Personalized cancer diagnosis

1. Business Problem

1.1. Description

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

Data: Memorial Sloan Kettering Cancer Center (MSKCC)

Download training_variants.zip and training_text.zip from Kaggle.

Context:

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

Problem statement :

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

1.2. Source/Useful Links

Some articles and reference blogs about the problem statement

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

1.3. Real-world/Business objectives and constraints.

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

2. Machine Learning Problem Formulation

2.1. Data

2.1.1. Data Overview

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

2.1.2. Example Data Point

training_variants

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

training_text

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

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

2.2.1. Type of Machine Learning Problem

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

2.2.2. Performance Metric

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

Metric(s):

- Multi class log-loss
- Confusion matrix

2.2.3. Machine Learing Objectives and Constraints

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

Constraints:

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

2.3. Train, CV and Test Datasets

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

3. Exploratory Data Analysis

In [236]:

```
import pandas as pd
import matplotlib.pyplot as plt
import re
import time
import warnings
import numpy as np
from nltk.corpus import stopwords
from sklearn.decomposition import TruncatedSVD
from sklearn.preprocessing import normalize
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.manifold import TSNE
import seaborn as sns
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import confusion_matrix
from sklearn.metrics.classification import accuracy_score, log_loss
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import SGDClassifier
from imblearn.over_sampling import SMOTE
from collections import Counter
from scipy.sparse import hstack
from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import SVC
from sklearn.model_selection import StratifiedKFold
from collections import Counter, defaultdict
from sklearn.calibration import CalibratedClassifierCV
from sklearn.naive_bayes import MultinomialNB
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
import math
from sklearn.metrics import normalized_mutual_info_score
from sklearn.ensemble import RandomForestClassifier
warnings.filterwarnings("ignore")

from mlxtend.classifier import StackingClassifier

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

3.1. Reading Data

3.1.1. Reading Gene and Variation Data

In [2]:

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

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

Out[2]:

	ID	Gene	Variation	Class
0	0	FAM58A	Truncating Mutations	1

	ID	Gene	Variation	Class
1	1	CBL	W802*	2
2	2	CBL	Q249E	2
3	3	CBL	N454D	3
4	4	CBL	L399V	4

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

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

3.1.2. Reading Text Data

In [3]:

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

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

Out[3]:

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

3.1.3. Preprocessing of text

In [4]:

```
# 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 [5]:

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

```
there is no text description for id: 1109
there is no text description for id: 1277
there is no text description for id: 1407
there is no text description for id: 1639
there is no text description for id: 2755
Time took for preprocessing the text : 182.34166489999998 seconds
```

In [6]:

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

Out[6]:

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

In [7]:

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

Out[7]:

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

In [8]:

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

In [9]:

```
result[result['ID']==1109]
```

Out[9]:

	ID	Gene	Variation	Class	TEXT

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

3.1.4. Test, Train and Cross Validation Split

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

In [10]:

```
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 [11]:

```
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])
```

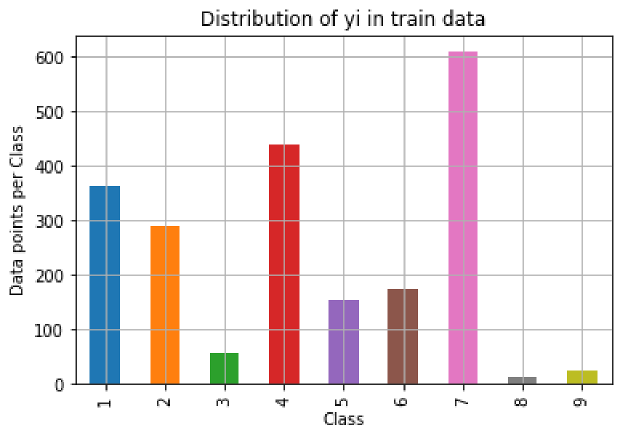
```
Number of data points in train data: 2124
Number of data points in test data: 665
Number of data points in cross validation data: 532
```

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

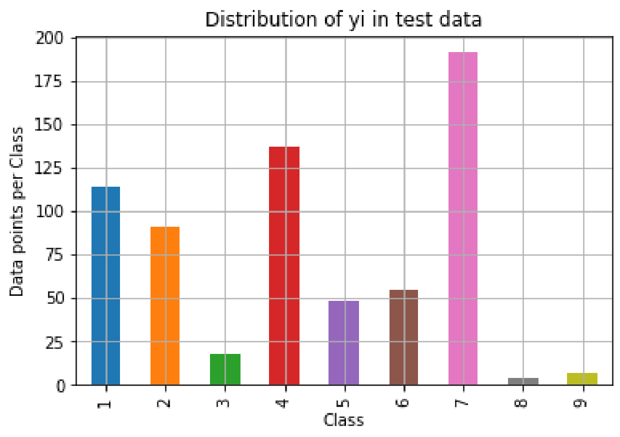
In [12]:

```
# 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 = 'rgbkymc'
train_class_distribution.plot(kind='bar')
plt.xlabel('Class')
plt.ylabel('Data points per Class')
plt.title('Distribution of yi in train data')
plt.grid()
plt.show()

# ref: argsort https://docs.scipy.org/doc/numpy/reference/generated/numpy.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.ro
und((train_class_distribution.values[i]/train_df.shape[0]*100), 3), '%)')


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

# ref: argsort https://docs.scipy.org/doc/numpy/reference/generated/numpy.argsort.html
# -(train_class_distribution.values): the minus sign will give us in decreasing order
```

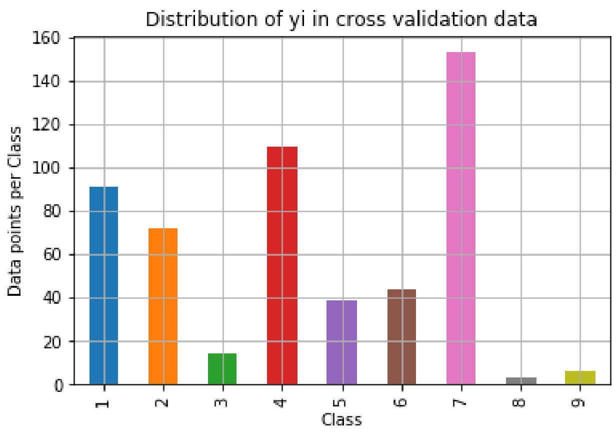
```
sorted_yi = np.argsort(-test_class_distribution.values)
for i in sorted_yi:
    print('Number of data points in class', i+1, ':',test_class_distribution.values[i], '(', np.rou
nd((test_class_distribution.values[i]/test_df.shape[0]*100), 3), '%)')

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

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



```
Number of data points in class 7 : 609 ( 28.672 %)
Number of data points in class 4 : 439 ( 20.669 %)
Number of data points in class 1 : 363 ( 17.09 %)
Number of data points in class 2 : 289 ( 13.606 %)
Number of data points in class 6 : 176 ( 8.286 %)
Number of data points in class 5 : 155 ( 7.298 %)
Number of data points in class 3 : 57 ( 2.684 %)
Number of data points in class 9 : 24 ( 1.13 %)
Number of data points in class 8 : 12 ( 0.565 %)
--------------------------------------------------------------------------------
```



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



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

FRom above plots you can see that the split for train , test and cross validation is equal

3.2 Prediction using a 'Random' Model

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

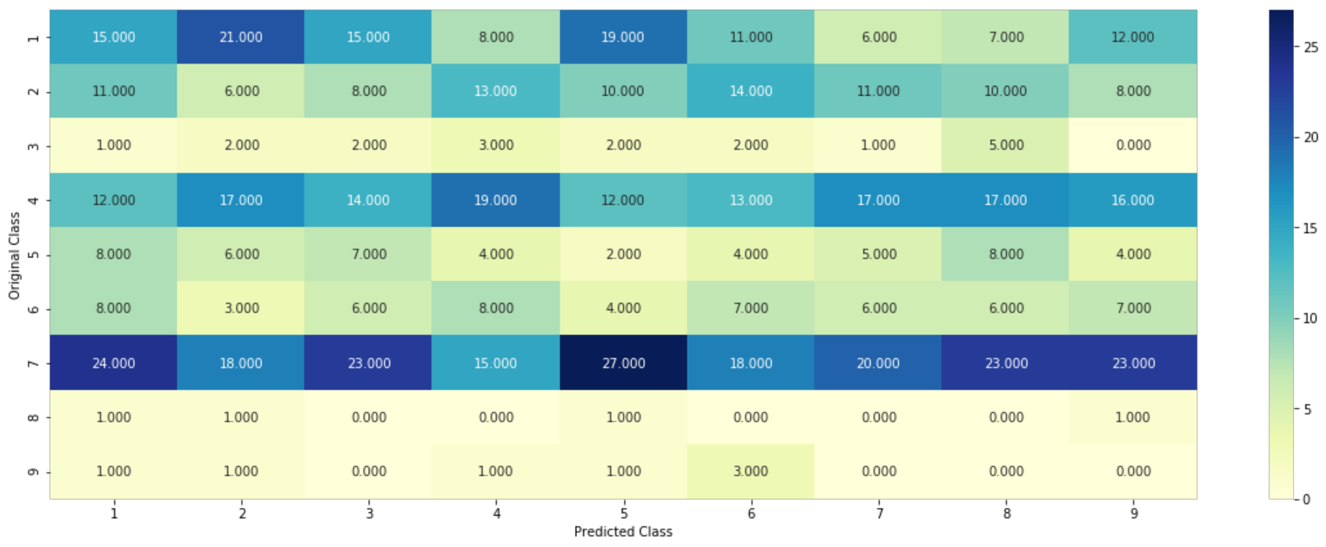
In [13]:

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

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

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

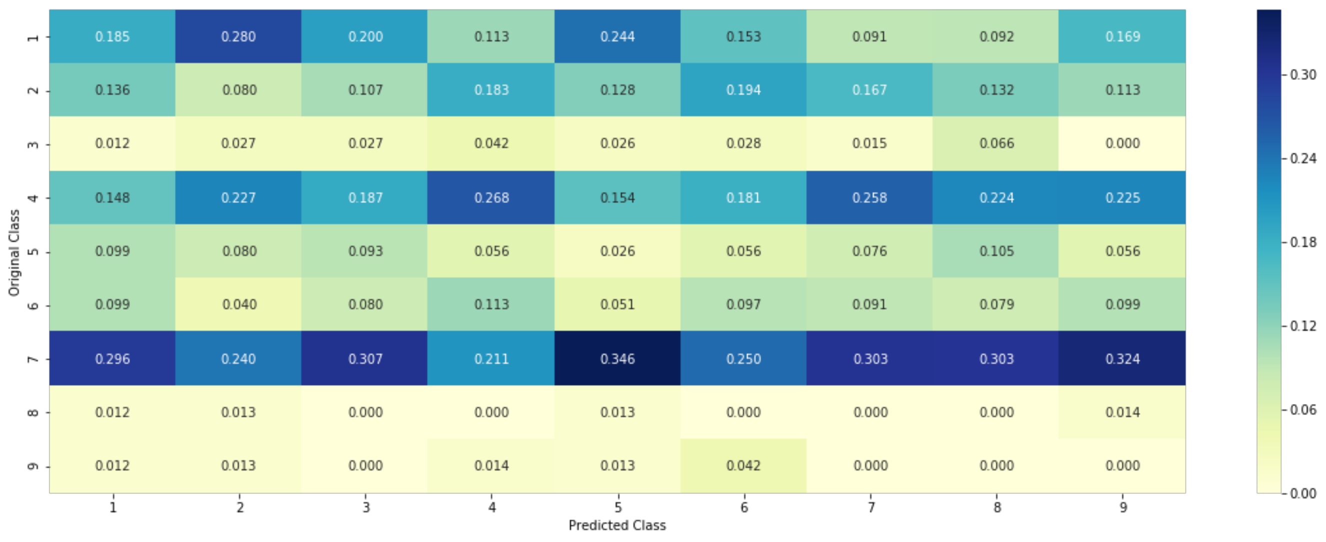
In [14]:

```
# 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]

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


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

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

```
Log loss on Cross Validation Data using Random Model 2.524319437435131
Log loss on Test Data using Random Model 2.4678509369458714
-------------------- Confusion matrix --------------------
```

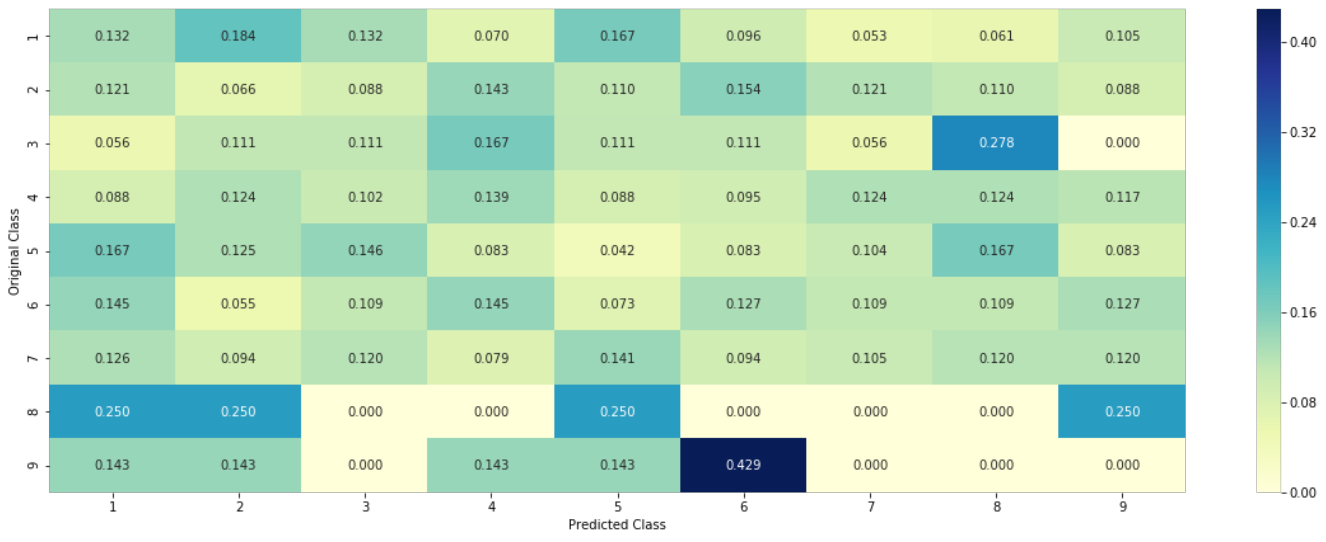


```
-------------------- Precision matrix (Columm Sum=1) --------------------
```



	1	2	3	4	5	6	7	8	9
1	0.185	0.280	0.200	0.113	0.244	0.153	0.091	0.092	0.169
2	0.136	0.080	0.107	0.183	0.128	0.194	0.167	0.132	0.113
3	0.012	0.027	0.027	0.042	0.026	0.028	0.015	0.066	0.000
4	0.148	0.227	0.187	0.268	0.154	0.181	0.258	0.224	0.225
5	0.099	0.080	0.093	0.056	0.026	0.056	0.076	0.105	0.056
6	0.099	0.040	0.080	0.113	0.051	0.097	0.091	0.079	0.099
7	0.296	0.240	0.307	0.211	0.346	0.250	0.303	0.303	0.324
8	0.012	0.013	0.000	0.000	0.013	0.000	0.000	0.000	0.014
9	0.012	0.013	0.000	0.014	0.013	0.042	0.000	0.000	0.000

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



	1	2	3	4	5	6	7	8	9
1	0.132	0.184	0.132	0.070	0.167	0.096	0.053	0.061	0.105
2	0.121	0.066	0.088	0.143	0.110	0.154	0.121	0.110	0.088
3	0.056	0.111	0.111	0.167	0.111	0.111	0.056	0.278	0.000
4	0.088	0.124	0.102	0.139	0.088	0.095	0.124	0.124	0.117
5	0.167	0.125	0.146	0.083	0.042	0.083	0.104	0.167	0.083
6	0.145	0.055	0.109	0.145	0.073	0.127	0.109	0.109	0.127
7	0.126	0.094	0.120	0.079	0.141	0.094	0.105	0.120	0.120
8	0.250	0.250	0.000	0.000	0.250	0.000	0.000	0.000	0.250
9	0.143	0.143	0.000	0.143	0.143	0.429	0.000	0.000	0.000

3.3 Univariate Analysis

In [15]:

```
# 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 / nu
mber of time it occurred in total data+90*alpha)
# gv_dict is like a look up table, for every gene it store a (1*9) 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 o
ccured in whole data
            vec.append((cls_cnt.shape[0] + alpha*10)/ (denominator + 90*alpha))

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

# Get Gene variation feature
def get_gv_feature(alpha, feature, df):
    # print(gv_dict)
    #     {'BRCA1': [0.20075757575757575, 0.03787878787878788, 0.068181818181818177,
0.13636363636363635, 0.25, 0.19318181818181818, 0.03787878787878788, 0.03787878787878788,
0.03787878787878788],
    #      'TP53': [0.32142857142857145, 0.061224489795918366, 0.061224489795918366,
0.27040816326530615, 0.061224489795918366, 0.066326530612244902, 0.051020408163265307, 0.051020408
163265307, 0.056122448979591837],
    #      'EGFR': [0.056818181818181816, 0.21590909090909091, 0.0625, 0.068181818181818177,
0.068181818181818177, 0.0625, 0.34659090909090912, 0.0625, 0.056818181818181816],
    #      'BRCA2': [0.13333333333333333, 0.060606060606060608, 0.060606060606060608,
0.078787878787878782, 0.13939393939393939394, 0.34545454545454546, 0.060606060606060608,
0.060606060606060608, 0.060606060606060608],
    #      'PTEN': [0.069182389937106917, 0.062893081761006289, 0.069182389937106917,
0.46540880503144655, 0.075471698113207544, 0.062893081761006289, 0.069182389937106917, 0.062893081
761006289, 0.062893081761006289],
    #      'KIT': [0.066225165562913912, 0.25165562913907286, 0.072847682119205295,
0.072847682119205295, 0.066225165562913912, 0.066225165562913912, 0.27152317880794702,
0.066225165562913912, 0.066225165562913912],
    #       'BRAF': [0.066666666666666666, 0.17999999999999999, 0.073333333333333334,
0.073333333333333334, 0.093333333333333338, 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 da
ta
    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 [139]:

```
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 : 232
BRCA1     177
TP53       98
EGFR       86
PTEN       82
BRCA2      70
BRAF       68
KIT        61
ERBB2      47
ALK        45
PDGFRA     39
Name: Gene, dtype: int64
```

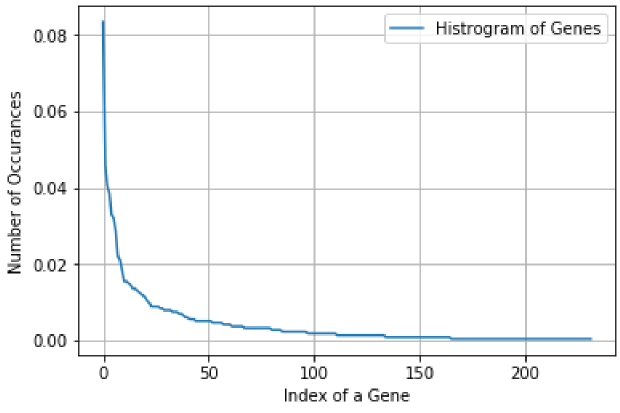
In [140]:

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

```
Ans: There are 232 different categories of genes in the train data, and they are distibuted as fol
lows
```

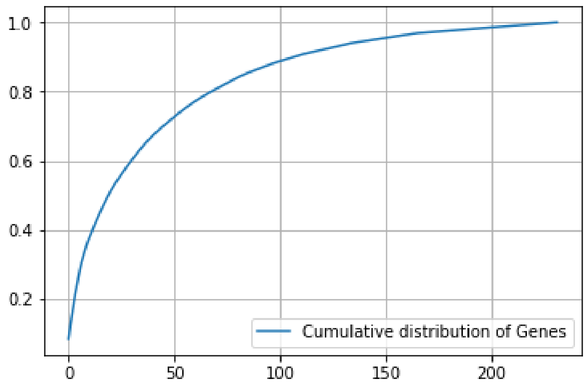
In [141]:

```
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 [142]:

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



Q3. How to featurize this Gene feature ?

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

1. One hot Encoding
2. Response coding

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

In [143]:

```
#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 [144]:

```
print("train_gene_feature_responseCoding is converted feature using respone coding method. The sha
pe of gene feature:", train_gene_feature_responseCoding.shape)
```

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

using Tfidf Vectorizer

In [145]:

```
# one-hot encoding of Gene feature.
gene_vectorizer = TfidfVectorizer()
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 [146]:

```
train_df['Gene'].head()
```

Out[146]:

```
1197    PIK3CA
908     PDGFRA
1917       SMO
1767      IDH2
1459     FGFR2
Name: Gene, dtype: object
```

In [147]:

```
gene_vectorizer.get_feature_names()
```

Out[147]:

```
['abl1',
 'acvr1',
 'ago2',
 'akt1',
 'akt2',
 'akt3',
 'alk',
 'apc',
 'ar',
 'araf',
 'arid1b',
 'arid2',
 'arid5b',
 'asxl1',
 'asxl2',
 'atm',
 'atrx',
 'aurka',
 'aurkb',
 'axin1',
 'axl',
 'b2m',
 'bap1',
 'bard1',
 'bcor',
 'braf',
 'brca1',
 'brca2',
 'brd4',
 'brip1',
 'btk',
 'card11',
 'carm1',
 'casp8',
 'cbl',
 'ccnd1',
 'ccnd3',
 'ccne1',
 'cdh1',
 'cdk12',
 'cdk4',
 'cdk6',
 'cdkn1a',
 'cdkn1b',
 'cdkn2a',
 'cdkn2b',
 'cdkn2c',
 'cebpa',
 'chek2',
 'cic',
 'crebbp',
 'ctcf',
 'ctnnb1',
 'ddr2',
 'dicer1',
 'dnmt3a',
 'dnmt3b',
 'egfr',
 'elf3',
 'ep300',
 'epas1',
 'erbb2',
 'erbb3',
 'erbb4',
 'ercc2',
 'ercc3',
 'ercc4',
 'erg',
 'esr1',
 'etv6',
 'ewsr1',
 'ezh2',
 'fam58a',
 'fanca',
 'fancc',
 'fat1',
 'fbxw7',
 'fgf19',
 'fgfr1',
 'fgfr2',
 'fgfr3',
 'fgfr4',
 'flt1',
 'flt3',
 'foxa1',
 'foxl2',
 'foxo1',
 'foxp1',
 'fubp1',
 'gata3',
 'gli1',
 'gna11',
 'gnas',
 'h3f3a',
 'hist1h1c',
 'hla',
 'hnf1a',
 'hras',
 'idh1',
 'idh2',
 'igf1r',
 'ikbke',
 'ikzf1',
 'il7r',
 'jak1',
 'jak2',
 'jun',
 'kdm5a',
 'kdm5c',
 'kdm6a',
 'kdr',
 'keap1',
 'kit',
 'klf4',
 'kmt2a',
 'kmt2b',
 'kmt2c',
 'kmt2d',
 'knstrn',
 'kras',
 'lats1',
 'map2k1',
 'map2k2',
 'map2k4',
 'map3k1',
 'mapk1',
 'mdm2',
 'med12',
 'mef2b',
 'men1',
 'met',
 'mga',
 'mlh1',
 'mpl',
 'msh2',
 'msh6',
 'mtor',
 'myc',
 'mycn',
 'myd88',
 'myod1',
 'nf1',
 'nf2',
 'nfe2l2',
 'nfkbia',
 'nkx2',
 'notch1',
 'notch2',
 'npm1',
 'nras',
 'nsd1',
 'ntrk1',
 'ntrk2',
 'ntrk3',
 'nup93',
 'pak1',
 'pbrm1',
 'pdgfra',
 'pdgfrb',
 'pik3ca',
 'pik3cb',
 'pik3cd',
 'pik3r1',
 'pik3r2',
 'pik3r3',
 'pim1',
 'pms1',
 'pms2',
 'pole',
 'ppm1d',
 'ppp2r1a',
 'ppp6c',
 'prdm1',
 'pten',
 'ptpn11',
 'ptprd',
 'ptprt',
 'rab35',
 'rac1',
 'rad21',
 'rad50',
 'rad51d',
 'rad54l',
 'raf1',
 'rasa1',
 'rb1',
 'rbm10',
 'ret',
 'rheb',
 'rhoa',
 'rictor',
 'rit1',
 'rnf43',
 'ros1',
 'rras2',
 'runx1',
 'rxra',
 'rybp',
 'sdhc',
 'setd2',
 'sf3b1',
 'smad2',
 'smad3',
 'smad4',
 'smarca4',
 'smarcb1',
 'smo',
 'sos1',
 'sox9',
 'spop',
 'src',
 'srsf2',
 'stat3',
 'stk11',
 'tcf7l2',
 'tert',
 'tet1',
 'tet2',
 'tgfbr1',
 'tgfbr2',
 'tmprss2',
 'tp53',
 'tp53bp1',
 'tsc1',
 'tsc2',
 'u2af1',
 'vhl',
 'whsc1l1',
 'xpo1',
 'xrcc2',
 'yap1']
```

In [148]:

```
train_gene_feature_onehotCoding.shape
```

Out[148]:

```
(2124, 231)
```

In [149]:

```
print("train_gene_feature_onehotCoding is converted feature using one-hot encoding method. The sha
pe of gene feature:", train_gene_feature_onehotCoding.shape)
```

```
train_gene_feature_onehotCoding is converted feature using one-hot encoding method. The shape of g
ene feature: (2124, 231)
```

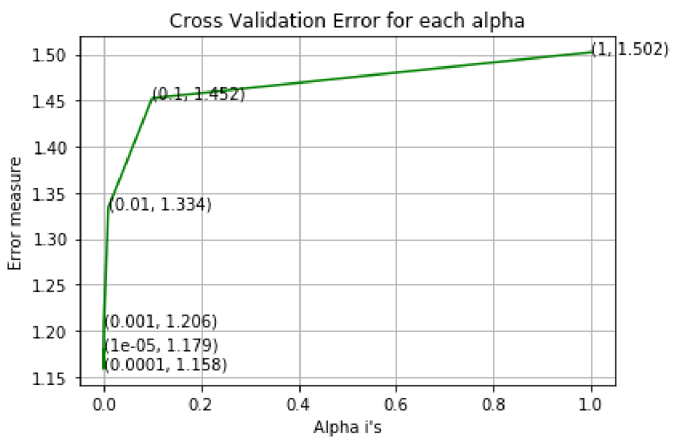
Q4. How good is this gene feature in predicting y_i?

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 [150]:

```
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_i
ter=None, tol=None,
# shuffle=True, verbose=0, epsilon=0.1, n_jobs=1, random_state=None, learning_rate='optimal', eta0
# 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_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.clas
ses_, eps=1e-15))

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


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

predict_y = sig_clf.predict_proba(train_gene_feature_onehotCoding)
print('For values of best alpha = ', alpha[best_alpha], "The train log loss 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_lo
ss(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, p
redict_y, labels=clf.classes_, eps=1e-15))
```

```
For values of alpha =  1e-05 The log loss is: 1.1789324328716622
For values of alpha =  0.0001 The log loss is: 1.1584558406031014
For values of alpha =  0.001 The log loss is: 1.206297692171881
For values of alpha =  0.01 The log loss is: 1.3336809038345863
For values of alpha =  0.1 The log loss is: 1.4524623872110125
For values of alpha =  1 The log loss is: 1.5024386555867797
```



```
For values of best alpha =  0.0001 The train log loss is: 1.0157042349735215
For values of best alpha =  0.0001 The train log loss is: 1.0157043342735215
For values of best alpha =  0.0001 The cross validation log loss is: 1.1584558406031014
For values of best alpha =  0.0001 The test log loss is: 1.155676968067995
```

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 [151]:

```
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.s
hape[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 641 out of 665 : 96.39097744360903
2. In cross validation data 518 out of  532 : 97.36842105263158
```

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 [152]:

```
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 : 1922
Truncating_Mutations    58
Deletion                50
Amplification           50
Fusions                 18
G12V                     4
Overexpression           4
G67R                     2
Q61H                     2
P130S                    2
R841K                    2
Name: Variation, dtype: int64
```

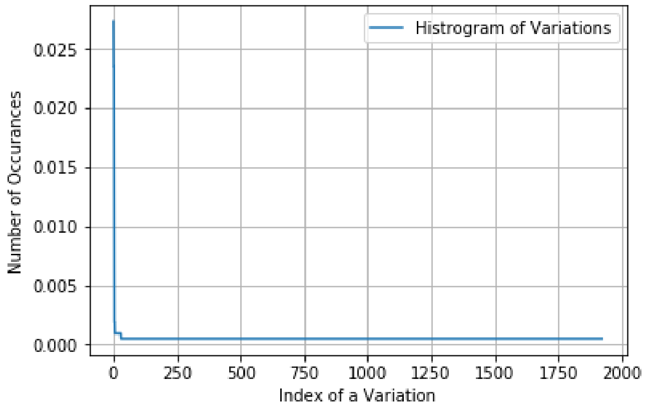
In [153]:

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

```
Ans: There are 1922 different categories of variations in the train data, and they are distibuted
as follows
```

In [154]:

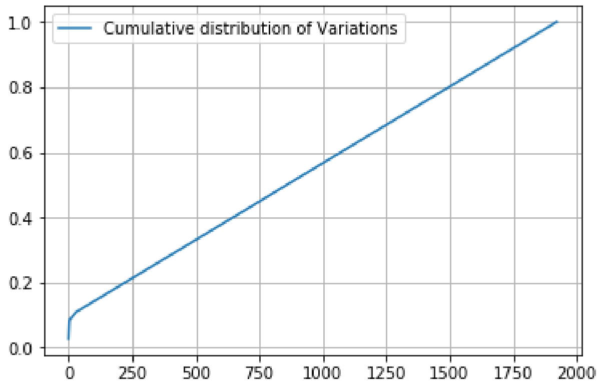
```
s = sum(unique_variations.values);
h = unique_variations.values/s;
plt.plot(h, label="Histrogram of Variations")
plt.xlabel('Index of a Variation')
plt.ylabel('Number of Occurances')
plt.ylabel('Number of Occurances')
plt.legend()
plt.grid()
plt.show()
```



In [155]:

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

```
[0.02730697 0.05084746 0.07438795 ... 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 [156]:

```
# 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 [157]:

```
print("train_variation_feature_responseCoding is a converted feature using the response coding met
hod. The shape of Variation feature:", train_variation_feature_responseCoding.shape)
```

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

In [158]:

```
# one-hot encoding of variation feature.
variation_vectorizer = TfidfVectorizer()
train_variation_feature_onehotCoding = variation_vectorizer.fit_transform(train_df['Variation'])
test_variation_feature_onehotCoding = variation_vectorizer.transform(test_df['Variation'])
cv_variation_feature_onehotCoding = variation_vectorizer.transform(cv_df['Variation'])
```

In [159]:

```
print("train_variation_feature_onehotEncoded is converted feature using the onne-hot encoding meth
od. The shape of Variation feature:", train_variation_feature_onehotCoding.shape)
```

```
train_variation_feature_onehotEncoded is converted feature using the onne-hot encoding method. The
shape of Variation feature: (2124, 1949)
```

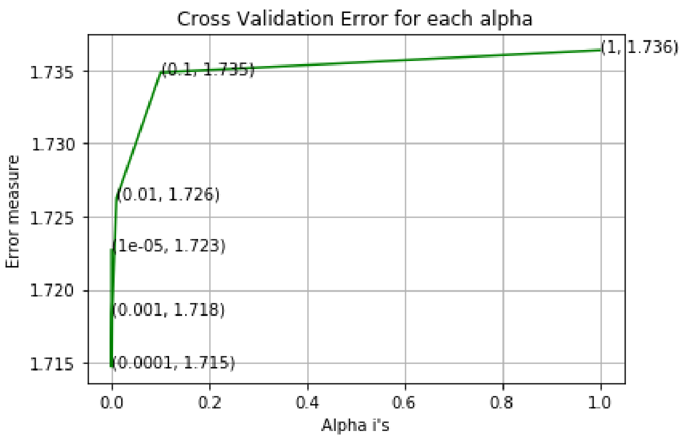
Q10. How good is this Variation feature in predicting y_i?

Let's build a model just like the earlier!

In [160]:

```
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_i
ter=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_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.clas
ses_, eps=1e-15))

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


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

predict_y = sig_clf.predict_proba(train_variation_feature_onehotCoding)
print('For values of best alpha = ', alpha[best_alpha], "The train log loss 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_lo
ss(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, p
redict_y, labels=clf.classes_, eps=1e-15))
```

```
For values of alpha =  1e-05 The log loss is: 1.722715682099923
For values of alpha =  0.0001 The log loss is: 1.7147100444560643
For values of alpha =  0.001 The log loss is: 1.7183806991049357
For values of alpha =  0.01 The log loss is: 1.7262487439521268
For values of alpha =  0.1 The log loss is: 1.734846804634527
For values of alpha =  1 The log loss is: 1.7363778250859812
```



```
For values of best alpha =  0.0001 The train log loss is: 0.7587359477895346
For values of best alpha =  0.0001 The cross validation log loss is: 1.7147100444560643
For values of best alpha =  0.0001 The test log loss is: 1.703239997893374
```

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 [161]:

```
print("Q12. How many data points are covered by total ", unique_variations.shape[0], " genes in te
st and cross validation data sets?")
test_coverage=test_df[test_df['Variation'].isin(list(set(train_df['Variation'])))].shape[0]
cv_coverage=cv_df[cv_df['Variation'].isin(list(set(train_df['Variation'])))].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.s
hape[0])*100)
```

```
Q12. How many data points are covered by total  1922  genes in test and cross validation data
sets?
Ans
1. In test data 71 out of 665 : 10.676691729323307
2. In cross validation data 49 out of  532 : 9.210526315789473
```

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 [162]:

```
# 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 [163]:

```
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 [164]:

```
text_vectorizer = TfidfVectorizer(max_features=1000)
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 featu
res) 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 : 1000
```

In [165]:

```
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 = []
```

```python
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 [166]:

```python
#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 [167]:

```python
# 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 [168]:

```python
# 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 [169]:

```python
#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 [170]:

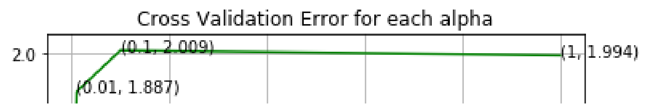
```python
# Number of words for a given frequency.
print(Counter(sorted_text_occur))
```

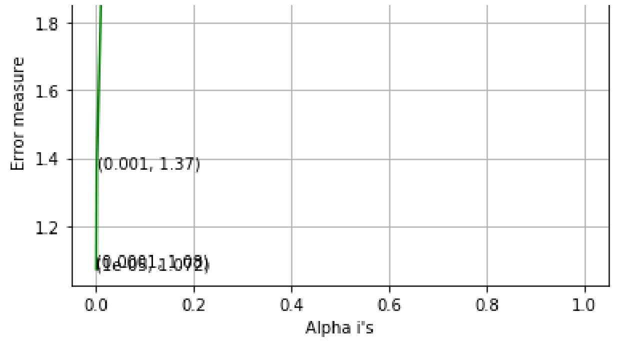
```
Counter({254.60031659999032: 1, 181.36967140535268: 1, 139.1244539073383: 1, 132.44783964715864: 1
, 129.2906964547604: 1, 120.7884618249594: 1, 119.97964270120609: 1, 115.58410664868339: 1,
111.31929777619457: 1, 107.43285486973713: 1, 106.36678981383885: 1, 91.82116117596945: 1,
91.39240996026298: 1, 81.60909398784166: 1, 81.34058076668651: 1, 81.13662118182577: 1,
80.31908029603358: 1, 79.94956858700233: 1, 79.12410704098112: 1, 77.93707299180987: 1,
77.1188383542651: 1, 75.33330143143255: 1, 71.43032555117375: 1, 70.99638045948535: 1,
69.56469032210441: 1, 68.95298695596104: 1, 65.57799106763069: 1, 65.12369825666272: 1,
65.12210525055642: 1, 65.05175043116745: 1, 64.35957179145133: 1, 64.24798540871899: 1,
62.90468497896144: 1, 62.825666594636566: 1, 61.83834545696788: 1, 59.48124972329728: 1,
58.12939577285367: 1, 57.16874510796442: 1, 56.69001499098711: 1, 55.967507322182: 1,
51.55023479288301: 1, 50.58005519893788: 1, 49.298876066985436: 1, 48.99149908110838: 1,
48.51736912045506: 1, 46.971723827220416: 1, 46.71220775369471: 1, 46.37513964499316: 1,
45.503375139735745: 1, 45.02900784771858: 1, 44.71044518775096: 1, 44.63606359554017: 1, 44.3680755
8074327: 1, 44.047300044878114: 1, 43.50080796756966: 1, 42.910730040171046: 1, 42.54590341089916:
1, 42.44806047835177: 1, 42.43893855117277: 1, 42.38290218986407: 1, 42.22203684905353: 1,
42.153954917783174: 1, 41.765872265031724: 1, 41.25810598631099: 1, 40.899241306638785: 1,
40.64602269830038: 1, 39.82020344185819: 1, 39.55619514226753: 1, 39.057592112580465: 1,
38.75246421992971: 1, 38.71138517700061: 1, 38.21384280734549: 1, 38.15808691348543: 1,
38.05096000105942: 1, 37.1744703151084: 1, 37.02179397488504: 1, 36.92308902672349: 1,
36.24320286279216: 1, 36.12304701364881: 1, 36.09685661443974: 1, 36.03943748612013: 1,
35.437108347611165: 1, 35.29322611396302: 1, 35.119526714520156: 1, 35.06091374060988: 1,
34.64001623788402: 1, 34.596019718130535: 1, 34.49984782088614: 1, 34.43012883066156: 1,
33.91394057300195: 1, 33.79841664761775: 1, 33.48539497719878: 1, 33.279191651956694: 1,
32.859294735793654: 1, 32.72863956707378: 1, 32.63214011458974: 1, 32.60185664818724: 1, 32.5458617
56877734: 1, 32.206466131407666: 1, 31.99029678200078: 1, 31.977512384064468: 1,
31.968918928210257: 1, 31.935400014827042: 1, 31.82373798834913: 1, 31.564114507854807: 1,
31.521672619924907: 1, 31.498403229655853: 1, 31.476442215126184: 1, 31.433372104637044: 1,
31.36692453115971: 1, 31.329957281245186: 1, 31.27427787487025: 1, 31.270678094878498: 1,
30.78373319484274: 1, 30.77626052636988: 1, 30.73810840953987: 1, 30.512712216159034: 1,
30.451958255803596: 1, 30.36149456187885: 1, 30.13143711180675: 1, 30.131171597111976: 1,
30.120853042310284: 1, 30.06014048293891: 1, 29.81416803480763: 1, 29.604480372477276: 1,
29.470764343870812: 1, 29.414661036198574: 1, 29.333386259866188: 1, 29.21063300556562: 1,
28.8635338385738: 1, 28.855021758509427: 1, 28.553661379177846: 1, 28.05623460482226: 1,
28.030618553023714: 1, 27.963610606747437: 1, 27.759373262962374: 1, 27.711564287562773: 1,
27.690426889119472: 1, 27.659874136872173: 1, 27.077734760691445: 1, 26.952675611859064: 1,
26.92113903421516: 1, 26.73970417414911: 1, 26.67488534714992: 1, 26.663359790518452: 1,
26.386386796885724: 1, 26.075070131836334: 1, 26.056438777927138: 1, 26.034741133948874: 1,
25.881751260542675: 1, 25.87972363519134: 1, 25.70274710320289: 1, 25.63300635730701: 1, 25.5286338
92423354: 1, 25.264962984994767: 1, 25.168721953943173: 1, 25.070497981005683: 1,
24.998889817678137: 1, 24.78331397975271: 1, 24.7615082741401: 1, 24.72797382215009: 1,
24.702575171914027: 1, 24.686316382015082: 1, 24.66206796961159: 1, 24.594432442688433: 1,
24.522174354944557: 1, 24.23763100108943: 1, 24.122958190881384: 1, 24.073444651313892: 1,
24.00121320093497: 1, 23.989187511216116: 1, 23.926105895046916: 1, 23.86026592441975: 1,
23.847607057925075: 1, 23.828414708095007: 1, 23.809455235492358: 1, 23.80495072609604: 1,
23.665210227549395: 1, 23.664808929568153: 1, 23.58983958634139: 1, 23.5145899068282: 1,
23.40384056213851: 1, 23.375722443921607: 1, 23.365920614336293: 1, 23.237669053122527: 1,
23.211560711765852: 1, 23.175761732303076: 1, 23.123922167641126: 1, 23.09737025789955: 1,
22.983306335579247: 1, 22.935400192757047: 1, 22.88682855002591: 1, 22.88673722206145: 1,
22.807659812196945: 1, 22.793052879681937: 1, 22.790818756072373: 1, 22.74010892825148: 1,
22.688070696037215: 1, 22.648942656122358: 1, 22.544964012204147: 1, 22.520438291661016: 1,
22.42982053896734: 1, 22.394967284448008: 1, 22.2415659427638: 1, 22.068030301958448: 1,
21.9934555747912: 1, 21.96649306136031: 1, 21.951464672471996: 1, 21.90797304151692: 1,
21.89886350227041: 1, 21.888667807236505: 1, 21.80599007531147: 1, 21.791869408826834: 1,
21.78675060786811: 1, 21.690539496052953: 1, 21.687357679570013: 1, 21.65976867418817: 1,
21.65901467779301: 1, 21.533828487103587: 1, 21.490430433323173: 1, 21.35542769431398: 1,
21.34210765654407: 1, 21.334574875547986: 1, 21.308968713612117: 1, 21.20286126800204: 1,
21.169931319167667: 1, 21.162812621711122: 1, 21.147437641233424: 1, 21.10930262506116: 1,
21.094919192598415: 1, 20.91590427401003: 1, 20.853095962771615: 1, 20.842678542028995: 1,
20.798029782962292: 1, 20.71867438819657: 1, 20.677597933417765: 1, 20.629364666943356: 1,
20.62795625915618: 1, 20.61192330112786: 1, 20.551119322326986: 1, 20.49302550135705: 1,
20.491617259202922: 1, 20.421746242221374: 1, 20.36034427529226: 1, 20.271673401891213: 1,
20.197403098825408: 1, 20.194883495647172: 1, 20.115156646824687: 1, 20.090817122976688: 1,
20.084872280885335: 1, 20.009596692061393: 1, 19.813974228524422: 1, 19.764673535590337: 1,
19.76322325956172: 1, 19.715511403647934: 1, 19.669182601159502: 1, 19.588232390514342: 1,
19.568806478334533: 1, 19.511936486041293: 1, 19.499890565724087: 1, 19.49978543210384: 1,
19.41399457292767: 1, 19.412784389820622: 1, 19.385035577352475: 1, 19.302964144707822: 1,
19.29278043811732: 1, 19.28928565225853: 1, 19.213345944901004: 1, 19.18891357467377: 1,
19.18428130209394: 1, 19.18311569611521: 1, 19.136439601166018: 1, 19.133938829932674: 1,
19.128068232616734: 1, 19.116384723263195: 1, 19.10888333114552: 1, 19.108809467990405: 1,
19.00374401232703: 1, 18.980780849638332: 1, 18.960894496208603: 1, 18.880734941190703: 1,
18.87848337681059: 1, 18.857580915273278: 1, 18.847298928793666: 1, 18.81853654075289: 1,
18.812980038502136: 1, 18.789514303810083: 1, 18.775279928321208: 1, 18.772555783047274: 1,
18.7394539762701: 1, 18.73759900700343: 1, 18.71990365440274: 1, 18.718770245574074: 1,
18.695370501271707: 1, 18.65964694681175: 1, 18.514597123546576: 1, 18.48973484597991: 1,
18.48239338820627: 1, 18.429377647178022: 1, 18.331516287230563: 1, 18.30401691105083: 1,
18.289687144564727: 1, 18.289319697298602: 1, 18.177485598189193: 1, 18.16433005824942: 1,
18.164262048287178: 1, 18.0441295522736: 1, 18.03069906411383: 1, 18.023742894960645: 1,
17.978445003728325: 1, 17.97802991223101: 1, 17.938342999801055: 1, 17.92276724866871: 1,
17.91396509894174: 1, 17.898186498406808: 1, 17.841964082204743: 1, 17.839235808469283: 1,
17.833750180194873: 1, 17.81540982568797: 1, 17.814698594364305: 1, 17.764111375758926: 1,
17.73695772925665: 1, 17.717106516315226: 1, 17.708482956833272: 1, 17.67801497286503: 1,
17.626425539526984: 1, 17.607856402473598: 1, 17.59282926405651: 1, 17.55190722899685: 1,
17.523517058773546: 1, 17.492869913101973: 1, 17.479190458066945: 1, 17.43882181653247: 1,
17.370657566165455: 1, 17.347705043923042: 1, 17.319074907957546: 1, 17.204204952474893: 1,
17.199201063919872: 1, 17.19178967334828: 1, 17.038899921999015: 1, 17.031251797055415: 1,
16.992887648374438: 1, 16.953975947137334: 1, 16.950446210176352: 1, 16.926705965278266: 1,
16.842162611902268: 1, 16.827913939581755: 1, 16.804898694507223: 1, 16.795585647927545: 1,
16.745675298767633: 1, 16.73936784106295: 1, 16.68263138586425: 1, 16.67669975635192: 1, 16.6599961
14945013: 1, 16.615704488505223: 1, 16.609498856047928: 1, 16.572466777269856: 1,
16.549287004989345: 1, 16.526872509767898: 1, 16.519511656234904: 1, 16.51815553049317: 1,
16.44988296125493: 1, 16.438251185726756: 1, 16.43601824240516: 1, 16.435788148898745: 1,
16.39755609792958: 1, 16.3962753424439: 1, 16.376368501633642: 1, 16.372593919928217: 1,
16.372246891991626: 1, 16.367611615121263: 1, 16.353453028738436: 1, 16.352759042415375: 1,
16.292016711390215: 1, 16.277560586827562: 1, 16.277533930028827: 1, 16.254128030822084: 1,
16.19778962111838: 1, 16.17000398470881: 1, 16.137884194761142: 1, 16.122724074148124: 1,
16.074704535101674: 1, 16.008239323903194: 1, 15.953293708554316: 1, 15.951504304651197: 1,
15.940661785851587: 1, 15.89705586438102: 1, 15.84634983626387: 1, 15.725567080812263: 1,
15.720995056137685: 1, 15.70699211833352: 1, 15.678491876930302: 1, 15.656644948061615: 1,
15.649513227649717: 1, 15.590439616526448: 1, 15.58709753775075: 1, 15.539758130635953: 1,
15.518105173204976: 1, 15.503715844235485: 1, 15.503204260249188: 1, 15.477579277008656: 1,
15.464759982071133: 1, 15.405477257195646: 1, 15.365165534370485: 1, 15.340423969663405: 1,
15.324369954953143: 1, 15.27594570362778: 1, 15.267033296046382: 1, 15.197569095906658: 1,
15.178768562594103: 1, 15.154393678471033: 1, 15.151245117398275: 1, 15.138247987008452: 1,
15.1213326704921: 1, 15.120113745957179: 1, 15.090732940665214: 1, 15.071232235710479: 1,
15.070082980228335: 1, 15.070007652239308: 1, 15.055956840518144: 1, 15.013760970570367: 1,
15.005346038606366: 1, 15.000411933902388: 1, 14.998248079837287: 1, 14.97848370844834: 1,
14.962889201659438: 1, 14.95522317395233: 1, 14.925645298073665: 1, 14.924862233014487: 1,
14.921898843058951: 1, 14.879631319877836: 1, 14.870809379427653: 1, 14.852264376770183: 1,
14.832109464156547: 1, 14.826570642146788: 1, 14.82423164647881: 1, 14.818942565389323: 1,
14.818897979201315: 1, 14.751492689654123: 1, 14.751378151747142: 1, 14.729745753471132: 1,
14.721383243019899: 1, 14.702876322035527: 1, 14.629203701620993: 1, 14.626995138824297: 1,
14.624098436551343: 1, 14.602865634311827: 1, 14.575346825177403: 1, 14.55986220680105: 1,
14.555004966845983: 1, 14.522689403706204: 1, 14.51551587569559: 1, 14.478708684566008: 1,
14.476586754863614: 1, 14.46809344195611: 1, 14.40312041651165: 1, 14.345814703848559: 1,
14.341750025516774: 1, 14.319210392935732: 1, 14.316372163381688: 1, 14.29613407254371: 1,
14.278951174823186: 1, 14.16759143434116: 1, 14.143321358665832: 1, 14.142830560460466: 1,
14.064844976863448: 1, 14.0543925297033: 1, 14.036065515599985: 1, 14.01166879648084: 1,
13.989711624111314: 1, 13.980658999908686: 1, 13.970942185707267: 1, 13.949227517051826: 1,
13.856392379367803: 1, 13.852635519643778: 1, 13.84752632530114: 1, 13.847119207316174: 1,
13.774027968284516: 1, 13.734259367288908: 1, 13.66868972488054: 1, 13.647546689929039: 1,
13.609016986392731: 1, 13.60510695266033: 1, 13.604482131721811: 1, 13.585536276623085: 1,
13.580308409068671: 1, 13.579873723837283: 1, 13.543228589861151: 1, 13.499915326149807: 1,
13.447393612339352: 1, 13.350603994236398: 1, 13.332329594427682: 1, 13.288384806602568: 1,
13.27475096716131: 1, 13.259471466801699: 1, 13.241490963444011: 1, 13.233067341688002: 1,
13.210934761621754: 1, 13.189062797326418: 1, 13.155556117906727: 1, 13.107131505887766: 1,
13.102029637877413: 1, 13.099277453940452: 1, 13.056989833242087: 1, 13.055697264020283: 1,
13.052576158578963: 1, 13.041551174025226: 1, 13.034486660060344: 1, 13.019925135075484: 1,
13.019074005517838: 1, 13.018481751082147: 1, 12.998562915004879: 1, 12.973034406215973: 1,
12.95886662992038: 1, 12.945619721598241: 1, 12.895287932587406: 1, 12.867777228696022: 1,
12.86170463246842: 1, 12.801519959428848: 1, 12.771092451982874: 1, 12.763822436013593: 1,
12.744609850166455: 1, 12.733617223619541: 1, 12.730384353942252: 1, 12.714616590039665: 1,
12.704870345967487: 1, 12.687290983002182: 1, 12.6740947829709: 1, 12.647597275397654: 1,
12.638155267683006: 1, 12.637229247351106: 1, 12.613879409706273: 1, 12.610915970724207: 1,
12.576697970575387: 1, 12.567008834417202: 1, 12.565474654753583: 1, 12.552384131158725: 1,
12.54854022849807: 1, 12.52889523431169: 1, 12.496103791968778: 1, 12.46593165569728: 1,
12.452803881847961: 1, 12.447457373308595: 1, 12.433361869909271: 1, 12.414630635537174: 1,
12.413915218375314: 1, 12.403037746278047: 1, 12.398921940753286: 1, 12.363495152642209: 1,
12.358198635851295: 1, 12.355317926648198: 1, 12.331652282734451: 1, 12.3163403840796: 1,
12.27334941417992: 1, 12.266464937765544: 1, 12.265433847827138: 1, 12.24670104673339: 1,
12.24486794806003: 1, 12.241343128232671: 1, 12.225508190233828: 1, 12.21524678942007: 1,
12.20559550839882: 1, 12.199220144332562: 1, 12.196148199588505: 1, 12.168920378139063: 1,
12.161972068849906: 1, 12.113621828452944: 1, 12.104907183739922: 1, 12.05430151407421: 1,
12.041063851916139: 1, 12.023286538622418: 1, 11.991681728268986: 1, 11.974932863658202: 1,
11.966629841078651: 1, 11.958767524062608: 1, 11.931510361068387: 1, 11.928295706415717: 1,
11.928100138038145: 1, 11.896502416892554: 1, 11.8750525881477: 1, 11.874373232423627: 1,
11.846922437057167: 1, 11.835084327682674: 1, 11.81373247658931: 1, 11.76910230268786: 1,
11.733837223687646: 1, 11.714806060051561: 1, 11.712980937090547: 1, 11.68412369162697: 1,
11.672278482950112: 1, 11.670859808602506: 1, 11.621689840442219: 1, 11.61571741379983: 1,
11.602101002965332: 1, 11.582036991827426: 1, 11.56445297925327: 1, 11.537900319158796: 1,
11.53461250566026: 1, 11.521411154221099: 1, 11.516844591889418: 1, 11.478145940092324: 1,
11.477559447118832: 1, 11.467307217498853: 1, 11.460356313145015: 1, 11.455750976457328: 1,
11.418829294949854: 1, 11.373493118368367: 1, 11.36782509834524: 1, 11.359930524842248: 1,
11.345915611287628: 1, 11.333802923397203: 1, 11.328274186167134: 1, 11.320440604651086: 1,
11.307544220629968: 1, 11.29837155719925: 1, 11.28213389841879: 1, 11.25946074646514: 1, 11.2493820
0723599: 1, 11.222270133537588: 1, 11.220231960013605: 1, 11.208303500952596: 1,
11.20780490129193: 1, 11.200388271209798: 1, 11.197078804963413: 1, 11.193284966851746: 1,
11.16899564843954: 1, 11.164144134675652: 1, 11.154237121581788: 1, 11.144194929785751: 1,
11.095235914709525: 1, 11.09482346927102: 1, 11.088501401558888: 1, 11.064511979892606: 1,
11.053748318192651: 1, 11.046114969378825: 1, 11.031416460084333: 1, 11.027769119790307: 1,
11.025036978362078: 1, 11.015559767920708: 1, 11.011641118434786: 1, 10.985404524167933: 1,
10.975215170607292: 1, 10.962265239507133: 1, 10.93664555936461: 1, 10.935472475717122: 1,
10.929522080241684: 1, 10.925421660852992: 1, 10.923449632200885: 1, 10.900922872449712: 1,
10.897926752868699: 1, 10.872120661449973: 1, 10.868346299637608: 1, 10.847969480269597: 1,
10.83971873655122: 1, 10.837574985439705: 1, 10.828889524214853: 1, 10.82007295321526: 1,
10.808658857748869: 1, 10.771794397786193: 1, 10.75872557540275: 1, 10.755940678733673: 1,
10.72490489208954: 1, 10.721492132725787: 1, 10.716110627602397: 1, 10.715855779182094: 1,
10.66649608705202: 1, 10.661322969459992: 1, 10.652818976053243: 1, 10.645743328109916: 1,
10.643043438822268: 1, 10.634916526007443: 1, 10.614526975438475: 1, 10.602972326264084: 1,
10.580821871323375: 1, 10.572088396128382: 1, 10.57018681378431: 1, 10.567036680053253: 1,
10.559765893168919: 1, 10.528855383499446: 1, 10.522530765364717: 1, 10.506853092956144: 1,
10.490940442873583: 1, 10.48558249389554: 1, 10.479467330359277: 1, 10.470425373534802: 1,
10.452015921207913: 1, 10.446569889921546: 1, 10.446545330958084: 1, 10.40209506829662: 1,
10.396556822216803: 1, 10.390247966170385: 1, 10.367591274521748: 1, 10.352470850856742: 1,
10.350926266789967: 1, 10.350274312189102: 1, 10.339542965686112: 1, 10.337733398469085: 1,
10.336057182853406: 1, 10.33079718355361: 1, 10.315822108285296: 1, 10.310015023676076: 1,
10.282442293329973: 1, 10.272503706020482: 1, 10.26652191124764: 1, 10.243776943456162: 1,
10.237830248114193: 1, 10.23306385622089: 1, 10.222833995243189: 1, 10.17384953220966: 1,
10.158227975794675: 1, 10.155709014682252: 1, 10.15339258135013: 1, 10.138908769590444: 1,
10.13845001421384: 1, 10.132967497115242: 1, 10.12467999793289: 1, 10.116672702615794: 1,
10.105310121199782: 1, 10.10301283328732: 1, 10.097598846503018: 1, 10.081413259608812: 1,
10.072805679396701: 1, 10.070215188614743: 1, 10.053391659658004: 1, 10.052210725863: 1,
10.050052910438213: 1, 10.028372263275026: 1, 10.025167732420151: 1, 9.985446446374329: 1,
9.979474625281194: 1, 9.964033451555554: 1, 9.962811035690201: 1, 9.947560985806582: 1,
9.945232517361372: 1, 9.929663136577224: 1, 9.926536892359005: 1, 9.913818085188074: 1,
9.88772175269389: 1, 9.883051627478846: 1, 9.880443758025578: 1, 9.873167730141185: 1,
9.842518001634488: 1, 9.839394342972433: 1, 9.829244362858699: 1, 9.819928677663311: 1,
9.80840440538896: 1, 9.803757453441468: 1, 9.796901642479828: 1, 9.782018317174142: 1,
9.770728723701867: 1, 9.760179494058859: 1, 9.752037586767637: 1, 9.745936064105456: 1,
9.732089343673836: 1, 9.726139238481563: 1, 9.72023089915591: 1, 9.711474625815745: 1,
9.705988680048563: 1, 9.660975504119165: 1, 9.655288389893931: 1, 9.64330581961827: 1,
9.640294170382386: 1, 9.63334042402339: 1, 9.615258747815847: 1, 9.603218141585954: 1,
9.576914908092464: 1, 9.575202000011489: 1, 9.570816249093667: 1, 9.565341872149915: 1,
9.548669893293312: 1, 9.524606384537286: 1, 9.4896883774141: 1, 9.459563210546793: 1,
9.458195249296658: 1, 9.423122346804472: 1, 9.411107175210226: 1, 9.403423203585122: 1,
9.39791037752865: 1, 9.389508101295146: 1, 9.375711908376557: 1, 9.374408883067138: 1,
9.37019186813743: 1, 9.368015249095578: 1, 9.34487809514192: 1, 9.335205745144625: 1,
9.333211091635327: 1, 9.332962529424078: 1, 9.302060045508979: 1, 9.297027441571323: 1,
9.287479621086531: 1, 9.280950999157906: 1, 9.256482037846338: 1, 9.24858825866379: 1,
9.227841446964026: 1, 9.22668989134842: 1, 9.219767779321847: 1, 9.20987639555663: 1,
9.199059384873115: 1, 9.196804018351036: 1, 9.188486671407139: 1, 9.187779446441207: 1,
9.187073084040348: 1, 9.180903860734086: 1, 9.17978523371326: 1, 9.170957183160365: 1,
9.17040296819234: 1, 9.162474235278676: 1, 9.141802080402535: 1, 9.137127708657857: 1,
9.131669862148446: 1, 9.09888224509611: 1, 9.098664561875191: 1, 9.09242599157786: 1,
9.07602822530646: 1, 9.068727668319838: 1, 9.0628788714391: 1, 9.054938734290786: 1,
9.049084969240294: 1, 9.041286205217435: 1, 9.040063705003648: 1, 9.039163517930291: 1,
9.030992086663387: 1, 9.030727407448653: 1, 9.023571400803592: 1, 8.991697376017799: 1,
8.991581182044458: 1, 8.981794667768593: 1, 8.964891642746075: 1, 8.960991758661978: 1,
8.960631820757577: 1, 8.946294895338069: 1, 8.943925447540341: 1, 8.918348661883376: 1,
8.916017151539915: 1, 8.900264198245283: 1, 8.886978428081743: 1, 8.867632719325542: 1,
8.864929879956122: 1, 8.858517172297615: 1, 8.841784297133852: 1, 8.823428878585995: 1,
8.819688716905132: 1, 8.800277394410756: 1, 8.790844050378517: 1, 8.790821110732772: 1,
8.781133877368736: 1, 8.775709568484418: 1, 8.748941970700267: 1, 8.745482637725503: 1,
8.741948285202207: 1, 8.734137485150063: 1, 8.732230480919966: 1, 8.72161085180069: 1,
8.72039779116211: 1, 8.717879177809824: 1, 8.714515759718102: 1, 8.691549939185066: 1,
8.665244983237107: 1, 8.638966496120648: 1, 8.634722153087028: 1, 8.634692279150949: 1,
8.626519298707926: 1, 8.617298080460252: 1, 8.6050294194739: 1, 8.592442525919054: 1,
8.583296280552483: 1, 8.581722307732703: 1, 8.580718254035352: 1, 8.574406497753742: 1,
8.566498447767296: 1, 8.529379883785976: 1, 8.52317986568665: 1, 8.521718419358875: 1,
8.499564642754464: 1, 8.49272612495865: 1, 8.490877367473415: 1, 8.486775844688713: 1,
8.482619322951424: 1, 8.480801842433909: 1, 8.471379771411497: 1, 8.458832707207845: 1,
8.457262926143187: 1, 8.449703365133924: 1, 8.441970915907895: 1, 8.42862254608282 4: 1,
8.41791286027911: 1, 8.406545134657064: 1, 8.406020752016282: 1, 8.394812450097401: 1,
8.389175227957345: 1, 8.389001275130942: 1, 8.321404168264378: 1, 8.320441879325777: 1,
8.319337068537514: 1, 8.30869056716971: 1, 8.306979920827633: 1, 8.281735208875421: 1,
8.260524509937213: 1, 8.255409930977773: 1, 8.232779543744176: 1, 8.225703277942598: 1,
8.224252979459543: 1, 8.222094293414974: 1, 8.18876547247572: 1, 8.160430569378294: 1,
8.14925489625076: 1, 8.140644431143391: 1, 8.11676991947235: 1, 8.102678343023683: 1,
8.067237014227853: 1, 8.067081767143398: 1, 8.03798059768669: 1, 8.035954227389201: 1,
8.009954643804345: 1, 8.009207120460458: 1, 8.00241276225978: 1, 7.9969515718070765: 1,
7.960188230035334: 1, 7.938195513914655: 1, 7.92973903578881: 1, 7.928967077726159: 1,
7.9267774106003195: 1, 7.918531710671114: 1, 7.912482510285999: 1, 7.909744340896359: 1, 7.90197943
5381144: 1, 7.897421943227811: 1, 7.894197626549555: 1, 7.88398255036892: 1, 7.883155278977621: 1,
7.883146702850926: 1, 7.869194841418173: 1, 7.852659029712529: 1, 7.831231737501506: 1,
7.827994498140319: 1, 7.826597263819556: 1, 7.819972982944601: 1, 7.781466782068: 1,
7.77940689424178: 1, 7.778648550470045: 1, 7.773998152311707: 1, 7.77252474401389: 1,
7.769987884137524: 1, 7.728346291797125: 1, 7.712228619641798: 1, 7.696542958233045: 1,
7.650866133154645: 1, 7.624079900084845: 1, 7.615439961641099: 1, 7.608015062086902: 1,
7.607514653416548: 1, 7.606830004397691: 1, 7.604551379754622: 1, 7.593795422138257: 1,
7.586786532215794: 1, 7.58475025616047: 1, 7.4915994301279145: 1, 7.469820799284037: 1,
7.463604526491814: 1, 7.454043049006329: 1, 7.439588058922469: 1, 7.438447140695082: 1,
7.432431775703279: 1, 7.424826722007258: 1, 7.41969183954611: 1, 7.397878214202971: 1,
7.375842950344784: 1, 7.3600225898710425: 1, 7.351906742689323: 1, 7.328960879122697: 1,
7.3113060051403895: 1, 7.284870382630733: 1, 7.260567178582891: 1, 7.223611425370105: 1, 7.17521788
2002423: 1, 7.157478712182725: 1, 7.141204001948199: 1, 7.08291415275933 3: 1, 7.071718516692469: 1,
7.070550608052932: 1, 7.063458490174363: 1, 7.05801882688816: 1, 7.0254706361120265: 1,
6.96560883679443: 1, 6.9543343629169065: 1, 6.9054154080066015: 1, 6.891767674990287: 1,
6.852417144697403: 1, 6.802577956305285: 1, 6.783619814452349: 1, 6.748302737528129: 1,
6.7251022907917175: 1, 6.72485848148541: 1, 6.683668149626544: 1, 6.682520967872663: 1,
6.603466188983558: 1, 6.567868847257307: 1, 6.428724425903165: 1, 6.301828073910257: 1,
6.000026238977595: 1})
```

In [171]:

```
# 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_i
ter=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.clas
ses_, 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=clf.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_lo
ss(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, p
redict_y, labels=clf.classes_, eps=1e-15))
```

```
For values of alpha =  1e-05 The log loss is: 1.071554136663342
For values of alpha =  0.0001 The log loss is: 1.0801741124140896
For values of alpha =  0.001 The log loss is: 1.3702794398743945
For values of alpha =  0.01 The log loss is: 1.8868544869563513
For values of alpha =  0.1 The log loss is: 2.0087784624709344
For values of alpha =  1 The log loss is: 1.9938353754816662
```





```
For values of best alpha =  1e-05 The train log loss is: 0.7380143513825383
For values of best alpha =  1e-05 The cross validation log loss is: 1.071554136663342
For values of best alpha =  1e-05 The test log loss is: 1.0408124364917009
```

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

Ans. Yes, it seems like!

In [172]:

```
def get_intersec_text(df):
    df_text_vec = TfidfVectorizer(max_features=1000)
    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 [173]:

```
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")
```

```
94.7 % of word of test data appeared in train data
94.1 % of word of Cross Validation appeared in train data
```

In [174]:

```
print()
from prettytable import PrettyTable
ptable = PrettyTable()
ptable.field_names=["feature","Train","CV","Test"]
ptable.add_row(["Gene ","1.015","1.15","1.15"])
ptable.add_row(["Variation","0.65","1.0079","0.02035"])
ptable.add_row(["Text","0.734","1.2354","1.1266"])

print(ptable)
print()
```

```
+-----------+-------+--------+---------+
|  feature  | Train |   CV   |   Test  |
+-----------+-------+--------+---------+
|   Gene    | 1.015 |  1.15  |   1.15  |
| Variation |  0.65 | 1.0079 | 0.02035 |
|    Text   | 0.734 | 1.2354 |  1.1266 |
+-----------+-------+--------+---------+
```

FROM THE UNIVARIATE ANALYSiS we understand that:-

1. TeXt is the most important among all three

2. For feature variation model overfits

4. Machine Learning Models

In [175]:

```
#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 [176]:

```
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 [177]:

```
# 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)

    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_r
o))
        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")
```

Stacking the three types of features

In [178]:

```
# 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)).tocs
r()
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 [179]:

```
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, 3180)
(number of data points * number of features) in test data =  (665, 3180)
(number of data points * number of features) in cross validation data = (532, 3180)
```

In [180]:

```
print(" Response encoding features :")
print("(number of data points * number of features) in train data = ", train_x_responseCoding.shap
e)
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

4.1.1. Naive Bayes

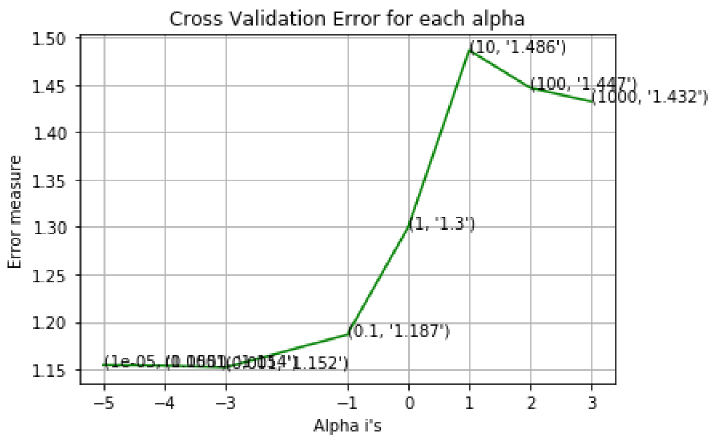
4.1.1.1. Hyper parameter tuning

In [181]:

```
# 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("Log Loss :",log_loss(cv_y, sig_clf_probs))

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


best_alpha = np.argmin(cv_log_error_array)
clf = MultinomialNB(alpha=alpha[best_alpha])
clf.fit(train_x_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=clf.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 alpha = 1e-05
Log Loss : 1.1548253913182243
for alpha = 0.0001
Log Loss : 1.1540821951846592
for alpha = 0.001
Log Loss : 1.1521862513991283
for alpha = 0.1
Log Loss : 1.1865345274640409
for alpha = 1
Log Loss : 1.299976493347943
for alpha = 10
Log Loss : 1.4863234834656882
for alpha = 100
Log Loss : 1.4467536628052837
for alpha = 1000
Log Loss : 1.4323880286124564
```



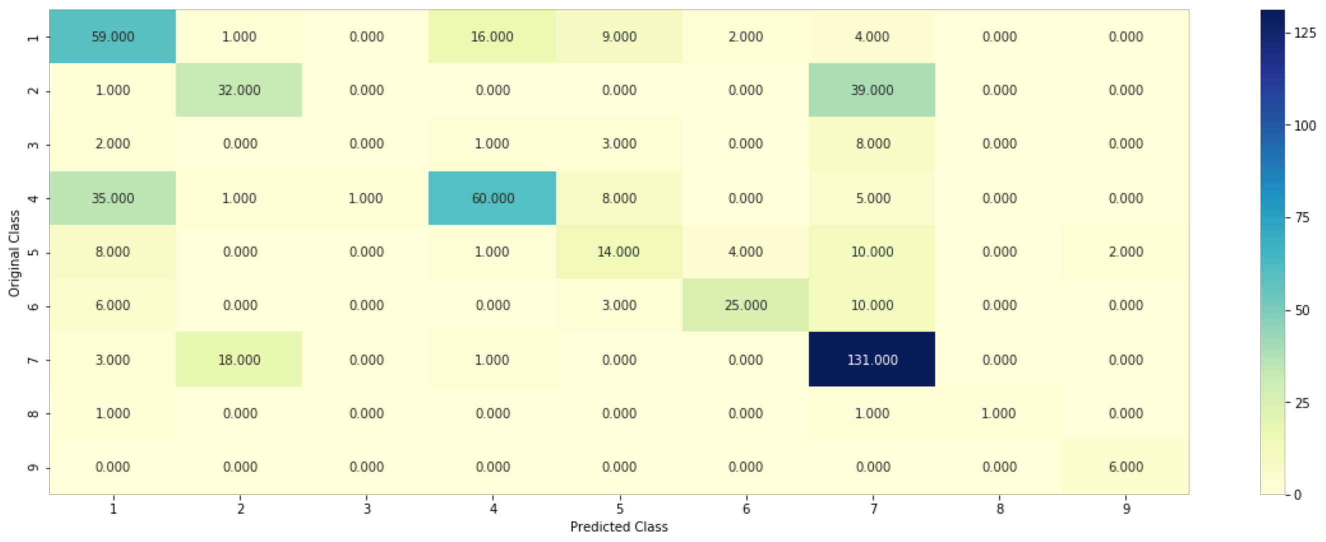
```
For values of best alpha =  0.001 The train log loss is: 0.5260865372653016
For values of best alpha =  0.001 The cross validation log loss is: 1.1521862513991283
For values of best alpha =  0.001 The test log loss is: 1.1488886877146576
```

4.1.1.2. Testing the model with best hyper paramters

In [182]:

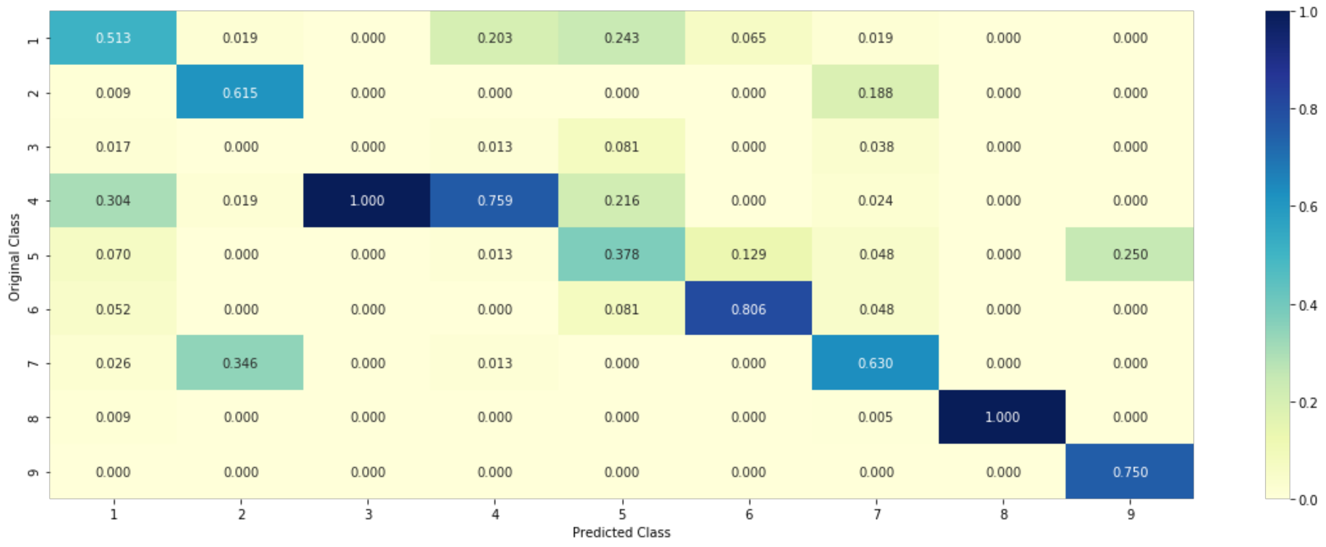
```
# find more about Multinomial Naive base function here http://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.MultinomialNB.html
# -------------------------
# default paramters
# sklearn.naive_bayes.MultinomialNB(alpha=1.0, fit_prior=True, class_prior=None)

# 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 : 1.1521862513991283
Number of missclassified point : 0.38345864661654133
-------------------- Confusion matrix --------------------
```



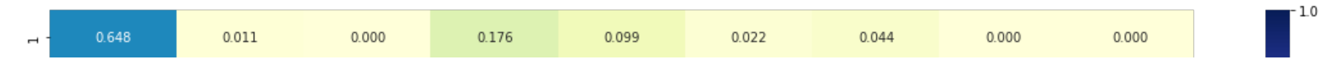
	1	2	3	4	5	6	7	8	9
1	59.000	1.000	0.000	16.000	9.000	2.000	4.000	0.000	0.000
2	1.000	32.000	0.000	0.000	0.000	0.000	39.000	0.000	0.000
3	2.000	0.000	0.000	1.000	3.000	0.000	8.000	0.000	0.000
4	35.000	1.000	1.000	60.000	8.000	0.000	5.000	0.000	0.000
5	8.000	0.000	0.000	1.000	14.000	4.000	10.000	0.000	2.000
6	6.000	0.000	0.000	0.000	3.000	25.000	10.000	0.000	0.000
7	3.000	18.000	0.000	1.000	0.000	0.000	131.000	0.000	0.000
8	1.000	0.000	0.000	0.000	0.000	0.000	1.000	1.000	0.000
9	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	6.000

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

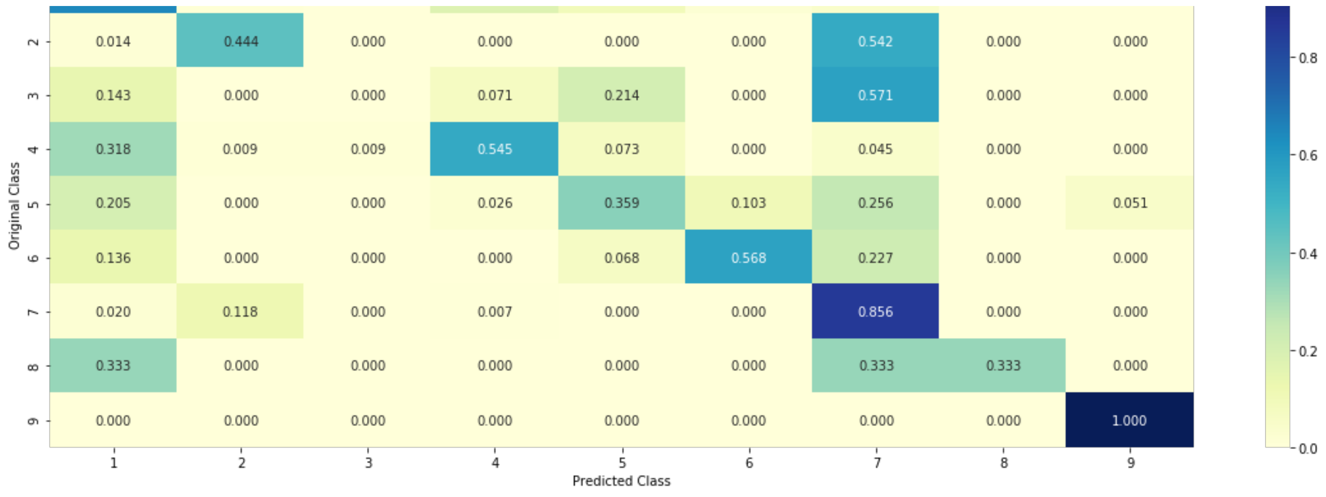


	1	2	3	4	5	6	7	8	9
1	0.513	0.019	0.000	0.203	0.243	0.065	0.019	0.000	0.000
2	0.009	0.615	0.000	0.000	0.000	0.000	0.188	0.000	0.000
3	0.017	0.000	0.000	0.013	0.081	0.000	0.038	0.000	0.000
4	0.304	0.019	1.000	0.759	0.216	0.000	0.024	0.000	0.000
5	0.070	0.000	0.000	0.013	0.378	0.129	0.048	0.000	0.250
6	0.052	0.000	0.000	0.000	0.081	0.806	0.048	0.000	0.000
7	0.026	0.346	0.000	0.013	0.000	0.000	0.630	0.000	0.000
8	0.009	0.000	0.000	0.000	0.000	0.000	0.005	1.000	0.000
9	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.750

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



1	0.648	0.011	0.000	0.176	0.099	0.022	0.044	0.000	0.000



4.1.1.3. Feature Importance, Correctly classified point

In [183]:

```
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: [[0.1717 0.0467 0.0102 0.6089 0.0351 0.0325 0.0888 0.0035 0.0027]]
Actual Class : 4
--------------------------------------------------
Out of the top  100  features  0 are present in query point
```

4.1.1.4. Feature Importance, Incorrectly classified point

In [184]:

```
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: [[0.0744 0.0518 0.0498 0.0737 0.0409 0.0352 0.6675 0.0037 0.003 ]]
Actual Class : 7
--------------------------------------------------
26 Text feature [05] present in test data point [True]
90 Text feature [10] present in test data point [True]
Out of the top  100  features  2 are present in query point
```

4.2. K Nearest Neighbour Classification

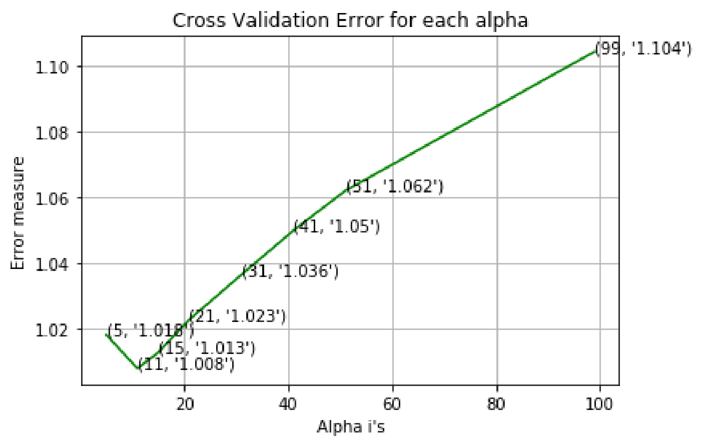
4.2.1. Hyper parameter tuning

In [185]:

```
# 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-ne
ighbors-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 alpha =", i)
    clf = KNeighborsClassifier(n_neighbors=i)
    clf.fit(train_x_responseCoding, train_y)
    sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
    sig_clf.fit(train_x_responseCoding, train_y)
    sig_clf_probs = sig_clf.predict_proba(cv_x_responseCoding)
    cv_log_error_array.append(log_loss(cv_y, sig_clf_probs, labels=clf.classes_, eps=1e-15))
    # to avoid rounding error while multiplying probabilites we use log-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 = KNeighborsClassifier(n_neighbors=alpha[best_alpha])
clf.fit(train_x_responseCoding, train_y)
sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
sig_clf.fit(train_x_responseCoding, train_y)

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

```
redict_y, labels=clf.classes_, eps=1e-15))
```

```
for alpha = 5
Log Loss : 1.0181338181196238
for alpha = 11
Log Loss : 1.0079242117954794
for alpha = 15
Log Loss : 1.0127917584630135
for alpha = 21
Log Loss : 1.0228105243669339
for alpha = 31
Log Loss : 1.0364146399800747
for alpha = 41
Log Loss : 1.0498612719447853
for alpha = 51
Log Loss : 1.0619738191612091
for alpha = 99
Log Loss : 1.1043038497847504
```



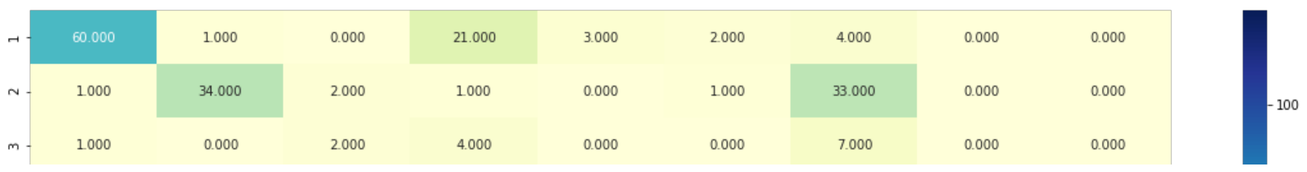
```
For values of best alpha =  11 The train log loss is: 0.6549266077383045
For values of best alpha =  11 The cross validation log loss is: 1.0079242117954794
For values of best alpha =  11 The test log loss is: 1.0203507230331548
```

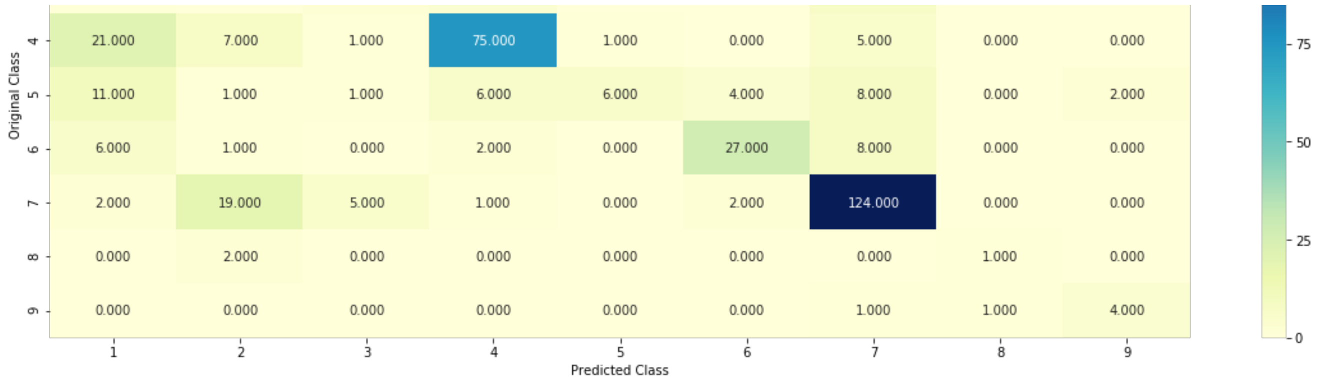
4.2.2. Testing the model with best hyper paramters

In [186]:

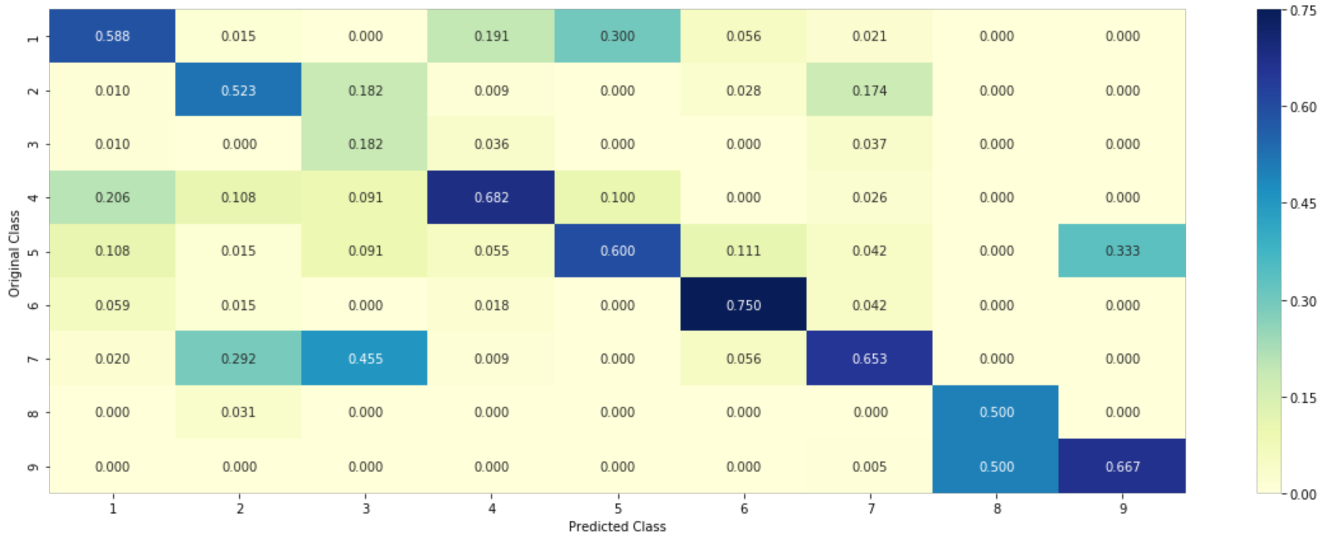
```
# find more about KNeighborsClassifier() here http://scikit-
learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html
# -------------------------
# default parameter
# KNeighborsClassifier(n_neighbors=5, weights='uniform', algorithm='auto', leaf_size=30, p=2,
# metric='minkowski', 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-ne
ighbors-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 : 1.0079242117954794
Number of mis-classified points : 0.37406015037593987
-------------------- Confusion matrix --------------------
```

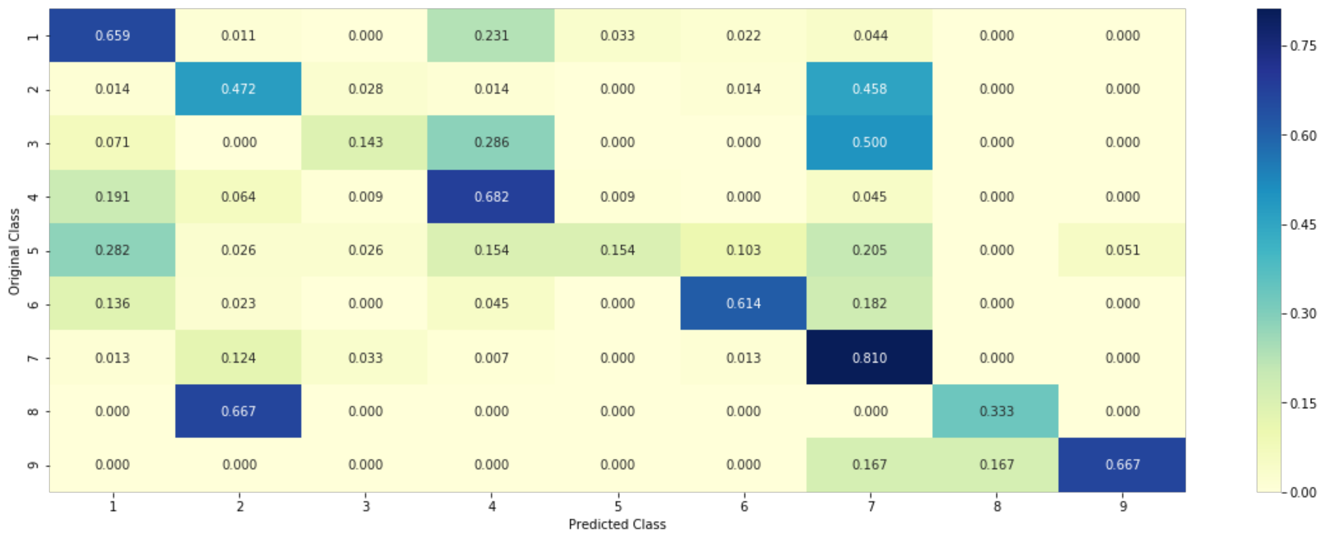




```
-------------------- Precision matrix (Columm Sum=1) --------------------
```



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



4.2.3.Sample Query point -1

In [187]:

```
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 : 7
Actual Class : 4
The  11  nearest neighbours of the test points belongs to classes [4 4 4 4 6 1 4 1 4 4 4]
Fequency of nearest points : Counter({4: 8, 1: 2, 6: 1})
```

4.2.4. Sample Query Point-2

In [188]:

```
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 be
longs to classes",train_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 11 and the nearest neighbours of the test points belongs to classes [3 3 7
7 7 7 7 7 7 3 7]
Fequency of nearest points : Counter({7: 8, 3: 3})
```

4.3. Logistic Regression

4.3.1. With Class balancing

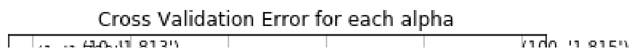
4.3.1.1. Hyper paramter tuning

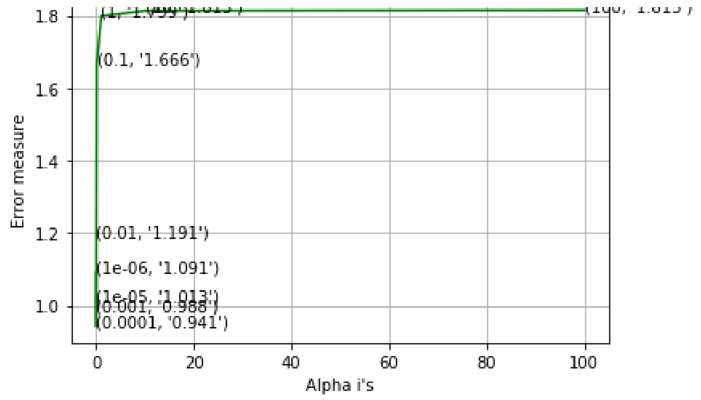
In [189]:

```
# 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_i
ter=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-in
tuition-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, 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("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', ran
dom_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=clf.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_lo
ss(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, p
redict_y, labels=clf.classes_, eps=1e-15))
```

```
for alpha = 1e-06
Log Loss : 1.0906835686687502
for alpha = 1e-05
Log Loss : 1.0130622652814738
for alpha = 0.0001
Log Loss : 0.9410139241547967
for alpha = 0.001
Log Loss : 0.9878851667760031
for alpha = 0.01
Log Loss : 1.1908398059574339
for alpha = 0.1
Log Loss : 1.6660196954683817
for alpha = 1
Log Loss : 1.7989970725784057
for alpha = 10
Log Loss : 1.8131396047194521
for alpha = 100
Log Loss : 1.8148444574863152
```


Cross Validation Error for each alpha



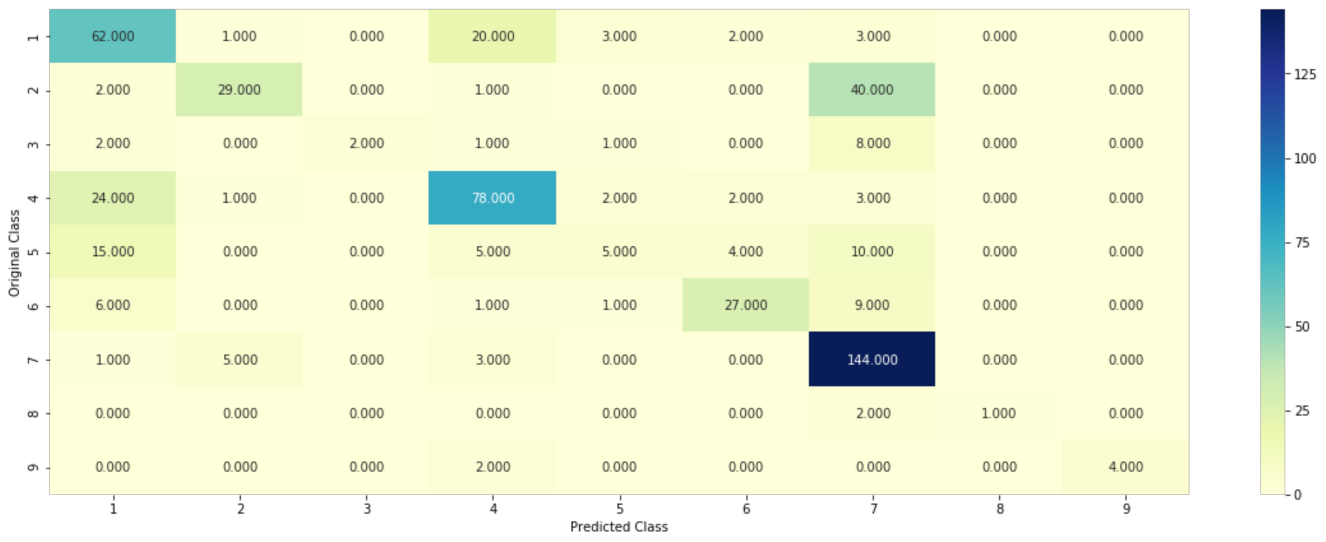
```
For values of best alpha =  0.0001 The train log loss is: 0.4568105113960679
For values of best alpha =  0.0001 The cross validation log loss is: 0.9410139241547967
For values of best alpha =  0.0001 The test log loss is: 0.941942987419883
```

4.3.1.2. Testing the model with best hyper paramters

In [190]:

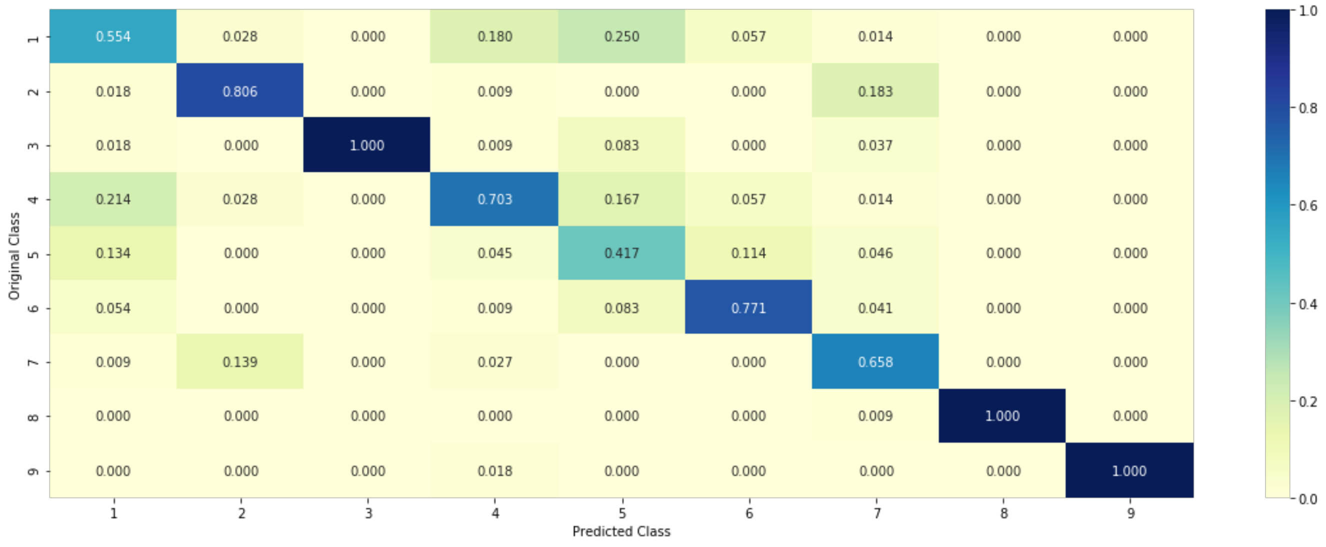
```python
# read more about SGDClassifier() at http://scikit-
learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html
# ------------------------------
# default parameters
# SGDClassifier(loss='hinge', penalty='l2', alpha=0.0001, l1_ratio=0.15, fit_intercept=True, max_i
ter=None, tol=None,
# 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-in
tuition-1/
#------------------------------
clf = SGDClassifier(class_weight='balanced', alpha=alpha[best_alpha], penalty='l2', loss='log', ran
dom_state=42)
predict_and_plot_confusion_matrix(train_x_onehotCoding, train_y, cv_x_onehotCoding, cv_y, clf)
```

```
Log loss : 0.9410139241547967
Number of mis-classified points : 0.3383458646616541
-------------------- Confusion matrix --------------------
```

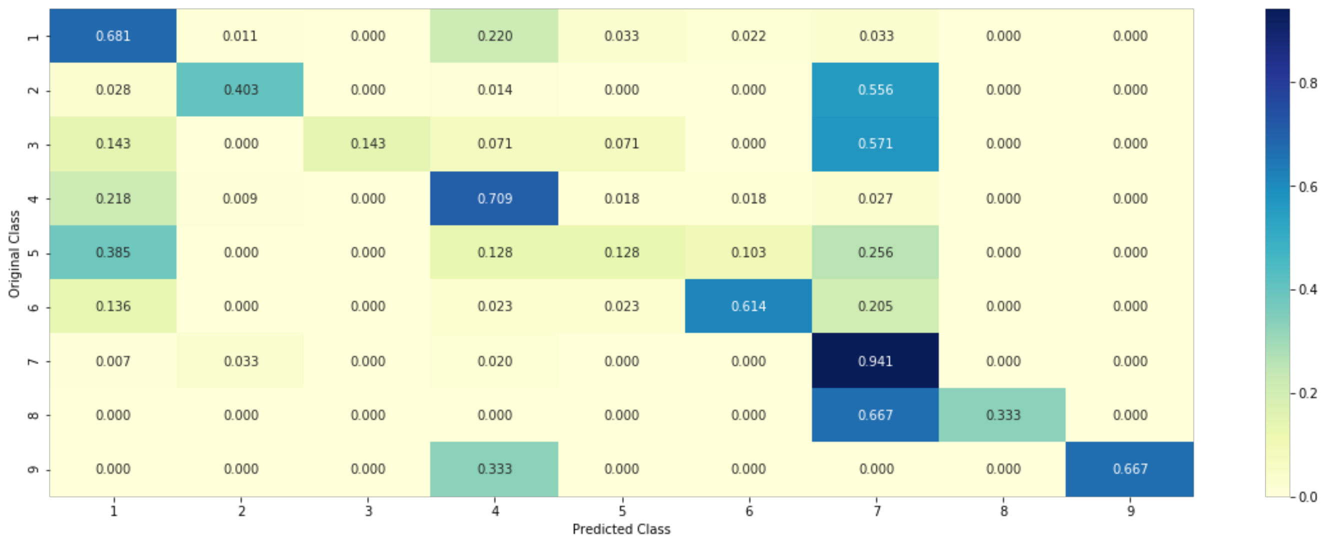


	1	2	3	4	5	6	7	8	9
1	62.000	1.000	0.000	20.000	3.000	2.000	3.000	0.000	0.000
2	2.000	29.000	0.000	1.000	0.000	0.000	40.000	0.000	0.000
3	2.000	0.000	2.000	1.000	1.000	0.000	8.000	0.000	0.000
4	24.000	1.000	0.000	78.000	2.000	2.000	3.000	0.000	0.000
5	15.000	0.000	0.000	5.000	5.000	4.000	10.000	0.000	0.000
6	6.000	0.000	0.000	1.000	1.000	27.000	9.000	0.000	0.000
7	1.000	5.000	0.000	3.000	0.000	0.000	144.000	0.000	0.000
8	0.000	0.000	0.000	0.000	0.000	0.000	2.000	1.000	0.000
9	0.000	0.000	0.000	2.000	0.000	0.000	0.000	0.000	4.000

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



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



4.3.1.3. Feature Importance

In [191]:

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

4.3.1.3.1. Correctly Classified point

In [192]:

```
# from tabulate import tabulate
clf = SGDClassifier(class_weight='balanced', alpha=alpha[best_alpha], penalty='l2', loss='log', ran
dom_state=42)
clf.fit(train_x_onehotCoding,train_y)
test_point_index = 1
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 : 4
Predicted Class Probabilities: [[1.839e-01 2.800e-03 6.000e-04 7.917e-01 5.100e-03 4.600e-03 6.600
e-03
  4.500e-03 2.000e-04]]
Actual Class : 4
--------------------------------------------------
175 Text feature [102] present in test data point [True]
417 Text feature [104] present in test data point [True]
Out of the top  500  features  2 are present in query point
```

4.3.1.3.2. Incorrectly Classified point

In [193]:

```
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: [[0.0152 0.0145 0.0435 0.0134 0.0656 0.0056 0.8369 0.0043 0.001 ]]
Actual Class : 7
--------------------------------------------------
Out of the top  500  features  0 are present in query point
```

4.3.2. Without Class balancing

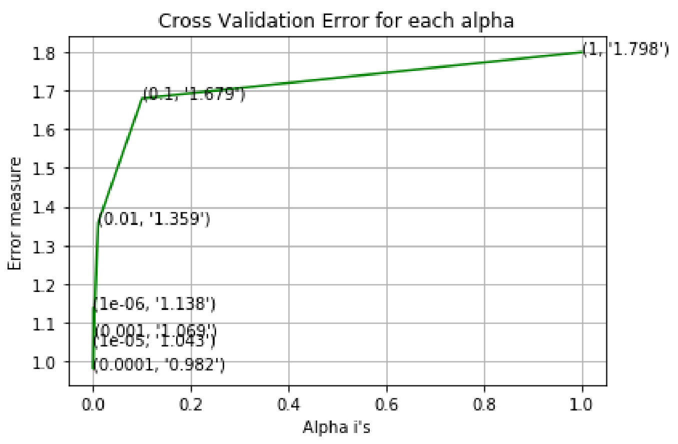
4.3.2.1. Hyper paramter tuning

In [194]:

```
# 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_i
ter=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-in
tuition-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, 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 = SGDClassifier(alpha=alpha[best_alpha], penalty='l2', loss='log', 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=clf.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_lo
ss(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, p
redict_y, labels=clf.classes_, eps=1e-15))
```

```
for alpha = 1e-06
Log Loss : 1.1376624757712681
for alpha = 1e-05
Log Loss : 1.043277111526117
for alpha = 0.0001
Log Loss : 0.9815422741499474
for alpha = 0.001
Log Loss : 1.0690008141672187
for alpha = 0.01
Log Loss : 1.359094946725142
for alpha = 0.1
Log Loss : 1.6791115844909457
for alpha = 1
```

```
Log Loss : 1.7981929615409313
```



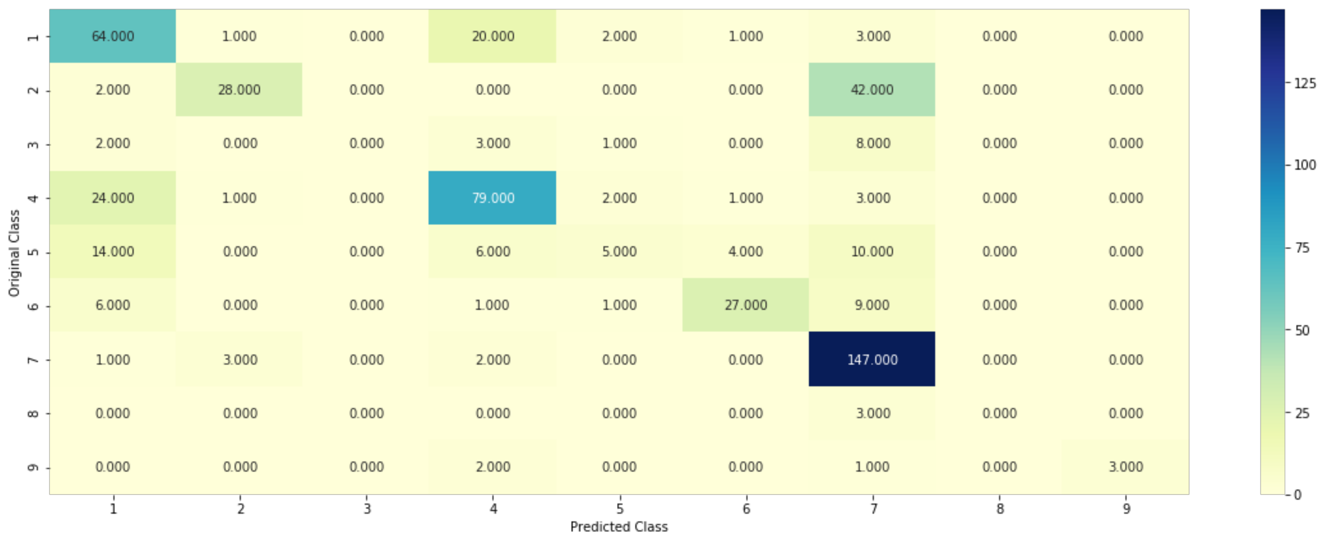
```
For values of best alpha =  0.0001 The train log loss is: 0.4478657079663619
For values of best alpha =  0.0001 The cross validation log loss is: 0.9815422741499474
For values of best alpha =  0.0001 The test log loss is: 0.9661995169763947
```

4.3.2.2. Testing model with best hyper parameters

In [195]:

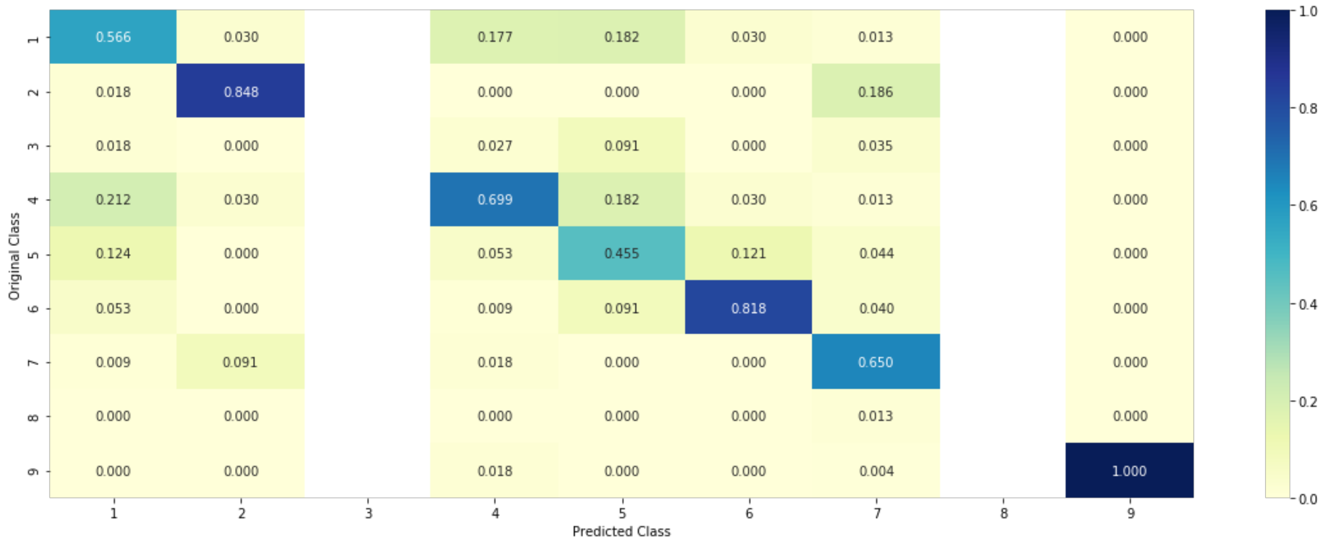
```
# read more about SGDClassifier() at http://scikit-
learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html
# ------------------------------
# default parameters
# SGDClassifier(loss='hinge', penalty='l2', alpha=0.0001, l1_ratio=0.15, fit_intercept=True, max_i
ter=None, tol=None,
# 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, train_y, cv_x_onehotCoding, cv_y, clf)
```

```
Log loss : 0.9815422741499474
Number of mis-classified points : 0.33646616541353386
-------------------- Confusion matrix --------------------
```

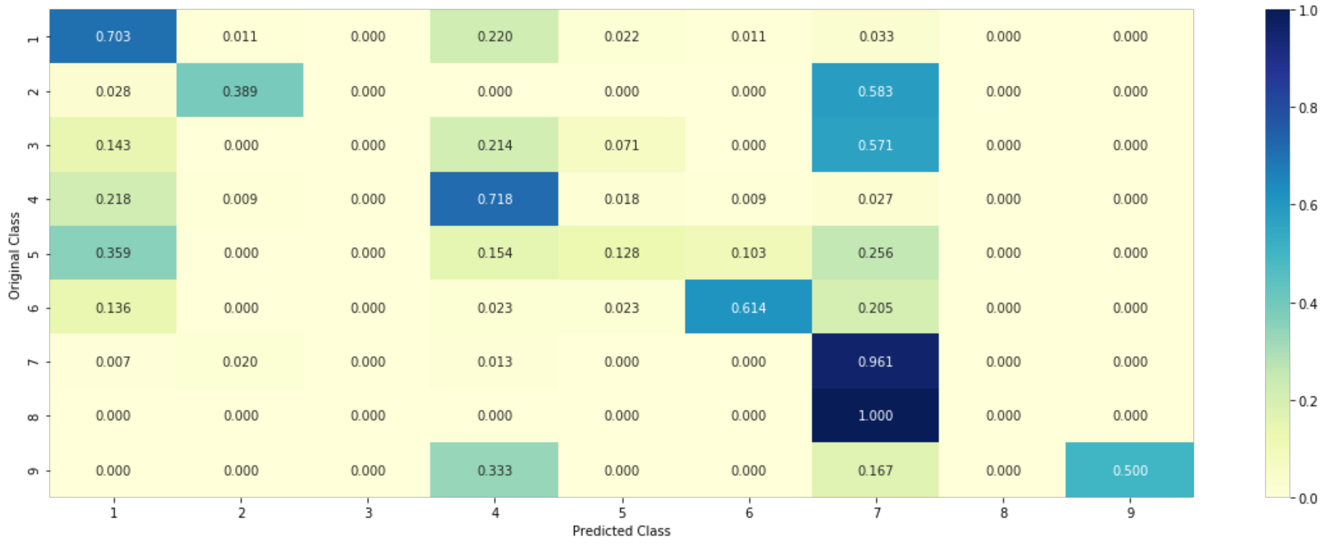


	1	2	3	4	5	6	7	8	9
1	64.000	1.000	0.000	20.000	2.000	1.000	3.000	0.000	0.000
2	2.000	28.000	0.000	0.000	0.000	0.000	42.000	0.000	0.000
3	2.000	0.000	0.000	3.000	1.000	0.000	8.000	0.000	0.000
4	24.000	1.000	0.000	79.000	2.000	1.000	3.000	0.000	0.000
5	14.000	0.000	0.000	6.000	5.000	4.000	10.000	0.000	0.000
6	6.000	0.000	0.000	1.000	1.000	27.000	9.000	0.000	0.000
7	1.000	3.000	0.000	2.000	0.000	0.000	147.000	0.000	0.000
8	0.000	0.000	0.000	0.000	0.000	0.000	3.000	0.000	0.000
9	0.000	0.000	0.000	2.000	0.000	0.000	1.000	0.000	3.000

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



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



FRom Observing label 8 you can see that the value is zero although the loss is less in Logistic regression but for minor class it works bad

4.3.2.3. Feature Importance, Correctly Classified point

In [196]:

```
clf = SGDClassifier(alpha=alpha[best_alpha], penalty='l2', loss='log', random_state=42)
clf.fit(train_x_onehotCoding,train_y)
test_point_index = 1
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 : 4
Predicted Class Probabilities: [[2.133e-01 2.800e-03 5.000e-04 7.640e-01 4.400e-03 4.100e-03 8.800
e-03
  2.000e-03 0.000e+00]]
Actual Class : 4
--------------------------------------------------
181 Text feature [102] present in test data point [True]
456 Text feature [104] present in test data point [True]
Out of the top  500  features  2 are present in query point
```

4.3.2.4. Feature Importance, Inorrectly Classified point

In [197]:

```
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: [[1.530e-02 1.370e-02 3.250e-02 1.370e-02 5.790e-02 5.100e-03 8.574
e-01
  3.900e-03 5.000e-04]]
Actual Class : 7
--------------------------------------------------
Out of the top  500  features  0 are present in query point
```

4.4. Linear Support Vector Machines

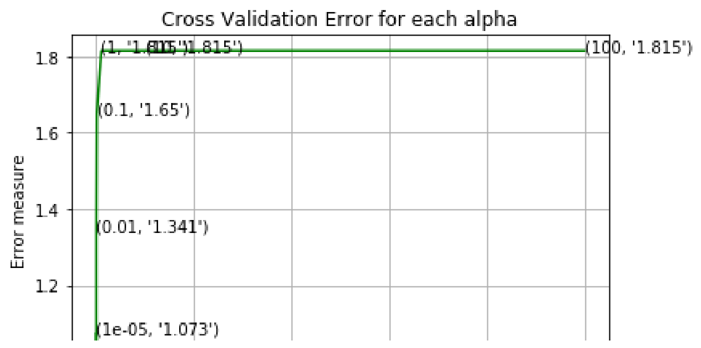
4.4.1. Hyper paramter tuning

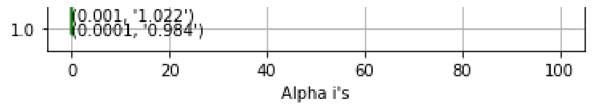
In [198]:

```
# 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, t
ol=0.001,
# cache_size=200, class_weight=None, verbose=False, max_iter=-1, decision_function_shape='ovr', ra
ndom_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', r
andom_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=clf.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_lo
ss(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, p
redict_y, labels=clf.classes_, eps=1e-15))
```

```
for C = 1e-05
Log Loss : 1.0733780464672438
for C = 0.0001
Log Loss : 0.9841423987277579
for C = 0.001
Log Loss : 1.0216950641892408
for C = 0.01
Log Loss : 1.3414560612928248
for C = 0.1
Log Loss : 1.6499583428917914
for C = 1
Log Loss : 1.8154700078965078
for C = 10
Log Loss : 1.8154713724198155
for C = 100
Log Loss : 1.8154297983762158
```





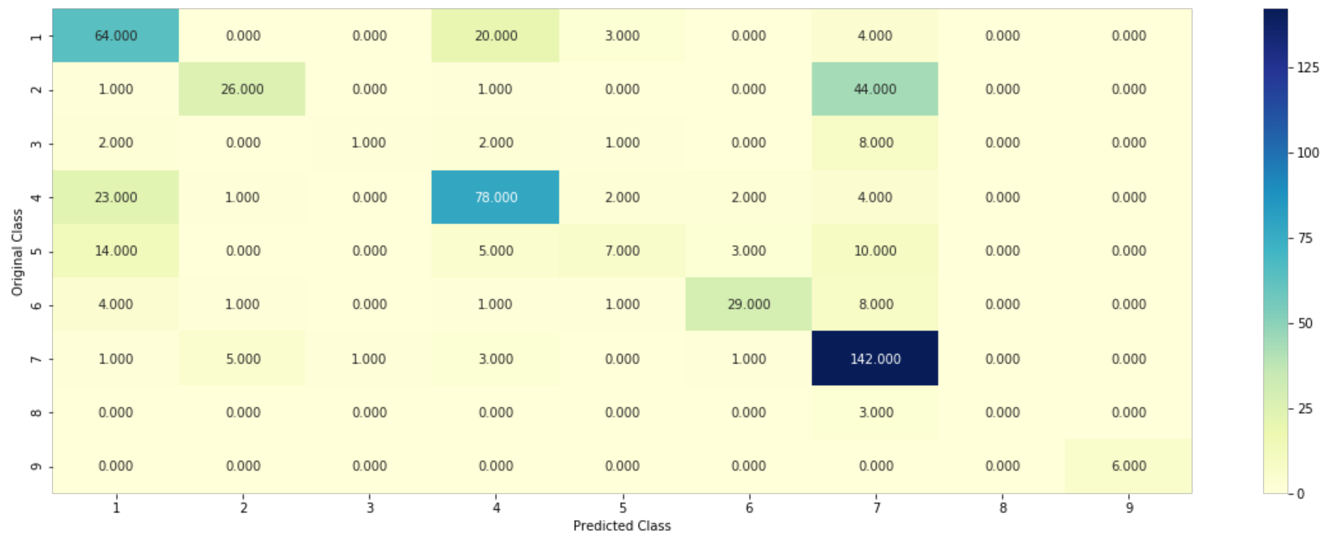
```
For values of best alpha =  0.0001 The train log loss is: 0.4024502206406451
For values of best alpha =  0.0001 The cross validation log loss is: 0.9841423987277579
For values of best alpha =  0.0001 The test log loss is: 0.9832928032626601
```

4.4.2. Testing model with best hyper parameters

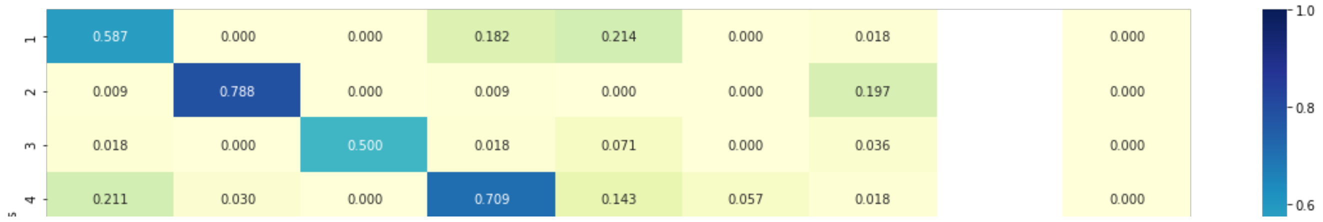
In [199]:

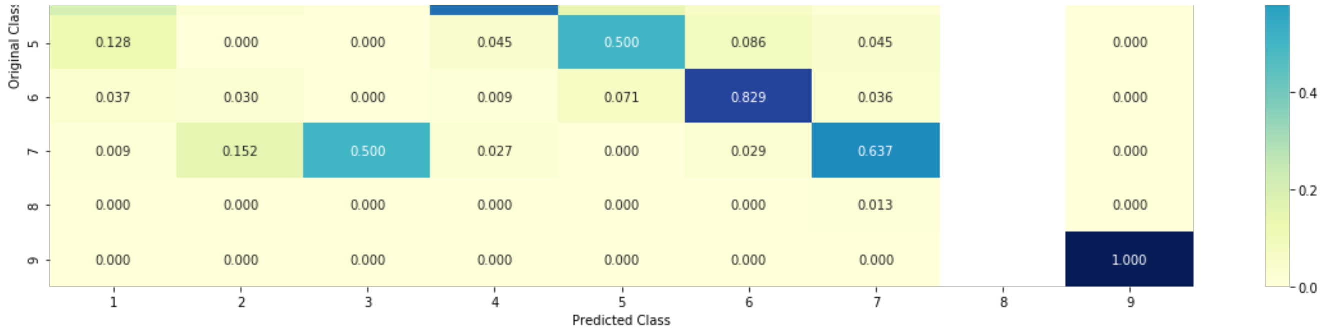
```
# read more about support vector machines with linear kernals here http://scikit-
learn.org/stable/modules/generated/sklearn.svm.SVC.html

# --------------------------------
# default parameters
# SVC(C=1.0, kernel='rbf', degree=3, gamma='auto', coef0=0.0, shrinking=True, probability=False, t
ol=0.001,
# cache_size=200, class_weight=None, verbose=False, max_iter=-1, decision_function_shape='ovr', ra
ndom_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 : 0.9841423987277579
Number of mis-classified points : 0.33646616541353386
-------------------- Confusion matrix --------------------
```

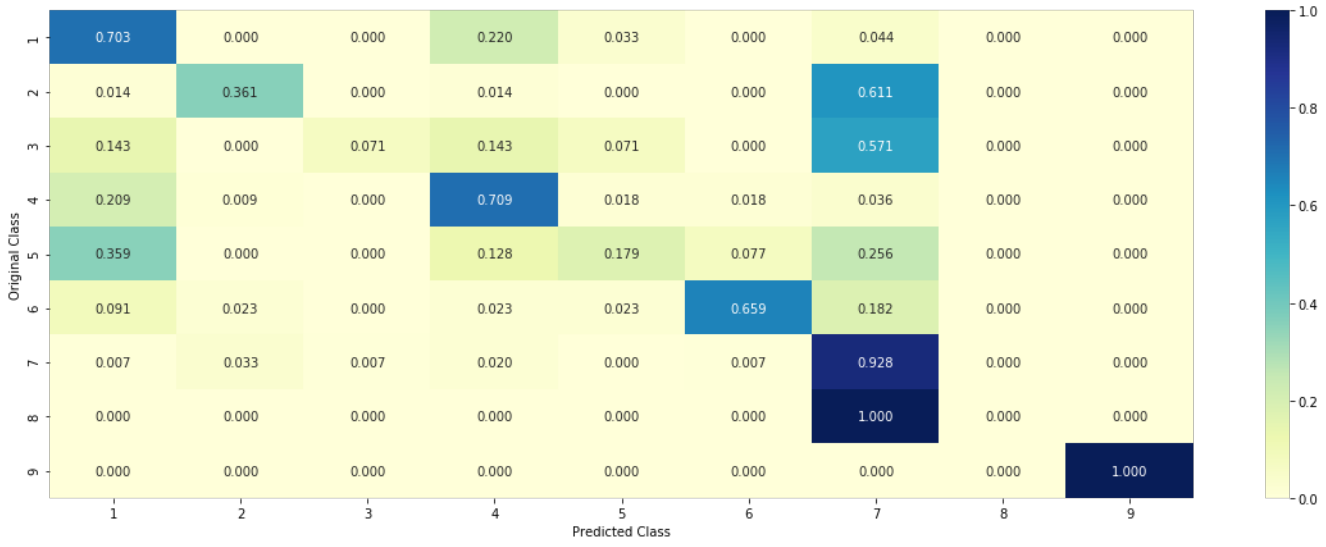


```
-------------------- Precision matrix (Columm Sum=1) --------------------
```





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



4.3.3. Feature Importance

4.3.3.1. For Correctly classified point

In [200]:

```
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 : 4
Predicted Class Probabilities: [[0.2711 0.0142 0.0027 0.6678 0.0154 0.0132 0.0119 0.003  0.0007]]
Actual Class : 4
--------------------------------------------------
147 Text feature [102] present in test data point [True]
Out of the top  500  features  1 are present in query point
```

4.3.3.2. For Incorrectly classified point

In [201]:

```
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: [[0.0514 0.018  0.0794 0.0279 0.0879 0.016  0.7152 0.0029 0.0013]]
Actual Class : 7
--------------------------------------------------
Out of the top  500  features  0 are present in query point
```

4.5 Random Forest Classifier

4.5.1. Hyper paramter tuning (With One hot Encoding)

In [202]:

```
# --------------------------------
# default parameters
# sklearn.ensemble.RandomForestClassifier(n_estimators=10, criterion='gini', max_depth=None, min_s
amples_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-fores
t-and-their-construction-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_alpha%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))
```

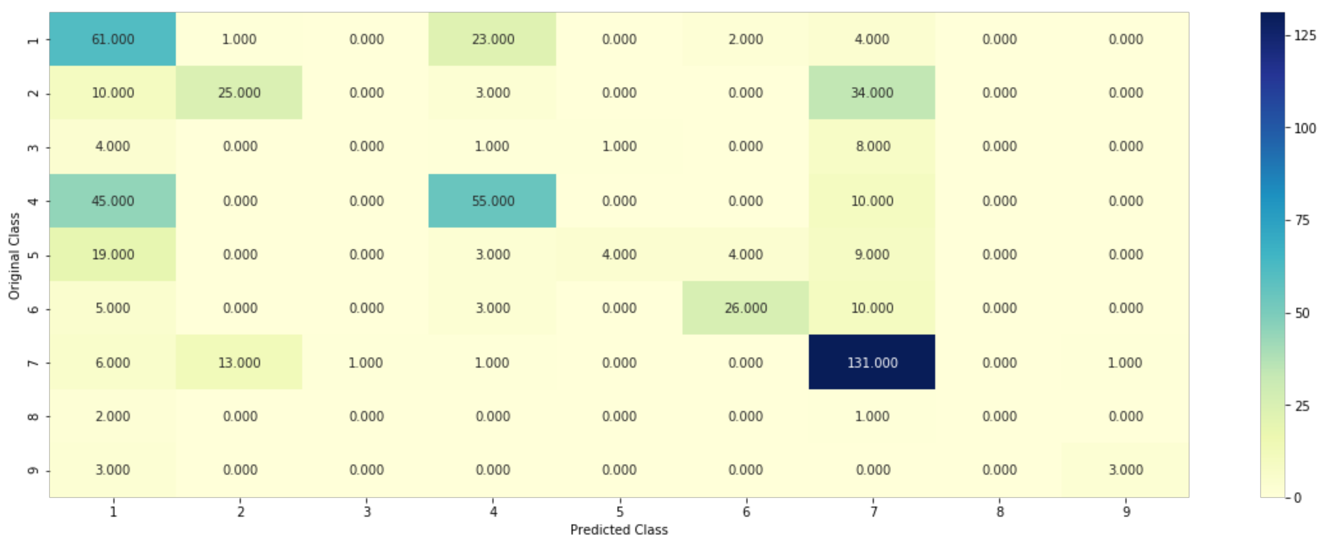
```
for n_estimators = 100 and max depth =  5
Log Loss : 1.196095648267023
for n_estimators = 100 and max depth =  10
Log Loss : 1.2325906900331034
for n_estimators = 200 and max depth =  5
Log Loss : 1.1817437567322706
for n_estimators = 200 and max depth =  10
Log Loss : 1.2252066578261676
for n_estimators = 500 and max depth =  5
Log Loss : 1.1748989562866288
for n_estimators = 500 and max depth =  10
Log Loss : 1.2171417071479138
for n_estimators = 1000 and max depth =  5
Log Loss : 1.1730079549662173
for n_estimators = 1000 and max depth =  10
Log Loss : 1.2204854925462258
for n_estimators = 2000 and max depth =  5
Log Loss : 1.1739907780695893
for n_estimators = 2000 and max depth =  10
Log Loss : 1.2212293677517425
For values of best estimator =  1000 The train log loss is: 0.859333043662656
For values of best estimator =  1000 The cross validation log loss is: 1.1730079549662173
For values of best estimator =  1000 The test log loss is: 1.1496357593253215
```

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

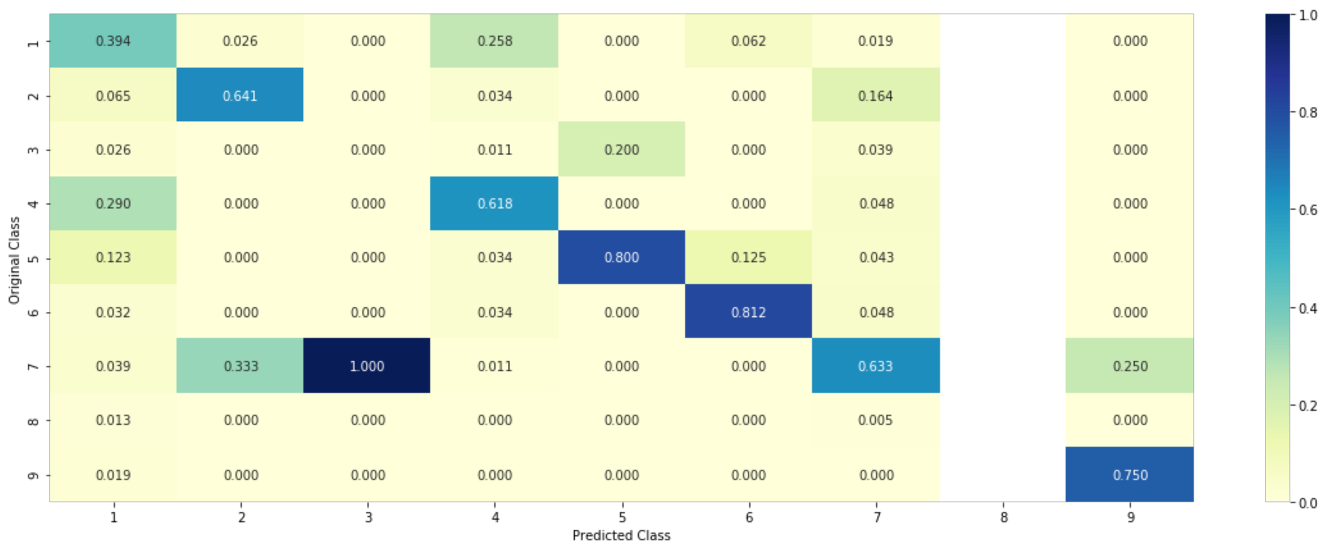
In [203]:

```
# --------------------------------
# default parameters
# sklearn.ensemble.RandomForestClassifier(n_estimators=10, criterion='gini', max_depth=None, min_s
amples_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-fores
t-and-their-construction-2/
# --------------------------------

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

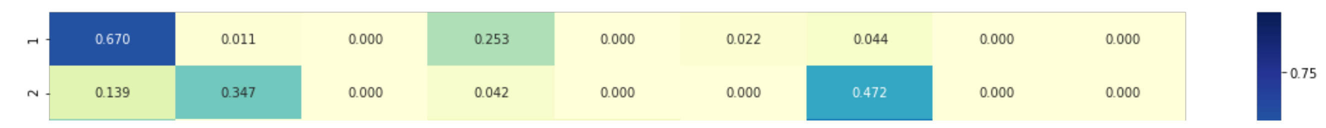
```
Log loss : 1.1730079549662173
Number of mis-classified points : 0.42669172932330 83
-------------------- Confusion matrix --------------------
```

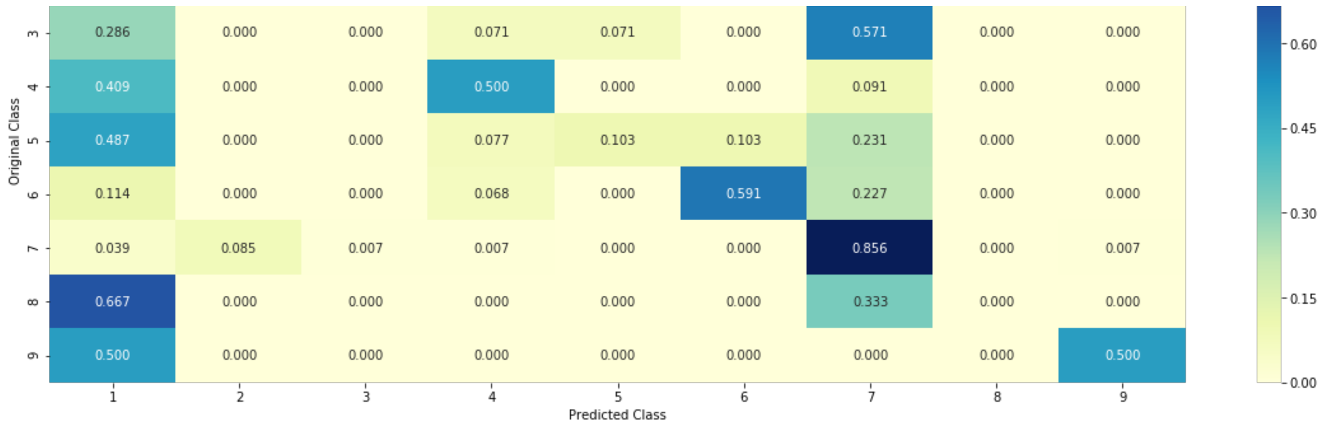


```
-------------------- Precision matrix (Columm Sum=1) --------------------
```



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





4.5.3. Feature Importance

4.5.3.1. Correctly Classified point

In [204]:

```
# test_point_index = 10
clf = RandomForestClassifier(n_estimators=alpha[int(best_alpha/2)], criterion='gini', max_depth=max
_depth[int(best_alpha%2)], random_state=42, n_jobs=-1)
clf.fit(train_x_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_index],test_df['Variation'].iloc[test_point_index], no_feature)
```

```
Predicted Class : 1
Predicted Class Probabilities: [[0.3544 0.0395 0.0113 0.3471 0.0473 0.036  0.0486 0.0075 0.1082]]
Actual Class : 4
--------------------------------------------------
Out of the top  100  features  0 are present in query point
```

4.5.3.2. Inorrectly Classified point

In [205]:

```
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_index],test_df['Variation'].iloc[test_point_index], no_feature)
```

```
Predicted Class : 7
Predicted Class Probabilities: [[0.0305 0.1194 0.039  0.0238 0.0471 0.0309 0.7027 0.0057 0.001 ]]
Actuall Class : 7
--------------------------------------------------
77 Text feature [05] 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 [206]:

```
# --------------------------------
# default parameters
# sklearn.ensemble.RandomForestClassifier(n_estimators=10, criterion='gini', max_depth=None, min_s
amples_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-fores
t-and-their-construction-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_alpha%4)], random_state=42, n_jobs=-1)
clf.fit(train_x_responseCoding, train_y)
sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
sig_clf.fit(train_x_responseCoding, train_y)

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

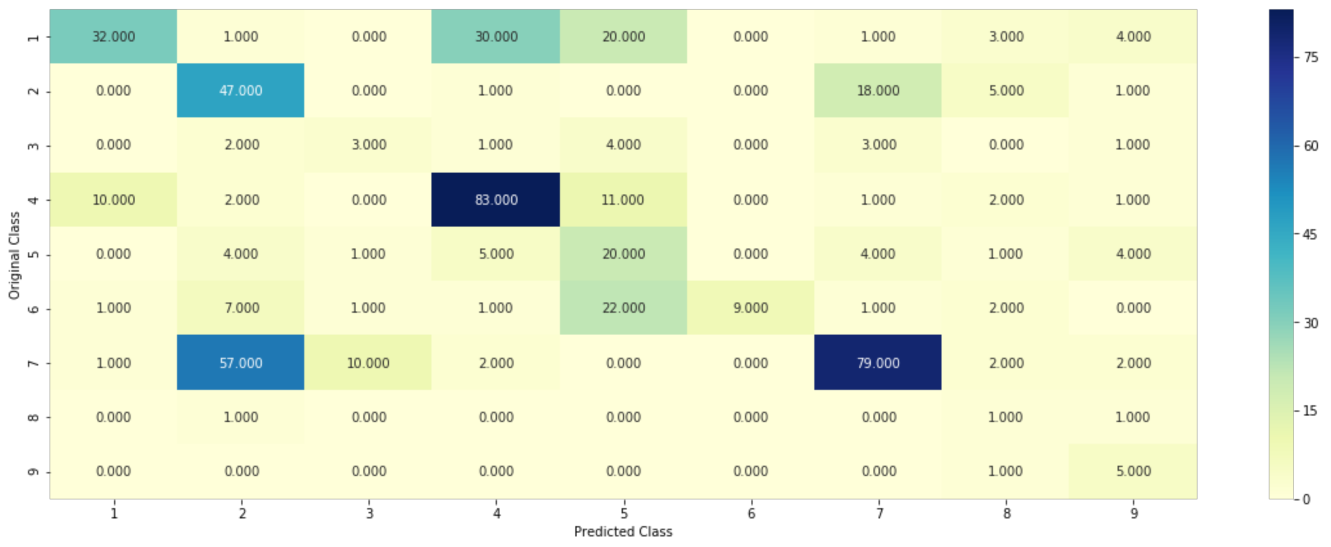
```
for n_estimators = 10 and max depth =  2
Log Loss : 2.03035683239261
for n_estimators = 10 and max depth =  3
Log Loss : 1.6194594057703353
for n_estimators = 10 and max depth =  5
Log Loss : 1.6833536115795273
for n_estimators = 10 and max depth =  10
Log Loss : 2.067296382059495
for n_estimators = 50 and max depth =  2
Log Loss : 1.5803859785347314
for n_estimators = 50 and max depth =  3
Log Loss : 1.3391128025121888
for n_estimators = 50 and max depth =  5
Log Loss : 1.3880914915951748
for n_estimators = 50 and max depth =  10
Log Loss : 1.7669747086668453
for n_estimators = 100 and max depth =  2
Log Loss : 1.415226946274109
for n_estimators = 100 and max depth =  3
Log Loss : 1.4026324150463212
for n_estimators = 100 and max depth =  5
Log Loss : 1.403542030560108
for n_estimators = 100 and max depth =  10
Log Loss : 1.7044552375282755
for n_estimators = 200 and max depth =  2
Log Loss : 1.5432450615444422
for n_estimators = 200 and max depth =  3
Log Loss : 1.3966298657070433
for n_estimators = 200 and max depth =  5
Log Loss : 1.4027043654586553
for n_estimators = 200 and max depth =  10
Log Loss : 1.6035435981695099
for n_estimators = 500 and max depth =  2
Log Loss : 1.586833776198398
for n_estimators = 500 and max depth =  3
Log Loss : 1.483309141403645
for n_estimators = 500 and max depth =  5
Log Loss : 1.391909855794367
for n_estimators = 500 and max depth =  10
Log Loss : 1.648909789778913
for n_estimators = 1000 and max depth =  2
Log Loss : 1.5445620979517145
for n_estimators = 1000 and max depth =  3
Log Loss : 1.46131018316094
for n_estimators = 1000 and max depth =  5
Log Loss : 1.3781247324800316
for n_estimators = 1000 and max depth =  10
Log Loss : 1.630888453271481
For values of best alpha =  50 The train log loss is: 0.15110503748933063
For values of best alpha =  50 The cross validation log loss is: 1.3391128025121888
For values of best alpha =  50 The test log loss is: 1.2641513380625646
```

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

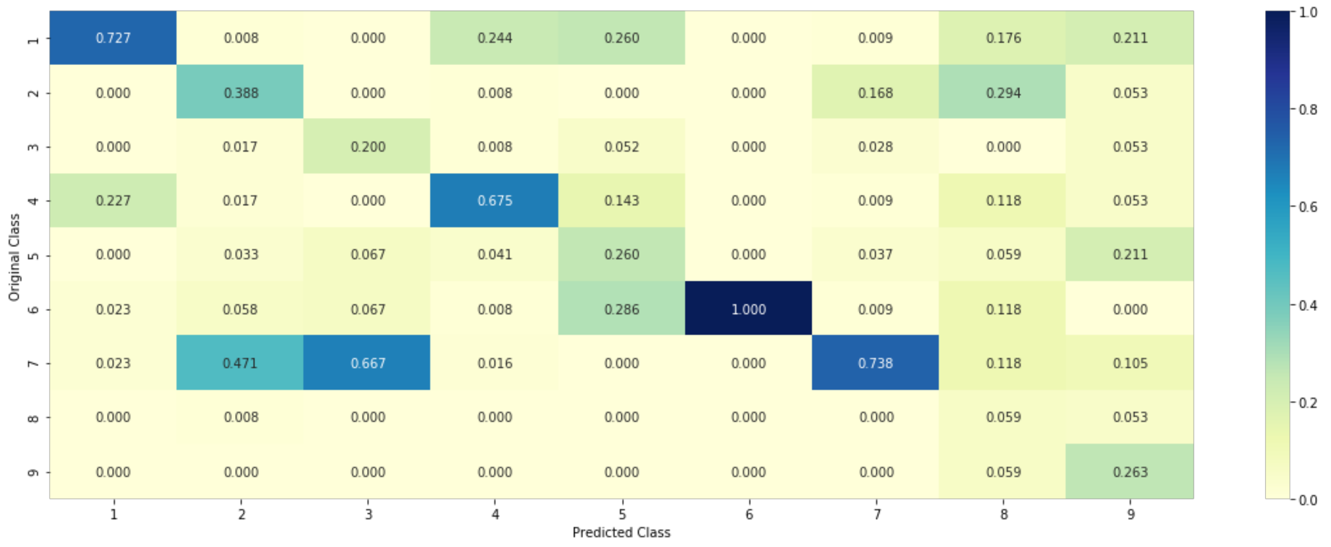
In [207]:

```
# --------------------------------
# default parameters
# sklearn.ensemble.RandomForestClassifier(n_estimators=10, criterion='gini', max_depth=None, min_s
amples_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-fores
t-and-their-construction-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)
```

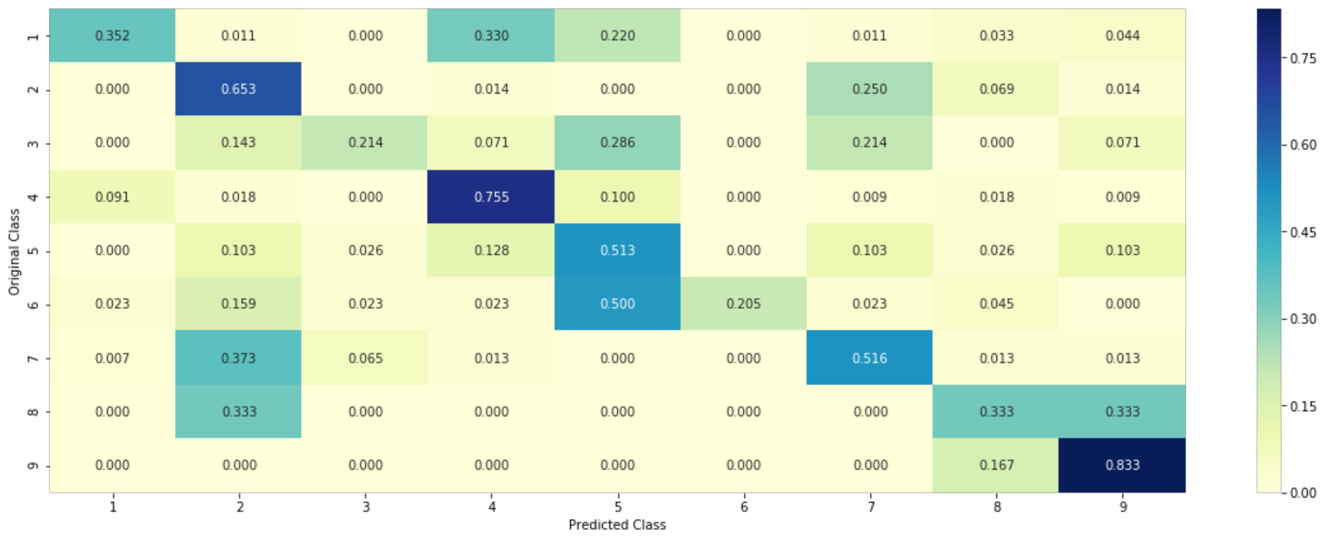
```
Log loss : 1.339112802512189
Number of mis-classified points : 0.4755639097744361
-------------------- Confusion matrix --------------------
```



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



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



4.5.5. Feature Importance

4.5.5.1. Correctly Classified point

In [208]:

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


test_point_index = 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 : 4
Predicted Class Probabilities: [[0.1629 0.0043 0.0474 0.6561 0.0533 0.0321 0.0009 0.0156 0.0274]]
Actual Class : 4
--------------------------------------------------
Variation is important feature
Variation is important feature
Variation is important feature
Gene is important feature
Text is important feature
Variation is important feature
Variation is important feature
Text is important feature
Text is important feature
Variation is important feature
Text is important feature
Gene is important feature
Text is important feature
Gene is important feature
Gene is important feature
Gene is important feature
Variation is important feature
```

```
Text is important feature
Variation is important feature
Text is important feature
Variation is important feature
Gene is important feature
Text is important feature
Gene is important feature
Gene is important feature
Text is important feature
Gene is important feature
```

4.5.5.2. Incorrectly Classified point

In [209]:

```
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("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: [[0.0077 0.1605 0.2558 0.0076 0.0165 0.0226 0.449  0.0681 0.0123]]
Actual Class : 7
--------------------------------------------------
Variation is important feature
Variation is important feature
Variation is important feature
Gene is important feature
Text is important feature
Variation is important feature
Variation is important feature
Text is important feature
Text is important feature
Variation is important feature
Text is important feature
Gene is important feature
Text is important feature
Gene is important feature
Gene is important feature
Gene is important feature
Variation is important feature
Text is important feature
Variation is important feature
Text is important feature
Variation is important feature
Gene is important feature
Text is important feature
Gene is important feature
Gene is important feature
Text is important feature
Gene is important feature
```

4.7 Stack the models

4.7.1 testing with hyper parameter tuning

In [210]:

```
# 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_i
ter=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-in
tuition-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, t
ol=0.001,
# cache_size=200, class_weight=None, verbose=False, max_iter=-1, decision_function_shape='ovr', ra
ndom_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_s
amples_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-fores
t-and-their-construction-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_onehot
Coding))))
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_p
robas=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, sc
lf.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
```

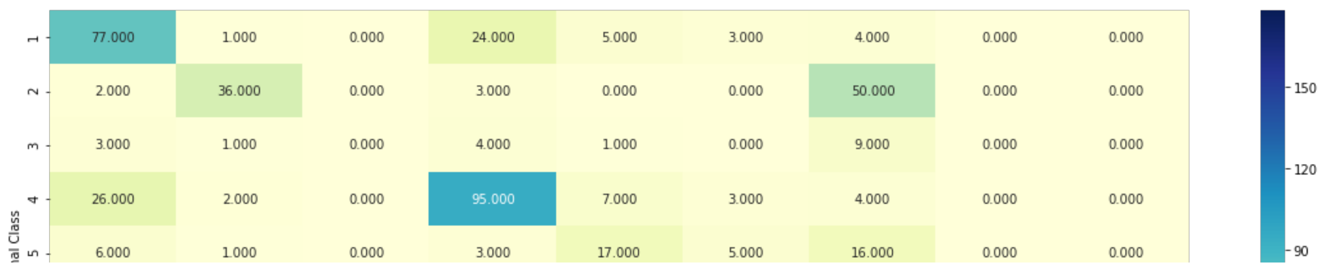
```
Logistic Regression :  Log Loss: 0.99
Support vector machines : Log Loss: 1.82
Naive Bayes : Log Loss: 1.15
--------------------------------------------------
Stacking Classifer : for the value of alpha: 0.000100 Log Loss: 2.178
Stacking Classifer : for the value of alpha: 0.001000 Log Loss: 2.033
Stacking Classifer : for the value of alpha: 0.010000 Log Loss: 1.498
Stacking Classifer : for the value of alpha: 0.100000 Log Loss: 1.128
Stacking Classifer : for the value of alpha: 1.000000 Log Loss: 1.304
Stacking Classifer : for the value of alpha: 10.000000 Log Loss: 1.744
```

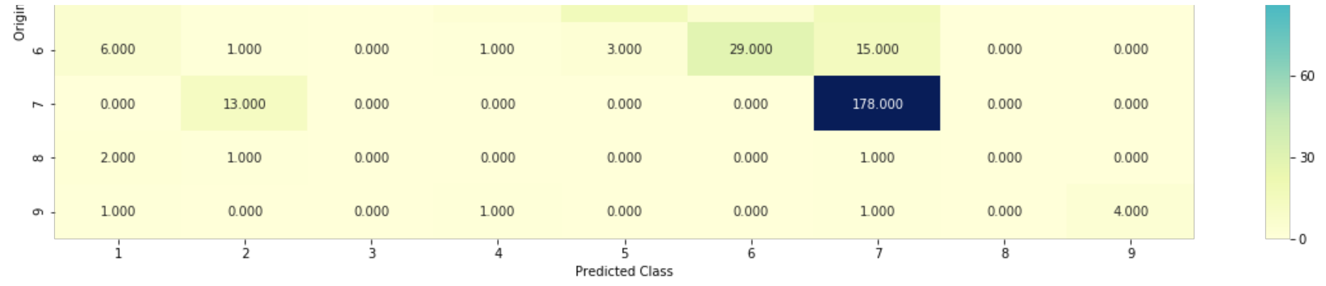
4.7.2 testing the model with the best hyper parameters

In [211]:

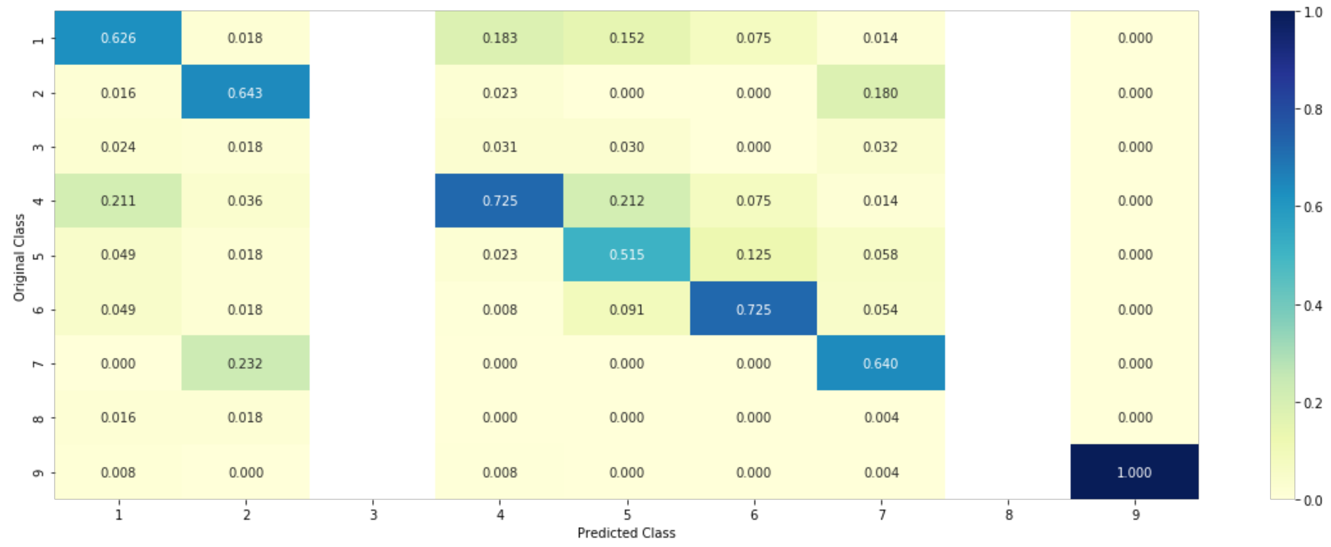
```
lr = LogisticRegression(C=0.1)
sclf = StackingClassifier(classifiers=[sig_clf1, sig_clf2, sig_clf3], meta_classifier=lr, use_proba
s=True)
sclf.fit(train_x_onehotCoding, train_y)

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 : 0.5379823116899818
Log loss (CV) on the stacking classifier : 1.1281918977644925
Log loss (test) on the stacking classifier : 1.1113953511653152
Number of missclassified point : 0.3443609022556391
-------------------- Confusion matrix --------------------
```

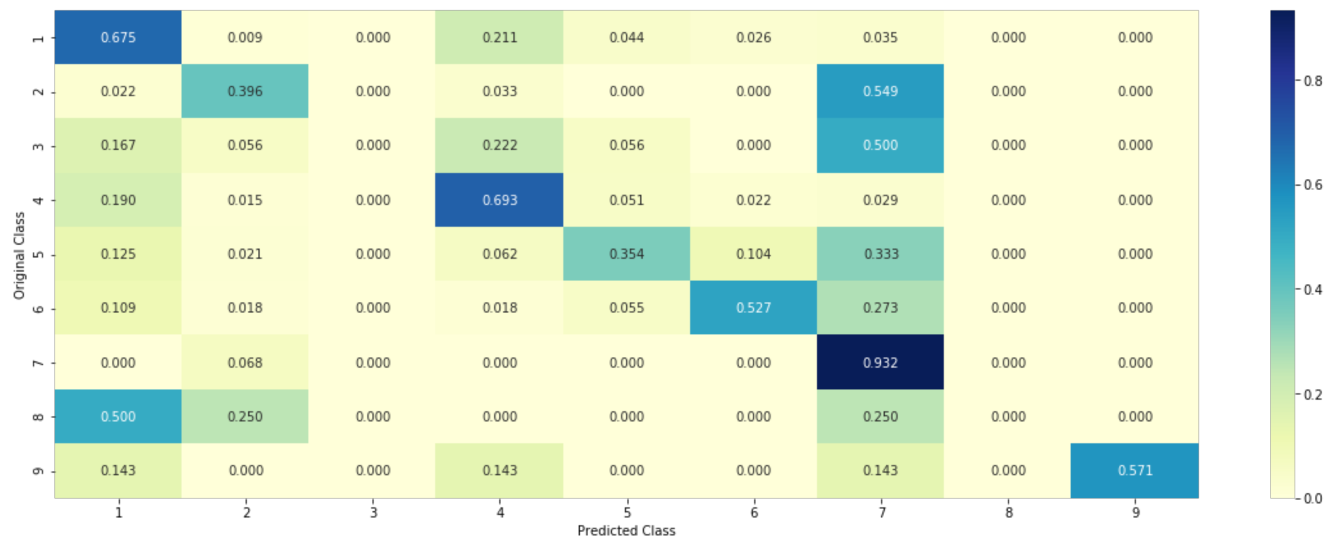




```
-------------------- Precision matrix (Columm Sum=1) --------------------
```



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



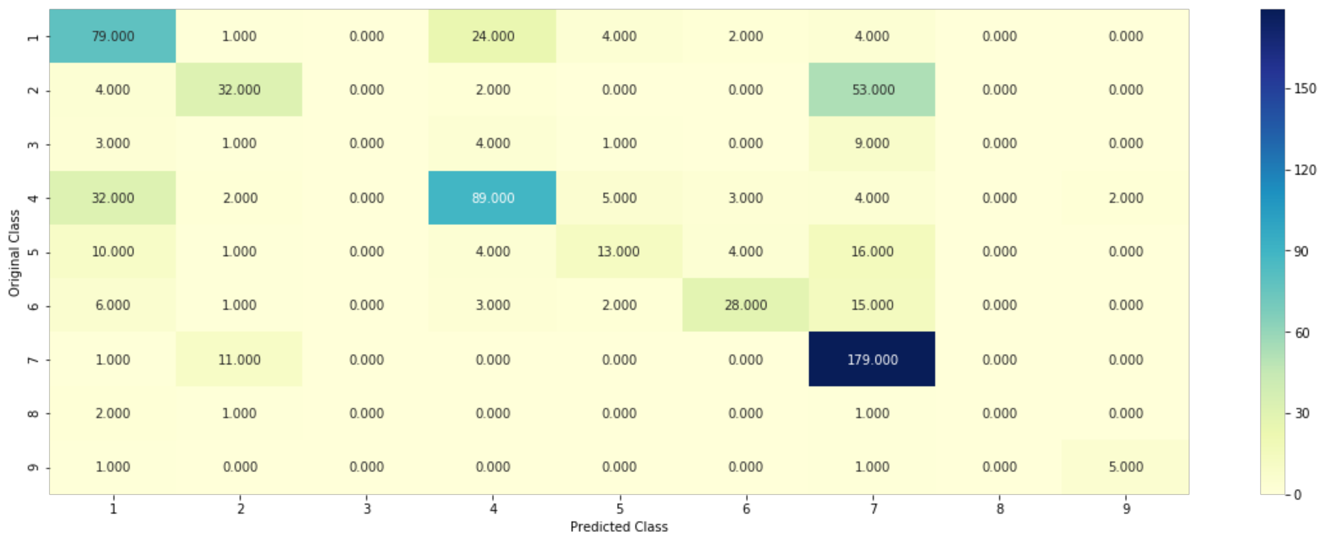
4.7.3 Maximum Voting classifier

In [212]:

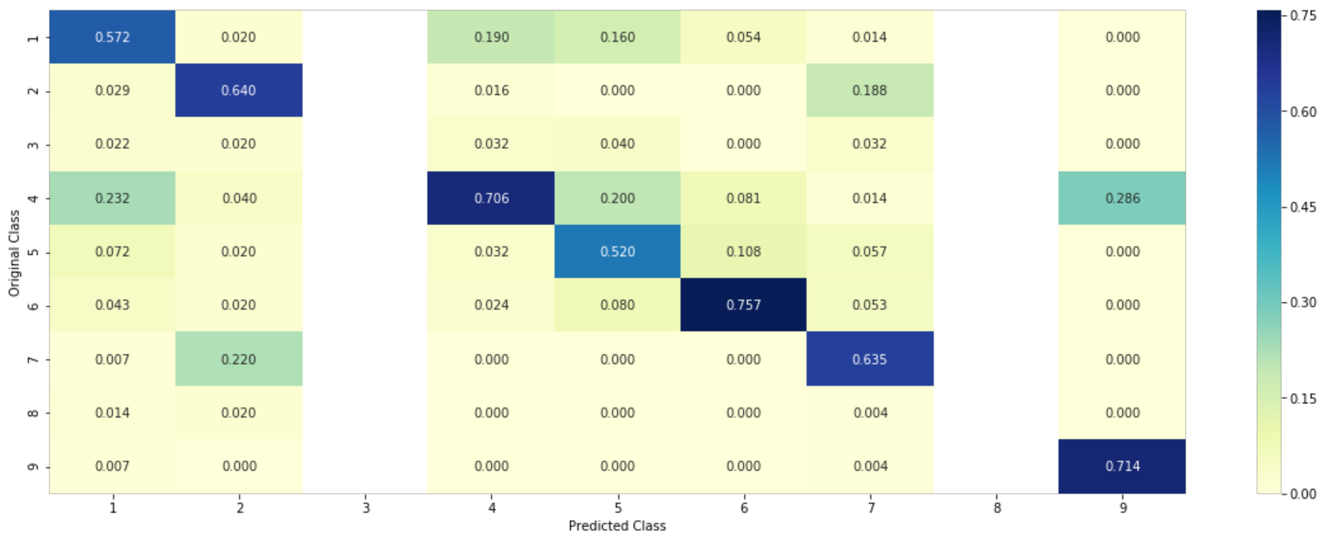
```
#Refer:http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.VotingClassifier.html
from sklearn.ensemble import VotingClassifier
vclf = VotingClassifier(estimators=[('lr', sig_clf1), ('svc', sig_clf2), ('rf', sig_clf3)], voting=
'soft')
vclf.fit(train_x_onehotCoding, train_y)
print("Log 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))
```

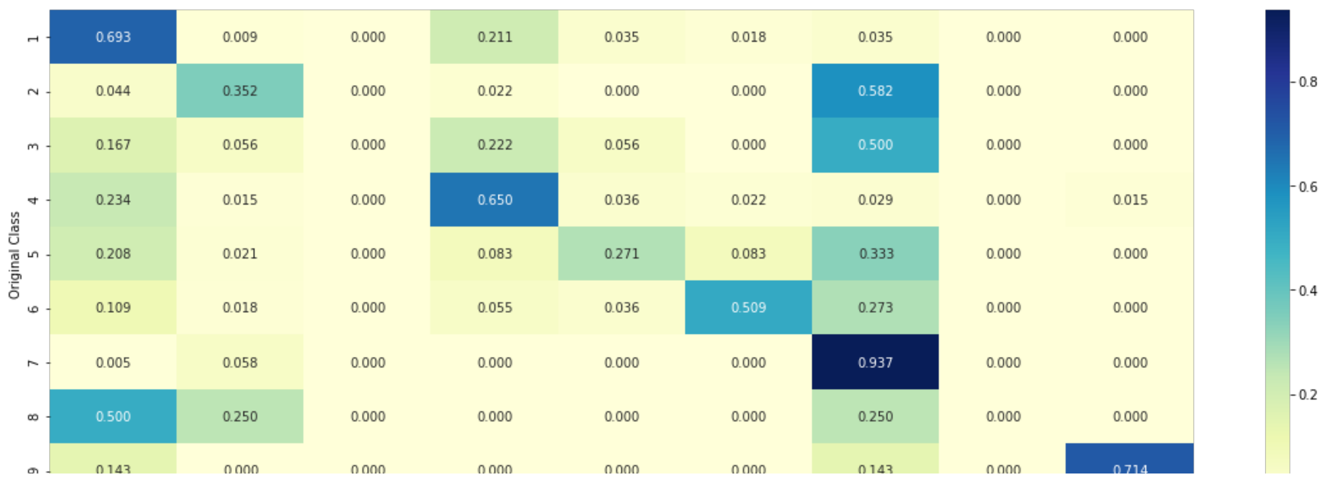
```
Log loss (train) on the VotingClassifier : 0.840481962729594
Log loss (CV) on the VotingClassifier : 1.1697973152902335
Log loss (test) on the VotingClassifier : 1.1664875309846314
Number of missclassified point : 0.3609022556390977
-------------------- Confusion matrix --------------------
```

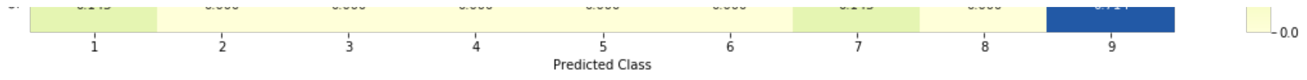


```
-------------------- Precision matrix (Columm Sum=1) --------------------
```



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





5. Assignments

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

In [213]:

```
print("Values we get when we use CountVectorizer")
from prettytable import PrettyTable
ptable = PrettyTable()
ptable.title = "*** Model Summary *** [Performance Metric: Log-Loss]"
ptable.field_names=["Model Name","Train","CV","Test","% Misclassified Points"]
ptable.add_row(["Naive Bayes","0.92","1.27","1.19","39"])
ptable.add_row(["KNN","0.65","1.007","1.02","37"])
ptable.add_row(["Logistic Regression With Class balancing","0.56","1.15","1.01","35"])
ptable.add_row(["Logistic Regression Without Class balancing","0.55","1.20","1.01","34"])
ptable.add_row(["Linear SVM","0.57","1.17","1.07","34"])
ptable.add_row(["Random Forest Classifier With One hot Encoding","0.72","1.16","1.12","38"])
ptable.add_row(["Random Forest Classifier With Response Coding","0.04","1.38","1.31","47"])
ptable.add_row(["Stack Models:LR+NB+SVM","0.15","1.15","1.07","34"])
ptable.add_row(["Maximum Voting classifier","0.90","1.22","1.17","35"])
print(ptable)
print()
```

```
Values we get when we use CountVectorizer
+-------------------------------------------------+-------+-------+------+------------------------+
|                    Model Name                   | Train |   CV  | Test | % Misclassified Points |
+-------------------------------------------------+-------+-------+------+------------------------+
|                   Naive Bayes                   |  0.92 |  1.27 | 1.19 |           39           |
|                       KNN                       |  0.65 | 1.007 | 1.02 |           37           |
|     Logistic Regression With Class balancing    |  0.56 |  1.15 | 1.01 |           35           |
|   Logistic Regression Without Class balancing   |  0.55 |  1.20 | 1.01 |           34           |
|                    Linear SVM                   |  0.57 |  1.17 | 1.07 |           34           |
| Random Forest Classifier With One hot Encoding  |  0.72 |  1.16 | 1.12 |           38           |
| Random Forest Classifier With Response Coding   |  0.04 |  1.38 | 1.31 |           47           |
|              Stack Models:LR+NB+SVM             |  0.15 |  1.15 | 1.07 |           34           |
|            Maximum Voting classifier            |  0.90 |  1.22 | 1.17 |           35           |
+-------------------------------------------------+-------+-------+------+------------------------+
```

In [239]:

```
print("Values we get when we use TfidfVectorizer with 1000 features")
from prettytable import PrettyTable
ptable = PrettyTable()
ptable.title = "*** Model Summary *** [Performance Metric: Log-Loss]"
ptable.field_names=["Model Name","Train","CV","Test","% Misclassified Points"]
ptable.add_row(["Naive Bayes","0.52","1.15","1.14","38"])
ptable.add_row(["KNN","0.65","1.007","1.02","37"])
ptable.add_row(["Logistic Regression With Class balancing","0.45","0.94","0.94","3"])
ptable.add_row(["Logistic Regression Without Class balancing","0.44","0.98","0.98","33"])
ptable.add_row(["Linear SVM","0.40","0.98","0.99","33"])
ptable.add_row(["Random Forest Classifier With One hot Encoding","0.40","1.17","1.14","42"])
ptable.add_row(["Random Forest Classifier With Response Coding","0.04","1.32","1.36","42"])
ptable.add_row(["Stack Models:LR+NB+SVM","0.53","1.13","1.11","34"])
ptable.add_row(["Maximum Voting classifier","0.84","1.16","1.16","36"])
print(ptable)
print()
```

```
Values we get when we use TfidfVectorizer with 1000 features
+-------------------------------------------------+-------+-------+------+------------------------+
|                    Model Name                   | Train |   CV  | Test | % Misclassified Points |
+-------------------------------------------------+-------+-------+------+------------------------+
|                   Naive Bayes                   |  0.52 |  1.15 | 1.14 |           38           |
|                      KNN                          |  0.65 | 1.007 | 1.02 |           37            |
|    Logistic Regression With Class balancing       |  0.45 |  0.94 | 0.94 |           34            |
|  Logistic Regression Without Class balancing      |  0.44 |  0.98 | 0.98 |           33            |
|                   Linear SVM                      |  0.40 |  0.98 | 0.99 |           33            |
| Random Forest Classifier With One hot Encoding    |  0.40 |  1.17 | 1.14 |           42            |
| Random Forest Classifier With Response Coding     |  0.04 |  1.32 | 1.36 |           42            |
|             Stack Models:LR+NB+SVM                |  0.53 |  1.13 | 1.11 |           34            |
|           Maximum Voting classifier               |  0.84 |  1.16 | 1.16 |           36            |
+---------------------------------------------------+-------+-------+------+-------------------------+
```

BY comparing both the vectorizer we can see that tfidf vectorizer performs better than Count Vectorizer because of low log loss

From the above table we can see that from all model linear SVM performs well Logistic regression without class balancing also gives low loss but for class with less no of point it gives precision and recall 0

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

In [225]:

```
# building a CountVectorizer with all the words that occured minimum 3 times in train data
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 = normalize(test_text_feature_onehotCoding_lr, axis=0)

cv_text_feature_onehotCoding_lr = text_vectorizer_lr.transform(cv_df['TEXT'])
cv_text_feature_onehotCoding = normalize(test_text_feature_onehotCoding_lr, axis=0)
test_text_feature_onehotCoding_lr = text_vectorizer_lr.transform(test_df['TEXT'])

test_text_feature_onehotCoding = normalize(test_text_feature_onehotCoding_lr, axis=0)

# getting all the feature names (words)
train_text_features_lr= text_vectorizer_lr.get_feature_names()

# train_text_feature_onehotCoding.sum(axis=0).A1 will sum every row and returns (1*number of featu
res) vector
train_text_fea_counts_lr = train_text_feature_onehotCoding_lr.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_lr = dict(zip(list(train_text_features_lr),train_text_fea_counts_lr))


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

```
Total number of unique words in train data : 778994
```

ngram_range=(1,2) is for inclding both unigram and bigram https://towardsdatascience.com/learning-nlp-language-models-with-real-data-cdff04c51c25

In [226]:

```
# one-hot encoding of Gene feature.
gene_vectorizer_g = CountVectorizer()
train_gene_feature_onehotCoding_g = gene_vectorizer_g.fit_transform(train_df['Gene'])
test_gene_feature_onehotCoding_g = gene_vectorizer_g.transform(test_df['Gene'])
cv_gene_feature_onehotCoding_g = gene_vectorizer_g.transform(cv_df['Gene'])
```

In [227]:

```
train_df['Gene'].head()
```

Out[227]:

```
1197    PIK3CA
908     PDGFRA
1917       SMO
1767      IDH2
1459     FGFR2
Name: Gene, dtype: object
```

In [228]:

```
print("shape of train: ",train_gene_feature_onehotCoding_g .shape)
```

```
shape of train:  (2124, 231)
```

In [229]:

```
# one-hot encoding of variation feature.
variation_vectorizer_g = CountVectorizer(
train_variation_feature_onehotCoding_g = variation_vectorizer_g.fit_transform(train_df['Variation'
])
test_variation_feature_onehotCoding_g = variation_vectorizer_g.transform(test_df['Variation'])
cv_variation_feature_onehotCoding_g = variation_vectorizer_g.transform(cv_df['Variation'])
```

In [230]:

```
print("shape of train: ",train_variation_feature_onehotCoding_g .shape)
```

```
shape of train:  (2124, 2055)
```

In [231]:

```
train_df['Variation'].head()
```

Out[231]:

```
1197    A1066V
908      D842V
1917     D473G
1767     R172S
1459     V755I
Name: Variation, dtype: object
```

In [232]:

```
train_gene_var_onehotCoding_g=
hstack((train_gene_feature_onehotCoding_g,train_variation_feature_onehotCoding_g))
test_gene_var_onehotCoding_g =
hstack((test_gene_feature_onehotCoding_g,test_variation_feature_onehotCoding_g))
cv_gene_var_onehotCoding_g =
hstack((cv_gene_feature_onehotCoding_g,cv_variation_feature_onehotCoding_g))
```

In [237]:

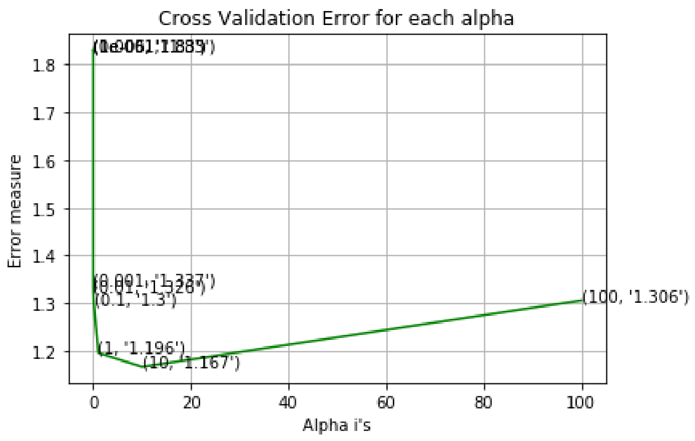
```
train_x_onehotCoding_grams = hstack((train_gene_var_onehotCoding_g,
train_text_feature_onehotCoding)).tocsr()
train_y = np.array(list(train_df['Class']))

test_x_onehotCoding_grams = hstack((test_gene_var_onehotCoding_g, test_text_feature_onehotCoding))
.tocsr()
test_y = np.array(list(test_df['Class']))

cv_x_onehotCoding_grams = hstack((cv_gene_var_onehotCoding_g, cv_text_feature_onehotCoding)).tocsr
()
cv_y = np.array(list(cv_df['Class']))
```

In [234]:

```
# 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_i
ter=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-in
tuition-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_grams, train_y)
    sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
    sig_clf.fit(train_x_onehotCoding_grams, train_y)
    sig_clf_probs = sig_clf.predict_proba(cv_x_onehotCoding_grams)
    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', ran
dom_state=42)
clf.fit(train_x_onehotCoding_grams, train_y)
sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
sig_clf.fit(train_x_onehotCoding_grams, train_y)

predict_y = sig_clf.predict_proba(train_x_onehotCoding_grams)
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_grams)
print('For values of best alpha = ', alpha[best_alpha], "The cross validation log loss is:",log_lo
ss(y_cv, predict_y, labels=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(test_x_onehotCoding_grams)
print('For values of best alpha = ', alpha[best_alpha], "The test log loss is:",log_loss(y_test, p
redict_y, labels=clf.classes_, eps=1e-15))
```

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

```
for alpha = 1e-05
Log Loss : 1.8304997567764278
for alpha = 0.0001
Log Loss : 1.8304997567764278
for alpha = 0.001
Log Loss : 1.337481418851759
for alpha = 0.01
Log Loss : 1.326443920620383
for alpha = 0.1
Log Loss : 1.3001625834812092
for alpha = 1
Log Loss : 1.1963960697052884
for alpha = 10
Log Loss : 1.1669334239369846
for alpha = 100
Log Loss : 1.306068204450498
```



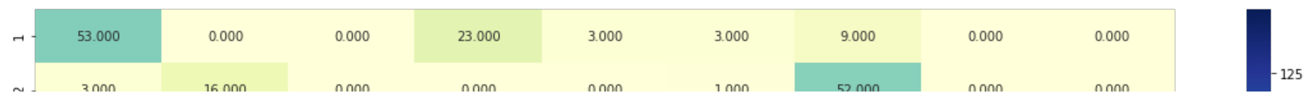
```
For values of best alpha =  10 The train log loss is: 0.8656226649835872
For values of best alpha =  10 The cross validation log loss is: 1.1669334239369846
For values of best alpha =  10 The test log loss is: 1.121948745433954
```

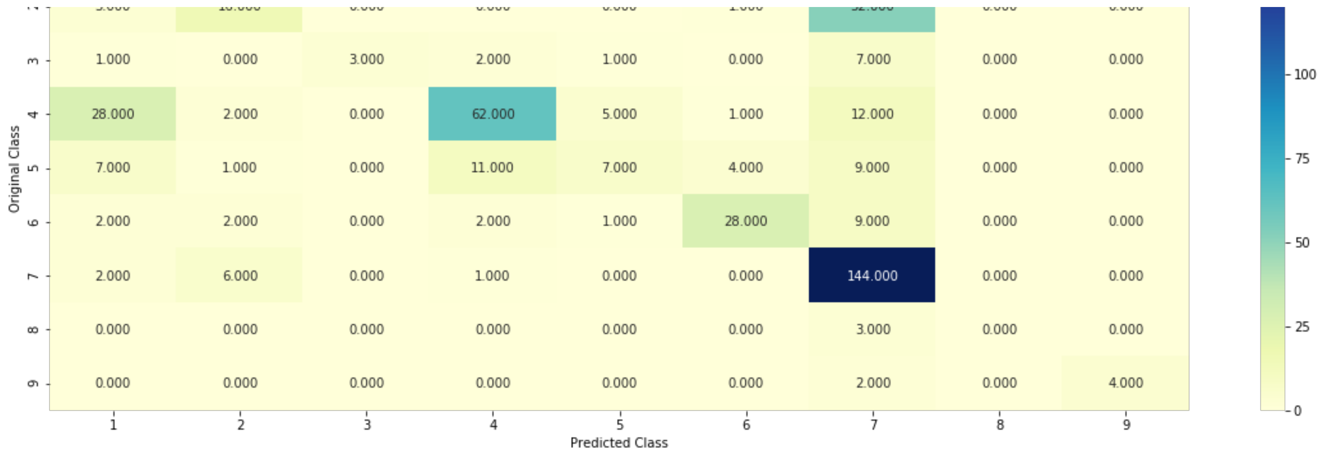
4.3.1.2. Testing the model with best hyper paramters

In [235]:

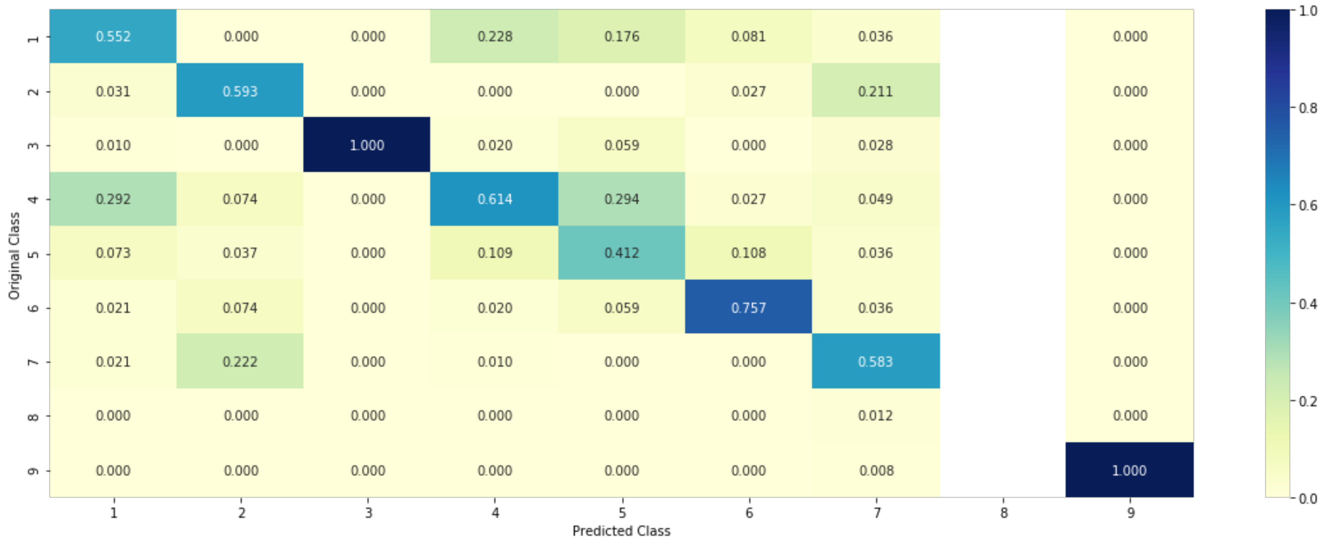
```
# read more about SGDClassifier() at http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html
# ------------------------------
# default parameters
# SGDClassifier(loss='hinge', penalty='l2', alpha=0.0001, l1_ratio=0.15, fit_intercept=True, max_iter=None, tol=None,
# 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_grams, train_y, cv_x_onehotCoding_grams, cv_y, clf)
```

```
Log loss : 1.1669334239369846
Number of mis-classified points : 0.4041353383458647
-------------------- Confusion matrix --------------------
```

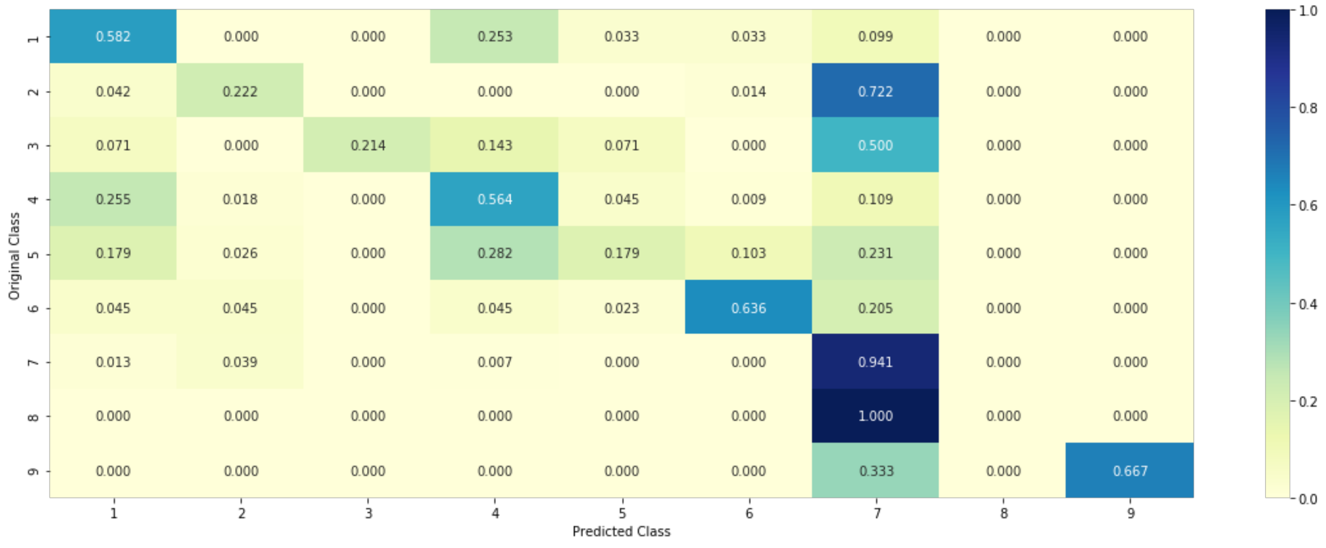




```
-------------------- Precision matrix (Columm Sum=1) --------------------
```



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



We apply logistic regression with balanced parameter on unigrams and bigrams because it performs better than unbalanced class.

Try any of the feature engineering techniques discussed in the course to reduce the CV and test log-loss to a value less than 1.0

We combine gene and variation feature into a single list and try to redue log loss

In [249]:

```
gene_variation = []

for gene in result['Gene'].values:
    gene_variation.append(gene)

for variation in result['Variation'].values:
    gene_variation.append(variation)
```

In [250]:

```
gene_variation[0:5]
```

Out[250]:

```
['FAM58A', 'CBL', 'CBL', 'CBL', 'CBL']
```

4.3.1.3. Feature Importance

In [251]:

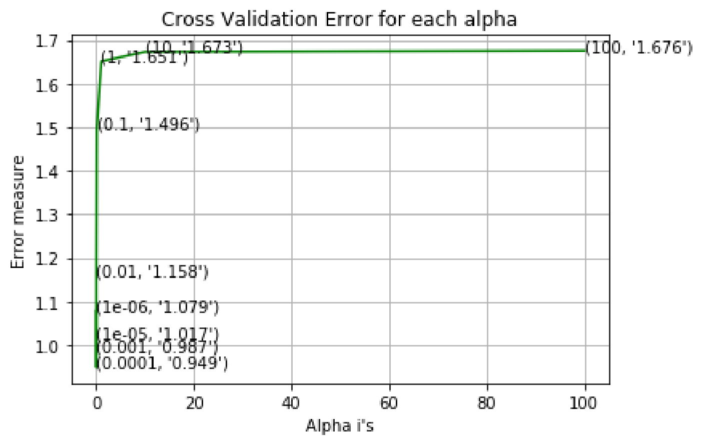
```
tfidfVectorizer = TfidfVectorizer(max_features=1000)
text2 = tfidfVectorizer.fit_transform(gene_variation)
gene_variation_features = tfidfVectorizer.get_feature_names()

train_text = tfidfVectorizer.transform(train_df['TEXT'])
test_text = tfidfVectorizer.transform(test_df['TEXT'])
cv_text = tfidfVectorizer.transform(cv_df['TEXT'])
```

Correctly Classified point

In [223]:

```
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, 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("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', ran
dom_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=clf.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_lo
print('For values of best alpha = ', alpha[best_alpha], "The cross validation log loss is:",log_lo
ss(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, p
redict_y, labels=clf.classes_, eps=1e-15))
```

```
for alpha = 1e-06
Log Loss : 1.0790083006553397
for alpha = 1e-05
Log Loss : 1.0174994205444525
for alpha = 0.0001
Log Loss : 0.9488873032368705
for alpha = 0.001
Log Loss : 0.9869456371544133
for alpha = 0.01
Log Loss : 1.1577315987229353
for alpha = 0.1
Log Loss : 1.496244595577183
for alpha = 1
Log Loss : 1.6510815794163722
for alpha = 10
Log Loss : 1.6729529226677695
for alpha = 100
Log Loss : 1.6755081115158559
```

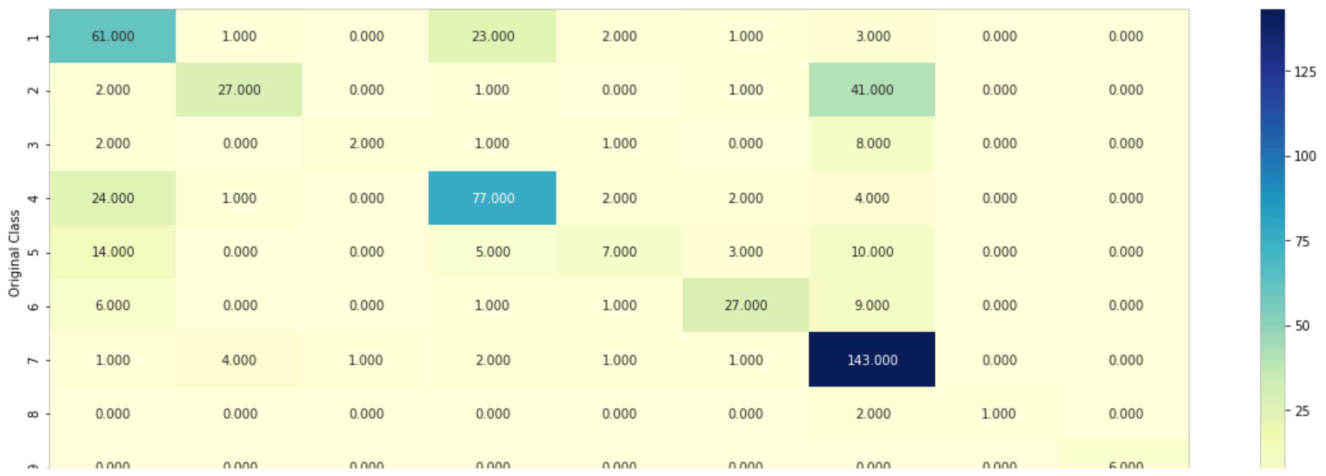


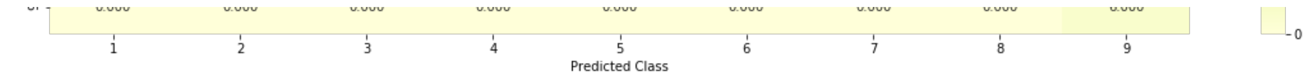
```
For values of best alpha =  0.0001 The train log loss is: 0.4709537778064926
For values of best alpha =  0.0001 The cross validation log loss is: 0.9488873032368705
For values of best alpha =  0.0001 The test log loss is: 0.9420298040730485
```

In [224]:

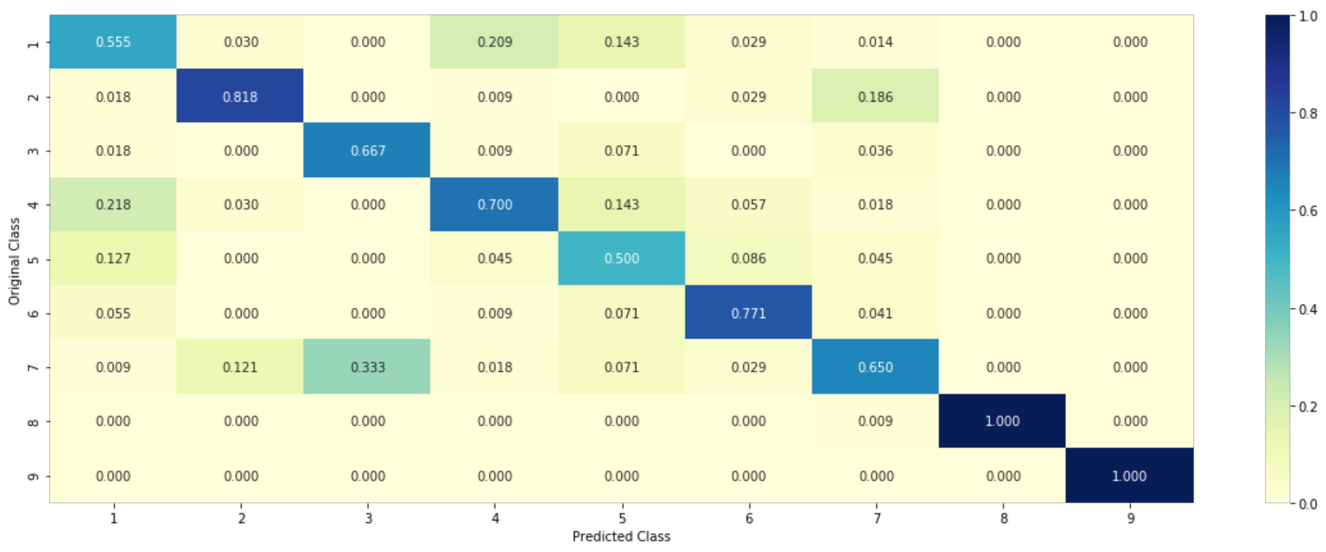
```
clf = SGDClassifier(class_weight='balanced', alpha=alpha[best_alpha], penalty='l2', loss='log', ran
dom_state=42)
predict_and_plot_confusion_matrix(train_x_onehotCoding, train_y, cv_x_onehotCoding, cv_y, clf)
```

```
Log loss : 0.9488873032368705
Number of mis-classified points : 0.34022556390977443
-------------------- Confusion matrix --------------------
```

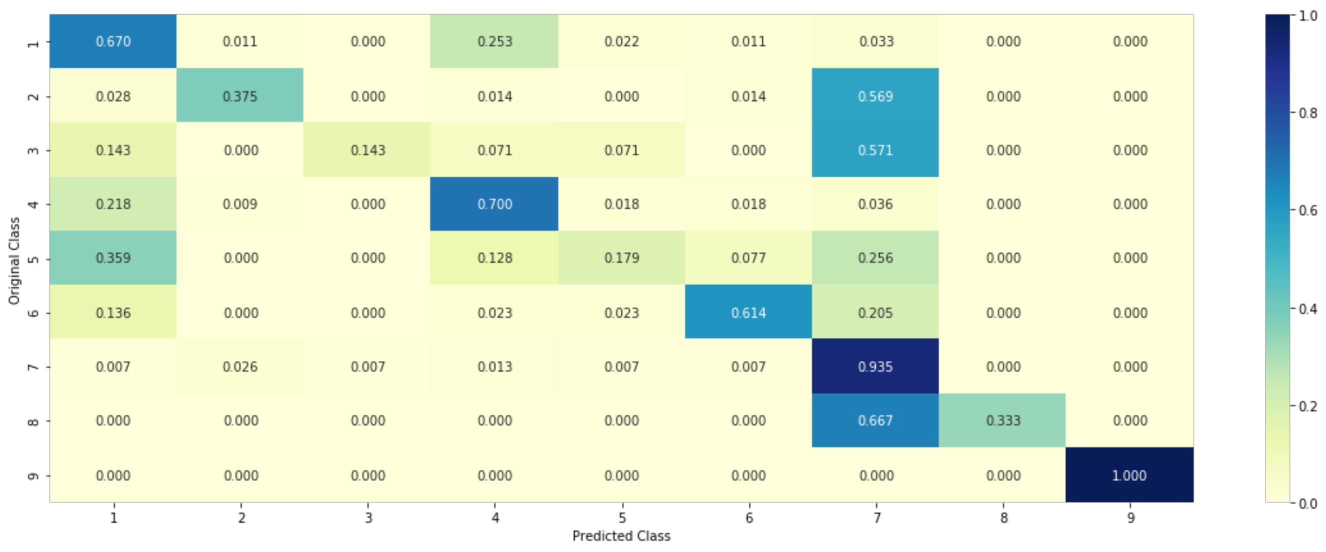




```
-------------------- Precision matrix (Columm Sum=1) --------------------
```



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

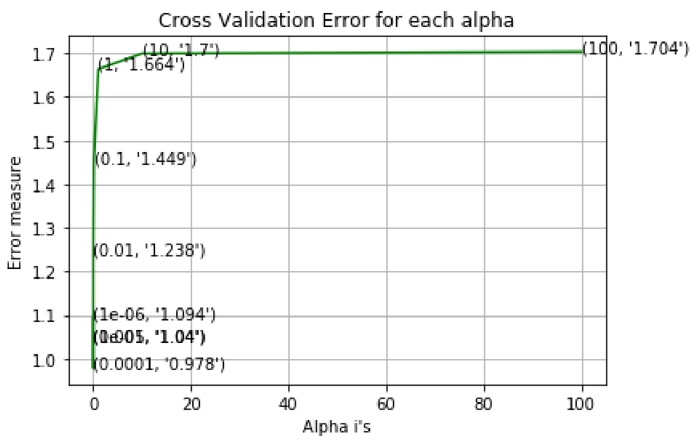


LOgistic regrssion without class imbalancing

In [243]:

```
alpha = [10 ** x for x in range(-6, 3)]
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, 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("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', ran
dom_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=clf.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_lo
ss(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, p
redict_y, labels=clf.classes_, eps=1e-15))
```

```
for alpha = 1e-06
Log Loss : 1.0936112077629152
for alpha = 1e-05
Log Loss : 1.0395284199500179
for alpha = 0.0001
Log Loss : 0.9775604484345919
for alpha = 0.001
Log Loss : 1.040433469439783
for alpha = 0.01
Log Loss : 1.237815868354822
for alpha = 0.1
Log Loss : 1.4488201774456086
for alpha = 1
Log Loss : 1.663856380871813
for alpha = 10
Log Loss : 1.699520644368618
for alpha = 100
Log Loss : 1.703676139187739
```

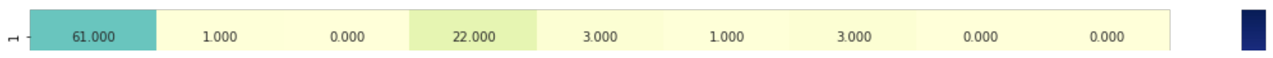


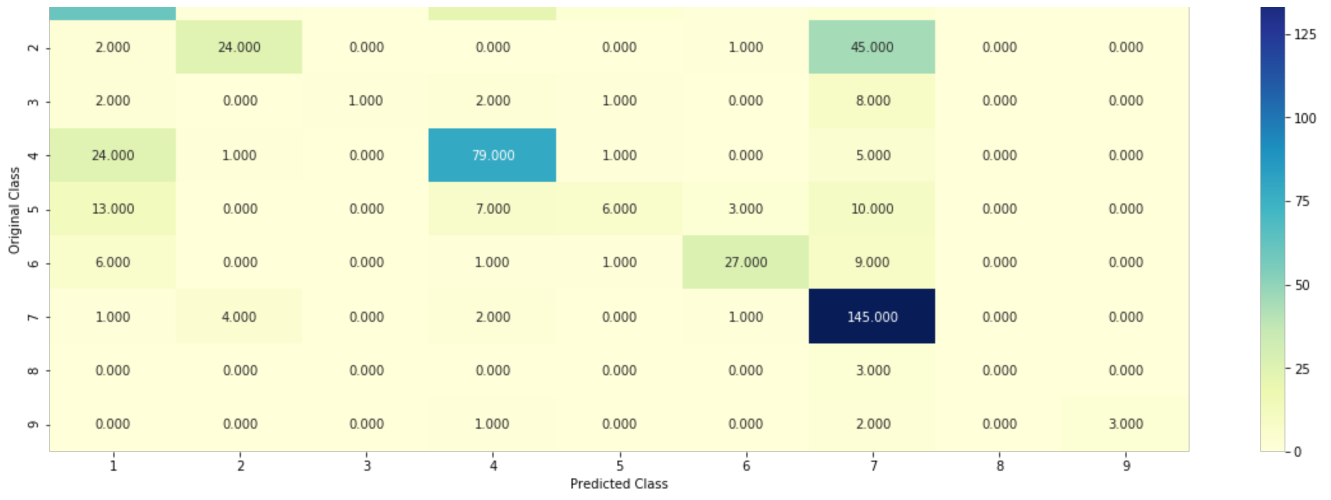
```
For values of best alpha =  0.0001 The train log loss is: 0.4709537778064926
For values of best alpha =  0.0001 The cross validation log loss is: 0.9488873032368705
For values of best alpha =  0.0001 The test log loss is: 0.9420298040730485
```

In [244]:

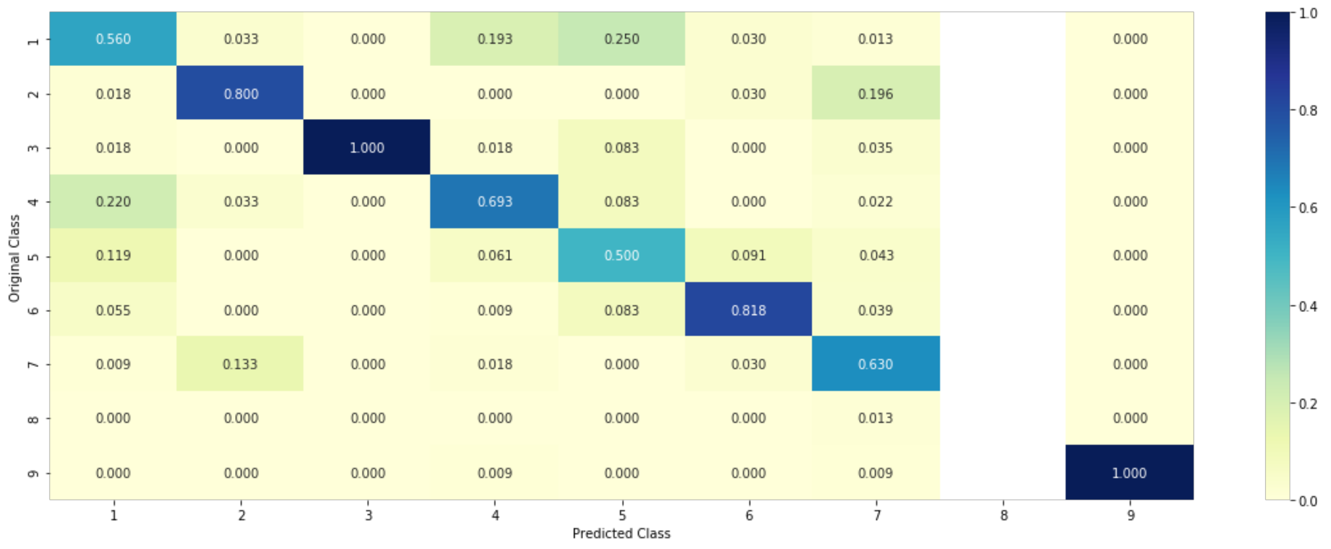
```
clf = SGDClassifier( alpha=alpha[best_alpha], penalty='l2', loss='log', random_state=42)
predict_and_plot_confusion_matrix(train_x_onehotCoding, train_y, cv_x_onehotCoding, cv_y, clf)
```

```
Log loss : 0.9775604484345919
Number of mis-classified points : 0.34962406015037595
-------------------- Confusion matrix --------------------
```

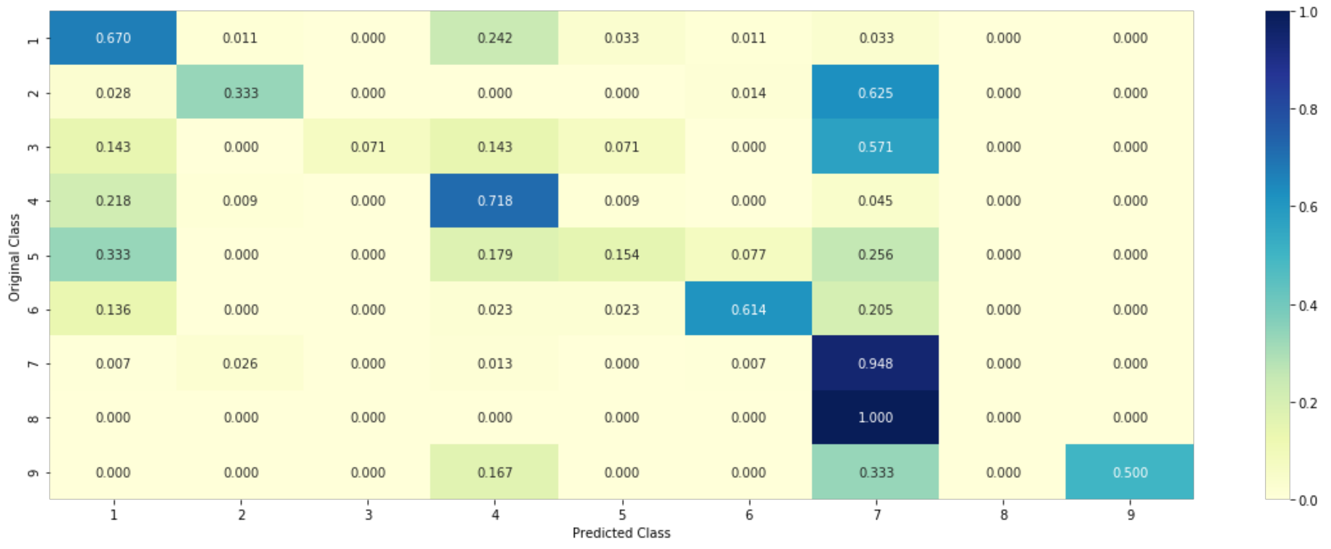




```
-------------------- Precision matrix (Columm Sum=1) --------------------
```



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



In [253]:

```
ptable = PrettyTable()
ptable.field_names=["Model Name","Train","CV","Test","% Misclassified Points"]
ptable.add_row(["Logistic Regression With Class balancing","0.45","0.94","0.94","34"])
ptable.add_row(["Logistic Regression With Class balancing","0.45","0.98","0.94","35"])

print(ptable)
```

```
+------------------------------------------+-------+------+------+------------------------+
|                Model Name                | Train |  CV  | Test | % Misclassified Points |
+------------------------------------------+-------+------+------+------------------------+
| Logistic Regression With Class balancing |  0.45 | 0.94 | 0.94 |           34           |
| Logistic Regression With Class balancing |  0.45 | 0.98 | 0.94 |           35           |
+------------------------------------------+-------+------+------+------------------------+
```

with feature engineering we reduce the log loss <1

END