]
95 Text feature [ability] present in test data point [True]
98 Text feature [phosphatase] present in test data point [True]
Out of the top  100  features  41 are present in query point
```

4.5.3.2. Inorrectly Classified point

In [129]:
```
test_point_index = 1
no_feature = 100
predicted_cls = sig_clf.predict(test_x_tfidf[test_point_index])
print("Predicted Class :", predicted_cls[0])
print("Predicted Class Probabilities:", np.round(sig_clf.predict_proba(test_x_
tfidf[test_point_index]),4))
print("Actual Class :", test_y[test_point_index])
indices = np.argsort(-clf.feature_importances_)
print("-"*50)
get_impfeature_names(indices[:no_feature], test_df['TEXT'].iloc[test_point_ind
ex],test_df['Gene'].iloc[test_point_index],test_df['Variation'].iloc[test_poin
t_index], no_feature)
```

```
Predicted Class : 4
Predicted Class Probabilities: [[0.2905 0.0447 0.0177 0.4896 0.0476 0.0426
0.05   0.0079 0.0094]]
Actual Class : 6
--------------------------------------------------
0 Text feature [kinase] present in test data point [True]
3 Text feature [phosphorylation] present in test data point [True]

4 Text feature [inhibitors] present in test data point [True]
7 Text feature [activation] present in test data point [True]
8 Text feature [suppressor] present in test data point [True]
```

```
10 Text feature [function] present in test data point [True]
11 Text feature [loss] present in test data point [True]
15 Text feature [defective] present in test data point [True]
20 Text feature [missense] present in test data point [True]
22 Text feature [downstream] present in test data point [True]
24 Text feature [treated] present in test data point [True]
26 Text feature [repair] present in test data point [True]
27 Text feature [protein] present in test data point [True]
28 Text feature [cells] present in test data point [True]
31 Text feature [stability] present in test data point [True]
32 Text feature [functional] present in test data point [True]
34 Text feature [yeast] present in test data point [True]
40 Text feature [expressing] present in test data point [True]
41 Text feature [variants] present in test data point [True]
43 Text feature [brca1] present in test data point [True]
47 Text feature [dose] present in test data point [True]
50 Text feature [patients] present in test data point [True]
54 Text feature [cell] present in test data point [True]
55 Text feature [inhibited] present in test data point [True]
58 Text feature [clinical] present in test data point [True]
61 Text feature [serum] present in test data point [True]
62 Text feature [inhibition] present in test data point [True]
63 Text feature [atp] present in test data point [True]
69 Text feature [dna] present in test data point [True]
73 Text feature [proteins] present in test data point [True]
74 Text feature [harboring] present in test data point [True]
76 Text feature [unstable] present in test data point [True]
78 Text feature [response] present in test data point [True]
81 Text feature [potential] present in test data point [True]
84 Text feature [null] present in test data point [True]
86 Text feature [pathway] present in test data point [True]
91 Text feature [functions] present in test data point [True]
93 Text feature [defect] present in test data point [True]
94 Text feature [patient] present in test data point [True]
95 Text feature [ability] present in test data point [True]
98 Text feature [phosphatase] present in test data point [True]
Out of the top  100  features  41 are present in query point
```

4.5.3. Hyper paramter tuning (With Response Coding)

In [130]:
```
alpha = [10,50,100,200,500,1000]
max_depth = [2,3,5,10]
cv_log_error_array = []
for i in alpha:
    for j in max_depth:
        print("for n_estimators =", i,"and max depth = ", j)
        clf = RandomForestClassifier(n_estimators=i, criterion='gini', max_dep
th=j, random_state=42, n_jobs=-1)
        clf.fit(train_x_responseCoding, train_y)
        sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
        sig_clf.fit(train_x_responseCoding, train_y)
        sig_clf_probs = sig_clf.predict_proba(cv_x_responseCoding)
        cv_log_error_array.append(log_loss(cv_y, sig_clf_probs, labels=clf.cla
sses_, eps=1e-15))
        print("Log Loss :",log_loss(cv_y, sig_clf_probs))
'''
fig, ax = plt.subplots()
features = np.dot(np.array(alpha)[:,None],np.array(max_depth)[None]).ravel()
ax.plot(features, cv_log_error_array,c='g')
for i, txt in enumerate(np.round(cv_log_error_array,3)):
```

```
    ax.annotate((alpha[int(i/4)],max_depth[int(i%4)],str(txt)), (features[i],c
v_log_error_array[i]))
plt.grid()
plt.title("Cross Validation Error for each alpha")
plt.xlabel("Alpha i's")
plt.ylabel("Error measure")
plt.show()
'''

best_alpha = np.argmin(cv_log_error_array)
clf = RandomForestClassifier(n_estimators=alpha[int(best_alpha/4)], criterion=
'gini', max_depth=max_depth[int(best_alpha%4)], random_state=42, n_jobs=-1)
clf.fit(train_x_responseCoding, train_y)
sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
sig_clf.fit(train_x_responseCoding, train_y)

predict_y = sig_clf.predict_proba(train_x_responseCoding)
print('For values of best alpha = ', alpha[int(best_alpha/4)], "The train log
 loss is:",log_loss(y_train, predict_y, labels=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(cv_x_responseCoding)
print('For values of best alpha = ', alpha[int(best_alpha/4)], "The cross vali
dation log loss is:",log_loss(y_cv, predict_y, labels=clf.classes_, eps=1e-15
))
predict_y = sig_clf.predict_proba(test_x_responseCoding)
print('For values of best alpha = ', alpha[int(best_alpha/4)], "The test log l
oss is:",log_loss(y_test, predict_y, labels=clf.classes_, eps=1e-15))
```

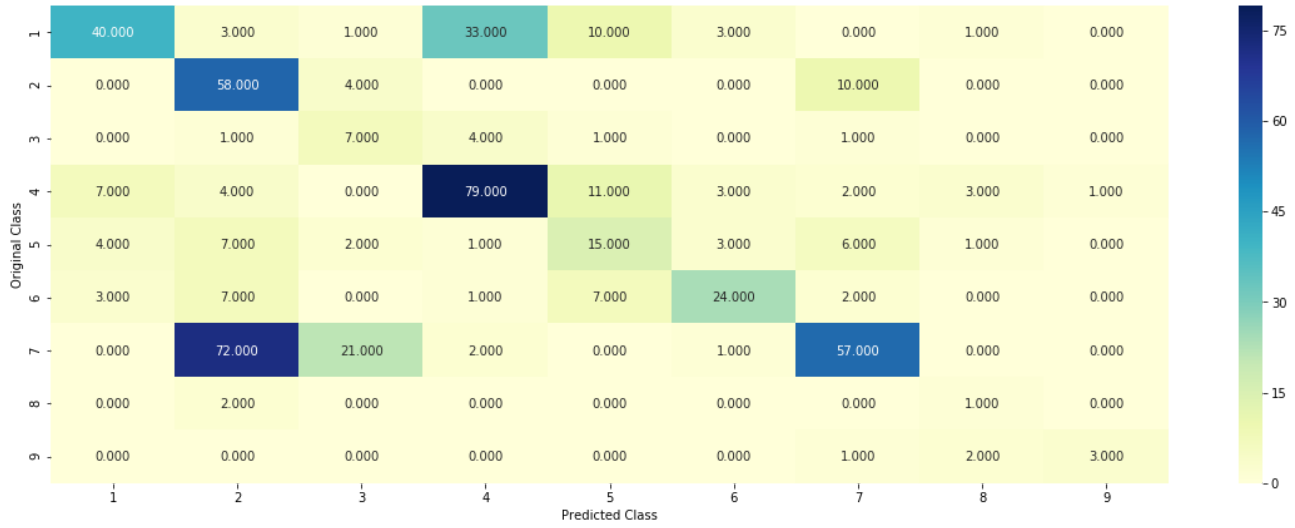
```
for n_estimators = 10 and max depth =  2
Log Loss : 2.1151344064251076
for n_estimators = 10 and max depth =  3
Log Loss : 1.716242550451281
for n_estimators = 10 and max depth =  5
Log Loss : 1.5862176282286726
for n_estimators = 10 and max depth =  10
Log Loss : 1.994604606326167
for n_estimators = 50 and max depth =  2
Log Loss : 1.6594925673149263
for n_estimators = 50 and max depth =  3
Log Loss : 1.4403344073229012
for n_estimators = 50 and max depth =  5
Log Loss : 1.2833746690178165
for n_estimators = 50 and max depth =  10
Log Loss : 1.6452209282780308
for n_estimators = 100 and max depth =  2
Log Loss : 1.5462062002542614
for n_estimators = 100 and max depth =  3
Log Loss : 1.5071402439897683
for n_estimators = 100 and max depth =  5
Log Loss : 1.2746363753066015
for n_estimators = 100 and max depth =  10
Log Loss : 1.746789062518738
for n_estimators = 200 and max depth =  2
Log Loss : 1.605821314531331
for n_estimators = 200 and max depth =  3
Log Loss : 1.518617312612672
for n_estimators = 200 and max depth =  5
Log Loss : 1.3364513886890783
for n_estimators = 200 and max depth =  10
Log Loss : 1.7647600791893772
for n_estimators = 500 and max depth =  2
Log Loss : 1.6449986748467795
for n_estimators = 500 and max depth =  3
```

```
Log Loss : 1.5541090105367301
for n_estimators = 500 and max depth =  5
Log Loss : 1.411373687189553
for n_estimators = 500 and max depth =  10
Log Loss : 1.7727905019734476
for n_estimators = 1000 and max depth =  2
Log Loss : 1.6185952314694714
for n_estimators = 1000 and max depth =  3
Log Loss : 1.5804073786496704
for n_estimators = 1000 and max depth =  5
Log Loss : 1.3847955798783924
for n_estimators = 1000 and max depth =  10
Log Loss : 1.7497495285032372
For values of best alpha =  100 The train log loss is: 0.057044230860352346
For values of best alpha =  100 The cross validation log loss is: 1.27463637
53066012
For values of best alpha =  100 The test log loss is: 1.3267204830924855
```

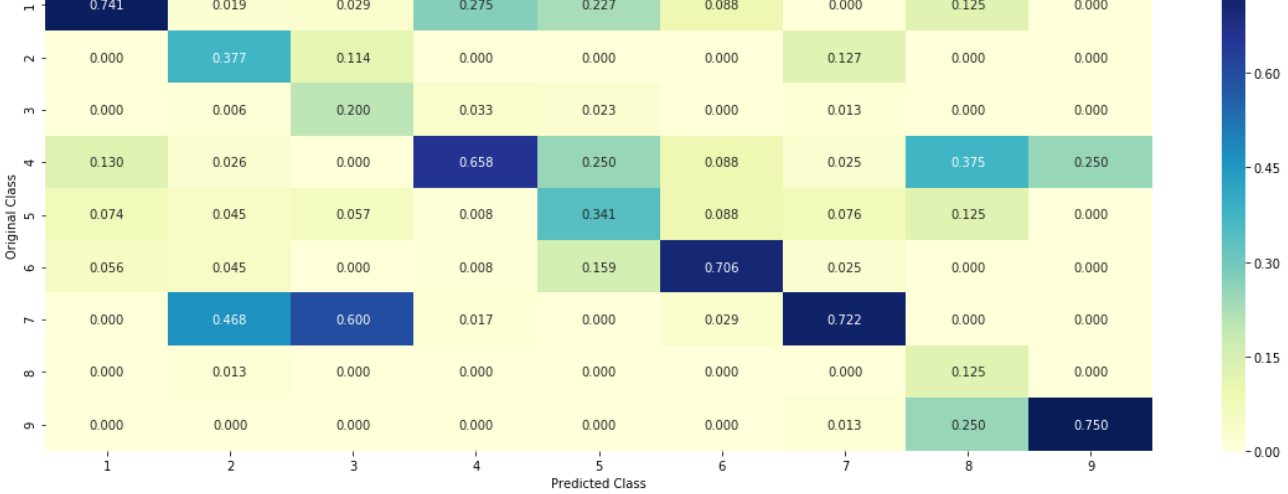
4.5.4. Testing model with best hyper parameters (Response Coding)

In [131]:
```
clf = RandomForestClassifier(max_depth=max_depth[int(best_alpha%4)], n_estimat
ors=alpha[int(best_alpha/4)], criterion='gini', max_features='auto',random_sta
te=42)
predict_and_plot_confusion_matrix(train_x_responseCoding, train_y,cv_x_respons
eCoding,cv_y, clf)
```

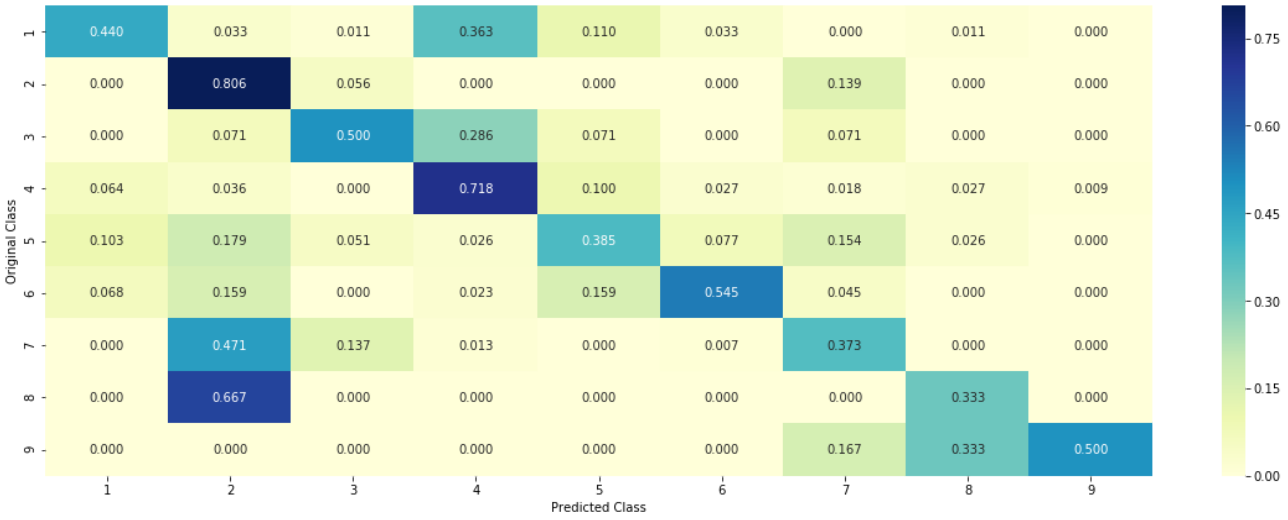
```
Log loss : 1.2746363753066012
Number of mis-classified points : 0.46616541353383456
-------------------- Confusion matrix --------------------
```




```
-------------------- Precision matrix (Columm Sum=1) --------------------
```



---------- Recall matrix (Row sum=1) ----------



4.5.5. Feature Importance

4.5.5.1. Correctly Classified point

In [136]:
```
clf = RandomForestClassifier(n_estimators=alpha[int(best_alpha/4)], criterion=
'gini', max_depth=max_depth[int(best_alpha%4)], random_state=42, n_jobs=-1)
clf.fit(train_x_responseCoding, train_y)
sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
sig_clf.fit(train_x_responseCoding, train_y)


test_point_index = 100
no_feature = 27
predicted_cls = sig_clf.predict(test_x_responseCoding[test_point_index].reshap
e(1,-1))
print("Predicted Class :", predicted_cls[0])
print("Predicted Class Probabilities:", np.round(sig_clf.predict_proba(test_x_
responseCoding[test_point_index].reshape(1,-1)),4))
print("Actual Class :", test_y[test_point_index])
indices = np.argsort(-clf.feature_importances_)
print("-"*50)
for i in indices:
    if i<9:
```

```
        print("Gene is important feature")
    elif i<18:
        print("Variation is important feature")
    else:
        print("Text is important feature")
```

```
Predicted Class : 4
Predicted Class Probabilities: [[0.166  0.0193 0.0997 0.5906 0.0355 0.0364
0.0051 0.0201 0.0273]]
Actual Class : 4
--------------------------------------------------
Variation is important feature
Variation is important feature
Variation is important feature
Variation is important feature
Gene is important feature
Variation is important feature
Variation is important feature
Text is important feature
Text is important feature
Gene is important feature
Text is important feature
Text is important feature
Text is important feature
Variation is important feature
Gene is important feature
Gene is important feature
Text is important feature
Gene is important feature
Variation is important feature
Gene is important feature
Variation is important feature
Text is important feature
Text is important feature
Gene is important feature
Text is important feature
Gene is important feature
Gene is important feature
```

4.5.5.2. Incorrectly Classified point

In [135]:
```
test_point_index = 22
predicted_cls = sig_clf.predict(test_x_responseCoding[test_point_index].reshap
e(1,-1))
print("Predicted Class :", predicted_cls[0])
print("Predicted Class Probabilities:", np.round(sig_clf.predict_proba(test_x_
responseCoding[test_point_index].reshape(1,-1)),4))
print("Actual Class :", test_y[test_point_index])
indices = np.argsort(-clf.feature_importances_)
print("-"*50)
for i in indices:
    if i<9:
        print("Gene is important feature")
    elif i<18:
        print("Variation is important feature")
    else:
        print("Text is important feature")
```

```
Predicted Class : 4

Predicted Class Probabilities: [[0.2827 0.0164 0.106  0.3615 0.0945 0.0972
```

```
0.0053 0.0172 0.0193]]
Actual Class : 6
--------------------------------------------------
Variation is important feature
Variation is important feature
Variation is important feature
Variation is important feature
Gene is important feature
Variation is important feature
Variation is important feature
Text is important feature
Text is important feature
Gene is important feature
Text is important feature
Text is important feature
Text is important feature
Variation is important feature
Gene is important feature
Gene is important feature
Text is important feature
Gene is important feature
Variation is important feature
Gene is important feature
Variation is important feature
Text is important feature
Text is important feature
Gene is important feature
Text is important feature
Gene is important feature
Gene is important feature
```

4.7 Stack the models

4.7.1 testing with hyper parameter tuning

In [106]:

```
clf1 = SGDClassifier(alpha=0.001, penalty='l2', loss='log', class_weight='bala
nced', random_state=0)
clf1.fit(train_x_tfidf, train_y)
sig_clf1 = CalibratedClassifierCV(clf1, method="sigmoid")

clf2 = SGDClassifier(alpha=1, penalty='l2', loss='hinge', class_weight='balanc
ed', random_state=0)
clf2.fit(train_x_tfidf, train_y)
sig_clf2 = CalibratedClassifierCV(clf2, method="sigmoid")


clf3 = MultinomialNB(alpha=0.001)
clf3.fit(train_x_tfidf, train_y)
sig_clf3 = CalibratedClassifierCV(clf3, method="sigmoid")

sig_clf1.fit(train_x_tfidf, train_y)
print("Logistic Regression :  Log Loss: %0.2f" % (log_loss(cv_y, sig_clf1.pred
ict_proba(cv_x_tfidf))))
sig_clf2.fit(train_x_tfidf, train_y)
print("Support vector machines : Log Loss: %0.2f" % (log_loss(cv_y, sig_clf2.p
redict_proba(cv_x_tfidf))))
sig_clf3.fit(train_x_tfidf, train_y)
print("Naive Bayes : Log Loss: %0.2f" % (log_loss(cv_y, sig_clf3.predict_proba
```

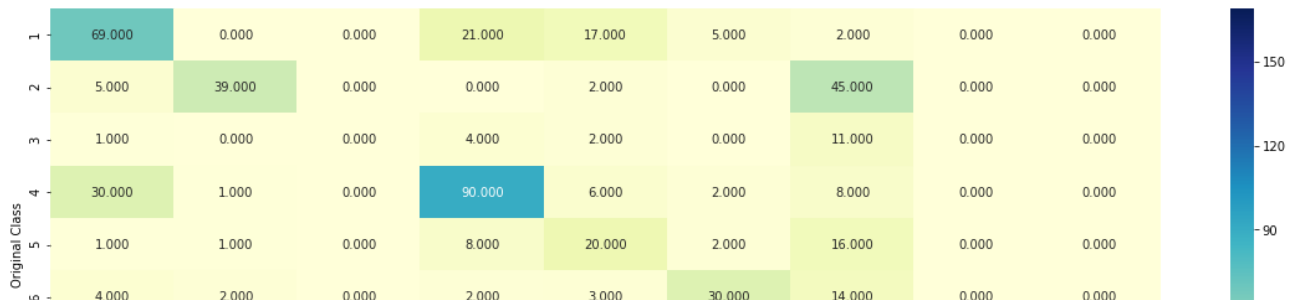
```
(cv_x_tfidf))))
print("-"*50)
alpha = [0.0001,0.001,0.01,0.1,1,10]
best_alpha = 999
for i in alpha:
    lr = LogisticRegression(C=i)
    sclf = StackingClassifier(classifiers=[sig_clf1, sig_clf2, sig_clf3], meta
_classifier=lr, use_probas=True)
    sclf.fit(train_x_tfidf, train_y)
    print("Stacking Classifer : for the value of alpha: %f Log Loss: %0.3f" %
(i, log_loss(cv_y, sclf.predict_proba(cv_x_tfidf))))
    log_error =log_loss(cv_y, sclf.predict_proba(cv_x_tfidf))
    if best_alpha > log_error:
        best_alpha = log_error
```

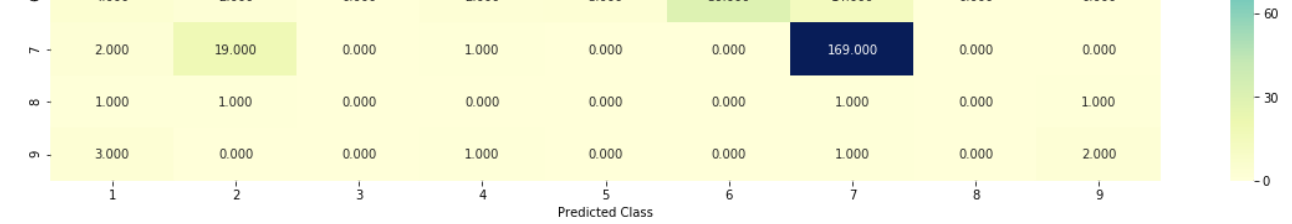
```
Logistic Regression :  Log Loss: 1.09
Support vector machines : Log Loss: 1.79
Naive Bayes : Log Loss: 1.22
--------------------------------------------------
Stacking Classifer : for the value of alpha: 0.000100 Log Loss: 2.178
Stacking Classifer : for the value of alpha: 0.001000 Log Loss: 2.032
Stacking Classifer : for the value of alpha: 0.010000 Log Loss: 1.502
Stacking Classifer : for the value of alpha: 0.100000 Log Loss: 1.181
Stacking Classifer : for the value of alpha: 1.000000 Log Loss: 1.432
Stacking Classifer : for the value of alpha: 10.000000 Log Loss: 1.935
```

4.7.2 testing the model with the best hyper parameters

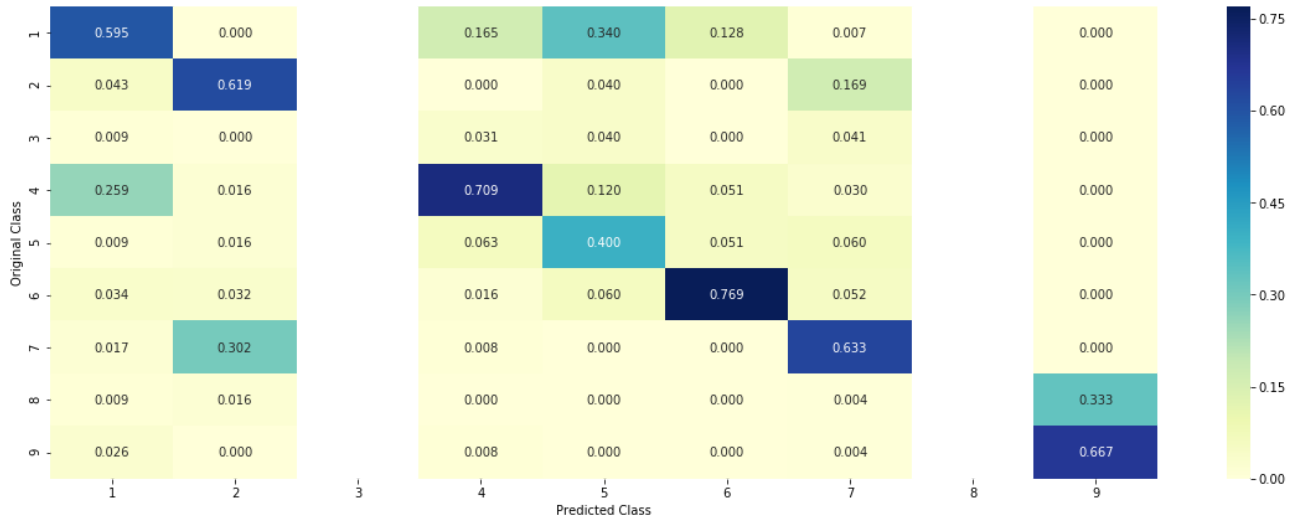
In [107]:
```
lr = LogisticRegression(C=0.1)
sclf = StackingClassifier(classifiers=[sig_clf1, sig_clf2, sig_clf3], meta_cla
ssifier=lr, use_probas=True)
sclf.fit(train_x_tfidf, train_y)

log_error = log_loss(train_y, sclf.predict_proba(train_x_tfidf))
print("Log loss (train) on the stacking classifier :",log_error)

log_error = log_loss(cv_y, sclf.predict_proba(cv_x_tfidf))
print("Log loss (CV) on the stacking classifier :",log_error)

log_error = log_loss(test_y, sclf.predict_proba(test_x_tfidf))
print("Log loss (test) on the stacking classifier :",log_error)

print("Number of missclassified point :", np.count_nonzero((sclf.predict(test_
x_tfidf)- test_y))/test_y.shape[0])
plot_confusion_matrix(test_y=test_y, predict_y=sclf.predict(test_x_tfidf))
```

```
Log loss (train) on the stacking classifier : 0.5308058438092175
Log loss (CV) on the stacking classifier : 1.180917718265983
Log loss (test) on the stacking classifier : 1.1205467095264456
Number of missclassified point : 0.3699248120300752
-------------------- Confusion matrix --------------------
```

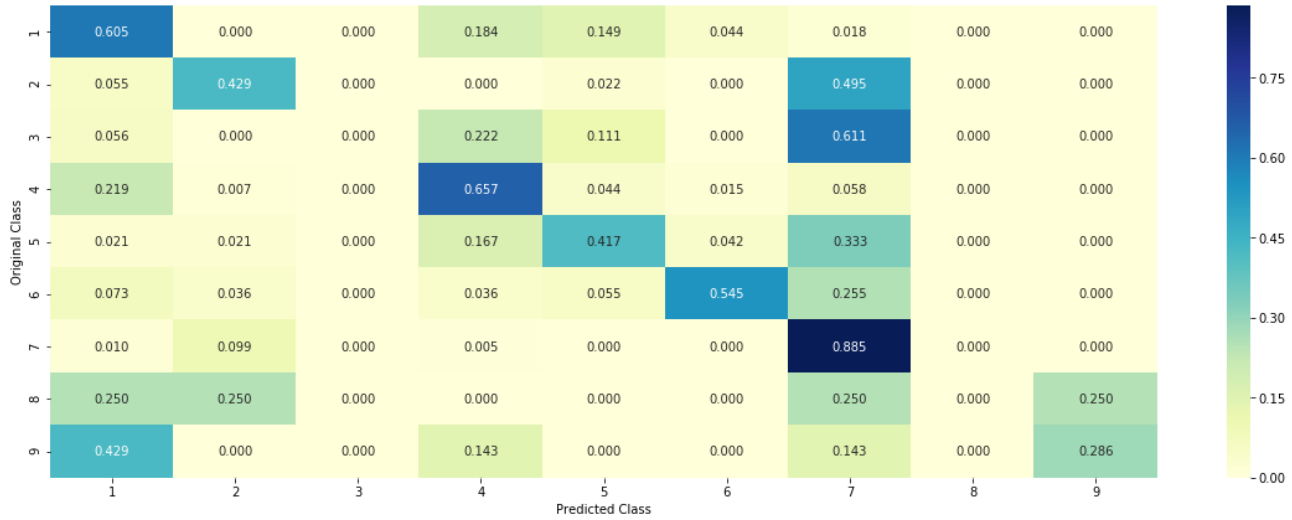




---------- Precision matrix (Columm Sum=1) ----------



---------- Recall matrix (Row sum=1) ----------



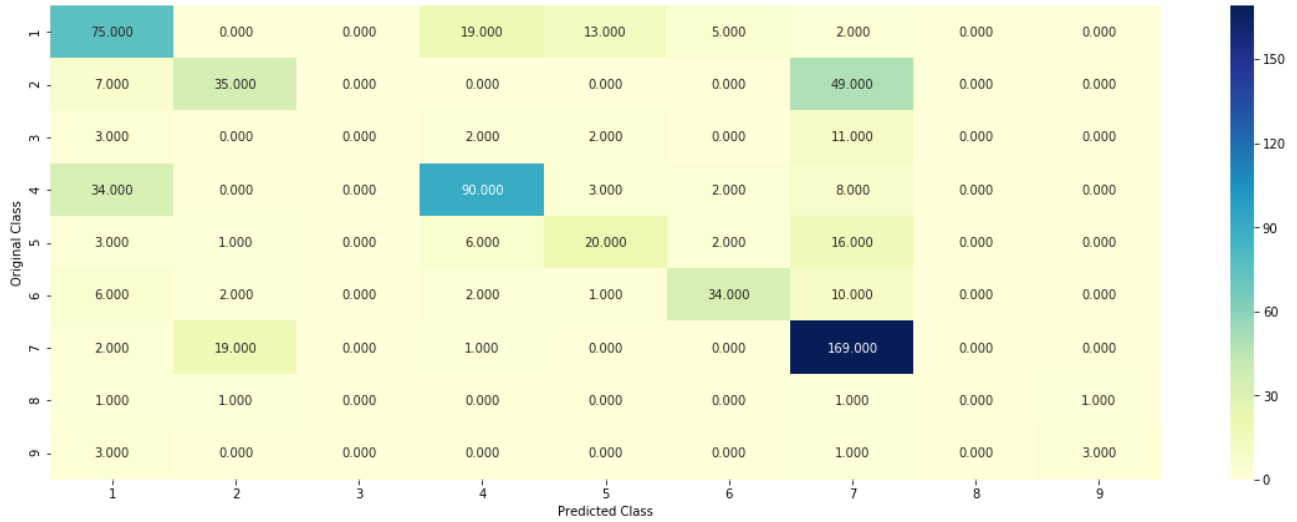
4.7.3 Maximum Voting classifier

In [108]:

```
#Refer:http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.VotingClassifier.html
from sklearn.ensemble import VotingClassifier
vclf = VotingClassifier(estimators=[('lr', sig_clf1), ('svc', sig_clf2), ('rf', sig_clf3)], voting='soft')
vclf.fit(train_x_tfidf, train_y)
print("Log loss (train) on the VotingClassifier :", log_loss(train_y, vclf.predict_proba(train_x_tfidf)))
print("Log loss (CV) on the VotingClassifier :", log_loss(cv_y, vclf.predict_proba(cv_x_tfidf)))
print("Log loss (test) on the VotingClassifier :", log_loss(test_y, vclf.predict_proba(test_x_tfidf)))
print("Number of missclassified point :", np.count_nonzero((vclf.predict(test_
```

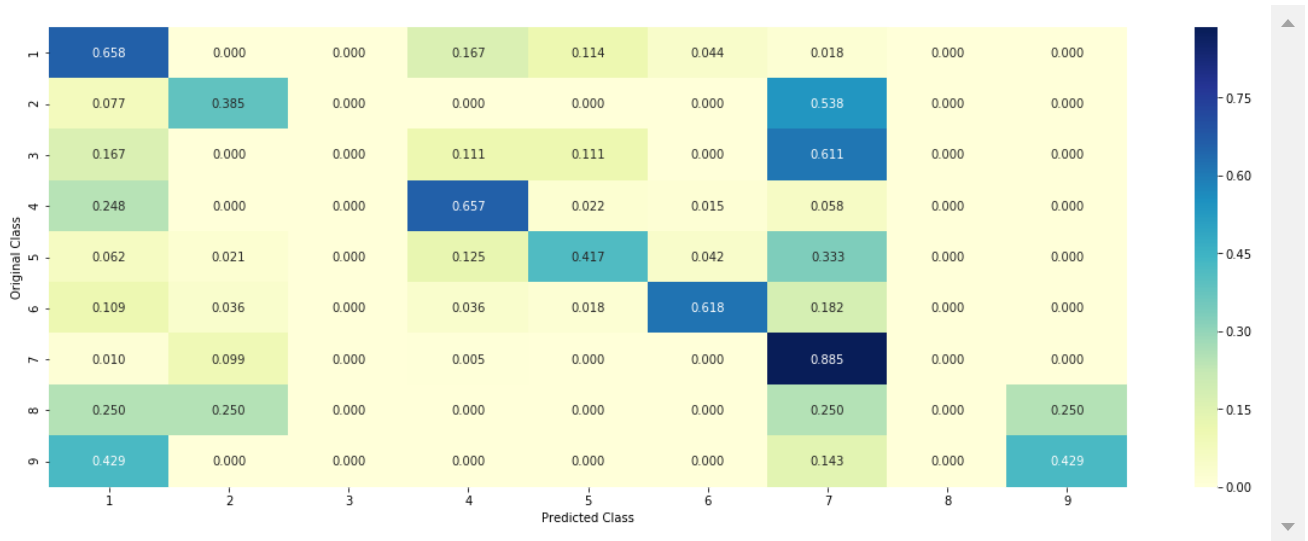
```
x_tfidf)- test_y))/test_y.shape[0])
plot_confusion_matrix(test_y=test_y, predict_y=vclf.predict(test_x_tfidf))
```

```
Log loss (train) on the VotingClassifier : 0.8293651302340408
Log loss (CV) on the VotingClassifier : 1.2087376132082466
Log loss (test) on the VotingClassifier : 1.1664431469139338
Number of missclassified point : 0.3593984962406015
-------------------- Confusion matrix --------------------
```




```
-------------------- Precision matrix (Columm Sum=1) --------------------
```


```
-------------------- Recall matrix (Row sum=1) --------------------
```



3. Apply Logistic regression with CountVectorizer Features, including both unigrams and bigrams

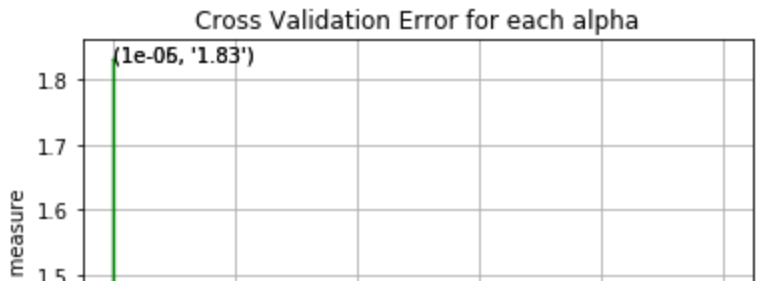
In [51]:
```
from sklearn.feature_extraction.text import CountVectorizer

# Gene Feature
cv=CountVectorizer()
x_train_gene=cv.fit_transform(train_df["Gene"])
x_cv_gene=cv.transform(cv_df["Gene"])
x_test_gene=cv.transform(test_df["Gene"])

# Variation Feature
cv=CountVectorizer()
x_train_var=cv.fit_transform(train_df["Variation"])
x_cv_var=cv.transform(cv_df["Variation"])
x_test_var=cv.transform(test_df["Variation"])

# Text Feature
cv=CountVectorizer(ngram_range=(1, 2))
x_train_text=cv.fit_transform(train_df["TEXT"])
x_cv_text=cv.transform(cv_df["TEXT"])
x_test_text=cv.transform(test_df["TEXT"])



# merging all the feature
train_gene_var = hstack((x_train_gene,x_train_var))
test_gene_var = hstack((x_test_gene,x_test_var))
cv_gene_var = hstack((x_cv_gene,x_cv_var))

train_x = hstack((train_gene_var, x_train_text)).tocsr()
train_y = np.array(list(train_df['Class']))

test_x = hstack((test_gene_var, x_test_text)).tocsr()
test_y = np.array(list(test_df['Class']))

cv_x = hstack((cv_gene_var, x_cv_text)).tocsr()
cv_y = np.array(list(cv_df['Class']))
```

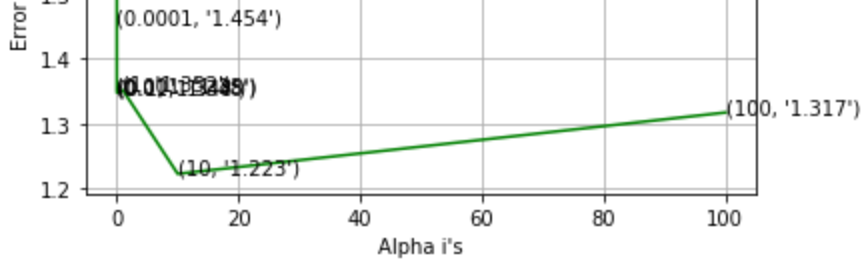
Logistic Regression With Class balancing Hyper paramter tuning

In [52]:
```
alpha = [10 ** x for x in range(-6, 3)]
cv_log_error_array = []
for i in alpha:
    print("for alpha =", i)
    clf = SGDClassifier(class_weight='balanced', alpha=i, penalty='l2', loss=
'log', random_state=42)
    clf.fit(train_x, train_y)
    sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
    sig_clf.fit(train_x, train_y)
    sig_clf_probs = sig_clf.predict_proba(cv_x)
    cv_log_error_array.append(log_loss(cv_y, sig_clf_probs, labels=clf.classes
_, eps=1e-15))
    # to avoid rounding error while multiplying probabilites we use log-probab
ility estimates
    print("Log Loss :",log_loss(cv_y, sig_clf_probs))

fig, ax = plt.subplots()
```

```
ax.plot(alpha, cv_log_error_array,c='g')
for i, txt in enumerate(np.round(cv_log_error_array,3)):
    ax.annotate((alpha[i],str(txt)), (alpha[i],cv_log_error_array[i]))
plt.grid()
plt.title("Cross Validation Error for each alpha")
plt.xlabel("Alpha i's")
plt.ylabel("Error measure")
plt.show()


best_alpha = np.argmin(cv_log_error_array)
clf = SGDClassifier(class_weight='balanced', alpha=alpha[best_alpha], penalty=
'l2', loss='log', random_state=42)
clf.fit(train_x, train_y)
sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
sig_clf.fit(train_x, train_y)

predict_y = sig_clf.predict_proba(train_x)
print('For values of best alpha = ', alpha[best_alpha], "The train log loss i
s:",log_loss(y_train, predict_y, labels=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(cv_x)
print('For values of best alpha = ', alpha[best_alpha], "The cross validation
 log loss is:",log_loss(y_cv, predict_y, labels=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(test_x)
print('For values of best alpha = ', alpha[best_alpha], "The test log loss i
s:",log_loss(y_test, predict_y, labels=clf.classes_, eps=1e-15))
```

```
for alpha = 1e-06
Log Loss : 1.8304997567764278
for alpha = 1e-05
Log Loss : 1.8304997567764278
for alpha = 0.0001
Log Loss : 1.4537982181367093
for alpha = 0.001
Log Loss : 1.3504937053259194
for alpha = 0.01
Log Loss : 1.348078272627178
for alpha = 0.1
Log Loss : 1.347882667263725
for alpha = 1
Log Loss : 1.3515709219078074
for alpha = 10
Log Loss : 1.223196806200396
for alpha = 100
Log Loss : 1.3173871972378963
```






```
For values of best alpha =  10 The train log loss is: 0.9113719891040861
For values of best alpha =  10 The cross validation log loss is: 1.223196806
200396
For values of best alpha =  10 The test log loss is: 1.194371510847823
```

4.3.1.2. Testing the model with best hyper paramters

In [ ]:
```
clf = SGDClassifier(class_weight='balanced', alpha=alpha[best_alpha], penalty=
'l2', loss='log', random_state=42)
predict_and_plot_confusion_matrix(train_x, train_y, cv_x, cv_y, clf)
```

4. Feature Engineering

1. Using unigram, Bigram,trigram and 4gram countvectoriser Feature With Logistic Regression

In [27]:
```
from sklearn.feature_extraction.text import CountVectorizer

# Gene Feature
cv=CountVectorizer()
x_train_gene=cv.fit_transform(train_df["Gene"])
x_cv_gene=cv.transform(cv_df["Gene"])
x_test_gene=cv.transform(test_df["Gene"])

# Variation Feature
cv=CountVectorizer()
x_train_var=cv.fit_transform(train_df["Variation"])
x_cv_var=cv.transform(cv_df["Variation"])
x_test_var=cv.transform(test_df["Variation"])

# Text Feature
cv=CountVectorizer(ngram_range=(1,4 ))
x_train_text=cv.fit_transform(train_df["TEXT"])
x_cv_text=cv.transform(cv_df["TEXT"])
x_test_text=cv.transform(test_df["TEXT"])

# merging all the feature
train_gene_var = hstack((x_train_gene,x_train_var))
test_gene_var = hstack((x_test_gene,x_test_var))
cv_gene_var = hstack((x_cv_gene,x_cv_var))

train_x = hstack((train_gene_var, x_train_text)).tocsr()
train_y = np.array(list(train_df['Class']))

test_x = hstack((test_gene_var, x_test_text)).tocsr()
test_y = np.array(list(test_df['Class']))
```

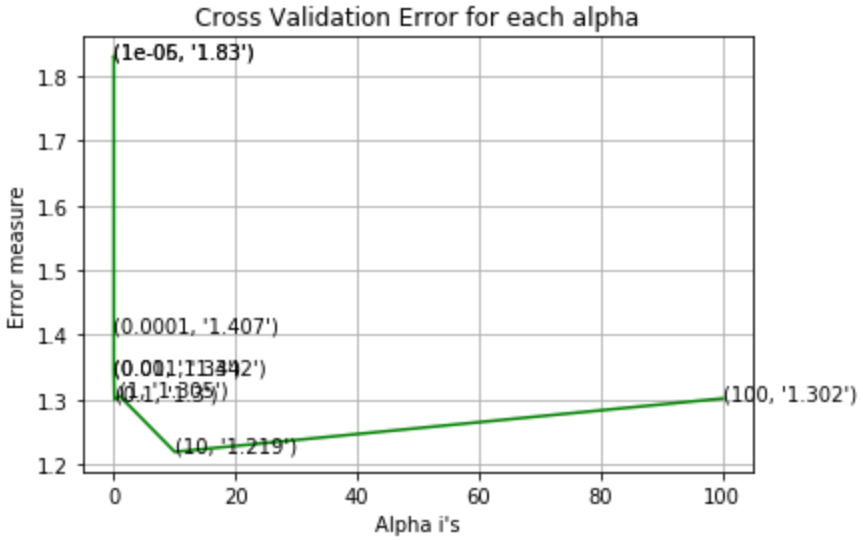
```
cv_x = hstack((cv_gene_var, x_cv_text)).tocsr()
cv_y = np.array(list(cv_df['Class']))
```

Logistic Regression With Class balancing Hyper paramter tuning

In [28]:
```
alpha = [10 ** x for x in range(-6, 3)]
cv_log_error_array = []
for i in alpha:
    print("for alpha =", i)
    clf = SGDClassifier(class_weight='balanced', alpha=i, penalty='l2', loss=
'log', random_state=42)
    clf.fit(train_x, train_y)
    sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
    sig_clf.fit(train_x, train_y)
    sig_clf_probs = sig_clf.predict_proba(cv_x)
    cv_log_error_array.append(log_loss(cv_y, sig_clf_probs, labels=clf.classes
_, eps=1e-15))
    # to avoid rounding error while multiplying probabilites we use log-probab
ility estimates
    print("Log Loss :",log_loss(cv_y, sig_clf_probs))

fig, ax = plt.subplots()
ax.plot(alpha, cv_log_error_array,c='g')
for i, txt in enumerate(np.round(cv_log_error_array,3)):
    ax.annotate((alpha[i],str(txt)), (alpha[i],cv_log_error_array[i]))
plt.grid()
plt.title("Cross Validation Error for each alpha")
plt.xlabel("Alpha i's")
plt.ylabel("Error measure")
plt.show()


best_alpha = np.argmin(cv_log_error_array)
clf = SGDClassifier(class_weight='balanced', alpha=alpha[best_alpha], penalty=
'l2', loss='log', random_state=42)
clf.fit(train_x, train_y)
sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
sig_clf.fit(train_x, train_y)

predict_y = sig_clf.predict_proba(train_x)
print('For values of best alpha = ', alpha[best_alpha], "The train log loss i
s:",log_loss(y_train, predict_y, labels=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(cv_x)
print('For values of best alpha = ', alpha[best_alpha], "The cross validation
 log loss is:",log_loss(y_cv, predict_y, labels=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(test_x)
print('For values of best alpha = ', alpha[best_alpha], "The test log loss i
s:",log_loss(y_test, predict_y, labels=clf.classes_, eps=1e-15))
```

```
for alpha = 1e-06
Log Loss : 1.8304997567764278

for alpha = 1e-05
Log Loss : 1.8304997567764278
for alpha = 0.0001
Log Loss : 1.407135550444909
for alpha = 0.001
Log Loss : 1.3417268082485332
for alpha = 0.01
Log Loss : 1.340352455389858
for alpha = 0.1
```

```
Log Loss : 1.3001011759206698
for alpha = 1
Log Loss : 1.305382573930625
for alpha = 10
Log Loss : 1.2193681149859081
for alpha = 100
Log Loss : 1.3018542573958714
```




```
For values of best alpha =  10 The train log loss is: 0.8928621660035653
For values of best alpha =  10 The cross validation log loss is: 1.219368114
9859081
For values of best alpha =  10 The test log loss is: 1.192587585439302
```

2. Using BIgram tfidf vectoriser Feature With Logistic Regression

In [42]:
```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfVectorizer


# Gene Feature
cv=CountVectorizer()
x_train_gene=cv.fit_transform(train_df["Gene"])
x_cv_gene=cv.transform(cv_df["Gene"])
x_test_gene=cv.transform(test_df["Gene"])

# Variation Feature
cv=CountVectorizer()
x_train_var=cv.fit_transform(train_df["Variation"])
x_cv_var=cv.transform(cv_df["Variation"])
x_test_var=cv.transform(test_df["Variation"])

# Text Feature
tfidf=TfidfVectorizer(ngram_range=(2,2 ))
x_train_text=tfidf.fit_transform(train_df["TEXT"])
x_cv_text=tfidf.transform(cv_df["TEXT"])
x_test_text=tfidf.transform(test_df["TEXT"])

# merging all the feature
train_gene_var = hstack((x_train_gene,x_train_var))
test_gene_var = hstack((x_test_gene,x_test_var))
cv_gene_var = hstack((x_cv_gene,x_cv_var))

train_x = hstack((train_gene_var, x_train_text)).tocsr()
```

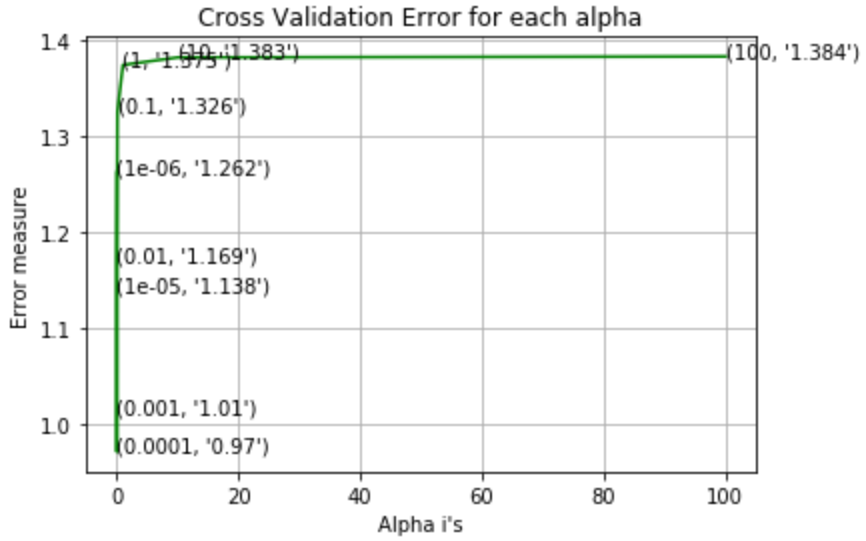
```
train_y = np.array(list(train_df['Class']))

test_x = hstack((test_gene_var, x_test_text)).tocsr()
test_y = np.array(list(test_df['Class']))

cv_x = hstack((cv_gene_var, x_cv_text)).tocsr()
cv_y = np.array(list(cv_df['Class']))
```

Logistic Regression With Class balancing Hyper paramter tuning

In [43]:
```
alpha = [10 ** x for x in range(-6, 3)]
cv_log_error_array = []
for i in alpha:
    print("for alpha =", i)
    clf = SGDClassifier(class_weight='balanced', alpha=i, penalty='l2', loss=
'log', random_state=42)
    clf.fit(train_x, train_y)
    sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
    sig_clf.fit(train_x, train_y)
    sig_clf_probs = sig_clf.predict_proba(cv_x)
    cv_log_error_array.append(log_loss(cv_y, sig_clf_probs, labels=clf.classes
_, eps=1e-15))
    # to avoid rounding error while multiplying probabilites we use log-probab
ility estimates
    print("Log Loss :",log_loss(cv_y, sig_clf_probs))

fig, ax = plt.subplots()
ax.plot(alpha, cv_log_error_array,c='g')
for i, txt in enumerate(np.round(cv_log_error_array,3)):
    ax.annotate((alpha[i],str(txt)), (alpha[i],cv_log_error_array[i]))
plt.grid()
plt.title("Cross Validation Error for each alpha")
plt.xlabel("Alpha i's")
plt.ylabel("Error measure")
plt.show()


best_alpha = np.argmin(cv_log_error_array)
clf = SGDClassifier(class_weight='balanced', alpha=alpha[best_alpha], penalty=
'l2', loss='log', random_state=42)
clf.fit(train_x, train_y)
sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
sig_clf.fit(train_x, train_y)

predict_y = sig_clf.predict_proba(train_x)
print('For values of best alpha = ', alpha[best_alpha], "The train log loss i
s:",log_loss(y_train, predict_y, labels=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(cv_x)
print('For values of best alpha = ', alpha[best_alpha], "The cross validation
 log loss is:",log_loss(y_cv, predict_y, labels=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(test_x)
print('For values of best alpha = ', alpha[best_alpha], "The test log loss i
s:",log_loss(y_test, predict_y, labels=clf.classes_, eps=1e-15))
```

```
for alpha = 1e-06
Log Loss : 1.262478459461488
for alpha = 1e-05
Log Loss : 1.1380131484014608
for alpha = 0.0001
Log Loss : 0.9703538090164371
for alpha = 0.001
```

```
Log Loss : 1.0100152811876142
for alpha = 0.01
Log Loss : 1.1687546834896385
for alpha = 0.1
Log Loss : 1.3257098192693062
for alpha = 1
Log Loss : 1.3747737753440623
for alpha = 10
Log Loss : 1.3826334312258144
for alpha = 100
Log Loss : 1.383638738670203
```




```
For values of best alpha =  0.0001 The train log loss is: 0.3883977770562982
For values of best alpha =  0.0001 The cross validation log loss is: 0.97035
38090164371
For values of best alpha =  0.0001 The test log loss is: 0.9533144706807126
```

3. Using bigram and trigram tfidf vectoriser

In [38]:
```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfVectorizer

# Gene Feature
cv=CountVectorizer()
x_train_gene=cv.fit_transform(train_df["Gene"])
x_cv_gene=cv.transform(cv_df["Gene"])
x_test_gene=cv.transform(test_df["Gene"])

# Variation Feature
cv=CountVectorizer()
x_train_var=cv.fit_transform(train_df["Variation"])
x_cv_var=cv.transform(cv_df["Variation"])
x_test_var=cv.transform(test_df["Variation"])

# Text Feature
cv=CountVectorizer(ngram_range=(2,3 ))
x_train_text=cv.fit_transform(train_df["TEXT"])
x_cv_text=cv.transform(cv_df["TEXT"])
x_test_text=cv.transform(test_df["TEXT"])

# merging all the feature
train_gene_var = hstack((x_train_gene,x_train_var))
test_gene_var = hstack((x_test_gene,x_test_var))
```

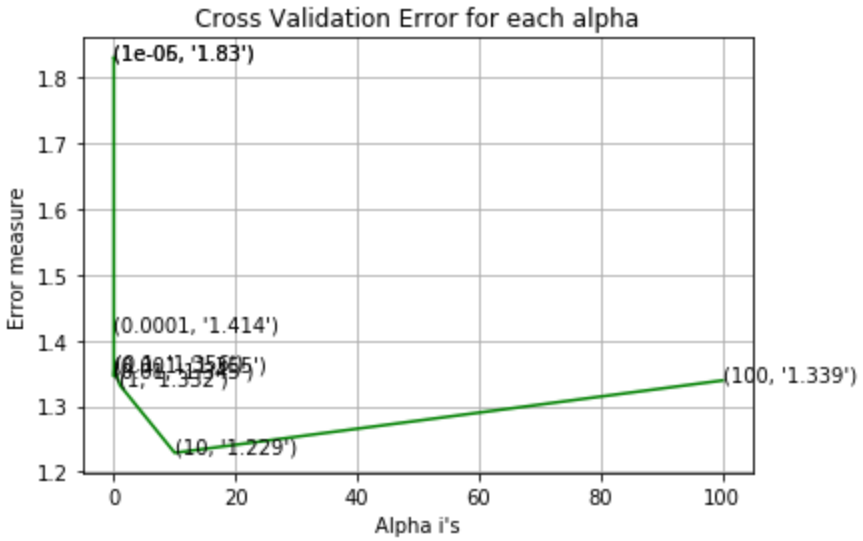
```
cv_gene_var = hstack((x_cv_gene,x_cv_var))

train_x = hstack((train_gene_var, x_train_text)).tocsr()
train_y = np.array(list(train_df['Class']))

test_x = hstack((test_gene_var, x_test_text)).tocsr()
test_y = np.array(list(test_df['Class']))

cv_x = hstack((cv_gene_var, x_cv_text)).tocsr()
cv_y = np.array(list(cv_df['Class']))
```

Logistic Regression With Class balancing Hyper paramter tuning

In [39]:
```
alpha = [10 ** x for x in range(-6, 3)]
cv_log_error_array = []
for i in alpha:
    print("for alpha =", i)
    clf = SGDClassifier(class_weight='balanced', alpha=i, penalty='l2', loss=
'log', random_state=42)
    clf.fit(train_x, train_y)
    sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
    sig_clf.fit(train_x, train_y)
    sig_clf_probs = sig_clf.predict_proba(cv_x)
    cv_log_error_array.append(log_loss(cv_y, sig_clf_probs, labels=clf.classes
_, eps=1e-15))
    # to avoid rounding error while multiplying probabilites we use log-probab
ility estimates
    print("Log Loss :",log_loss(cv_y, sig_clf_probs))

fig, ax = plt.subplots()
ax.plot(alpha, cv_log_error_array,c='g')
for i, txt in enumerate(np.round(cv_log_error_array,3)):
    ax.annotate((alpha[i],str(txt)), (alpha[i],cv_log_error_array[i]))
plt.grid()
plt.title("Cross Validation Error for each alpha")
plt.xlabel("Alpha i's")
plt.ylabel("Error measure")
plt.show()


best_alpha = np.argmin(cv_log_error_array)
clf = SGDClassifier(class_weight='balanced', alpha=alpha[best_alpha], penalty=
'l2', loss='log', random_state=42)
clf.fit(train_x, train_y)
sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
sig_clf.fit(train_x, train_y)

predict_y = sig_clf.predict_proba(train_x)
print('For values of best alpha = ', alpha[best_alpha], "The train log loss i
s:",log_loss(y_train, predict_y, labels=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(cv_x)
print('For values of best alpha = ', alpha[best_alpha], "The cross validation
 log loss is:",log_loss(y_cv, predict_y, labels=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(test_x)
print('For values of best alpha = ', alpha[best_alpha], "The test log loss i
s:",log_loss(y_test, predict_y, labels=clf.classes_, eps=1e-15))
```

```
for alpha = 1e-06
Log Loss : 1.8304997567764278
for alpha = 1e-05
Log Loss : 1.8304997567764278
```

```
for alpha = 0.0001
Log Loss : 1.4143757708337763
for alpha = 0.001
Log Loss : 1.3549301225530226
for alpha = 0.01
Log Loss : 1.3451960913266776
for alpha = 0.1
Log Loss : 1.3563067014371575
for alpha = 1
Log Loss : 1.331955859609987
for alpha = 10
Log Loss : 1.2289435932121924
for alpha = 100
Log Loss : 1.3390697748339953
```




```
For values of best alpha =  10 The train log loss is: 0.9366628817307023
For values of best alpha =  10 The cross validation log loss is: 1.228943593
2121924
For values of best alpha =  10 The test log loss is: 1.1988017471390509
```

4. Using Only trigram tfidf with Logistic regression

In [40]:

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfVectorizer

# Gene Feature
cv=CountVectorizer()
x_train_gene=cv.fit_transform(train_df["Gene"])
x_cv_gene=cv.transform(cv_df["Gene"])
x_test_gene=cv.transform(test_df["Gene"])

# Variation Feature
cv=CountVectorizer()
x_train_var=cv.fit_transform(train_df["Variation"])
x_cv_var=cv.transform(cv_df["Variation"])
x_test_var=cv.transform(test_df["Variation"])

# Text Feature
tfidf=TfidfVectorizer(ngram_range=(3,3 ))
x_train_text=tfidf.fit_transform(train_df["TEXT"])
x_cv_text=tfidf.transform(cv_df["TEXT"])
x_test_text=tfidf.transform(test_df["TEXT"])
```

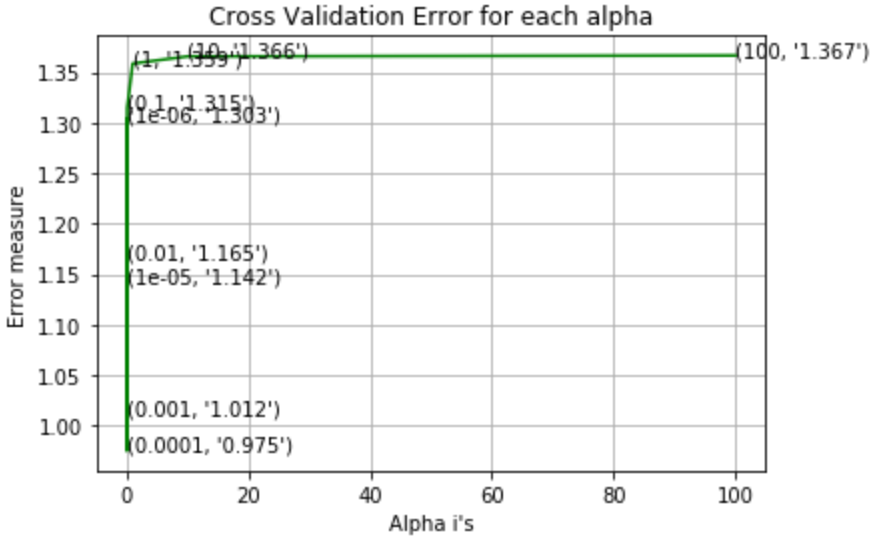
```
# merging all the feature
train_gene_var = hstack((x_train_gene,x_train_var))
test_gene_var = hstack((x_test_gene,x_test_var))
cv_gene_var = hstack((x_cv_gene,x_cv_var))

train_x = hstack((train_gene_var, x_train_text)).tocsr()
train_y = np.array(list(train_df['Class']))

test_x = hstack((test_gene_var, x_test_text)).tocsr()
test_y = np.array(list(test_df['Class']))

cv_x = hstack((cv_gene_var, x_cv_text)).tocsr()
cv_y = np.array(list(cv_df['Class']))
```

Logistic Regression With Class balancing Hyper paramter tuning

In [41]:
```
alpha = [10 ** x for x in range(-6, 3)]
cv_log_error_array = []
for i in alpha:
    print("for alpha =", i)
    clf = SGDClassifier(class_weight='balanced', alpha=i, penalty='l2', loss=
'log', random_state=42)
    clf.fit(train_x, train_y)
    sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
    sig_clf.fit(train_x, train_y)
    sig_clf_probs = sig_clf.predict_proba(cv_x)
    cv_log_error_array.append(log_loss(cv_y, sig_clf_probs, labels=clf.classes
_, eps=1e-15))
    # to avoid rounding error while multiplying probabilites we use log-probab
ility estimates
    print("Log Loss :",log_loss(cv_y, sig_clf_probs))

fig, ax = plt.subplots()
ax.plot(alpha, cv_log_error_array,c='g')
for i, txt in enumerate(np.round(cv_log_error_array,3)):
    ax.annotate((alpha[i],str(txt)), (alpha[i],cv_log_error_array[i]))
plt.grid()
plt.title("Cross Validation Error for each alpha")
plt.xlabel("Alpha i's")
plt.ylabel("Error measure")
plt.show()


best_alpha = np.argmin(cv_log_error_array)
clf = SGDClassifier(class_weight='balanced', alpha=alpha[best_alpha], penalty=
'l2', loss='log', random_state=42)
clf.fit(train_x, train_y)
sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
sig_clf.fit(train_x, train_y)

predict_y = sig_clf.predict_proba(train_x)
print('For values of best alpha = ', alpha[best_alpha], "The train log loss i
s:",log_loss(y_train, predict_y, labels=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(cv_x)
print('For values of best alpha = ', alpha[best_alpha], "The cross validation
 log loss is:",log_loss(y_cv, predict_y, labels=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(test_x)
print('For values of best alpha = ', alpha[best_alpha], "The test log loss i
s:",log_loss(y_test, predict_y, labels=clf.classes_, eps=1e-15))
```

```
for alpha = 1e-06
```

```
Log Loss : 1.3029234580208533
for alpha = 1e-05
Log Loss : 1.142249215102017
for alpha = 0.0001
Log Loss : 0.9751853994712354
for alpha = 0.001
Log Loss : 1.0116351451020542
for alpha = 0.01
Log Loss : 1.1649499284903657
for alpha = 0.1
Log Loss : 1.3146428262021865
for alpha = 1
Log Loss : 1.3585912262772668
for alpha = 10
Log Loss : 1.3656792140494465
for alpha = 100
Log Loss : 1.3665983952564853
```




```
For values of best alpha =  0.0001 The train log loss is: 0.3882761441935036
For values of best alpha =  0.0001 The cross validation log loss is: 0.97518
53994712354
For values of best alpha =  0.0001 The test log loss is: 0.9613716671191279
```

5. Using Only 4gram tfidf with Logistic regression

In [45]:

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfVectorizer

# Gene Feature
cv=CountVectorizer()
x_train_gene=cv.fit_transform(train_df["Gene"])
x_cv_gene=cv.transform(cv_df["Gene"])
x_test_gene=cv.transform(test_df["Gene"])

# Variation Feature
cv=CountVectorizer()
x_train_var=cv.fit_transform(train_df["Variation"])
x_cv_var=cv.transform(cv_df["Variation"])
x_test_var=cv.transform(test_df["Variation"])

# Text Feature
tfidf=TfidfVectorizer(ngram_range=(4,4 ))
x_train_text=tfidf.fit_transform(train_df["TEXT"])
```

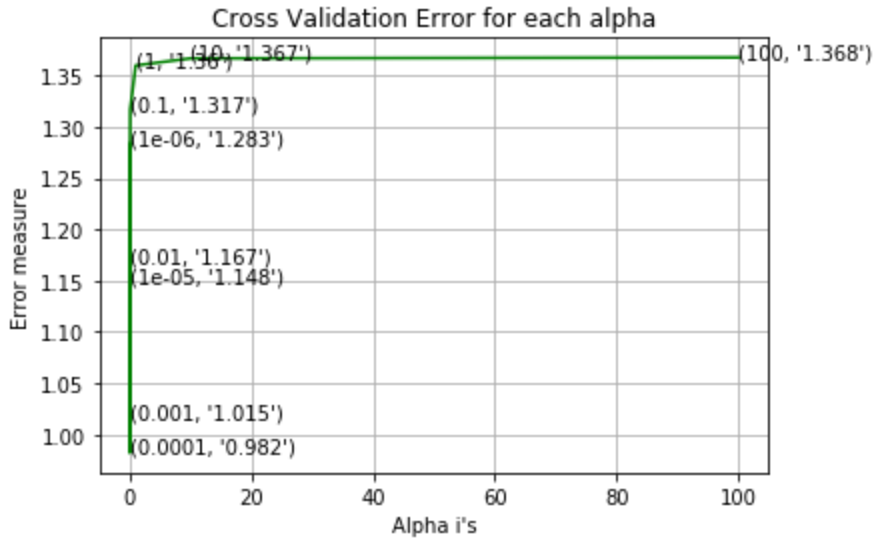
```
x_cv_text=tfidf.transform(cv_df["TEXT"])
x_test_text=tfidf.transform(test_df["TEXT"])

# merging all the feature
train_gene_var = hstack((x_train_gene,x_train_var))
test_gene_var = hstack((x_test_gene,x_test_var))
cv_gene_var = hstack((x_cv_gene,x_cv_var))

train_x = hstack((train_gene_var, x_train_text)).tocsr()
train_y = np.array(list(train_df['Class']))

test_x = hstack((test_gene_var, x_test_text)).tocsr()
test_y = np.array(list(test_df['Class']))

cv_x = hstack((cv_gene_var, x_cv_text)).tocsr()
cv_y = np.array(list(cv_df['Class']))
```

Logistic Regression With Class balancing Hyper paramter tuning

In [46]:
```
alpha = [10 ** x for x in range(-6, 3)]
cv_log_error_array = []
for i in alpha:
    print("for alpha =", i)
    clf = SGDClassifier(class_weight='balanced', alpha=i, penalty='l2', loss=
'log', random_state=42)
    clf.fit(train_x, train_y)
    sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
    sig_clf.fit(train_x, train_y)
    sig_clf_probs = sig_clf.predict_proba(cv_x)
    cv_log_error_array.append(log_loss(cv_y, sig_clf_probs, labels=clf.classes
_, eps=1e-15))
    # to avoid rounding error while multiplying probabilites we use log-probab
ility estimates
    print("Log Loss :",log_loss(cv_y, sig_clf_probs))

fig, ax = plt.subplots()
ax.plot(alpha, cv_log_error_array,c='g')
for i, txt in enumerate(np.round(cv_log_error_array,3)):
    ax.annotate((alpha[i],str(txt)), (alpha[i],cv_log_error_array[i]))
plt.grid()
plt.title("Cross Validation Error for each alpha")
plt.xlabel("Alpha i's")
plt.ylabel("Error measure")
plt.show()


best_alpha = np.argmin(cv_log_error_array)
clf = SGDClassifier(class_weight='balanced', alpha=alpha[best_alpha], penalty=
'l2', loss='log', random_state=42)
clf.fit(train_x, train_y)
sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
sig_clf.fit(train_x, train_y)

predict_y = sig_clf.predict_proba(train_x)
print('For values of best alpha = ', alpha[best_alpha], "The train log loss i
s:",log_loss(y_train, predict_y, labels=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(cv_x)
print('For values of best alpha = ', alpha[best_alpha], "The cross validation
 log loss is:",log_loss(y_cv, predict_y, labels=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(test_x)
```

```
print('For values of best alpha = ', alpha[best_alpha], "The test log loss i
s:",log_loss(y_test, predict_y, labels=clf.classes_, eps=1e-15))
```

```
for alpha = 1e-06
Log Loss : 1.283076991101578
for alpha = 1e-05
Log Loss : 1.1475141327316043
for alpha = 0.0001
Log Loss : 0.9820180054747053
for alpha = 0.001
Log Loss : 1.015425057131513
for alpha = 0.01
Log Loss : 1.1674758896429183
for alpha = 0.1
Log Loss : 1.3167102025004243
for alpha = 1
Log Loss : 1.3598311249159847
for alpha = 10
Log Loss : 1.3667394231225376
for alpha = 100
Log Loss : 1.367635403716636
```




```
For values of best alpha =  0.0001 The train log loss is: 0.3903872866550658
3
For values of best alpha =  0.0001 The cross validation log loss is: 0.98201
80054747053
For values of best alpha =  0.0001 The test log loss is: 0.9684446144974879
```

6. Using trigram and 4gram tfidf with Logistic regression

In [48]:

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfVectorizer

# Gene Feature
cv=CountVectorizer()
x_train_gene=cv.fit_transform(train_df["Gene"])
x_cv_gene=cv.transform(cv_df["Gene"])
x_test_gene=cv.transform(test_df["Gene"])

# Variation Feature
cv=CountVectorizer()
x_train_var=cv.fit_transform(train_df["Variation"])
x_cv_var=cv.transform(cv_df["Variation"])
```

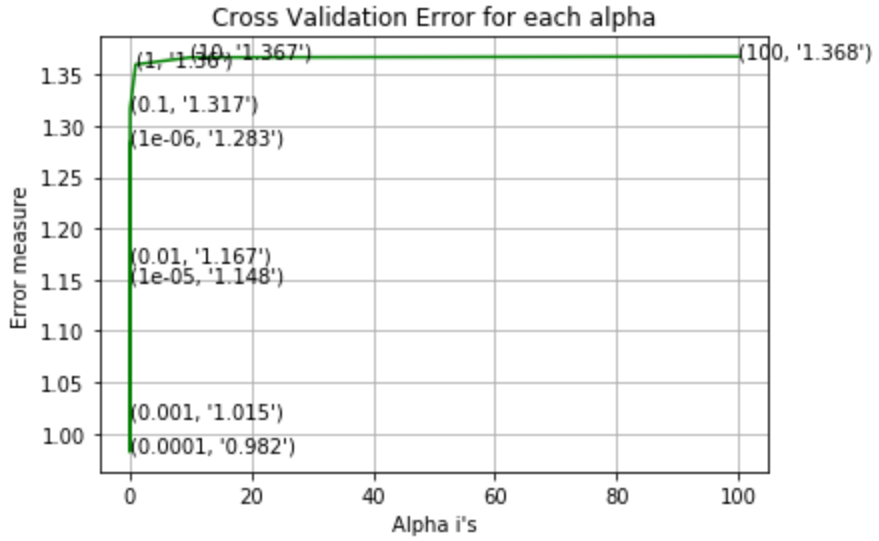
```
x_test_var=cv.transform(test_df["Variation"])

# Text Feature
tfidf=TfidfVectorizer(ngram_range=(3,4 ))
x_train_text=tfidf.fit_transform(train_df["TEXT"])
x_cv_text=tfidf.transform(cv_df["TEXT"])
x_test_text=tfidf.transform(test_df["TEXT"])

# merging all the feature
train_gene_var = hstack((x_train_gene,x_train_var))
test_gene_var = hstack((x_test_gene,x_test_var))
cv_gene_var = hstack((x_cv_gene,x_cv_var))

train_x = hstack((train_gene_var, x_train_text)).tocsr()
train_y = np.array(list(train_df['Class']))

test_x = hstack((test_gene_var, x_test_text)).tocsr()
test_y = np.array(list(test_df['Class']))

cv_x = hstack((cv_gene_var, x_cv_text)).tocsr()
cv_y = np.array(list(cv_df['Class']))
```

Logistic Regression With Class balancing Hyper paramter tuning

In [46]:
```
alpha = [10 ** x for x in range(-6, 3)]
cv_log_error_array = []
for i in alpha:
    print("for alpha =", i)
    clf = SGDClassifier(class_weight='balanced', alpha=i, penalty='l2', loss=
'log', random_state=42)
    clf.fit(train_x, train_y)
    sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
    sig_clf.fit(train_x, train_y)
    sig_clf_probs = sig_clf.predict_proba(cv_x)
    cv_log_error_array.append(log_loss(cv_y, sig_clf_probs, labels=clf.classes
_, eps=1e-15))
    # to avoid rounding error while multiplying probabilites we use log-probab
ility estimates
    print("Log Loss :",log_loss(cv_y, sig_clf_probs))

fig, ax = plt.subplots()
ax.plot(alpha, cv_log_error_array,c='g')
for i, txt in enumerate(np.round(cv_log_error_array,3)):
    ax.annotate((alpha[i],str(txt)), (alpha[i],cv_log_error_array[i]))
plt.grid()
plt.title("Cross Validation Error for each alpha")
plt.xlabel("Alpha i's")
plt.ylabel("Error measure")
plt.show()


best_alpha = np.argmin(cv_log_error_array)
clf = SGDClassifier(class_weight='balanced', alpha=alpha[best_alpha], penalty=
'l2', loss='log', random_state=42)
clf.fit(train_x, train_y)
sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
sig_clf.fit(train_x, train_y)

predict_y = sig_clf.predict_proba(train_x)
print('For values of best alpha = ', alpha[best_alpha], "The train log loss i
```

```
s:",log_loss(y_train, predict_y, labels=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(cv_x)
print('For values of best alpha = ', alpha[best_alpha], "The cross validation
 log loss is:",log_loss(y_cv, predict_y, labels=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(test_x)
print('For values of best alpha = ', alpha[best_alpha], "The test log loss i
s:",log_loss(y_test, predict_y, labels=clf.classes_, eps=1e-15))
```

```
for alpha = 1e-06
Log Loss : 1.283076991101578
for alpha = 1e-05
Log Loss : 1.1475141327316043
for alpha = 0.0001
Log Loss : 0.9820180054747053
for alpha = 0.001
Log Loss : 1.015425057131513
for alpha = 0.01
Log Loss : 1.1674758896429183
for alpha = 0.1
Log Loss : 1.3167102025004243
for alpha = 1
Log Loss : 1.3598311249159847
for alpha = 10
Log Loss : 1.3667394231225376
for alpha = 100
Log Loss : 1.367635403716636
```




```
For values of best alpha =  0.0001 The train log loss is: 0.3903872866550658
3
For values of best alpha =  0.0001 The cross validation log loss is: 0.98201
80054747053
For values of best alpha =  0.0001 The test log loss is: 0.9684446144974879
```

7. Using unigram, bigram, tri gram, 4gram tfidf with Logistic regression

In [49]:

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfVectorizer

# Gene Feature
cv=CountVectorizer()
x_train_gene=cv.fit_transform(train_df["Gene"])
x_cv_gene=cv.transform(cv_df["Gene"])
x_test_gene=cv.transform(test_df["Gene"])
```

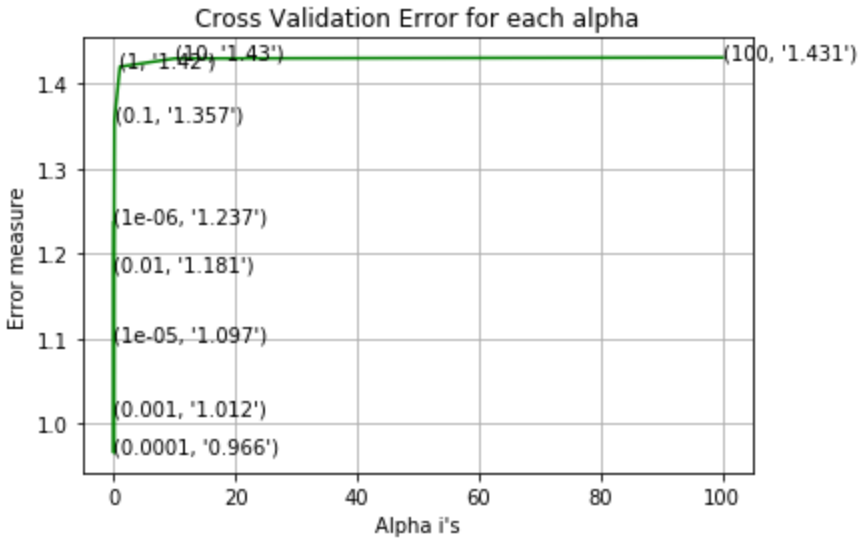
```
# Variation Feature
cv=CountVectorizer()
x_train_var=cv.fit_transform(train_df["Variation"])
x_cv_var=cv.transform(cv_df["Variation"])
x_test_var=cv.transform(test_df["Variation"])

# Text Feature
tfidf=TfidfVectorizer(ngram_range=(1,4 ))
x_train_text=tfidf.fit_transform(train_df["TEXT"])
x_cv_text=tfidf.transform(cv_df["TEXT"])
x_test_text=tfidf.transform(test_df["TEXT"])

# merging all the feature
train_gene_var = hstack((x_train_gene,x_train_var))
test_gene_var = hstack((x_test_gene,x_test_var))
cv_gene_var = hstack((x_cv_gene,x_cv_var))

train_x = hstack((train_gene_var, x_train_text)).tocsr()
train_y = np.array(list(train_df['Class']))

test_x = hstack((test_gene_var, x_test_text)).tocsr()
test_y = np.array(list(test_df['Class']))

cv_x = hstack((cv_gene_var, x_cv_text)).tocsr()
cv_y = np.array(list(cv_df['Class']))
```

Logistic Regression With Class balancing Hyper paramter tuning

In [50]:
```
alpha = [10 ** x for x in range(-6, 3)]
cv_log_error_array = []
for i in alpha:
    print("for alpha =", i)
    clf = SGDClassifier(class_weight='balanced', alpha=i, penalty='l2', loss=
'log', random_state=42)
    clf.fit(train_x, train_y)
    sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
    sig_clf.fit(train_x, train_y)
    sig_clf_probs = sig_clf.predict_proba(cv_x)
    cv_log_error_array.append(log_loss(cv_y, sig_clf_probs, labels=clf.classes
_, eps=1e-15))
    # to avoid rounding error while multiplying probabilites we use log-probab
ility estimates
    print("Log Loss :",log_loss(cv_y, sig_clf_probs))

fig, ax = plt.subplots()
ax.plot(alpha, cv_log_error_array,c='g')
for i, txt in enumerate(np.round(cv_log_error_array,3)):
    ax.annotate((alpha[i],str(txt)), (alpha[i],cv_log_error_array[i]))
plt.grid()
plt.title("Cross Validation Error for each alpha")
plt.xlabel("Alpha i's")
plt.ylabel("Error measure")
plt.show()


best_alpha = np.argmin(cv_log_error_array)
clf = SGDClassifier(class_weight='balanced', alpha=alpha[best_alpha], penalty=
'l2', loss='log', random_state=42)
clf.fit(train_x, train_y)
```

```
sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
sig_clf.fit(train_x, train_y)

predict_y = sig_clf.predict_proba(train_x)
print('For values of best alpha = ', alpha[best_alpha], "The train log loss i
s:",log_loss(y_train, predict_y, labels=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(cv_x)
print('For values of best alpha = ', alpha[best_alpha], "The cross validation
 log loss is:",log_loss(y_cv, predict_y, labels=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(test_x)
print('For values of best alpha = ', alpha[best_alpha], "The test log loss i
s:",log_loss(y_test, predict_y, labels=clf.classes_, eps=1e-15))
```

```
for alpha = 1e-06
Log Loss : 1.2365095163033968
for alpha = 1e-05
Log Loss : 1.0971123911962597
for alpha = 0.0001
Log Loss : 0.9655924191249846
for alpha = 0.001
Log Loss : 1.0118632991051582
for alpha = 0.01
Log Loss : 1.1806123516785845
for alpha = 0.1
Log Loss : 1.3573554580248164
for alpha = 1
Log Loss : 1.4200727829306072
for alpha = 10
Log Loss : 1.429710441078211
for alpha = 100
Log Loss : 1.4309094456351552
```




```
For values of best alpha =  0.0001 The train log loss is: 0.3931342079225767
For values of best alpha =  0.0001 The cross validation log loss is: 0.96559
24191249846
For values of best alpha =  0.0001 The test log loss is: 0.9584827331654904
```

8. Taking reviews of length along with other features

In [60]:

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfVectorizer

# Gene Feature
```

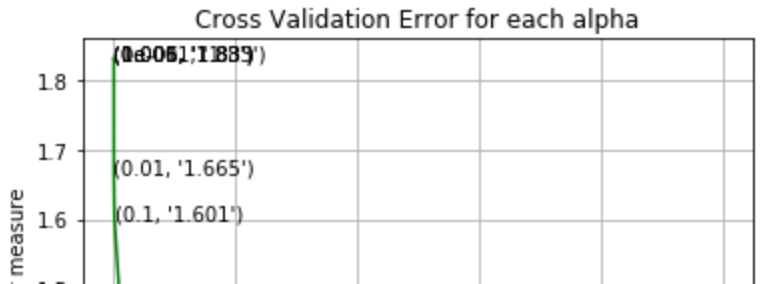
```
cv=CountVectorizer()
x_train_gene=cv.fit_transform(train_df["Gene"])
x_cv_gene=cv.transform(cv_df["Gene"])
x_test_gene=cv.transform(test_df["Gene"])

# Variation Feature
cv=CountVectorizer()
x_train_var=cv.fit_transform(train_df["Variation"])
x_cv_var=cv.transform(cv_df["Variation"])
x_test_var=cv.transform(test_df["Variation"])

# Text Feature
tfidf=TfidfVectorizer(ngram_range=(4,4 ))
x_train_text=tfidf.fit_transform(train_df["TEXT"])
x_cv_text=tfidf.transform(cv_df["TEXT"])
x_test_text=tfidf.transform(test_df["TEXT"])

from sklearn.preprocessing import StandardScaler
sc=StandardScaler(with_mean=False)
x_train_text=sc.fit_transform(x_train_text)
x_cv_text=sc.transform(x_cv_text)
x_test_text=sc.transform(x_test_text)

# merging all the feature
train_gene_var = hstack((x_train_gene,x_train_var))
test_gene_var = hstack((x_test_gene,x_test_var))
cv_gene_var = hstack((x_cv_gene,x_cv_var))

train_x = hstack((train_gene_var, x_train_text)).tocsr()
train_y = np.array(list(train_df['Class']))

test_x = hstack((test_gene_var, x_test_text)).tocsr()
test_y = np.array(list(test_df['Class']))

cv_x = hstack((cv_gene_var, x_cv_text)).tocsr()
cv_y = np.array(list(cv_df['Class']))
```

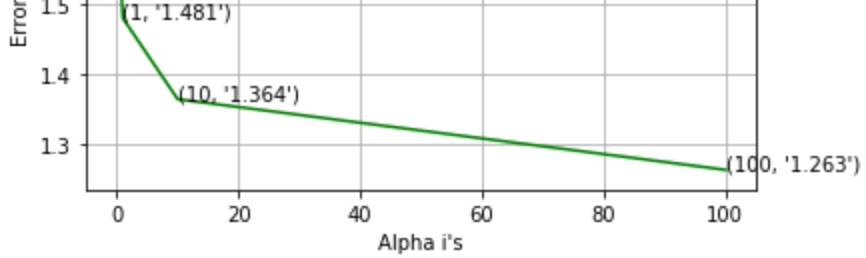
Logistic Regression With Class balancing Hyper paramter tuning

In [61]:
```
alpha = [10 ** x for x in range(-6, 3)]
cv_log_error_array = []
for i in alpha:
    print("for alpha =", i)
    clf = SGDClassifier(class_weight='balanced', alpha=i, penalty='l2', loss=
'hinge', random_state=42)
    clf.fit(train_x, train_y)
    sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
    sig_clf.fit(train_x, train_y)
    sig_clf_probs = sig_clf.predict_proba(cv_x)
    cv_log_error_array.append(log_loss(cv_y, sig_clf_probs, labels=clf.classes
_, eps=1e-15))
    # to avoid rounding error while multiplying probabilites we use log-probab
ility estimates
    print("Log Loss :",log_loss(cv_y, sig_clf_probs))

fig, ax = plt.subplots()
ax.plot(alpha, cv_log_error_array,c='g')
for i, txt in enumerate(np.round(cv_log_error_array,3)):
    ax.annotate((alpha[i],str(txt)), (alpha[i],cv_log_error_array[i]))
plt.grid()
```

```
plt.title("Cross Validation Error for each alpha")
plt.xlabel("Alpha i's")
plt.ylabel("Error measure")
plt.show()


best_alpha = np.argmin(cv_log_error_array)
clf = SGDClassifier(class_weight='balanced', alpha=alpha[best_alpha], penalty=
'l2', loss='log', random_state=42)
clf.fit(train_x, train_y)
sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
sig_clf.fit(train_x, train_y)

predict_y = sig_clf.predict_proba(train_x)
print('For values of best alpha = ', alpha[best_alpha], "The train log loss i
s:",log_loss(y_train, predict_y, labels=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(cv_x)
print('For values of best alpha = ', alpha[best_alpha], "The cross validation
 log loss is:",log_loss(y_cv, predict_y, labels=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(test_x)
print('For values of best alpha = ', alpha[best_alpha], "The test log loss i
s:",log_loss(y_test, predict_y, labels=clf.classes_, eps=1e-15))
```

```
for alpha = 1e-06
Log Loss : 1.8304997567764278
for alpha = 1e-05
Log Loss : 1.8304997567764278
for alpha = 0.0001
Log Loss : 1.8304997567764278
for alpha = 0.001
Log Loss : 1.8304997567764278
for alpha = 0.01
Log Loss : 1.6650368977521315
for alpha = 0.1
Log Loss : 1.6010749902346118
for alpha = 1
Log Loss : 1.4805581539488626
for alpha = 10
Log Loss : 1.364469484927393
for alpha = 100
Log Loss : 1.2631090571942207
```






```
For values of best alpha =  100 The train log loss is: 0.5432547748536471
For values of best alpha =  100 The cross validation log loss is: 1.17706760
08508216
For values of best alpha =  100 The test log loss is: 1.236628469521574
```

Results

In [24]:

```
# pretty table 1

from prettytable import PrettyTable
x=PrettyTable(field_names=["vectoriser","Model","train logloss","CV loss","tes
t loss","mis-classification"])

x.add_row(["onehotEncoding\n","NB",0.90,1.27,1.21,"39.84%"])
x.add_row(["ResponseCoding\n","KNN",0.705,1.30,1002,"39.47%"])
x.add_row(["onehotEncoding\n","LR+Balacing",0.61,1.14,1.048,"34.77%"])
x.add_row(["onehotEncoding\n","LR+NoBalancing",0.628,1.185,1.054,"36.28%"])
x.add_row(["onehotEncoding\n","SVM+balancing",0.739,1.132,1.063,"39.47%"])
x.add_row(["onehotEncoding\n","RF",0.703,1.192,1.097,"36.84%"])
x.add_row(["Response codin\n","RF",0.052,1.325,1.211,"46.8%"])
x.add_row(["onehotEncoding\n","Stacking(NB+LR+SVM)",0.663,1.177,1.081,"36.24%"
])
x.add_row(["onehotEncoding\n","Voting Class(NB+LR+SVM)",0.910,1.237,1.52,"33.9
9%"])

print(x)
```

```
+---------------+-------------------------+---------------+---------+------
-----+--------------------+
|   vectoriser   |          Model          | train logloss | CV loss | test
loss | mis-classification |
+---------------+-------------------------+---------------+---------+------
-----+--------------------+
| onehotEncoding |            NB           |      0.9      |   1.27  |    1.
21   |       39.84%       |
|                |                         |               |         |
|                    |
| ResponseCoding |           KNN           |     0.705     |   1.3   |    10
02   |       39.47%       |
|                |                         |               |         |
|                    |
| onehotEncoding |       LR+Balacing       |      0.61     |   1.14  |   1.0
48   |       34.77%       |

|                |                         |               |         |
|                    |
| onehotEncoding |      LR+NoBalancing     |     0.628     |  1.185  |   1.0
54   |       36.28%       |
|                |                         |               |         |
|                    |
| onehotEncoding |      SVM+balancing      |     0.739     |  1.132  |   1.0
63   |       39.47%       |
```

```
|                 |                           |               |         |
|                    |
| onehotEncoding |            RF             |     0.703     |  1.192  |   1.0
97    |       36.84%       |
|                 |                           |               |         |
|                    |
| Response codin |            RF             |     0.052     |  1.325  |   1.2
11    |       46.8%        |
|                 |                           |               |         |
|                    |
| onehotEncoding |   Stacking(NB+LR+SVM)     |     0.663     |  1.177  |   1.0
81    |       36.24%       |
|                 |                           |               |         |
|                    |
| onehotEncoding | Voting Class(NB+LR+SVM)   |      0.91     |  1.237  |    1.
52    |       33.99%       |
|                 |                           |               |         |
|                    |
+----------------+---------------------------+---------------+---------+------
-----+--------------------+
```

In [29]:

```
# pretty table 2
# Assignment: Tfidf on Text feature with Only 1000 Feature

from prettytable import PrettyTable
x=PrettyTable(field_names=["vectoriser","Model","train logloss","CV loss","tes
t loss","mis-classification"])

x.add_row(["onehotEncoding\n","NB",0.52,1.21,1.16,"40.4%"])
x.add_row(["onehotEncoding\n","LR+Balacing",0.71,1.08,1.01,"36.09%"])
x.add_row(["onehotEncoding\n","LR+NoBalancing",0.43,1.12,1.01,"35.9%"])
x.add_row(["onehotEncoding\n","SVM+balancing",0.43,1.09,0.98,"34.5%"])
x.add_row(["onehotEncoding\n","RF",0.82,1.23,1.14,"40.3%"])
x.add_row(["onehotEncoding\n","Stacking(NB+LR+SVM)",0.5,1.18,1.12,"36.9%"])
x.add_row(["onehotEncoding\n","Voting Class(NB+LR+SVM)",0.82,1.20,1.16,"35.9%"
])


print(x)
```

```
+----------------+---------------------------+---------------+---------+------
-----+--------------------+
|   vectoriser   |           Model           | train logloss | CV loss | test
loss | mis-classification |
+----------------+---------------------------+---------------+---------+------
-----+--------------------+
| onehotEncoding |            NB             |      0.52     |   1.21  |    1.
16   |       40.4%        |
|                 |                           |               |         |
|                    |
| onehotEncoding |       LR+Balacing         |      0.71     |   1.08  |    1.
01   |       36.09%       |
|                 |                           |               |         |

|                    |
| onehotEncoding |      LR+NoBalancing       |      0.43     |   1.12  |    1.
01   |       35.9%        |
|                 |                           |               |         |
|                    |
| onehotEncoding |      SVM+balancing        |      0.43     |   1.09  |    0.
98   |       34.5%        |
|                 |                           |               |         |
```

```
|                  |
| onehotEncoding |            RF           |      0.82     |   1.23  |    1.
14   |       40.3%        |
|                  |                         |               |         |
|                  |
| onehotEncoding |   Stacking(NB+LR+SVM)   |      0.5      |   1.18  |    1.
12   |       36.9%        |
|                  |                         |               |         |
|                  |
| onehotEncoding | Voting Class(NB+LR+SVM) |      0.82     |   1.2   |    1.
16   |       35.9%        |
|                  |                         |               |         |
|                  |
+----------------+-------------------------+---------------+---------+------
-----+--------------------+
```

In [29]:

```
# pretty table 3
# Assignment: Feature engineering on Text feature

from prettytable import PrettyTable
x=PrettyTable(field_names=["vectoriser","feature Eng","Model","train logloss",
"CV loss","test loss"])

x.add_row(["onehotEncoding \n","unigram+bigram+trigram+4gram\n","LR",0.89,1.21
,1.19])
x.add_row(["tfidf\n","bigram only","LR",0.38,0.94,0.95])
x.add_row(["onehotEncoding\n","bigram+trigram","LR",0.93,1.22,1.19])
x.add_row(["tfidf\n","trigram only","LR",0.38,0.97,0.96])
x.add_row(["tfidf\n","4gram only","LR",0.39,0.98,0.96])
x.add_row(["tfidf\n","trigram+4gram","LR",0.39,0.98,0.96])
x.add_row(["tfidf\n","unigram+bigram+trigram+4gram\n","LR",0.39,0.96,0.95])
x.add_row(["tfidf\n","text_length as aditional feature\n","LR",0.54,1.17,1.23
])

print(x)
```

```
+----------------+----------------------------------+-------+--------------
-+---------+-----------+
|    vectoriser    |           feature Eng            | Model | train logloss
| CV loss | test loss |
+----------------+----------------------------------+-------+--------------
-+---------+-----------+
| onehotEncoding  |   unigram+bigram+trigram+4gram   |   LR  |      0.89
|   1.21  |    1.19   |
|                  |                                  |       |
|         |           |
|      tfidf       |           bigram only            |   LR  |      0.38
|   0.94  |    0.95   |
|                  |                                  |       |
|         |           |
|  onehotEncoding |          bigram+trigram          |   LR  |      0.93
|   1.22  |    1.19   |
|                  |                                  |       |
|         |           |
|      tfidf       |           trigram only           |   LR  |      0.38

|   0.97  |    0.96   |
|                  |                                  |       |
|         |           |
|      tfidf       |            4gram only            |   LR  |      0.39
|   0.98  |    0.96   |
|                  |                                  |       |
```

tfidf	trigram+4gram	LR	0.39	0.98	0.96
tfidf	unigram+bigram+trigram+4gram	LR	0.39	0.96	0.95
tfidf	text_length as aditional feature	LR	0.54	1.17	1.23

Observation

1. Best model: liner model(logistic regression) with class balancing
2. Best featuriser: tfidf bigram on Text feature
3. Model interpretability: good
4. model complexity : good
5. Best Result using best model with optimum hyperparameter: train loss=0.38, CV loss=0.94, test_loss=0.95.