# personalized-medicine-redefining-cancer-treatmen

June 22, 2021

Cancer Treatment

[114]:
```
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 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
```

## 0.1 Exploring and reading the datafiles

[3]:
```
# Loading training data
data_variants = pd.read_csv('data/training_variants')
# training_text dataset uses "||" as a seperator and has the headers seperated␣
↪by "," so we skip that row and declare headers
data_text =pd.read_csv("data/
↪training_text",sep="\|\|",engine="python",names=["ID","TEXT"],skiprows=1)
```

Exploring the datasets **data_variants**

[4]:
```
data_variants.head()
```

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

Some data explanation

ID : row id 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 : class value 1-9, this genetic mutation has been classified on

This is be our result data

[5]:
```
data_variants.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3321 entries, 0 to 3320
Data columns (total 4 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   ID         3321 non-null   int64
 1   Gene       3321 non-null   object
 2   Variation  3321 non-null   object
 3   Class      3321 non-null   int64
dtypes: int64(2), object(2)
memory usage: 103.9+ KB
```

There are no null-values in the data_varuiants

[6]:
```
data_variants.describe()
```

[6]:
```
                ID        Class
count  3321.000000  3321.000000
mean   1660.000000     4.365854
std     958.834449     2.309781
min       0.000000     1.000000
25%     830.000000     2.000000
50%    1660.000000     4.000000
75%    2490.000000     7.000000
max    3320.000000     9.000000
```

Mathematical description of the dataset

[7]:
```
# Checking dimention of data
# even though we already know from count and columns exploration (data_variants.
↪head(3))
data_variants.shape
```

[7]:
```
(3321, 4)
```

[8]:
```
# Clecking column in above data set
# even though we already know from data_variants.head(3)
data_variants.columns
```

[8]:
```
Index(['ID', 'Gene', 'Variation', 'Class'], dtype='object')
```

**data_text**

[9]:
```
data_text.head(3)
```

[9]:
```
   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…
```

Even though we knew its column names which we defined ourselfs, we notice (from deeper exploration of datasets) that ID column is common and matches fro both datasets

[10]:
```
data_text.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3321 entries, 0 to 3320
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   ID      3321 non-null   int64
 1   TEXT    3316 non-null   object
```

```
dtypes: int64(1), object(1)
memory usage: 52.0+ KB
```

data_text has hull values

```
[11]: # data_text.describe() is not useful for text data values
      data_text.columns
```

```
[11]: Index(['ID', 'TEXT'], dtype='object')
```

```
[12]: # checking the dimentions (which we already know)
      data_text.shape
```

```
[12]: (3321, 2)
```

```
[13]: # Confirmation of aviable results
      data_variants.Class.unique()
```

```
[13]: array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

This is descrete data so it is ***classification*** problem and since there are multiple descrete output possible so we can call it ***Multi class*** classification problem

***Important note*** : This is medical related problem so correct results are very important. Error can be really costly here so we would like to have result for each class in terms of Probablity. We might not be much bothered about time taken by ML algorithm as far as it is reasonable.

We also want our model to be highly interpritable because a medical practitionar want to also give proper reasonining on why ML algorithm is predicting any class.

We will evaluate our model using Confution matrix and Multi class log-loss

Data preparation

```
[14]: # We would like to remove all stop words like a, is, an, the, ...
      # so we collecting all of them from nltk library
      stop_words = set(stopwords.words('english'))
```

```
[15]: # defining function to remove all stop words from data
      def data_text_preprocess(total_text, ind, col):
          # Remove int values from text data as that might not be imp
          if type(total_text) is not int:
              string = ""
              # replacing all special char with space
              total_text = re.sub('[^a-zA-Z0-9\n]', ' ', str(total_text))
              # replacing multiple spaces with single space
              total_text = re.sub('\s+',' ', str(total_text))
              # bring whole text to same lower-case scale.
              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 text
            if not word in stop_words:
                string += word + " "

        data_text[col][ind] = string
```

[16]:
```
# applying data_text_preprocess to data_text
for index, row in data_text.iterrows():
    if type(row['TEXT']) is str:
        data_text_preprocess(row['TEXT'], index, 'TEXT')
```

[17]:
```
# merging both gene_variations and text data based on ID
result = pd.merge(data_variants, data_text,on='ID', how='left')
result.head()
```

[17]:
```
   ID    Gene             Variation  Class  \
0   0  FAM58A  Truncating Mutations      1
1   1     CBL                 W802*      2
2   2     CBL                 Q249E      2
3   3     CBL                 N454D      3
4   4     CBL                 L399V      4

                                                TEXT
0  cyclin dependent kinases cdks regulate variety…
1  abstract background non small cell lung cancer…
2  abstract background non small cell lung cancer…
3  recent evidence demonstrated acquired uniparen…
4  oncogenic mutations monomeric casitas b lineag…
```

[18]:
```
# checking for missing values
# missing values may create qualitive problematics in our final analysis
result[result.isnull().any(axis=1)]
```

[18]:
```
        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
```

We can see many rows with missing data. Now the question is what to do with this missing value. One way could be that we can drop these rows having missing values or we can do some imputation in it. Let's go with imputation only. But question is what to impute here

[19]:
```
# Imputing the missing values as "Gene" + "Variation" text
# This is the example imputation used in the course (check note in the␣
↪beggining)
result.loc[result['TEXT'].isnull(),'TEXT'] = result['Gene'] +'␣
↪'+result['Variation']
```

[20]:
```
# Confirming that tere are no missing values
result[result.isnull().any(axis=1)]
```

[20]:
```
Empty DataFrame
Columns: [ID, Gene, Variation, Class, TEXT]
Index: []
```

Awesome, so all missing values are gone now.

## 0.2 Creating Training, Test and Validation data

Before we split the data into taining, test and validation data set. We want to ensure that all spaces in Gene and Variation column to be replaced by _.

[21]:
```
y_true = result['Class'].values
# replacing spaces with "_"
result.Gene      = result.Gene.str.replace('\s+', '_')
result.Variation = result.Variation.str.replace('\s+', '_')
```

[22]:
```
# Splitting the data into train and test set
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 now into train validation and cross validation
train_df, cv_df, y_train, y_cv = train_test_split(X_train, y_train,␣
↪stratify=y_train, test_size=0.2)
```

[23]:
```
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
```

**Checking on train, validation, and test datasets ditribution**

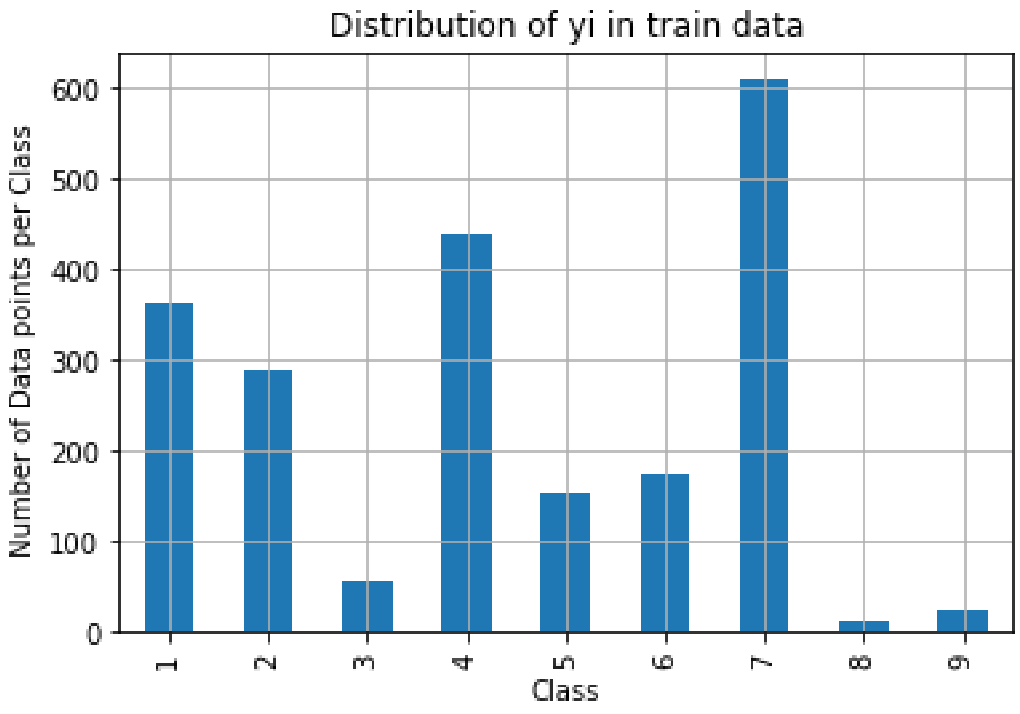
[24]:
```
# counting class values for each set and sorting for better comparison
train_class_distribution = train_df['Class'].value_counts().sort_index(axis=0,␣
↪level=None, ascending=True, inplace=False, kind='quicksort',␣
↪na_position='last', sort_remaining=True)
test_class_distribution = test_df['Class'].value_counts().sort_index(axis=0,␣
↪level=None, ascending=True, inplace=False, kind='quicksort',␣
↪na_position='last', sort_remaining=True)
```

```
cv_class_distribution = cv_df['Class'].value_counts().sort_index(axis=0,␣
↪level=None, ascending=True, inplace=False, kind='quicksort',␣
↪na_position='last', sort_remaining=True)
```

[25]:
```
train_class_distribution
```

[25]:
```
1    363
2    289
3     57
4    439
5    155
6    176
7    609
8     12
9     24
Name: Class, dtype: int64
```

[26]:
```
# Visualizing train class distrubution
my_colors = 'rgbkymc'
train_class_distribution.plot(kind='bar')
plt.xlabel('Class')
plt.ylabel(' Number of Data points per Class')
plt.title('Distribution of yi in train data')
plt.grid()
plt.show()
```

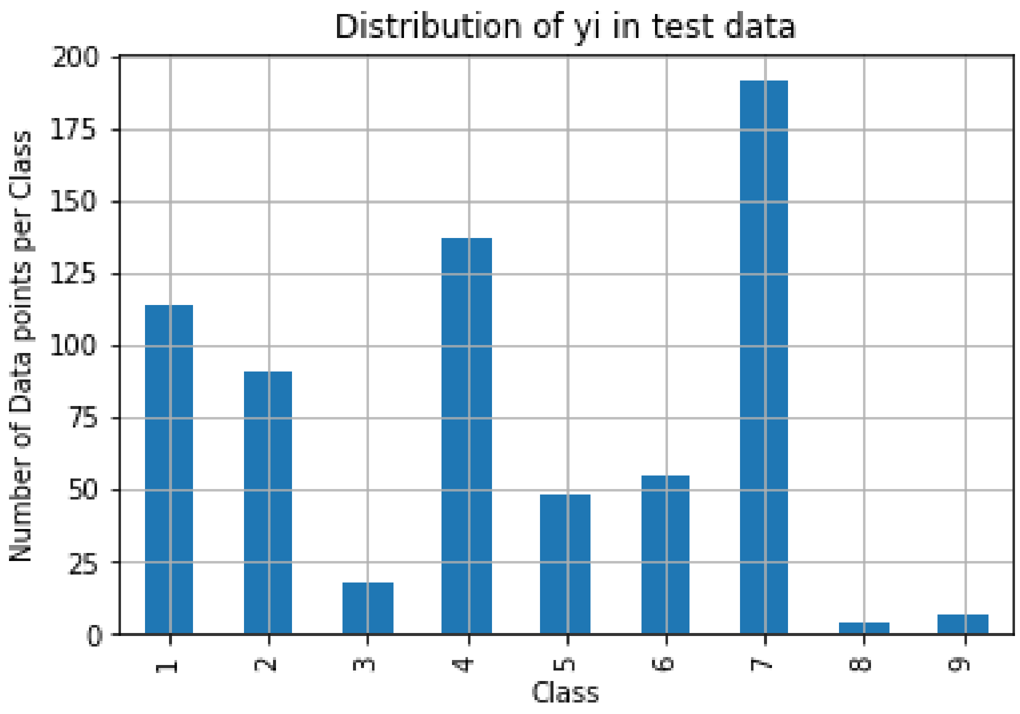


** Visualizing for train class distrubution**

[27]:
```
# Printing distribution in percentage form
sorted_yi = np.argsort(-train_class_distribution.values)
for i in sorted_yi:
    print('Number of data points in class', i+1, ':',train_class_distribution.
 ↪values[i], '(', np.round((train_class_distribution.values[i]/train_df.
 ↪shape[0]*100), 3), '%)')
```

```
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 %)
```

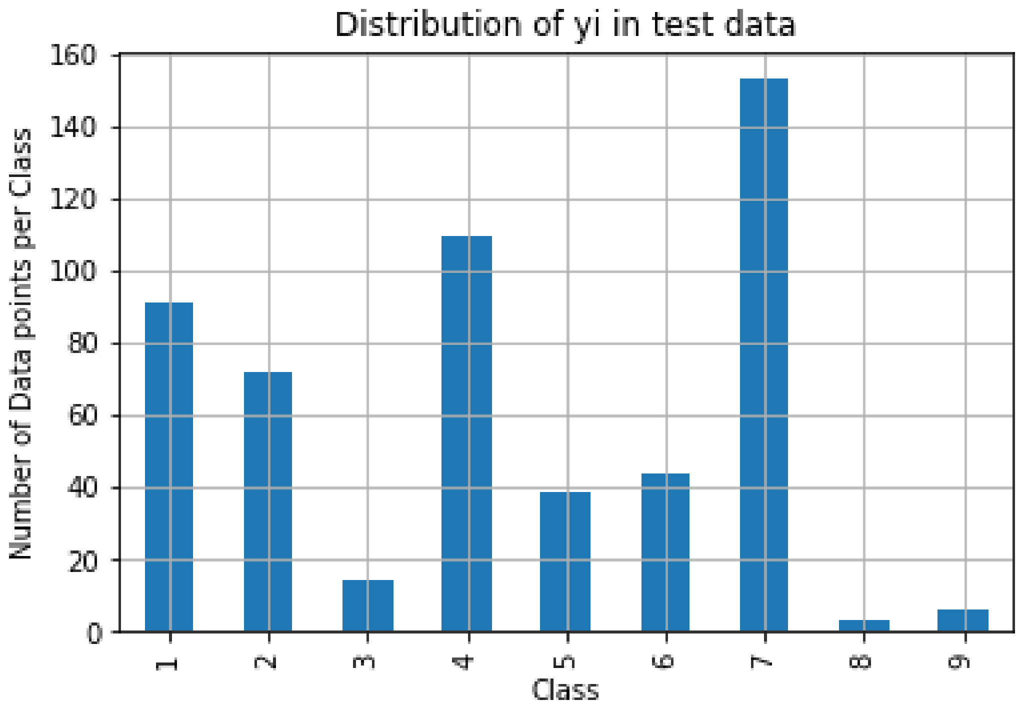
[28]:
```
# test set diribution visualization
my_colors = 'rgbkymc'
test_class_distribution.plot(kind='bar')
plt.xlabel('Class')
plt.ylabel('Number of Data points per Class')
plt.title('Distribution of yi in test data')
plt.grid()
plt.show()
```



[29]:
```
# test set distribution in percentage form
sorted_yi = np.argsort(-test_class_distribution.values)
for i in sorted_yi:
    print('Number of data points in class', i+1, ':',test_class_distribution.
 ↪values[i], '(', np.round((test_class_distribution.values[i]/test_df.
 ↪shape[0]*100), 3), '%)')
```

```
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 %)
```

[30]:
```
# cross validation set diribution visualization
my_colors = 'rgbkymc'
cv_class_distribution.plot(kind='bar')
plt.xlabel('Class')
plt.ylabel('Number of Data points per Class')
plt.title('Distribution of yi in test data')
plt.grid()
plt.show()
```



[31]:
```
# cross validation set distribution in percentage form
sorted_yi = np.argsort(-cv_class_distribution.values)
for i in sorted_yi:
    print('Number of data points in class', i+1, ':',test_class_distribution.
↪values[i], '(', np.round((test_class_distribution.values[i]/test_df.
↪shape[0]*100), 3), '%)')
```

```
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 %)
```

## 0.3 Building a Random model - WORST MODEL

[32]:
```
# getting the length of our datasets
test_data_len = test_df.shape[0]
cv_data_len = cv_df.shape[0]

# creating an 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)
    # setting random values so each row sum is equal to 1
    cv_predicted_y[i] = ((rand_probs/sum(sum(rand_probs)))[0])
# Evaluating and printing the Log Loss for worst model
# All models that will be generated should not be worse than worst model
print("Log loss on Cross Validation Data using Random␣
 ↪Model",log_loss(y_cv,cv_predicted_y, eps=1e-15))

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

```
Log loss on Cross Validation Data using Random Model 2.4983107431467615
Log loss on Test Data using Random Model 2.4450267324737074
```

[33]:
```
# Lets get the index of max probablity
predicted_y =np.argmax(test_predicted_y, axis=1)
predicted_y
```

[33]:
```
array([4, 7, 3, 5, 6, 2, 1, 1, 1, 5, 6, 8, 5, 1, 8, 0, 3, 5, 6, 6, 0, 3,
       8, 8, 6, 5, 7, 1, 5, 4, 0, 8, 7, 7, 5, 4, 7, 5, 4, 1, 1, 8, 8, 8,
       6, 8, 7, 1, 6, 7, 0, 0, 2, 5, 4, 2, 1, 0, 4, 8, 1, 2, 7, 5, 5, 1,
       2, 3, 0, 6, 7, 8, 5, 4, 8, 2, 1, 6, 6, 7, 2, 2, 8, 0, 7, 2, 7, 1,
       4, 8, 1, 2, 3, 6, 6, 6, 4, 2, 1, 7, 1, 2, 2, 2, 4, 8, 1, 5, 5, 3,
       2, 1, 0, 0, 1, 7, 4, 4, 2, 1, 0, 1, 6, 6, 7, 7, 2, 0, 1, 7, 5, 3,
       6, 0, 7, 8, 4, 5, 5, 6, 6, 2, 6, 0, 1, 5, 4, 0, 4, 6, 7, 1, 6, 6,
       8, 7, 4, 8, 6, 4, 8, 2, 7, 8, 1, 1, 6, 6, 2, 2, 8, 6, 8, 4, 2, 8,
       0, 3, 6, 8, 0, 7, 5, 0, 1, 7, 7, 7, 3, 1, 2, 1, 8, 8, 2, 0, 3, 3,
       7, 8, 3, 0, 6, 3, 6, 7, 0, 4, 3, 8, 2, 6, 8, 7, 7, 4, 8, 5, 8, 1,
       3, 6, 0, 5, 5, 7, 7, 0, 6, 3, 5, 0, 5, 6, 2, 7, 1, 5, 4, 8, 8, 5,
       1, 7, 4, 7, 6, 0, 2, 1, 5, 2, 7, 3, 6, 3, 5, 2, 2, 5, 8, 4, 4, 6,
       1, 5, 2, 2, 5, 0, 3, 4, 8, 6, 3, 2, 8, 4, 8, 0, 5, 0, 2, 5, 8, 4,
       7, 5, 8, 5, 7, 8, 5, 7, 8, 0, 5, 4, 7, 4, 0, 0, 6, 4, 0, 7, 5, 3,
       7, 1, 4, 7, 3, 4, 3, 7, 8, 3, 3, 8, 0, 3, 2, 3, 5, 1, 1, 6, 5, 7,
       7, 0, 5, 0, 8, 0, 1, 3, 8, 8, 7, 6, 6, 6, 3, 2, 4, 4, 5, 6, 4, 2,
       3, 3, 2, 0, 1, 2, 3, 3, 6, 5, 6, 1, 3, 7, 2, 4, 0, 7, 2, 7, 1, 3,
       8, 2, 2, 2, 3, 4, 1, 1, 3, 1, 2, 6, 1, 6, 0, 6, 7, 2, 4, 0, 5, 5,
       3, 4, 6, 6, 4, 4, 3, 2, 8, 5, 4, 4, 2, 0, 3, 6, 7, 1, 1, 0, 7, 0,
       8, 5, 2, 0, 5, 1, 7, 4, 8, 0, 5, 1, 5, 2, 8, 1, 0, 3, 1, 5, 4, 6,
       2, 5, 5, 4, 1, 2, 2, 2, 0, 6, 6, 2, 6, 2, 7, 6, 5, 2, 2, 2, 2, 2,
       5, 7, 0, 6, 7, 6, 6, 2, 4, 0, 8, 0, 2, 6, 7, 5, 5, 0, 5, 7, 5, 1,
       1, 8, 4, 5, 8, 6, 3, 2, 7, 8, 5, 6, 6, 3, 4, 2, 5, 5, 0, 1, 5, 3,
       6, 4, 2, 0, 6, 7, 1, 4, 6, 7, 5, 8, 4, 3, 0, 8, 0, 2, 6, 2, 2, 2,
       1, 1, 1, 0, 4, 6, 4, 4, 2, 0, 4, 8, 8, 4, 7, 5, 7, 3, 2, 3, 4, 8,
       2, 6, 6, 8, 4, 3, 8, 7, 1, 1, 3, 1, 8, 1, 2, 8, 3, 6, 8, 0, 7, 0,
       5, 7, 7, 6, 2, 7, 6, 3, 6, 5, 6, 2, 7, 0, 0, 1, 1, 5, 4, 1, 0, 3,
       0, 2, 8, 6, 3, 3, 6, 6, 6, 3, 3, 5, 6, 0, 2, 7, 3, 5, 7, 8, 7, 4,
       3, 7, 6, 7, 1, 5, 3, 4, 2, 5, 0, 2, 5, 2, 6, 2, 5, 7, 3, 8, 2, 7,
       3, 5, 1, 0, 3, 6, 0, 8, 3, 7, 4, 8, 0, 8, 3, 4, 7, 6, 6, 5, 8, 1,
       4, 6, 5, 6, 5])
```

The index value ranging from 0 to 8. Each value should be increased by 1.

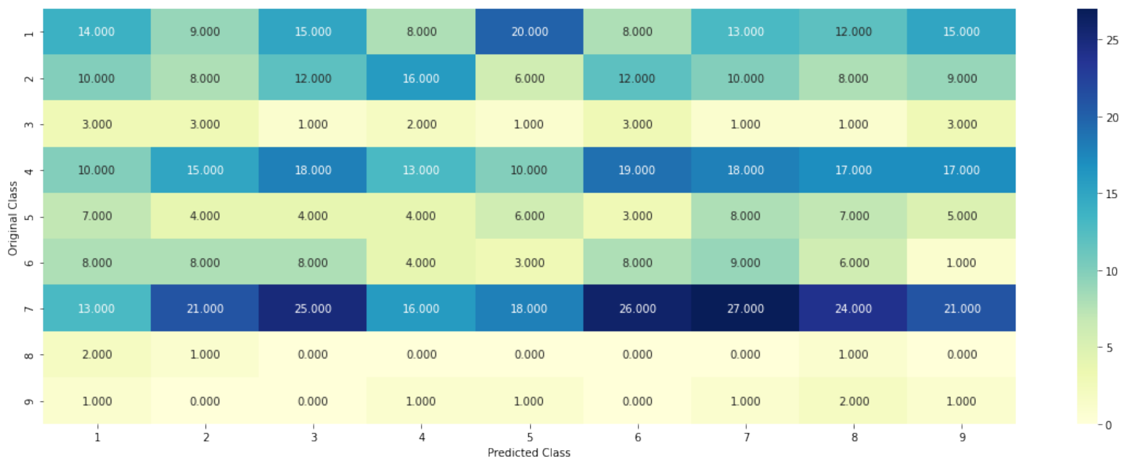
[34]:
```
predicted_y = predicted_y + 1
```

**Confusion Matrix**

[35]:
```
C = confusion_matrix(y_test, predicted_y)
C
```

[35]:
```
array([[14,  9, 15,  8, 20,  8, 13, 12, 15],
       [10,  8, 12, 16,  6, 12, 10,  8,  9],
       [ 3,  3,  1,  2,  1,  3,  1,  1,  3],
       [10, 15, 18, 13, 10, 19, 18, 17, 17],
       [ 7,  4,  4,  4,  6,  3,  8,  7,  5],
       [ 8,  8,  8,  4,  3,  8,  9,  6,  1],
       [13, 21, 25, 16, 18, 26, 27, 24, 21],
       [ 2,  1,  0,  0,  0,  0,  0,  1,  0],
       [ 1,  0,  0,  1,  1,  0,  1,  2,  1]])
```

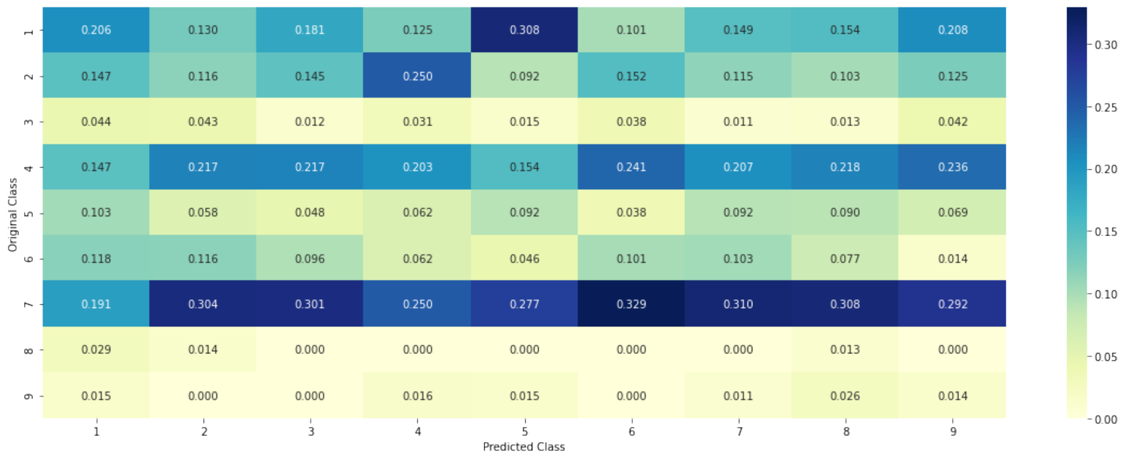
[36]:
```
# Displaying predictions
labels = [1,2,3,4,5,6,7,8,9]
plt.figure(figsize=(20,7))
sns.heatmap(C, annot=True, cmap="YlGnBu", fmt=".3f", xticklabels=labels,␣
 ↪yticklabels=labels)
plt.xlabel('Predicted Class')
plt.ylabel('Original Class')
plt.show()
```



The diagonal of confusion matrix gives the correctly predicted values (values with class 1 predicted as class 1 for the first cell of example)

**Precision matrix**

[37]:
```python
# generating precision matrix
B =(C/C.sum(axis=0))
# diplaying precision matrix
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()
```

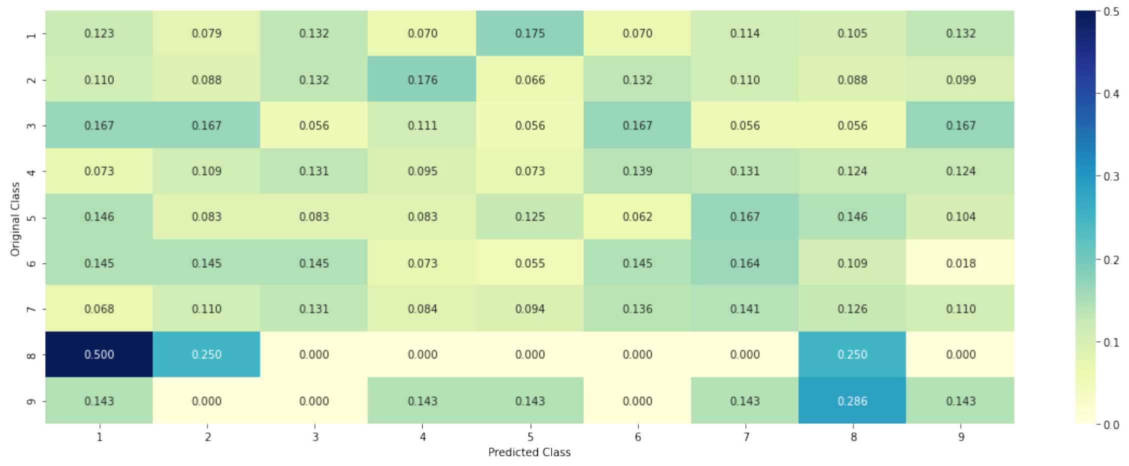


Precision matrix is same as confusion matrix put cells are expressed as percentages in the same column (values predicted as class Xi) Cell 1,1 shows how many percent of points predicted as Class

1 are actualy Class 1

**Recall matrix**

```
[38]: # generating recall matrix
A =(((C.T)/(C.sum(axis=1))).T)
# diplaying recall matrix
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()
```



recall matrix is same as confusion matrix but values are expressed as percentage in row direction meaning each cell represents how percent of class i are predicted as class j

## 0.4 Evaluating Gene Column

Now we will look at each independent column to make sure its relavent for my target variable but the question is, how? Let's understand with our first column Gene which is categorial in nature.

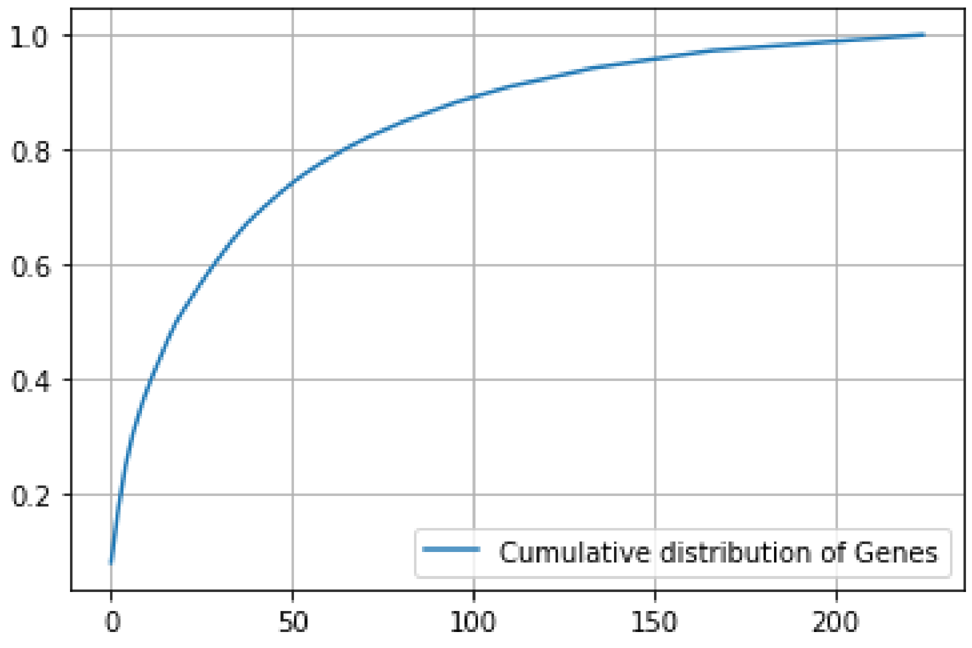
So, lets explore column **Gene** and lets look at its distribution.

```
[39]: # Creating unique genes series
unique_genes = train_df['Gene'].value_counts()
# measuring the length of unique genes matrix
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 : 225
BRCA1     171
TP53      102
```

```
EGFR        98
PTEN        81
BRCA2       78
KIT         60
BRAF        57
ERBB2       45
ALK         43
PDGFRA      41
Name: Gene, dtype: int64
```

[40]:
```python
# Cumulative distribution of unique genes
s = sum(unique_genes.values);
h = unique_genes.values/s;
c = np.cumsum(h)
plt.plot(c,label='Cumulative distribution of Genes')
plt.grid()
plt.legend()
plt.show()
```



From the graph - Top 50 Gene attributes make almost 75% of all data

We need to convert these categorical variable to appropirate format which my machine learning algorithm will be able to take as an input.

So we have 2 techniques to deal with it.

```
***One-hot encoding***
***Response Encoding*** (Mean imputation)
```

We will use both of them to see which one work the best. So lets start encoding using one hot encoder For Response encoding we will add columns representing class values filled with values of probability of that class given GENE - i

**One hot encoder of genes classification**

```
[41]: # one-hot encoding of Gene feature.
gene_vectorizer = CountVectorizer()
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'])
```

```
[42]: # checking classified train set shape
train_gene_feature_onehotCoding.shape
```

```
[42]: (2124, 225)
```

```
[43]: # plotting added column names
gene_vectorizer.get_feature_names()
```

```
[43]: ['abl1',
 'acvr1',
 'ago2',
 'akt1',
 'akt2',
 'akt3',
 'alk',
 'apc',
 'ar',
 'araf',
 'arid1a',
 'arid1b',
 'arid5b',
 'asxl2',
 'atm',
 'atrx',
 'aurka',
 'aurkb',
 'axin1',
 'b2m',
 'bap1',
 'bcl2l11',
 'bcor',
 'braf',
 'brca1',
 'brca2',
 'brd4',
 'brip1',
 'btk',
 'card11',
 'carm1',
 'casp8',
 'cbl',
 'ccnd1',
 'ccnd3',
 'ccne1',
 'cdh1',
 'cdk12',
 'cdk4',
 'cdk6',
 'cdk8',
 'cdkn1a',
 'cdkn1b',
 'cdkn2a',
 'cdkn2b',
 'chek2',
 'cic',
 'crebbp',
 'ctcf',
 'ctla4',
 'ctnnb1',
 'ddr2',
 'dicer1',
 'dnmt3a',
 'dnmt3b',
 'egfr',
 'eif1ax',
 'elf3',
 'ep300',
 'epas1',
 'epcam',
 'erbb2',
 'erbb3',
 'erbb4',
 'ercc2',
 'ercc4',
 'erg',
 'errfi1',
 'esr1',
 'etv1',
 'etv6',
 'ewsr1',
 'ezh2',
 'fanca',
 'fancc',
 'fat1',
 'fbxw7',
 'fgf19',
 'fgfr1',
 'fgfr2',
 'fgfr3',
 'fgfr4',
 'flt1',
 'flt3',
 'foxa1',
 'foxl2',
 'foxo1',
 'fubp1',
 'gata3',
 'gli1',
 'gnaq',
 'gnas',
 'h3f3a',
 'hist1h1c',
 'hla',
 'hnf1a',
 'hras',
 'idh1',
 'idh2',
 'igf1r',
 'ikzf1',
 'il7r',
 'inpp4b',
 'jak1',
 'jak2',
 'jun',
 'kdm5a',
 'kdm5c',
 'kdm6a',
 'kdr',
 'keap1',
 'kit',
 'klf4',
 'kmt2a',
 'kmt2c',
 'kmt2d',
 'knstrn',
 'kras',
 'map2k1',
 'map2k2',
 'map2k4',
 'map3k1',
 'mapk1',
 'mdm4',
 'med12',
 'mef2b',
 'men1',
 'met',
 'mga',
 'mlh1',
 'mpl',
 'msh2',
 'msh6',
 'mtor',
 'myc',
 'mycn',
 'myd88',
 'nf1',
 'nf2',
 'nfe2l2',
 'nfkbia',
 'nkx2',
 'notch1',
 'notch2',
 'npm1',
 'nras',
 'nsd1',
 'ntrk1',
 'ntrk2',
 'ntrk3',
 'nup93',
 'pak1',
 'pbrm1',
 'pdgfra',
 'pdgfrb',
 'pik3ca',
 'pik3cb',
 'pik3cd',
 'pik3r1',
 'pik3r2',
 'pim1',
 'pms1',
 'pms2',
 'pole',
 'ppp2r1a',
 'ppp6c',
 'prdm1',
 'ptch1',
 'pten',
 'ptpn11',
 'ptprd',
 'ptprt',
 'rab35',
 'rac1',
 'rad21',
 'rad50',
 'rad51b',
 'raf1',
 'rasa1',
 'rb1',
 'rbm10',
 'ret',
 'rheb',
 'rhoa',
 'rit1',
 'ros1',
 'runx1',
 'rxra',
 'rybp',
 'sdhb',
 'sdhc',
 'setd2',
 'sf3b1',
 'shoc2',
 'shq1',
 'smad2',
 'smad3',
 'smad4',
 'smarca4',
 'smarcb1',
 'smo',
 'sos1',
 'sox9',
 'spop',
 'src',
 'srsf2',
 'stag2',
 'stat3',
 'stk11',
 'tcf3',
 'tert',
 'tet1',
 'tet2',
 'tgfbr1',
 'tmprss2',
 'tp53',
 'tsc1',
 'tsc2',
 'u2af1',
 'vegfa',
 'vhl',
 'whsc1',
 'xpo1',
 'xrcc2',
 'yap1']
```

** Response Encoding of genes classification** Code is as given in the course (i do not fully understand Response Encoding yet). Course link given in description at the begining of this notebook

[44]:
```
# ----Notes----
# code for response coding with Laplace smoothing.
# alpha : used for laplace smoothing
# feature: ['gene', 'variation']
# df: ['train_df', 'test_df', 'cv_df']
# algorithm
# ----------
# Consider all unique values and the number of occurances of given feature in␣
 ↪train data dataframe
# build a vector (1*9) , the first element = (number of times it occured in␣
 ↪class1 + 10*alpha / number of time it 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
    #         ...}
    # print(train_df['Variation'].value_counts())
    # output:
    # {
    # Truncating_Mutations                     63
    # Deletion                                 43
    # Amplification                            43
    # Fusions                                  22
    # Overexpression                            3
    # E17K                                      3
    # Q61L                                      3
    # S222D                                     2
    # P130S                                     2
    # ...
    # }
    value_count = train_df[feature].value_counts()

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

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

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

            # cls_cnt.shape[0](numerator) will contain the number of time that␣
↪particular feature occured in whole 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, 0.051020408163265307, 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, 0.1393939393939394, 0.
 ↪34545454545454546, 0.060606060606060608, 0.060606060606060608, 0.
 ↪060606060606060608],
    #      'PTEN': [0.069182389937106917, 0.062893081761006289, 0.
 ↪069182389937106917, 0.46540880503144655, 0.075471698113207544, 0.
 ↪062893081761006289, 0.069182389937106917, 0.062893081761006289, 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 data
    gv_fea = []
    # for every feature values in the given data frame we will check if it is␣
 ↪there in the train data then we will add the feature to gv_fea
```

```
    # if not we will add [1/9,1/9,1/9,1/9,1/9,1/9,1/9,1/9,1/9] to gv_fea
    for index, row in df.iterrows():
        if row[feature] in dict(value_count).keys():
            gv_fea.append(gv_dict[row[feature]])
        else:
            gv_fea.append([1/9,1/9,1/9,1/9,1/9,1/9,1/9,1/9,1/9])
#               gv_fea.append([-1,-1,-1,-1,-1,-1,-1,-1,-1])
    return gv_fea
```

[45]:
```
# 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))
```

[46]:
```
# exploring training gene shape
train_gene_feature_responseCoding.shape
```

[46]:
```
(2124, 9)
```

Response encoding adds a number of classes equal to number of classes (posible results)

We will build model having only gene column with one hot encoder with simple model like Logistic regression. If log loss with only one column Gene comes out to be better than random model, than this feature is important.

[47]:
```
# We need a hyperparemeter for SGD classifier.
# giving alpha a set of ranges to compare
alpha = [10 ** x for x in range(-5, 1)]
```

[48]:
```
# We will be using SGD classifier
# http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.
 ↪SGDClassifier.html
# We will also be using Calibrated Classifier to get the result into probablity␣
 ↪format t be used for log loss
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.classes_, eps=1e-15))
```

```
For values of alpha =  1e-05 The log loss is: 1.1988978880599888
For values of alpha =  0.0001 The log loss is: 1.1907921428324062
For values of alpha =  0.001 The log loss is: 1.251415171375968
For values of alpha =  0.01 The log loss is: 1.3661590369405565
For values of alpha =  0.1 The log loss is: 1.4194498089685597
For values of alpha =  1 The log loss is: 1.448480654741542
```

[49]:
```
# Lets plot the same to check the best Alpha value
#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()


# because the graph is not clear this section is commented and written to just␣
↪give the best alpha
def print_best_alpha(alpha_arr, loss_arr):
    print("The best alpha is: " + str(alpha_arr[loss_arr.index(min(loss_arr))])␣
↪+ "\nThe best alpha index is: " + str(loss_arr.index(min(loss_arr))))
print_best_alpha(alpha, cv_log_error_array)
```

```
The best alpha is: 0.0001
The best alpha index is: 1
```

[50]:
```
# Lets use best alpha value as we can see from above graph and compute log loss
# Building a very simple model using just gene column to check error decrease␣
↪from worst model
# building a model with just one feature gives information on how much that␣
↪feature is meaningful for the final result
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_loss(y_cv, predict_y, labels=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(test_gene_feature_onehotCoding)
print('For values of best alpha = ', alpha[best_alpha], "The test log loss is:
↪",log_loss(y_test, predict_y, labels=clf.classes_, eps=1e-15))
```

```
For values of best alpha =  0.0001 The train log loss is: 0.9795845316837705
For values of best alpha =  0.0001 The cross validation log loss is:
1.1907921428324062
For values of best alpha =  0.0001 The test log loss is: 1.238282196746077
```

Now lets check how many values are overlapping between train, test or between CV and train

```
[51]: # checking for overlaping between train set and [cross_validation, test] set
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]
```

```
[52]: print('1. In test data',test_coverage, 'out of',test_df.shape[0], ":
↪",(test_coverage/test_df.shape[0])*100)
print('2. In cross validation data',cv_coverage, 'out of ',cv_df.shape[0],":"␣
↪,(cv_coverage/cv_df.shape[0])*100)
```

```
1. In test data 632 out of 665 : 95.03759398496241
2. In cross validation data 515 out of  532 : 96.80451127819549
```

OVerlaping of treining set with cross-validation and test set is very high which is very good for our models

## 0.5 Evaluating Variation column

Variation is also a categorical variable so we have to deal in same way like we have done for ***Gene*** column. We will again get the one hot encoder and response enoding variable for variation column.

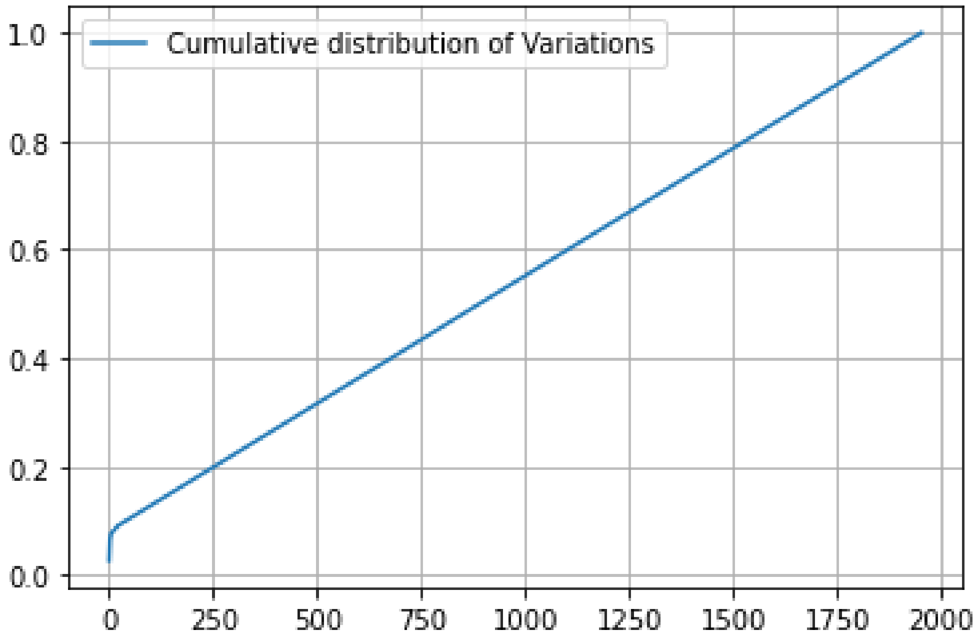
```
[53]: unique_variations = train_df['Variation'].value_counts()
print('Number of Unique Variations :', unique_variations.shape[0])
# the top 10 variations that occured most
print(unique_variations.head(10))
```

```
Number of Unique Variations : 1952
Truncating_Mutations    57
Deletion                46
Amplification           36
Fusions                 16
Overexpression           5
```

```
T58I                     3
G13C                     2
ETV6-NTRK3_Fusion        2
Q61H                     2
F28L                     2
Name: Variation, dtype: int64
```

[54]:
```
# looking at the comulative distribution of unique variation values
s = sum(unique_variations.values);
h = unique_variations.values/s;
c = np.cumsum(h)
print(c)
plt.plot(c,label='Cumulative distribution of Variations')
plt.grid()
plt.legend()
plt.show()
```

```
[0.02683616 0.04849341 0.06544256 … 0.99905838 0.99952919 1.        ]
```



## 0.6 Evaluating Variation column

Variation is also a categorical variable so we have to deal in same way like we have done for Gene column. We will again get the one hot encoder and response enoding variable for variation column.

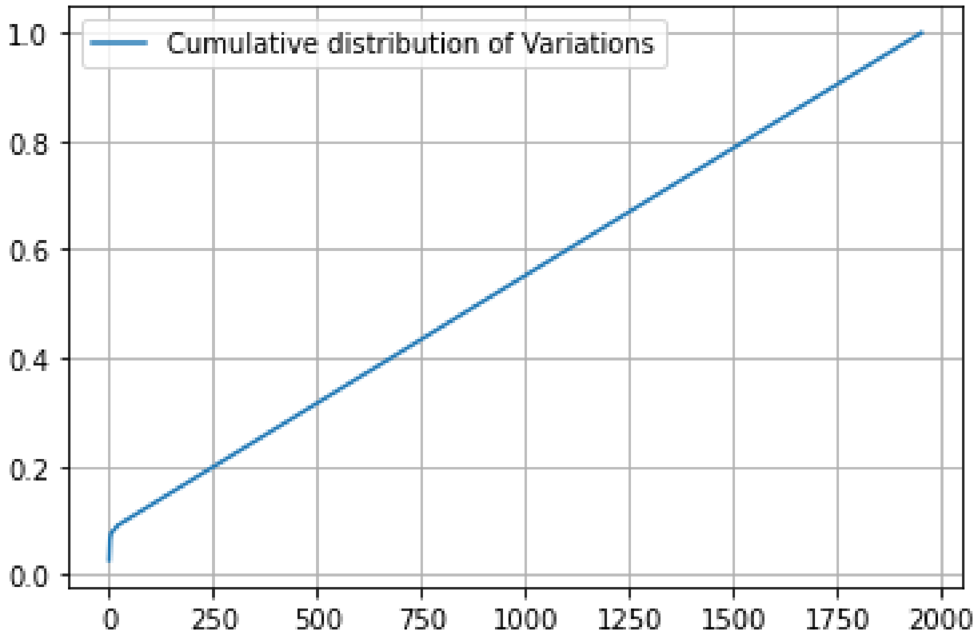
[55]:
```
unique_variations = train_df['Variation'].value_counts()
print('Number of Unique Variations :', unique_variations.shape[0])
```

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

```
Number of Unique Variations : 1952
Truncating_Mutations    57
Deletion                46
Amplification           36
Fusions                 16
Overexpression           5
T58I                     3
G13C                     2
ETV6-NTRK3_Fusion        2
Q61H                     2
F28L                     2
Name: Variation, dtype: int64
```

[56]:
```
# ploting the distribution of variation values
s = sum(unique_variations.values);
h = unique_variations.values/s;
c = np.cumsum(h)
print(c)
plt.plot(c,label='Cumulative distribution of Variations')
plt.grid()
plt.legend()
plt.show()
```

```
[0.02683616 0.04849341 0.06544256 … 0.99905838 0.99952919 1.        ]
```



[57]:
```
# one-hot encoding of variation values.
variation_vectorizer = CountVectorizer()
train_variation_feature_onehotCoding = variation_vectorizer.
↪fit_transform(train_df['Variation'])
test_variation_feature_onehotCoding = variation_vectorizer.
↪transform(test_df['Variation'])
cv_variation_feature_onehotCoding = variation_vectorizer.
↪transform(cv_df['Variation'])
```

[58]:
```
# The shape of one hot encoder column for variation
train_variation_feature_onehotCoding.shape
```

[58]:
```
(2124, 1976)
```

[59]:
```
# Response encoding of variation values.
# 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))
```

[60]:
```
# the shape of this response encoding result
train_variation_feature_responseCoding.shape
```

[60]:
```
(2124, 9)
```

[61]:
```
# Lets again build the model with only column name of variation column
# We need a hyperparemeter for SGD classifier.
alpha = [10 ** x for x in range(-5, 1)]
# We will be using SGD classifier
# http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.
↪SGDClassifier.html
# We will also be using Calibrated Classifier to get the result into probablity␣
↪format t be used for log loss
cv_log_error_array=[]
for i in alpha:
    clf = SGDClassifier(alpha=i, penalty='l2', loss='log', random_state=42)
    clf.fit(train_variation_feature_onehotCoding, y_train)
    sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
    sig_clf.fit(train_variation_feature_onehotCoding, y_train)
    predict_y = sig_clf.predict_proba(cv_variation_feature_onehotCoding)

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

```
For values of alpha =  1e-05 The log loss is: 1.7141045692201116
For values of alpha =  0.0001 The log loss is: 1.709502946278758
For values of alpha =  0.001 The log loss is: 1.7173042028158032
For values of alpha =  0.01 The log loss is: 1.7275108511038793
For values of alpha =  0.1 The log loss is: 1.7363861296986798
For values of alpha =  1 The log loss is: 1.73757998584688
```

[62]:
```
print_best_alpha(alpha, cv_log_error_array)
```

```
The best alpha is: 0.0001
The best alpha index is: 1
```

[63]:
```
# checking for error on a simple model buided using just variation column
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_loss(y_cv, predict_y, labels=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(test_variation_feature_onehotCoding)
print('For values of best alpha = ', alpha[best_alpha], "The test log loss is:
↪",log_loss(y_test, predict_y, labels=clf.classes_, eps=1e-15))
```

```
For values of best alpha =  0.0001 The train log loss is: 0.6085583338786402
For values of best alpha =  0.0001 The cross validation log loss is:
1.709502946278758
For values of best alpha =  0.0001 The test log loss is: 1.6929151540351735
```

There is very high difference on error on the training set with errors in cross-validation and test set

[64]:
```
# checking overlaping of training set with other 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('1. In test data',test_coverage, 'out of',test_df.shape[0], ":
 ↪",(test_coverage/test_df.shape[0])*100)
print('2. In cross validation data',cv_coverage, 'out of ',cv_df.shape[0],":"␣
 ↪,(cv_coverage/cv_df.shape[0])*100)
```

```
1. In test data 93 out of 665 : 13.984962406015036
2. In cross validation data 53 out of  532 : 9.962406015037594
```

There is very low overlapping. The low overlapping is the main reason for error increase from training set to test and cross-validation set This feature may be considered to be dropped

## 0.7 Evaluating text column

text column is little different from other columns because it cointains many words in a single cell we have already removed all stop words from our text data cells

[65]:
```
# 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

#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
```

[66]:
```
# building a CountVectorizer with all the words that occured minimum 3 times in␣
 ↪train data
text_vectorizer = CountVectorizer(min_df=3)
train_text_feature_onehotCoding = text_vectorizer.
 ↪fit_transform(train_df['TEXT'])
# getting all the feature names (words)
train_text_features= text_vectorizer.get_feature_names()

# train_text_feature_onehotCoding.sum(axis=0).A1 will sum every row and returns␣
 ↪(1*number of features) vector
train_text_fea_counts = train_text_feature_onehotCoding.sum(axis=0).A1

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

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

```
Total number of unique words in train data : 53256
```

[67]:
```
dict_list = []
# dict_list =[] contains 9 dictoinaries each corresponds to a class
for i in range(1,10):
    cls_text = train_df[train_df['Class']==i]
    # build a word dict based on the words in that class
    dict_list.append(extract_dictionary_paddle(cls_text))
    # append it to dict_list

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


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

[68]:
```
# response coding of text features
# text column calculations teke a long time
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)
```

[69]:
```
# 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
```

[70]:
```
# 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)
```

[71]:
```
#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()))
```

[72]:
```
# Number of words for a given frequency.
print(Counter(sorted_text_occur))
```

```
Counter({3: 5613, 4: 3647, 6: 2916, 5: 2765, 8: 2326, 7: 2053, 12: 1642, 9:
1593, 10: 1355, 11: 980, 14: 929, 13: 874, 15: 868, 16: 793, 18: 713, 24: 605,
17: 583, 20: 569, 21: 479, 19: 463, 28: 405, 27: 398, 42: 391, 30: 376, 22: 376,
25: 358, 26: 348, 23: 341, 32: 308, 36: 283, 29: 282, 53: 267, 31: 267, 35: 255,
34: 254, 33: 246, 38: 221, 40: 215, 39: 212, 48: 194, 37: 192, 44: 175, 50: 174,
43: 170, 45: 166, 54: 160, 41: 158, 51: 153, 47: 152, 46: 152, 63: 149, 49: 144,
56: 143, 60: 136, 57: 134, 58: 129, 52: 123, 76: 119, 72: 119, 55: 116, 62: 115,
64: 106, 65: 105, 61: 104, 59: 103, 68: 98, 70: 97, 66: 97, 84: 95, 80: 87, 67:
86, 74: 85, 69: 85, 90: 81, 77: 81, 79: 80, 83: 77, 73: 77, 96: 74, 78: 74, 75:
72, 82: 71, 93: 68, 71: 67, 91: 65, 86: 65, 105: 64, 99: 64, 81: 62, 106: 60,
92: 60, 85: 59, 112: 58, 95: 58, 114: 57, 94: 57, 87: 57, 100: 56, 104: 55, 101:
55, 98: 54, 88: 53, 107: 51, 115: 49, 111: 49, 97: 49, 124: 48, 120: 48, 103:
48, 134: 47, 108: 47, 128: 46, 117: 46, 144: 45, 109: 44, 102: 44, 89: 44, 113:
43, 130: 42, 110: 42, 119: 41, 126: 39, 125: 39, 143: 38, 141: 38, 122: 38, 147:
36, 139: 36, 127: 36, 140: 35, 137: 34, 123: 34, 118: 34, 185: 33, 135: 33, 133:
```

```
33, 121: 33, 116: 33, 240: 32, 159: 32, 148: 32, 142: 32, 131: 32, 150: 31, 145:
31, 136: 31, 210: 30, 160: 30, 158: 30, 152: 30, 138: 29, 173: 28, 170: 28, 168:
28, 155: 28, 153: 28, 151: 28, 146: 28, 180: 27, 156: 27, 179: 26, 166: 26, 132:
26, 212: 25, 183: 25, 175: 25, 167: 25, 154: 25, 202: 24, 191: 24, 190: 24, 162:
24, 129: 24, 244: 23, 200: 23, 192: 23, 189: 23, 182: 23, 157: 23, 208: 22, 171:
22, 272: 21, 227: 21, 222: 21, 215: 21, 204: 21, 169: 21, 161: 21, 149: 21, 260:
20, 256: 20, 216: 20, 195: 20, 186: 20, 181: 20, 178: 20, 165: 20, 163: 20, 242:
19, 233: 19, 219: 19, 206: 19, 203: 19, 193: 19, 174: 19, 172: 19, 269: 18, 265:
18, 261: 18, 232: 18, 226: 18, 211: 18, 207: 18, 310: 17, 291: 17, 290: 17, 266:
17, 245: 17, 239: 17, 229: 17, 217: 17, 214: 17, 196: 17, 187: 17, 176: 17, 284:
16, 280: 16, 273: 16, 250: 16, 237: 16, 236: 16, 225: 16, 224: 16, 221: 16, 218:
16, 205: 16, 201: 16, 198: 16, 188: 16, 299: 15, 298: 15, 289: 15, 288: 15, 274:
15, 246: 15, 241: 15, 230: 15, 228: 15, 184: 15, 164: 15, 400: 14, 397: 14, 292:
14, 264: 14, 249: 14, 231: 14, 220: 14, 213: 14, 199: 14, 177: 14, 372: 13, 354:
13, 309: 13, 308: 13, 283: 13, 282: 13, 281: 13, 263: 13, 253: 13, 252: 13, 247:
13, 243: 13, 235: 13, 384: 12, 370: 12, 340: 12, 327: 12, 319: 12, 302: 12, 276:
12, 275: 12, 270: 12, 257: 12, 255: 12, 251: 12, 248: 12, 238: 12, 234: 12, 223:
12, 197: 12, 432: 11, 410: 11, 396: 11, 376: 11, 374: 11, 356: 11, 350: 11, 345:
11, 342: 11, 338: 11, 333: 11, 324: 11, 320: 11, 317: 11, 314: 11, 301: 11, 287:
11, 277: 11, 268: 11, 254: 11, 194: 11, 407: 10, 402: 10, 393: 10, 369: 10, 351:
10, 346: 10, 336: 10, 326: 10, 318: 10, 315: 10, 313: 10, 304: 10, 303: 10, 296:
10, 293: 10, 285: 10, 279: 10, 262: 10, 259: 10, 209: 10, 756: 9, 646: 9, 557:
9, 549: 9, 488: 9, 481: 9, 439: 9, 435: 9, 431: 9, 412: 9, 380: 9, 377: 9, 373:
9, 368: 9, 362: 9, 352: 9, 337: 9, 328: 9, 322: 9, 316: 9, 312: 9, 300: 9, 278:
9, 271: 9, 648: 8, 573: 8, 541: 8, 539: 8, 535: 8, 516: 8, 494: 8, 461: 8, 453:
8, 452: 8, 442: 8, 430: 8, 429: 8, 424: 8, 421: 8, 417: 8, 405: 8, 385: 8, 375:
8, 371: 8, 364: 8, 360: 8, 357: 8, 349: 8, 347: 8, 334: 8, 325: 8, 286: 8, 267:
8, 258: 8, 754: 7, 591: 7, 567: 7, 536: 7, 508: 7, 507: 7, 484: 7, 471: 7, 467:
7, 438: 7, 418: 7, 416: 7, 408: 7, 406: 7, 395: 7, 394: 7, 392: 7, 381: 7, 363:
7, 361: 7, 355: 7, 339: 7, 329: 7, 305: 7, 295: 7, 984: 6, 884: 6, 795: 6, 787:
6, 743: 6, 722: 6, 718: 6, 681: 6, 654: 6, 652: 6, 629: 6, 623: 6, 606: 6, 588:
6, 578: 6, 561: 6, 560: 6, 546: 6, 545: 6, 543: 6, 532: 6, 531: 6, 526: 6, 523:
6, 513: 6, 512: 6, 485: 6, 483: 6, 479: 6, 470: 6, 454: 6, 449: 6, 447: 6, 443:
6, 441: 6, 437: 6, 436: 6, 434: 6, 428: 6, 427: 6, 426: 6, 409: 6, 403: 6, 398:
6, 390: 6, 387: 6, 378: 6, 367: 6, 358: 6, 353: 6, 348: 6, 332: 6, 321: 6, 306:
6, 1309: 5, 1208: 5, 1173: 5, 972: 5, 929: 5, 927: 5, 871: 5, 863: 5, 852: 5,
830: 5, 784: 5, 782: 5, 747: 5, 731: 5, 729: 5, 725: 5, 711: 5, 683: 5, 656: 5,
655: 5, 642: 5, 636: 5, 624: 5, 611: 5, 609: 5, 603: 5, 600: 5, 585: 5, 579: 5,
574: 5, 571: 5, 559: 5, 552: 5, 540: 5, 537: 5, 530: 5, 528: 5, 524: 5, 515: 5,
506: 5, 504: 5, 500: 5, 493: 5, 491: 5, 480: 5, 478: 5, 477: 5, 473: 5, 472: 5,
469: 5, 466: 5, 460: 5, 459: 5, 458: 5, 456: 5, 450: 5, 445: 5, 440: 5, 425: 5,
423: 5, 420: 5, 414: 5, 413: 5, 383: 5, 365: 5, 343: 5, 335: 5, 330: 5, 323: 5,
311: 5, 297: 5, 2069: 4, 1731: 4, 1499: 4, 1336: 4, 1146: 4, 1114: 4, 1109: 4,
1101: 4, 1057: 4, 1045: 4, 1036: 4, 1025: 4, 981: 4, 963: 4, 961: 4, 943: 4,
935: 4, 933: 4, 923: 4, 911: 4, 908: 4, 905: 4, 901: 4, 887: 4, 864: 4, 858: 4,
842: 4, 837: 4, 836: 4, 826: 4, 808: 4, 791: 4, 781: 4, 772: 4, 762: 4, 733: 4,
727: 4, 719: 4, 708: 4, 703: 4, 692: 4, 688: 4, 685: 4, 684: 4, 679: 4, 675: 4,
674: 4, 673: 4, 672: 4, 668: 4, 663: 4, 658: 4, 653: 4, 651: 4, 650: 4, 647: 4,
641: 4, 635: 4, 621: 4, 618: 4, 614: 4, 613: 4, 601: 4, 592: 4, 590: 4, 589: 4,
584: 4, 577: 4, 558: 4, 554: 4, 553: 4, 550: 4, 547: 4, 544: 4, 538: 4, 533: 4,
522: 4, 511: 4, 505: 4, 501: 4, 496: 4, 495: 4, 489: 4, 486: 4, 465: 4, 463: 4,
457: 4, 451: 4, 446: 4, 422: 4, 419: 4, 411: 4, 404: 4, 391: 4, 389: 4, 388: 4,
386: 4, 382: 4, 366: 4, 359: 4, 344: 4, 341: 4, 331: 4, 294: 4, 3468: 3, 2891:
3, 2714: 3, 2516: 3, 2286: 3, 2255: 3, 2158: 3, 2131: 3, 2087: 3, 1892: 3, 1883:
3, 1878: 3, 1810: 3, 1779: 3, 1745: 3, 1708: 3, 1666: 3, 1637: 3, 1636: 3, 1635:
3, 1629: 3, 1598: 3, 1589: 3, 1573: 3, 1526: 3, 1517: 3, 1469: 3, 1467: 3, 1435:
3, 1399: 3, 1389: 3, 1367: 3, 1350: 3, 1325: 3, 1321: 3, 1314: 3, 1301: 3, 1299:
3, 1291: 3, 1287: 3, 1285: 3, 1272: 3, 1256: 3, 1252: 3, 1223: 3, 1221: 3, 1217:
3, 1207: 3, 1201: 3, 1196: 3, 1194: 3, 1186: 3, 1167: 3, 1158: 3, 1156: 3, 1155:
3, 1143: 3, 1141: 3, 1135: 3, 1134: 3, 1111: 3, 1096: 3, 1083: 3, 1072: 3, 1068:
3, 1049: 3, 1028: 3, 1009: 3, 992: 3, 990: 3, 986: 3, 983: 3, 975: 3, 974: 3,
971: 3, 970: 3, 969: 3, 966: 3, 962: 3, 959: 3, 957: 3, 949: 3, 930: 3, 920: 3,
912: 3, 909: 3, 906: 3, 902: 3, 894: 3, 892: 3, 883: 3, 882: 3, 879: 3, 877: 3,
874: 3, 870: 3, 869: 3, 867: 3, 848: 3, 843: 3, 840: 3, 835: 3, 831: 3, 825: 3,
821: 3, 818: 3, 812: 3, 805: 3, 804: 3, 800: 3, 799: 3, 797: 3, 789: 3, 783: 3,
779: 3, 778: 3, 776: 3, 775: 3, 769: 3, 766: 3, 761: 3, 753: 3, 752: 3, 750: 3,
748: 3, 744: 3, 737: 3, 735: 3, 723: 3, 721: 3, 720: 3, 717: 3, 714: 3, 707: 3,
704: 3, 699: 3, 698: 3, 695: 3, 693: 3, 691: 3, 689: 3, 680: 3, 676: 3, 667: 3,
666: 3, 662: 3, 661: 3, 660: 3, 657: 3, 645: 3, 639: 3, 638: 3, 637: 3, 632: 3,
631: 3, 625: 3, 612: 3, 610: 3, 605: 3, 604: 3, 602: 3, 595: 3, 587: 3, 581: 3,
576: 3, 575: 3, 570: 3, 569: 3, 564: 3, 562: 3, 556: 3, 542: 3, 520: 3, 519: 3,
518: 3, 517: 3, 509: 3, 502: 3, 497: 3, 476: 3, 474: 3, 468: 3, 464: 3, 462: 3,
455: 3, 448: 3, 444: 3, 415: 3, 401: 3, 399: 3, 379: 3, 307: 3, 12318: 2, 11221:
2, 9750: 2, 7725: 2, 7645: 2, 6516: 2, 6421: 2, 6414: 2, 6224: 2, 5836: 2, 5767:
2, 5670: 2, 5652: 2, 5445: 2, 4889: 2, 4835: 2, 4669: 2, 4217: 2, 4182: 2, 3988:
2, 3774: 2, 3687: 2, 3651: 2, 3599: 2, 3457: 2, 3427: 2, 3411: 2, 3394: 2, 3388:
2, 3383: 2, 3354: 2, 3349: 2, 3316: 2, 3298: 2, 3279: 2, 3270: 2, 3233: 2, 3179:
2, 3177: 2, 3169: 2, 3089: 2, 3043: 2, 2985: 2, 2975: 2, 2952: 2, 2927: 2, 2884:
2, 2879: 2, 2862: 2, 2843: 2, 2819: 2, 2787: 2, 2776: 2, 2756: 2, 2748: 2, 2708:
2, 2685: 2, 2680: 2, 2639: 2, 2594: 2, 2590: 2, 2573: 2, 2568: 2, 2550: 2, 2489:
2, 2450: 2, 2433: 2, 2414: 2, 2409: 2, 2384: 2, 2354: 2, 2338: 2, 2327: 2, 2311:
2, 2300: 2, 2279: 2, 2260: 2, 2233: 2, 2228: 2, 2219: 2, 2214: 2, 2174: 2, 2166:
2, 2161: 2, 2146: 2, 2136: 2, 2121: 2, 2118: 2, 2116: 2, 2073: 2, 2068: 2, 2064:
2, 2054: 2, 2042: 2, 2039: 2, 2024: 2, 2003: 2, 1999: 2, 1997: 2, 1992: 2, 1981:
2, 1976: 2, 1975: 2, 1969: 2, 1953: 2, 1947: 2, 1944: 2, 1935: 2, 1932: 2, 1928:
2, 1923: 2, 1909: 2, 1897: 2, 1890: 2, 1880: 2, 1876: 2, 1874: 2, 1868: 2, 1859:
2, 1845: 2, 1834: 2, 1832: 2, 1826: 2, 1813: 2, 1807: 2, 1798: 2, 1784: 2, 1777:
2, 1776: 2, 1770: 2, 1766: 2, 1756: 2, 1755: 2, 1734: 2, 1728: 2, 1704: 2, 1682:
2, 1674: 2, 1661: 2, 1654: 2, 1647: 2, 1645: 2, 1631: 2, 1630: 2, 1615: 2, 1606:
2, 1594: 2, 1591: 2, 1584: 2, 1583: 2, 1580: 2, 1578: 2, 1569: 2, 1563: 2, 1556:
2, 1555: 2, 1542: 2, 1530: 2, 1529: 2, 1525: 2, 1523: 2, 1520: 2, 1519: 2, 1512:
2, 1508: 2, 1502: 2, 1501: 2, 1498: 2, 1497: 2, 1490: 2, 1489: 2, 1483: 2, 1481:
2, 1480: 2, 1474: 2, 1472: 2, 1456: 2, 1449: 2, 1437: 2, 1429: 2, 1428: 2, 1423:
2, 1417: 2, 1414: 2, 1413: 2, 1409: 2, 1408: 2, 1405: 2, 1400: 2, 1397: 2, 1396:
2, 1395: 2, 1388: 2, 1387: 2, 1379: 2, 1370: 2, 1369: 2, 1355: 2, 1352: 2, 1346:
2, 1342: 2, 1341: 2, 1340: 2, 1339: 2, 1335: 2, 1332: 2, 1329: 2, 1327: 2, 1322:
```

```
2, 1319: 2, 1312: 2, 1307: 2, 1306: 2, 1304: 2, 1295: 2, 1290: 2, 1289: 2, 1284:
2, 1274: 2, 1268: 2, 1259: 2, 1258: 2, 1253: 2, 1250: 2, 1248: 2, 1247: 2, 1240:
2, 1230: 2, 1229: 2, 1228: 2, 1227: 2, 1225: 2, 1219: 2, 1218: 2, 1213: 2, 1209:
2, 1200: 2, 1199: 2, 1198: 2, 1197: 2, 1195: 2, 1193: 2, 1191: 2, 1188: 2, 1184:
2, 1180: 2, 1172: 2, 1170: 2, 1169: 2, 1163: 2, 1162: 2, 1159: 2, 1154: 2, 1140:
2, 1139: 2, 1136: 2, 1133: 2, 1132: 2, 1130: 2, 1128: 2, 1127: 2, 1126: 2, 1122:
2, 1120: 2, 1118: 2, 1117: 2, 1116: 2, 1107: 2, 1106: 2, 1105: 2, 1097: 2, 1095:
2, 1093: 2, 1091: 2, 1090: 2, 1086: 2, 1081: 2, 1080: 2, 1077: 2, 1074: 2, 1071:
2, 1070: 2, 1065: 2, 1064: 2, 1063: 2, 1050: 2, 1048: 2, 1047: 2, 1044: 2, 1042:
2, 1039: 2, 1037: 2, 1035: 2, 1033: 2, 1030: 2, 1014: 2, 1004: 2, 1003: 2, 1000:
2, 999: 2, 965: 2, 956: 2, 951: 2, 950: 2, 944: 2, 942: 2, 940: 2, 936: 2, 932:
2, 916: 2, 904: 2, 895: 2, 890: 2, 886: 2, 885: 2, 881: 2, 880: 2, 878: 2, 876:
2, 866: 2, 862: 2, 861: 2, 854: 2, 847: 2, 845: 2, 844: 2, 839: 2, 834: 2, 829:
2, 827: 2, 824: 2, 823: 2, 819: 2, 817: 2, 816: 2, 814: 2, 809: 2, 807: 2, 803:
2, 798: 2, 793: 2, 788: 2, 785: 2, 774: 2, 773: 2, 760: 2, 759: 2, 758: 2, 757:
2, 751: 2, 745: 2, 742: 2, 741: 2, 740: 2, 739: 2, 738: 2, 736: 2, 734: 2, 732:
2, 730: 2, 726: 2, 724: 2, 715: 2, 713: 2, 709: 2, 705: 2, 701: 2, 697: 2, 694:
2, 690: 2, 686: 2, 678: 2, 677: 2, 670: 2, 665: 2, 659: 2, 644: 2, 643: 2, 640:
2, 630: 2, 628: 2, 627: 2, 620: 2, 619: 2, 617: 2, 615: 2, 598: 2, 596: 2, 594:
2, 586: 2, 583: 2, 580: 2, 568: 2, 566: 2, 563: 2, 551: 2, 548: 2, 529: 2, 525:
2, 521: 2, 514: 2, 510: 2, 503: 2, 499: 2, 498: 2, 492: 2, 475: 2, 150030: 1,
119076: 1, 79358: 1, 66282: 1, 65306: 1, 65300: 1, 64997: 1, 62659: 1, 60999: 1,
54288: 1, 53776: 1, 49333: 1, 48575: 1, 47310: 1, 45736: 1, 42919: 1, 42243: 1,
42180: 1, 41961: 1, 41807: 1, 39682: 1, 38837: 1, 38520: 1, 38269: 1, 37755: 1,
36906: 1, 36020: 1, 35706: 1, 35322: 1, 34886: 1, 34352: 1, 32952: 1, 32623: 1,
32585: 1, 31576: 1, 31224: 1, 28994: 1, 27824: 1, 26398: 1, 25529: 1, 25393: 1,
25360: 1, 25351: 1, 25333: 1, 24266: 1, 24045: 1, 23885: 1, 23651: 1, 23647: 1,
23549: 1, 23266: 1, 23112: 1, 22064: 1, 21834: 1, 21600: 1, 21596: 1, 21591: 1,
21162: 1, 21082: 1, 20475: 1, 20208: 1, 20042: 1, 20015: 1, 19912: 1, 19794: 1,
19774: 1, 19135: 1, 19125: 1, 19064: 1, 18863: 1, 18628: 1, 18603: 1, 18480: 1,
18443: 1, 18282: 1, 18222: 1, 18213: 1, 18077: 1, 18044: 1, 17953: 1, 17767: 1,
17635: 1, 17499: 1, 17349: 1, 17290: 1, 17267: 1, 17084: 1, 17048: 1, 16961: 1,
16901: 1, 16822: 1, 16815: 1, 16490: 1, 16482: 1, 16239: 1, 16220: 1, 15973: 1,
15831: 1, 15710: 1, 15600: 1, 15507: 1, 15481: 1, 15405: 1, 15339: 1, 15298: 1,
15271: 1, 15191: 1, 15141: 1, 15020: 1, 14845: 1, 14784: 1, 14761: 1, 14423: 1,
14380: 1, 14313: 1, 14222: 1, 14192: 1, 14004: 1, 13964: 1, 13827: 1, 13825: 1,
13745: 1, 13576: 1, 13493: 1, 13400: 1, 13232: 1, 13198: 1, 13116: 1, 13085: 1,
13026: 1, 12860: 1, 12836: 1, 12819: 1, 12761: 1, 12746: 1, 12742: 1, 12660: 1,
12650: 1, 12587: 1, 12530: 1, 12402: 1, 12371: 1, 12341: 1, 12242: 1, 12224: 1,
12158: 1, 12107: 1, 12098: 1, 12095: 1, 12076: 1, 12073: 1, 12022: 1, 11936: 1,
11924: 1, 11923: 1, 11910: 1, 11879: 1, 11793: 1, 11786: 1, 11763: 1, 11733: 1,
11731: 1, 11725: 1, 11704: 1, 11697: 1, 11682: 1, 11569: 1, 11546: 1, 11533: 1,
11358: 1, 11344: 1, 11294: 1, 11252: 1, 11162: 1, 11156: 1, 11106: 1, 11044: 1,
10823: 1, 10816: 1, 10770: 1, 10738: 1, 10531: 1, 10523: 1, 10521: 1, 10506: 1,
10298: 1, 10254: 1, 10116: 1, 10115: 1, 10106: 1, 10098: 1, 10092: 1, 9992: 1,
9937: 1, 9919: 1, 9891: 1, 9880: 1, 9837: 1, 9816: 1, 9755: 1, 9672: 1, 9633: 1,
9621: 1, 9568: 1, 9566: 1, 9542: 1, 9491: 1, 9431: 1, 9412: 1, 9363: 1, 9338: 1,
9324: 1, 9272: 1, 9219: 1, 9179: 1, 9169: 1, 9127: 1, 9109: 1, 9067: 1, 9001: 1,
8981: 1, 8980: 1, 8962: 1, 8943: 1, 8915: 1, 8910: 1, 8884: 1, 8856: 1, 8847: 1,
8803: 1, 8707: 1, 8706: 1, 8694: 1, 8692: 1, 8664: 1, 8660: 1, 8637: 1, 8586: 1,
8548: 1, 8526: 1, 8514: 1, 8503: 1, 8493: 1, 8472: 1, 8462: 1, 8301: 1, 8261: 1,
8248: 1, 8150: 1, 8078: 1, 8074: 1, 8064: 1, 8053: 1, 8039: 1, 8031: 1, 8010: 1,
7998: 1, 7969: 1, 7961: 1, 7952: 1, 7926: 1, 7914: 1, 7891: 1, 7889: 1, 7879: 1,
7871: 1, 7868: 1, 7846: 1, 7838: 1, 7809: 1, 7804: 1, 7793: 1, 7756: 1, 7743: 1,
7731: 1, 7724: 1, 7662: 1, 7639: 1, 7607: 1, 7585: 1, 7562: 1, 7504: 1, 7480: 1,
7448: 1, 7411: 1, 7407: 1, 7404: 1, 7369: 1, 7344: 1, 7331: 1, 7306: 1, 7272: 1,
7224: 1, 7208: 1, 7206: 1, 7168: 1, 7128: 1, 7112: 1, 7109: 1, 7098: 1, 7076: 1,
7072: 1, 7071: 1, 7061: 1, 7052: 1, 7040: 1, 7034: 1, 6996: 1, 6985: 1, 6969: 1,
6954: 1, 6953: 1, 6949: 1, 6941: 1, 6934: 1, 6931: 1, 6918: 1, 6911: 1, 6857: 1,
6840: 1, 6830: 1, 6829: 1, 6815: 1, 6797: 1, 6792: 1, 6772: 1, 6768: 1, 6762: 1,
6748: 1, 6715: 1, 6707: 1, 6701: 1, 6694: 1, 6645: 1, 6643: 1, 6638: 1, 6634: 1,
6624: 1, 6566: 1, 6558: 1, 6489: 1, 6487: 1, 6483: 1, 6441: 1, 6376: 1, 6370: 1,
6316: 1, 6304: 1, 6292: 1, 6271: 1, 6270: 1, 6264: 1, 6237: 1, 6219: 1, 6201: 1,
6178: 1, 6149: 1, 6131: 1, 6123: 1, 6111: 1, 6103: 1, 6078: 1, 6057: 1, 6048: 1,
6044: 1, 6026: 1, 6021: 1, 6020: 1, 6012: 1, 5988: 1, 5975: 1, 5958: 1, 5931: 1,
5926: 1, 5892: 1, 5889: 1, 5838: 1, 5834: 1, 5833: 1, 5828: 1, 5795: 1, 5789: 1,
5782: 1, 5776: 1, 5771: 1, 5738: 1, 5733: 1, 5703: 1, 5700: 1, 5696: 1, 5689: 1,
5651: 1, 5639: 1, 5630: 1, 5623: 1, 5605: 1, 5592: 1, 5577: 1, 5576: 1, 5536: 1,
5527: 1, 5512: 1, 5505: 1, 5502: 1, 5501: 1, 5495: 1, 5480: 1, 5469: 1, 5461: 1,
5457: 1, 5454: 1, 5428: 1, 5425: 1, 5421: 1, 5419: 1, 5392: 1, 5375: 1, 5350: 1,
5348: 1, 5340: 1, 5339: 1, 5338: 1, 5333: 1, 5325: 1, 5319: 1, 5315: 1, 5308: 1,
5274: 1, 5273: 1, 5269: 1, 5251: 1, 5228: 1, 5208: 1, 5187: 1, 5162: 1, 5154: 1,
5127: 1, 5109: 1, 5093: 1, 5075: 1, 5071: 1, 5062: 1, 5051: 1, 5048: 1, 5037: 1,
5030: 1, 5027: 1, 5017: 1, 5008: 1, 5001: 1, 4996: 1, 4991: 1, 4990: 1, 4974: 1,
4947: 1, 4939: 1, 4923: 1, 4922: 1, 4909: 1, 4906: 1, 4898: 1, 4893: 1, 4876: 1,
4875: 1, 4871: 1, 4842: 1, 4838: 1, 4837: 1, 4833: 1, 4831: 1, 4822: 1, 4804: 1,
4790: 1, 4775: 1, 4773: 1, 4768: 1, 4756: 1, 4754: 1, 4749: 1, 4748: 1, 4743: 1,
4740: 1, 4731: 1, 4727: 1, 4701: 1, 4692: 1, 4685: 1, 4680: 1, 4679: 1, 4648: 1,
4645: 1, 4640: 1, 4618: 1, 4611: 1, 4598: 1, 4589: 1, 4583: 1, 4578: 1, 4563: 1,
4550: 1, 4538: 1, 4530: 1, 4527: 1, 4526: 1, 4509: 1, 4494: 1, 4491: 1, 4484: 1,
4482: 1, 4457: 1, 4447: 1, 4437: 1, 4432: 1, 4429: 1, 4427: 1, 4418: 1, 4416: 1,
4404: 1, 4395: 1, 4390: 1, 4388: 1, 4384: 1, 4380: 1, 4375: 1, 4372: 1, 4369: 1,
4368: 1, 4366: 1, 4361: 1, 4344: 1, 4342: 1, 4341: 1, 4340: 1, 4337: 1, 4336: 1,
4331: 1, 4330: 1, 4321: 1, 4317: 1, 4313: 1, 4311: 1, 4307: 1, 4301: 1, 4282: 1,
4279: 1, 4269: 1, 4263: 1, 4256: 1, 4253: 1, 4238: 1, 4222: 1, 4208: 1, 4192: 1,
4176: 1, 4175: 1, 4173: 1, 4148: 1, 4144: 1, 4141: 1, 4140: 1, 4139: 1, 4137: 1,
4134: 1, 4123: 1, 4110: 1, 4107: 1, 4094: 1, 4089: 1, 4087: 1, 4086: 1, 4081: 1,
4078: 1, 4074: 1, 4073: 1, 4059: 1, 4057: 1, 4053: 1, 4046: 1, 4037: 1, 4016: 1,
4012: 1, 3997: 1, 3991: 1, 3984: 1, 3982: 1, 3976: 1, 3975: 1, 3968: 1, 3961: 1,
3955: 1, 3948: 1, 3943: 1, 3938: 1, 3937: 1, 3935: 1, 3926: 1, 3923: 1, 3920: 1,
3915: 1, 3898: 1, 3897: 1, 3891: 1, 3884: 1, 3866: 1, 3851: 1, 3835: 1, 3834: 1,
3827: 1, 3816: 1, 3813: 1, 3812: 1, 3806: 1, 3802: 1, 3801: 1, 3796: 1, 3794: 1,
3788: 1, 3784: 1, 3782: 1, 3779: 1, 3766: 1, 3765: 1, 3764: 1, 3741: 1, 3739: 1,
3733: 1, 3727: 1, 3719: 1, 3718: 1, 3715: 1, 3707: 1, 3702: 1, 3692: 1, 3684: 1,
3682: 1, 3666: 1, 3659: 1, 3657: 1, 3644: 1, 3641: 1, 3640: 1, 3639: 1, 3638: 1,
3634: 1, 3633: 1, 3627: 1, 3621: 1, 3612: 1, 3610: 1, 3603: 1, 3602: 1, 3598: 1,
3591: 1, 3587: 1, 3586: 1, 3578: 1, 3573: 1, 3568: 1, 3563: 1, 3545: 1, 3541: 1,
3539: 1, 3536: 1, 3533: 1, 3532: 1, 3508: 1, 3506: 1, 3495: 1, 3489: 1, 3480: 1,
3474: 1, 3471: 1, 3465: 1, 3454: 1, 3452: 1, 3451: 1, 3447: 1, 3443: 1, 3438: 1,
3436: 1, 3434: 1, 3431: 1, 3426: 1, 3421: 1, 3413: 1, 3401: 1, 3396: 1, 3385: 1,
3374: 1, 3366: 1, 3365: 1, 3362: 1, 3361: 1, 3356: 1, 3355: 1, 3353: 1, 3351: 1,
3343: 1, 3327: 1, 3325: 1, 3319: 1, 3318: 1, 3314: 1, 3311: 1, 3307: 1, 3306: 1,
3302: 1, 3297: 1, 3292: 1, 3289: 1, 3288: 1, 3286: 1, 3285: 1, 3281: 1, 3271: 1,
3263: 1, 3262: 1, 3261: 1, 3260: 1, 3255: 1, 3254: 1, 3242: 1, 3238: 1, 3234: 1,
3228: 1, 3226: 1, 3224: 1, 3220: 1, 3219: 1, 3216: 1, 3211: 1, 3200: 1, 3199: 1,
3198: 1, 3197: 1, 3189: 1, 3167: 1, 3165: 1, 3161: 1, 3156: 1, 3153: 1, 3150: 1,
3143: 1, 3139: 1, 3137: 1, 3133: 1, 3132: 1, 3131: 1, 3128: 1, 3120: 1, 3107: 1,
3106: 1, 3104: 1, 3103: 1, 3100: 1, 3097: 1, 3094: 1, 3087: 1, 3075: 1, 3073: 1,
3072: 1, 3070: 1, 3062: 1, 3059: 1, 3057: 1, 3055: 1, 3050: 1, 3048: 1, 3041: 1,
3030: 1, 3029: 1, 3020: 1, 3013: 1, 3002: 1, 3001: 1, 2998: 1, 2997: 1, 2991: 1,
2986: 1, 2984: 1, 2982: 1, 2978: 1, 2968: 1, 2963: 1, 2960: 1, 2959: 1, 2955: 1,
2945: 1, 2938: 1, 2936: 1, 2933: 1, 2928: 1, 2926: 1, 2920: 1, 2914: 1, 2907: 1,
2905: 1, 2886: 1, 2885: 1, 2875: 1, 2872: 1, 2871: 1, 2870: 1, 2861: 1, 2856: 1,
2855: 1, 2848: 1, 2838: 1, 2831: 1, 2828: 1, 2827: 1, 2806: 1, 2804: 1, 2802: 1,
2791: 1, 2789: 1, 2783: 1, 2782: 1, 2768: 1, 2757: 1, 2750: 1, 2742: 1, 2741: 1,
2737: 1, 2733: 1, 2730: 1, 2725: 1, 2722: 1, 2715: 1, 2709: 1, 2700: 1, 2696: 1,
2694: 1, 2683: 1, 2672: 1, 2671: 1, 2667: 1, 2664: 1, 2654: 1, 2645: 1, 2641: 1,
2640: 1, 2638: 1, 2632: 1, 2631: 1, 2628: 1, 2627: 1, 2617: 1, 2616: 1, 2615: 1,
2614: 1, 2611: 1, 2610: 1, 2601: 1, 2600: 1, 2597: 1, 2595: 1, 2589: 1, 2586: 1,
2585: 1, 2584: 1, 2578: 1, 2576: 1, 2572: 1, 2569: 1, 2564: 1, 2563: 1, 2562: 1,
2556: 1, 2546: 1, 2539: 1, 2534: 1, 2532: 1, 2520: 1, 2519: 1, 2518: 1, 2517: 1,
2515: 1, 2512: 1, 2510: 1, 2509: 1, 2507: 1, 2506: 1, 2505: 1, 2504: 1, 2499: 1,
2494: 1, 2493: 1, 2487: 1, 2486: 1, 2485: 1, 2478: 1, 2477: 1, 2476: 1, 2473: 1,
2472: 1, 2466: 1, 2462: 1, 2461: 1, 2460: 1, 2455: 1, 2452: 1, 2437: 1, 2434: 1,
2429: 1, 2428: 1, 2427: 1, 2423: 1, 2420: 1, 2417: 1, 2416: 1, 2415: 1, 2413: 1,
2406: 1, 2401: 1, 2399: 1, 2397: 1, 2395: 1, 2392: 1, 2391: 1, 2390: 1, 2386: 1,
2380: 1, 2378: 1, 2368: 1, 2363: 1, 2362: 1, 2355: 1, 2353: 1, 2350: 1, 2346: 1,
2345: 1, 2344: 1, 2343: 1, 2340: 1, 2337: 1, 2336: 1, 2331: 1, 2325: 1, 2324: 1,
2322: 1, 2319: 1, 2317: 1, 2310: 1, 2304: 1, 2303: 1, 2301: 1, 2298: 1, 2292: 1,
2291: 1, 2284: 1, 2282: 1, 2280: 1, 2277: 1, 2274: 1, 2264: 1, 2261: 1, 2257: 1,
2253: 1, 2252: 1, 2250: 1, 2248: 1, 2247: 1, 2245: 1, 2241: 1, 2239: 1, 2238: 1,
2236: 1, 2230: 1, 2224: 1, 2217: 1, 2212: 1, 2205: 1, 2193: 1, 2192: 1, 2190: 1,
2189: 1, 2187: 1, 2186: 1, 2185: 1, 2184: 1, 2172: 1, 2165: 1, 2164: 1, 2162: 1,
2159: 1, 2157: 1, 2153: 1, 2151: 1, 2149: 1, 2144: 1, 2141: 1, 2138: 1, 2130: 1,
2125: 1, 2114: 1, 2112: 1, 2110: 1, 2109: 1, 2106: 1, 2105: 1, 2104: 1, 2101: 1,
2097: 1, 2096: 1, 2093: 1, 2090: 1, 2083: 1, 2080: 1, 2074: 1, 2060: 1, 2052: 1,
2050: 1, 2047: 1, 2036: 1, 2034: 1, 2033: 1, 2032: 1, 2027: 1, 2025: 1, 2017: 1,
2015: 1, 2014: 1, 2013: 1, 2012: 1, 2009: 1, 2008: 1, 2006: 1, 2004: 1, 2002: 1,
1998: 1, 1996: 1, 1993: 1, 1980: 1, 1978: 1, 1974: 1, 1972: 1, 1970: 1, 1967: 1,
1966: 1, 1960: 1, 1958: 1, 1957: 1, 1956: 1, 1954: 1, 1949: 1, 1942: 1, 1940: 1,
1937: 1, 1931: 1, 1930: 1, 1926: 1, 1925: 1, 1922: 1, 1921: 1, 1918: 1, 1913: 1,
1910: 1, 1904: 1, 1902: 1, 1898: 1, 1893: 1, 1889: 1, 1881: 1, 1879: 1, 1872: 1,
1870: 1, 1861: 1, 1855: 1, 1853: 1, 1852: 1, 1851: 1, 1847: 1, 1840: 1, 1837: 1,
1835: 1, 1830: 1, 1829: 1, 1828: 1, 1827: 1, 1825: 1, 1823: 1, 1822: 1, 1821: 1,
1820: 1, 1819: 1, 1818: 1, 1817: 1, 1815: 1, 1808: 1, 1805: 1, 1803: 1, 1802: 1,
1801: 1, 1795: 1, 1794: 1, 1793: 1, 1790: 1, 1788: 1, 1787: 1, 1781: 1, 1780: 1,
1778: 1, 1775: 1, 1774: 1, 1772: 1, 1771: 1, 1768: 1, 1765: 1, 1764: 1, 1762: 1,
1760: 1, 1758: 1, 1754: 1, 1753: 1, 1751: 1, 1748: 1, 1747: 1, 1741: 1, 1738: 1,
1737: 1, 1735: 1, 1733: 1, 1729: 1, 1727: 1, 1720: 1, 1717: 1, 1716: 1, 1713: 1,
1711: 1, 1710: 1, 1709: 1, 1705: 1, 1699: 1, 1697: 1, 1696: 1, 1694: 1, 1693: 1,
1692: 1, 1690: 1, 1687: 1, 1686: 1, 1685: 1, 1683: 1, 1679: 1, 1675: 1, 1671: 1,
1670: 1, 1669: 1, 1665: 1, 1664: 1, 1663: 1, 1660: 1, 1659: 1, 1657: 1, 1655: 1,
1653: 1, 1646: 1, 1643: 1, 1642: 1, 1640: 1, 1632: 1, 1626: 1, 1624: 1, 1623: 1,
1622: 1, 1620: 1, 1617: 1, 1614: 1, 1613: 1, 1612: 1, 1611: 1, 1608: 1, 1607: 1,
1605: 1, 1604: 1, 1603: 1, 1602: 1, 1600: 1, 1599: 1, 1595: 1, 1593: 1, 1590: 1,
1587: 1, 1586: 1, 1581: 1, 1579: 1, 1577: 1, 1574: 1, 1572: 1, 1570: 1, 1568: 1,
1566: 1, 1565: 1, 1564: 1, 1560: 1, 1559: 1, 1557: 1, 1554: 1, 1552: 1, 1548: 1,
1547: 1, 1546: 1, 1545: 1, 1541: 1, 1540: 1, 1539: 1, 1538: 1, 1534: 1, 1531: 1,
1528: 1, 1527: 1, 1516: 1, 1515: 1, 1509: 1, 1507: 1, 1505: 1, 1496: 1, 1495: 1,
1493: 1, 1488: 1, 1485: 1, 1484: 1, 1479: 1, 1478: 1, 1473: 1, 1471: 1, 1470: 1,
1468: 1, 1464: 1, 1463: 1, 1459: 1, 1457: 1, 1455: 1, 1452: 1, 1451: 1, 1445: 1,
1442: 1, 1441: 1, 1440: 1, 1439: 1, 1438: 1, 1430: 1, 1427: 1, 1426: 1, 1425: 1,
1418: 1, 1415: 1, 1411: 1, 1407: 1, 1406: 1, 1404: 1, 1403: 1, 1398: 1, 1392: 1,
1391: 1, 1386: 1, 1384: 1, 1382: 1, 1378: 1, 1377: 1, 1375: 1, 1372: 1, 1364: 1,
1361: 1, 1358: 1, 1357: 1, 1348: 1, 1347: 1, 1345: 1, 1344: 1, 1343: 1, 1338: 1,
1337: 1, 1334: 1, 1333: 1, 1331: 1, 1326: 1, 1323: 1, 1318: 1, 1317: 1, 1316: 1,
1313: 1, 1305: 1, 1298: 1, 1297: 1, 1294: 1, 1293: 1, 1288: 1, 1283: 1, 1282: 1,
1281: 1, 1276: 1, 1273: 1, 1271: 1, 1270: 1, 1269: 1, 1266: 1, 1264: 1, 1262: 1,
1261: 1, 1257: 1, 1255: 1, 1246: 1, 1244: 1, 1242: 1, 1235: 1, 1234: 1, 1232: 1,
1231: 1, 1222: 1, 1220: 1, 1216: 1, 1214: 1, 1212: 1, 1211: 1, 1206: 1, 1205: 1,
1203: 1, 1190: 1, 1183: 1, 1181: 1, 1178: 1, 1176: 1, 1175: 1, 1171: 1, 1165: 1,
1160: 1, 1157: 1, 1153: 1, 1152: 1, 1150: 1, 1149: 1, 1148: 1, 1147: 1, 1129: 1,
1125: 1, 1123: 1, 1115: 1, 1110: 1, 1108: 1, 1103: 1, 1102: 1, 1100: 1, 1099: 1,
1098: 1, 1094: 1, 1092: 1, 1089: 1, 1088: 1, 1085: 1, 1082: 1, 1076: 1, 1075: 1,
1069: 1, 1067: 1, 1061: 1, 1060: 1, 1059: 1, 1058: 1, 1055: 1, 1052: 1, 1046: 1,
1043: 1, 1041: 1, 1040: 1, 1034: 1, 1031: 1, 1029: 1, 1027: 1, 1023: 1, 1022: 1,
1021: 1, 1019: 1, 1018: 1, 1017: 1, 1013: 1, 1011: 1, 1008: 1, 1007: 1, 1006: 1,
1005: 1, 1002: 1, 1001: 1, 998: 1, 997: 1, 996: 1, 995: 1, 994: 1, 993: 1, 989:
1, 987: 1, 982: 1, 979: 1, 977: 1, 976: 1, 973: 1, 968: 1, 967: 1, 964: 1, 960:
1, 955: 1, 953: 1, 952: 1, 948: 1, 947: 1, 945: 1, 941: 1, 931: 1, 921: 1, 919:
1, 918: 1, 917: 1, 915: 1, 914: 1, 913: 1, 907: 1, 903: 1, 899: 1, 898: 1, 897:
1, 896: 1, 893: 1, 891: 1, 889: 1, 872: 1, 868: 1, 860: 1, 859: 1, 857: 1, 856:
1, 855: 1, 850: 1, 849: 1, 841: 1, 833: 1, 815: 1, 813: 1, 790: 1, 777: 1, 771:
1, 770: 1, 768: 1, 767: 1, 764: 1, 763: 1, 755: 1, 746: 1, 728: 1, 716: 1, 712:
1, 710: 1, 706: 1, 702: 1, 696: 1, 687: 1, 671: 1, 664: 1, 634: 1, 633: 1, 608:
1, 607: 1, 599: 1, 593: 1, 582: 1, 572: 1, 565: 1, 555: 1, 534: 1, 527: 1, 490:
1, 487: 1, 482: 1, 433: 1})
```

[73]:
```
cv_log_error_array=[]
for i in alpha:
    clf = SGDClassifier(alpha=i, penalty='l2', loss='log', random_state=42)
    clf.fit(train_text_feature_onehotCoding, y_train)

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

```
For values of alpha =  1e-05 The log loss is: 1.2721829822908244
For values of alpha =  0.0001 The log loss is: 1.099474116965219
For values of alpha =  0.001 The log loss is: 1.1239105704089265
For values of alpha =  0.01 The log loss is: 1.2384219505980087
For values of alpha =  0.1 The log loss is: 1.4241586829998831
For values of alpha =  1 The log loss is: 1.642766769688418
```

[74]:
```
print_best_alpha(alpha, cv_log_error_array)
```

```
The best alpha is: 0.0001
The best alpha index is: 1
```

[75]:
```
# Simple model with only text data to evaluate its importance and check error
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_loss(y_cv, predict_y, labels=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(test_text_feature_onehotCoding)
print('For values of best alpha = ', alpha[best_alpha], "The test log loss is:
↪",log_loss(y_test, predict_y, labels=clf.classes_, eps=1e-15))
```

```
For values of best alpha =  0.0001 The train log loss is: 0.648485394704729
For values of best alpha =  0.0001 The cross validation log loss is:
1.099474116965219
For values of best alpha =  0.0001 The test log loss is: 1.235792920444658
```

[76]:
```
# Checking text overlap
def get_intersec_text(df):
    df_text_vec = CountVectorizer(min_df=3)
    df_text_fea = df_text_vec.fit_transform(df['TEXT'])
    df_text_features = df_text_vec.get_feature_names()

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

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")
```

```
96.274 % of word of test data appeared in train data
97.874 % of word of Cross Validation appeared in train data
```

## 0.8 Data prepration for Machine Learning models

[77]:
```
# Functions delcaration

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)

# 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)

    A =(((C.T)/(C.sum(axis=1))).T)

    B =(C/C.sum(axis=0))
    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()


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)

# 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_no))
        else:
            word = text_vec.get_feature_names()[v-(fea1_len+fea2_len)]
            yes_no = True if word in text.split() else False
            if yes_no:
                word_present += 1
                print(i, "Text feature [{}] present in test data point [{}]".
 ↪format(word,yes_no))

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

Combining all 3 features together

[78]:
```
# merging gene, variance and text features

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

train_gene_var_onehotCoding =␣
 ↪hstack((train_gene_feature_onehotCoding,train_variation_feature_onehotCoding))
test_gene_var_onehotCoding =␣
 ↪hstack((test_gene_feature_onehotCoding,test_variation_feature_onehotCoding))
cv_gene_var_onehotCoding =␣
 ↪hstack((cv_gene_feature_onehotCoding,cv_variation_feature_onehotCoding))
```

```
train_x_onehotCoding = hstack((train_gene_var_onehotCoding,␣
 ↪train_text_feature_onehotCoding)).tocsr()
train_y = np.array(list(train_df['Class']))

test_x_onehotCoding = hstack((test_gene_var_onehotCoding,␣
 ↪test_text_feature_onehotCoding)).tocsr()
test_y = np.array(list(test_df['Class']))

cv_x_onehotCoding = hstack((cv_gene_var_onehotCoding,␣
 ↪cv_text_feature_onehotCoding)).tocsr()
cv_y = np.array(list(cv_df['Class']))


train_gene_var_responseCoding = np.
 ↪hstack((train_gene_feature_responseCoding,train_variation_feature_responseCoding))
test_gene_var_responseCoding = np.
 ↪hstack((test_gene_feature_responseCoding,test_variation_feature_responseCoding))
cv_gene_var_responseCoding = np.
 ↪hstack((cv_gene_feature_responseCoding,cv_variation_feature_responseCoding))

train_x_responseCoding = np.hstack((train_gene_var_responseCoding,␣
 ↪train_text_feature_responseCoding))
test_x_responseCoding = np.hstack((test_gene_var_responseCoding,␣
 ↪test_text_feature_responseCoding))
cv_x_responseCoding = np.hstack((cv_gene_var_responseCoding,␣
 ↪cv_text_feature_responseCoding))


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)
print(" Response encoding features :")
print("(number of data points * number of features) in train data = ",␣
 ↪train_x_responseCoding.shape)
print("(number of data points * number of features) in test data = ",␣
 ↪test_x_responseCoding.shape)
print("(number of data points * number of features) in cross validation data␣
 ↪=", cv_x_responseCoding.shape)
```

```
One hot encoding features :
(number of data points * number of features) in train data =  (2124, 55457)
(number of data points * number of features) in test data =  (665, 55457)
```

```
(number of data points * number of features) in cross validation data = (532,
55457)
 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)
```

## 0.9 Building Machine Learning models

## 0.10 Naive Bayes

[79]:
```
# http://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.
 ↪MultinomialNB.html
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)
    # MultinomialNB is used for multi class classification
    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))

print_best_alpha(alpha, cv_log_error_array)
```

```
for alpha = 1e-05
Log Loss : 1.2772367409831071
for alpha = 0.0001
Log Loss : 1.2757405184068615
for alpha = 0.001
Log Loss : 1.270251486108716
for alpha = 0.1
Log Loss : 1.2464245671312086
for alpha = 1
Log Loss : 1.256857498516061
for alpha = 10
Log Loss : 1.3833717634434648
for alpha = 100
Log Loss : 1.3600948522454712
for alpha = 1000
Log Loss : 1.309119867422757
```

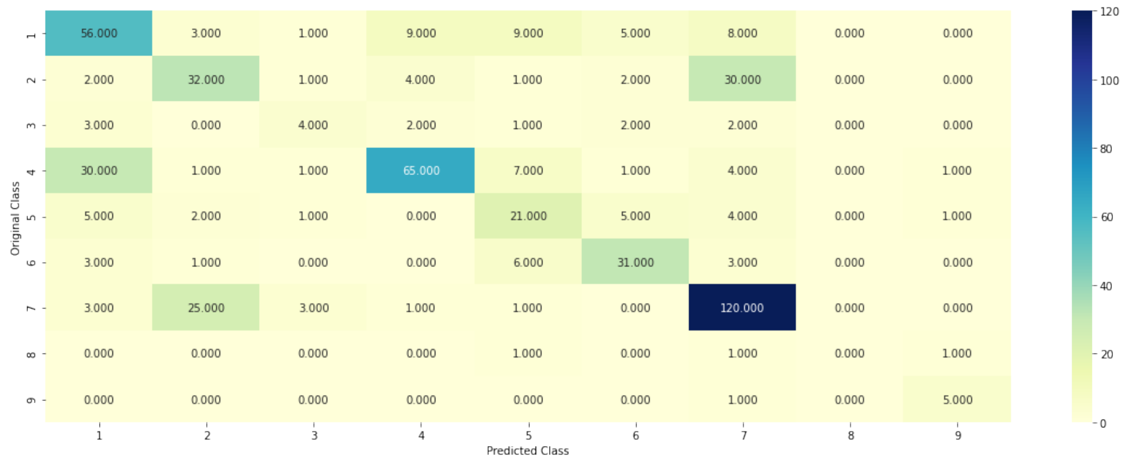
```
The best alpha is: 0.1
The best alpha index is: 3
```

[80]:
```
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))
```

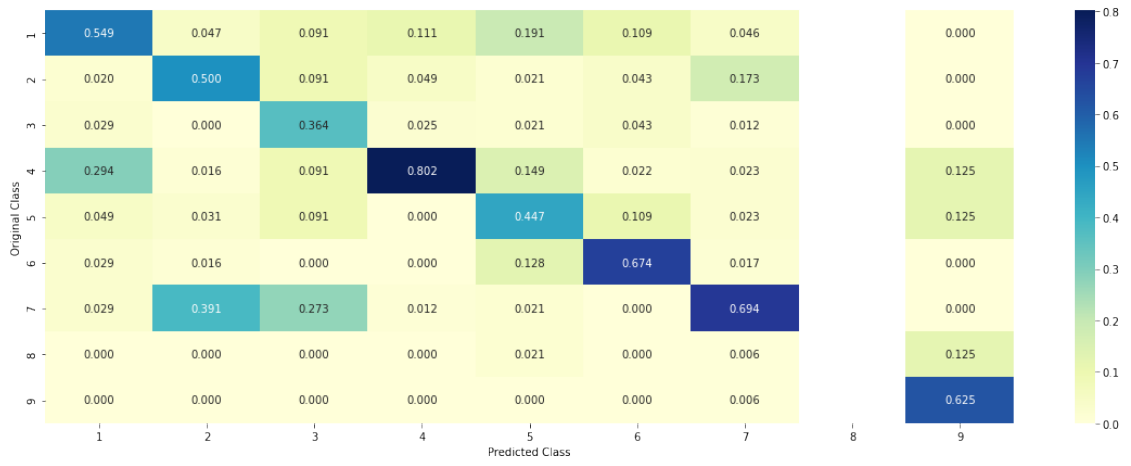
```
For values of best alpha =  0.1 The train log loss is: 0.8526757766833339
For values of best alpha =  0.1 The cross validation log loss is:
1.2464245671312086
For values of best alpha =  0.1 The test log loss is: 1.3128752779765938
```

[81]:
```
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()))
```

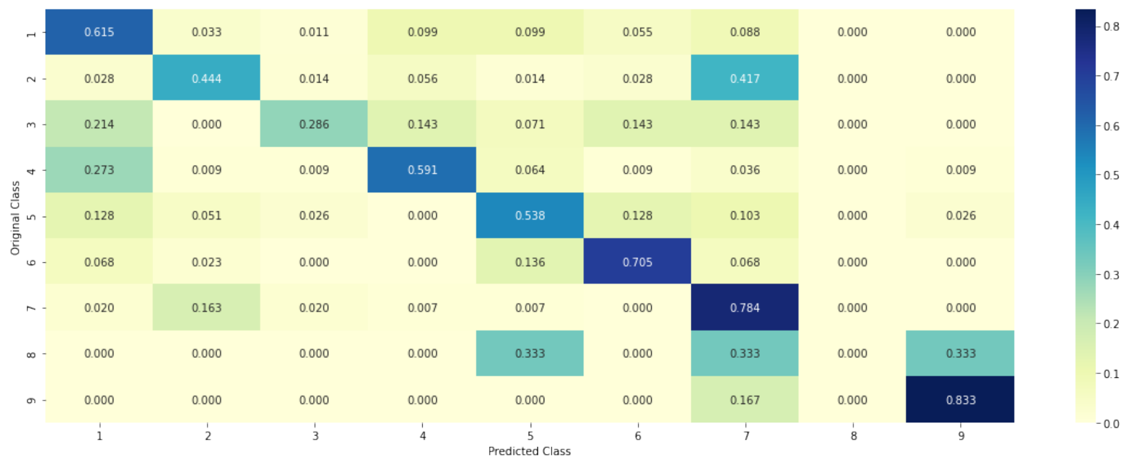
```
Log Loss : 1.2464245671312086
Number of missclassified point : 0.37218045112781956
-------------------- Confusion matrix --------------------
```



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



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



From precision and recall matrix diagonal values show correctly predicted values Checking these matrixes we can also see between which classes is confusion happening

Interpretability of Naive Bayes

[82]:
```
# checked item is defined apriory, without any meaning
# we can check whichever item we want
# defining item to be checked
test_point_index = 2
# important features to be printed
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: [[0.077  0.0634 0.014  0.1058 0.0332 0.0317
0.6676 0.004  0.0033]]
Actual Class : 7
--------------------------------------------------
16 Text feature [kinase] present in test data point [True]
17 Text feature [presence] present in test data point [True]
18 Text feature [activating] present in test data point [True]
19 Text feature [downstream] present in test data point [True]
```

```
20 Text feature [independent] present in test data point [True]
25 Text feature [inhibitor] present in test data point [True]
26 Text feature [well] present in test data point [True]
27 Text feature [expressing] present in test data point [True]
28 Text feature [recently] present in test data point [True]
29 Text feature [previously] present in test data point [True]
30 Text feature [activation] present in test data point [True]
31 Text feature [cell] present in test data point [True]
32 Text feature [contrast] present in test data point [True]
33 Text feature [cells] present in test data point [True]
34 Text feature [also] present in test data point [True]
35 Text feature [potential] present in test data point [True]
36 Text feature [shown] present in test data point [True]
37 Text feature [showed] present in test data point [True]
38 Text feature [growth] present in test data point [True]
39 Text feature [found] present in test data point [True]
40 Text feature [10] present in test data point [True]
41 Text feature [however] present in test data point [True]
42 Text feature [suggest] present in test data point [True]
43 Text feature [compared] present in test data point [True]
44 Text feature [similar] present in test data point [True]
45 Text feature [higher] present in test data point [True]
46 Text feature [observed] present in test data point [True]
47 Text feature [addition] present in test data point [True]
48 Text feature [mutations] present in test data point [True]
49 Text feature [treated] present in test data point [True]
50 Text feature [described] present in test data point [True]
52 Text feature [may] present in test data point [True]
53 Text feature [obtained] present in test data point [True]
54 Text feature [factor] present in test data point [True]
55 Text feature [proliferation] present in test data point [True]
57 Text feature [1a] present in test data point [True]
58 Text feature [approximately] present in test data point [True]
59 Text feature [interestingly] present in test data point [True]
60 Text feature [inhibition] present in test data point [True]
61 Text feature [using] present in test data point [True]
62 Text feature [3a] present in test data point [True]
63 Text feature [respectively] present in test data point [True]
64 Text feature [identified] present in test data point [True]
65 Text feature [activated] present in test data point [True]
66 Text feature [3b] present in test data point [True]
67 Text feature [reported] present in test data point [True]
68 Text feature [including] present in test data point [True]
69 Text feature [12] present in test data point [True]
70 Text feature [phosphorylation] present in test data point [True]
71 Text feature [studies] present in test data point [True]
72 Text feature [followed] present in test data point [True]
73 Text feature [various] present in test data point [True]
```

```
74 Text feature [total] present in test data point [True]
75 Text feature [inhibitors] present in test data point [True]
76 Text feature [new] present in test data point [True]
77 Text feature [three] present in test data point [True]
78 Text feature [mechanism] present in test data point [True]
79 Text feature [fig] present in test data point [True]
81 Text feature [confirmed] present in test data point [True]
82 Text feature [occur] present in test data point [True]
83 Text feature [either] present in test data point [True]
84 Text feature [molecular] present in test data point [True]
85 Text feature [consistent] present in test data point [True]
87 Text feature [confirm] present in test data point [True]
88 Text feature [sensitive] present in test data point [True]
89 Text feature [suggests] present in test data point [True]
90 Text feature [although] present in test data point [True]
91 Text feature [two] present in test data point [True]
92 Text feature [mutation] present in test data point [True]
93 Text feature [different] present in test data point [True]
94 Text feature [due] present in test data point [True]
95 Text feature [increase] present in test data point [True]
96 Text feature [figure] present in test data point [True]
97 Text feature [show] present in test data point [True]
98 Text feature [revealed] present in test data point [True]
99 Text feature [enhanced] present in test data point [True]
Out of the top  100  features  76 are present in query point
```

May have to check comparing with a Naive Bayes builded using only text data as input

Looking at another item

[83]:
```
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 : 5
Predicted Class Probabilities: [[0.1016 0.0834 0.0185 0.1394 0.4425 0.0423
0.1628 0.0052 0.0043]]
Actual Class : 4
--------------------------------------------------
```

```
5 Text feature [neutral] present in test data point [True]
8 Text feature [assays] present in test data point [True]
12 Text feature [functional] present in test data point [True]
Out of the top  100  features  3 are present in query point
```

## 1 K Nearest Neighbour Classification

[84]:
```
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))
print("")
print_best_alpha(alpha, cv_log_error_array)
```

```
for alpha = 5
Log Loss : 1.0587095490258613
for alpha = 11
Log Loss : 1.074858782876268
for alpha = 15
Log Loss : 1.0830905463973681
for alpha = 21
Log Loss : 1.095072992000385
for alpha = 31
Log Loss : 1.114826777681961
for alpha = 41
Log Loss : 1.13301441582936
for alpha = 51
Log Loss : 1.1478034524781693
for alpha = 99
Log Loss : 1.156847776021154

The best alpha is: 5
The best alpha index is: 0
```

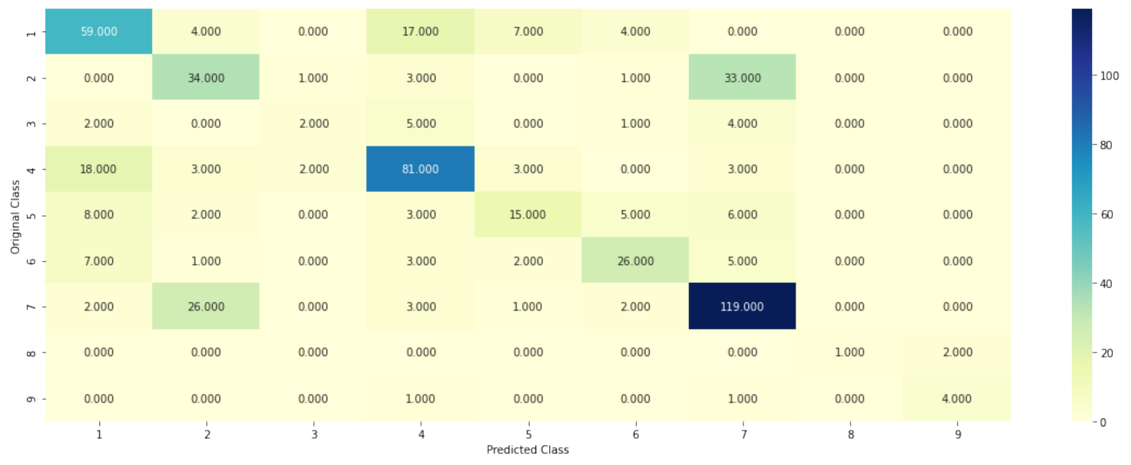
May want to dense alpha values between 5 and 15 for a better alpha, but for the purpose of this kernel will take 11

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

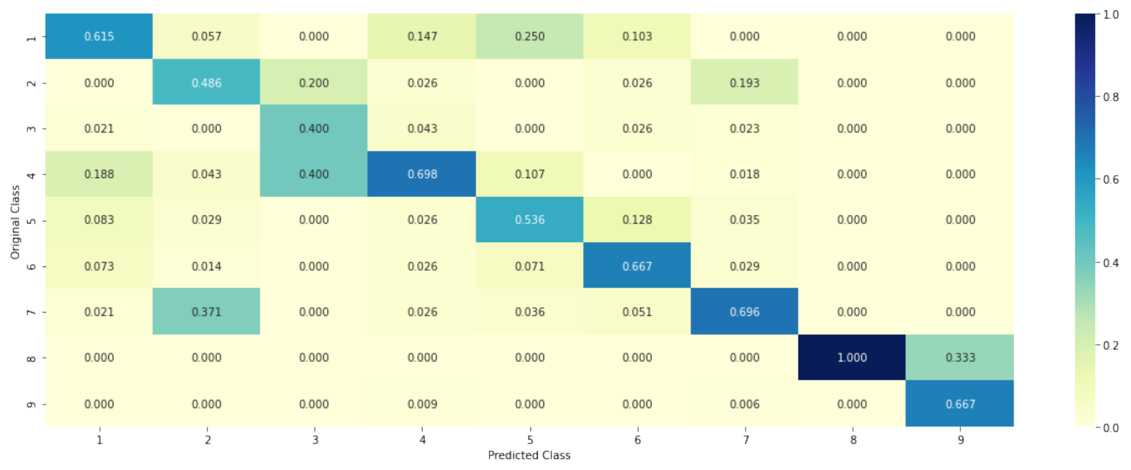
```
For values of best alpha =  5 The train log loss is: 0.44790408188976055
For values of best alpha =  5 The cross validation log loss is:
1.0587095490258613
For values of best alpha =  5 The test log loss is: 1.1368923111769251
```

```
[86]: clf = KNeighborsClassifier(n_neighbors=alpha[best_alpha])
      predict_and_plot_confusion_matrix(train_x_responseCoding, train_y,␣
       ↪cv_x_responseCoding, cv_y, clf)
```

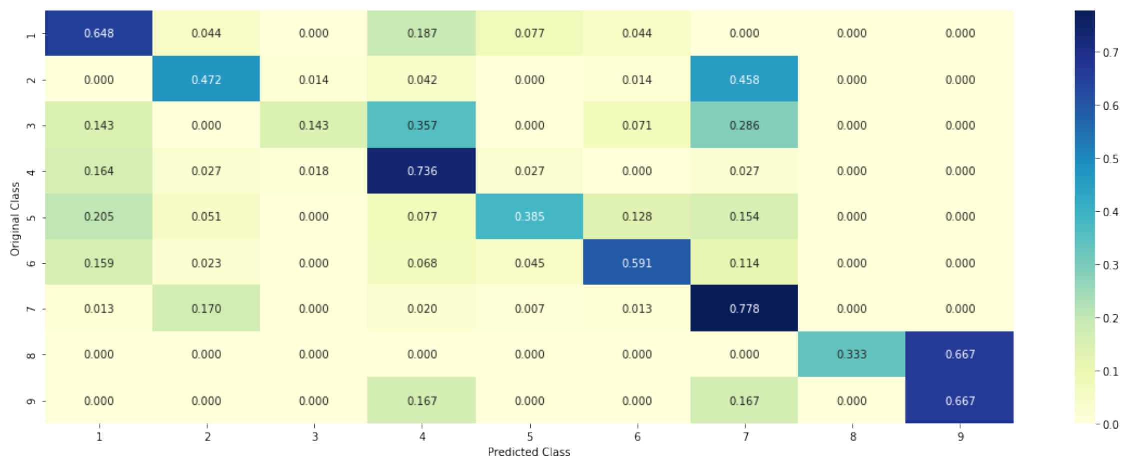
```
Log loss : 1.0587095490258613
Number of mis-classified points : 0.35902255639097747
-------------------- Confusion matrix --------------------
```



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



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



[87]:
```
# Lets look at few test points
clf = KNeighborsClassifier(n_neighbors=alpha[best_alpha])
clf.fit(train_x_responseCoding, train_y)
sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
sig_clf.fit(train_x_responseCoding, train_y)

test_point_index = 1
predicted_cls = sig_clf.predict(test_x_responseCoding[0].reshape(1,-1))
print("Predicted Class :", predicted_cls[0])
print("Actual Class :", test_y[test_point_index])
neighbors = clf.kneighbors(test_x_responseCoding[test_point_index].reshape(1,␣
 ↪-1), alpha[best_alpha])
```

```
print("The ",alpha[best_alpha]," nearest neighbours of the test points belongs␣
↪to classes",train_y[neighbors[1][0]])
print("Fequency of nearest points :",Counter(train_y[neighbors[1][0]]))
```

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

[88]:
```
clf = KNeighborsClassifier(n_neighbors=alpha[best_alpha])
clf.fit(train_x_responseCoding, train_y)
sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
sig_clf.fit(train_x_responseCoding, train_y)

test_point_index = 100

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

```
Predicted Class : 4
Actual Class : 4
the k value for knn is 5 and the nearest neighbours of the test points belongs
to classes [5 4 4 4 4]
Fequency of nearest points : Counter({4: 4, 5: 1})
```

## 2 Logistic Regression

logistic regression with ballancing of classes

[90]:
```
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))
print("")
print_best_alpha(alpha, cv_log_error_array)
```

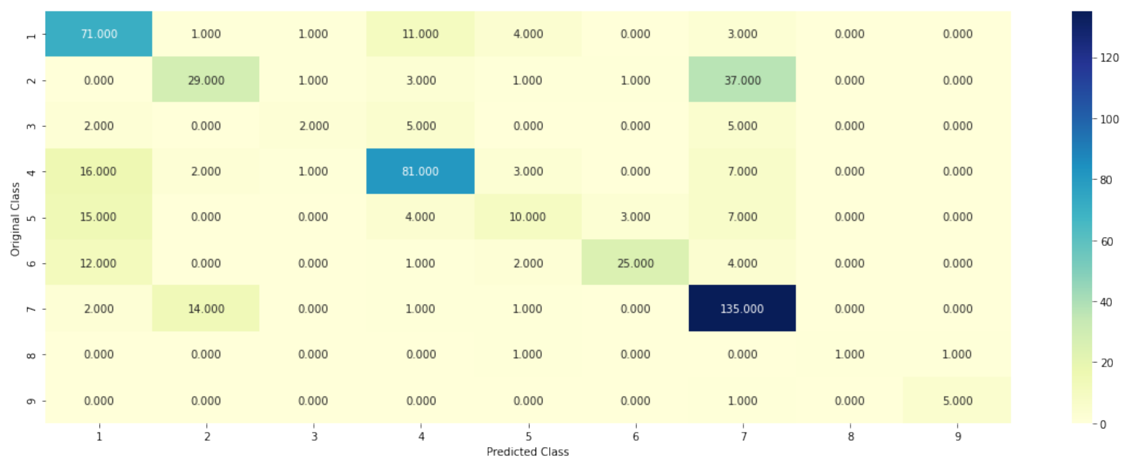
```
for alpha = 1e-06
Log Loss : 1.3497093406473863
for alpha = 1e-05
Log Loss : 1.2950280592437216
for alpha = 0.0001
Log Loss : 1.1110092263689428
for alpha = 0.001
Log Loss : 1.074115673621701
for alpha = 0.01
Log Loss : 1.1314175972482177
for alpha = 0.1
Log Loss : 1.4648396344774646
for alpha = 1
Log Loss : 1.694290394268692
for alpha = 10
Log Loss : 1.7223210557367996
for alpha = 100
Log Loss : 1.7253925391967233

The best alpha is: 0.001
The best alpha index is: 3
```

[91]:
```
best_alpha = np.argmin(cv_log_error_array)
clf = SGDClassifier(class_weight='balanced', alpha=alpha[best_alpha],␣
↪penalty='l2', loss='log', random_state=42)
clf.fit(train_x_onehotCoding, 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 values of best alpha =  0.001 The train log loss is: 0.5105776633624867
For values of best alpha =  0.001 The cross validation log loss is:
1.074115673621701
For values of best alpha =  0.001 The test log loss is: 1.1526170372094005
```

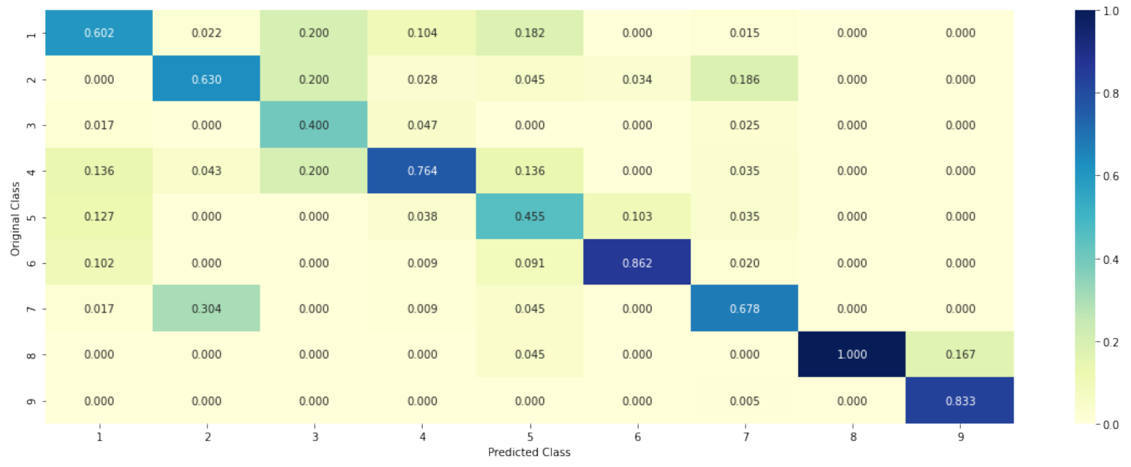
[92]:
```
clf = SGDClassifier(class_weight='balanced', alpha=alpha[best_alpha],␣
 ↪penalty='l2', loss='log', random_state=42)
predict_and_plot_confusion_matrix(train_x_onehotCoding, train_y,␣
 ↪cv_x_onehotCoding, cv_y, clf)
```

```
Log loss : 1.074115673621701
Number of mis-classified points : 0.325187969924812
-------------------- Confusion matrix --------------------
```



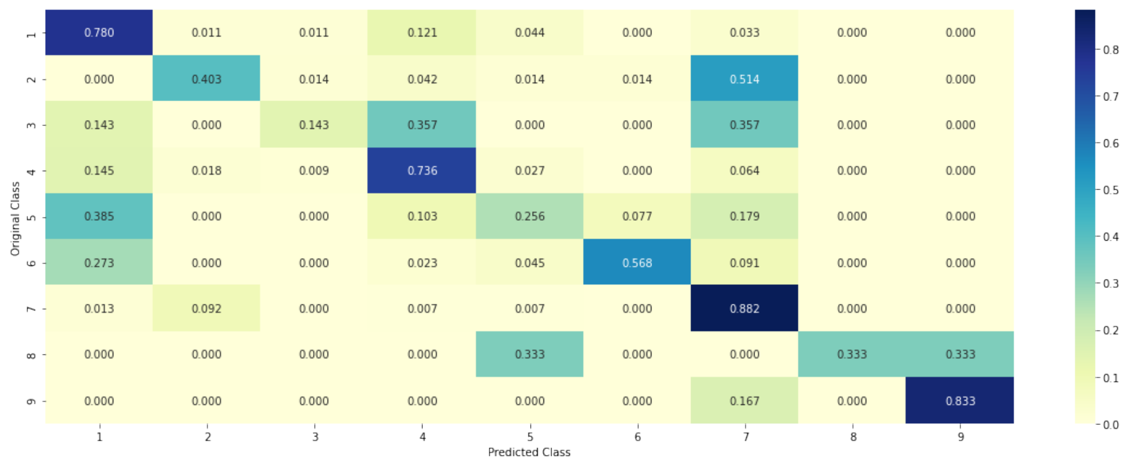
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 71.000 | 1.000 | 1.000 | 11.000 | 4.000 | 0.000 | 3.000 | 0.000 | 0.000 |
| 2 | 0.000 | 29.000 | 1.000 | 3.000 | 1.000 | 1.000 | 37.000 | 0.000 | 0.000 |
| 3 | 2.000 | 0.000 | 2.000 | 5.000 | 0.000 | 0.000 | 5.000 | 0.000 | 0.000 |
| 4 | 16.000 | 2.000 | 1.000 | 81.000 | 3.000 | 0.000 | 7.000 | 0.000 | 0.000 |
| 5 | 15.000 | 0.000 | 0.000 | 4.000 | 10.000 | 3.000 | 7.000 | 0.000 | 0.000 |
| 6 | 12.000 | 0.000 | 0.000 | 1.000 | 2.000 | 25.000 | 4.000 | 0.000 | 0.000 |
| 7 | 2.000 | 14.000 | 0.000 | 1.000 | 1.000 | 0.000 | 135.000 | 0.000 | 0.000 |
| 8 | 0.000 | 0.000 | 0.000 | 0.000 | 1.000 | 0.000 | 0.000 | 1.000 | 1.000 |
| 9 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 1.000 | 0.000 | 5.000 |

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



|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0.602 | 0.022 | 0.200 | 0.104 | 0.182 | 0.000 | 0.015 | 0.000 | 0.000 |
| 2 | 0.000 | 0.630 | 0.200 | 0.028 | 0.045 | 0.034 | 0.186 | 0.000 | 0.000 |
| 3 | 0.017 | 0.000 | 0.400 | 0.047 | 0.000 | 0.000 | 0.025 | 0.000 | 0.000 |
| 4 | 0.136 | 0.043 | 0.200 | 0.764 | 0.136 | 0.000 | 0.035 | 0.000 | 0.000 |
| 5 | 0.127 | 0.000 | 0.000 | 0.038 | 0.455 | 0.103 | 0.035 | 0.000 | 0.000 |
| 6 | 0.102 | 0.000 | 0.000 | 0.009 | 0.091 | 0.862 | 0.020 | 0.000 | 0.000 |
| 7 | 0.017 | 0.304 | 0.000 | 0.009 | 0.045 | 0.000 | 0.678 | 0.000 | 0.000 |
| 8 | 0.000 | 0.000 | 0.000 | 0.000 | 0.045 | 0.000 | 0.000 | 1.000 | 0.167 |
| 9 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.005 | 0.000 | 0.833 |

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



Logistic regression interpreability

[93]:
```
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']))
```

[94]:
```
# from tabulate import tabulate
clf = SGDClassifier(class_weight='balanced', alpha=alpha[best_alpha],␣
↪penalty='l2', loss='log', random_state=42)
```

```
clf.fit(train_x_onehotCoding,train_y)
test_point_index = 1
no_feature = 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: [[0.0831 0.2388 0.0223 0.0914 0.0474 0.0592
0.4401 0.0075 0.0103]]
Actual Class : 7
--------------------------------------------------
24 Text feature [constitutively] present in test data point [True]
35 Text feature [nude] present in test data point [True]
45 Text feature [activated] present in test data point [True]
49 Text feature [constitutive] present in test data point [True]
79 Text feature [oncogene] present in test data point [True]
137 Text feature [extracellular] present in test data point [True]
143 Text feature [tk1] present in test data point [True]
179 Text feature [activation] present in test data point [True]
211 Text feature [ligand] present in test data point [True]
239 Text feature [stably] present in test data point [True]
242 Text feature [tk2] present in test data point [True]
272 Text feature [transformation] present in test data point [True]
289 Text feature [kinase] present in test data point [True]
328 Text feature [tyrosine] present in test data point [True]
349 Text feature [ba] present in test data point [True]
366 Text feature [murine] present in test data point [True]
400 Text feature [f3] present in test data point [True]
Out of the top  500  features  17 are present in query point
```

[95]:
```
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: [[0.2025 0.0121 0.0157 0.3599 0.308  0.0915
0.0034 0.004  0.0029]]
Actual Class : 4
--------------------------------------------------
137 Text feature [instability] present in test data point [True]
190 Text feature [382] present in test data point [True]
202 Text feature [adapted] present in test data point [True]
311 Text feature [cmmrd] present in test data point [True]
468 Text feature [microsatellite] present in test data point [True]
Out of the top  500  features  5 are present in query point
```

Notes from some different runs: None of top 100 and 2 of top 500 top features on this particular missclassified point None of top 100 and 18 of top 500 top features on this particular missclassified point None of top 200 and 7 of top 500 top features on this particular missclassified point None of top 100 seems constent confusion (logically)

Logistic regression without balancing

[96]:
```
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))
print("")
print_best_alpha(alpha, cv_log_error_array)
```

```
for alpha = 1e-06
Log Loss : 1.2908779040202831
for alpha = 1e-05
Log Loss : 1.2590311356289654
for alpha = 0.0001
Log Loss : 1.1075231754795696
for alpha = 0.001
Log Loss : 1.0815711224720403
for alpha = 0.01
```

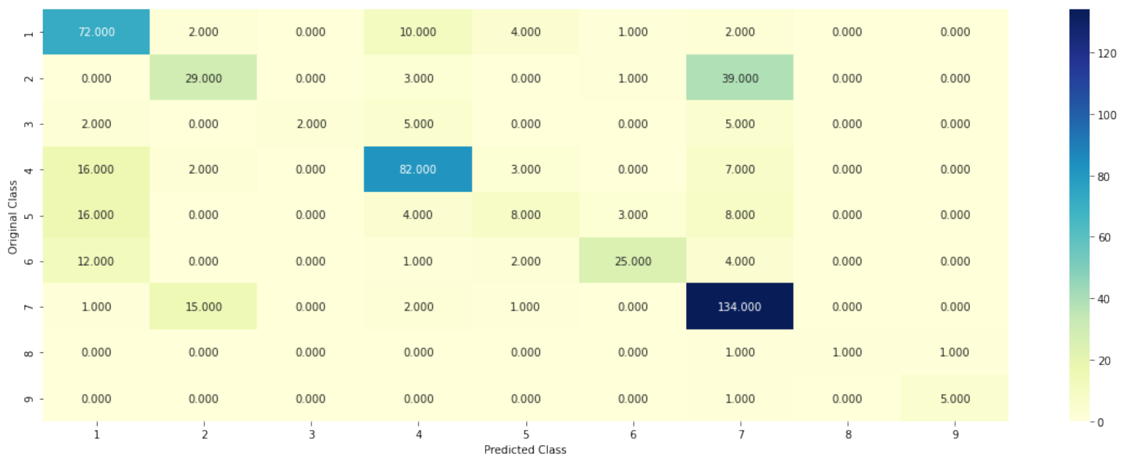
```
Log Loss : 1.1773799272301868
for alpha = 0.1
Log Loss : 1.3256253560462177
for alpha = 1
Log Loss : 1.5740467277372516

The best alpha is: 0.001
The best alpha index is: 3
```

[97]:
```
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_loss(y_cv, predict_y, labels=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(test_x_onehotCoding)
print('For values of best alpha = ', alpha[best_alpha], "The test log loss is:
 ↪",log_loss(y_test, predict_y, labels=clf.classes_, eps=1e-15))
```

```
For values of best alpha =  0.001 The train log loss is: 0.5085015257966655
For values of best alpha =  0.001 The cross validation log loss is:
1.0815711224720403
For values of best alpha =  0.001 The test log loss is: 1.1634584797518273
```

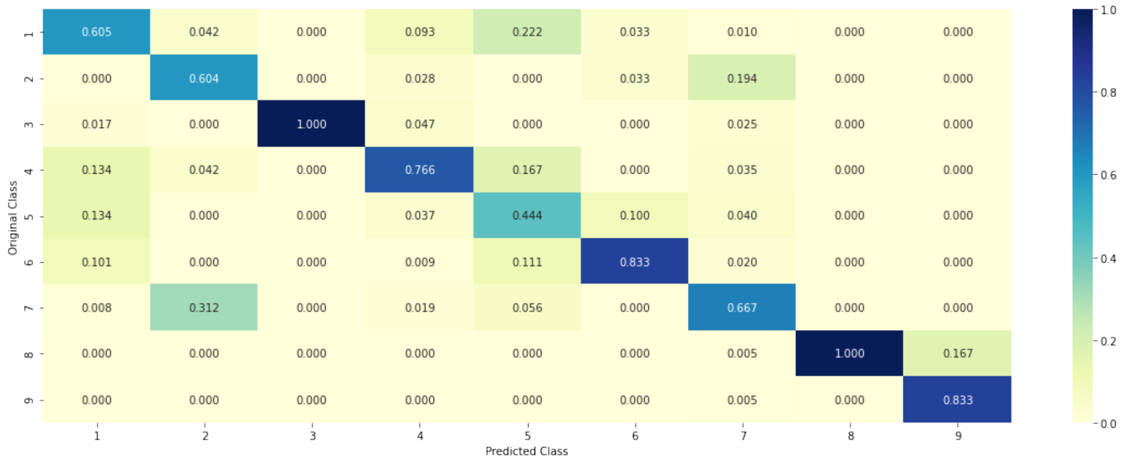
[98]:
```
clf = SGDClassifier(alpha=alpha[best_alpha], penalty='l2', loss='log',␣
 ↪random_state=42)
predict_and_plot_confusion_matrix(train_x_onehotCoding, train_y,␣
 ↪cv_x_onehotCoding, cv_y, clf)
```

```
Log loss : 1.0815711224720403
Number of mis-classified points : 0.32706766917293234
-------------------- Confusion matrix --------------------
```



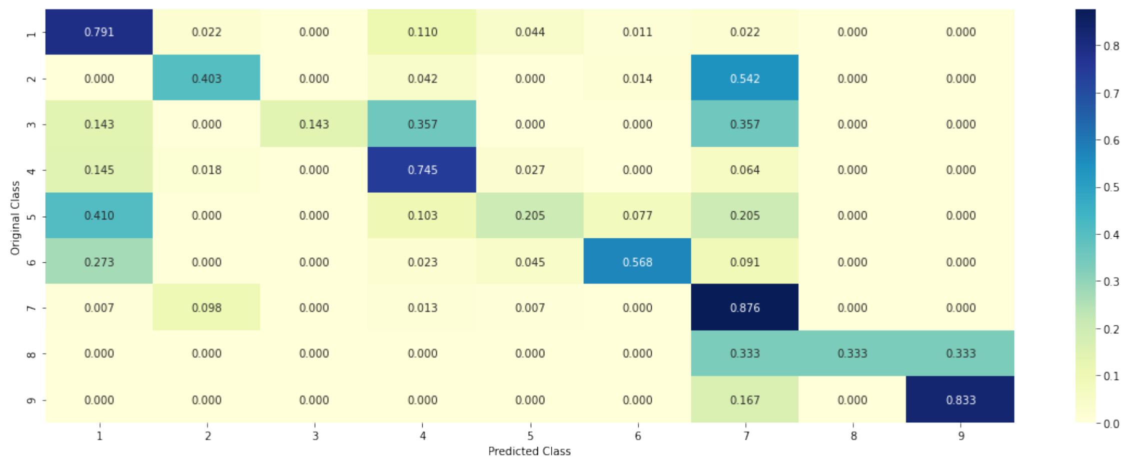
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 72.000 | 2.000 | 0.000 | 10.000 | 4.000 | 1.000 | 2.000 | 0.000 | 0.000 |
| 2 | 0.000 | 29.000 | 0.000 | 3.000 | 0.000 | 1.000 | 39.000 | 0.000 | 0.000 |
| 3 | 2.000 | 0.000 | 2.000 | 5.000 | 0.000 | 0.000 | 5.000 | 0.000 | 0.000 |
| 4 | 16.000 | 2.000 | 0.000 | 82.000 | 3.000 | 0.000 | 7.000 | 0.000 | 0.000 |
| 5 | 16.000 | 0.000 | 0.000 | 4.000 | 8.000 | 3.000 | 8.000 | 0.000 | 0.000 |
| 6 | 12.000 | 0.000 | 0.000 | 1.000 | 2.000 | 25.000 | 4.000 | 0.000 | 0.000 |
| 7 | 1.000 | 15.000 | 0.000 | 2.000 | 1.000 | 0.000 | 134.000 | 0.000 | 0.000 |
| 8 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 1.000 | 1.000 | 1.000 |
| 9 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 1.000 | 0.000 | 5.000 |

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



| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0.605 | 0.042 | 0.000 | 0.093 | 0.222 | 0.033 | 0.010 | 0.000 | 0.000 |
| 2 | 0.000 | 0.604 | 0.000 | 0.028 | 0.000 | 0.033 | 0.194 | 0.000 | 0.000 |
| 3 | 0.017 | 0.000 | 1.000 | 0.047 | 0.000 | 0.000 | 0.025 | 0.000 | 0.000 |
| 4 | 0.134 | 0.042 | 0.000 | 0.766 | 0.167 | 0.000 | 0.035 | 0.000 | 0.000 |
| 5 | 0.134 | 0.000 | 0.000 | 0.037 | 0.444 | 0.100 | 0.040 | 0.000 | 0.000 |
| 6 | 0.101 | 0.000 | 0.000 | 0.009 | 0.111 | 0.833 | 0.020 | 0.000 | 0.000 |
| 7 | 0.008 | 0.312 | 0.000 | 0.019 | 0.056 | 0.000 | 0.667 | 0.000 | 0.000 |
| 8 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.005 | 1.000 | 0.167 |
| 9 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.005 | 0.000 | 0.833 |

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



Testing query point and interpretability

[99]:
```python
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 : 7
Predicted Class Probabilities: [[0.0738 0.2264 0.0285 0.079  0.0496 0.0592
0.4589 0.0096 0.015 ]]
Actual Class : 7
--------------------------------------------------
52 Text feature [constitutively] present in test data point [True]
110 Text feature [nude] present in test data point [True]
122 Text feature [activated] present in test data point [True]
125 Text feature [constitutive] present in test data point [True]
180 Text feature [oncogene] present in test data point [True]
273 Text feature [stably] present in test data point [True]
278 Text feature [extracellular] present in test data point [True]
289 Text feature [tk1] present in test data point [True]
```

```
297 Text feature [activation] present in test data point [True]
317 Text feature [transformation] present in test data point [True]
339 Text feature [ba] present in test data point [True]
341 Text feature [kinase] present in test data point [True]
396 Text feature [f3] present in test data point [True]
425 Text feature [ligand] present in test data point [True]
447 Text feature [tk2] present in test data point [True]
486 Text feature [tyrosine] present in test data point [True]
Out of the top  500  features  16 are present in query point
```

## 3 Linear Support Vector Machines

[100]:
```
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))
print("")
print_best_alpha(alpha, cv_log_error_array)
```

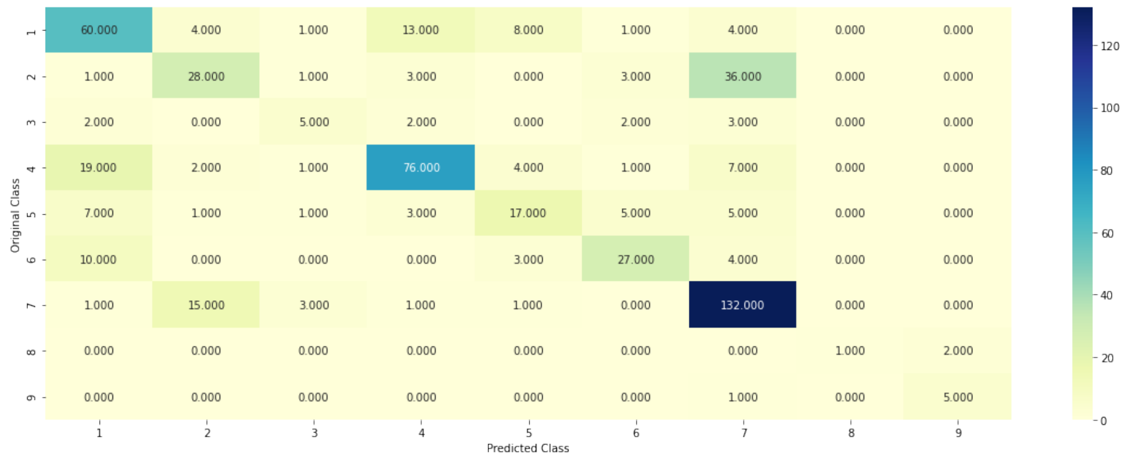
```
for C = 1e-05
Log Loss : 1.332081971282043
for C = 0.0001
Log Loss : 1.241496544243648
for C = 0.001
Log Loss : 1.0951933226853459
for C = 0.01
Log Loss : 1.081277561931896
for C = 0.1
Log Loss : 1.3763349868914128
for C = 1
Log Loss : 1.7122342245702478
for C = 10
Log Loss : 1.7259457630017707
for C = 100
Log Loss : 1.725945566813617

The best alpha is: 0.01
The best alpha index is: 3
```

[101]:
```
best_alpha = np.argmin(cv_log_error_array)
# clf = SVC(C=i,kernel='linear',probability=True, class_weight='balanced')
clf = SGDClassifier(class_weight='balanced', alpha=alpha[best_alpha],␣
↪penalty='l2', loss='hinge', random_state=42)
clf.fit(train_x_onehotCoding, train_y)
sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
sig_clf.fit(train_x_onehotCoding, train_y)

predict_y = sig_clf.predict_proba(train_x_onehotCoding)
print('For values of best alpha = ', alpha[best_alpha], "The train log loss is:
↪",log_loss(y_train, predict_y, labels=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))
```

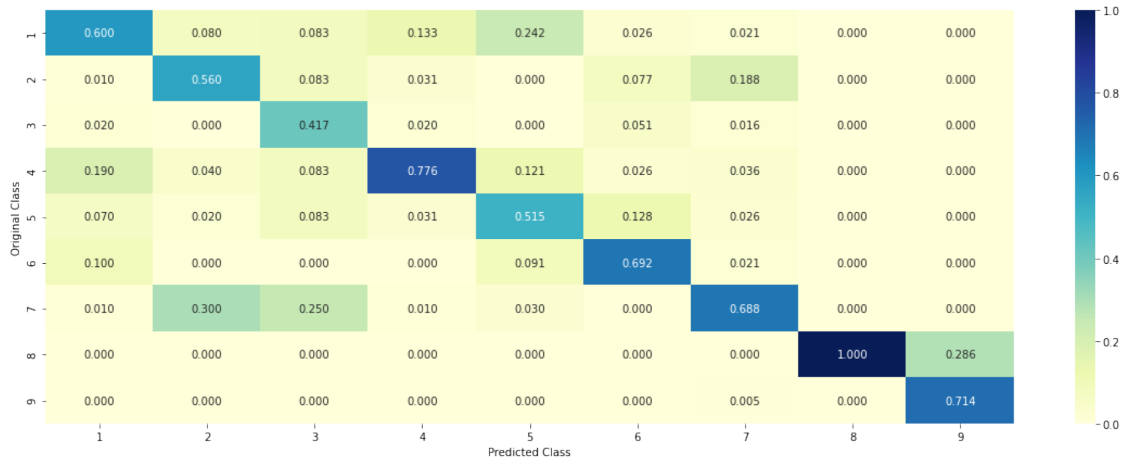
```
For values of best alpha =  0.01 The train log loss is: 0.7177803857393527
For values of best alpha =  0.01 The cross validation log loss is:
1.081277561931896
For values of best alpha =  0.01 The test log loss is: 1.19027652332489
```

[102]:
```
clf = SGDClassifier(alpha=alpha[best_alpha], penalty='l2', loss='hinge',␣
↪random_state=42,class_weight='balanced')
predict_and_plot_confusion_matrix(train_x_onehotCoding,␣
↪train_y,cv_x_onehotCoding,cv_y, clf)
```

```
Log loss : 1.081277561931896
Number of mis-classified points : 0.34022556390977443
-------------------- Confusion matrix --------------------
```

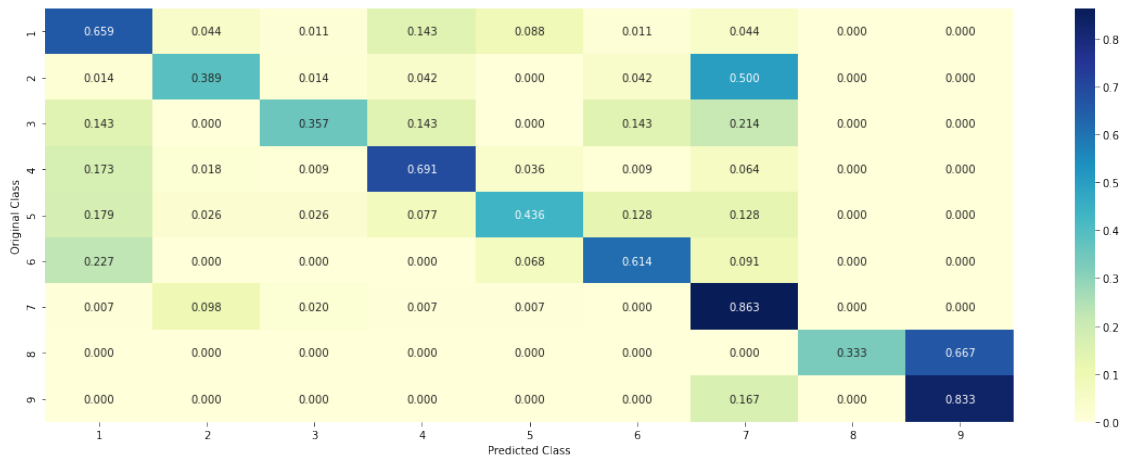


```
-------------------- Precision matrix (Columm Sum=1) --------------------
```



|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0.600 | 0.080 | 0.083 | 0.133 | 0.242 | 0.026 | 0.021 | 0.000 | 0.000 |
| 2 | 0.010 | 0.560 | 0.083 | 0.031 | 0.000 | 0.077 | 0.188 | 0.000 | 0.000 |
| 3 | 0.020 | 0.000 | 0.417 | 0.020 | 0.000 | 0.051 | 0.016 | 0.000 | 0.000 |
| 4 | 0.190 | 0.040 | 0.083 | 0.776 | 0.121 | 0.026 | 0.036 | 0.000 | 0.000 |
| 5 | 0.070 | 0.020 | 0.083 | 0.031 | 0.515 | 0.128 | 0.026 | 0.000 | 0.000 |
| 6 | 0.100 | 0.000 | 0.000 | 0.000 | 0.091 | 0.692 | 0.021 | 0.000 | 0.000 |
| 7 | 0.010 | 0.300 | 0.250 | 0.010 | 0.030 | 0.000 | 0.688 | 0.000 | 0.000 |
| 8 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 1.000 | 0.286 |
| 9 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.005 | 0.000 | 0.714 |

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



|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0.659 | 0.044 | 0.011 | 0.143 | 0.088 | 0.011 | 0.044 | 0.000 | 0.000 |
| 2 | 0.014 | 0.389 | 0.014 | 0.042 | 0.000 | 0.042 | 0.500 | 0.000 | 0.000 |
| 3 | 0.143 | 0.000 | 0.357 | 0.143 | 0.000 | 0.143 | 0.214 | 0.000 | 0.000 |
| 4 | 0.173 | 0.018 | 0.009 | 0.691 | 0.036 | 0.009 | 0.064 | 0.000 | 0.000 |
| 5 | 0.179 | 0.026 | 0.026 | 0.077 | 0.436 | 0.128 | 0.128 | 0.000 | 0.000 |
| 6 | 0.227 | 0.000 | 0.000 | 0.000 | 0.068 | 0.614 | 0.091 | 0.000 | 0.000 |
| 7 | 0.007 | 0.098 | 0.020 | 0.007 | 0.007 | 0.000 | 0.863 | 0.000 | 0.000 |
| 8 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.333 | 0.667 |
| 9 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.167 | 0.000 | 0.833 |

[103]:
```
clf = SGDClassifier(alpha=alpha[best_alpha], penalty='l2', loss='hinge',␣
 ↪random_state=42)
clf.fit(train_x_onehotCoding,train_y)
test_point_index = 1
# test_point_index = 100
no_feature = 500
predicted_cls = sig_clf.predict(test_x_onehotCoding[test_point_index])
print("Predicted Class :", predicted_cls[0])
print("Predicted Class Probabilities:", np.round(sig_clf.
↪predict_proba(test_x_onehotCoding[test_point_index]),4))
print("Actual Class :", test_y[test_point_index])
indices = np.argsort(-clf.coef_)[predicted_cls-1][:,:no_feature]
print("-"*50)
get_impfeature_names(indices[0], test_df['TEXT'].
↪iloc[test_point_index],test_df['Gene'].
↪iloc[test_point_index],test_df['Variation'].iloc[test_point_index],␣
↪no_feature)
```

```
Predicted Class : 7
Predicted Class Probabilities: [[0.1217 0.2478 0.0217 0.1349 0.0571 0.0543
0.3469 0.0075 0.008 ]]
Actual Class : 7
--------------------------------------------------
36 Text feature [constitutively] present in test data point [True]
38 Text feature [nude] present in test data point [True]
74 Text feature [activated] present in test data point [True]
135 Text feature [tk1] present in test data point [True]
159 Text feature [oncogene] present in test data point [True]
166 Text feature [stably] present in test data point [True]
182 Text feature [constitutive] present in test data point [True]
223 Text feature [tk2] present in test data point [True]
239 Text feature [extracellular] present in test data point [True]
306 Text feature [activation] present in test data point [True]
398 Text feature [grew] present in test data point [True]
469 Text feature [kinase] present in test data point [True]
Out of the top  500  features  12 are present in query point
```

## 4 Random Forest Classifier

[104]:
```
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))
```

```
for n_estimators = 100 and max depth =  5
Log Loss : 1.2565422522238885
for n_estimators = 100 and max depth =  10
Log Loss : 1.164423370599573
for n_estimators = 200 and max depth =  5
Log Loss : 1.2479314765592489
for n_estimators = 200 and max depth =  10
Log Loss : 1.158147703478899
for n_estimators = 500 and max depth =  5
Log Loss : 1.2401119952488662
for n_estimators = 500 and max depth =  10
Log Loss : 1.1520341944684371
for n_estimators = 1000 and max depth =  5
Log Loss : 1.2407551791690425
for n_estimators = 1000 and max depth =  10
Log Loss : 1.1517907344797522
for n_estimators = 2000 and max depth =  5
Log Loss : 1.2374495389483529
for n_estimators = 2000 and max depth =  10
Log Loss : 1.1513661957058137
```

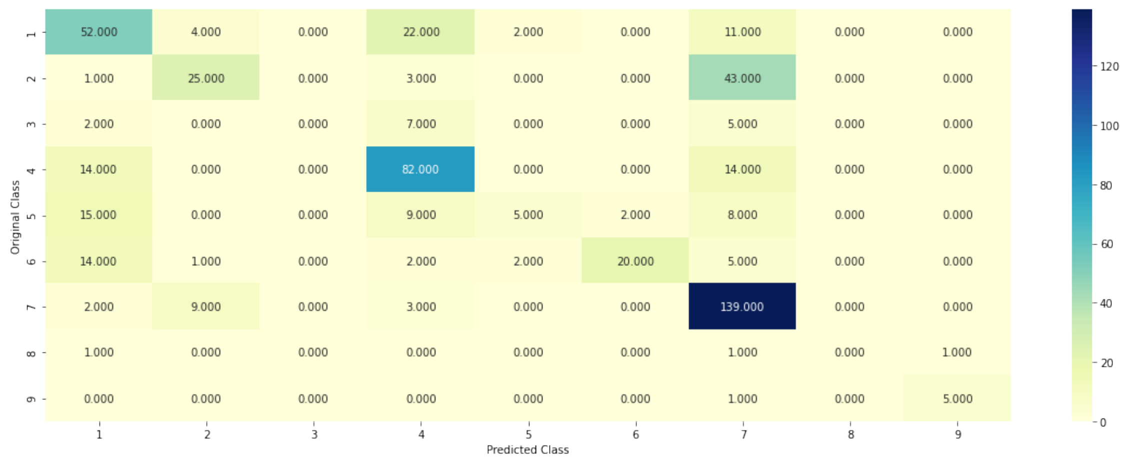
[105]:
```
best_alpha = np.argmin(cv_log_error_array)
clf = RandomForestClassifier(n_estimators=alpha[int(best_alpha/2)],␣
 ↪criterion='gini', max_depth=max_depth[int(best_alpha%2)], random_state=42,␣
 ↪n_jobs=-1)
clf.fit(train_x_onehotCoding, train_y)
sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
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 values of best estimator =  2000 The train log loss is: 0.694672023605049
For values of best estimator =  2000 The cross validation log loss is:
1.1513661957058137
For values of best estimator =  2000 The test log loss is: 1.1785550523354904
```

[106]:

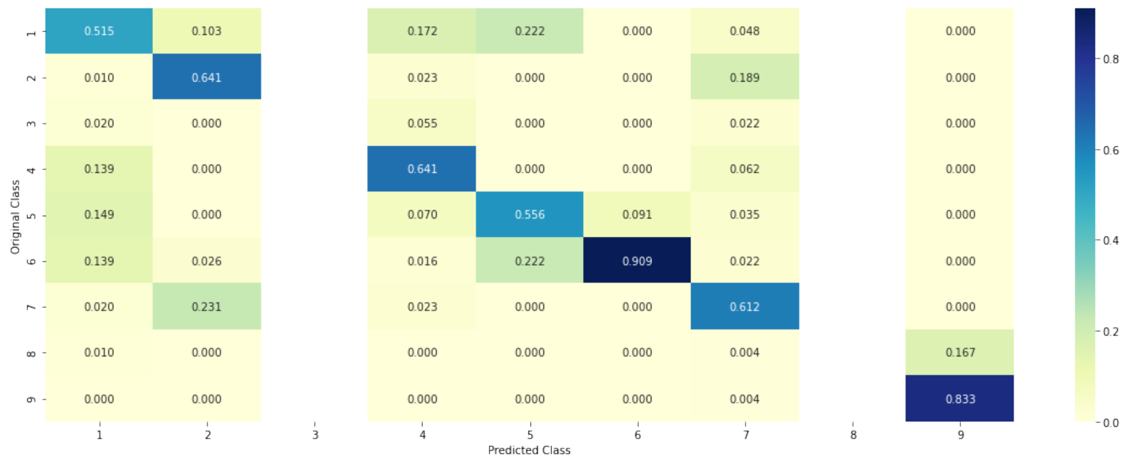
```
clf = RandomForestClassifier(n_estimators=alpha[int(best_alpha/2)],␣
↪criterion='gini', max_depth=max_depth[int(best_alpha%2)], random_state=42,␣
↪n_jobs=-1)
predict_and_plot_confusion_matrix(train_x_onehotCoding,␣
↪train_y,cv_x_onehotCoding,cv_y, clf)
```

```
Log loss : 1.1513661957058137
Number of mis-classified points : 0.38345864661654133
-------------------- Confusion matrix --------------------
```



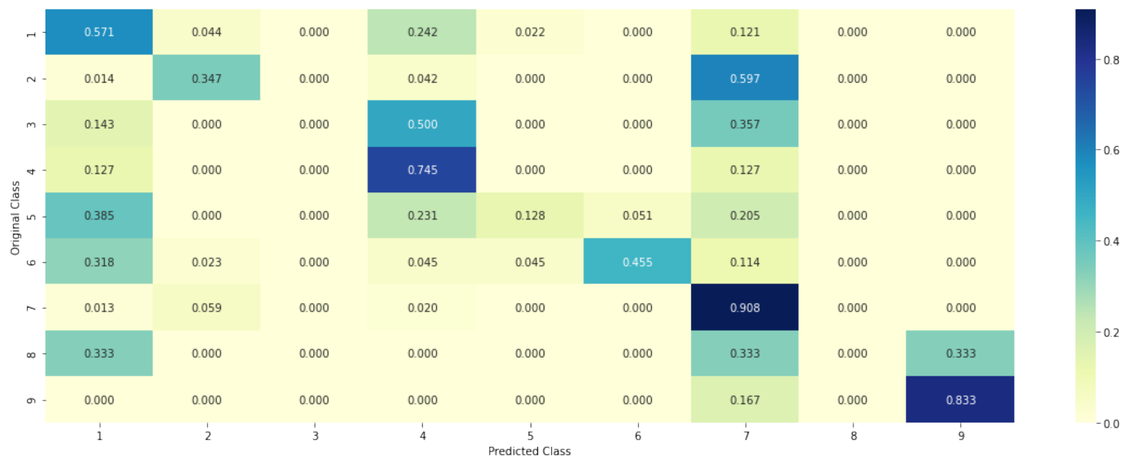
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 52.000 | 4.000 | 0.000 | 22.000 | 2.000 | 0.000 | 11.000 | 0.000 | 0.000 |
| 2 | 1.000 | 25.000 | 0.000 | 3.000 | 0.000 | 0.000 | 43.000 | 0.000 | 0.000 |
| 3 | 2.000 | 0.000 | 0.000 | 7.000 | 0.000 | 0.000 | 5.000 | 0.000 | 0.000 |
| 4 | 14.000 | 0.000 | 0.000 | 82.000 | 0.000 | 0.000 | 14.000 | 0.000 | 0.000 |
| 5 | 15.000 | 0.000 | 0.000 | 9.000 | 5.000 | 2.000 | 8.000 | 0.000 | 0.000 |
| 6 | 14.000 | 1.000 | 0.000 | 2.000 | 2.000 | 20.000 | 5.000 | 0.000 | 0.000 |
| 7 | 2.000 | 9.000 | 0.000 | 3.000 | 0.000 | 0.000 | 139.000 | 0.000 | 0.000 |
| 8 | 1.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 1.000 | 0.000 | 1.000 |
| 9 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 1.000 | 0.000 | 5.000 |

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



|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0.515 | 0.103 |  | 0.172 | 0.222 | 0.000 | 0.048 |  | 0.000 |
| 2 | 0.010 | 0.641 |  | 0.023 | 0.000 | 0.000 | 0.189 |  | 0.000 |
| 3 | 0.020 | 0.000 |  | 0.055 | 0.000 | 0.000 | 0.022 |  | 0.000 |
| 4 | 0.139 | 0.000 |  | 0.641 | 0.000 | 0.000 | 0.062 |  | 0.000 |
| 5 | 0.149 | 0.000 |  | 0.070 | 0.556 | 0.091 | 0.035 |  | 0.000 |
| 6 | 0.139 | 0.026 |  | 0.016 | 0.222 | 0.909 | 0.022 |  | 0.000 |
| 7 | 0.020 | 0.231 |  | 0.023 | 0.000 | 0.000 | 0.612 |  | 0.000 |
| 8 | 0.010 | 0.000 |  | 0.000 | 0.000 | 0.000 | 0.004 |  | 0.167 |
| 9 | 0.000 | 0.000 |  | 0.000 | 0.000 | 0.000 | 0.004 |  | 0.833 |

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



|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0.571 | 0.044 | 0.000 | 0.242 | 0.022 | 0.000 | 0.121 | 0.000 | 0.000 |
| 2 | 0.014 | 0.347 | 0.000 | 0.042 | 0.000 | 0.000 | 0.597 | 0.000 | 0.000 |
| 3 | 0.143 | 0.000 | 0.000 | 0.500 | 0.000 | 0.000 | 0.357 | 0.000 | 0.000 |
| 4 | 0.127 | 0.000 | 0.000 | 0.745 | 0.000 | 0.000 | 0.127 | 0.000 | 0.000 |
| 5 | 0.385 | 0.000 | 0.000 | 0.231 | 0.128 | 0.051 | 0.205 | 0.000 | 0.000 |
| 6 | 0.318 | 0.023 | 0.000 | 0.045 | 0.045 | 0.455 | 0.114 | 0.000 | 0.000 |
| 7 | 0.013 | 0.059 | 0.000 | 0.020 | 0.000 | 0.000 | 0.908 | 0.000 | 0.000 |
| 8 | 0.333 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.333 | 0.000 | 0.333 |
| 9 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.167 | 0.000 | 0.833 |

[107]:

```
# 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 : 7
Predicted Class Probabilities: [[0.1098 0.139  0.0235 0.1534 0.0586 0.0497
0.4515 0.0071 0.0075]]
Actual Class : 7
--------------------------------------------------
0 Text feature [kinase] present in test data point [True]
2 Text feature [phosphorylation] present in test data point [True]
3 Text feature [tyrosine] present in test data point [True]
4 Text feature [activated] present in test data point [True]
```

```
6 Text feature [activation] present in test data point [True]
10 Text feature [constitutive] present in test data point [True]
11 Text feature [function] present in test data point [True]
17 Text feature [cells] present in test data point [True]
19 Text feature [oncogenic] present in test data point [True]
21 Text feature [constitutively] present in test data point [True]
23 Text feature [receptor] present in test data point [True]
35 Text feature [ba] present in test data point [True]
42 Text feature [cell] present in test data point [True]
45 Text feature [proliferation] present in test data point [True]
46 Text feature [f3] present in test data point [True]
53 Text feature [oncogene] present in test data point [True]
62 Text feature [expression] present in test data point [True]
65 Text feature [lines] present in test data point [True]
67 Text feature [extracellular] present in test data point [True]
76 Text feature [transformation] present in test data point [True]
86 Text feature [proteins] present in test data point [True]
95 Text feature [il] present in test data point [True]
96 Text feature [dna] present in test data point [True]
98 Text feature [presence] present in test data point [True]
99 Text feature [mutant] present in test data point [True]
Out of the top  100  features  25 are present in query point
```

## 4.1 RF with Response Coding

[108]:
```
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))


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)
```

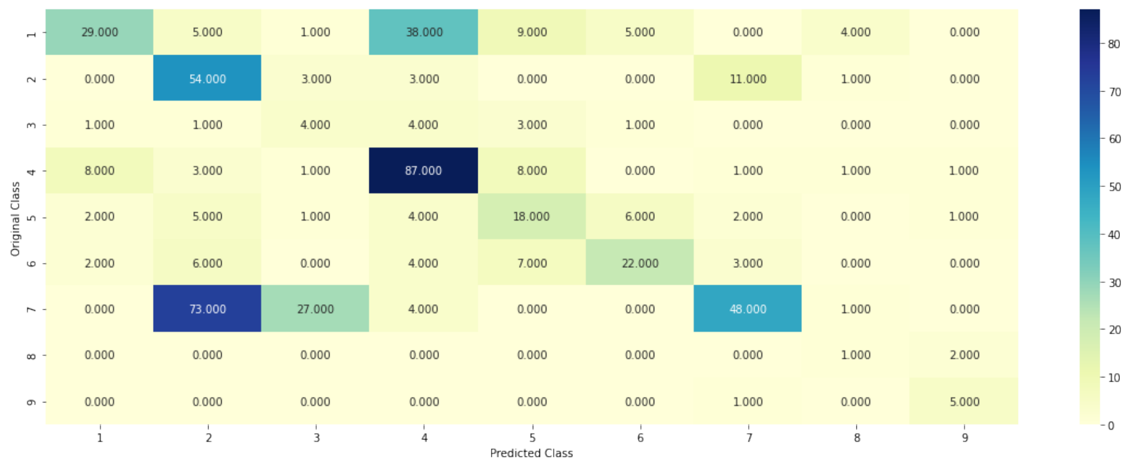
```
sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
sig_clf.fit(train_x_responseCoding, train_y)

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 : 2.019011558757825
for n_estimators = 10 and max depth =  3
Log Loss : 1.6836830088809915
for n_estimators = 10 and max depth =  5
Log Loss : 1.4709967015832548
for n_estimators = 10 and max depth =  10
Log Loss : 1.9066898602863065
for n_estimators = 50 and max depth =  2
Log Loss : 1.5898458702429565
for n_estimators = 50 and max depth =  3
Log Loss : 1.3833003257090064
for n_estimators = 50 and max depth =  5
Log Loss : 1.2833788362382992
for n_estimators = 50 and max depth =  10
Log Loss : 1.7657298571315907
for n_estimators = 100 and max depth =  2
Log Loss : 1.4601838920233587
for n_estimators = 100 and max depth =  3
Log Loss : 1.40911038483864
for n_estimators = 100 and max depth =  5
Log Loss : 1.3176183811501851
for n_estimators = 100 and max depth =  10
Log Loss : 1.6555218634868276
for n_estimators = 200 and max depth =  2
Log Loss : 1.4860554893539555
for n_estimators = 200 and max depth =  3
Log Loss : 1.3873613261741593
for n_estimators = 200 and max depth =  5
Log Loss : 1.348287034752093
for n_estimators = 200 and max depth =  10
Log Loss : 1.681743803335293
for n_estimators = 500 and max depth =  2
```

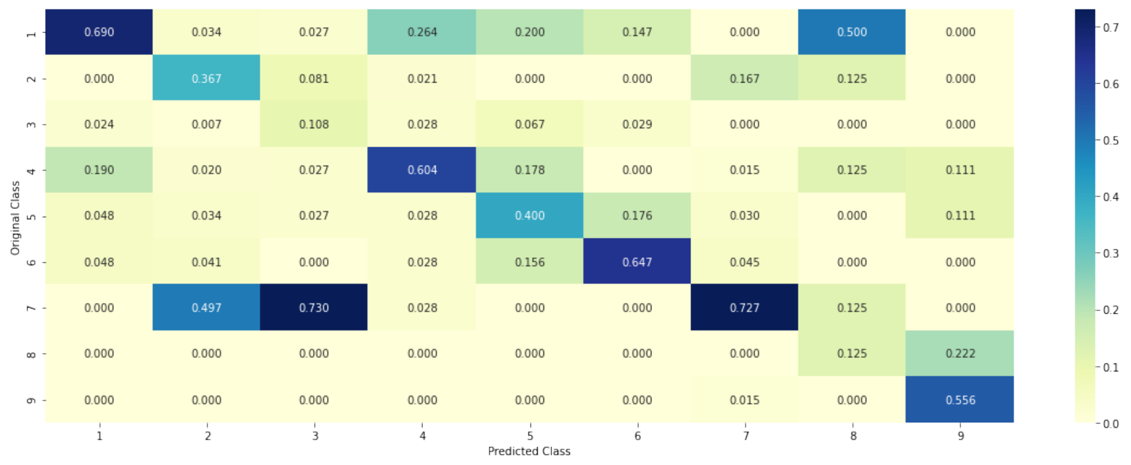
```
Log Loss : 1.5679719930684377
for n_estimators = 500 and max depth =  3
Log Loss : 1.457283610902369
for n_estimators = 500 and max depth =  5
Log Loss : 1.3692398051935435
for n_estimators = 500 and max depth =  10
Log Loss : 1.7320880746674154
for n_estimators = 1000 and max depth =  2
Log Loss : 1.5440446124172909
for n_estimators = 1000 and max depth =  3
Log Loss : 1.4515796481751302
for n_estimators = 1000 and max depth =  5
Log Loss : 1.3645504800040156
for n_estimators = 1000 and max depth =  10
Log Loss : 1.7175778617068542
For values of best alpha =  50 The train log loss is: 0.06593142888575093
For values of best alpha =  50 The cross validation log loss is:
1.2833788362382992
For values of best alpha =  50 The test log loss is: 1.38521174031375
```

[109]:
```
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)
```

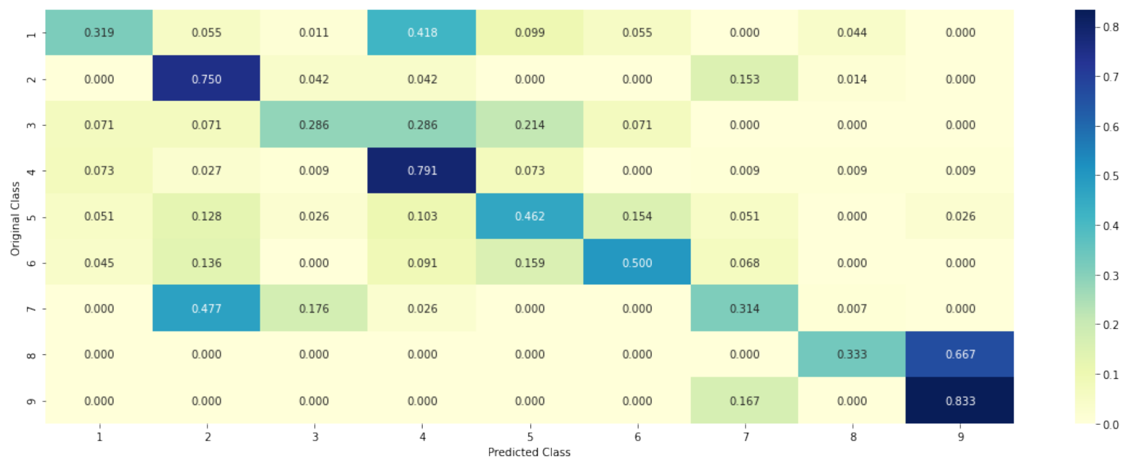
```
Log loss : 1.2833788362382996
Number of mis-classified points : 0.49624060150375937
-------------------- Confusion matrix --------------------
```



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



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



[110]:
```
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 : 2
Predicted Class Probabilities: [[0.0133 0.455  0.1365 0.0202 0.028  0.0253
0.2711 0.0426 0.008 ]]
Actual Class : 7
--------------------------------------------------
Variation is important feature
Variation is important feature
Variation is important feature
Gene is important feature
Variation is important feature
Variation is important feature
Variation is important feature
Text is important feature
Text is important feature
Text is important feature
Text is important feature
Text is important feature
Gene is important feature
Variation 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
Gene is important feature
Variation is important feature
Text is important feature
Text is important feature
Gene is important feature
Gene is important feature
Gene is important feature
```

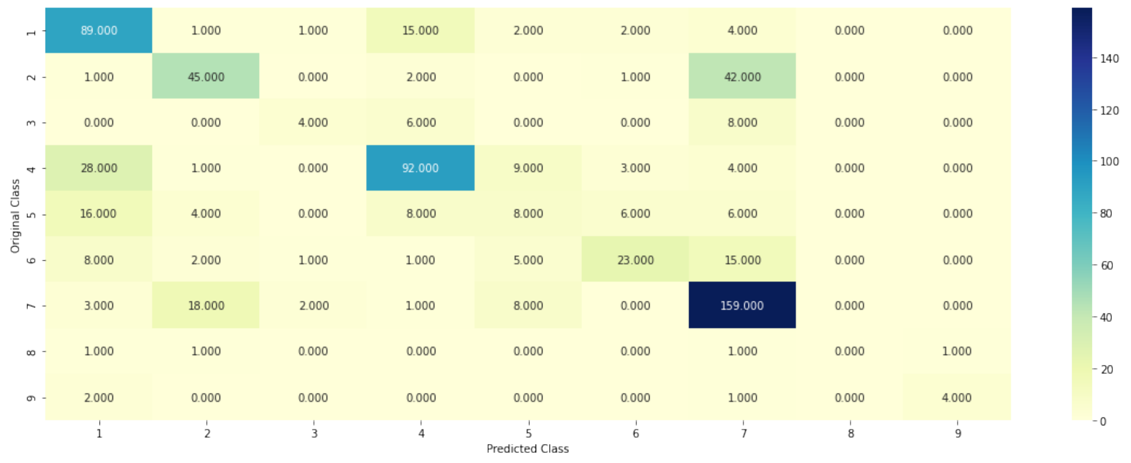
## 5 Stacking model

```
[111]: clf1 = SGDClassifier(alpha=0.001, penalty='l2', loss='log',␣
        ↪class_weight='balanced', random_state=0)
       clf1.fit(train_x_onehotCoding, train_y)
       sig_clf1 = CalibratedClassifierCV(clf1, method="sigmoid")

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


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

       sig_clf1.fit(train_x_onehotCoding, train_y)
       print("Logistic Regression :  Log Loss: %0.2f" % (log_loss(cv_y, sig_clf1.
        ↪predict_proba(cv_x_onehotCoding))))
       sig_clf2.fit(train_x_onehotCoding, train_y)
       print("Support vector machines : Log Loss: %0.2f" % (log_loss(cv_y, sig_clf2.
        ↪predict_proba(cv_x_onehotCoding))))
       sig_clf3.fit(train_x_onehotCoding, train_y)
       print("Naive Bayes : Log Loss: %0.2f" % (log_loss(cv_y, sig_clf3.
        ↪predict_proba(cv_x_onehotCoding))))
       print("-"*50)
       alpha = [0.0001,0.001,0.01,0.1,1,10]
       best_alpha = 999
       for i in alpha:
           lr = LogisticRegression(C=i)
           sclf = StackingClassifier(classifiers=[sig_clf1, sig_clf2, sig_clf3],␣
        ↪meta_classifier=lr, use_probas=True)
           sclf.fit(train_x_onehotCoding, train_y)
           print("Stacking Classifer : for the value of alpha: %f Log Loss: %0.3f" %␣
        ↪(i, log_loss(cv_y, sclf.predict_proba(cv_x_onehotCoding))))
           log_error =log_loss(cv_y, sclf.predict_proba(cv_x_onehotCoding))
           if best_alpha > log_error:
               best_alpha = log_error
```

```
Logistic Regression :  Log Loss: 1.08
Support vector machines : Log Loss: 1.71
Naive Bayes : Log Loss: 1.27
--------------------------------------------------
Stacking Classifer : for the value of alpha: 0.000100 Log Loss: 1.818
Stacking Classifer : for the value of alpha: 0.001000 Log Loss: 1.719
Stacking Classifer : for the value of alpha: 0.010000 Log Loss: 1.306
Stacking Classifer : for the value of alpha: 0.100000 Log Loss: 1.116
Stacking Classifer : for the value of alpha: 1.000000 Log Loss: 1.316
Stacking Classifer : for the value of alpha: 10.000000 Log Loss: 1.579
```

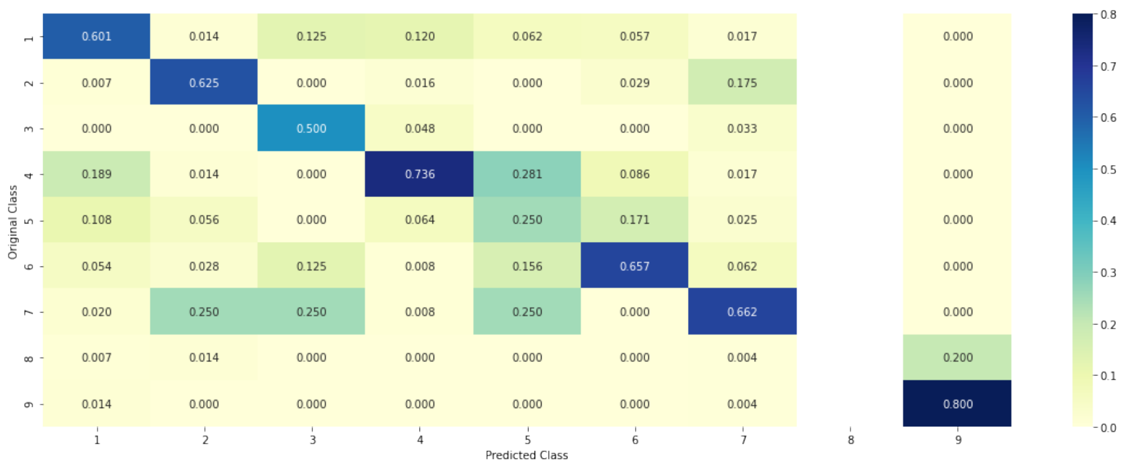
[112]:
```
lr = LogisticRegression(C=0.1)
sclf = StackingClassifier(classifiers=[sig_clf1, sig_clf2, sig_clf3],␣
 ↪meta_classifier=lr, use_probas=True)
sclf.fit(train_x_onehotCoding, train_y)

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

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

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

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

```
Log loss (train) on the stacking classifier : 0.501541165064261
Log loss (CV) on the stacking classifier : 1.1160944198293306
Log loss (test) on the stacking classifier : 1.1955539699907047
Number of missclassified point : 0.362406015037594
-------------------- Confusion matrix --------------------
```

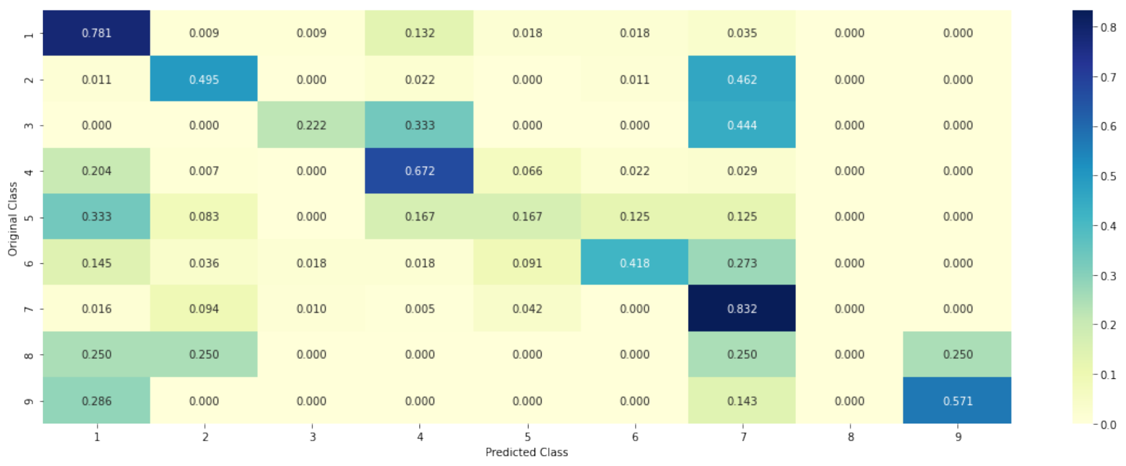


|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 89.000 | 1.000 | 1.000 | 15.000 | 2.000 | 2.000 | 4.000 | 0.000 | 0.000 |
| 2 | 1.000 | 45.000 | 0.000 | 2.000 | 0.000 | 1.000 | 42.000 | 0.000 | 0.000 |
| 3 | 0.000 | 0.000 | 4.000 | 6.000 | 0.000 | 0.000 | 8.000 | 0.000 | 0.000 |
| 4 | 28.000 | 1.000 | 0.000 | 92.000 | 9.000 | 3.000 | 4.000 | 0.000 | 0.000 |
| 5 | 16.000 | 4.000 | 0.000 | 8.000 | 8.000 | 6.000 | 6.000 | 0.000 | 0.000 |
| 6 | 8.000 | 2.000 | 1.000 | 1.000 | 5.000 | 23.000 | 15.000 | 0.000 | 0.000 |
| 7 | 3.000 | 18.000 | 2.000 | 1.000 | 8.000 | 0.000 | 159.000 | 0.000 | 0.000 |
| 8 | 1.000 | 1.000 | 0.000 | 0.000 | 0.000 | 0.000 | 1.000 | 0.000 | 1.000 |
| 9 | 2.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 1.000 | 0.000 | 4.000 |

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



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

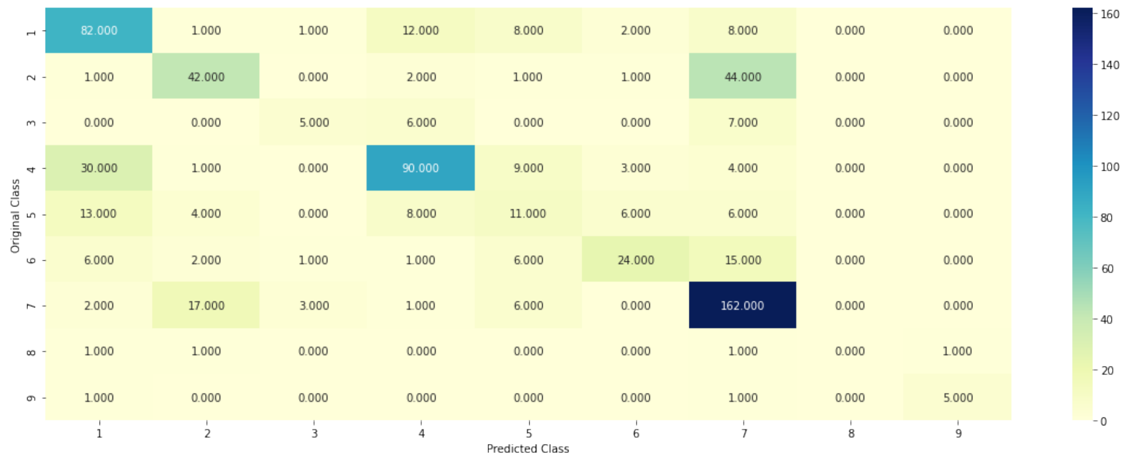


## 6 Maximum voting Classifier

[113]:
```
#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)))
```

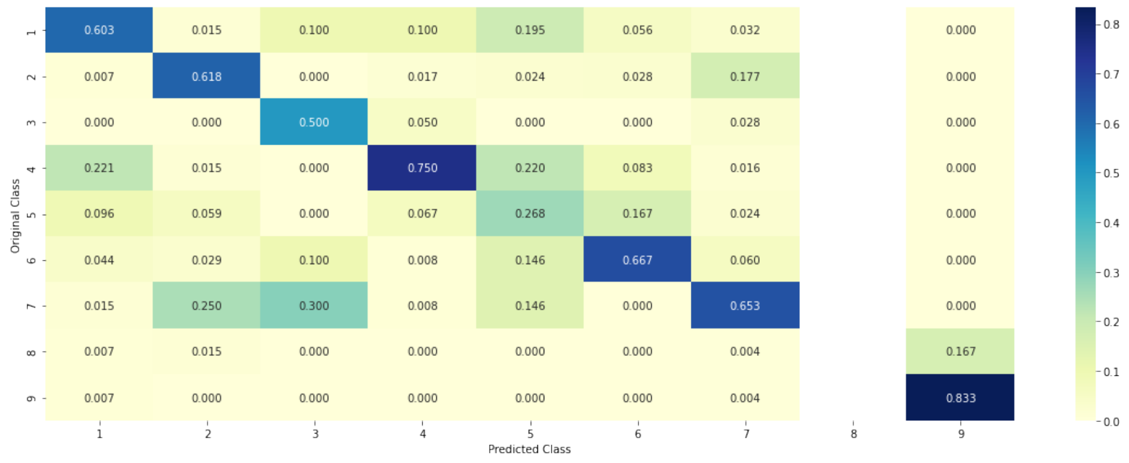
```
print("Log loss (test) on the VotingClassifier :", log_loss(test_y, vclf.
 ↪predict_proba(test_x_onehotCoding)))
print("Number of missclassified point :", np.count_nonzero((vclf.
 ↪predict(test_x_onehotCoding)- test_y))/test_y.shape[0])
plot_confusion_matrix(test_y=test_y, predict_y=vclf.
 ↪predict(test_x_onehotCoding))
```

```
Log loss (train) on the VotingClassifier : 0.8587983986980064
Log loss (CV) on the VotingClassifier : 1.1723541208907562
Log loss (test) on the VotingClassifier : 1.2405947426215598
Number of missclassified point : 0.3669172932330827
-------------------- Confusion matrix --------------------
```



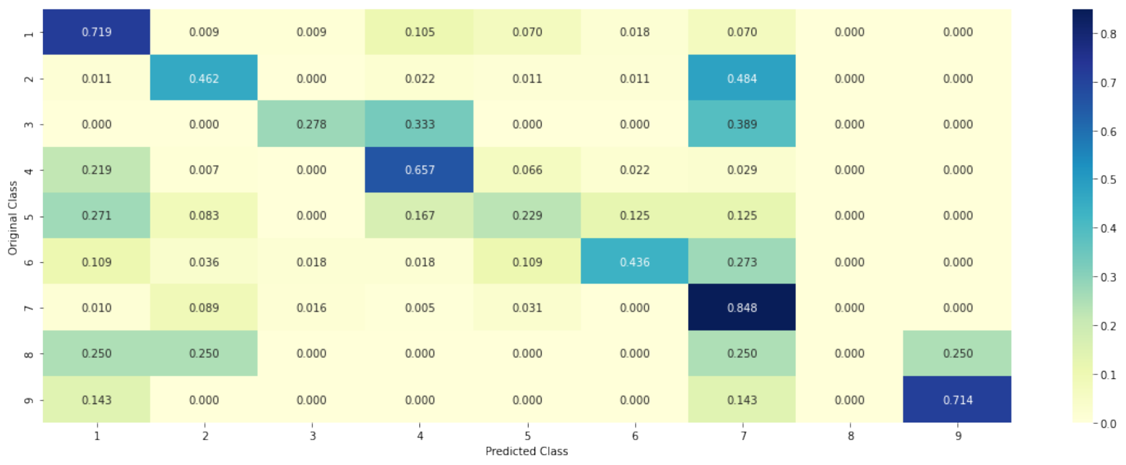
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 82.000 | 1.000 | 1.000 | 12.000 | 8.000 | 2.000 | 8.000 | 0.000 | 0.000 |
| 2 | 1.000 | 42.000 | 0.000 | 2.000 | 1.000 | 1.000 | 44.000 | 0.000 | 0.000 |
| 3 | 0.000 | 0.000 | 5.000 | 6.000 | 0.000 | 0.000 | 7.000 | 0.000 | 0.000 |
| 4 | 30.000 | 1.000 | 0.000 | 90.000 | 9.000 | 3.000 | 4.000 | 0.000 | 0.000 |
| 5 | 13.000 | 4.000 | 0.000 | 8.000 | 11.000 | 6.000 | 6.000 | 0.000 | 0.000 |
| 6 | 6.000 | 2.000 | 1.000 | 1.000 | 6.000 | 24.000 | 15.000 | 0.000 | 0.000 |
| 7 | 2.000 | 17.000 | 3.000 | 1.000 | 6.000 | 0.000 | 162.000 | 0.000 | 0.000 |
| 8 | 1.000 | 1.000 | 0.000 | 0.000 | 0.000 | 0.000 | 1.000 | 0.000 | 1.000 |
| 9 | 1.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 1.000 | 0.000 | 5.000 |

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



| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0.603 | 0.015 | 0.100 | 0.100 | 0.195 | 0.056 | 0.032 | | 0.000 |
| 2 | 0.007 | 0.618 | 0.000 | 0.017 | 0.024 | 0.028 | 0.177 | | 0.000 |
| 3 | 0.000 | 0.000 | 0.500 | 0.050 | 0.000 | 0.000 | 0.028 | | 0.000 |
| 4 | 0.221 | 0.015 | 0.000 | 0.750 | 0.220 | 0.083 | 0.016 | | 0.000 |
| 5 | 0.096 | 0.059 | 0.000 | 0.067 | 0.268 | 0.167 | 0.024 | | 0.000 |
| 6 | 0.044 | 0.029 | 0.100 | 0.008 | 0.146 | 0.667 | 0.060 | | 0.000 |
| 7 | 0.015 | 0.250 | 0.300 | 0.008 | 0.146 | 0.000 | 0.653 | | 0.000 |
| 8 | 0.007 | 0.015 | 0.000 | 0.000 | 0.000 | 0.000 | 0.004 | | 0.167 |
| 9 | 0.007 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.004 | | 0.833 |

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



| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0.719 | 0.009 | 0.009 | 0.105 | 0.070 | 0.018 | 0.070 | 0.000 | 0.000 |
| 2 | 0.011 | 0.462 | 0.000 | 0.022 | 0.011 | 0.011 | 0.484 | 0.000 | 0.000 |
| 3 | 0.000 | 0.000 | 0.278 | 0.333 | 0.000 | 0.000 | 0.389 | 0.000 | 0.000 |
| 4 | 0.219 | 0.007 | 0.000 | 0.657 | 0.066 | 0.022 | 0.029 | 0.000 | 0.000 |
| 5 | 0.271 | 0.083 | 0.000 | 0.167 | 0.229 | 0.125 | 0.125 | 0.000 | 0.000 |
| 6 | 0.109 | 0.036 | 0.018 | 0.018 | 0.109 | 0.436 | 0.273 | 0.000 | 0.000 |
| 7 | 0.010 | 0.089 | 0.016 | 0.005 | 0.031 | 0.000 | 0.848 | 0.000 | 0.000 |
| 8 | 0.250 | 0.250 | 0.000 | 0.000 | 0.000 | 0.000 | 0.250 | 0.000 | 0.250 |
| 9 | 0.143 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.143 | 0.000 | 0.714 |

```
[ ]:
```