Personalized cancer diagnosis

By- Ravi Krishna

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 (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 (https://www.youtube.com/watch?v=UwbuW7oK8rk)
3. https://www.youtube.com/watch?v=qxXRKVompI8 (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 (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 (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 [1]:
```
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 collections import Counter
from scipy.sparse import hstack

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
import math
from sklearn.metrics import normalized_mutual_info_score
from sklearn.ensemble import RandomForestClassifier
warnings.filterwarnings("ignore")



from sklearn import model_selection

from imblearn.over_sampling import SMOTE
from mlxtend.classifier import StackingClassifier
```

```
from sklearn.linear_model import LogisticRegression
from sklearn.multiclass import OneVsRestClassifier
```

In [2]:
```
data = pd.read_csv('training_variants.csv')
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[2]:

	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

In [3]:
```
data.shape
```

Out[3]:
```
(3321, 4)
```

In [4]:
```
# check nan value
data.isnull().sum()
```

Out[4]:
```
ID           0
Gene         0
Variation    0
Class        0
dtype: int64
```

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

Reading Text Data

In [5]:
```
# note the seprator in this file
data_text =pd.read_csv("training_text.csv",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[5]:

	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...

In [6]:
```
# replacing na values in college with No college
data_text["TEXT"].fillna("No Value", inplace = True)
```

In [7]:
```
# check the null value in data_text
data_text.isnull().sum()
```

Out[7]:
```
ID      0
TEXT    0
dtype: int64
```

3.1.3. Preprocessing of text

In [8]:
```
# 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 [9]:
```
#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")
```

```
Time took for preprocessing the text : 249.68624421411303 seconds
```

In [10]:
```
#merging both gene_variations and text data based on ID
result = pd.merge(data,data_text,on= 'ID',how = 'left')
result.head()
```

Out[10]:

	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...

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 [11]:
```
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 output 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 distribution 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

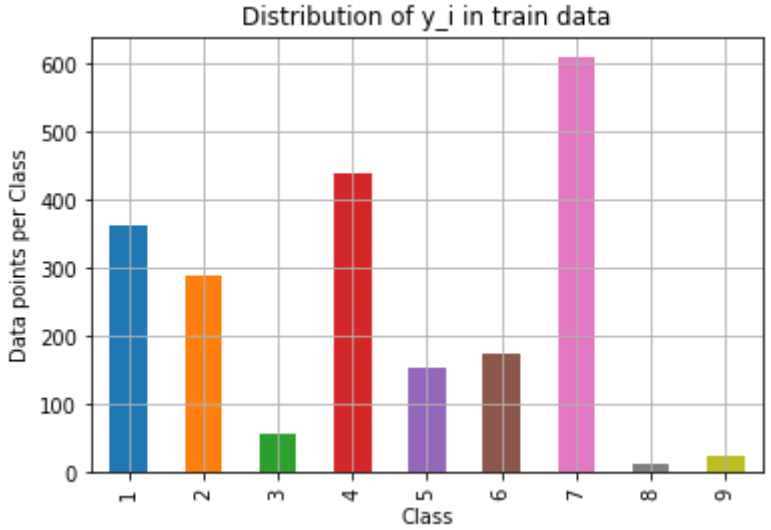
In [12]:
```
print('Number of data points in train data:', train_df.shape[0])
print('Number of data points in test data:', test_df.shape[0])
print('Number of data points in cross validation data:', cv_df.shape[0])
```

```
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
```

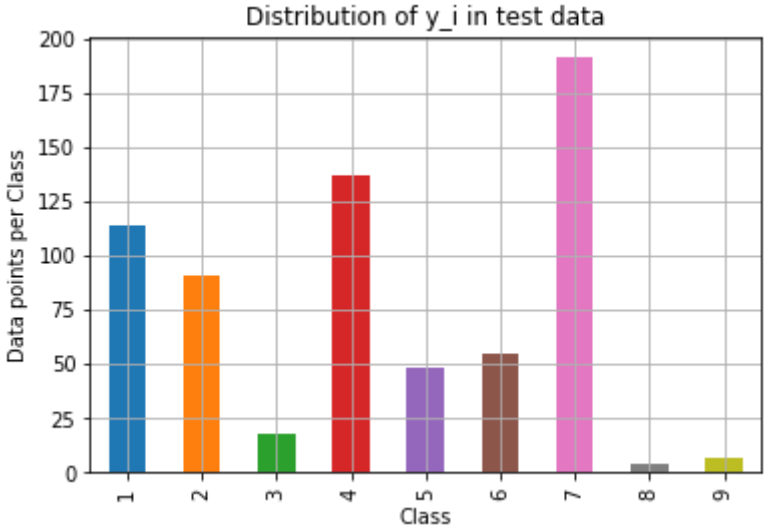
3.1.4.2. Distribution of y_i's in Train, Test and Cross Validation datasets

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

my_colors = ['r','g','b','k','y','m','c']
train_class_distribution.plot(kind='bar')
plt.xlabel('Class')
plt.ylabel('Data points per Class')
plt.title('Distribution of y_i in train 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, ':',train_class_distribution.values[i], '(', np.round((train_class_di
stribution.values[i]/train_df.shape[0]*100), 3), '%)')

    #ends

print('-'*80)
my_colors = ['r','g','b','k','y','m','c']
test_class_distribution.plot(kind='bar')
plt.xlabel('Class')
plt.ylabel('Data points per Class')
plt.title('Distribution of y_i in test data')
plt.grid()
plt.show()

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_dist
ribution.values[i]/test_df.shape[0]*100), 3), '%)')

    #ends

print('-'*80)
my_colors = ['r','g','b','k','y','m','c']
cv_class_distribution.plot(kind='bar')
```

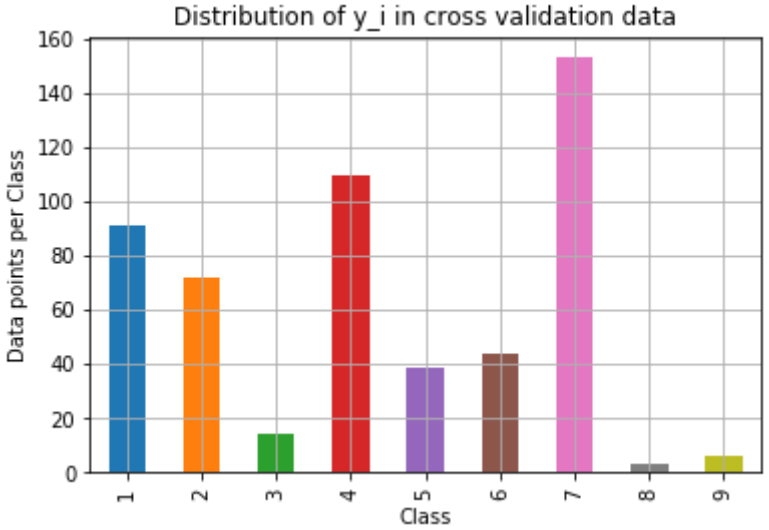
```
plt.xlabel('Class')
plt.ylabel('Data points per Class')
plt.title('Distribution of y_i in cross validation data')
plt.grid()
plt.show()

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_distribu
tion.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 [35]:
```
# 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 are predicted class j

    A =(((C.T)/(C.sum(axis=1))).T)
    #divid each element of the confusion matrix with the sum of elements in that 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 that 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 C in heatmap format
    print("-"*20, "Confusion matrix", "-"*20)
    plt.figure(figsize=(20,7))
    sns.heatmap(C, annot=True, cmap="YlGnBu", fmt=".3f", xticklabels=labels, yticklabels=labels)
    plt.xlabel('Predicted Class')
    plt.ylabel('Original Class')
    plt.show()

    # representing B in heatmap format FOR PRECISION
    print("-"*20, "Precision matrix (Columm Sum=1)", "-"*20)
```

```
    plt.figure(figsize=(20,7))
    sns.heatmap(B, annot=True, cmap="YlGnBu", fmt=".3f", xticklabels=labels, yticklabels=labels)
    plt.xlabel('Predicted Class')
    plt.ylabel('Original Class')
    plt.show()

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

In [16]:
```
# 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 their sum
# ref: https://stackoverflow.com/a/18662466/4084039

test_data_len = test_df.shape[0]
cv_data_len = cv_df.shape[0]

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))
```

```
Log loss on Cross Validation Data using Random Model 2.4827779901832203
```

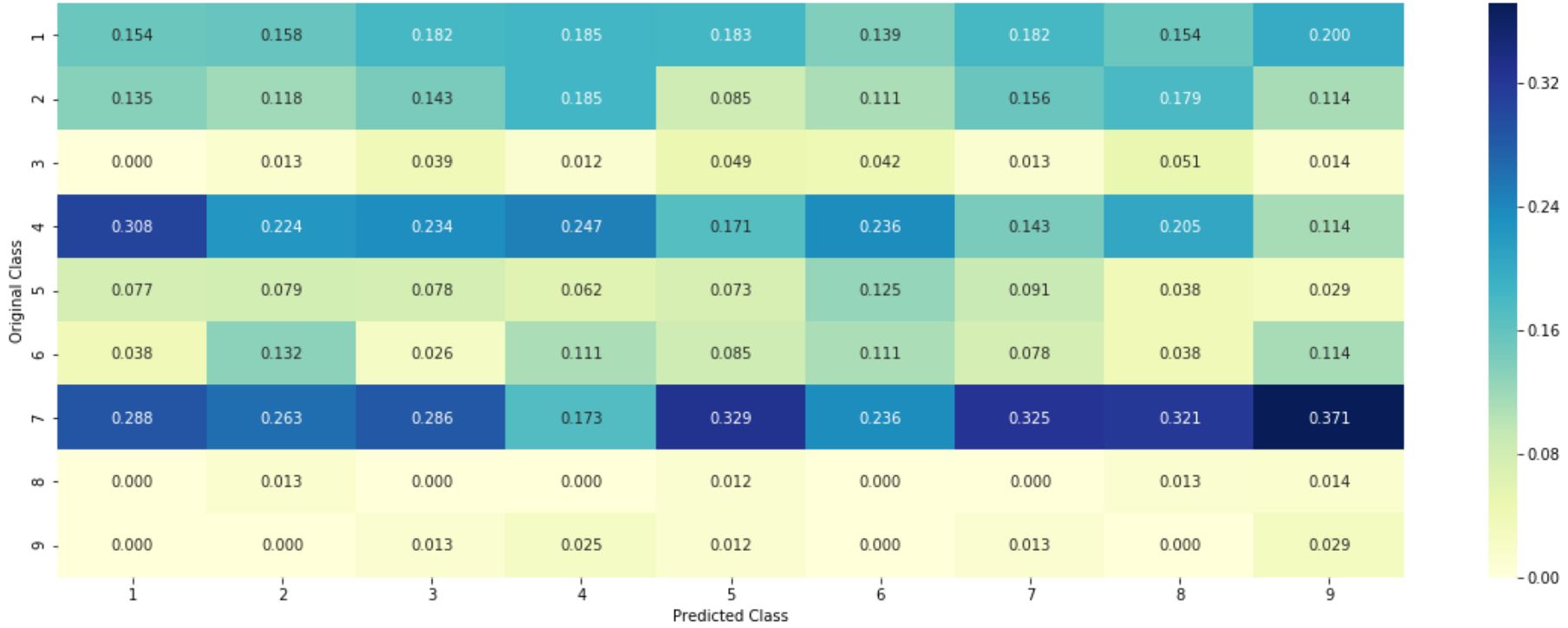
In [17]:
```
# Test-Set error.
#we create a output array that has exactly same as the test data
test_predicted_y = np.zeros((test_data_len,9))
for i in range(test_data_len):
    rand_probs = np.random.rand(1,9)
    test_predicted_y[i] = ((rand_probs/sum(sum(rand_probs)))[0])
print("Log loss on Test Data using Random Model",log_loss(y_test,test_predicted_y, eps=1e-15))
```

```
Log loss on Test Data using Random Model 2.543907507426031
```

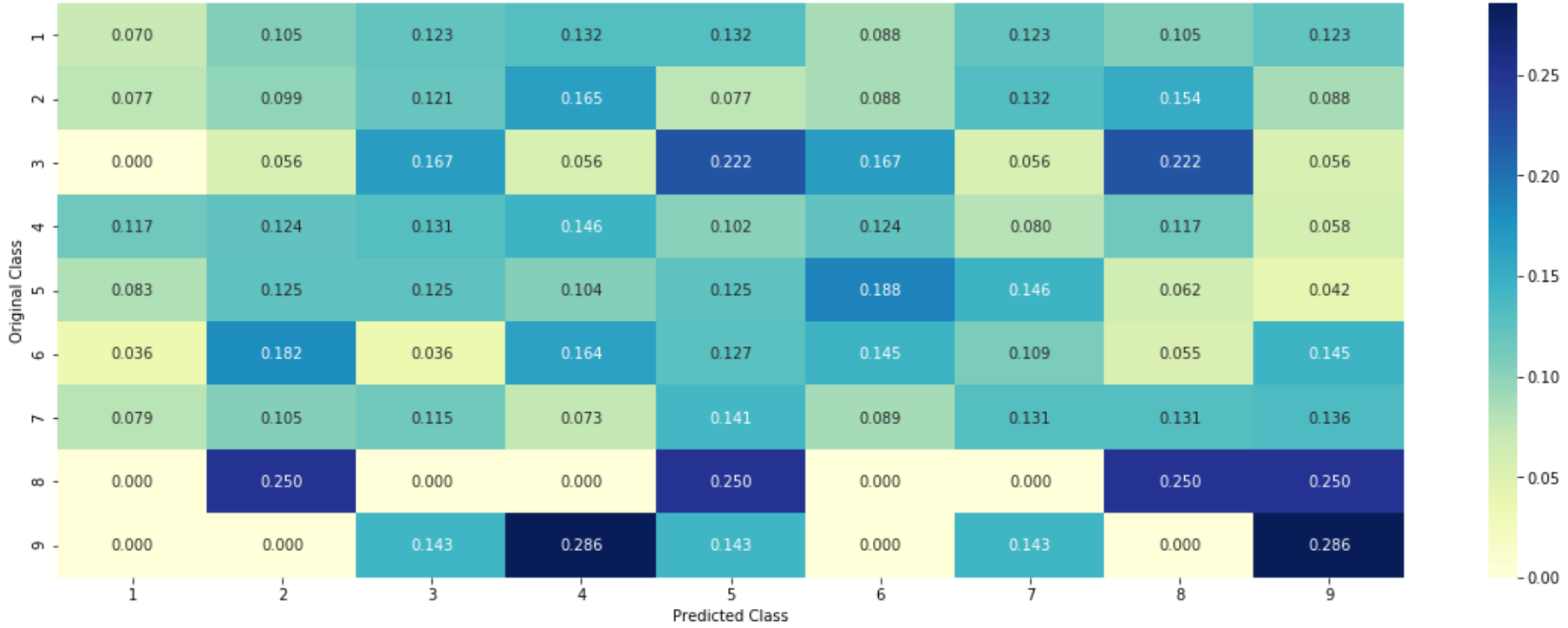
In [18]:
```
predicted_y =np.argmax(test_predicted_y, axis=1)
plot_confusion_matrix(y_test, predicted_y+1)
```

```
-------------------- 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 class1 + 10*alpha / number of time it occu
rred in total data+90*alpha)
# gv_dict is like a look up table, for every gene it store a (1*9) representation 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 occured 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 dat
a
            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, 0.068181818181818177, 0.13636363636363635, 0.25, 0.1931
```

```
8181818181818, 0.03787878787878788, 0.03787878787878788, 0.03787878787878788],
    #      'TP53': [0.32142857142857145, 0.061224489795918366, 0.061224489795918366, 0.27040816326530615, 0.0612244897
95918366, 0.066326530612244902, 0.051020408163265307, 0.051020408163265307, 0.056122448979591837],
    #      'EGFR': [0.056818181818181816, 0.21590909090909091, 0.0625, 0.068181818181818177, 0.068181818181818177, 0.0
625, 0.34659090909090912, 0.0625, 0.056818181818181816],
    #      'BRCA2': [0.13333333333333333, 0.060606060606060608, 0.060606060606060608, 0.078787878787878782, 0.13939393
93939394, 0.34545454545454546, 0.060606060606060608, 0.060606060606060608, 0.060606060606060608],
    #      'PTEN': [0.069182389937106917, 0.062893081761006289, 0.069182389937106917, 0.46540880503144655, 0.075471698
113207544, 0.062893081761006289, 0.069182389937106917, 0.062893081761006289, 0.062893081761006289],
    #      'KIT': [0.066225165562913912, 0.25165562913907286, 0.072847682119205295, 0.072847682119205295, 0.0662251655
62913912, 0.066225165562913912, 0.27152317880794702, 0.066225165562913912, 0.066225165562913912],
    #      'BRAF': [0.066666666666666666, 0.17999999999999999, 0.073333333333333334, 0.073333333333333334, 0.093333333
333333338, 0.080000000000000002, 0.29999999999999999, 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 feature 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?

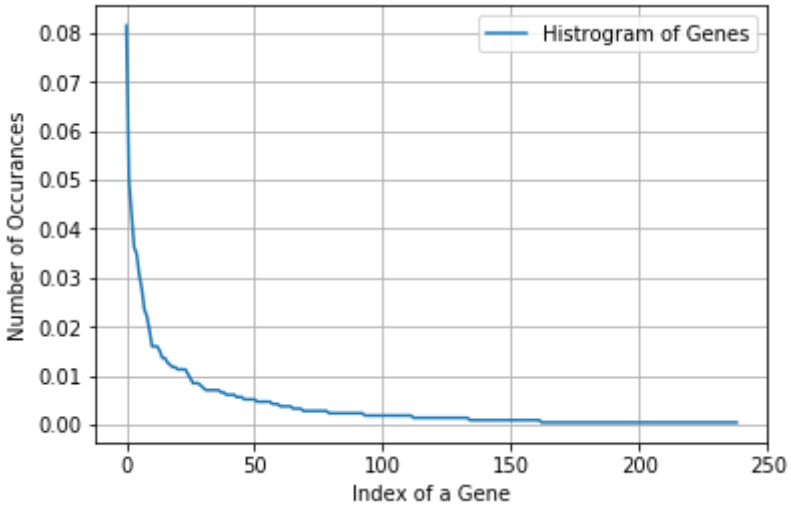
In [18]:
```
unique_genes = train_df['Gene'].value_counts()
print('Number of Unique Genes :', unique_genes.shape[0])
# the top 10 genes that occured most
print(unique_genes.head(10))
```

```
Number of Unique Genes : 233
BRCA1     170
TP53      103
EGFR       95
PTEN       85
BRCA2      83
KIT        63
BRAF       59
ERBB2      51
PIK3CA     39
PDGFRA     36
Name: Gene, dtype: int64
```

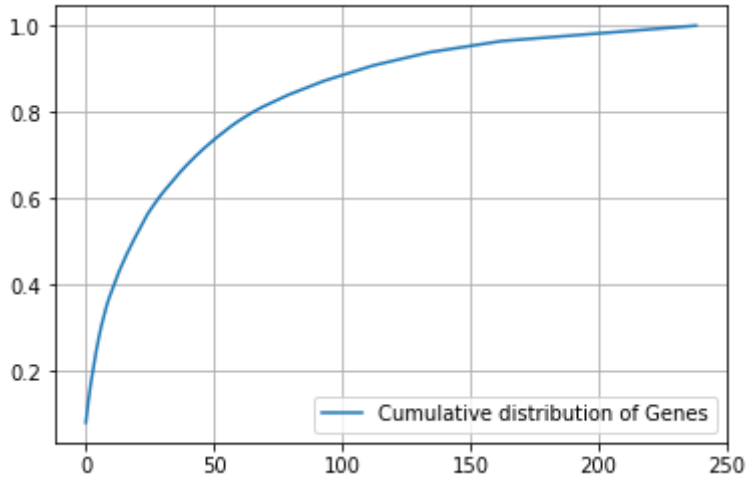
In [129]:
```
print("Ans: There are", unique_genes.shape[0] ,"different categories of genes in the train data, and they are distibut
ed as follows",)
```

```
Ans: There are 239 different categories of genes in the train data, and they are distibuted as follows
```

In [130]:
```
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 [131]:
```
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

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 [15]:
```
#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", train_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 [21]:
```
print("train_gene_feature_responseCoding is converted feature using respone coding method. The shape of gene feature:"
, train_gene_feature_responseCoding.shape)
```

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

Assignment Section

1. Task 1, Use TfidfVectorizer for all models

2. Task 2, Use top 1000 words for tfidf

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

4. Task 4, Apply feature engineering that gives log loss less than 1

1. Task 1, Use TfidfVectorizer for all models

In [26]:
```
# one-hot encoding of Gene feature.
gene_vectorizer = TfidfVectorizer(ngram_range=(1,1))#unigrams
train_gene_feature_onehotCoding = gene_vectorizer.fit_transform(train_df['Gene'])
test_gene_feature_onehotCoding = gene_vectorizer.transform(test_df['Gene'])
cv_gene_feature_onehotCoding = gene_vectorizer.transform(cv_df['Gene'])
```

In [20]:
```
train_df['Gene'].head(10)
```

Out[20]:
```
2153     PTEN
1773    XRCC2
2882    BRCA2
2538    BRCA1
1306     MLH1
249      EGFR
107      MSH6
146      EGFR
1035     TSC2
2414    PTPRD
Name: Gene, dtype: object
```

In [25]:
```
gene_vectorizer.get_feature_names()
```

```
Out[25]: ['abl1',
          'acvr1',
          'ago2',
          'akt1',
          'akt2',
          'akt3',
          'alk',
          'apc',
          'ar',
          'araf',
          'arid1a',
          'arid1b',
          'arid2',
          'arid5b',
          'asxl1',
          'atm',
          'atrx',
          'aurka',
          'aurkb',
          'axl',
          'b2m',
          'bap1',
          'bard1',
          'bcl2',
          'bcl2l11',
          'bcor',
          'braf',
          'brca1',
          'brca2',
          'brd4',
          'brip1',
          'btk',
          'card11',
          'carm1',
          'casp8',
          'cbl',
          'ccnd1',
          'ccnd2',
          'ccnd3',
          'ccne1',
          'cdh1',
```

```
'cdk12',
'cdk4',
'cdk6',
'cdkn1a',
'cdkn1b',
'cdkn2a',
'cdkn2b',
'cdkn2c',
'chek2',
'cic',
'crebbp',
'ctcf',
'ctla4',
'ctnnb1',
'ddr2',
'dicer1',
'dnmt3a',
'dnmt3b',
'dusp4',
'egfr',
'eif1ax',
'elf3',
'ep300',
'epas1',
'epcam',
'erbb2',
'erbb3',
'erbb4',
'ercc2',
'ercc3',
'ercc4',
'erg',
'errfi1',
'esr1',
'etv1',
'etv6',
'ewsr1',
'ezh2',
'fanca',
'fancc',
'fat1',
'fbxw7',
```

```
'fgfr1',
'fgfr2',
'fgfr3',
'fgfr4',
'flt1',
'flt3',
'foxa1',
'foxl2',
'foxo1',
'fubp1',
'gata3',
'gli1',
'gnas',
'h3f3a',
'hist1h1c',
'hla',
'hnf1a',
'hras',
'idh1',
'idh2',
'igf1r',
'ikzf1',
'il7r',
'inpp4b',
'jak1',
'jak2',
'jun',
'kdm5c',
'kdm6a',
'kdr',
'keap1',
'kit',
'kmt2a',
'kmt2b',
'kmt2c',
'knstrn',
'kras',
'lats2',
'map2k1',
'map2k2',
'map2k4',
'map3k1',
```

```
'mapk1',
'mdm4',
'med12',
'mef2b',
'men1',
'met',
'mga',
'mlh1',
'mpl',
'msh2',
'msh6',
'mtor',
'myc',
'mycn',
'myd88',
'ncor1',
'nf1',
'nf2',
'nfe2l2',
'nfkbia',
'nkx2',
'notch1',
'npm1',
'nras',
'nsd1',
'ntrk1',
'ntrk2',
'ntrk3',
'nup93',
'pak1',
'pax8',
'pbrm1',
'pdgfra',
'pdgfrb',
'pik3ca',
'pik3cb',
'pik3cd',
'pik3r1',
'pik3r2',
'pik3r3',
'pim1',
'pms1',
```

```
'pms2',
'pole',
'ppm1d',
'ppp2r1a',
'ppp6c',
'prdm1',
'ptch1',
'pten',
'ptpn11',
'ptprd',
'ptprt',
'rab35',
'rac1',
'rad21',
'rad50',
'rad51b',
'rad51c',
'raf1',
'rasa1',
'rb1',
'rbm10',
'ret',
'rheb',
'rhoa',
'rictor',
'rit1',
'ros1',
'runx1',
'rxra',
'rybp',
'sdhb',
'sdhc',
'setd2',
'sf3b1',
'shoc2',
'shq1',
'smad2',
'smad3',
'smad4',
'smarca4',
'smarcb1',
'smo',
```

```
'sos1',
'sox9',
'spop',
'src',
'srsf2',
'stat3',
'stk11',
'tert',
'tet1',
'tet2',
'tgfbr1',
'tgfbr2',
'tmprss2',
'tp53',
'tp53bp1',
'tsc1',
'tsc2',
'u2af1',
'vhl',
'xpo1',
'xrcc2',
'yap1']
```

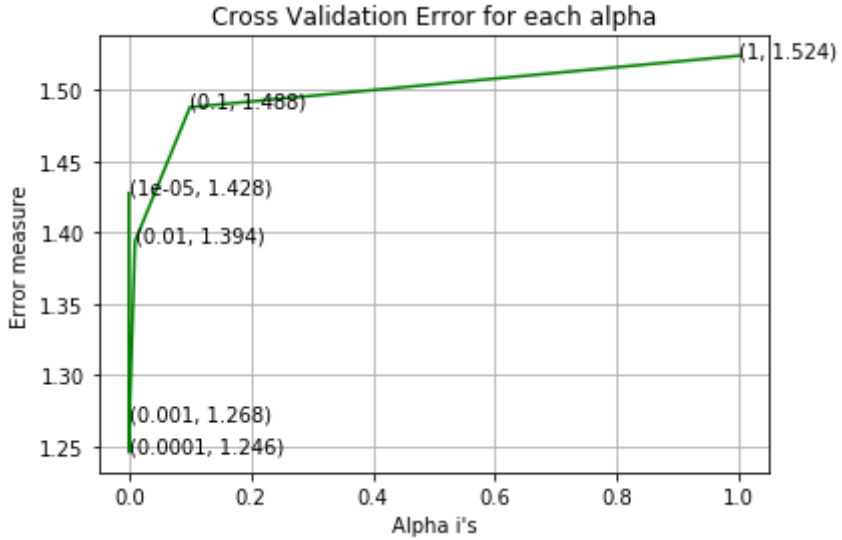
Q4. How good is this gene feature in predicting y_i?

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 [26]:
```
alpha = [10 ** x for x in range(-5, 1)] # hyperparam for SGD classifier.

# read more about SGDClassifier() at http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html
# ------------------------------
# default parameters
# SGDClassifier(loss='hinge', penalty='l2', alpha=0.0001, l1_ratio=0.15, fit_intercept=True, max_iter=None, tol=None,
# shuffle=True, verbose=0, epsilon=0.1, n_jobs=1, random_state=None, learning_rate='optimal', eta0=0.0, power_t=0.5,
# class_weight=None, warm_start=False, average=False, n_iter=None)

# some of methods
# fit(X, y[, coef_init, intercept_init, ...])     Fit linear model with Stochastic Gradient Descent.
# predict(X)    Predict class labels for samples in X.


cv_log_error_array = []

for i in alpha:
    clf = SGDClassifier(loss='log',penalty='l2',alpha = i,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_, eps=1e-15))
    print('For values of alpha = ', i, "The log loss is:",log_loss(y_cv, predict_y, labels=clf.classes_, eps=1e-15))
```

```
For values of alpha =  1e-05 The log loss is: 1.4277066483293113
For values of alpha =  0.0001 The log loss is: 1.2458590683769941
For values of alpha =  0.001 The log loss is: 1.2680504025912644
For values of alpha =  0.01 The log loss is: 1.393989020517406
For values of alpha =  0.1 The log loss is: 1.487952520014476
For values of alpha =  1 The log loss is: 1.5240757891010241
```

In [27]:
```
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()
```



In [28]:
```
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)
```

Out[28]:
```
CalibratedClassifierCV(base_estimator=SGDClassifier(alpha=0.0001, average=False, class_weight=None,
       early_stopping=False, epsilon=0.1, eta0=0.0, fit_intercept=True,
       l1_ratio=0.15, learning_rate='optimal', loss='log', max_iter=None,
       n_iter=None, n_iter_no_change=5, n_jobs=None, penalty='l2',
       power_t=0.5, random_state=42, shuffle=True, tol=None,
       validation_fraction=0.1, verbose=0, warm_start=False),
            cv='warn', method='sigmoid')
```

In [29]:
```
predict_y = sig_clf.predict_proba(train_gene_feature_onehotCoding)
print('For values of best alpha = ', alpha[best_alpha], "The train log loss is:",
      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 is:",
      log_loss(y_test, predict_y, labels=clf.classes_, eps=1e-15))
```

```
For values of best alpha =  0.0001 The train log loss is: 1.043717620447401
For values of best alpha =  0.0001 The cross validation log loss is: 1.2458590683769941
For values of best alpha =  0.0001 The test log loss is: 1.236494812986371
```

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], ":",(test_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  232  genes in train dataset?
Ans
1. In test data 646 out of 665 : 97.14285714285714
2. In cross validation data 515 out of  532 : 96.80451127819549
```

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?

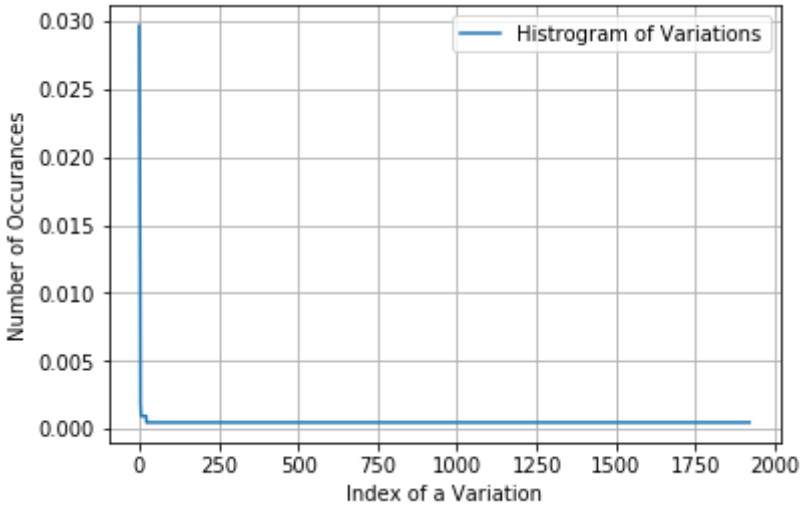
In [33]:
```
unique_variations = train_df['Variation'].value_counts()
print('Number of Unique Variations :', unique_variations.shape[0])
# the top 10 variations that occured most
print(unique_variations.head(10))
```

```
Number of Unique Variations : 1921
Truncating_Mutations    63
Deletion                52
Amplification           46
Fusions                 24
G12V                     4
Overexpression           3
T58I                     2
ETV6-NTRK3_Fusion        2
P130S                    2
Q61H                     2
Name: Variation, dtype: int64
```

In [34]:
```
print("Ans: There are", unique_variations.shape[0] ,"different categories of variations in the train data, and they ar
e distibuted as follows",)
```

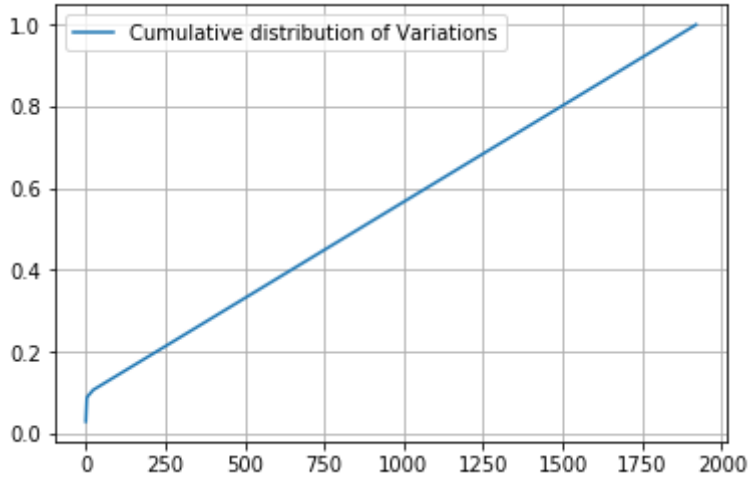
```
Ans: There are 1921 different categories of variations in the train data, and they are distibuted as follows
```

In [35]:
```
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 [36]:
```
c = np.cumsum(h)
print(c)
plt.plot(c,label='Cumulative distribution of Variations')
plt.grid()
plt.legend()
plt.show()
```

```
[0.02966102 0.05414313 0.07580038 ... 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 [16]:
```
# 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 [38]:
```
print("train_variation_feature_responseCoding is a converted feature using the response coding method. The shape of Va
riation feature:", train_variation_feature_responseCoding.shape)
```

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

In [28]:
```
# one-hot encoding of variation feature.
variation_vectorizer = TfidfVectorizer(ngram_range=(1,1))
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'])
```

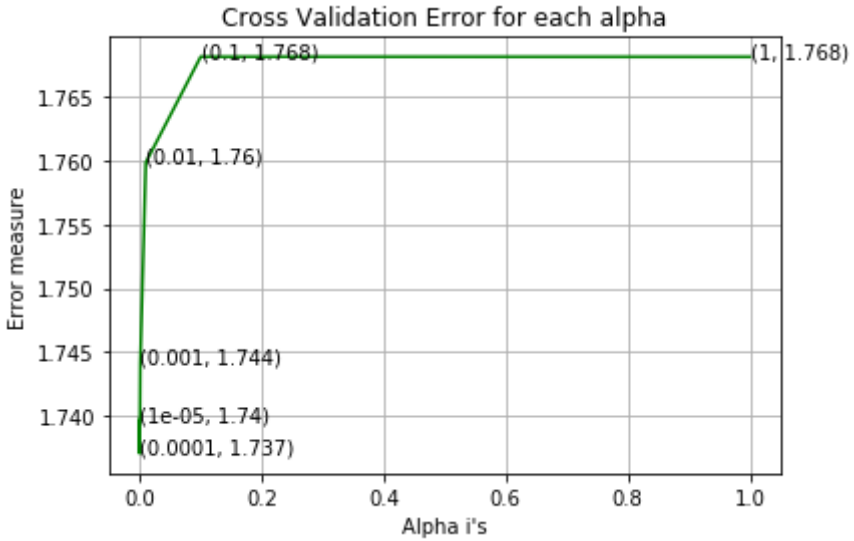
Q10. How good is this Variation feature in predicting y_i?

Let's build a model just like the earlier!

In [40]:
```
alpha = [10 ** x for x in range(-5, 1)]

# read more about SGDClassifier() at http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html
# ------------------------------
# default parameters
# SGDClassifier(loss='hinge', penalty='l2', alpha=0.0001, l1_ratio=0.15, fit_intercept=True, max_iter=None, tol=None,
# shuffle=True, verbose=0, epsilon=0.1, n_jobs=1, random_state=None, learning_rate='optimal', eta0=0.0, power_t=0.5,
# class_weight=None, warm_start=False, average=False, n_iter=None)

# some of methods
# fit(X, y[, coef_init, intercept_init, ...])     Fit linear model with Stochastic Gradient Descent.
# predict(X)    Predict class labels for samples in X.


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

```
For values of alpha =  1e-05 The log loss is: 1.7396563641234508
For values of alpha =  0.0001 The log loss is: 1.7370313404242068
For values of alpha =  0.001 The log loss is: 1.7440809096266048
For values of alpha =  0.01 The log loss is: 1.7598369532967972
For values of alpha =  0.1 The log loss is: 1.7681377406463816
For values of alpha =  1 The log loss is: 1.7681269672786877
```

In [41]:
```
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()
```



In [42]:
```
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)
```

Out[42]:
```
CalibratedClassifierCV(base_estimator=SGDClassifier(alpha=0.0001, average=False, class_weight=None,
       early_stopping=False, epsilon=0.1, eta0=0.0, fit_intercept=True,
       l1_ratio=0.15, learning_rate='optimal', loss='log', max_iter=None,
       n_iter=None, n_iter_no_change=5, n_jobs=None, penalty='l2',
       power_t=0.5, random_state=42, shuffle=True, tol=None,
       validation_fraction=0.1, verbose=0, warm_start=False),
            cv='warn', method='sigmoid')
```

In [43]:
```
predict_y = sig_clf.predict_proba(train_variation_feature_onehotCoding)
print('For values of best alpha = ', alpha[best_alpha], "The train log loss is:",
      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 is:",
      log_loss(y_test, predict_y, labels=clf.classes_, eps=1e-15))
```

```
For values of best alpha =  0.0001 The train log loss is: 0.6920637989525468
For values of best alpha =  0.0001 The cross validation log loss is: 1.7370313404242068
For values of best alpha =  0.0001 The test log loss is: 1.7214012496402096
```

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 [44]:
```
print("Q12. How many data points are covered by total ", unique_variations.shape[0], " genes in test and cross validat
ion 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'])))].shape[0]
print('Ans\n1. In test data',test_coverage, 'out of',test_df.shape[0], ":",(test_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  1921  genes in test and cross validation data sets?
Ans
1. In test data 69 out of 665 : 10.37593984962406
2. In cross validation data 44 out of  532 : 8.270676691729323
```

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 [17]:
```
# 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 [18]:
```
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_dict.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
```

In [19]:
```
# building a CountVectorizer with all the words that occured minimum 3 times in train data
text_vectorizer = TfidfVectorizer(min_df=3,ngram_range=(1,1),max_features=2000)
train_text_feature_onehotCoding = 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 returns (1*number of features) vector
train_text_fea_counts = train_text_feature_onehotCoding.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 : 2000
```

In [20]:
```
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 [21]:
```
#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 [22]:
```
# 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_text_feature_responseCoding.sum(axis=1)).T
cv_text_feature_responseCoding = (cv_text_feature_responseCoding.T/cv_text_feature_responseCoding.sum(axis=1)).T
```

In [23]:
```
# don't forget to normalize every feature
train_text_feature_onehotCoding = normalize(train_text_feature_onehotCoding, axis=0)

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

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

In [52]:
```
#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 [53]:
```
# Number of words for a given frequency.
print(Counter(sorted_text_occur))
```

```
Counter({207.02733506161096: 1, 139.43965395835238: 1, 123.59372391819772: 1, 109.42257777788723: 1, 99.8506854678345
4: 1, 96.55480011903177: 1, 95.52386119013462: 1, 95.30624606406973: 1, 94.37631614906044: 1, 89.33510884988704: 1, 8
5.81518295101024: 1, 77.29038635783971: 1, 76.35742670209501: 1, 74.72520345134006: 1, 72.23123366133166: 1, 71.21857
60548949: 1, 65.17344301236898: 1, 64.46236595245013: 1, 64.22916323071577: 1, 63.647806497780394: 1, 61.162427022487
46: 1, 61.044274187137006: 1, 56.91851281239383: 1, 55.6331310634885: 1, 55.581586188304755: 1, 55.120617487644544:
1, 54.90952484048327: 1, 54.14641792492305: 1, 53.40068655970865: 1, 52.676958926775235: 1, 52.383904388637056: 1, 5
2.10085639498383: 1, 51.519940811851164: 1, 51.25364636251695: 1, 49.32555075541631: 1, 48.1386722821889: 1, 47.81297
19735249: 1, 46.65292310429546: 1, 45.6689721490592: 1, 45.53383564346477: 1, 45.292669197368234: 1, 43.8294271712204
2: 1, 43.38696654220223: 1, 43.31212880485689: 1, 42.814661823262604: 1, 40.93391813015982: 1, 40.1227370224938: 1, 3
9.456275993286006: 1, 37.905046517049634: 1, 37.881634091589184: 1, 37.566586165414925: 1, 37.45964405177299: 1, 36.3
7757403812752: 1, 36.24242341850244: 1, 35.9863839417224: 1, 35.69256478137128: 1, 35.44732775410572: 1, 35.218773330
25185: 1, 34.94582352179562: 1, 34.67160682925671: 1, 34.65214691796747: 1, 34.63166202188562: 1, 33.96372865184151:
1, 33.63817020492576: 1, 33.611615429053195: 1, 33.4814143282298: 1, 33.437407940868084: 1, 33.31686972096057: 1, 33.
25938969631703: 1, 33.16905856166982: 1, 32.90213047430313: 1, 31.852376382888448: 1, 31.81201703695193: 1, 31.524078
922977434: 1, 31.47871968993798: 1, 31.473416366758563: 1, 31.113035057210322: 1, 30.53644912976458: 1, 30.2979926394
7597: 1, 30.23196088895165: 1, 29.657917763477275: 1, 29.293235884852987: 1, 29.22672426985134: 1, 29.04748036952437
6: 1, 28.990673542111693: 1, 28.786058443330138: 1, 28.631338815592642: 1, 28.549327550649043: 1, 28.35465239622323:
1, 27.794019523220097: 1, 27.652822546061387: 1, 27.529364623284767: 1, 27.39948723054621: 1, 27.3700848278186: 1, 2
7.112262614606394: 1, 27.06978666731181: 1, 26.931742739834917: 1, 26.877809155771796: 1, 26.7519407367612: 1, 26.675
29814734531: 1, 26.669870391202792: 1, 26.665025931761235: 1, 26.611257492536897: 1, 26.581424924746628: 1, 26.430945
28068823: 1, 26.11254633647296: 1, 26.109551042885048: 1, 26.000399436385695: 1, 25.86716357102837: 1, 25.72252349538
549: 1, 25.583115600878475: 1, 25.52713329734503: 1, 25.351986775823534: 1, 25.213759944174043: 1, 25.16519354423457
2: 1, 25.088708113787824: 1, 25.058837602076867: 1, 24.838321955650912: 1, 24.785206965230106: 1, 24.71779838606812:
1, 24.672179133580283: 1, 24.388405582093906: 1, 24.302927064690387: 1, 24.29179724753713: 1, 24.241568230818054: 1,
24.14844137800596: 1, 24.119954112665578: 1, 24.078168598399653: 1, 24.075112962313593: 1, 23.95016231855185: 1, 23.8
58400271806783: 1, 23.809882696348808: 1, 23.611697905805446: 1, 23.571365578161107: 1, 23.56560787553969: 1, 23.4649
8029559531: 1, 23.297251631150946: 1, 22.798325198251842: 1, 22.796821520690834: 1, 22.516746503372726: 1, 22.3878796
0520732: 1, 22.28245080697957: 1, 22.22193996248504: 1, 22.20234645741347: 1, 22.054956687998438: 1, 21.8940470001969
24: 1, 21.80365347618769: 1, 21.572522112445373: 1, 21.457396062565717: 1, 21.435774051670375: 1, 21.321799551091896:
1, 21.31468572559492: 1, 21.305549240968308: 1, 21.182103736325892: 1, 21.029103372077554: 1, 20.995445138743374: 1,
20.977313504691182: 1, 20.824085338376275: 1, 20.768399730380334: 1, 20.644550540062188: 1, 20.63010758345227: 1, 20.
603324257053252: 1, 20.57026758291512: 1, 20.412041200952: 1, 20.308195471057115: 1, 20.26667600301239: 1, 20.1557284
91845235: 1, 20.11913343347898: 1, 20.0929143390234: 1, 20.0901058099042: 1, 20.03343638004528: 1, 19.97460477370442
8: 1, 19.972653471121493: 1, 19.934501791625355: 1, 19.925485998792105: 1, 19.858671656431127: 1, 19.841935606711434:
1, 19.753712032998145: 1, 19.723684240183637: 1, 19.46706709342951: 1, 19.41650744477349: 1, 19.396339189853844: 1, 1
9.328395813002324: 1, 19.32185986210355: 1, 19.303299582812826: 1, 19.170337581246603: 1, 19.042327563711126: 1, 19.0
17840859464794: 1, 18.9646447204634: 1, 18.923912621321296: 1, 18.91168867755904: 1, 18.822816245297265: 1, 18.758826
863415706: 1, 18.725689157146864: 1, 18.65862477305544: 1, 18.638199556130786: 1, 18.63513927084624: 1, 18.5981546577
5466: 1, 18.5569792743227: 1, 18.471929773070066: 1, 18.44601005855426: 1, 18.435585726070848: 1, 18.39094879986391:
1, 18.382948447312295: 1, 18.258983612129136: 1, 18.232695292065618: 1, 18.20352544818101: 1, 18.0083586314469: 1, 1
7.96781789367892: 1, 17.950377253942705: 1, 17.946677828112914: 1, 17.935181143104753: 1, 17.92716168069594: 1, 17.91
```

366357432469: 1, 17.897573746498168: 1, 17.89557827184761: 1, 17.87620648265208: 1, 17.780531693574602: 1, 17.7692036
85841816: 1, 17.7557328806303: 1, 17.728167829761773: 1, 17.70946021916487: 1, 17.59849674986747: 1, 17.5701083303630
6: 1, 17.554833041992495: 1, 17.544763948393918: 1, 17.507672443447948: 1, 17.44764264030471: 1, 17.426120830727417:
1, 17.422213590090017: 1, 17.38106092693728: 1, 17.322027388752733: 1, 17.31535725745116: 1, 17.294132628330317: 1, 1
7.26863155924445: 1, 17.266699040128973: 1, 17.208675587198922: 1, 17.098128443184105: 1, 17.094916107086124: 1, 17.0
56720036188157: 1, 17.05505709782781: 1, 17.022992523919793: 1, 17.022840827808654: 1, 16.875760933827095: 1, 16.8588
34720678274: 1, 16.75858234005406: 1, 16.6561127872091: 1, 16.6171255051823: 1, 16.590849254101364: 1, 16.58946479307
4825: 1, 16.561244244315816: 1, 16.521992419695092: 1, 16.50624455703905: 1, 16.422875788070595: 1, 16.39399480965578
3: 1, 16.374168424382713: 1, 16.34590738887733: 1, 16.291555458758992: 1, 16.286207949343556: 1, 16.262881161285925:
1, 16.253617986612575: 1, 16.221902296918888: 1, 16.220225897988893: 1, 16.209998188883784: 1, 16.160789951218142: 1,
16.160414874155943: 1, 16.136225469309856: 1, 16.123985962432396: 1, 16.10622458038411: 1, 15.952082224195662: 1, 15.
944631840278419: 1, 15.93763180596941: 1, 15.872163373863858: 1, 15.850335779124796: 1, 15.757332929699102: 1, 15.722
466200720413: 1, 15.710396927955475: 1, 15.688110443779552: 1, 15.65935907415538: 1, 15.648088304678403: 1, 15.587047
674341937: 1, 15.566868020319305: 1, 15.548473017944529: 1, 15.522266687416744: 1, 15.463046364305537: 1, 15.45033829
4633392: 1, 15.423744449874748: 1, 15.399905190941334: 1, 15.387087920141731: 1, 15.382268947826251: 1, 15.3616723997
64247: 1, 15.353512290115908: 1, 15.296186960555517: 1, 15.289813567405462: 1, 15.184913111893586: 1, 15.183044470188
717: 1, 15.148268202812453: 1, 15.141319472694729: 1, 15.081553282748319: 1, 14.990897079885443: 1, 14.98853672146836
8: 1, 14.95253228946864: 1, 14.948075920168327: 1, 14.912626845359974: 1, 14.89475938550285: 1, 14.88823046951093: 1,
14.823482031272206: 1, 14.777417750625855: 1, 14.688776541794253: 1, 14.663138303699196: 1, 14.638095124413095: 1, 1
4.627769673356712: 1, 14.614651218623221: 1, 14.613682808764025: 1, 14.549153016951887: 1, 14.537740350152522: 1, 14.
526665398495743: 1, 14.507045863865654: 1, 14.436200282258389: 1, 14.430148933335008: 1, 14.420764301040073: 1, 14.41
9139054545294: 1, 14.402373253810744: 1, 14.382922469206376: 1, 14.347245503335596: 1, 14.339281015857765: 1, 14.3128
89518994389: 1, 14.310250174690513: 1, 14.28419108878531: 1, 14.25806981072177: 1, 14.240800212601469: 1, 14.23640198
3041947: 1, 14.226118661012018: 1, 14.197372882328592: 1, 14.14920591157028: 1, 14.138352359466241: 1, 14.10916682981
497: 1, 14.072773642103792: 1, 14.071284732099704: 1, 14.046947074922308: 1, 14.03970957348557: 1, 13.97727760613631
2: 1, 13.946232325557204: 1, 13.943634778609772: 1, 13.84867485549852: 1, 13.842357273513176: 1, 13.82663221154436:
1, 13.789941276604525: 1, 13.744467537753158: 1, 13.73631631471863: 1, 13.71986966727405: 1, 13.572178638284468: 1, 1
3.540495820067031: 1, 13.530934824834747: 1, 13.516831139279583: 1, 13.439565992230214: 1, 13.43337139945072: 1, 13.4
25689676228618: 1, 13.391024186314604: 1, 13.36990394623678: 1, 13.36966396819527: 1, 13.362057415947131: 1, 13.35533
94499398: 1, 13.341468582969352: 1, 13.33330135127389: 1, 13.329037140430204: 1, 13.322030475954486: 1, 13.3135779994
45714: 1, 13.313153048379037: 1, 13.31312831084089: 1, 13.30522234790007: 1, 13.296807569717554: 1, 13.28467603791869
3: 1, 13.266775890305308: 1, 13.208623811150964: 1, 13.208033410315478: 1, 13.191273920024283: 1, 13.178682922009466:
1, 13.173500051874434: 1, 13.13066701242706: 1, 13.114811695010197: 1, 13.11406575094555: 1, 13.089686612708169: 1, 1
3.088105533258709: 1, 13.086648221970256: 1, 13.062538654958262: 1, 13.04304542796269: 1, 13.033634090512065: 1, 13.0
25673539127743: 1, 13.010280045254227: 1, 12.94697495279454: 1, 12.908605444836777: 1, 12.906591687494323: 1, 12.8753
23357192487: 1, 12.871733909506535: 1, 12.835266416502066: 1, 12.833386418058678: 1, 12.818420592188037: 1, 12.780956
71413106: 1, 12.779732754957347: 1, 12.713615252221803: 1, 12.711330333775832: 1, 12.706686860857744: 1, 12.703057769
151798: 1, 12.692243681422434: 1, 12.687500153956062: 1, 12.678049652481453: 1, 12.667721676463144: 1, 12.66510825417
0692: 1, 12.633164310732663: 1, 12.625930525330046: 1, 12.5816504678913: 1, 12.56035851182093: 1, 12.555562350821637:
1, 12.54919709026242: 1, 12.523064816062229: 1, 12.519270781038681: 1, 12.47249042035395: 1, 12.472006581030058: 1, 1
2.463420284670608: 1, 12.458827481759768: 1, 12.444107172081006: 1, 12.414639548255352: 1, 12.409093153252782: 1, 12.
364010803320445: 1, 12.35201245868551: 1, 12.327568174734328: 1, 12.313065917059662: 1, 12.276281453174064: 1, 12.261

738483432561: 1, 12.249807708775675: 1, 12.23045725731126: 1, 12.224474368492954: 1, 12.21409953766735: 1, 12.2000548
41339194: 1, 12.193481449820322: 1, 12.18936827557849: 1, 12.176805408011413: 1, 12.146309841442445: 1, 12.1163512499
0661: 1, 12.111812994137553: 1, 12.072147344128581: 1, 12.032623696001556: 1, 12.030192368647388: 1, 11.9562112129222
79: 1, 11.950094238065125: 1, 11.941122446979605: 1, 11.92809989439896: 1, 11.905456167713721: 1, 11.88412026253066:
1, 11.882440864337623: 1, 11.868561809058127: 1, 11.860050892707648: 1, 11.855016225945874: 1, 11.81283440815519: 1,
11.809864485483216: 1, 11.772537769066856: 1, 11.743750814812339: 1, 11.740720092284876: 1, 11.73948331427301: 1, 11.
710803357913518: 1, 11.700555796311464: 1, 11.671268652286265: 1, 11.662194031544407: 1, 11.629899385895987: 1, 11.62
4670791831173: 1, 11.610453268201855: 1, 11.608616517290354: 1, 11.597734315809811: 1, 11.593610272458381: 1, 11.5834
07720172625: 1, 11.561873416578948: 1, 11.540802239509208: 1, 11.537635320785348: 1, 11.536472948413756: 1, 11.507054
293087116: 1, 11.501842510369736: 1, 11.498044386437853: 1, 11.48976452784278: 1, 11.47499554835626: 1, 11.4709204605
06878: 1, 11.469255706307642: 1, 11.440787840456794: 1, 11.4282041177594: 1, 11.426590619135187: 1, 11.38546197908697
8: 1, 11.370554421367899: 1, 11.36904787808656: 1, 11.333499545695869: 1, 11.282732248535737: 1, 11.2795659508505: 1,
11.250971322762181: 1, 11.238512288413627: 1, 11.210124396626846: 1, 11.205748327562427: 1, 11.191450144409925: 1, 1
1.190504780002552: 1, 11.175530092247302: 1, 11.167197970259789: 1, 11.152653024144458: 1, 11.123756253829846: 1, 11.
117954111039852: 1, 11.116852030729387: 1, 11.109168460658392: 1, 11.090051055039856: 1, 11.086672862126518: 1, 11.07
9352232290653: 1, 11.055695268415384: 1, 11.043019614357544: 1, 11.034622818652966: 1, 11.026557034346084: 1, 10.9939
27653371872: 1, 10.951841649503747: 1, 10.943457422446203: 1, 10.895594844190018: 1, 10.88657044838635: 1, 10.8682077
03526217: 1, 10.866241129469733: 1, 10.828976544870105: 1, 10.815905783492314: 1, 10.814792284767401: 1, 10.776012895
310236: 1, 10.754856819392113: 1, 10.742656428798217: 1, 10.730859138603027: 1, 10.717887284696944: 1, 10.71276286980
2303: 1, 10.703466787568729: 1, 10.69486053931201: 1, 10.693909718657807: 1, 10.684744270956568: 1, 10.6733153309963
3: 1, 10.652332175950438: 1, 10.64471483655236: 1, 10.634161109924515: 1, 10.633179804895766: 1, 10.609230144523252:
1, 10.598288051717352: 1, 10.597068198157595: 1, 10.583716958108026: 1, 10.566103016309441: 1, 10.537969965591502: 1,
10.530551402260183: 1, 10.515462577625271: 1, 10.51280205417492: 1, 10.502624802278847: 1, 10.488188682849687: 1, 10.
462370280774053: 1, 10.445167825939274: 1, 10.430502672525218: 1, 10.42471470495752: 1, 10.423103670008015: 1, 10.411
880054701813: 1, 10.40911896332033: 1, 10.377447783461575: 1, 10.350924897891984: 1, 10.34453239635855: 1, 10.3419775
78161135: 1, 10.335399578141438: 1, 10.321241493165173: 1, 10.318892826244813: 1, 10.300210386924606: 1, 10.258843208
185136: 1, 10.245610129183294: 1, 10.20015663388233: 1, 10.197224078112855: 1, 10.16863653199792: 1, 10.1101515363527
03: 1, 10.106119847183665: 1, 10.103093236526846: 1, 10.100129725017018: 1, 10.09669576400224: 1, 10.092749726140086:
1, 10.083497578275718: 1, 10.080301188238625: 1, 10.077609122301556: 1, 10.077456845781274: 1, 10.068383239973143: 1,
10.038483205918478: 1, 10.035292803832506: 1, 10.018524225417591: 1, 9.999642324232301: 1, 9.99562300296502: 1, 9.992
45198552268: 1, 9.989016146338951: 1, 9.960260948118469: 1, 9.956631099771736: 1, 9.94304506532438: 1, 9.936486411708
534: 1, 9.920523528001638: 1, 9.910897458658534: 1, 9.90990326794901: 1, 9.899784666420748: 1, 9.86695325336572: 1,
9.862239759700984: 1, 9.847415108181847: 1, 9.825725725057914: 1, 9.818074169025135: 1, 9.722512301257735: 1, 9.71010
2242262195: 1, 9.707240158013377: 1, 9.692506836686428: 1, 9.664159923248166: 1, 9.65497060216983: 1, 9.6387456536309
75: 1, 9.615562823481085: 1, 9.608333071225763: 1, 9.607402118163408: 1, 9.603841381944562: 1, 9.603457210451767: 1,
9.59838785537357: 1, 9.591200707302542: 1, 9.58578029092909: 1, 9.57779383969498: 1, 9.558126962444986: 1, 9.53971460
1704375: 1, 9.534845495654286: 1, 9.533008115929608: 1, 9.502659582774092: 1, 9.488784427436055: 1, 9.48173405591905
5: 1, 9.458440260409576: 1, 9.454106818205254: 1, 9.449168830797618: 1, 9.442700224541047: 1, 9.428495028227259: 1,
9.42801355760389: 1, 9.425448019516699: 1, 9.413406263703337: 1, 9.410510978325812: 1, 9.409688607964615: 1, 9.381761
079643852: 1, 9.377155380035923: 1, 9.36865484818303: 1, 9.36725717657489: 1, 9.350679282560813: 1, 9.33824121963136
2: 1, 9.321114850970195: 1, 9.31967347520972: 1, 9.317085164935257: 1, 9.316615842107181: 1, 9.306957087852792: 1, 9.
302127252411003: 1, 9.302121959425639: 1, 9.296117820332288: 1, 9.29130553306949: 1, 9.281538879711558: 1, 9.26751782

5360676: 1, 9.238490282175716: 1, 9.228619927363306: 1, 9.225477911971925: 1, 9.223650456655317: 1, 9.20811066207455
3: 1, 9.206969706046817: 1, 9.202147087829704: 1, 9.183002666017707: 1, 9.170472050386426: 1, 9.167904703225165: 1,
9.157425564321114: 1, 9.131787651189278: 1, 9.129349655934602: 1, 9.120671902800938: 1, 9.118821727742075: 1, 9.10912
041207267: 1, 9.05179788678398: 1, 9.049040844051833: 1, 9.026817084025442: 1, 9.01707970527872: 1, 8.99803069875870
4: 1, 8.989646125635847: 1, 8.978103784903416: 1, 8.97708817165743: 1, 8.959711721334058: 1, 8.947818837311187: 1, 8.
939787090656674: 1, 8.916925414614468: 1, 8.908887936365497: 1, 8.907728081660231: 1, 8.891434909318892: 1, 8.8858275
36022324: 1, 8.880219930003111: 1, 8.87943432923207: 1, 8.873361446520418: 1, 8.872865459032154: 1, 8.86312591598492
2: 1, 8.837178468022419: 1, 8.83336949925866: 1, 8.83059547436948: 1, 8.823449313261069: 1, 8.812016389752342: 1, 8.8
0766025980011: 1, 8.807115528906673: 1, 8.804202442546744: 1, 8.80174076589241: 1, 8.799605074232888: 1, 8.7920901903
2436: 1, 8.782495537694235: 1, 8.78069969445114: 1, 8.77891793702372: 1, 8.778207800174782: 1, 8.777463886744748: 1,
8.776445434849853: 1, 8.76787586744332: 1, 8.758280361461592: 1, 8.74299859380075: 1, 8.740119451399107: 1, 8.7364502
52986838: 1, 8.731906068708236: 1, 8.707711034205335: 1, 8.69855809260829: 1, 8.696590318770435: 1, 8.69130346209004
8: 1, 8.664939957659893: 1, 8.664492339158286: 1, 8.643244481387692: 1, 8.640138768311337: 1, 8.626041136980815: 1,
8.614573518641464: 1, 8.61322974052999: 1, 8.612726212526812: 1, 8.590096758479445: 1, 8.57927708507323: 1, 8.5711231
0766358: 1, 8.562647898846393: 1, 8.558363095333036: 1, 8.551304385031552: 1, 8.549909196303275: 1, 8.52715669949161
8: 1, 8.525399327610666: 1, 8.514962961778728: 1, 8.502817482682634: 1, 8.50080258513815: 1, 8.496126042190324: 1, 8.
4951823827355: 1, 8.493939283081128: 1, 8.485868965913395: 1, 8.474417510639896: 1, 8.471993166957335: 1, 8.469702379
554173: 1, 8.468654262850349: 1, 8.468125998826773: 1, 8.455069487202659: 1, 8.438520888401193: 1, 8.423152627437753:
1, 8.417789399621372: 1, 8.409440798715114: 1, 8.409438759419624: 1, 8.398488867966297: 1, 8.396830979961393: 1, 8.38
9634429499361: 1, 8.388909182939575: 1, 8.381910654284932: 1, 8.373606504073992: 1, 8.372811707419892: 1, 8.369927656
48504: 1, 8.36135849197295: 1, 8.357956587521748: 1, 8.354470246183187: 1, 8.35126895454216: 1, 8.348989439847937: 1,
8.347695327634469: 1, 8.33247138582945: 1, 8.332414684050931: 1, 8.330446753778523: 1, 8.325091108748232: 1, 8.310499
380034994: 1, 8.30011981567664: 1, 8.298961578588107: 1, 8.277165099571965: 1, 8.264579093332562: 1, 8.26174976866004
4: 1, 8.248722529836673: 1, 8.248367939283572: 1, 8.243588039854371: 1, 8.238695999317702: 1, 8.233863184417958: 1,
8.231851950965352: 1, 8.229737740650759: 1, 8.22455130901938: 1, 8.219716077704637: 1, 8.217917283742219: 1, 8.214072
908899688: 1, 8.209729819528748: 1, 8.204496210080576: 1, 8.202241172540218: 1, 8.18268824963425: 1, 8.16913116503963
5: 1, 8.168827780169392: 1, 8.161706980527951: 1, 8.158610715144658: 1, 8.146588393856728: 1, 8.140890252103056: 1,
8.137289092680044: 1, 8.136173201894549: 1, 8.13349125401545: 1, 8.131226397231268: 1, 8.130547429160059: 1, 8.121451
921131378: 1, 8.113851480482959: 1, 8.111013435225983: 1, 8.103278961630268: 1, 8.099849627496027: 1, 8.0966002679078
42: 1, 8.093006546047704: 1, 8.08980353218482: 1, 8.082569088644403: 1, 8.082005058238876: 1, 8.080110144610273: 1,
8.073644116505424: 1, 8.039552356052917: 1, 8.034669099383954: 1, 8.022884570556347: 1, 8.022839265503777: 1, 8.00645
910000543: 1, 8.005882187698733: 1, 8.003211110613881: 1, 7.9991246389066095: 1, 7.994268458044416: 1, 7.994011816381
8115: 1, 7.993080816850282: 1, 7.982990881498248: 1, 7.973539189523201: 1, 7.9632385856862005: 1, 7.951509549930879:
1, 7.945168491102233: 1, 7.943087901436637: 1, 7.93609097292793: 1, 7.9352190930944655: 1, 7.922040095288362: 1, 7.89
3533861952513: 1, 7.887656694930871: 1, 7.876451136433873: 1, 7.866722524353363: 1, 7.86273921311183: 1, 7.8621250785
42895: 1, 7.85984599701578: 1, 7.8509091545449206: 1, 7.84391864748679: 1, 7.84018410712274: 1, 7.835344096166625: 1,
7.824377448876952: 1, 7.797053709767206: 1, 7.795222579887679: 1, 7.791920520453497: 1, 7.783724649695494: 1, 7.75694
5868192109: 1, 7.749586519076206: 1, 7.7425202927429035: 1, 7.737217594959095: 1, 7.734035308793206: 1, 7.73175979130
258: 1, 7.729158807410229: 1, 7.72617540403948: 1, 7.722174227675708: 1, 7.717285723626459: 1, 7.701610260390765: 1,
7.701094666341178: 1, 7.693273067507912: 1, 7.692916793440256: 1, 7.6905899219613225: 1, 7.6885634884955865: 1, 7.684
743411839149: 1, 7.677468248003092: 1, 7.671901683563239: 1, 7.6701637437371035: 1, 7.668359242144081: 1, 7.663639400
277241: 1, 7.655503904188905: 1, 7.642121446530912: 1, 7.634722826366698: 1, 7.6139313024094895: 1, 7.59305469978900

2: 1, 7.570437410021775: 1, 7.558572023050311: 1, 7.5542015276198775: 1, 7.5469156247396665: 1, 7.53732477462876: 1,
7.535316528126275: 1, 7.533373323372746: 1, 7.530114416578627: 1, 7.52956027272303: 1, 7.524056381730896: 1, 7.513940
922033961: 1, 7.512164368715503: 1, 7.50726226223508: 1, 7.504193789696598: 1, 7.504027698283211: 1, 7.49640024234615
95: 1, 7.4788097648377265: 1, 7.474005204450102: 1, 7.468970349230978: 1, 7.460364656742975: 1, 7.457475623003629: 1,
7.451212235653959: 1, 7.449114399931393: 1, 7.4443635167559075: 1, 7.437281769045582: 1, 7.433106773540231: 1, 7.4311
07574272237: 1, 7.430039631476817: 1, 7.4146046234293035: 1, 7.414482164310281: 1, 7.410768631623274: 1, 7.4027011829
52282: 1, 7.39901559165906: 1, 7.397227267952933: 1, 7.386004923436125: 1, 7.377784241985818: 1, 7.377717712546819:
1, 7.37345154917062: 1, 7.3725261802763615: 1, 7.3628195243744035: 1, 7.362640094616224: 1, 7.362401706065516: 1, 7.3
543395994179575: 1, 7.344218197903971: 1, 7.335900406035097: 1, 7.324238805549109: 1, 7.318477485299223: 1, 7.3150911
17648822: 1, 7.307640581713995: 1, 7.304323169425849: 1, 7.2924845464382635: 1, 7.289408629275778: 1, 7.2871153792724
92: 1, 7.272732730428631: 1, 7.268437947192508: 1, 7.267696580236214: 1, 7.266276438077642: 1, 7.264219837426012: 1,
7.2618281412791275: 1, 7.260056501589093: 1, 7.257841180484126: 1, 7.256366577782145: 1, 7.252128387558594: 1, 7.2473
69385004752: 1, 7.234843222474524: 1, 7.227031800596335: 1, 7.219246787909529: 1, 7.2171585746007185: 1, 7.2121341455
75877: 1, 7.2107023897888585: 1, 7.2102602869073955: 1, 7.20439304295446: 1, 7.202629290287394: 1, 7.194472693309472:
1, 7.1917888163326795: 1, 7.1873480136964405: 1, 7.186765577836803: 1, 7.177679585675998: 1, 7.1678487576405825: 1,
7.167308066507802: 1, 7.166751271696343: 1, 7.163911769749954: 1, 7.160977288513875: 1, 7.158949109085324: 1, 7.14999
3572875815: 1, 7.148821052142876: 1, 7.1401011889176464: 1, 7.137161954956605: 1, 7.135185039210812: 1, 7.12842315828
9825: 1, 7.12653347574293: 1, 7.124564013602859: 1, 7.123135824170136: 1, 7.122604839622907: 1, 7.106430772818424: 1,
7.094186519520942: 1, 7.08766334900416: 1, 7.0790254820848615: 1, 7.075678048645687: 1, 7.075366685371808: 1, 7.07421
0334098523: 1, 7.050688023112589: 1, 7.040725273316: 1, 7.028332025767263: 1, 7.022914422137376: 1, 7.02291410406192
1: 1, 7.022068930477929: 1, 7.021837491015282: 1, 7.0187413179393925: 1, 7.018556667135323: 1, 7.007302990071647: 1,
7.004728949010318: 1, 6.98182468093658: 1, 6.980968847721785: 1, 6.978598244362853: 1, 6.975130967740427: 1, 6.970518
265904324: 1, 6.955798741262404: 1, 6.940213888982808: 1, 6.940180333325123: 1, 6.93905625746122: 1, 6.93549880917014
5: 1, 6.926247976146128: 1, 6.916622537938887: 1, 6.9154564728927195: 1, 6.913349151858342: 1, 6.903535710459677: 1,
6.903329804781966: 1, 6.900899553967346: 1, 6.898173416917645: 1, 6.897235382122138: 1, 6.893702141503202: 1, 6.89082
9965232869: 1, 6.884951468937643: 1, 6.884473224890981: 1, 6.876853521643247: 1, 6.866097884208758: 1, 6.848616202970
9075: 1, 6.848462002492946: 1, 6.837396305787209: 1, 6.832506528523592: 1, 6.831018904987701: 1, 6.825595665507493:
1, 6.819644186524865: 1, 6.81931664542049: 1, 6.813231691309165: 1, 6.806592586535522: 1, 6.789458264281253: 1, 6.788
663139335635: 1, 6.787918366204205: 1, 6.783444903861303: 1, 6.7830684892777775: 1, 6.780365917533769: 1, 6.772311312
396286: 1, 6.769907984675903: 1, 6.769110544200561: 1, 6.760698101616306: 1, 6.757784337588007: 1, 6.757573942207211:
1, 6.753545708012907: 1, 6.743815893328987: 1, 6.742938959791224: 1, 6.738590431358275: 1, 6.729098538304268: 1, 6.72
6137065402353: 1, 6.708154938083828: 1, 6.7054180885202666: 1, 6.704801891330013: 1, 6.703501276779939: 1, 6.70322428
27070325: 1, 6.700565040166586: 1, 6.696345359383066: 1, 6.696328993792122: 1, 6.688537773049988: 1, 6.68191540192238
4: 1, 6.678476723205921: 1, 6.670977975371603: 1, 6.669613041811153: 1, 6.668910464286518: 1, 6.661573369579118: 1,
6.658313092216683: 1, 6.657405915058727: 1, 6.656735848228627: 1, 6.638796347971882: 1, 6.634588946401105: 1, 6.62096
0554243777: 1, 6.617510063242299: 1, 6.611304678079388: 1, 6.58665083869649: 1, 6.585742362207105: 1, 6.5792984505833
13: 1, 6.577314025723322: 1, 6.576142162547389: 1, 6.574225176612528: 1, 6.569996584689828: 1, 6.568231315902981: 1,
6.564520808490623: 1, 6.5638294481509245: 1, 6.557684000441849: 1, 6.549617145822391: 1, 6.540010489065365: 1, 6.5253
6549793787: 1, 6.515480991973285: 1, 6.510935058926848: 1, 6.508350825264151: 1, 6.503006585729408: 1, 6.497574007307
379: 1, 6.4919197088218565: 1, 6.491454414301741: 1, 6.489574019764819: 1, 6.488177272216958: 1, 6.486329567964291:
1, 6.475126199070034: 1, 6.472681224390287: 1, 6.460724275078624: 1, 6.4584709219577325: 1, 6.457941120249733: 1, 6.4
5733401109377: 1, 6.4569181706774055: 1, 6.439982198744177: 1, 6.434503307106112: 1, 6.428433908093976: 1, 6.42587778

0282667: 1, 6.425656484663902: 1, 6.422731772787022: 1, 6.416177973271484: 1, 6.414293155126135: 1, 6.41353800442597
9: 1, 6.403687627245544: 1, 6.401015167688054: 1, 6.398817027307103: 1, 6.39855215972115: 1, 6.394817328066122: 1, 6.
379343028530031: 1, 6.377659752253703: 1, 6.371348451944884: 1, 6.368217684503827: 1, 6.366469352386251: 1, 6.3579288
14268639: 1, 6.3528750161569025: 1, 6.352600602984866: 1, 6.344580859167441: 1, 6.337169441927111: 1, 6.3338980788933
84: 1, 6.332791255327682: 1, 6.331123466620794: 1, 6.32515938275316: 1, 6.316540630848421: 1, 6.3156090413501875: 1,
6.314670108321912: 1, 6.313438609116804: 1, 6.31019058712178: 1, 6.308309552602153: 1, 6.3065824879684405: 1, 6.30434
067982865: 1, 6.300495781197276: 1, 6.294698723391203: 1, 6.290023507644476: 1, 6.288024916210727: 1, 6.2832253923122
98: 1, 6.274815898399573: 1, 6.27194434635641: 1, 6.2651081597989835: 1, 6.260415459834149: 1, 6.254875131618762: 1,
6.247939448884559: 1, 6.24755579838309: 1, 6.24460243979595: 1, 6.239261177545843: 1, 6.229197871431784: 1, 6.2211787
80587034: 1, 6.220216679773641: 1, 6.216117798253587: 1, 6.215615826888739: 1, 6.21300967241096: 1, 6.212183528330700
5: 1, 6.210872984461777: 1, 6.209219477838143: 1, 6.207853747102305: 1, 6.2063367354379: 1, 6.200046026443157: 1, 6.1
91357777373677: 1, 6.186249787971275: 1, 6.185320157661087: 1, 6.185317918007332: 1, 6.183106530329362: 1, 6.17744301
9219704: 1, 6.1769336755140385: 1, 6.176845921517219: 1, 6.159701385164087: 1, 6.155851647572191: 1, 6.15334515046739
1: 1, 6.149851380226797: 1, 6.149094821677437: 1, 6.137573451328701: 1, 6.136209549332921: 1, 6.119858879442652: 1,
6.113654392971037: 1, 6.0717257936543945: 1, 6.07044591011576: 1, 6.066689433291192: 1, 6.06344970477146: 1, 6.060729
789041354: 1, 6.060562757617054: 1, 6.056018519542322: 1, 6.05072557591515: 1, 6.050378877976073: 1, 6.04232892260192
5: 1, 6.034455923700227: 1, 6.03376826402244: 1, 6.027456074783749: 1, 6.021704383566648: 1, 6.015869920221733: 1, 6.
012302678930118: 1, 6.011054352651012: 1, 6.008500979220004: 1, 6.0049164643752135: 1, 5.998025285370867: 1, 5.995928
976176188: 1, 5.983431477561986: 1, 5.9833795724465135: 1, 5.980486654948563: 1, 5.969519058423198: 1, 5.967861181497
628: 1, 5.965450031389211: 1, 5.959252043205215: 1, 5.952611025250494: 1, 5.9474621232675045: 1, 5.947161961543962:
1, 5.9449028753052415: 1, 5.9377362640546405: 1, 5.936175850627677: 1, 5.935694792454576: 1, 5.935495147329775: 1, 5.
934345402506151: 1, 5.930542026851283: 1, 5.927125214164202: 1, 5.9267692010066675: 1, 5.915773937476189: 1, 5.914530
957704886: 1, 5.9102202765961485: 1, 5.907139273990187: 1, 5.907006191531718: 1, 5.905462756423442: 1, 5.900162235902
398: 1, 5.895285703164371: 1, 5.895167850911177: 1, 5.890281944294776: 1, 5.885212265347265: 1, 5.885045113941012: 1,
5.882099978574137: 1, 5.88148088662294: 1, 5.86422716915436: 1, 5.861462033621415: 1, 5.859060924592715: 1, 5.8586990
10989731: 1, 5.857424938106185: 1, 5.857406809304134: 1, 5.8556893275141295: 1, 5.852906981501353: 1, 5.8474058745572
16: 1, 5.843427427035808: 1, 5.8301718952356225: 1, 5.822108372632362: 1, 5.818870623293402: 1, 5.814097570053829: 1,
5.813028816433999: 1, 5.808075394072818: 1, 5.806566146283274: 1, 5.800910655566695: 1, 5.800524292781766: 1, 5.79084
499944927: 1, 5.783716856399386: 1, 5.776914471263938: 1, 5.771669863024761: 1, 5.7617180795639085: 1, 5.758922740290
45: 1, 5.755350555054604: 1, 5.747638207190218: 1, 5.747141759902156: 1, 5.745773014266575: 1, 5.744554914750363: 1,
5.741510244721255: 1, 5.740315695512481: 1, 5.734962177640591: 1, 5.7262240177481045: 1, 5.717238207230254: 1, 5.7155
09296780351: 1, 5.710940277068059: 1, 5.70670352439463: 1, 5.7065477335586845: 1, 5.705014693504247: 1, 5.70119523698
2985: 1, 5.6973603537286: 1, 5.696738981742135: 1, 5.696459636369966: 1, 5.686905710050173: 1, 5.6866025521116095: 1,
5.679988193556849: 1, 5.679582561952585: 1, 5.678672618328894: 1, 5.674649700948158: 1, 5.674054746415287: 1, 5.67094
5505911174: 1, 5.669542884467587: 1, 5.667799667175273: 1, 5.664923215511805: 1, 5.657073066395963: 1, 5.649934185978
003: 1, 5.641648818140331: 1, 5.640853436435719: 1, 5.640807148906594: 1, 5.638943522606612: 1, 5.638182729246735: 1,
5.636823722349117: 1, 5.62877843130586: 1, 5.627329905608375: 1, 5.626281579391398: 1, 5.624662079566293: 1, 5.617891
391838963: 1, 5.616305919510885: 1, 5.6034069924610765: 1, 5.600994435309625: 1, 5.58488523834058: 1, 5.5828001086302
43: 1, 5.578073772419765: 1, 5.573762146211482: 1, 5.573307401794613: 1, 5.57242527259311: 1, 5.5716627548520234: 1,
5.564604410534355: 1, 5.564526120351723: 1, 5.555344926005428: 1, 5.540452247980634: 1, 5.53422248221464: 1, 5.523105
740049692: 1, 5.5217193150800234: 1, 5.521289079158398: 1, 5.513959078998838: 1, 5.506283511311241: 1, 5.495787872370
217: 1, 5.491848140769782: 1, 5.483429487485388: 1, 5.481385602635064: 1, 5.481012758458531: 1, 5.4741264305166455:

1, 5.472258354520226: 1, 5.46976662537295: 1, 5.469169513443069: 1, 5.468646317161219: 1, 5.467799433426488: 1, 5.460
332856172096: 1, 5.456727621561489: 1, 5.449319934521713: 1, 5.446063573214205: 1, 5.442881440032189: 1, 5.4362987557
96414: 1, 5.435946503693791: 1, 5.4294400215604535: 1, 5.423960618116489: 1, 5.420766433673453: 1, 5.419697811602872:
1, 5.4184390585857685: 1, 5.416887476603736: 1, 5.409421950538672: 1, 5.406205344110293: 1, 5.4049065287781355: 1, 5.
404616607085841: 1, 5.404084120323672: 1, 5.403679785197234: 1, 5.402726494374618: 1, 5.392218795059599: 1, 5.3904304
67111775: 1, 5.389288164444441: 1, 5.3803583988590065: 1, 5.379532335650242: 1, 5.376609961199742: 1, 5.3729898659799
415: 1, 5.370942781963471: 1, 5.37081508768818: 1, 5.3675982256514745: 1, 5.357774852721819: 1, 5.356076136791217: 1,
5.355760448652253: 1, 5.353549685231477: 1, 5.346973008325695: 1, 5.346150082653942: 1, 5.34416427999501: 1, 5.338226
728634832: 1, 5.329177154346276: 1, 5.323928943695857: 1, 5.3237165012402174: 1, 5.32370000226949: 1, 5.3174912595016
78: 1, 5.316425306075376: 1, 5.315638959723731: 1, 5.314245269569142: 1, 5.305944536930434: 1, 5.303616414204295: 1,
5.2973528564264205: 1, 5.2931423359931316: 1, 5.286677803867469: 1, 5.273058917373477: 1, 5.2715962188567795: 1, 5.26
9455680494848: 1, 5.2615075366188675: 1, 5.249153207410081: 1, 5.2484121709076454: 1, 5.244670454498079: 1, 5.2431189
35864228: 1, 5.231886415284325: 1, 5.231202204471623: 1, 5.230729552297828: 1, 5.226454536794267: 1, 5.22342887323070
3: 1, 5.223281035391141: 1, 5.222332801123112: 1, 5.219842734714686: 1, 5.219811904858967: 1, 5.216319113220327: 1,
5.213130914791641: 1, 5.2102486604312315: 1, 5.208353266611422: 1, 5.207819737703572: 1, 5.199995584608915: 1, 5.1999
74931004034: 1, 5.199465773003341: 1, 5.195598922347856: 1, 5.192989650320792: 1, 5.1927042139639985: 1, 5.1883375109
477265: 1, 5.180733438368005: 1, 5.177373511069053: 1, 5.175392837162258: 1, 5.160800307656636: 1, 5.147842539904723:
1, 5.147170794318572: 1, 5.146367229698908: 1, 5.143371298685507: 1, 5.140481559759953: 1, 5.1380375956392665: 1, 5.1
368561723214565: 1, 5.135775497006212: 1, 5.13379180789576: 1, 5.1178047187086895: 1, 5.112358463427596: 1, 5.1105430
54320008: 1, 5.109766449456286: 1, 5.097710232735255: 1, 5.091841194196406: 1, 5.091305648981569: 1, 5.09091106253637
45: 1, 5.09030471363295: 1, 5.0876722835139425: 1, 5.086159429890548: 1, 5.081470731993154: 1, 5.07914368846196: 1,
5.076229248647061: 1, 5.073884338578104: 1, 5.073684759642195: 1, 5.072809371575564: 1, 5.072683072904992: 1, 5.06934
6062545997: 1, 5.06845447003838: 1, 5.067386164536373: 1, 5.06677573308471: 1, 5.0658543948184045: 1, 5.0582471729964
06: 1, 5.057884512800933: 1, 5.054498384175971: 1, 5.054214502167192: 1, 5.051943235730025: 1, 5.048003712719248: 1,
5.0479368635996265: 1, 5.047706920578542: 1, 5.0450264778428515: 1, 5.043179303608816: 1, 5.042785361132565: 1, 5.042
362460830785: 1, 5.041282599503271: 1, 5.040579944588767: 1, 5.040150421396626: 1, 5.038167823152552: 1, 5.0362294780
29731: 1, 5.036078204086145: 1, 5.035627534025664: 1, 5.035280303569663: 1, 5.030939864525626: 1, 5.0303422284016435:
1, 5.030060511168855: 1, 5.028639060990557: 1, 5.028297857413403: 1, 5.022891305497391: 1, 5.018149627847771: 1, 5.01
4952673717038: 1, 5.010933526978713: 1, 5.0102997099707505: 1, 5.0102349152812735: 1, 5.010073845815599: 1, 5.0053595
016407835: 1, 5.004147008910419: 1, 5.003840585017802: 1, 5.000226049921742: 1, 4.99511253595058: 1, 4.98875946065161
8: 1, 4.98700019488442: 1, 4.975786618775082: 1, 4.974222337046277: 1, 4.96986504454896: 1, 4.968021742504003: 1, 4.9
61956194948886: 1, 4.961410799376447: 1, 4.9480438167190615: 1, 4.9404812472862405: 1, 4.937507032075209: 1, 4.936837
802666634: 1, 4.935569917767215: 1, 4.917957099115607: 1, 4.916235563292207: 1, 4.913666569251573: 1, 4.9127406471752
56: 1, 4.909238529422977: 1, 4.907426070740436: 1, 4.902984802835384: 1, 4.900776998845662: 1, 4.89974564850514: 1,
4.8928575929213265: 1, 4.891208154270839: 1, 4.8898026651251465: 1, 4.888905700647589: 1, 4.888370827856373: 1, 4.880
571450327621: 1, 4.871742303044715: 1, 4.870415774600772: 1, 4.868686102597515: 1, 4.867355895681423: 1, 4.8623762197
47749: 1, 4.861061074617565: 1, 4.856826081409582: 1, 4.856028883370021: 1, 4.853712879264069: 1, 4.852845109828953:
1, 4.850700653552605: 1, 4.849390426325272: 1, 4.848521563248774: 1, 4.841012113601029: 1, 4.840600171904417: 1, 4.83
8191775653725: 1, 4.834450390149484: 1, 4.82937322538547: 1, 4.8056536997208115: 1, 4.805118265289741: 1, 4.800435200
724018: 1, 4.7912393201464445: 1, 4.780936193061303: 1, 4.777083667721549: 1, 4.774222105192065: 1, 4.76946570331864
2: 1, 4.76798521346323: 1, 4.767119382040827: 1, 4.763426798990837: 1, 4.7633413941985845: 1, 4.760620109010504: 1,
4.758924489383673: 1, 4.754712786890196: 1, 4.751174227566311: 1, 4.750738207530829: 1, 4.748221991670108: 1, 4.74760

5455063163: 1, 4.746953882948743: 1, 4.745280260372787: 1, 4.744011341249297: 1, 4.7423654365384245: 1, 4.74166797862
0501: 1, 4.740732772252957: 1, 4.739078898308588: 1, 4.7381881011921605: 1, 4.733240692345087: 1, 4.727944958982979:
1, 4.7271736740705075: 1, 4.727169818812276: 1, 4.7250521348213255: 1, 4.724929122779139: 1, 4.721350118526363: 1, 4.
720518507169271: 1, 4.720034987791293: 1, 4.718852928246673: 1, 4.718801187538428: 1, 4.716149190480846: 1, 4.7136872
19729396: 1, 4.712402354208977: 1, 4.70769064022001: 1, 4.706729102779123: 1, 4.698821521600712: 1, 4.69643926592603
6: 1, 4.693283419106785: 1, 4.691918478384293: 1, 4.689959620351378: 1, 4.68833794223851: 1, 4.683620623334618: 1, 4.
683207989319338: 1, 4.675234192474559: 1, 4.671317012372601: 1, 4.668290262538031: 1, 4.663628499676044: 1, 4.6618907
85427372: 1, 4.6609649425964: 1, 4.656716386497832: 1, 4.656454434523723: 1, 4.656072492704421: 1, 4.655835781501933:
1, 4.6554849433171315: 1, 4.6535727342269: 1, 4.653106875467424: 1, 4.65136699853691: 1, 4.6512878654138055: 1, 4.650
711759335154: 1, 4.646762432432245: 1, 4.64614832260395: 1, 4.6420489304124 48: 1, 4.63651658557801: 1, 4.635470093632
976: 1, 4.63525560496821: 1, 4.631700837185394: 1, 4.626461047526169: 1, 4.6154525218313935: 1, 4.615389687982638: 1,
4.613409266652996: 1, 4.612444393447893: 1, 4.605667755025811: 1, 4.603096093938328: 1, 4.601658173575813: 1, 4.60070
146616054: 1, 4.582277490847804: 1, 4.581038126278604: 1, 4.579980856542601: 1, 4.579010516172184: 1, 4.5704565793007
73: 1, 4.566915597043502: 1, 4.554706641381156: 1, 4.55121161066759: 1, 4.540983854275984: 1, 4.5390383362862305: 1,
4.532501307815416: 1, 4.531165800057887: 1, 4.53028685592169: 1, 4.525388419629061: 1, 4.524235118835014: 1, 4.524023
487458356: 1, 4.520576450936406: 1, 4.51994069655437: 1, 4.518447832780786: 1, 4.516218151706811: 1, 4.49780455987221
8: 1, 4.493480029113384: 1, 4.490502608594201: 1, 4.488485760644489: 1, 4.486504428895354: 1, 4.483571498755627: 1,
4.4727584242871705: 1, 4.47017850010271: 1, 4.470068249972104: 1, 4.466997960538578: 1, 4.466408865772714: 1, 4.46429
28448819434: 1, 4.461689070977189: 1, 4.459581952563905: 1, 4.458938125264788: 1, 4.4566348633249975: 1, 4.4547670122
12055: 1, 4.443446612843721: 1, 4.436862014009203: 1, 4.427724522227287: 1, 4.427237963969843: 1, 4.42657156765769:
1, 4.42250820534321: 1, 4.420567449205459: 1, 4.411430304205425: 1, 4.409822617013915: 1, 4.404315935039339: 1, 4.400
354389510793: 1, 4.399389031136292: 1, 4.397342952029495: 1, 4.396081422149422: 1, 4.394211947370373: 1, 4.3927004568
84545: 1, 4.387314155085476: 1, 4.386437839804702: 1, 4.3821933116921326: 1, 4.378439052227279: 1, 4.377922416622814:
1, 4.376579779389059: 1, 4.376440803137253: 1, 4.3731219178096365: 1, 4.369328341853214: 1, 4.368563977456611: 1, 4.3
64850285513815: 1, 4.361413570406701: 1, 4.358467283825471: 1, 4.3582801360583945: 1, 4.354802749322948: 1, 4.3454197
2192753: 1, 4.344571556611702: 1, 4.343003060545807: 1, 4.342628756062477: 1, 4.333404530716712: 1, 4.31944938451522
7: 1, 4.317609851949443: 1, 4.316695028595643: 1, 4.3150749918015565: 1, 4.313807007759333: 1, 4.309384979159088: 1,
4.309099942129516: 1, 4.306028734426785: 1, 4.3057514856423005: 1, 4.304255935259336: 1, 4.303680265194701: 1, 4.3022
0914793237: 1, 4.297755025658394: 1, 4.292796517246215: 1, 4.292360148438682: 1, 4.288090042462147: 1, 4.286449541012
053: 1, 4.286175415658613: 1, 4.284552338277161: 1, 4.27555831417394: 1, 4.269743148052134: 1, 4.2659023929169795: 1,
4.265181008518996: 1, 4.264336111407003: 1, 4.257640379632892: 1, 4.25252678086092: 1, 4.2496778224122735: 1, 4.24598
3884532065: 1, 4.244710661529569: 1, 4.2426891671099165: 1, 4.242265218387242: 1, 4.238994553186641: 1, 4.23827013002
819: 1, 4.234534476954691: 1, 4.234423696988997: 1, 4.233502504829359: 1, 4.2270196894540435: 1, 4.226183914045679:
1, 4.224642147555326: 1, 4.2218113682885114: 1, 4.221371070879928: 1, 4.2160519290374365: 1, 4.21520434708455: 1, 4.2
14645703391472: 1, 4.2140736004805674: 1, 4.212598854101765: 1, 4.2102108262963025: 1, 4.2097098548650385: 1, 4.20925
1657018884: 1, 4.209148136819567: 1, 4.205015465053285: 1, 4.197225695959905: 1, 4.195661706387897: 1, 4.194843645926
747: 1, 4.187871058325643: 1, 4.186492610049276: 1, 4.186196372489022: 1, 4.185886067164947: 1, 4.182737336659107: 1,
4.182728115177218: 1, 4.1785535855814295: 1, 4.17749865132584: 1, 4.174447417909692: 1, 4.170299211123154: 1, 4.16820
158742985: 1, 4.164565325286522: 1, 4.163696779852769: 1, 4.158682303725826: 1, 4.157883471746877: 1, 4.1576975760633
7: 1, 4.154207288953335: 1, 4.153191203798238: 1, 4.14925341304927: 1, 4.14781039874265: 1, 4.144255885702986: 1, 4.1
417488521391235: 1, 4.1411527477941705: 1, 4.136861992088778: 1, 4.125712451511644: 1, 4.120359572796367: 1, 4.120199
1663627116: 1, 4.119069471931545: 1, 4.117532121840901: 1, 4.116172312265524: 1, 4.111223941486301: 1, 4.107785513968

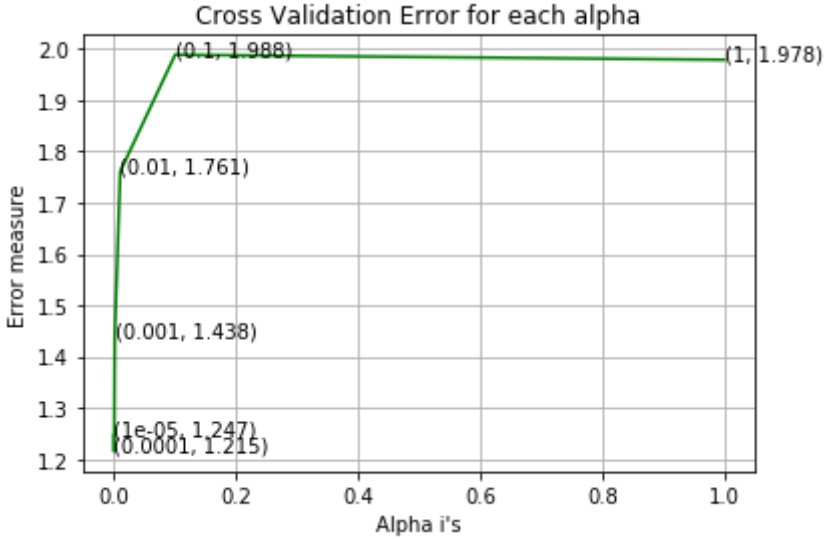
746: 1, 4.105624503768519: 1, 4.104780920184989: 1, 4.104678483080071: 1, 4.102736161451063: 1, 4.102179678307983: 1, 4.076400782852502: 1, 4.070446175151217: 1, 4.065830934401661: 1, 4.059736910063325: 1, 4.059490020805868: 1, 4.059102570049180 5: 1, 4.0589906836927705: 1, 4.055759821152228: 1, 4.053696180357644: 1, 4.04762579274984: 1, 4.044316258861213: 1, 4.044215202765014: 1, 4.04108442706049: 1, 4.0386939265991355: 1, 4.031736739554946: 1, 4.029399889438986: 1, 4.02909421019828: 1, 4.028066476813946: 1, 4.024461847693416: 1, 4.021227263224112: 1, 4.017348307761275: 1, 4.013191086806148: 1, 4.0107959163074405: 1, 4.004353896092293: 1, 4.001874680591065: 1, 3.996719646825992: 1, 3.9931779599090462: 1, 3.9840957683699645: 1, 3.9834653995172835: 1, 3.9793196971686364: 1, 3.9752331082175076: 1, 3.9711813656293016: 1, 3.96951169044261: 1, 3.9652600926689945: 1, 3.9615287976432425: 1, 3.9505387480442447: 1, 3.9496069490601418: 1, 3.9341936744934087: 1, 3.932014921913127: 1, 3.9248078332445826: 1, 3.923884894688853: 1, 3.9227210571460773: 1, 3.922517107004228: 1, 3.9143023331207356: 1, 3.8982528613502567: 1, 3.893816190843095: 1, 3.8920806520475724: 1, 3.888188384676964: 1, 3.887479090898293: 1, 3.8859714679500303: 1, 3.8842134604185232: 1, 3.8747242372912214: 1, 3.8665783625417776: 1, 3.8655556085508374: 1, 3.865372136338904: 1, 3.8644089608076135: 1, 3.863634333444394: 1, 3.862747365944774: 1, 3.8617741662797895: 1, 3.859618347006292: 1, 3.8517984720976797: 1, 3.8503297868248705: 1, 3.8489140035618723: 1, 3.8440154532942215: 1, 3.8384067543433287: 1, 3.8374581153076504: 1, 3.8341387466017354: 1, 3.8323657367252735: 1, 3.8322759140926363: 1, 3.8304342714039072: 1, 3.8250792571769536: 1, 3.8243261890522957: 1, 3.823924519232019: 1, 3.820785800192157: 1, 3.8178568180175336: 1, 3.8084666458564818: 1, 3.807428410951876: 1, 3.8034810338581444: 1, 3.800861245216206: 1, 3.7931573768527924: 1, 3.790357568881886: 1, 3.789263130654843: 1, 3.7887412680793195: 1, 3.787959590696835: 1, 3.7860486174916645: 1, 3.7824154789461923: 1, 3.7824025388854907: 1, 3.7822284799617316: 1, 3.780582492929493: 1, 3.776998577592272: 1, 3.768346428897963: 1, 3.7671807089367455: 1, 3.7653495581835363: 1, 3.764931661927105: 1, 3.762284629431026: 1, 3.759283004783983: 1, 3.7500131678812623: 1, 3.7473310889578464: 1, 3.745172884076121: 1, 3.7417932182798483: 1, 3.7379176292477703: 1, 3.7373535368406094: 1, 3.7360053484501505: 1, 3.727320920962759: 1, 3.726109464460388: 1, 3.725800855424467: 1, 3.7147132135736896: 1, 3.7137491426410993: 1, 3.71191683847105: 1, 3.709699909354859: 1, 3.709386424360302: 1, 3.708079285853423: 1, 3.6970802799882945: 1, 3.696210543184579: 1, 3.695621141977781: 1, 3.6923307747987013: 1, 3.6843680356151256: 1, 3.6813577513908227: 1, 3.6786821156272937: 1, 3.678157438983562: 1, 3.676867140554149: 1, 3.672324864048702: 1, 3.6588880300001865: 1, 3.655644409283966: 1, 3.6536485699975363: 1, 3.645048570986411: 1, 3.6342230302027914: 1, 3.6328741381797793: 1, 3.6321264611744697: 1, 3.629497025506761: 1, 3.6225032380720465: 1, 3.6188024258650913: 1, 3.616926060588833: 1, 3.6148843817070206: 1, 3.6081582195900475: 1, 3.6042468257378784: 1, 3.593125173609 1006: 1, 3.590528393719344: 1, 3.5864703430068823: 1, 3.5829569325388153: 1, 3.5794142839615355: 1, 3.5753947489810507: 1, 3.5752323656177496: 1, 3.570051883427126: 1, 3.56437671175183: 1, 3.563493636646569: 1, 3.562477536872245: 1, 3.5586687986411913: 1, 3.5583924522194006: 1, 3.5526295690244645: 1, 3.534495239453429: 1, 3.5336925244787682: 1, 3.5302742987503195: 1, 3.520257806991459: 1, 3.519836616160662: 1, 3.5120424141012276: 1, 3.511883806215822: 1, 3.510574152085562: 1, 3.5085669092734637: 1, 3.504129072344607: 1, 3.5015868010295645: 1, 3.501369229804301: 1, 3.4874256703809143: 1, 3.486211090449389: 1, 3.4848929018070094: 1, 3.4796583426699006: 1, 3.4736899980028237: 1, 3.4720354301111196: 1, 3.451615942832394: 1, 3.4477195971480663: 1, 3.43944084314827: 1, 3.435704856442015: 1, 3.4220385644479334: 1, 3.4199027124722026: 1, 3.4062741583274905: 1, 3.405899411672747: 1, 3.395001831152265: 1, 3.392682418926099: 1, 3.389151720539676: 1, 3.387968106473728: 1, 3.3854317175504387: 1, 3.383978502040268: 1, 3.383226357988007: 1, 3.38215072066649: 1, 3.366005685020408: 1, 3.3592028403086487: 1, 3.358645778896409: 1, 3.3570042277521854: 1, 3.3526881911427884: 1, 3.347950927641227: 1, 3.3292199661706334: 1, 3.3214342336907343: 1, 3.3201793744475063: 1, 3.298443515494985: 1, 3.2974354859758304: 1, 3.2908244981154042: 1, 3.278189880826381: 1, 3.27479358353746: 1, 3.2742469379030292: 1, 3.270491260255584: 1, 3.269951784969656: 1, 3.2674068541548693: 1, 3.26541491639953: 1, 3.2617388360412303: 1, 3.2433902363649247: 1, 3.241263615851571: 1, 3.2405163482328097: 1, 3.235427356137531: 1, 3.216877888463073: 1, 3.211115364069015: 1, 3.1933024778752572: 1, 3.1783043872566634: 1, 3.165941164707314: 1, 3.142

```
2925766702425: 1, 3.1405622982101993: 1, 3.133323712979683: 1, 3.1138329653340575: 1, 3.0898249558513373: 1, 3.088974
588545892: 1, 3.0755440471911704: 1, 3.039692972867657: 1, 3.02120945938044: 1, 3.0124523669885868: 1, 2.961710279871
1232: 1, 2.901222599832102: 1, 2.900070591304457: 1, 2.885951170385234: 1, 2.8129760785470572: 1, 2.79185913547278:
1, 2.3275233508278275: 1, 2.196538903293827: 1})
```

In [54]:
```
# Train a Logistic regression+Calibration model using text features whicha re on-hot encoded
alpha = [10 ** x for x in range(-5, 1)]

# read more about SGDClassifier() at http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html
# ------------------------------
# default parameters
# SGDClassifier(loss='hinge', penalty='l2', alpha=0.0001, l1_ratio=0.15, fit_intercept=True, max_iter=None, tol=None,
# shuffle=True, verbose=0, epsilon=0.1, n_jobs=1, random_state=None, learning_rate='optimal', eta0=0.0, power_t=0.5,
# class_weight=None, warm_start=False, average=False, n_iter=None)

# some of methods
# fit(X, y[, coef_init, intercept_init, ...])     Fit linear model with Stochastic Gradient Descent.
# predict(X)    Predict class labels for samples in X.

#------------------------------
# video link:
#------------------------------


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

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

predict_y = sig_clf.predict_proba(train_text_feature_onehotCoding)
print('For values of best alpha = ', alpha[best_alpha], "The train log loss is:",log_loss(y_train, predict_y, labels=c
lf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(cv_text_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_text_feature_onehotCoding)
print('For values of best alpha = ', alpha[best_alpha], "The test log loss is:",log_loss(y_test, predict_y, labels=clf
.classes_, eps=1e-15))
```

```
For values of alpha =  1e-05 The log loss is: 1.2473217402069148
For values of alpha =  0.0001 The log loss is: 1.2150164429100236
For values of alpha =  0.001 The log loss is: 1.4378365557150556
For values of alpha =  0.01 The log loss is: 1.7606492005525152
For values of alpha =  0.1 The log loss is: 1.9882939583239234
For values of alpha =  1 The log loss is: 1.978436136642287
```




```
For values of best alpha =  0.0001 The train log loss is: 0.7620156539138682
For values of best alpha =  0.0001 The cross validation log loss is: 1.2150164429100236
For values of best alpha =  0.0001 The test log loss is: 1.0541686735272682
```

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

Ans. Yes, it seems like!

In [55]:
```
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 [56]:
```
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")
```

```
6.89 % of word of test data appeared in train data
7.571 % of word of Cross Validation appeared in train data
```

4. Machine Learning Models

In [33]:
```
#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 belongs 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 [60]:
```
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 [38]:
```
# 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 = CountVectorizer(min_df=3,ngram_range=(1,2))

    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 [{}]".format(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 [{}]".format(word,yes_no))

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

In [29]:
```
# STACKING TECHNIQUE


# 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_onehotCoding = hstack((train_gene_feature_onehotCoding,
                                      train_variation_feature_onehotCoding))

test_gene_var_onehotCoding = hstack((test_gene_feature_onehotCoding,
                                     test_variation_feature_onehotCoding))

cv_gene_var_onehotCoding = hstack((cv_gene_feature_onehotCoding,
                                   cv_variation_feature_onehotCoding))


train_x_onehotCoding = hstack((train_gene_var_onehotCoding,
                               train_text_feature_onehotCoding)).tocsr()

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

test_x_onehotCoding = hstack((test_gene_var_onehotCoding,
                              test_text_feature_onehotCoding)).tocsr()

test_y = np.array(list(test_df['Class']))

cv_x_onehotCoding = hstack((cv_gene_var_onehotCoding,
                            cv_text_feature_onehotCoding)).tocsr()

cv_y = np.array(list(cv_df['Class']))


train_gene_var_responseCoding = np.hstack((train_gene_feature_responseCoding,
                                           train_variation_feature_responseCoding))
```

```
test_gene_var_responseCoding = np.hstack((test_gene_feature_responseCoding,
                                          test_variation_feature_responseCoding))

cv_gene_var_responseCoding = np.hstack((cv_gene_feature_responseCoding,
                                        cv_variation_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_feature_responseCoding))

cv_x_responseCoding = np.hstack((cv_gene_var_responseCoding,
                                 cv_text_feature_responseCoding))
```

In [64]:
```
print("One hot encoding features :")
print("(number of data points * number of features) in train data = ", train_x_onehotCoding.shape)
print("(number of data points * number of features) in test data = ", test_x_onehotCoding.shape)
print("(number of data points * number of features) in cross validation data =", cv_x_onehotCoding.shape)
```

```
One hot encoding features :
(number of data points * number of features) in train data =  (2124, 4183)
(number of data points * number of features) in test data =  (665, 4183)
(number of data points * number of features) in cross validation data = (532, 4183)
```

In [65]:
```
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_responseCoding.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 = (532, 27)
```

Feature engineering on one hot encoded features

In [30]:
```
train_x_onehotCodingFE=np.sqrt(train_x_onehotCoding)
test_x_onehotCodingFE=np.sqrt(test_x_onehotCoding)
cv_x_onehotCodingFE=np.sqrt(cv_x_onehotCoding)
```

In [67]:
```
print("One hot encoding of features engineered features:")
print("(number of data points * number of features) in train data = ", train_x_onehotCodingFE.shape)
print("(number of data points * number of features) in test data = ", test_x_onehotCodingFE.shape)
print("(number of data points * number of features) in cross validation data =", cv_x_onehotCodingFE.shape)
```

```
One hot encoding of features engineered features:
(number of data points * number of features) in train data =  (2124, 4183)
(number of data points * number of features) in test data =  (665, 4183)
(number of data points * number of features) in cross validation data = (532, 4183)
```

In [68]:
```
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_responseCoding.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 = (532, 27)
```

4.1. Base Line Model Naive base

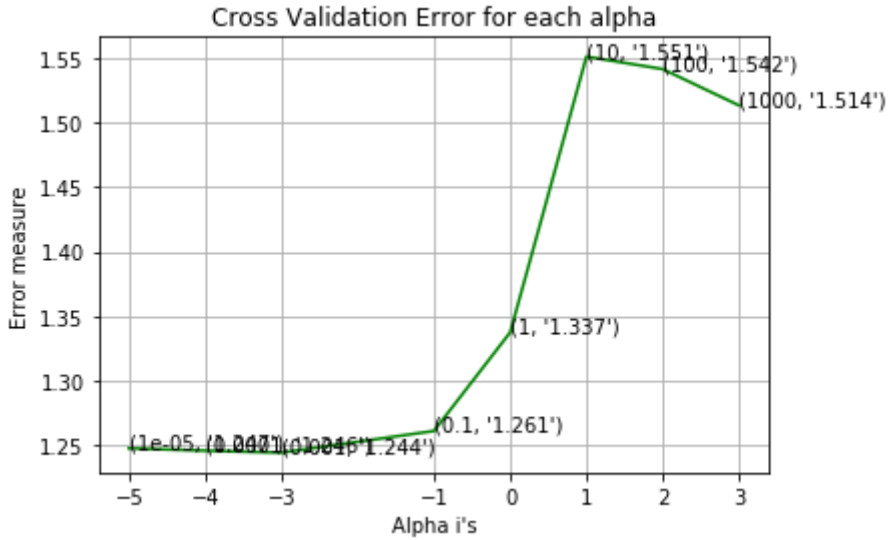
4.1.1.1. Hyper parameter tuning

In [87]:
```
# find more about Multinomial Naive base function here http://scikit-learn.org/stable/modules/generated/sklearn.naive_
bayes.MultinomialNB.html
# -------------------------
# default paramters
# sklearn.naive_bayes.MultinomialNB(alpha=1.0, fit_prior=True, class_prior=None)

# some of methods of MultinomialNB()
# fit(X, y[, sample_weight])    Fit Naive Bayes classifier according to X, y
# predict(X)    Perform classification on an array of test vectors X.
# predict_log_proba(X)  Return log-probability estimates for the test vector X.
# -----------------------
# video link: https://www.appliedaicourse.com/course/applied-ai-course-online/lessons/naive-bayes-algorithm-1/
# -----------------------


# find more about CalibratedClassifierCV here at http://scikit-learn.org/stable/modules/generated/sklearn.calibration.
CalibratedClassifierCV.html
# ----------------------------
# default paramters
# sklearn.calibration.CalibratedClassifierCV(base_estimator=None, method='sigmoid', cv=3)
#
# some of the methods of CalibratedClassifierCV()
# fit(X, y[, sample_weight])    Fit the calibrated model
# get_params([deep])    Get parameters for this estimator.
# predict(X)    Predict the target of new samples.
# predict_proba(X)      Posterior probabilities of classification
# ----------------------------
# video link: https://www.appliedaicourse.com/course/applied-ai-course-online/lessons/naive-bayes-algorithm-1/
# -----------------------


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_onehotCoding, train_y)
    sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
    sig_clf.fit(train_x_onehotCoding, train_y)
    sig_clf_probs = sig_clf.predict_proba(cv_x_onehotCoding)
    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-probability estimates
    print("for alpha = {} -----> Log Loss {}". format(i,log_loss(cv_y, sig_clf_probs)))
```

```
for alpha = 1e-05 -----> Log Loss 1.247491669245583
for alpha = 0.0001 -----> Log Loss 1.245679898704893
for alpha = 0.001 -----> Log Loss 1.2441355187830807
for alpha = 0.1 -----> Log Loss 1.260936037804692
for alpha = 1 -----> Log Loss 1.3374229086968037
for alpha = 10 -----> Log Loss 1.5513543842990123
for alpha = 100 -----> Log Loss 1.5416812684522725
for alpha = 1000 -----> Log Loss 1.513511114653115
```

In [88]:
```
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()
```



In [89]:
```
best_alpha = np.argmin(cv_log_error_array)
clf = MultinomialNB(alpha=alpha[best_alpha])
clf.fit(train_x_onehotCoding, train_y)
sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
sig_clf.fit(train_x_onehotCoding, train_y)
```

Out[89]:
```
CalibratedClassifierCV(base_estimator=MultinomialNB(alpha=0.001, class_prior=None, fit_prior=True),
            cv='warn', method='sigmoid')
```

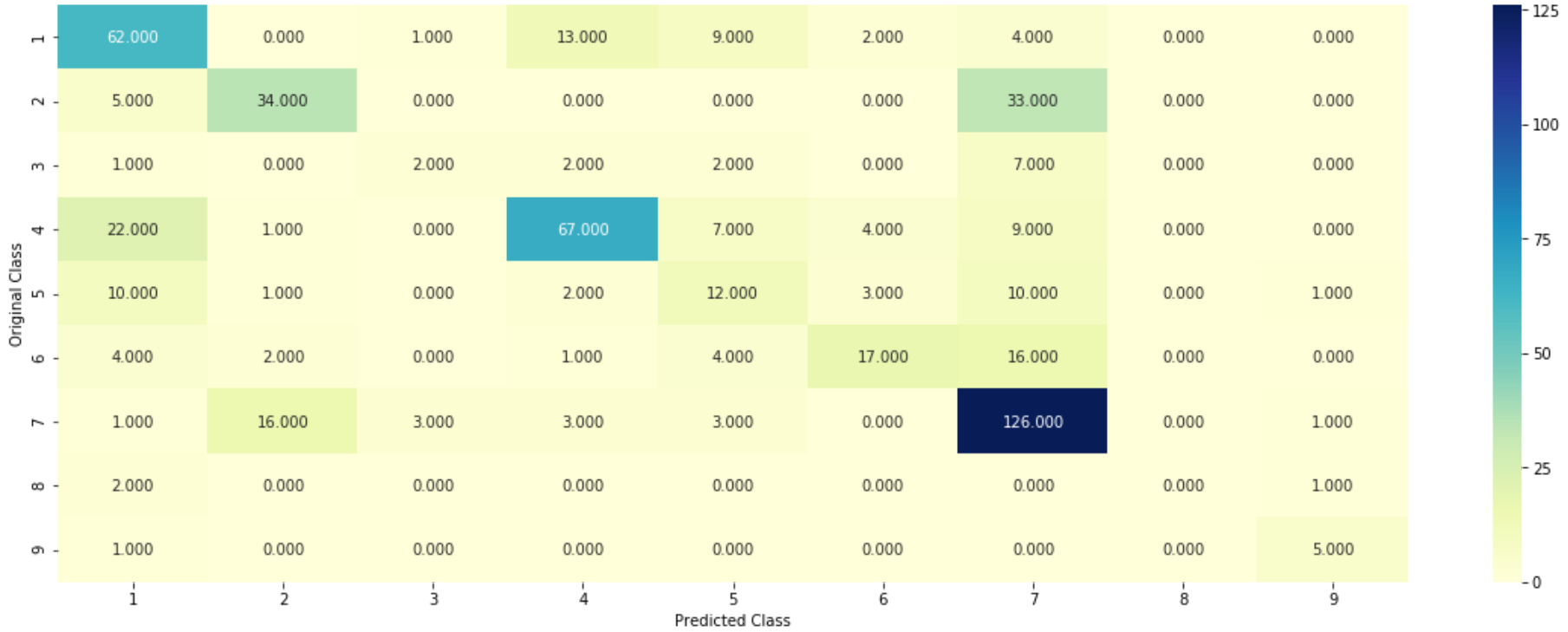
In [90]:
```
predict_y = sig_clf.predict_proba(train_x_onehotCoding)
print('For values of best alpha = ', alpha[best_alpha], "The train log loss is:",log_loss(y_train, predict_y, labels=c
lf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(cv_x_onehotCoding)
print('For values of best alpha = ', 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_x_onehotCoding)
print('For values of best alpha = ', alpha[best_alpha], "The test log loss is:",log_loss(y_test, predict_y, labels=clf
.classes_, eps=1e-15))
```

```
For values of best alpha =  0.001 The train log loss is: 0.5635743875020112
For values of best alpha =  0.001 The cross validation log loss is: 1.2441355187830807
For values of best alpha =  0.001 The test log loss is: 1.169423133287461
```

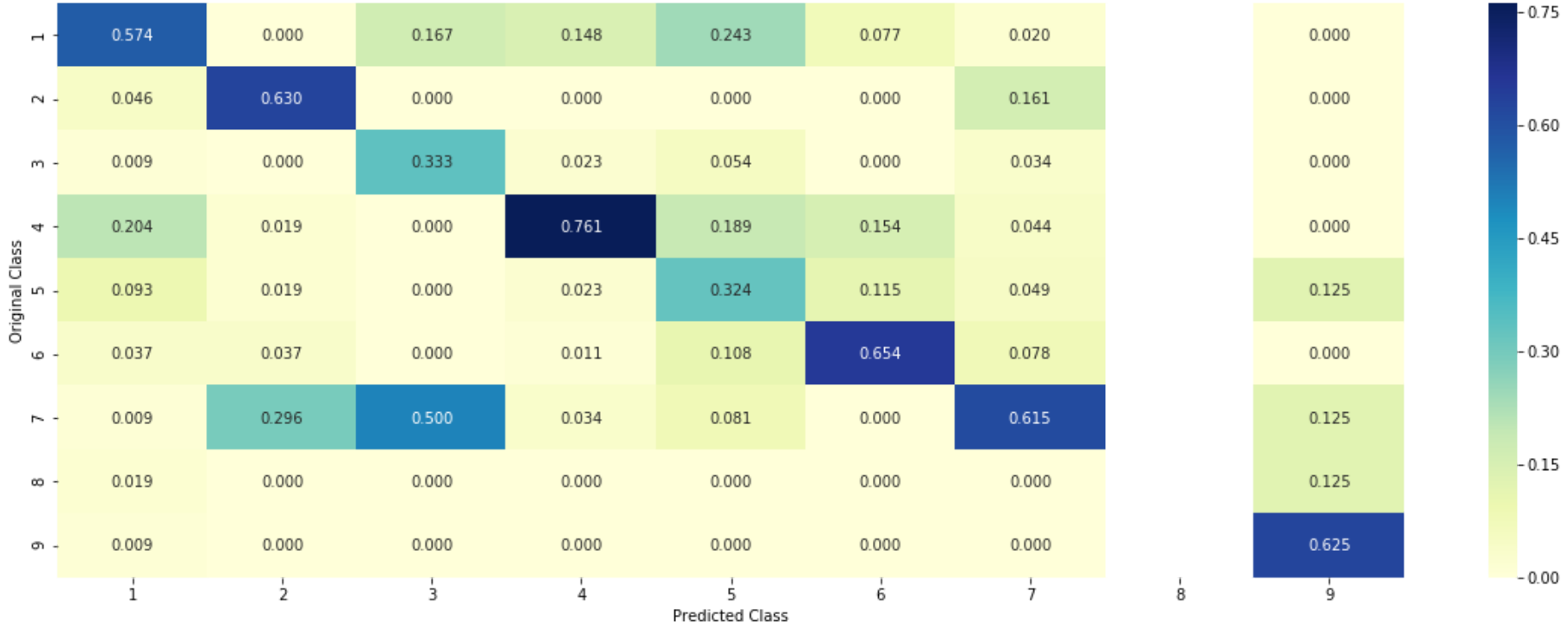
4.1.1.2. Testing the model with best hyper paramters

In [91]:
```
# find more about Multinomial Naive base function here http://scikit-learn.org/stable/modules/generated/sklearn.naive_
bayes.MultinomialNB.html
# -------------------------
# default paramters
# sklearn.naive_bayes.MultinomialNB(alpha=1.0, fit_prior=True, class_prior=None)

# some of methods of MultinomialNB()
# fit(X, y[, sample_weight])    Fit Naive Bayes classifier according to X, y
# predict(X)    Perform classification on an array of test vectors X.
# predict_log_proba(X)  Return log-probability estimates for the test vector X.
# -----------------------
# video link: https://www.appliedaicourse.com/course/applied-ai-course-online/lessons/naive-bayes-algorithm-1/
# -----------------------


# find more about CalibratedClassifierCV here at http://scikit-learn.org/stable/modules/generated/sklearn.calibration.
CalibratedClassifierCV.html
# ----------------------------
# default paramters
# sklearn.calibration.CalibratedClassifierCV(base_estimator=None, method='sigmoid', cv=3)
#
# some of the methods of CalibratedClassifierCV()
# fit(X, y[, sample_weight])    Fit the calibrated model
# get_params([deep])    Get parameters for this estimator.
# predict(X)    Predict the target of new samples.
# predict_proba(X)      Posterior probabilities of classification
# ----------------------------

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

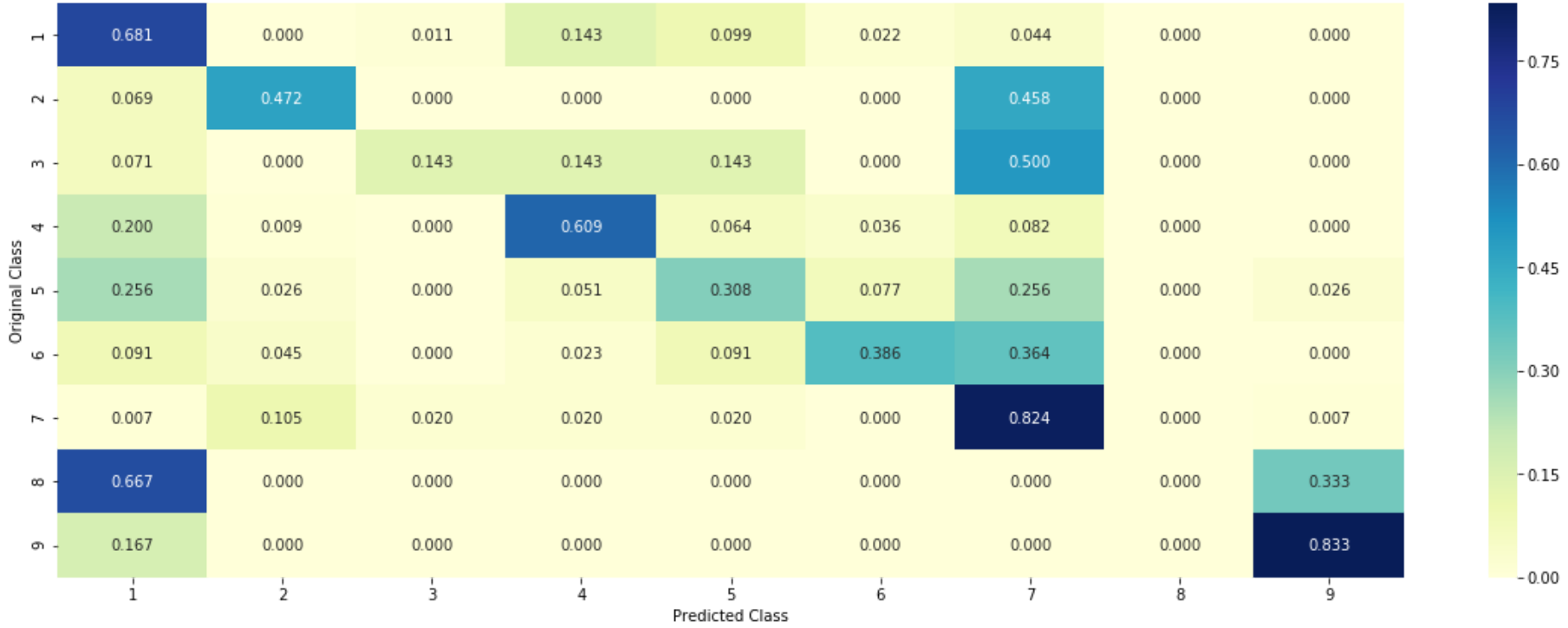
```
Log Loss : 1.2441355187830807
Number of missclassified point : 0.3890977443609023
-------------------- Confusion matrix --------------------
```




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



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



4.1.1.3. Feature Importance, Correctly classified point

In [92]:
```
test_point_index = 1
no_feature = 100
predicted_cls = sig_clf.predict(test_x_onehotCoding[test_point_index])
print("Predicted Class :", predicted_cls[0])
print("Predicted Class Probabilities:", np.round(sig_clf.predict_proba(test_x_onehotCoding[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: [[0.0548 0.0461 0.012  0.7337 0.035  0.0326 0.079  0.0047 0.0019]]
Actual Class : 7
--------------------------------------------------
Out of the top  100  features  0 are present in query point
```

4.1.1.4. Feature Importance, Incorrectly classified point

In [93]:
```
test_point_index = 100
no_feature = 100
predicted_cls = sig_clf.predict(test_x_onehotCoding[test_point_index])
print("Predicted Class :", predicted_cls[0])
print("Predicted Class Probabilities:", np.round(sig_clf.predict_proba(test_x_onehotCoding[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 : 7
Predicted Class Probabilities: [[0.0572 0.0654 0.0128 0.0667 0.0374 0.0346 0.7188 0.005  0.002 ]]
Actual Class : 7
--------------------------------------------------
Out of the top  100  features  0 are present in query point
```

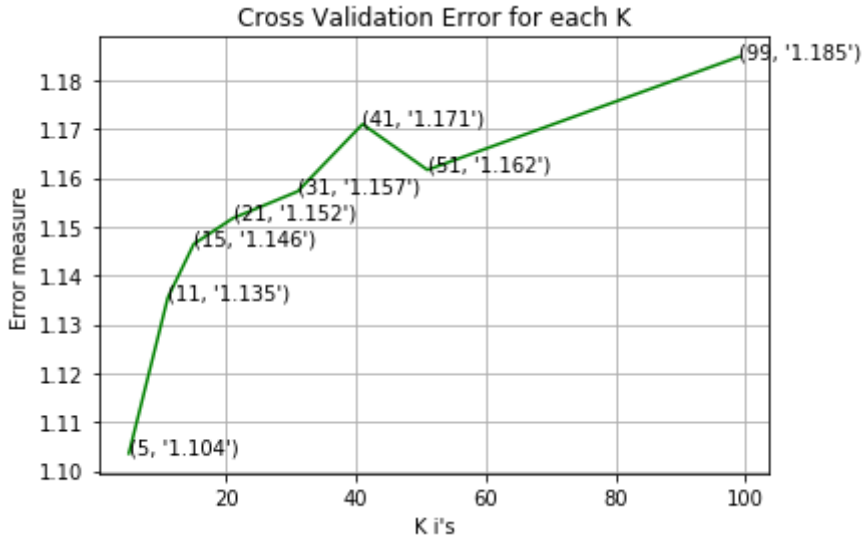
4.2. K Nearest Neighbour Classification

4.2.1. Hyper parameter tuning

In [95]:
```
# 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)

# methods of
# fit(X, y) : Fit the model using X as training data and y as target values
# predict(X):Predict the class labels for the provided data
# predict_proba(X):Return probability estimates for the test data X.
#-----------------------------------
# video link: https://www.appliedaicourse.com/course/applied-ai-course-online/lessons/k-nearest-neighbors-geometric-intuition-with-a-toy-example-1/
#-----------------------------------


# find more about CalibratedClassifierCV here at http://scikit-learn.org/stable/modules/generated/sklearn.calibration.CalibratedClassifierCV.html
# ----------------------------
# default paramters
# sklearn.calibration.CalibratedClassifierCV(base_estimator=None, method='sigmoid', cv=3)
#
# some of the methods of CalibratedClassifierCV()
# fit(X, y[, sample_weight])    Fit the calibrated model
# get_params([deep])    Get parameters for this estimator.
# predict(X)    Predict the target of new samples.
# predict_proba(X)      Posterior probabilities of classification
#-----------------------------------
# video link:
#-----------------------------------


alpha = [5, 11, 15, 21, 31, 41, 51, 99]
cv_log_error_array = []
for i in alpha:
    #print("for k =", 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-probability estimates
    print("for k ={} -----> Log Loss{}".format(i,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 K")
plt.xlabel("K 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 K = ', alpha[best_alpha], "The train log loss is:",log_loss(y_train, predict_y, labels=clf.c
lasses_, eps=1e-15))
predict_y = sig_clf.predict_proba(cv_x_responseCoding)
print('For values of best K = ', alpha[best_alpha], "The cross validation log loss is:",log_loss(y_cv, predict_y, labe
ls=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(test_x_responseCoding)
print('For values of best K = ', alpha[best_alpha], "The test log loss is:",log_loss(y_test, predict_y, labels=clf.cla
sses_, eps=1e-15))
```

```
for k =5 -----> Log Loss1.103553232412858
for k =11 -----> Log Loss1.1354767313405498
for k =15 -----> Log Loss1.1464716858088795
for k =21 -----> Log Loss1.1517313740448434
for k =31 -----> Log Loss1.157241154697575
for k =41 -----> Log Loss1.1710274581174727
for k =51 -----> Log Loss1.1616512338351952
for k =99 -----> Log Loss1.1848451161994125
```

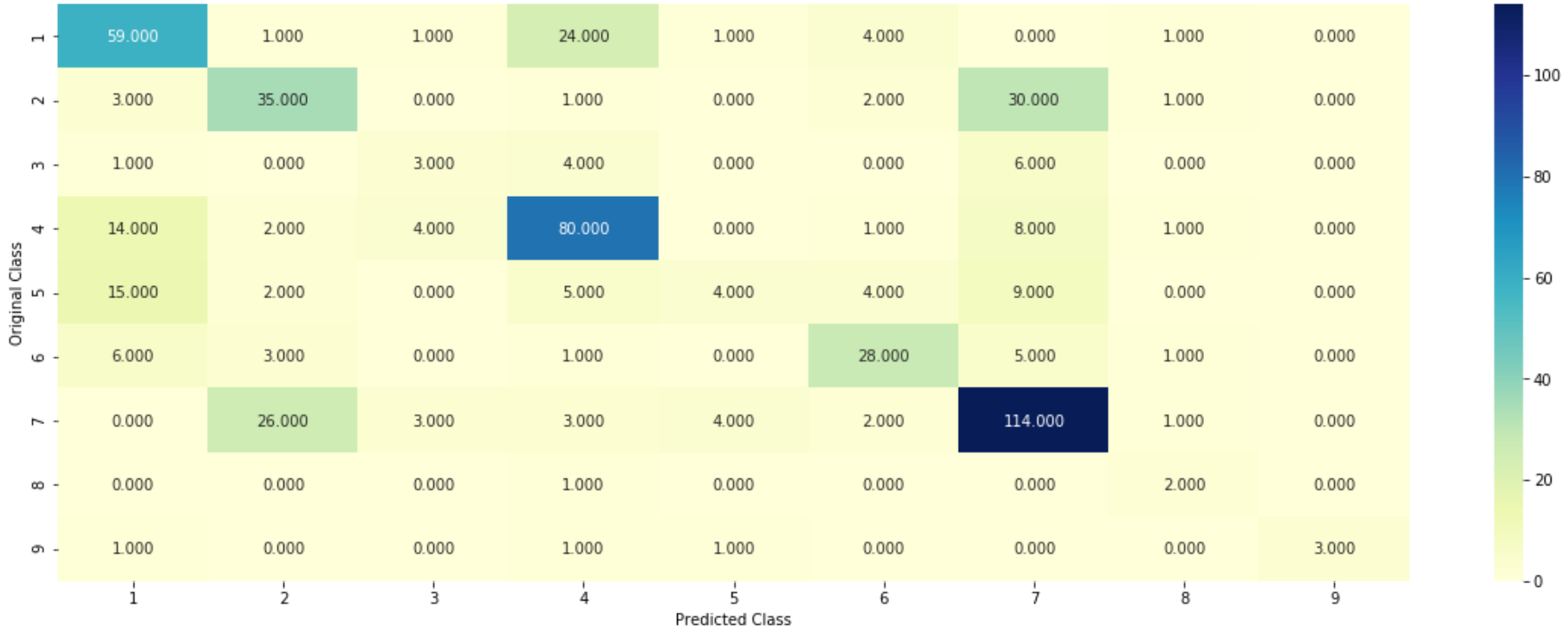



```
For values of best K =  5 The train log loss is: 0.47513289626445343
For values of best K =  5 The cross validation log loss is: 1.103553232412858
For values of best K =  5 The test log loss is: 1.0587913365269643
```

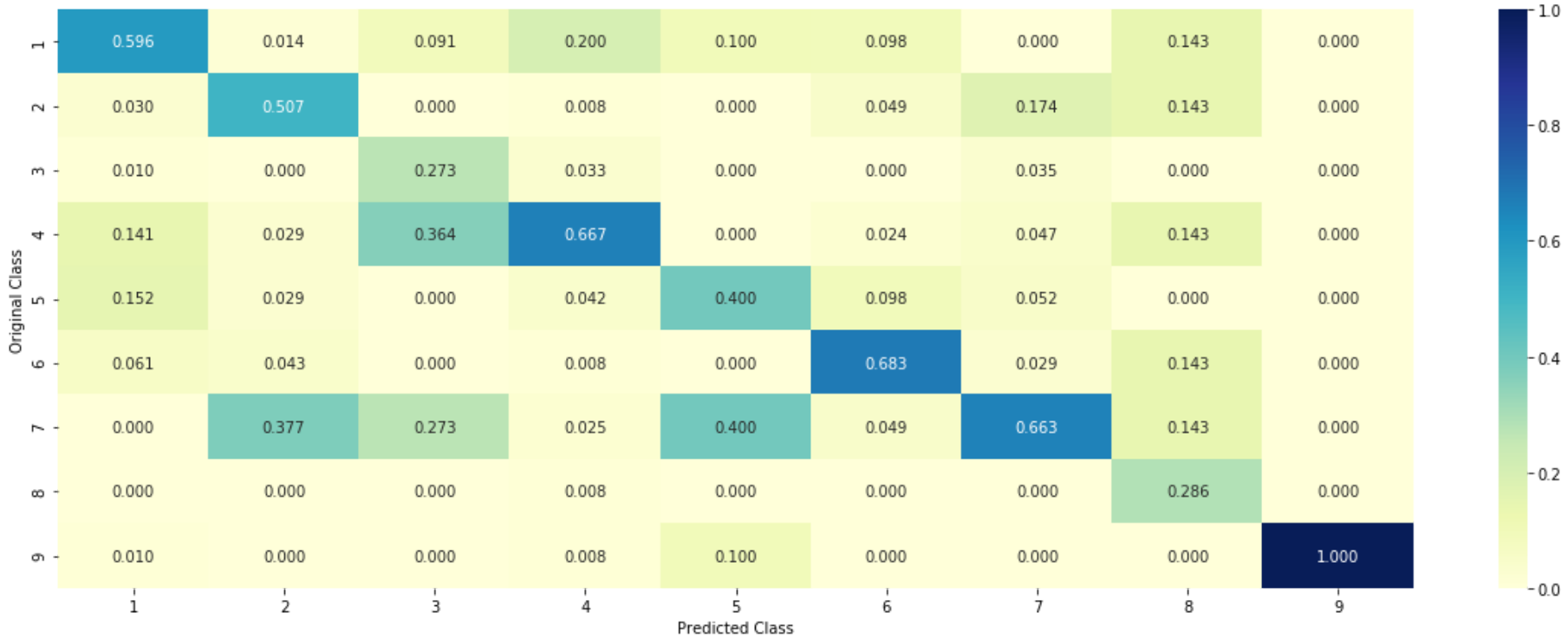
4.2.2. Testing the model with best hyper paramters

In [96]:
```
# 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)

# methods of
# fit(X, y) : Fit the model using X as training data and y as target values
# predict(X):Predict the class labels for the provided data
# predict_proba(X):Return probability estimates for the test data X.
#-----------------------------------
# video link: https://www.appliedaicourse.com/course/applied-ai-course-online/lessons/k-nearest-neighbors-geometric-intuition-with-a-toy-example-1/
#-----------------------------------
clf = KNeighborsClassifier(n_neighbors=alpha[best_alpha])
predict_and_plot_confusion_matrix(train_x_responseCoding, train_y, cv_x_responseCoding, cv_y, clf)
```

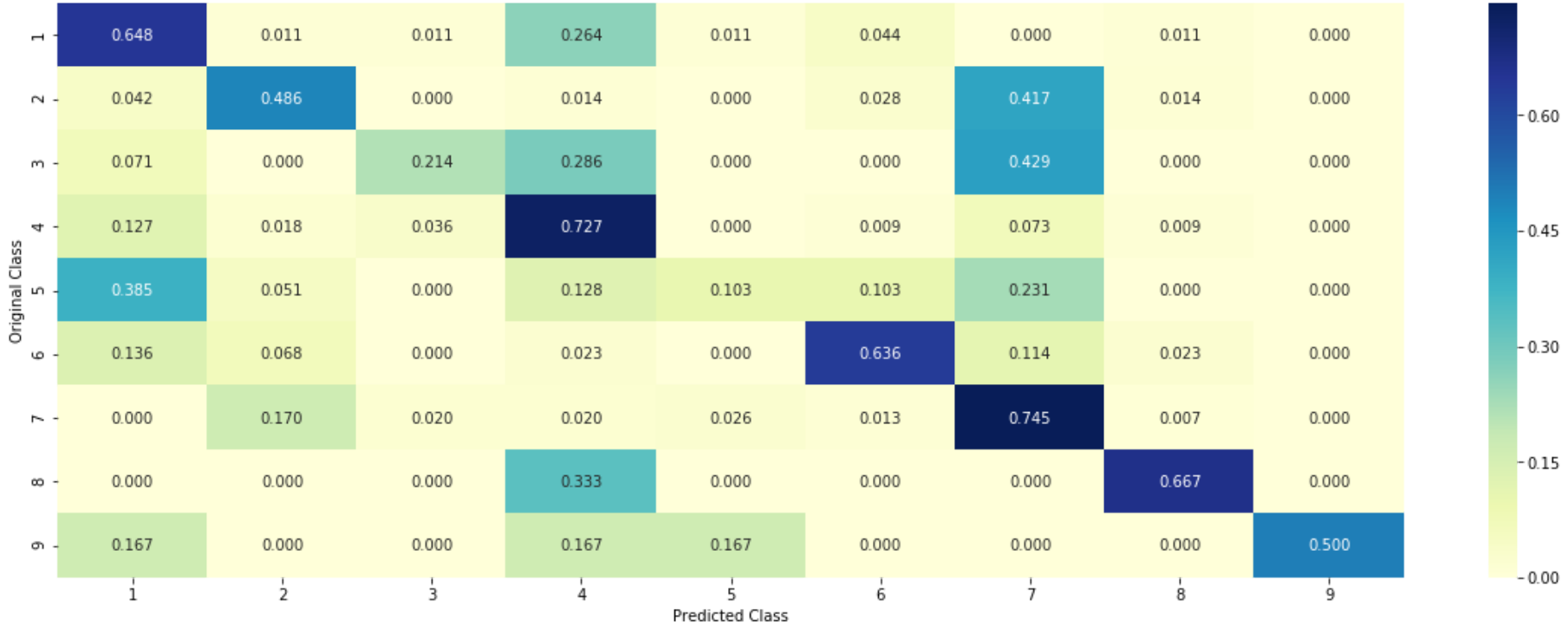
```
Log loss : 1.103553232412858
Number of mis-classified points : 0.38345864661654133
-------------------- Confusion matrix --------------------
```




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



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



In [97]:
```
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 = 1
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 : 1
Actual Class : 7
The  5  nearest neighbours of the test points belongs to classes [4 4 4 4 4]
Fequency of nearest points : Counter({4: 5})
```

4.2.4. Sample Query Point-2 for KNN

In [98]:
```
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[test_point_index].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 k value for knn is",alpha[best_alpha],"and the nearest neighbours of the test points belongs to classes",tr
ain_y[neighbors[1][0]])
print("Fequency of nearest points :",Counter(train_y[neighbors[1][0]]))
```

```
Predicted Class : 7
Actual Class : 7
the k value for knn is 5 and the nearest neighbours of the test points belongs to classes [5 7 7 1 7]
Fequency of nearest points : Counter({7: 3, 5: 1, 1: 1})
```

Task 3: Apply Logistic Regression with count vectorizer with unigram and bigram

4.3. Logistic Regression

In [26]:
```
#Making one_hot encoding features for logistic regression model by count vectorizer using unigram and bigram

# one-hot encoding of Gene feature
gene_vectorizer_LR = CountVectorizer()
train_gene_feature_onehotCoding_LR = gene_vectorizer_LR.fit_transform(train_df['Gene'])
test_gene_feature_onehotCoding_LR = gene_vectorizer_LR.transform(test_df['Gene'])
cv_gene_feature_onehotCoding_LR = gene_vectorizer_LR.transform(cv_df['Gene'])

# one-hot encoding of variation feature.
variation_vectorizer_LR = CountVectorizer()
train_variation_feature_onehotCoding_LR = variation_vectorizer_LR.fit_transform(train_df['Variation'])
test_variation_feature_onehotCoding_LR = variation_vectorizer_LR.transform(test_df['Variation'])
cv_variation_feature_onehotCoding_LR = variation_vectorizer_LR.transform(cv_df['Variation'])

#one-hot encoding for Text feature
text_vectorizer_LR = CountVectorizer(min_df=3,ngram_range=(1, 2))
train_text_feature_onehotCoding_LR = text_vectorizer_LR.fit_transform(train_df['TEXT'])
train_text_feature_onehotCoding_LR = normalize(train_text_feature_onehotCoding_LR ,axis=0)

test_text_feature_onehotCoding_LR = text_vectorizer_LR.transform(test_df['TEXT'])
test_text_feature_onehotCoding_LR = normalize(test_text_feature_onehotCoding_LR ,axis=0)

cv_text_feature_onehotCoding_LR = text_vectorizer_LR.transform(cv_df['TEXT'])
cv_text_feature_onehotCoding_LR = normalize(cv_text_feature_onehotCoding_LR ,axis=0)

#stacking all the features(gene,vartions,text of one-hot encoded)
train_gene_var_onehotCoding_LR = hstack((train_gene_feature_onehotCoding_LR ,train_variation_feature_onehotCoding_LR))
test_gene_var_onehotCoding_LR = hstack((test_gene_feature_onehotCoding_LR,test_variation_feature_onehotCoding_LR))
cv_gene_var_onehotCoding_LR = hstack((cv_gene_feature_onehotCoding_LR,cv_variation_feature_onehotCoding_LR))

train_x_onehotCoding_LR = hstack((train_gene_var_onehotCoding_LR, train_text_feature_onehotCoding_LR)).tocsr()
train_y = np.array(list(train_df['Class']))

test_x_onehotCoding_LR = hstack((test_gene_var_onehotCoding_LR, test_text_feature_onehotCoding_LR)).tocsr()
test_y = np.array(list(test_df['Class']))

cv_x_onehotCoding_LR = hstack((cv_gene_var_onehotCoding_LR, cv_text_feature_onehotCoding_LR)).tocsr()
cv_y = np.array(list(cv_df['Class']))

print("One hot encoding features :")
print("(number of data points * number of features) in train data = ", train_x_onehotCoding_LR.shape)
```

```
print("(number of data points * number of features) in test data = ", test_x_onehotCoding_LR.shape)
print("(number of data points * number of features) in cross validation data =", cv_x_onehotCoding_LR.shape)
```

```
One hot encoding features :
(number of data points * number of features) in train data =  (2124, 759560)
(number of data points * number of features) in test data =  (665, 759560)
(number of data points * number of features) in cross validation data = (532, 759560)
```

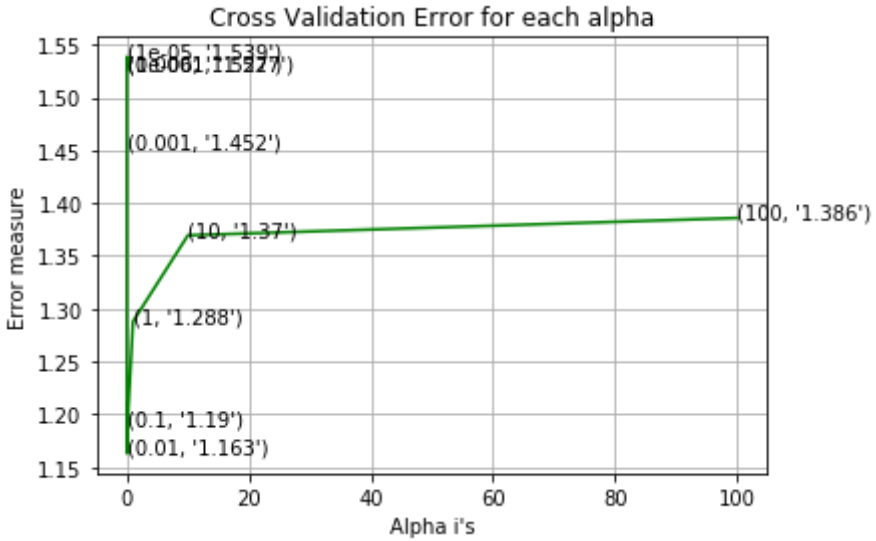
4.3.1. With Class balancing

4.3.1.1. Hyper paramter tuning

In [101]:
```
# read more about SGDClassifier() at http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html
# ------------------------------
# default parameters
# SGDClassifier(loss='hinge', penalty='l2', alpha=0.0001, l1_ratio=0.15, fit_intercept=True, max_iter=None, tol=None,
# shuffle=True, verbose=0, epsilon=0.1, n_jobs=1, random_state=None, learning_rate='optimal', eta0=0.0, power_t=0.5,
# class_weight=None, warm_start=False, average=False, n_iter=None)

# some of methods
# fit(X, y[, coef_init, intercept_init, ...])     Fit linear model with Stochastic Gradient Descent.
# predict(X)    Predict class labels for samples in X.

#-------------------------------
# video link: https://www.appliedaicourse.com/course/applied-ai-course-online/lessons/geometric-intuition-1/
#-------------------------------


# find more about CalibratedClassifierCV here at http://scikit-learn.org/stable/modules/generated/sklearn.calibration.CalibratedClassifierCV.html
# ----------------------------
# default paramters
# sklearn.calibration.CalibratedClassifierCV(base_estimator=None, method='sigmoid', cv=3)
#
# some of the methods of CalibratedClassifierCV()
# fit(X, y[, sample_weight])    Fit the calibrated model
# get_params([deep])    Get parameters for this estimator.
# predict(X)    Predict the target of new samples.
# predict_proba(X)      Posterior probabilities of classification
#-----------------------------------
# video link:
#-----------------------------------

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_onehotCoding_LR, train_y)
    sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
    sig_clf.fit(train_x_onehotCoding_LR, train_y)
    sig_clf_probs = sig_clf.predict_proba(cv_x_onehotCoding_LR)
```

```
    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-probability 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_onehotCoding_LR, train_y)
sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
sig_clf.fit(train_x_onehotCoding_LR, train_y)

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

```
for alpha = 1e-06
Log Loss : 1.526971311207314
for alpha = 1e-05
Log Loss : 1.5386623502716792
for alpha = 0.0001
Log Loss : 1.5266915853209928
for alpha = 0.001
Log Loss : 1.4522367672319636
for alpha = 0.01
Log Loss : 1.1628639564707144
for alpha = 0.1
Log Loss : 1.1899739328129815
for alpha = 1
Log Loss : 1.287797395891601
for alpha = 10
Log Loss : 1.3695018048796566
for alpha = 100
Log Loss : 1.3859080701489557
```

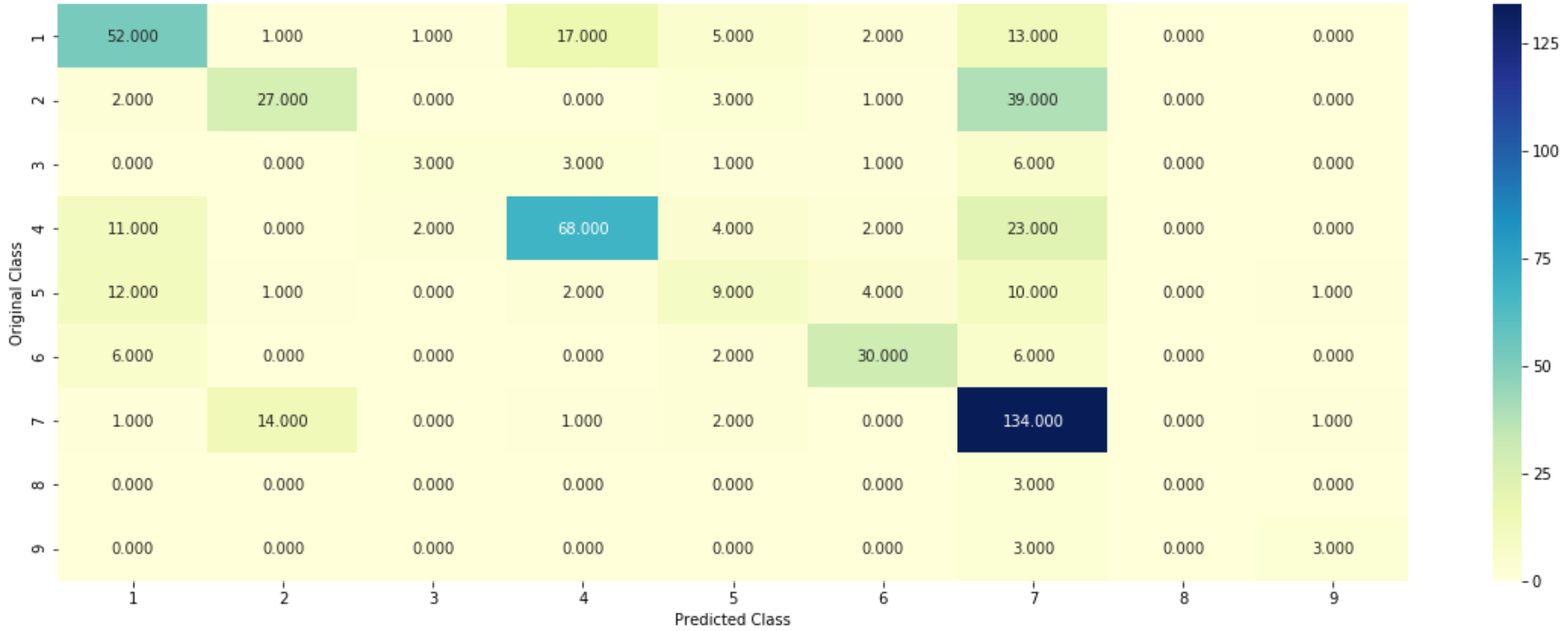



```
For values of best alpha =  0.01 The train log loss is: 0.8765083328319367
For values of best alpha =  0.01 The cross validation log loss is: 1.1628639564707144
For values of best alpha =  0.01 The test log loss is: 1.1952757829548217
```

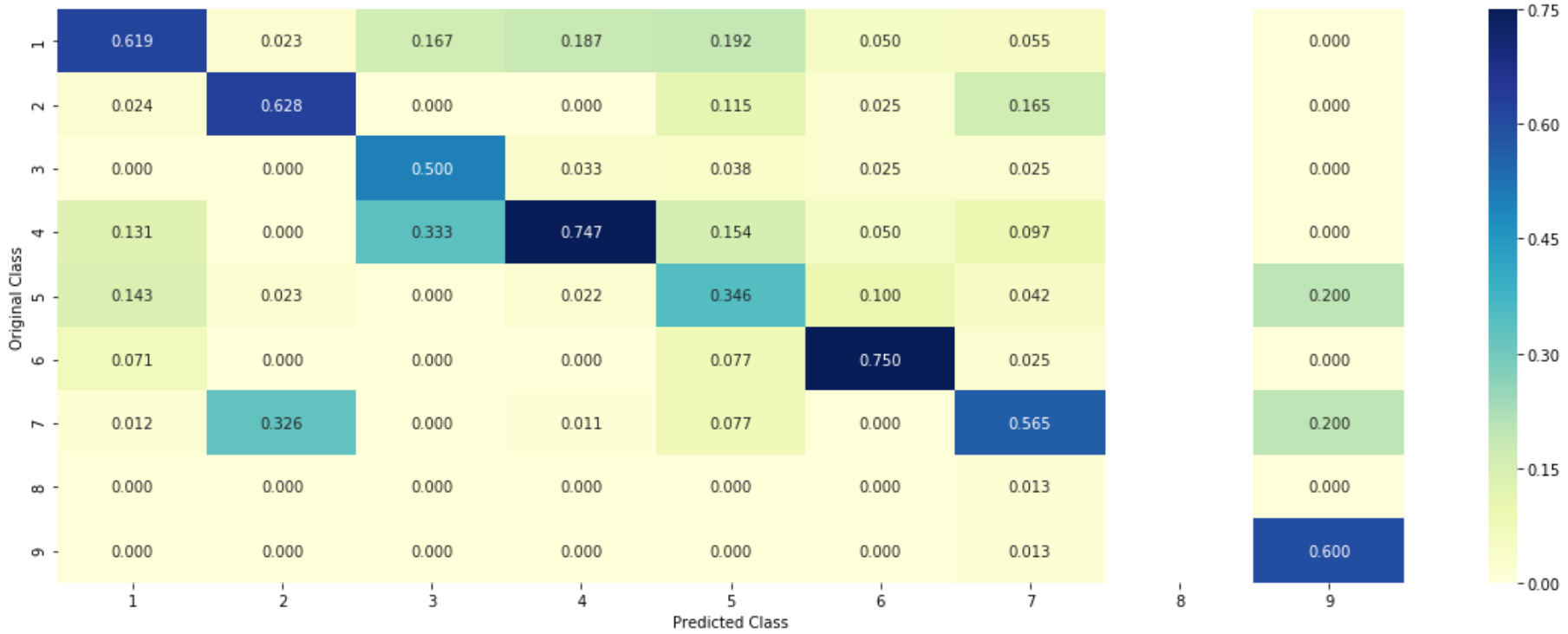
4.3.1.2. Testing the model with best hyper paramters

In [102]:
```
# read more about SGDClassifier() at http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html
# ------------------------------
# default parameters
# SGDClassifier(loss='hinge', penalty='l2', alpha=0.0001, l1_ratio=0.15, fit_intercept=True, max_iter=None, tol=None,
# shuffle=True, verbose=0, epsilon=0.1, n_jobs=1, random_state=None, learning_rate='optimal', eta0=0.0, power_t=0.5,
# class_weight=None, warm_start=False, average=False, n_iter=None)

# some of methods
# fit(X, y[, coef_init, intercept_init, ...])     Fit linear model with Stochastic Gradient Descent.
# predict(X)    Predict class labels for samples in X.

#-------------------------------
# video link: https://www.appliedaicourse.com/course/applied-ai-course-online/lessons/geometric-intuition-1/
#-------------------------------
clf = SGDClassifier(class_weight='balanced', alpha=alpha[best_alpha], penalty='l2', loss='log', random_state=42)
predict_and_plot_confusion_matrix(train_x_onehotCoding_LR, train_y, cv_x_onehotCoding_LR, cv_y, clf)
```

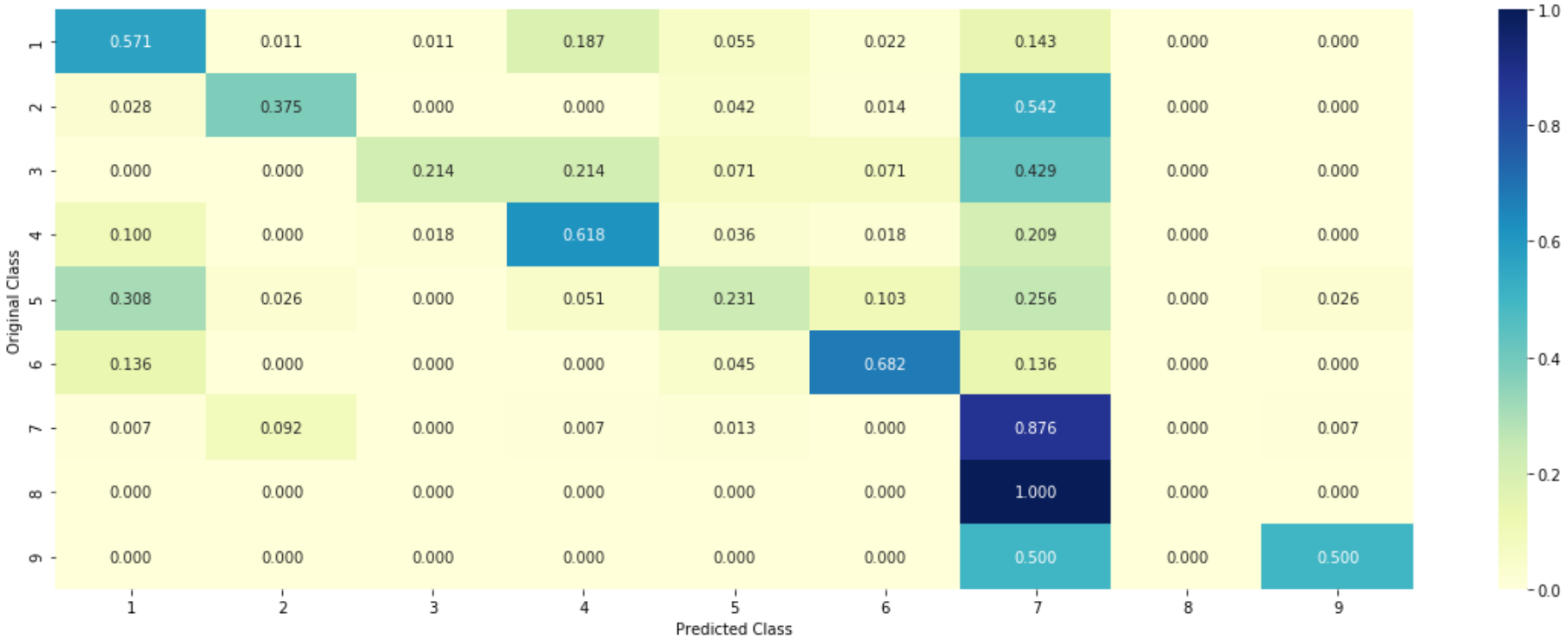
```
Log loss : 1.1628639564707144
Number of mis-classified points : 0.38721804511278196
-------------------- Confusion matrix --------------------
```




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



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



4.3.1.3. Feature Importance

In [103]:
```
# 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_LR(indices, text, gene, var, no_features):
    gene_count_vec = CountVectorizer()
    var_count_vec = CountVectorizer()
    text_count_vec = CountVectorizer(ngram_range=(1, 2))

    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_count_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 [{}]".format(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 [{}]".format(word,yes_no))

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

4.3.1.3.1. Correctly Classified point

In [104]:
```
train_x_onehotCoding.shape
train_y.shape
```

Out[104]:
```
(2124,)
```

In [105]:
```
clf = SGDClassifier(class_weight='balanced', alpha=alpha[best_alpha], penalty='l2', loss='log', random_state=42)
clf.fit(train_x_onehotCoding_LR,train_y)
test_point_index =1
no_feature = 500
predicted_cls = sig_clf.predict(test_x_onehotCoding_LR[test_point_index])
print("Predicted Class :", predicted_cls[0])
print("Predicted Class Probabilities:", np.round(sig_clf.predict_proba(test_x_onehotCoding_LR[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_LR(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: [[0.0694 0.2297 0.0146 0.4883 0.0361 0.011  0.1338 0.0108 0.0063]]
Actual Class : 7
--------------------------------------------------
261 Text feature [carcinomas] present in test data point [True]
Out of the top  500  features  1 are present in query point
```

4.3.1.3.2. Incorrectly Classified point

In [107]:
```
test_point_index = 100
no_feature = 100
predicted_cls = sig_clf.predict(test_x_onehotCoding_LR[test_point_index])
print("Predicted Class :", predicted_cls[0])
print("Predicted Class Probabilities:", np.round(sig_clf.predict_proba(test_x_onehotCoding_LR[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_LR(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 : 7
Predicted Class Probabilities: [[0.1732 0.1516 0.0234 0.1351 0.0713 0.0554 0.3774 0.0057 0.007 ]]
Actual Class : 7
--------------------------------------------------
Out of the top  100  features  0 are present in query point
```

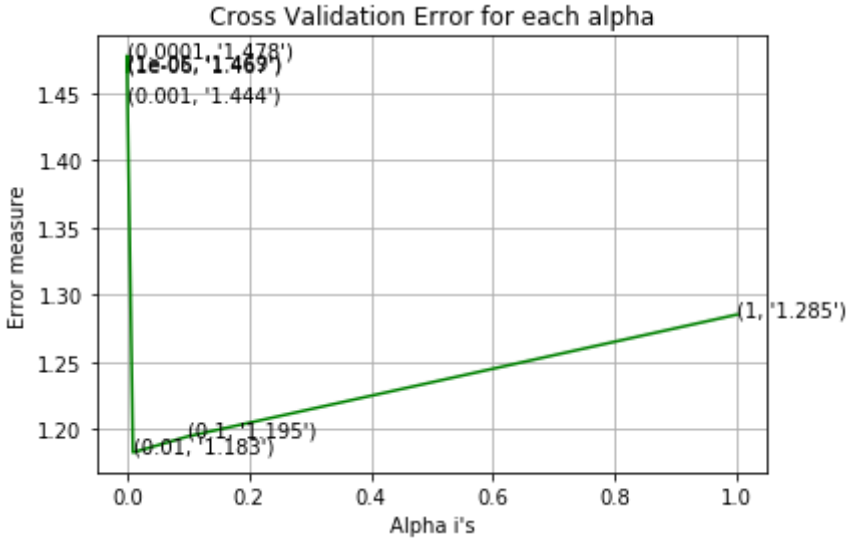
4.3.2. Without Class balancing

4.3.2.1. Hyper paramter tuning

In [108]:
```
# read more about SGDClassifier() at http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html
# ------------------------------
# default parameters
# SGDClassifier(loss='hinge', penalty='l2', alpha=0.0001, l1_ratio=0.15, fit_intercept=True, max_iter=None, tol=None,
# shuffle=True, verbose=0, epsilon=0.1, n_jobs=1, random_state=None, learning_rate='optimal', eta0=0.0, power_t=0.5,
# class_weight=None, warm_start=False, average=False, n_iter=None)

# some of methods
# fit(X, y[, coef_init, intercept_init, ...])     Fit linear model with Stochastic Gradient Descent.
# predict(X)    Predict class labels for samples in X.

#-------------------------------
# video link: https://www.appliedaicourse.com/course/applied-ai-course-online/lessons/geometric-intuition-1/
#-------------------------------



# find more about CalibratedClassifierCV here at http://scikit-learn.org/stable/modules/generated/sklearn.calibration.CalibratedClassifierCV.html
# ----------------------------
# default paramters
# sklearn.calibration.CalibratedClassifierCV(base_estimator=None, method='sigmoid', cv=3)
#
# some of the methods of CalibratedClassifierCV()
# fit(X, y[, sample_weight])    Fit the calibrated model
# get_params([deep])    Get parameters for this estimator.
# predict(X)    Predict the target of new samples.
# predict_proba(X)      Posterior probabilities of classification
#-----------------------------------
# video link:
#-----------------------------------

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

```
    sig_clf_probs = sig_clf.predict_proba(cv_x_onehotCoding_LR)
    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)# here we have not written clas
s_weight='balanced'
clf.fit(train_x_onehotCoding_LR, train_y)
sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
sig_clf.fit(train_x_onehotCoding_LR, train_y)

predict_y = sig_clf.predict_proba(train_x_onehotCoding_LR)
print('For values of best alpha = ', alpha[best_alpha], "The train log loss is:",log_loss(y_train, predict_y, labels=c
lf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(cv_x_onehotCoding_LR)
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_onehotCoding_LR)
print('For values of best alpha = ', alpha[best_alpha], "The test log loss is:",log_loss(y_test, predict_y, labels=clf
.classes_, eps=1e-15))
```

```
for alpha = 1e-06
Log Loss : 1.4665532137033077
for alpha = 1e-05
Log Loss : 1.4685171825273864
for alpha = 0.0001
Log Loss : 1.477599201777331
for alpha = 0.001
Log Loss : 1.4435520394153087
for alpha = 0.01
Log Loss : 1.1829103454047591
for alpha = 0.1
Log Loss : 1.1948630032134728
for alpha = 1
Log Loss : 1.2853418476335345
```

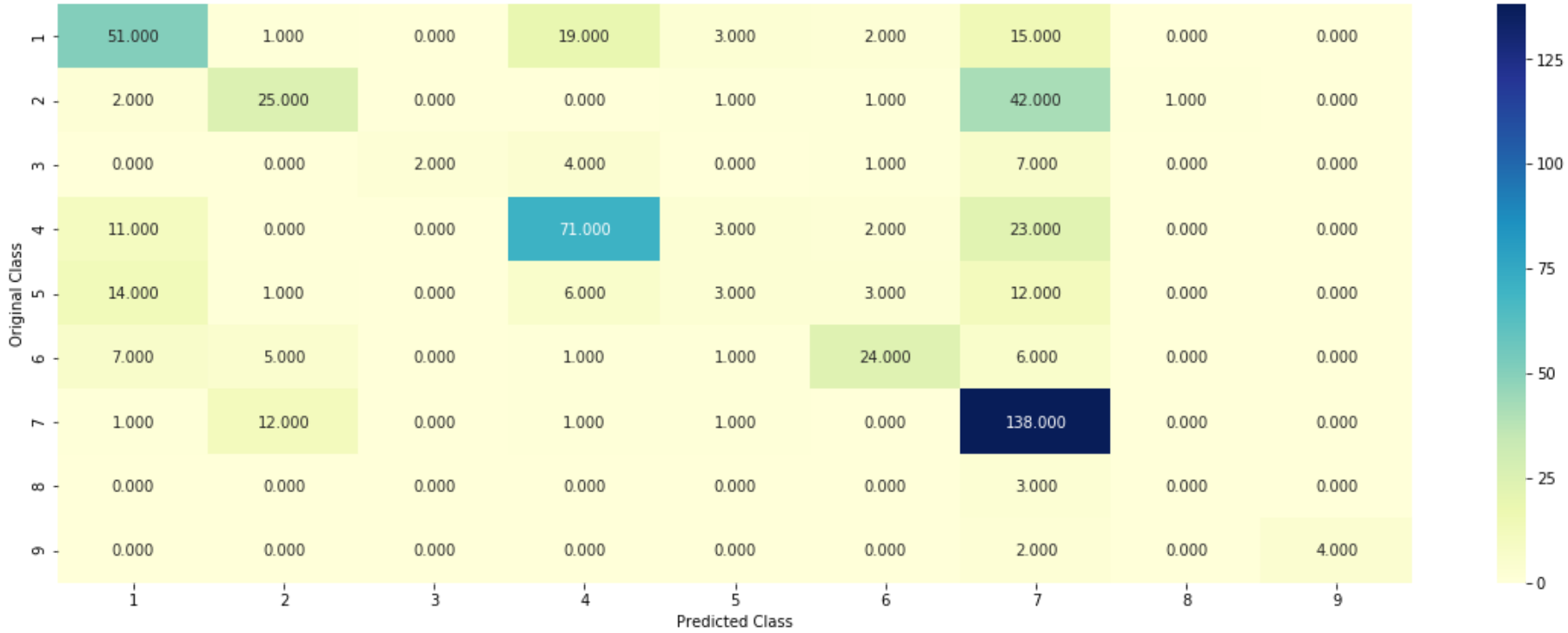



```
For values of best alpha =  0.01 The train log loss is: 0.8886156253613645
For values of best alpha =  0.01 The cross validation log loss is: 1.1829103454047591
For values of best alpha =  0.01 The test log loss is: 1.2225456836748452
```

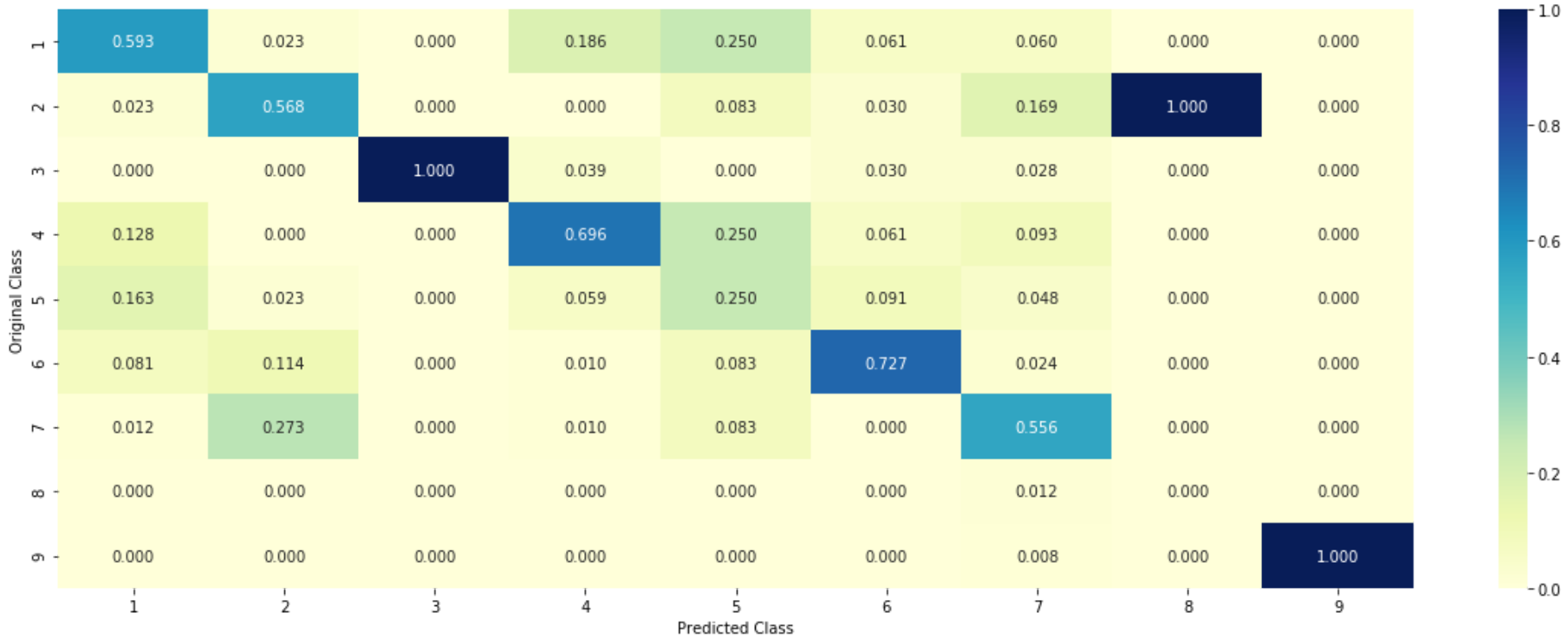
4.3.2.2. Testing model with best hyper parameters

In [109]:
```
# read more about SGDClassifier() at http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html
# ------------------------------
# default parameters
# SGDClassifier(loss='hinge', penalty='l2', alpha=0.0001, l1_ratio=0.15, fit_intercept=True, max_iter=None, tol=None,
# shuffle=True, verbose=0, epsilon=0.1, n_jobs=1, random_state=None, learning_rate='optimal', eta0=0.0, power_t=0.5,
# class_weight=None, warm_start=False, average=False, n_iter=None)

# some of methods
# fit(X, y[, coef_init, intercept_init, ...])     Fit linear model with Stochastic Gradient Descent.
# predict(X)    Predict class labels for samples in X.

#-------------------------------
# video link:
#-------------------------------

clf = SGDClassifier(alpha=alpha[best_alpha], penalty='l2', loss='log', random_state=42)
predict_and_plot_confusion_matrix(train_x_onehotCoding_LR, train_y, cv_x_onehotCoding_LR, cv_y, clf)
```

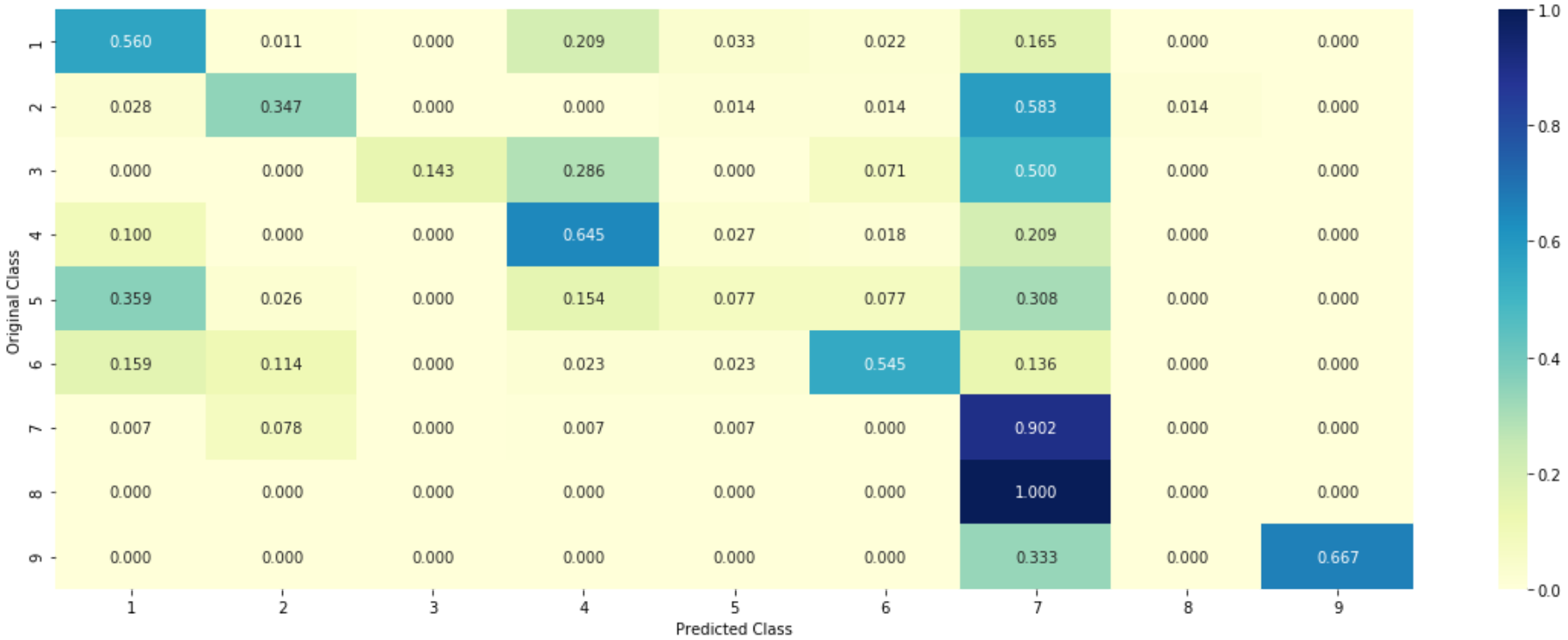
```
Log loss : 1.1829103454047591
Number of mis-classified points : 0.40225563909774437
-------------------- Confusion matrix --------------------
```




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



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



4.3.2.3. Feature Importance, Correctly Classified point

In [110]:
```
clf = SGDClassifier(alpha=alpha[best_alpha], penalty='l2', loss='log', random_state=42)
clf.fit(train_x_onehotCoding_LR,train_y)
test_point_index = 1
no_feature = 500
predicted_cls = sig_clf.predict(test_x_onehotCoding_LR[test_point_index])
print("Predicted Class :", predicted_cls[0])
print("Predicted Class Probabilities:", np.round(sig_clf.predict_proba(test_x_onehotCoding_LR[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_LR(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: [[0.0791 0.2113 0.0075 0.5009 0.0272 0.0087 0.1554 0.009  0.001 ]]
Actual Class : 7
--------------------------------------------------
220 Text feature [carcinomas] present in test data point [True]
Out of the top  500  features  1 are present in query point
```

4.3.2.4. Feature Importance, Inorrectly Classified point

In [111]:
```
test_point_index = 100
no_feature = 500
predicted_cls = sig_clf.predict(test_x_onehotCoding_LR[test_point_index])
print("Predicted Class :", predicted_cls[0])
print("Predicted Class Probabilities:", np.round(sig_clf.predict_proba(test_x_onehotCoding_LR[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)
```

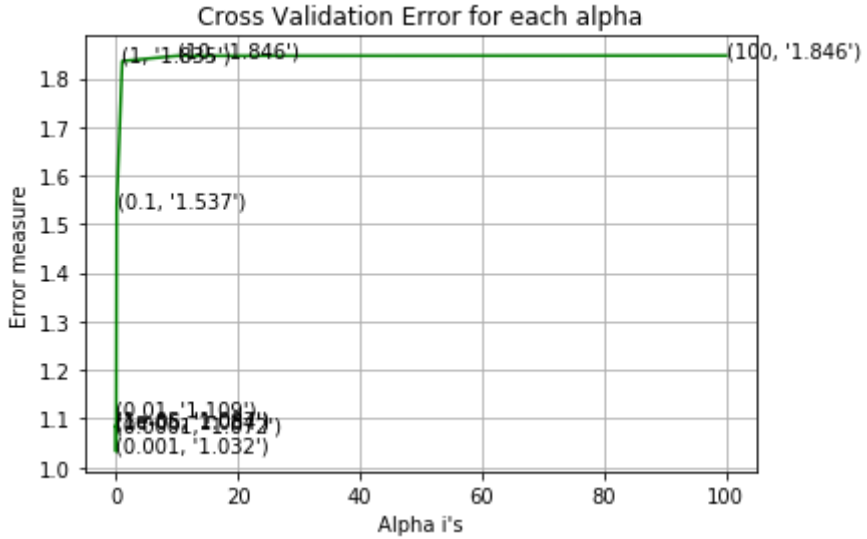
```
Predicted Class : 7
Predicted Class Probabilities: [[0.1721 0.1521 0.0278 0.1312 0.0754 0.0586 0.3698 0.0064 0.0065]]
Actual Class : 7
--------------------------------------------------
Out of the top  500  features  0 are present in query point
```

Logistic Regression using TFIDF vectorizer with feature engineering features

In [112]:
```
# read more about SGDClassifier() at http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html
# ------------------------------

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_onehotCodingFE, train_y)
    sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
    sig_clf.fit(train_x_onehotCodingFE, train_y)
    sig_clf_probs = sig_clf.predict_proba(cv_x_onehotCodingFE)
    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-probability 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_onehotCodingFE, train_y)
sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
sig_clf.fit(train_x_onehotCodingFE, train_y)

predict_y = sig_clf.predict_proba(train_x_onehotCodingFE)
print('For values of best alpha = ', alpha[best_alpha], "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_onehotCodingFE)
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_onehotCodingFE)
```

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

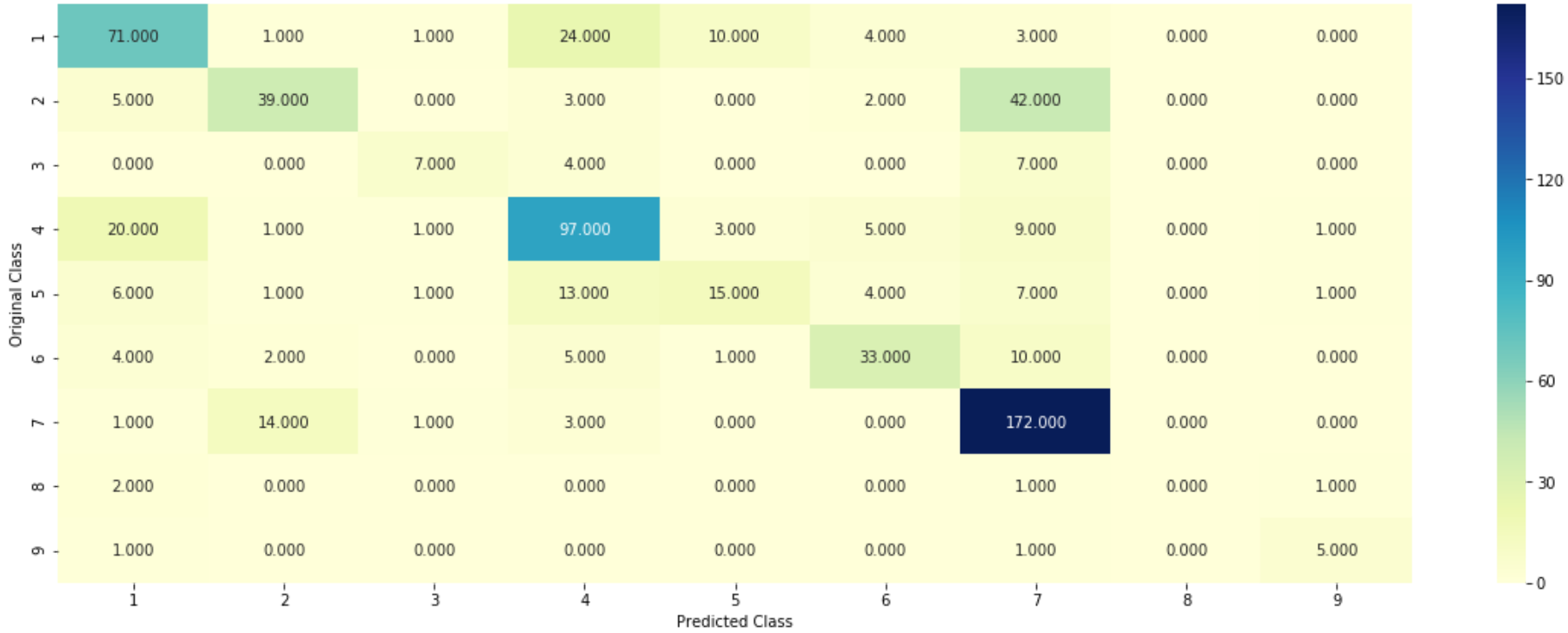
```
for alpha = 1e-06
Log Loss : 1.083821739016169
for alpha = 1e-05
Log Loss : 1.0868178423544974
for alpha = 0.0001
Log Loss : 1.0724055267477606
for alpha = 0.001
Log Loss : 1.032136557996591
for alpha = 0.01
Log Loss : 1.1088810097920543
for alpha = 0.1
Log Loss : 1.5367919152002336
for alpha = 1
Log Loss : 1.835137915131387
for alpha = 10
Log Loss : 1.8459414645922774
for alpha = 100
Log Loss : 1.8464178871572867
```




```
For values of best alpha =  0.001 The train log loss is: 0.6123783471421275
For values of best alpha =  0.001 The cross validation log loss is: 1.032136557996591
For values of best alpha =  0.001 The test log loss is: 0.9465105468033816
```

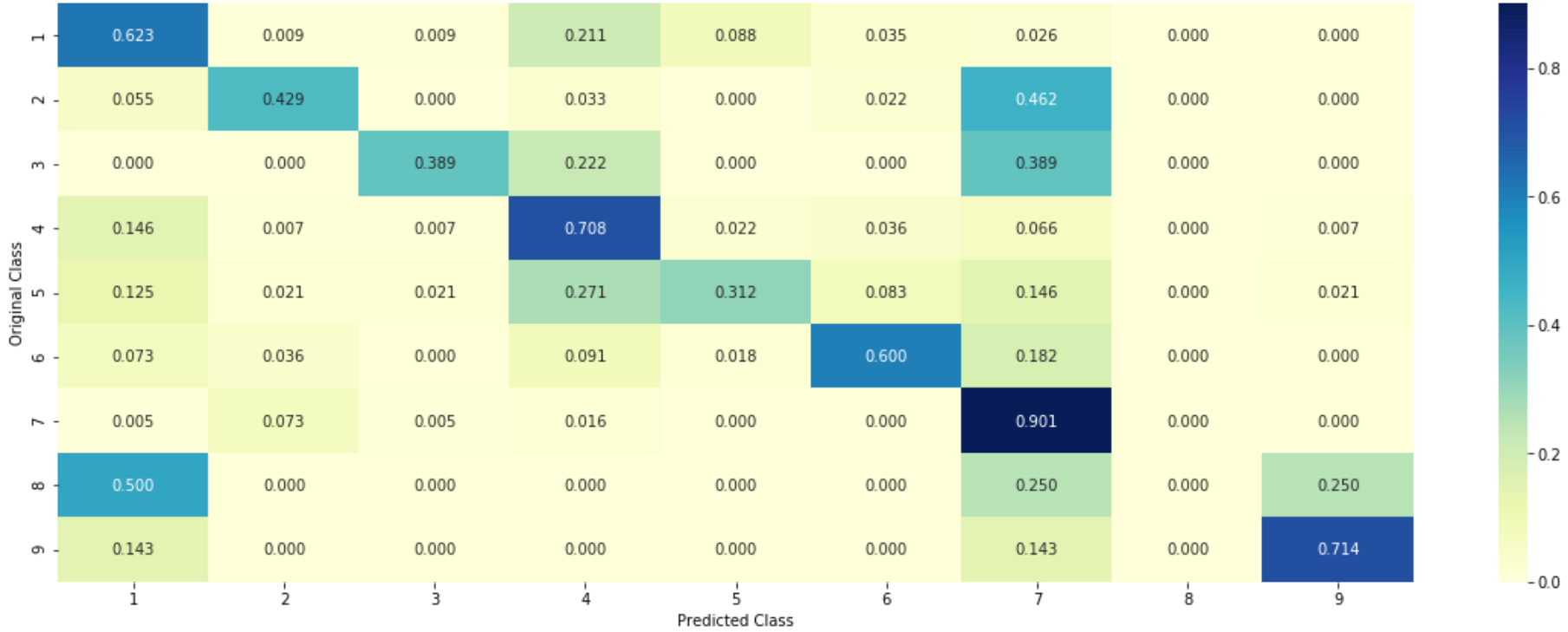
In [113]:
```
# read more about SGDClassifier() at http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html
# ------------------------------
# default parameters
# SGDClassifier(loss='hinge', penalty='l2', alpha=0.0001, l1_ratio=0.15, fit_intercept=True, max_iter=None, tol=None,
# shuffle=True, verbose=0, epsilon=0.1, n_jobs=1, random_state=None, learning_rate='optimal', eta0=0.0, power_t=0.5,
# class_weight=None, warm_start=False, average=False, n_iter=None)

# some of methods
# fit(X, y[, coef_init, intercept_init, ...])     Fit linear model with Stochastic Gradient Descent.
# predict(X)    Predict class labels for samples in X.

#------------------------------
# video link: https://www.appliedaicourse.com/course/applied-ai-course-online/lessons/geometric-intuition-1/
#------------------------------
clf = SGDClassifier(class_weight='balanced', alpha=alpha[best_alpha], penalty='l2', loss='log', random_state=42)
predict_and_plot_confusion_matrix(train_x_onehotCodingFE, train_y, test_x_onehotCodingFE, test_y, clf)
```

```
Log loss : 0.9465105468033816
Number of mis-classified points : 0.3398496240601504
-------------------- Confusion matrix --------------------
```




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

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



Obervation

We use logistic regression using TFIDF vectorizer for all three feature such as Gene,Variation,Text and implemented feature engineering as square root of vectorizer and found log-loss is 0.946 and misclassified points are 0.339

Correctly Classified points

In [114]:
```
# from tabulate import tabulate
clf = SGDClassifier(class_weight='balanced', alpha=alpha[best_alpha], penalty='l2', loss='log', random_state=42)
clf.fit(train_x_onehotCoding,train_y)
test_point_index = 1
no_feature = 100
predicted_cls = sig_clf.predict(test_x_onehotCoding[test_point_index])
print("Predicted Class :", predicted_cls[0])
print("Predicted Class Probabilities:", np.round(sig_clf.predict_proba(test_x_onehotCoding[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: [[0.1925 0.2022 0.0206 0.2945 0.0628 0.0752 0.1104 0.0051 0.0368]]
Actual Class : 7
--------------------------------------------------
Out of the top  100  features  0 are present in query point
```

Incorrectly Classified point

In [115]:
```
test_point_index = 100
no_feature = 500
predicted_cls = sig_clf.predict(test_x_onehotCoding[test_point_index])
print("Predicted Class :", predicted_cls[0])
print("Predicted Class Probabilities:", np.round(sig_clf.predict_proba(test_x_onehotCoding[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 : 2
Predicted Class Probabilities: [[0.1778 0.2119 0.0268 0.1155 0.1657 0.0874 0.1918 0.0045 0.0187]]
Actual Class : 7
--------------------------------------------------
Out of the top  500  features  0 are present in query point
```

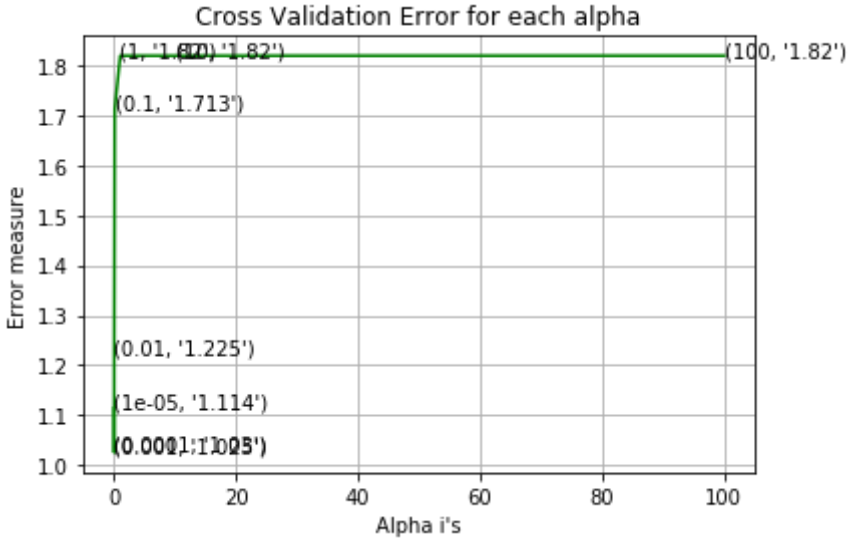
4.4. Linear Support Vector Machines

4.4.1. Hyper paramter tuning

In [31]:
```
# read more about support vector machines with linear kernals here http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

# --------------------------------
# default parameters
# SVC(C=1.0, kernel='rbf', degree=3, gamma='auto', coef0=0.0, shrinking=True, probability=False, tol=0.001,
# cache_size=200, class_weight=None, verbose=False, max_iter=-1, decision_function_shape='ovr', random_state=None)

# Some of methods of SVM()
# fit(X, y, [sample_weight])    Fit the SVM model according to the given training data.
# predict(X)    Perform classification on samples in X.
# --------------------------------
# video link: https://www.appliedaicourse.com/course/applied-ai-course-online/lessons/mathematical-derivation-copy-8/
# --------------------------------



# find more about CalibratedClassifierCV here at http://scikit-learn.org/stable/modules/generated/sklearn.calibration.CalibratedClassifierCV.html
# ----------------------------
# default paramters
# sklearn.calibration.CalibratedClassifierCV(base_estimator=None, method='sigmoid', cv=3)
#
# some of the methods of CalibratedClassifierCV()
# fit(X, y[, sample_weight])    Fit the calibrated model
# get_params([deep])    Get parameters for this estimator.
# predict(X)    Predict the target of new samples.
# predict_proba(X)      Posterior probabilities of classification
#-------------------------------------
# video link:
#-------------------------------------

alpha = [10 ** x for x in range(-5, 3)]
cv_log_error_array = []
for i in alpha:
    print("for C =", i)
#     clf = SVC(C=i,kernel='linear',probability=True, class_weight='balanced')
    clf = SGDClassifier( class_weight='balanced', alpha=i, penalty='l2', loss='hinge', random_state=42)
    clf.fit(train_x_onehotCoding, train_y)
    sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
    sig_clf.fit(train_x_onehotCoding, train_y)
```

```
    sig_clf_probs = sig_clf.predict_proba(cv_x_onehotCoding)
    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 = SVC(C=i,kernel='linear',probability=True, class_weight='balanced')
clf = SGDClassifier(class_weight='balanced', alpha=alpha[best_alpha], penalty='l2', loss='hinge', random_state=42)
clf.fit(train_x_onehotCoding, train_y)
sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
sig_clf.fit(train_x_onehotCoding, train_y)

predict_y = sig_clf.predict_proba(train_x_onehotCoding)
print('For values of best alpha = ', alpha[best_alpha], "The train log loss is:",log_loss(y_train, predict_y, labels=c
lf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(cv_x_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_x_onehotCoding)
print('For values of best alpha = ', alpha[best_alpha], "The test log loss is:",log_loss(y_test, predict_y, labels=clf
.classes_, eps=1e-15))
```

```
for C = 1e-05
Log Loss : 1.1137049321264103
for C = 0.0001
Log Loss : 1.0295760364061246
for C = 0.001
Log Loss : 1.0251509418867966
for C = 0.01
Log Loss : 1.2248003696516199
for C = 0.1
Log Loss : 1.713120337856678
for C = 1
Log Loss : 1.8196045692242897
for C = 10
Log Loss : 1.819604484711596
for C = 100
Log Loss : 1.819604503074025
```

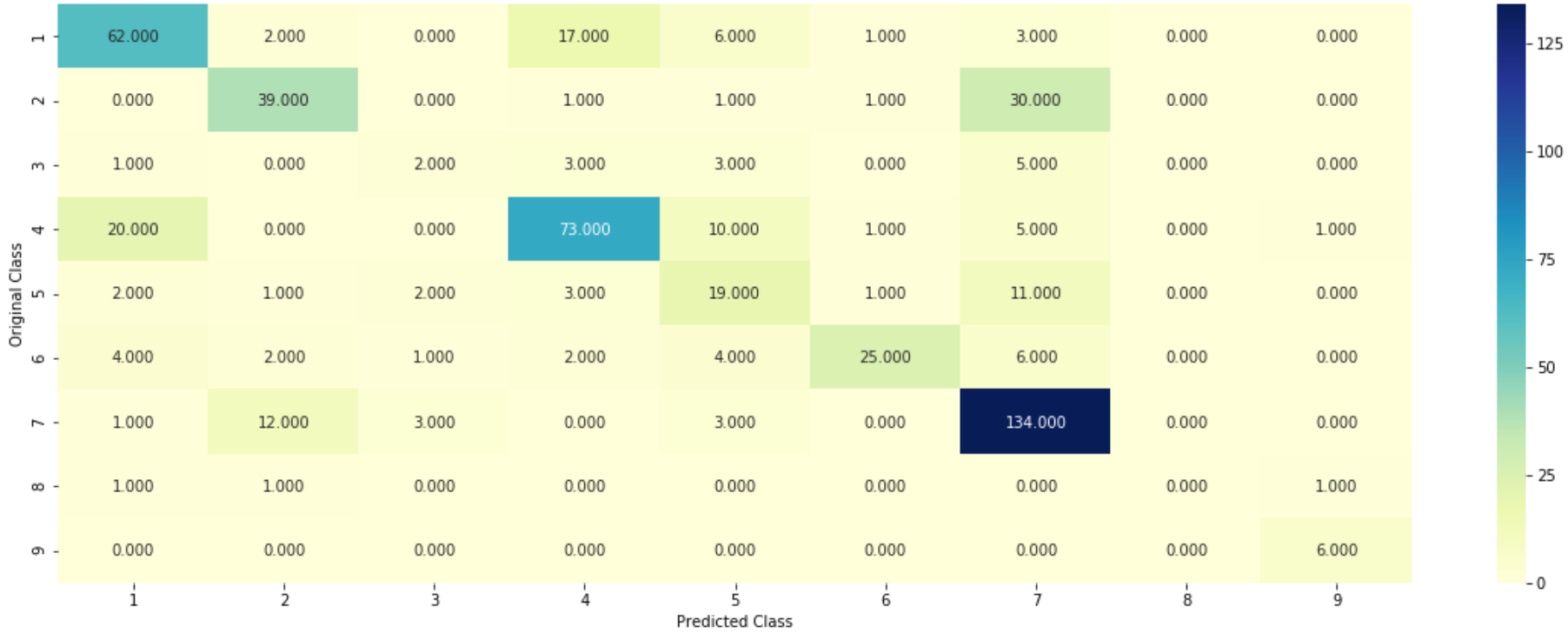



```
For values of best alpha =  0.001 The train log loss is: 0.5684338035686777
For values of best alpha =  0.001 The cross validation log loss is: 1.0251509418867966
For values of best alpha =  0.001 The test log loss is: 1.0064956595650647
```

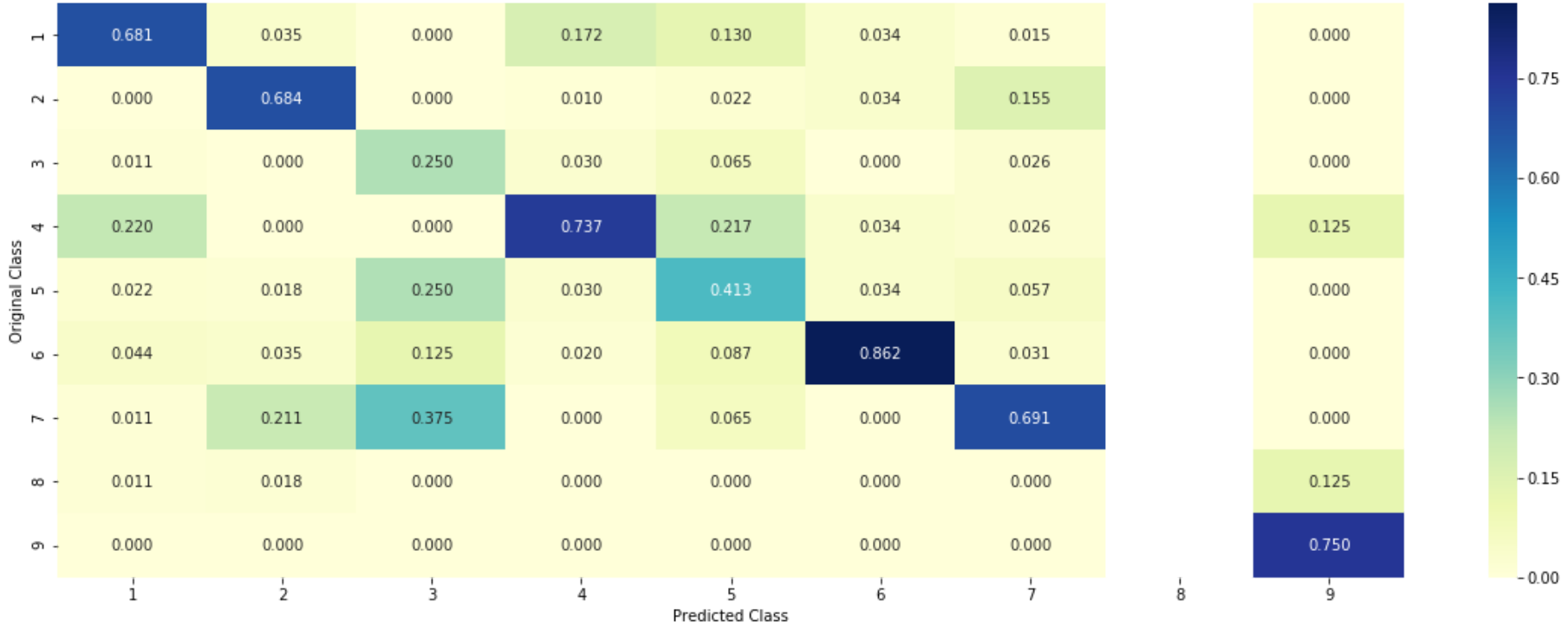
4.4.2. Testing model with best hyper parameters

In [36]:
```
# read more about support vector machines with linear kernals here http://scikit-learn.org/stable/modules/generated/sk
learn.svm.SVC.html

# ----------------------------------
# default parameters
# SVC(C=1.0, kernel='rbf', degree=3, gamma='auto', coef0=0.0, shrinking=True, probability=False, tol=0.001,
# cache_size=200, class_weight=None, verbose=False, max_iter=-1, decision_function_shape='ovr', random_state=None)

# Some of methods of SVM()
# fit(X, y, [sample_weight])    Fit the SVM model according to the given training data.
# predict(X)    Perform classification on samples in X.
# ----------------------------------
# video link: https://www.appliedaicourse.com/course/applied-ai-course-online/lessons/mathematical-derivation-copy-8/
# ----------------------------------


# clf = SVC(C=alpha[best_alpha],kernel='linear',probability=True, class_weight='balanced')
clf = SGDClassifier(alpha=alpha[best_alpha], penalty='l2', loss='hinge', random_state=42,class_weight='balanced')
predict_and_plot_confusion_matrix(train_x_onehotCoding, train_y,cv_x_onehotCoding,cv_y, clf)
```

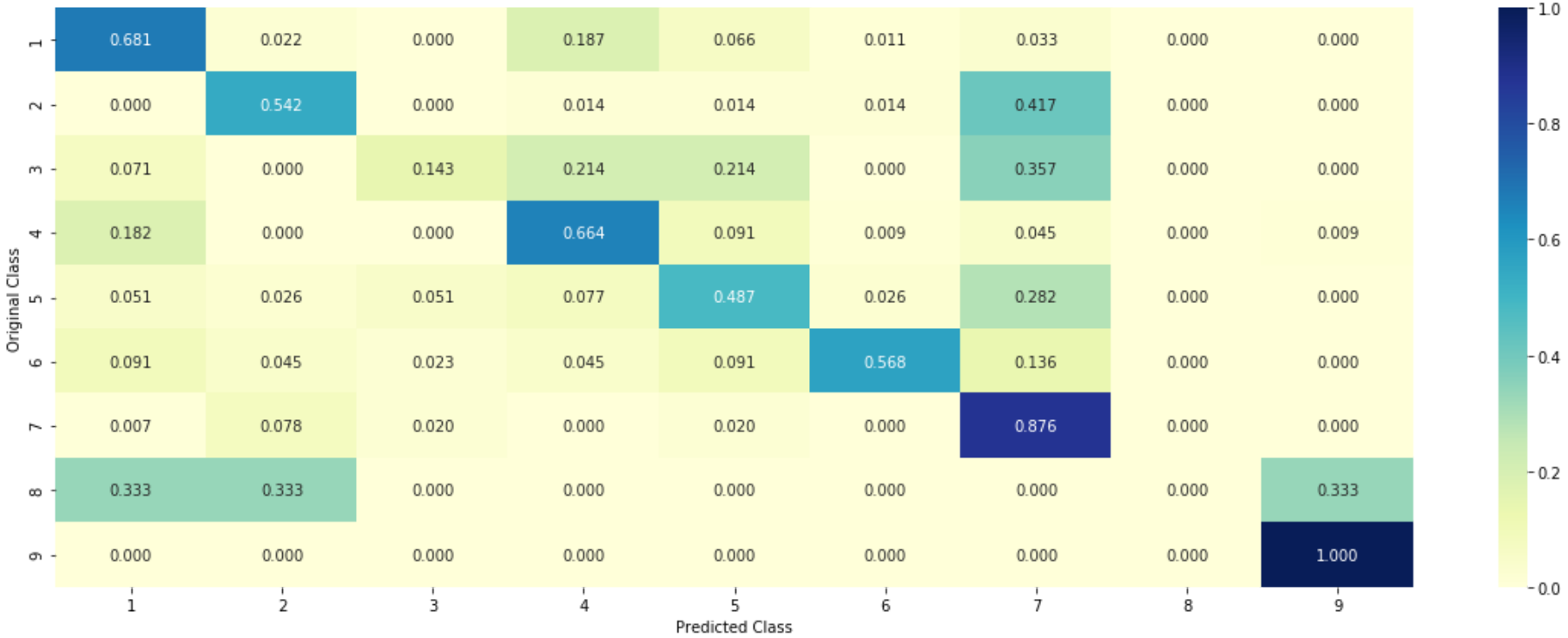
```
Log loss : 1.0251509418867966
Number of mis-classified points : 0.3233082706766917
-------------------- Confusion matrix --------------------
```




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



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



4.3.3. Feature Importance

4.3.3.1. For Correctly classified point

In [39]:
```
%%time
clf = SGDClassifier(alpha=alpha[best_alpha], penalty='l2', loss='hinge', random_state=42)
clf.fit(train_x_onehotCoding,train_y)
test_point_index = 1
# test_point_index = 100
no_feature = 500
predicted_cls = sig_clf.predict(test_x_onehotCoding[test_point_index])
print("Predicted Class :", predicted_cls[0])
print("Predicted Class Probabilities:", np.round(sig_clf.predict_proba(test_x_onehotCoding[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 : 7
Predicted Class Probabilities: [[2.850e-02 5.860e-02 3.400e-03 4.060e-02 2.640e-02 1.680e-02 8.221e-01
  4.000e-04 3.200e-03]]
Actual Class : 7
--------------------------------------------------
Out of the top  500  features  0 are present in query point
Wall time: 5min 39s
```

4.3.3.2. For Incorrectly classified point

In [40]:
```
%%time
test_point_index = 100
no_feature = 500
predicted_cls = sig_clf.predict(test_x_onehotCoding[test_point_index])
print("Predicted Class :", predicted_cls[0])
print("Predicted Class Probabilities:", np.round(sig_clf.predict_proba(test_x_onehotCoding[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 : 7
Predicted Class Probabilities: [[0.0525 0.1666 0.0012 0.0433 0.0501 0.0452 0.6352 0.0012 0.0047]]
Actual Class : 7
--------------------------------------------------
Out of the top  500  features  0 are present in query point
Wall time: 5min 42s
```

4.5 Random Forest Classifier

4.5.1. Hyper paramter tuning (With One hot Encoding)

In [41]:
```
# ------------------------------
# default parameters
# sklearn.ensemble.RandomForestClassifier(n_estimators=10, criterion='gini', max_depth=None, min_samples_split=2,
# min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features='auto', max_leaf_nodes=None, min_impurity_decrease=0.
0,
# min_impurity_split=None, bootstrap=True, oob_score=False, n_jobs=1, random_state=None, verbose=0, warm_start=False,
# class_weight=None)

# Some of methods of RandomForestClassifier()
# fit(X, y, [sample_weight])    Fit the SVM model according to the given training data.
# predict(X)    Perform classification on samples in X.
# predict_proba (X)     Perform classification on samples in X.

# some of attributes of  RandomForestClassifier()
# feature_importances_ : array of shape = [n_features]
# The feature importances (the higher, the more important the feature).

# ------------------------------
# video link: https://www.appliedaicourse.com/course/applied-ai-course-online/lessons/random-forest-and-their-construc
tion-2/
# ------------------------------


# find more about CalibratedClassifierCV here at http://scikit-learn.org/stable/modules/generated/sklearn.calibration.
CalibratedClassifierCV.html
# ----------------------------
# default paramters
# sklearn.calibration.CalibratedClassifierCV(base_estimator=None, method='sigmoid', cv=3)
#
# some of the methods of CalibratedClassifierCV()
# fit(X, y[, sample_weight])    Fit the calibrated model
# get_params([deep])    Get parameters for this estimator.
# predict(X)    Predict the target of new samples.
# predict_proba(X)      Posterior probabilities of classification
#-----------------------------------
# video link:
#-----------------------------------

alpha = [100,200,500,1000,2000]
max_depth = [5, 10]
cv_log_error_array = []
```

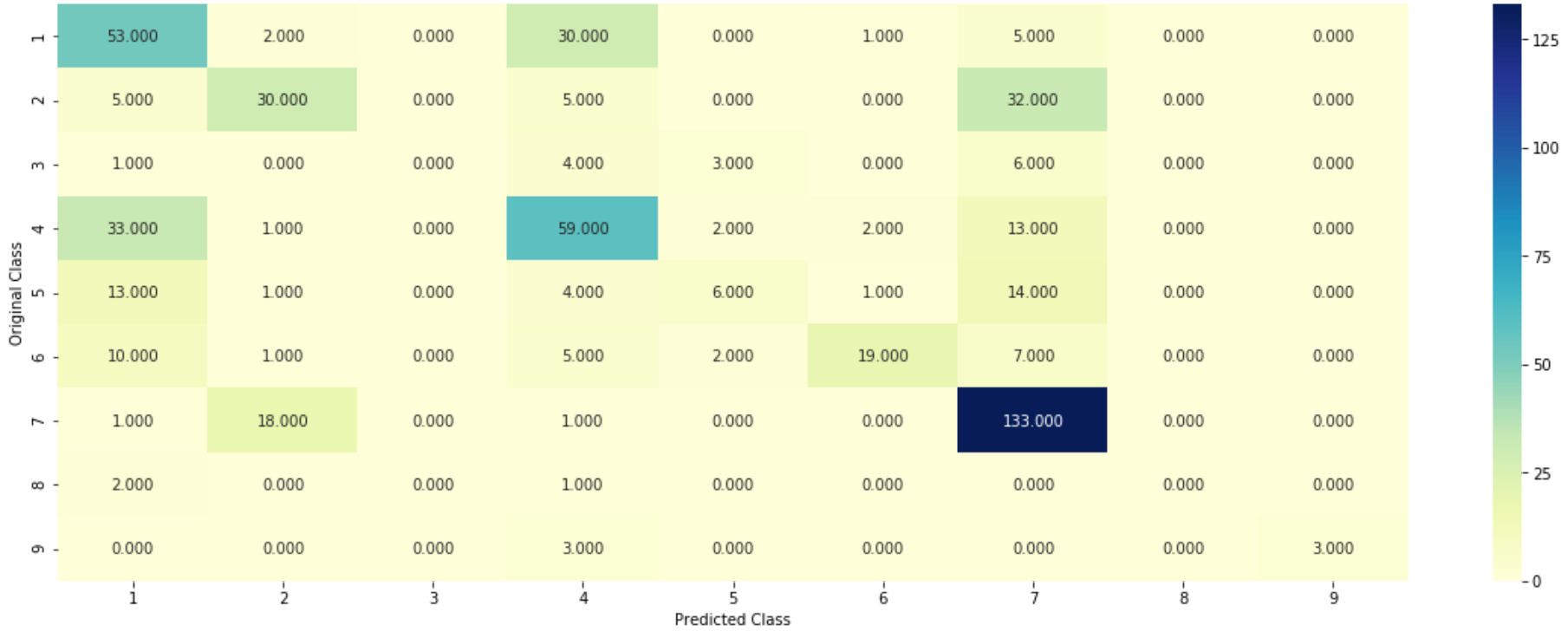
```
for i in alpha:
    for j in max_depth:
        print("for n_estimators =", i,"and max depth = ", j)
        clf = RandomForestClassifier(n_estimators=i, criterion='gini', max_depth=j, random_state=42, n_jobs=-1)
        clf.fit(train_x_onehotCoding, train_y)
        sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
        sig_clf.fit(train_x_onehotCoding, train_y)
        sig_clf_probs = sig_clf.predict_proba(cv_x_onehotCoding)
        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()
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],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 = RandomForestClassifier(n_estimators=alpha[int(best_alpha/2)], criterion='gini', max_depth=max_depth[int(best_alp
ha%2)], random_state=42, n_jobs=-1)
clf.fit(train_x_onehotCoding, train_y)
sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
sig_clf.fit(train_x_onehotCoding, train_y)

predict_y = sig_clf.predict_proba(train_x_onehotCoding)
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_onehotCoding)
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=1e-15))
predict_y = sig_clf.predict_proba(test_x_onehotCoding)
print('For values of best estimator = ', alpha[int(best_alpha/2)], "The test log loss is:",log_loss(y_test, predict_y,
 labels=clf.classes_, eps=1e-15))
```

```
for n_estimators = 100 and max depth =  5
Log Loss : 1.2058062909738223
for n_estimators = 100 and max depth =  10
Log Loss : 1.2515407256802213
for n_estimators = 200 and max depth =  5
Log Loss : 1.1919436833687997
for n_estimators = 200 and max depth =  10
Log Loss : 1.2364559940525628
for n_estimators = 500 and max depth =  5
Log Loss : 1.1929061303907207
for n_estimators = 500 and max depth =  10
Log Loss : 1.231924935559586
for n_estimators = 1000 and max depth =  5
Log Loss : 1.1909951282137456
for n_estimators = 1000 and max depth =  10
Log Loss : 1.2253146194839353
for n_estimators = 2000 and max depth =  5
Log Loss : 1.1901255319611563
for n_estimators = 2000 and max depth =  10
Log Loss : 1.2224376301280562
For values of best estimator =  2000 The train log loss is: 0.8769144330158822
For values of best estimator =  2000 The cross validation log loss is: 1.1901255319611566
For values of best estimator =  2000 The test log loss is: 1.1516954354004543
```

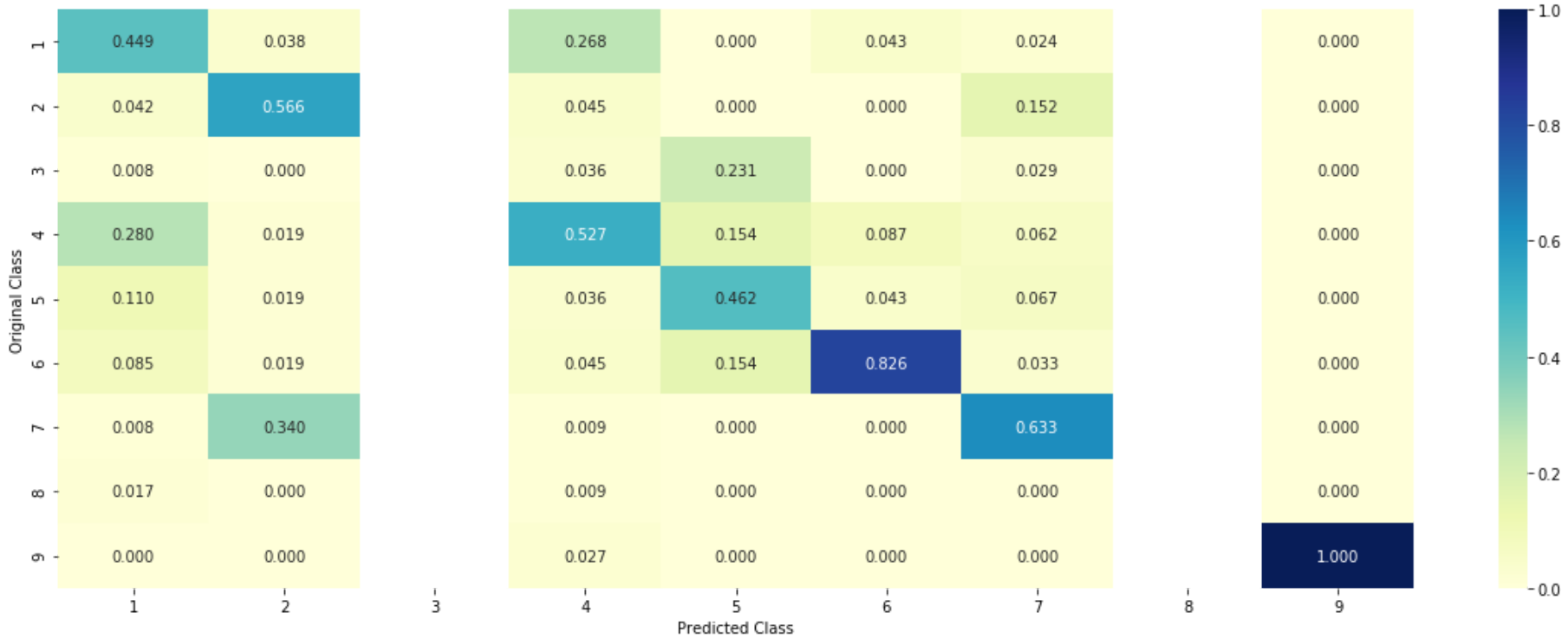
4.5.2. Testing model with best hyper parameters (One Hot Encoding)

In [42]:
```
# ------------------------------
# default parameters
# sklearn.ensemble.RandomForestClassifier(n_estimators=10, criterion='gini', max_depth=None, min_samples_split=2,
# min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features='auto', max_leaf_nodes=None, min_impurity_decrease=0.
0,
# min_impurity_split=None, bootstrap=True, oob_score=False, n_jobs=1, random_state=None, verbose=0, warm_start=False,
# class_weight=None)

# Some of methods of RandomForestClassifier()
# fit(X, y, [sample_weight])    Fit the SVM model according to the given training data.
# predict(X)    Perform classification on samples in X.
# predict_proba (X)     Perform classification on samples in X.

# some of attributes of  RandomForestClassifier()
# feature_importances_ : array of shape = [n_features]
# The feature importances (the higher, the more important the feature).

# ------------------------------
# video link: https://www.appliedaicourse.com/course/applied-ai-course-online/lessons/random-forest-and-their-construc
tion-2/
# ------------------------------

clf = RandomForestClassifier(n_estimators=alpha[int(best_alpha/2)], criterion='gini', max_depth=max_depth[int(best_alp
ha%2)], random_state=42, n_jobs=-1)
predict_and_plot_confusion_matrix(train_x_onehotCoding, train_y,cv_x_onehotCoding,cv_y, clf)
```

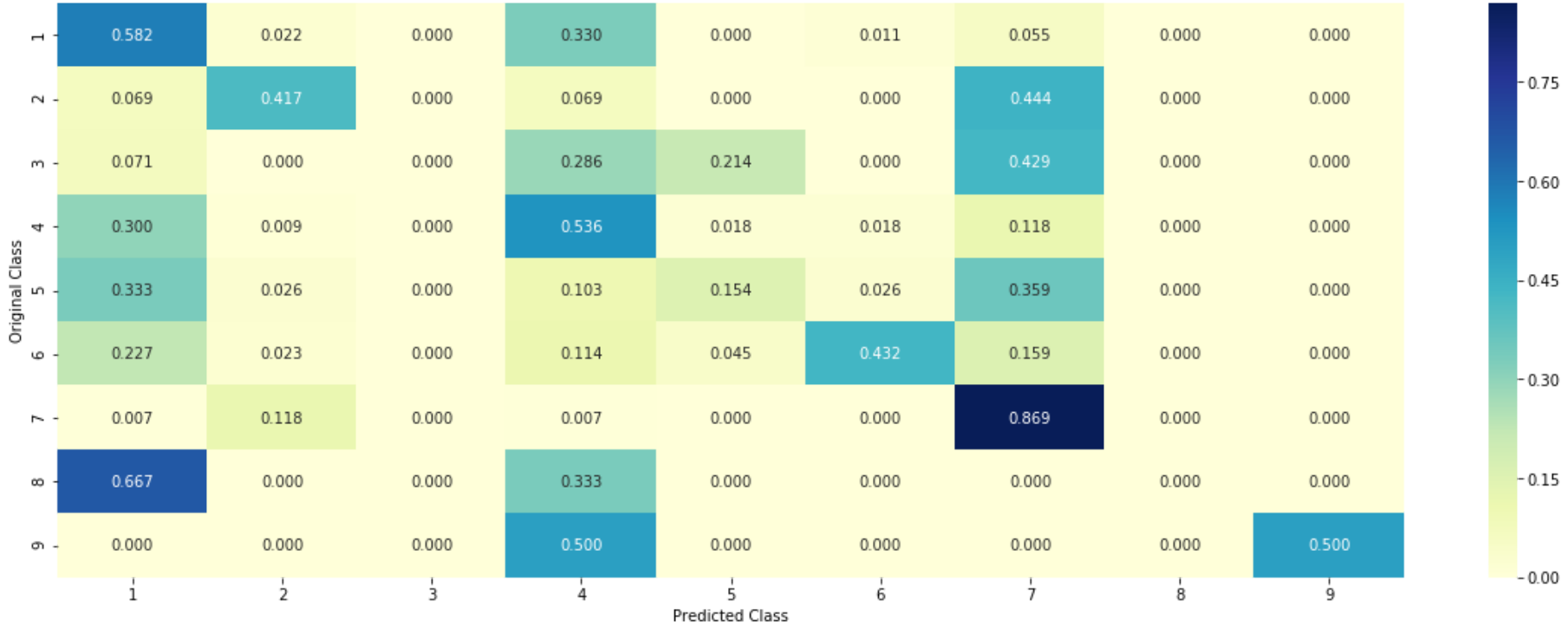
```
Log loss : 1.1901255319611563
Number of mis-classified points : 0.43045112781954886
-------------------- Confusion matrix --------------------
```




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



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



4.5.3. Feature Importance

4.5.3.1. Correctly Classified point

In [43]:
```
# test_point_index = 10
clf = RandomForestClassifier(n_estimators=alpha[int(best_alpha/2)], criterion='gini', max_depth=max_depth[int(best_alp
ha%2)], random_state=42, n_jobs=-1)
clf.fit(train_x_onehotCoding, train_y)
sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
sig_clf.fit(train_x_onehotCoding, train_y)

test_point_index = 1
no_feature = 100
predicted_cls = sig_clf.predict(test_x_onehotCoding[test_point_index])
print("Predicted Class :", predicted_cls[0])
print("Predicted Class Probabilities:", np.round(sig_clf.predict_proba(test_x_onehotCoding[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_index],test_df['Gene'].iloc[test_point_inde
x],test_df['Variation'].iloc[test_point_index], no_feature)
```

```
Predicted Class : 7
Predicted Class Probabilities: [[0.0402 0.243  0.0169 0.0266 0.0372 0.0378 0.5922 0.0049 0.0011]]
Actual Class : 7
--------------------------------------------------
76 Text feature [02] present in test data point [True]
Out of the top  100  features  1 are present in query point
```

4.5.3.2. Inorrectly Classified point

In [44]:
```
test_point_index = 100
no_feature = 100
predicted_cls = sig_clf.predict(test_x_onehotCoding[test_point_index])
print("Predicted Class :", predicted_cls[0])
print("Predicted Class Probabilities:", np.round(sig_clf.predict_proba(test_x_onehotCoding[test_point_index]),4))
print("Actuall 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_index],test_df['Gene'].iloc[test_point_inde
x],test_df['Variation'].iloc[test_point_index], no_feature)
```

```
Predicted Class : 7
Predicted Class Probabilities: [[0.0126 0.3496 0.0134 0.0087 0.0284 0.0299 0.5538 0.0028 0.0007]]
Actuall Class : 7
--------------------------------------------------
76 Text feature [02] present in test data point [True]
Out of the top  100  features  1 are present in query point
```

4.5.3. Hyper paramter tuning (With Response Coding)

In [45]:
```
# ------------------------------
# default parameters
# sklearn.ensemble.RandomForestClassifier(n_estimators=10, criterion='gini', max_depth=None, min_samples_split=2,
# min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features='auto', max_leaf_nodes=None, min_impurity_decrease=0.
0,
# min_impurity_split=None, bootstrap=True, oob_score=False, n_jobs=1, random_state=None, verbose=0, warm_start=False,
# class_weight=None)

# Some of methods of RandomForestClassifier()
# fit(X, y, [sample_weight])    Fit the SVM model according to the given training data.
# predict(X)    Perform classification on samples in X.
# predict_proba (X)     Perform classification on samples in X.

# some of attributes of  RandomForestClassifier()
# feature_importances_ : array of shape = [n_features]
# The feature importances (the higher, the more important the feature).

# ------------------------------
# video link: https://www.appliedaicourse.com/course/applied-ai-course-online/lessons/random-forest-and-their-construc
tion-2/
# ------------------------------


# find more about CalibratedClassifierCV here at http://scikit-learn.org/stable/modules/generated/sklearn.calibration.
CalibratedClassifierCV.html
# ----------------------------
# default paramters
# sklearn.calibration.CalibratedClassifierCV(base_estimator=None, method='sigmoid', cv=3)
#
# some of the methods of CalibratedClassifierCV()
# fit(X, y[, sample_weight])    Fit the calibrated model
# get_params([deep])    Get parameters for this estimator.
# predict(X)    Predict the target of new samples.
# predict_proba(X)      Posterior probabilities of classification
#-------------------------------------
# video link:
#-------------------------------------

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_depth=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.classes_, 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],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 = RandomForestClassifier(n_estimators=alpha[int(best_alpha/4)], criterion='gini', max_depth=max_depth[int(best_alp
ha%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, l
abels=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 validation log loss is:",log_loss(y_cv, pred
ict_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 loss is:",log_loss(y_test, predict_y, lab
els=clf.classes_, eps=1e-15))
```

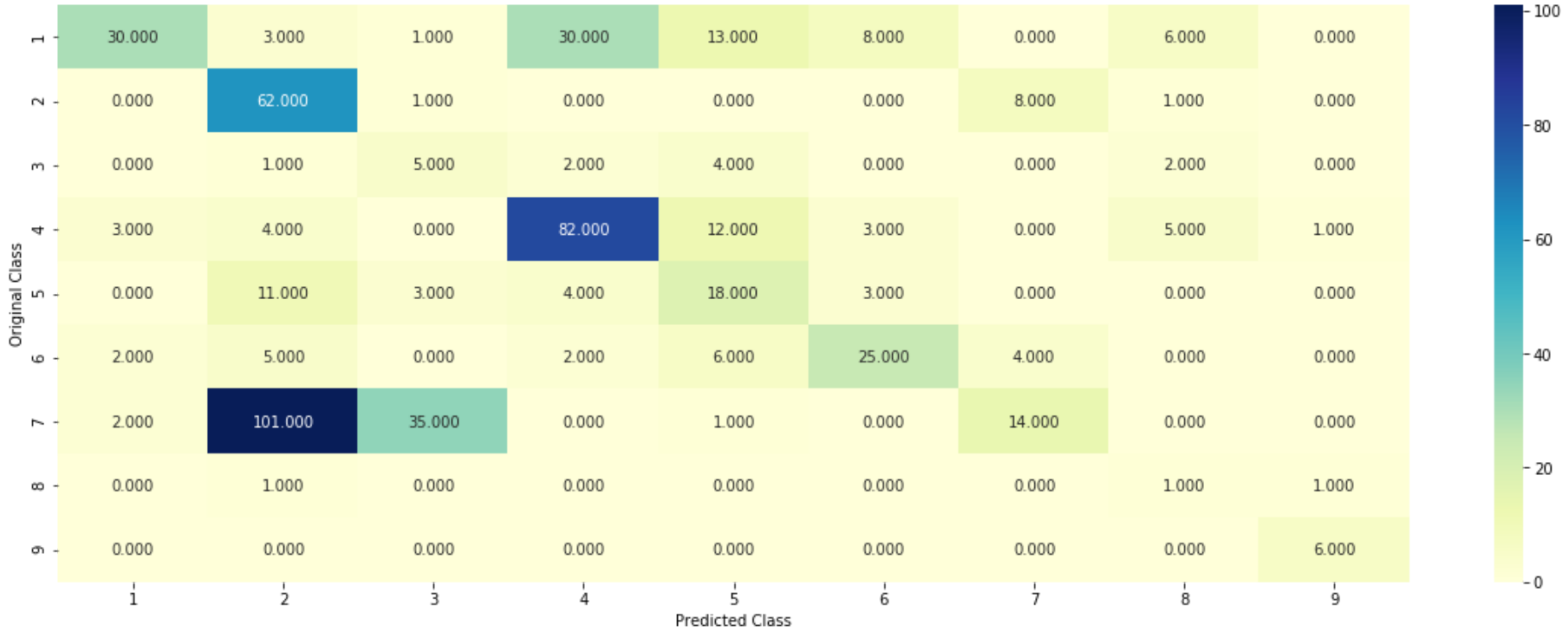
```
for n_estimators = 10 and max depth =  2
Log Loss : 2.121168855735514
for n_estimators = 10 and max depth =  3
Log Loss : 1.7006781420159711
for n_estimators = 10 and max depth =  5
Log Loss : 1.4056412284730369
for n_estimators = 10 and max depth =  10
Log Loss : 1.846864985390235
for n_estimators = 50 and max depth =  2
Log Loss : 1.7249999968157448
for n_estimators = 50 and max depth =  3
Log Loss : 1.4181305989462831
for n_estimators = 50 and max depth =  5
Log Loss : 1.3799630854936158
for n_estimators = 50 and max depth =  10
Log Loss : 1.6843258779008994
for n_estimators = 100 and max depth =  2
Log Loss : 1.5298875588579697
for n_estimators = 100 and max depth =  3
Log Loss : 1.493488826646144
for n_estimators = 100 and max depth =  5
Log Loss : 1.3828067200183298
for n_estimators = 100 and max depth =  10
Log Loss : 1.6708934796553525
for n_estimators = 200 and max depth =  2
Log Loss : 1.5920252308990268
for n_estimators = 200 and max depth =  3
Log Loss : 1.5004105275348263
for n_estimators = 200 and max depth =  5
Log Loss : 1.429187775610955
for n_estimators = 200 and max depth =  10
Log Loss : 1.6931047930156682
for n_estimators = 500 and max depth =  2
Log Loss : 1.7157244885551617
for n_estimators = 500 and max depth =  3
Log Loss : 1.5684045326602118
for n_estimators = 500 and max depth =  5
Log Loss : 1.4044330656835644
for n_estimators = 500 and max depth =  10
Log Loss : 1.7490076493750095
for n_estimators = 1000 and max depth =  2
```

```
Log Loss : 1.6706634299746126
for n_estimators = 1000 and max depth =  3
Log Loss : 1.5690344903579458
for n_estimators = 1000 and max depth =  5
Log Loss : 1.40981010845083
for n_estimators = 1000 and max depth =  10
Log Loss : 1.7223475342655636
For values of best alpha =  50 The train log loss is: 0.05278144175374067
For values of best alpha =  50 The cross validation log loss is: 1.3799630854936158
For values of best alpha =  50 The test log loss is: 1.3957894129939916
```

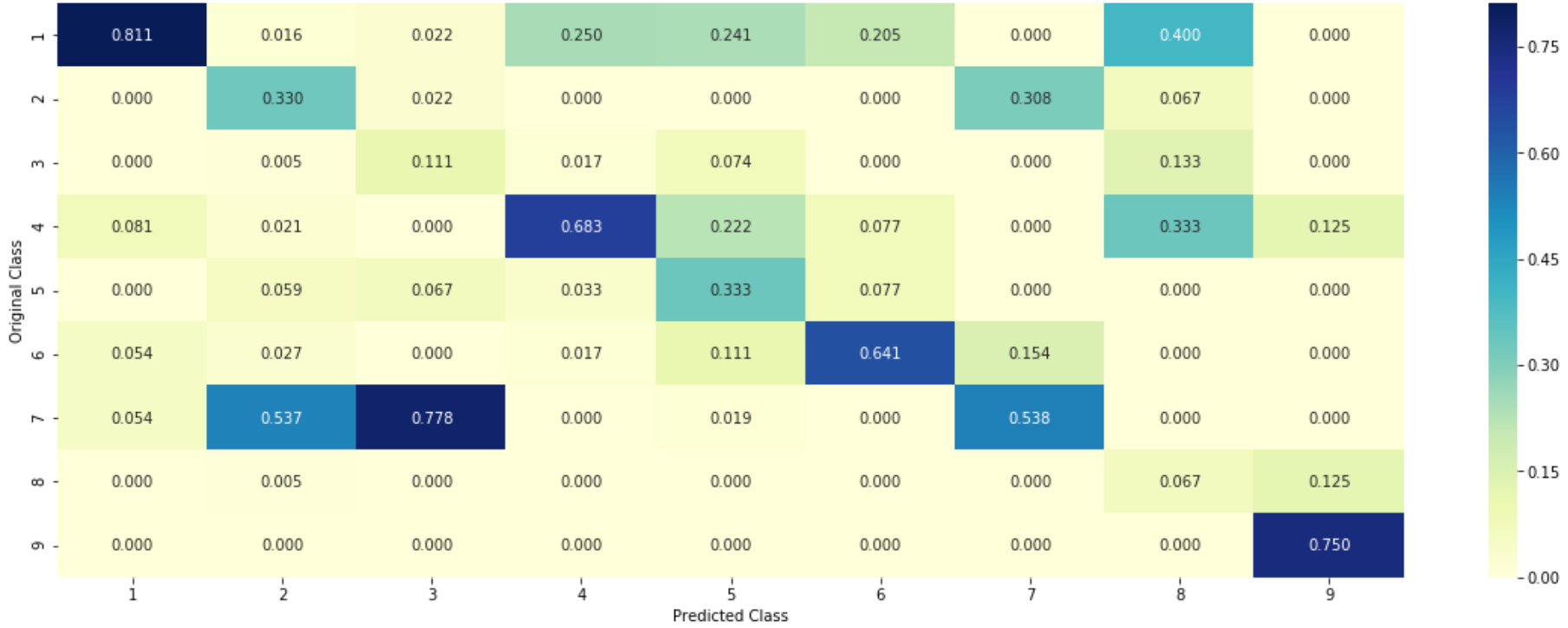
4.5.4. Testing model with best hyper parameters (Response Coding)

In [46]:
```
# ------------------------------
# default parameters
# sklearn.ensemble.RandomForestClassifier(n_estimators=10, criterion='gini', max_depth=None, min_samples_split=2,
# min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features='auto', max_leaf_nodes=None, min_impurity_decrease=0.
0,
# min_impurity_split=None, bootstrap=True, oob_score=False, n_jobs=1, random_state=None, verbose=0, warm_start=False,
# class_weight=None)

# Some of methods of RandomForestClassifier()
# fit(X, y, [sample_weight])    Fit the SVM model according to the given training data.
# predict(X)    Perform classification on samples in X.
# predict_proba (X)     Perform classification on samples in X.

# some of attributes of  RandomForestClassifier()
# feature_importances_ : array of shape = [n_features]
# The feature importances (the higher, the more important the feature).

# ------------------------------
# video link: https://www.appliedaicourse.com/course/applied-ai-course-online/lessons/random-forest-and-their-construc
tion-2/
# ------------------------------

clf = RandomForestClassifier(max_depth=max_depth[int(best_alpha%4)], n_estimators=alpha[int(best_alpha/4)], criterion
='gini', max_features='auto',random_state=42)
predict_and_plot_confusion_matrix(train_x_responseCoding, train_y,cv_x_responseCoding,cv_y, clf)
```

```
Log loss : 1.379963085493616
Number of mis-classified points : 0.543233082706767
-------------------- 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 [47]:
```
clf = RandomForestClassifier(n_estimators=alpha[int(best_alpha/4)], criterion='gini', max_depth=max_depth[int(best_alp
ha%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 = 1
no_feature = 27
predicted_cls = sig_clf.predict(test_x_responseCoding[test_point_index].reshape(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 : 7
Predicted Class Probabilities: [[0.0022 0.0027 0.0025 0.0031 0.0013 0.0019 0.9838 0.0011 0.0014]]
Actual Class : 7
--------------------------------------------------
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
Text is important feature
Gene is important feature
Text is important feature
Text is important feature
Variation is important feature
Gene is important feature
Gene is important feature
Gene is important feature
Variation is important feature
Text is important feature
Gene is important feature
Text is important feature
Variation is important feature
Text is important feature
Text is important feature
Gene 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 [48]:
```
# read more about SGDClassifier() at http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html
# ------------------------------
# default parameters
# SGDClassifier(loss='hinge', penalty='l2', alpha=0.0001, l1_ratio=0.15, fit_intercept=True, max_iter=None, tol=None,
# shuffle=True, verbose=0, epsilon=0.1, n_jobs=1, random_state=None, learning_rate='optimal', eta0=0.0, power_t=0.5,
# class_weight=None, warm_start=False, average=False, n_iter=None)

# some of methods
# fit(X, y[, coef_init, intercept_init, ...])     Fit linear model with Stochastic Gradient Descent.
# predict(X)    Predict class labels for samples in X.

#-------------------------------
# video link: https://www.appliedaicourse.com/course/applied-ai-course-online/lessons/geometric-intuition-1/
#-------------------------------


# read more about support vector machines with linear kernals here http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html
# ------------------------------
# default parameters
# SVC(C=1.0, kernel='rbf', degree=3, gamma='auto', coef0=0.0, shrinking=True, probability=False, tol=0.001,
# cache_size=200, class_weight=None, verbose=False, max_iter=-1, decision_function_shape='ovr', random_state=None)

# Some of methods of SVM()
# fit(X, y, [sample_weight])    Fit the SVM model according to the given training data.
# predict(X)    Perform classification on samples in X.
# ------------------------------
# video link: https://www.appliedaicourse.com/course/applied-ai-course-online/lessons/mathematical-derivation-copy-8/
# ------------------------------


# read more about support vector machines with linear kernals here http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html
# ------------------------------
# default parameters
# sklearn.ensemble.RandomForestClassifier(n_estimators=10, criterion='gini', max_depth=None, min_samples_split=2,
# min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features='auto', max_leaf_nodes=None, min_impurity_decrease=0.0,
# min_impurity_split=None, bootstrap=True, oob_score=False, n_jobs=1, random_state=None, verbose=0, warm_start=False,
# class_weight=None)
```

```
# Some of methods of RandomForestClassifier()
# fit(X, y, [sample_weight])    Fit the SVM model according to the given training data.
# predict(X)    Perform classification on samples in X.
# predict_proba (X)     Perform classification on samples in X.

# some of attributes of  RandomForestClassifier()
# feature_importances_ : array of shape = [n_features]
# The feature importances (the higher, the more important the feature).

# --------------------------------
# video link: https://www.appliedaicourse.com/course/applied-ai-course-online/lessons/random-forest-and-their-construc
tion-2/
# --------------------------------


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

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


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

sig_clf1.fit(train_x_onehotCoding, train_y)
print("Logistic Regression :  Log Loss: %0.2f" % (log_loss(cv_y, sig_clf1.predict_proba(cv_x_onehotCoding))))
sig_clf2.fit(train_x_onehotCoding, train_y)
print("Support vector machines : Log Loss: %0.2f" % (log_loss(cv_y, sig_clf2.predict_proba(cv_x_onehotCoding))))
sig_clf3.fit(train_x_onehotCoding, train_y)
print("Naive Bayes : Log Loss: %0.2f" % (log_loss(cv_y, sig_clf3.predict_proba(cv_x_onehotCoding))))
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_onehotCoding, train_y)
```

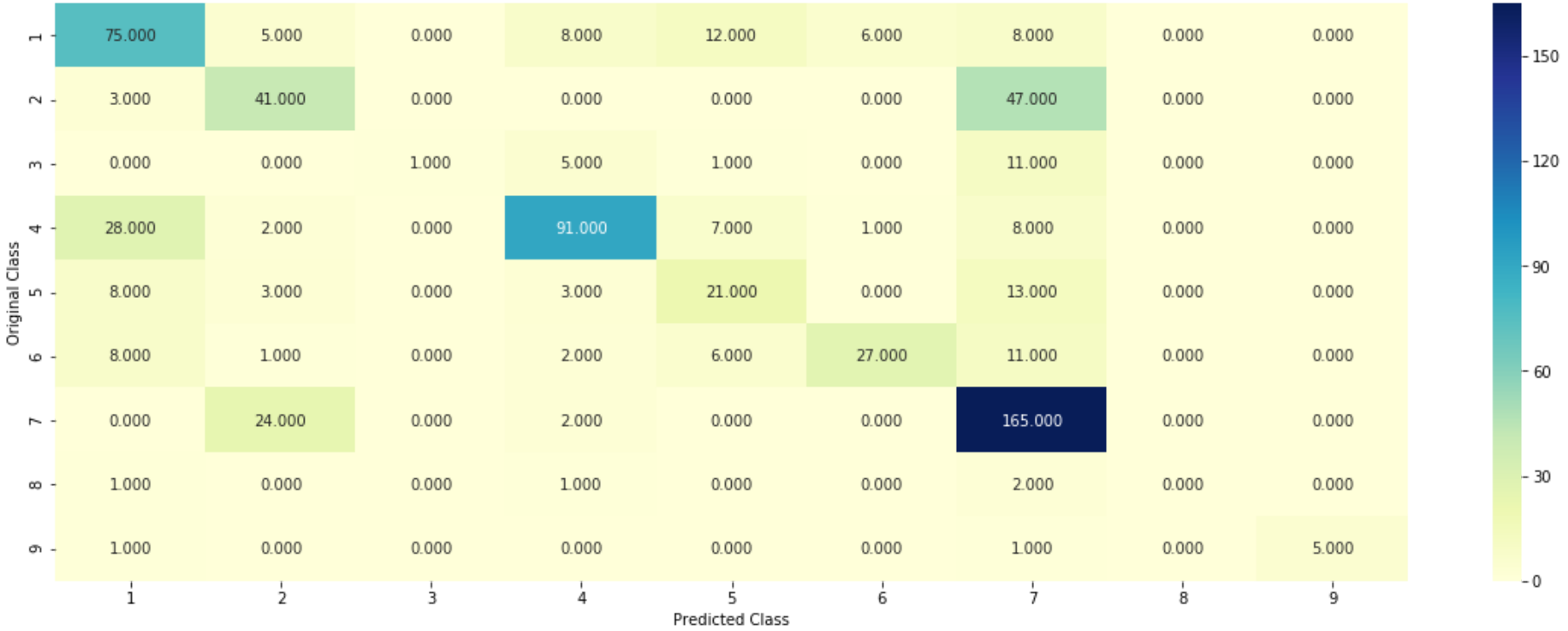
```
    print("Stacking Classifer : for the value of alpha: %f Log Loss: %0.3f" % (i, log_loss(cv_y, sclf.predict_proba(cv
_x_onehotCoding))))
    log_error =log_loss(cv_y, sclf.predict_proba(cv_x_onehotCoding))
    if best_alpha > log_error:
        best_alpha = log_error
```

```
Logistic Regression :  Log Loss: 1.01
Support vector machines : Log Loss: 1.82
Naive Bayes : Log Loss: 1.17
---------------------------------------------------
Stacking Classifer : for the value of alpha: 0.000100 Log Loss: 2.177
Stacking Classifer : for the value of alpha: 0.001000 Log Loss: 2.029
Stacking Classifer : for the value of alpha: 0.010000 Log Loss: 1.480
Stacking Classifer : for the value of alpha: 0.100000 Log Loss: 1.113
Stacking Classifer : for the value of alpha: 1.000000 Log Loss: 1.265
Stacking Classifer : for the value of alpha: 10.000000 Log Loss: 1.651
```

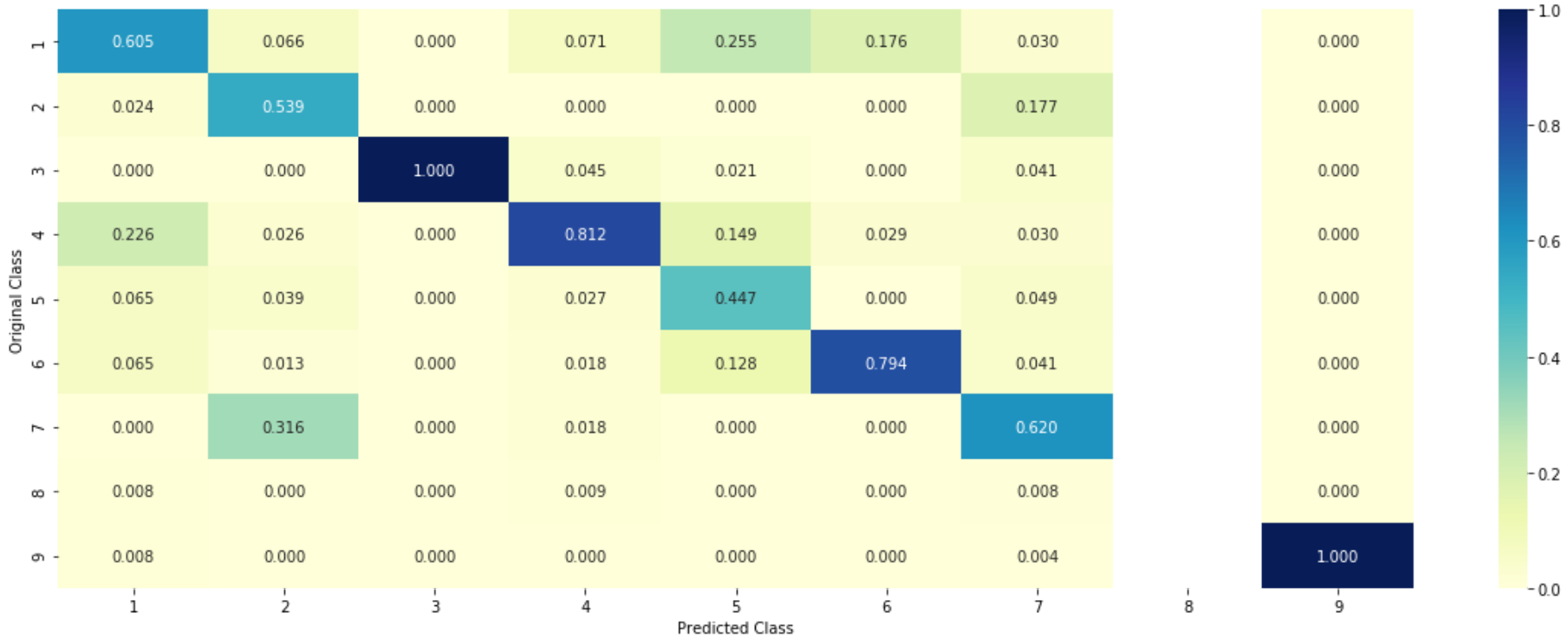
4.7.2 testing the model with the best hyper parameters

In [49]:
```
lr = LogisticRegression(C=0.1)
sclf = StackingClassifier(classifiers=[sig_clf1, sig_clf2, sig_clf3], meta_classifier=lr, use_probas=True)
sclf.fit(train_x_onehotCoding, train_y)

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

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

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

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

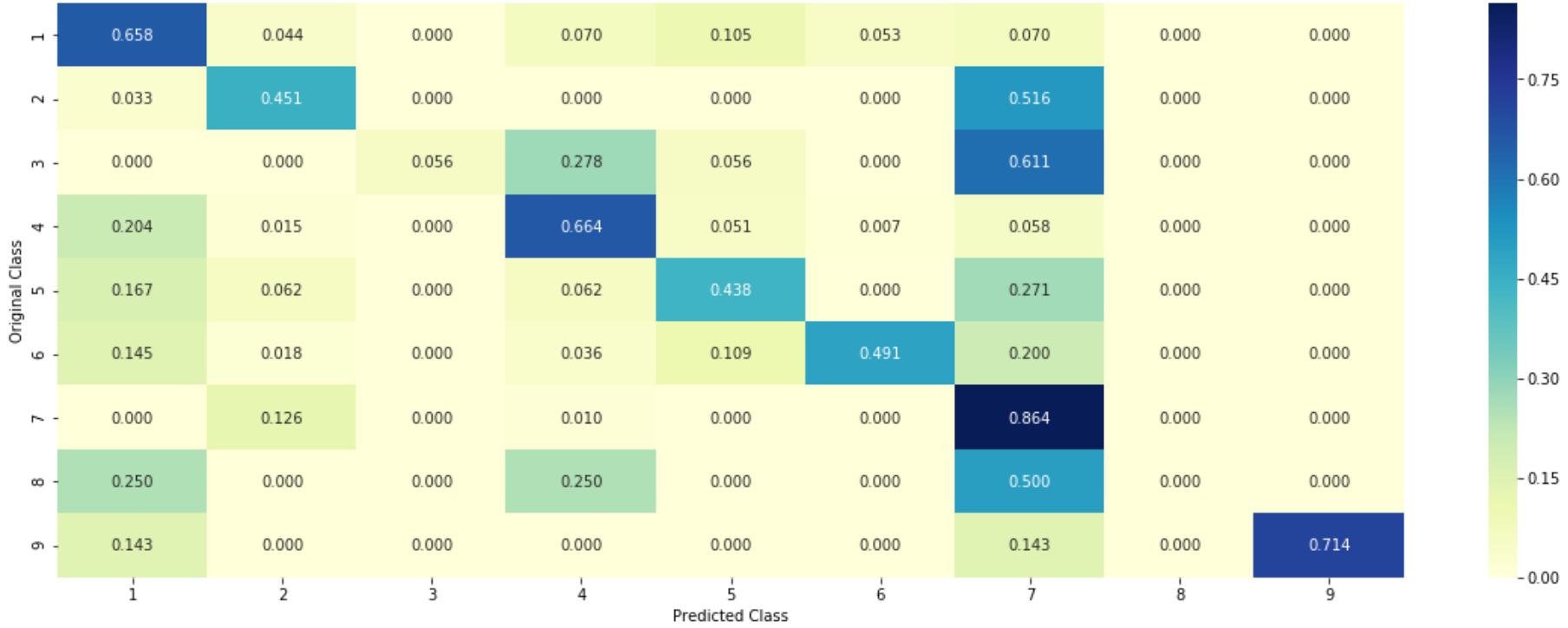
```
Log loss (train) on the stacking classifier : 0.5792511661131973
Log loss (CV) on the stacking classifier : 1.1125648572313287
Log loss (test) on the stacking classifier : 1.1409032736214493
Number of missclassified point : 0.3593984962406015
-------------------- Confusion matrix --------------------
```




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



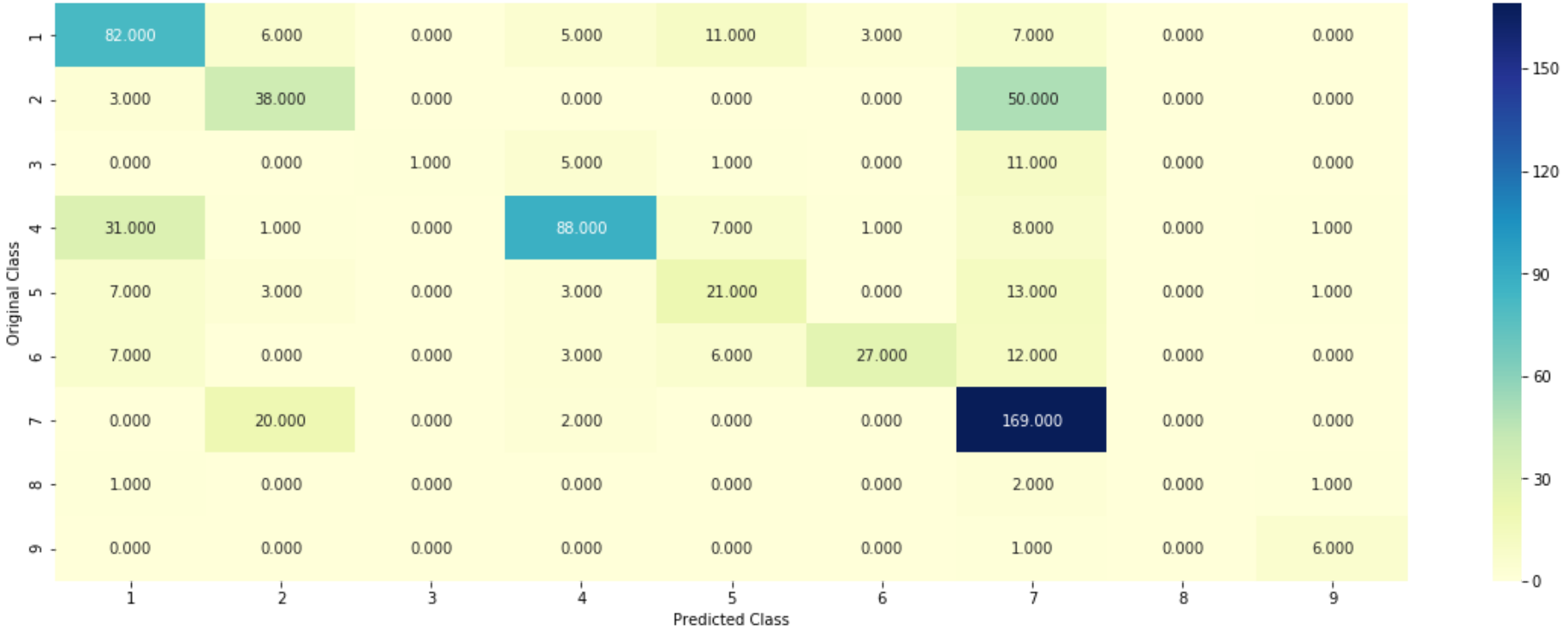
---------- Recall matrix (Row sum=1) ----------



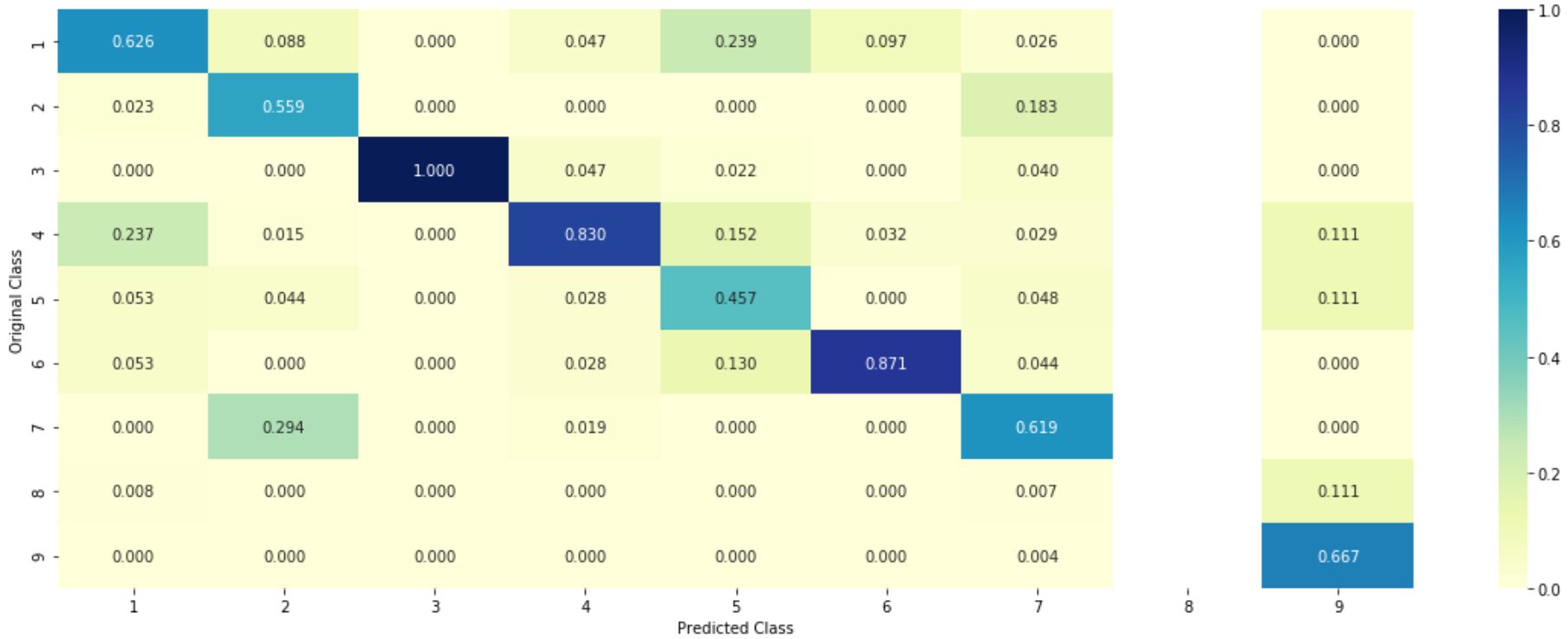
4.7.3 Maximum Voting classifier

In [50]:
```
#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_onehotCoding, train_y)
print("Log loss (train) on the VotingClassifier :", log_loss(train_y, vclf.predict_proba(train_x_onehotCoding)))
print("Log loss (CV) on the VotingClassifier :", log_loss(cv_y, vclf.predict_proba(cv_x_onehotCoding)))
print("Log loss (test) on the VotingClassifier :", log_loss(test_y, vclf.predict_proba(test_x_onehotCoding)))
print("Number of missclassified point :", np.count_nonzero((vclf.predict(test_x_onehotCoding)- test_y))/test_y.shape[0
])
plot_confusion_matrix(test_y=test_y, predict_y=vclf.predict(test_x_onehotCoding))
```

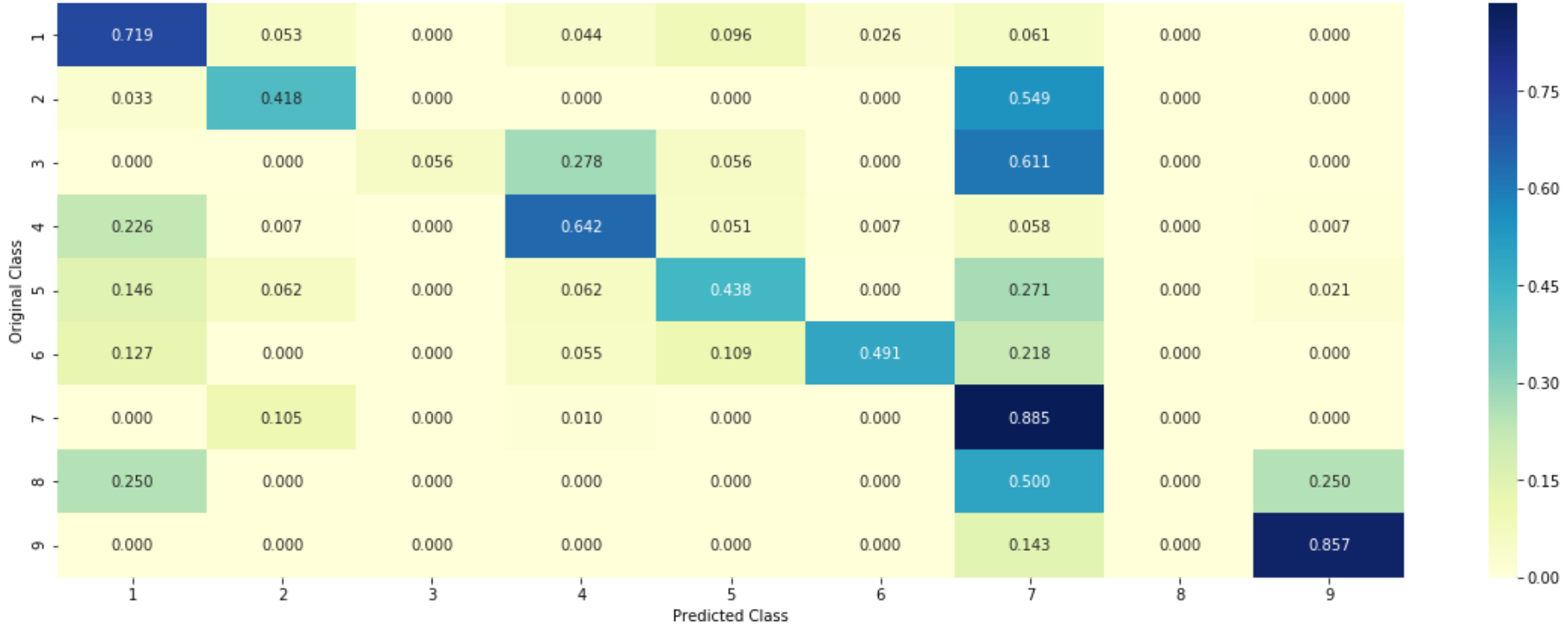
```
Log loss (train) on the VotingClassifier : 0.8486513343697619
Log loss (CV) on the VotingClassifier : 1.1652193874731789
Log loss (test) on the VotingClassifier : 1.162165126025522
Number of missclassified point : 0.35037593984962406
-------------------- Confusion matrix --------------------
```




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




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



Analysis of Personalized Cancer Diagnosis Case Study

Model	Best Hyperparameters	Train error	CV error	Test error	Misclassified Points	Log - Loss
Naive Bayes + one-hot Encoding	alpha=0.001	0.563	1.244	1.169	38.9%	1.244
KNN + Response coding	K=5	0.475	1.03	1.058	38.3%	1.103
Logistic Regression unigram+bigram + class balance	alpha= 0.01	0.876	1.162	1.195	38.7%	1.16
Logistic Regression unigram+bigram + without class balance	alpha= 0.01	0.888	1.182	1.222	40.2%	1.18
Logistic Regression tfidf vectorizer(with 2000 max words)+FE(square root)	alpha= 0.001	0.612	1.032	0.946	33.9%	0.946
Linear SVM + one-hot encoding	alpha=0.0001	0.568	1.025	1.006	32.3%	1.025
Random Forest + one-hot encoding	best-estimators=2000	0.87	1.190	1.151	43.0%	1.190
Random Forest + one-hot encoding	alpha=50	0.052	1.379	1.395	54.3%	1.379
Stack Classifier(LR+LrSVM+NB)	alpha=0.10	0.579	1.112	1.140	35.9%	1.113
Maximum voting Classifier(LR+SVM+RF)	alpha=0.10	0.84	1.165	1.162	35.0%	1.113

Observation

- We did onehot encoding featurization for TfidfVectorizer() with 2000 features
- We accomplished onehot encoding for Logistic Regression models with un-igram and bi-gram
- On Logistic Regression model we did tfidf vectorizer with 2000 features and did Feature Engineering such as square root which have given log-loss 0.94. Using square root feature engineering we find the best model among all models.
- Second good model is Liner Support Vector Machine

---xxx---