PersonalizedCancerDiagnosis

June 6, 2021

Personalized cancer diagnosis

1. Business Problem

1.1. Description

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

Data: Memorial Sloan Kettering Cancer Center (MSKCC)

Download training_variants.zip and training_text.zip from Kaggle.

Context:

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

Problem statement :

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

1.2. Source/Useful Links

Some articles and reference blogs about the problem statement

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

1.3. Real-world/Business objectives and constraints.

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

2. Machine Learning Problem Formulation

2.1. Data

2.1.1. Data Overview

- Source: https://www.kaggle.com/c/msk-redefining-cancer-treatment/data
- We have two data files: one conatins the information about the genetic mutations and the other contains the clinical evidence (text) that human experts/pathologists use to classify the genetic mutations.

- Both these data files are have a common column called ID
- Data file's information:

  training_variants (ID , Gene, Variations, Class)

  training_text (ID, Text)

[ ]:
```
# loading datafiles
!wget --header 'Host: storage.googleapis.com' --user-agent 'Mozilla/5.0␣
↪(Windows NT 10.0; Win64; x64; rv:89.0) Gecko/20100101 Firefox/89.0' --header␣
↪'Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/
↪*;q=0.8' --header 'Accept-Language: en-US,en;q=0.5' --referer 'https://www.
↪kaggle.com/' --header 'Upgrade-Insecure-Requests: 1' 'https://storage.
↪googleapis.com/kagglesdsdata/competitions/6841/44307/training_variants.zip?
↪GoogleAccessId=web-data@kaggle-161607.iam.gserviceaccount.
↪com&Expires=1623175132&Signature=iALYRMW1ADTxIw4Pbes0Ap90Ls9fyWfmYYTg4MmLDDhsIxcoesoJG02B1M
↪zip' --output-document 'training_variants.zip'

!wget --header 'Host: storage.googleapis.com' --user-agent 'Mozilla/5.0␣
↪(Windows NT 10.0; Win64; x64; rv:89.0) Gecko/20100101 Firefox/89.0' --header␣
↪'Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/
↪*;q=0.8' --header 'Accept-Language: en-US,en;q=0.5' --referer 'https://www.
↪kaggle.com/' --header 'Upgrade-Insecure-Requests: 1' 'https://storage.
↪googleapis.com/kagglesdsdata/competitions/6841/44307/training_text.zip?
↪GoogleAccessId=web-data@kaggle-161607.iam.gserviceaccount.
↪com&Expires=1623175127&Signature=tF4Zpt7zAOatfqho1UlsyPUo7L5BRHgGR1ZRO2Frr%2FSp%2B7b9R1YVhk
↪zip' --output-document 'training_text.zip'
```

```
--2021-06-06 14:07:42--  https://storage.googleapis.com/kagglesdsdata/competitio
ns/6841/44307/training_variants.zip?GoogleAccessId=web-data@kaggle-161607.iam.gs
erviceaccount.com&Expires=1623175132&Signature=iALYRMW1ADTxIw4Pbes0Ap90Ls9fyWfmY
YTg4MmLDDhsIxcoesoJG02B1MYOHHJTF7MroWXpCzS3hOLEfZkILvNbfmU0Bn0U8IByXlk9ChuSYiN9q
cJbFkiI4IyXRAl%2BVKK0tZ1j1chyWZcNS%2B5BIt12PWwcCavFmLsfzI0%2FpicRmVkuKrSM2IAYj0J
E4DCLo15lg3ZkF%2Bjrue%2FVc%2BA75wVA0ARlGi8vM5XD0xkEMe%2BDKknTt%2FkL9HzRUV8B7gO2H
n%2FB0%2BOKQOTfe4N3AFmyYRkhU8C3bd1s6KMOM%2Flhd08hI%2FAu8pLIfuxDnEPK6Zwvod%2BwECr
brxeb97dhGxC%2BFw%3D%3D&response-content-
disposition=attachment%3B+filename%3Dtraining_variants.zip
Resolving storage.googleapis.com (storage.googleapis.com)… 142.251.33.208,
172.217.15.80, 172.217.9.208, …
Connecting to storage.googleapis.com
(storage.googleapis.com)|142.251.33.208|:443… connected.
HTTP request sent, awaiting response… 200 OK
Length: 24831 (24K) [application/zip]
Saving to: ‘training_variants.zip’

training_variants.z 100%[===================>]  24.25K  --.-KB/s    in 0s

2021-06-06 14:07:42 (66.3 MB/s) - ‘training_variants.zip’ saved [24831/24831]
```

```
--2021-06-06 14:07:42--  https://storage.googleapis.com/kagglesdsdata/competitio
ns/6841/44307/training_text.zip?GoogleAccessId=web-data@kaggle-161607.iam.gservi
ceaccount.com&Expires=1623175127&Signature=tF4Zpt7zAOatfqho1UlsyPUo7L5BRHgGR1ZRO
2Frr%2FSp%2B7b9R1YVhktAy1PR2Reaxgu9iCGA1Zg%2BpL%2F70KhpDiSerOl7if8x79vCujP3hBdRd
hX1xKRI7xHnajWYBHVJkma1QAsQ%2BUXh9%2F8Cdl%2FMs4O7wxJbOwt69HuFhWpSO60czSontZzUmxB
4Vro3niKtsi0vcHE1LZ5jiNoxTCtDScLmPprhvybjhgLhGpYc9j9yZfhiNrTo%2B3qBVMOeC8CJiDqAe
PWm%2FRd1pcp9vVsZQ7WQGGMsnTtMYDCaoXH9DbhlQQ9mHbla8x%2BTFQZx1yezsCJPGIaBPQnpyh6Re
5NZbQ%3D%3D&response-content-
disposition=attachment%3B+filename%3Dtraining_text.zip
Resolving storage.googleapis.com (storage.googleapis.com)… 142.250.73.208,
172.253.62.128, 172.253.122.128, …
Connecting to storage.googleapis.com
(storage.googleapis.com)|142.250.73.208|:443… connected.
HTTP request sent, awaiting response… 200 OK
Length: 63917183 (61M) [application/zip]
Saving to: ‘training_text.zip’

training_text.zip   100%[===================>]  60.96M   223MB/s    in 0.3s

2021-06-06 14:07:43 (223 MB/s) - ‘training_text.zip’ saved [63917183/63917183]
```

[ ]:
```
!unzip '/content/training_variants.zip'
!unzip '/content/training_text.zip'
```

```
Archive:  /content/training_variants.zip
  inflating: training_variants
Archive:  /content/training_text.zip
  inflating: training_text
```

2.1.2. Example Data Point

training_variants

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

training_text

ID,Text 0||Cyclin-dependent kinases (CDKs) regulate a variety of fundamental cellular processes. CDK10 stands out as one of the last orphan CDKs for which no activating cyclin has been identified and no kinase activity revealed. Previous work has shown that CDK10 silencing increases ETS2 (v-ets erythroblastosis virus E26 oncogene homolog 2)-driven activation of the MAPK pathway, which confers tamoxifen resistance to breast cancer cells. The precise mechanisms by which CDK10 modulates ETS2 activity, and more generally the functions of CDK10, remain elusive. Here we demonstrate that CDK10 is a cyclin-dependent kinase by identifying cyclin M as an activating cyclin. Cyclin M, an orphan cyclin, is the product of FAM58A, whose mutations cause STAR syndrome, a human developmental anomaly whose features include toe syndactyly, telecanthus, and anogenital and renal malformations. We show that STAR syndrome-associated cyclin M mutants are unable to interact with CDK10. Cyclin M silencing phenocopies CDK10 silencing in increasing c-Raf and in conferring tamoxifen resistance to breast cancer cells. CDK10/cyclin M phosphorylates

ETS2 in vitro, and in cells it positively controls ETS2 degradation by the proteasome. ETS2 protein levels are increased in cells derived from a STAR patient, and this increase is attributable to decreased cyclin M levels. Altogether, our results reveal an additional regulatory mechanism for ETS2, which plays key roles in cancer and development. They also shed light on the molecular mechanisms underlying STAR syndrome.Cyclin-dependent kinases (CDKs) play a pivotal role in the control of a number of fundamental cellular processes (1). The human genome contains 21 genes encoding proteins that can be considered as members of the CDK family owing to their sequence similarity with bona fide CDKs, those known to be activated by cyclins (2). Although discovered almost 20 y ago (3, 4), CDK10 remains one of the two CDKs without an identified cyclin partner. This knowledge gap has largely impeded the exploration of its biological functions. CDK10 can act as a positive cell cycle regulator in some cells (5, 6) or as a tumor suppressor in others (7, 8). CDK10 interacts with the ETS2 (v-ets erythroblastosis virus E26 oncogene homolog 2) transcription factor and inhibits its transcriptional activity through an unknown mechanism (9). CDK10 knockdown derepresses ETS2, which increases the expression of the c-Raf protein kinase, activates the MAPK pathway, and induces resistance of MCF7 cells to tamoxifen (6). …

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

2.2.1. Type of Machine Learning Problem

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

2.2.2. Performance Metric

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

Metric(s): * Multi class log-loss * Confusion matrix

2.2.3. Machine Learing Objectives and Constraints

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

Constraints:

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

2.3. Train, CV and Test Datasets

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

3. Exploratory Data Analysis

[ ]:
```
import pandas as pd
import matplotlib.pyplot as plt
import re
import time
import warnings
import numpy as np
from nltk.corpus import stopwords
from sklearn.decomposition import TruncatedSVD
```

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

from mlxtend.classifier import StackingClassifier

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

```
/usr/local/lib/python3.7/dist-packages/sklearn/utils/deprecation.py:144:
FutureWarning: The sklearn.metrics.classification module is  deprecated in
version 0.22 and will be removed in version 0.24. The corresponding classes /
functions should instead be imported from sklearn.metrics. Anything that cannot
be imported from sklearn.metrics is now part of the private API.
  warnings.warn(message, FutureWarning)
/usr/local/lib/python3.7/dist-packages/sklearn/externals/six.py:31:
FutureWarning: The module is deprecated in version 0.21 and will be removed in
version 0.23 since we've dropped support for Python 2.7. Please rely on the
official version of six (https://pypi.org/project/six/).
  "(https://pypi.org/project/six/).", FutureWarning)
/usr/local/lib/python3.7/dist-packages/sklearn/utils/deprecation.py:144:
FutureWarning: The sklearn.neighbors.base module is  deprecated in version 0.22
and will be removed in version 0.24. The corresponding classes / functions
should instead be imported from sklearn.neighbors. Anything that cannot be
imported from sklearn.neighbors is now part of the private API.
  warnings.warn(message, FutureWarning)
```

3.1. Reading Data

3.1.1. Reading Gene and Variation Data

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

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

[ ]:

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

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

ID : the id of the row used to link the mutation to the clinical evidence

Gene : the gene where this genetic mutation is located

Variation : the aminoacid change for this mutations

Class : 1-9 the class this genetic mutation has been classified on

3.1.2. Reading Text Data

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

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

[ ]:

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

3.1.3. Preprocessing of text

[ ]:
```
import nltk
nltk.download('stopwords')
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data…
[nltk_data]   Unzipping corpora/stopwords.zip.
```

[ ]:
```
True
```

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


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

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

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

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

```
there is no text description for id: 1109
there is no text description for id: 1277
there is no text description for id: 1407
there is no text description for id: 1639
there is no text description for id: 2755
Time took for preprocessing the text : 31.313768 seconds
```

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

```
[ ]:    ID    Gene  … Class                                                TEXT
     0   0  FAM58A  …     1  cyclin dependent kinases cdks regulate variety…
     1   1     CBL  …     2  abstract background non small cell lung cancer…
     2   2     CBL  …     2  abstract background non small cell lung cancer…
     3   3     CBL  …     3  recent evidence demonstrated acquired uniparen…
     4   4     CBL  …     4  oncogenic mutations monomeric casitas b lineag…

     [5 rows x 5 columns]
```

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

```
[ ]:         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
```

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

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

```
[ ]:         ID   Gene Variation  Class          TEXT
     1109  1109  FANCA    S1088F      1  FANCA S1088F
```

3.1.4. Test, Train and Cross Validation Split

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

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

     # split the data into test and train by maintaining same distribution of output␣
      ↪varaible 'y_true' [stratify=y_true]
     X_train, test_df, y_train, y_test = train_test_split(result, y_true,␣
      ↪stratify=y_true, test_size=0.2, random_state = 32)
     # split the train data into train and cross validation by maintaining same␣
      ↪distribution of output varaible 'y_train' [stratify=y_train]
     train_df, cv_df, y_train, y_cv = train_test_split(X_train, y_train,␣
      ↪stratify=y_train, test_size=0.2, random_state = 32)
```

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

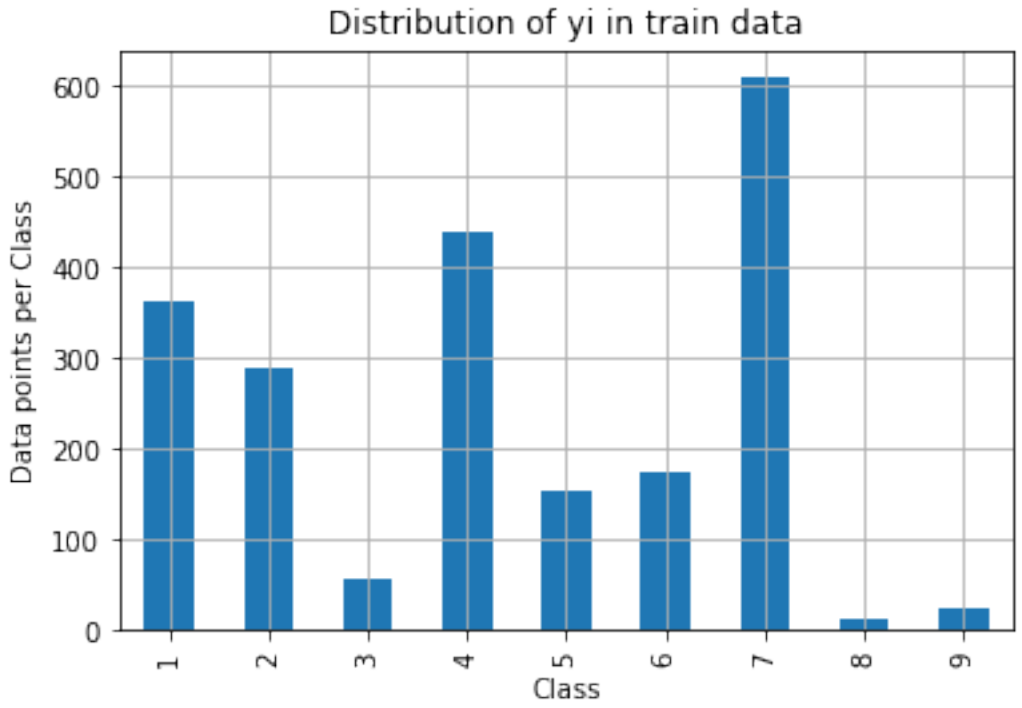
[ ]:
```
print('Number of data points in train data:', train_df.shape[0])
print('Number of data points in test data:', test_df.shape[0])
print('Number of data points in cross validation data:', cv_df.shape[0])
```

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

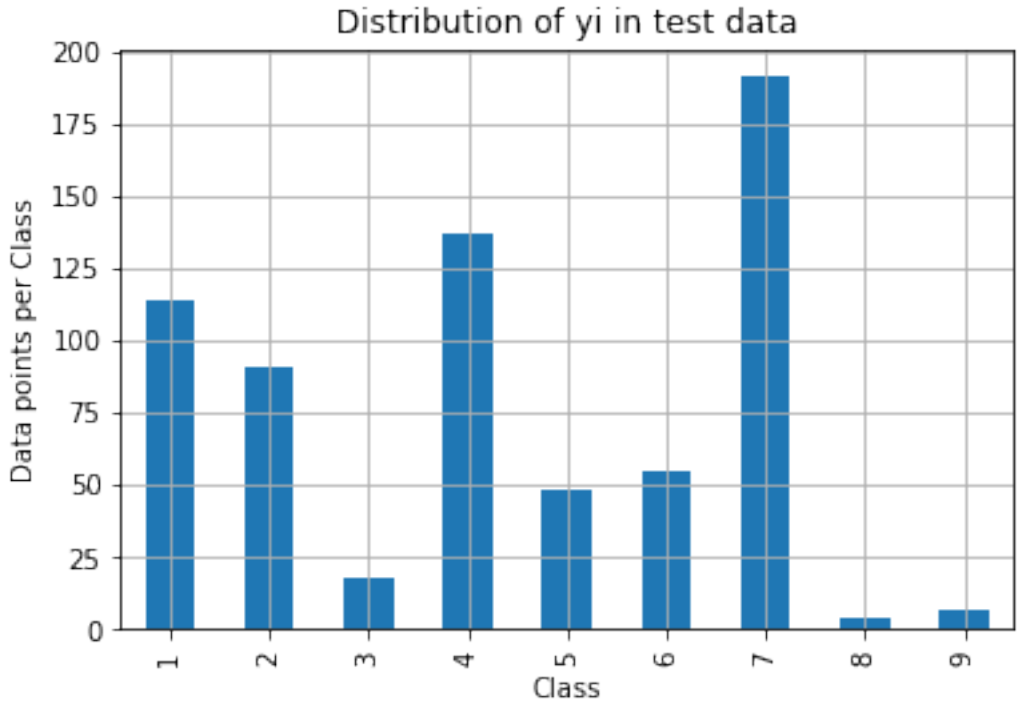
3.1.4.2. Distribution of y_i's in Train, Test and Cross Validation datasets

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

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

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


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

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

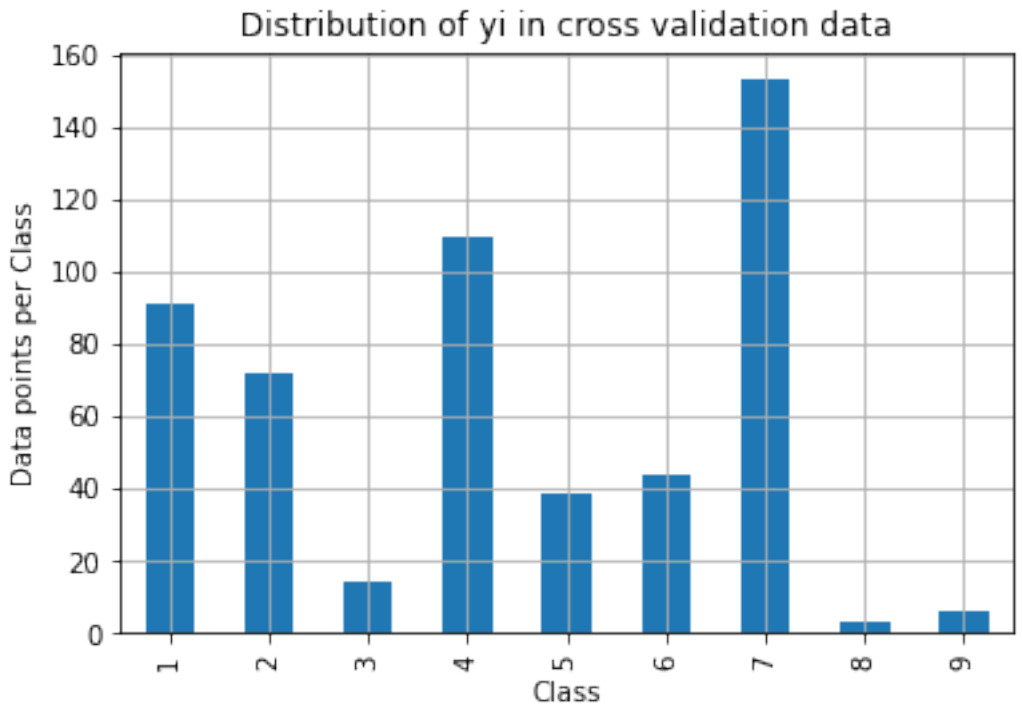
```
sorted_yi = np.argsort(-test_class_distribution.values)
for i in sorted_yi:
    print('Number of data points in class', i+1, ':',test_class_distribution.
 ↪values[i], '(', np.round((test_class_distribution.values[i]/test_df.
 ↪shape[0]*100), 3), '%)')

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

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



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




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




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

3.2 Prediction using a 'Random' Model

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

[ ]:
```
# This function plots the confusion matrices given y_i, y_i_hat.
def plot_confusion_matrix(test_y, predict_y):
    C = confusion_matrix(test_y, predict_y)
    # C = 9,9 matrix, each cell (i,j) represents number of points of class i␣
 ↪are predicted class j

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

    # C = [[1, 2],
```

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

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

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

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

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

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

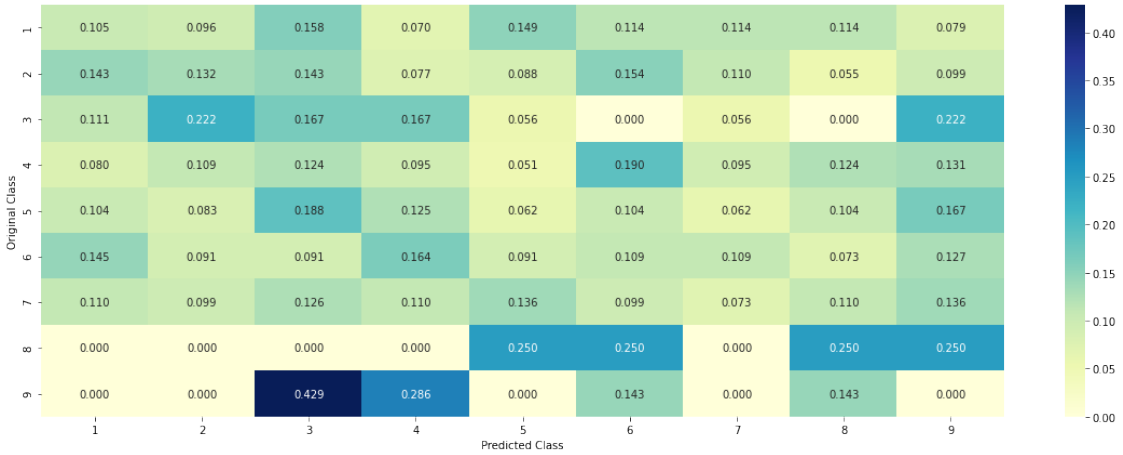
```
    plt.xlabel('Predicted Class')
    plt.ylabel('Original Class')
    plt.show()
```

[ ]:
```
# we need to generate 9 numbers and the sum of numbers should be 1
# one solution is to genarate 9 numbers and divide each of the numbers by their␣
 ↪sum
# ref: https://stackoverflow.com/a/18662466/4084039
test_data_len = test_df.shape[0]
cv_data_len = cv_df.shape[0]

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


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

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

```
Log loss on Cross Validation Data using Random Model 2.529160600306151
Log loss on Test Data using Random Model 2.527183255051364
-------------------- Confusion matrix --------------------
```

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

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



3.3 Univariate Analysis

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

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

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

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

            # cls_cnt.shape[0](numerator) will contain the number of time that␣
↪particular feature occured in whole 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': [0.200757575757575, 0.037878787878788, 0.
↪068181818181818177, 0.13636363636363635, 0.25, 0.1931818181818181, 0.
↪037878787878788, 0.037878787878788, 0.037878787878788],
    #      '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': [0.1333333333333333, 0.060606060606060608, 0.
↪060606060606060608, 0.078787878787878782, 0.139393939393394, 0.
↪3454545454545546, 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.
↪07333333333333334, 0.07333333333333334, 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
```

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

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

3.2.1 Univariate Analysis on Gene Feature

Q1. Gene, What type of feature it is ?

Ans. Gene is a categorical variable

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

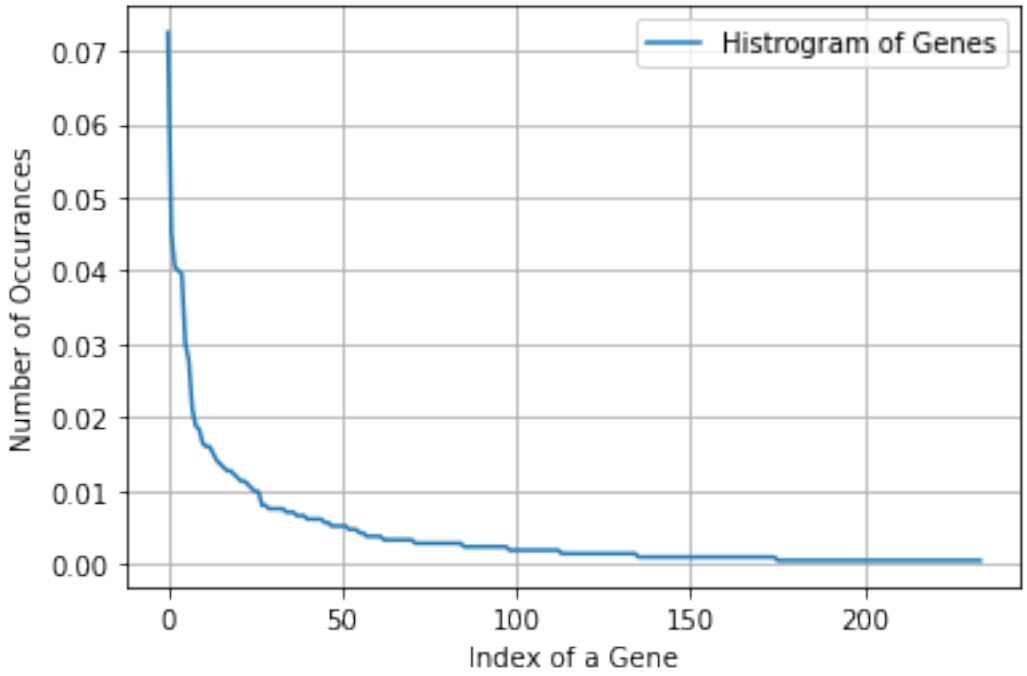
[ ]:
```
unique_genes = train_df['Gene'].value_counts()
print('Number of Unique Genes :', unique_genes.shape[0])
# the top 10 genes that occured most
print(unique_genes.head(10))
```

```
Number of Unique Genes : 234
BRCA1     154
TP53       96
BRCA2      86
EGFR       85
PTEN       84
KIT        64
BRAF       59
ERBB2      45
ALK        40
PDGFRA     39
Name: Gene, dtype: int64
```

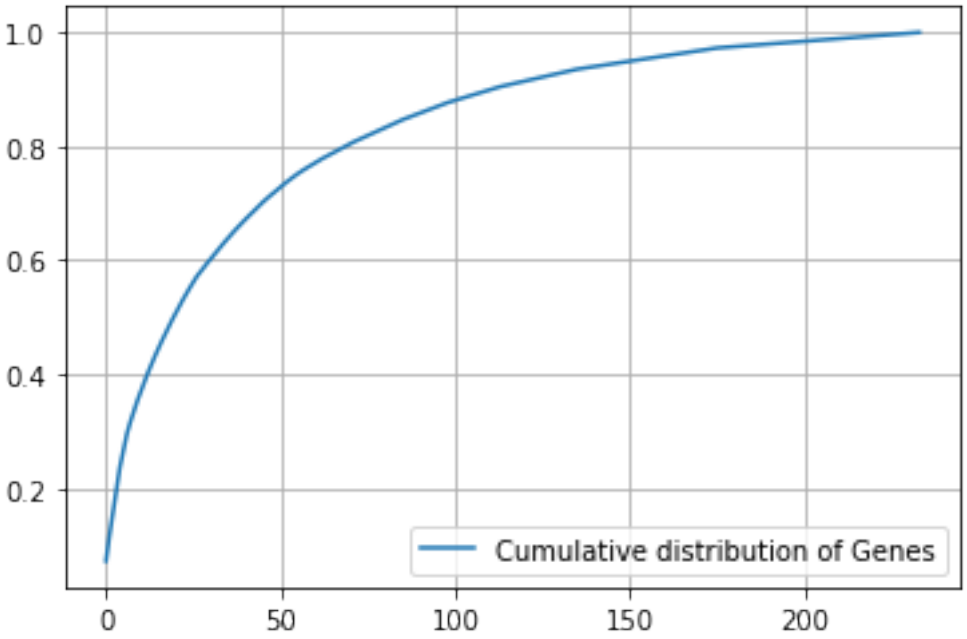
[ ]:
```
print("Ans: There are", unique_genes.shape[0] ,"different categories of genes␣
 ↪in the train data, and they are distibuted as follows",)
```

```
Ans: There are 234 different categories of genes in the train data, and they are
distibuted as follows
```

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



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



Q3. How to featurize this Gene feature ?

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

One hot Encoding

Response coding

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

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

[ ]:
```

```

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

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

[ ]:
```
# 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'])
```

[ ]:
```

```

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

[ ]:
```
1125      MET
403      TP53
3001      KIT
1575     SDHB
1963    MAPK1
Name: Gene, dtype: object
```

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

[ ]:
```
['abl1',
 'acvr1',
 'ago2',
 'akt1',
 'akt2',
 'akt3',
 'alk',
 'apc',
 'ar',
 'araf',
 'arid1b',
 'arid2',
 'arid5b',
 'asxl2',
 'atm',
 'atr',
 'atrx',
 'aurka',
 'aurkb',
 'axin1',
 'axl',
```

```
'b2m',
'bap1',
'bcl10',
'bcl2l11',
'bcor',
'braf',
'brca1',
'brca2',
'brd4',
'brip1',
'btk',
'card11',
'carm1',
'casp8',
'cbl',
'ccnd1',
'ccnd3',
'ccne1',
'cdh1',
'cdk12',
'cdk4',
'cdk6',
'cdk8',
'cdkn1a',
'cdkn1b',
'cdkn2a',
'cdkn2b',
'cdkn2c',
'cebpa',
'chek2',
'cic',
'crebbp',
'ctcf',
'ctla4',
'ctnnb1',
'ddr2',
'dicer1',
'dnmt3a',
'dnmt3b',
'dusp4',
'egfr',
'eif1ax',
'elf3',
'ep300',
'epas1',
'erbb2',
'erbb3',
```

```
'erbb4',
'ercc2',
'ercc4',
'erg',
'errfi1',
'esr1',
'etv1',
'etv6',
'ewsr1',
'ezh2',
'fanca',
'fat1',
'fbxw7',
'fgf4',
'fgfr1',
'fgfr2',
'fgfr3',
'fgfr4',
'flt1',
'flt3',
'foxa1',
'foxl2',
'foxo1',
'foxp1',
'gata3',
'gli1',
'gnas',
'h3f3a',
'hist1h1c',
'hla',
'hnf1a',
'hras',
'idh1',
'idh2',
'igf1r',
'il7r',
'inpp4b',
'jak1',
'jak2',
'kdm5a',
'kdm5c',
'kdm6a',
'kdr',
'keap1',
'kit',
'klf4',
'kmt2a',
```

```
'kmt2b',
'kmt2c',
'kmt2d',
'knstrn',
'kras',
'lats2',
'map2k1',
'map2k2',
'map2k4',
'map3k1',
'mapk1',
'mdm2',
'mdm4',
'med12',
'mef2b',
'men1',
'met',
'mga',
'mlh1',
'mpl',
'msh2',
'msh6',
'mtor',
'myc',
'mycn',
'myd88',
'myod1',
'ncor1',
'nf1',
'nf2',
'nfe2l2',
'nfkbia',
'nkx2',
'notch1',
'notch2',
'npm1',
'nras',
'ntrk1',
'ntrk2',
'ntrk3',
'nup93',
'pak1',
'pbrm1',
'pdgfra',
'pdgfrb',
'pik3ca',
'pik3cb',
```

```
'pik3cd',
'pik3r1',
'pik3r2',
'pik3r3',
'pim1',
'pms1',
'pms2',
'pole',
'ppm1d',
'ppp2r1a',
'ppp6c',
'prdm1',
'ptch1',
'pten',
'ptpn11',
'ptprd',
'ptprt',
'rab35',
'rac1',
'rad50',
'rad51c',
'raf1',
'rara',
'rasa1',
'rb1',
'ret',
'rheb',
'rhoa',
'rictor',
'rit1',
'ros1',
'rras2',
'runx1',
'rxra',
'rybp',
'sdhb',
'sdhc',
'setd2',
'sf3b1',
'shq1',
'smad2',
'smad3',
'smad4',
'smarca4',
'smarcb1',
'smo',
'sos1',
```

```
 'sox9',
 'spop',
 'src',
 'srsf2',
 'stag2',
 'stat3',
 'stk11',
 'tcf3',
 'tcf7l2',
 'tert',
 'tet1',
 'tet2',
 'tgfbr1',
 'tgfbr2',
 'tmprss2',
 'tp53',
 'tp53bp1',
 'tsc1',
 'tsc2',
 'u2af1',
 'vhl',
 'xpo1',
 'xrcc2',
 'yap1']
```

[ ]:
```
print("train_gene_feature_onehotCoding is converted feature using one-hot␣
↪encoding method. The shape of gene feature:",␣
↪train_gene_feature_onehotCoding.shape)
```

```
train_gene_feature_onehotCoding is converted feature using one-hot encoding
method. The shape of gene feature: (2124, 233)
```

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

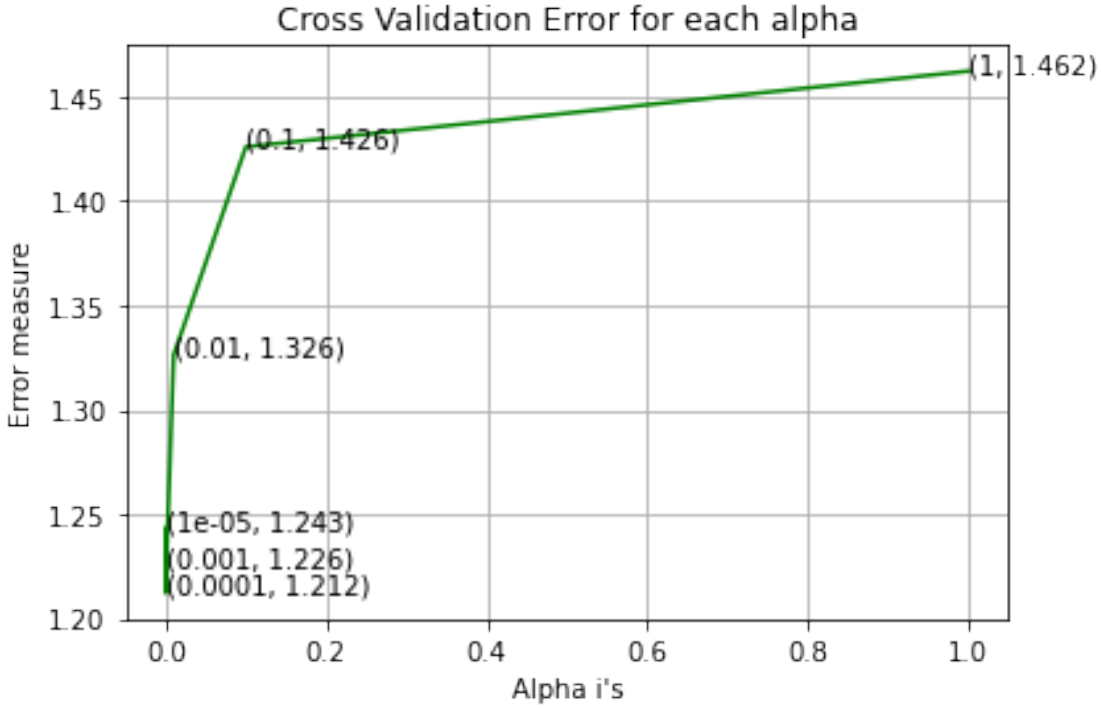
There are many ways to estimate how good a feature is, in predicting y_i. One of the good methods is to build a proper ML model using just this feature. In this case, we will build a logistic regression model using only Gene feature (one hot encoded) to predict y_i.

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

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

```
# class_weight=None, warm_start=False, average=False, n_iter=None)

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

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


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

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


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

predict_y = sig_clf.predict_proba(train_gene_feature_onehotCoding)
print('For values of best alpha = ', alpha[best_alpha], "The train log loss 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 alpha =  1e-05 The log loss is: 1.2433355044070549
For values of alpha =  0.0001 The log loss is: 1.2123049986023322
For values of alpha =  0.001 The log loss is: 1.225820050769069
For values of alpha =  0.01 The log loss is: 1.3261892395661667
For values of alpha =  0.1 The log loss is: 1.4259855533200692
For values of alpha =  1 The log loss is: 1.462235896919596
```




```
For values of best alpha =  0.0001 The train log loss is: 0.9721800805762538
For values of best alpha =  0.0001 The cross validation log loss is:
1.2123049986023322
For values of best alpha =  0.0001 The test log loss is: 1.206394037332043
```

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

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

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

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

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

```
Q6. How many data points in Test and CV datasets are covered by the  234  genes
in train dataset?
Ans
1. In test data 646 out of 665 : 97.14285714285714
2. In cross validation data 516 out of  532 : 96.99248120300751
```

3.2.2 Univariate Analysis on Variation Feature

Q7. Variation, What type of feature is it ?

Ans. Variation is a categorical variable

Q8. How many categories are there?

[ ]:
```
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))
```

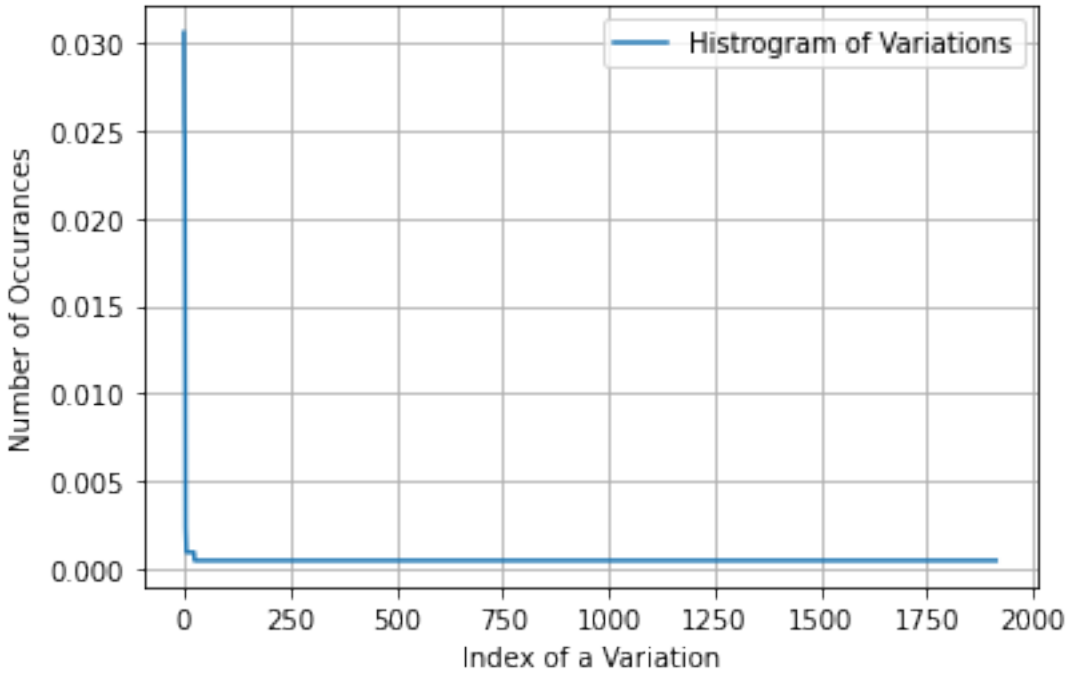
```
Number of Unique Variations : 1913
Truncating_Mutations    65
Amplification           54
Deletion                52
Fusions                 19
Overexpression           5
G12V                     3
Q61K                     2
S222D                    2
P130S                    2
R170W                    2
Name: Variation, dtype: int64
```

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

```
Ans: There are 1913 different categories of variations in the train data, and
they are distibuted as follows
```

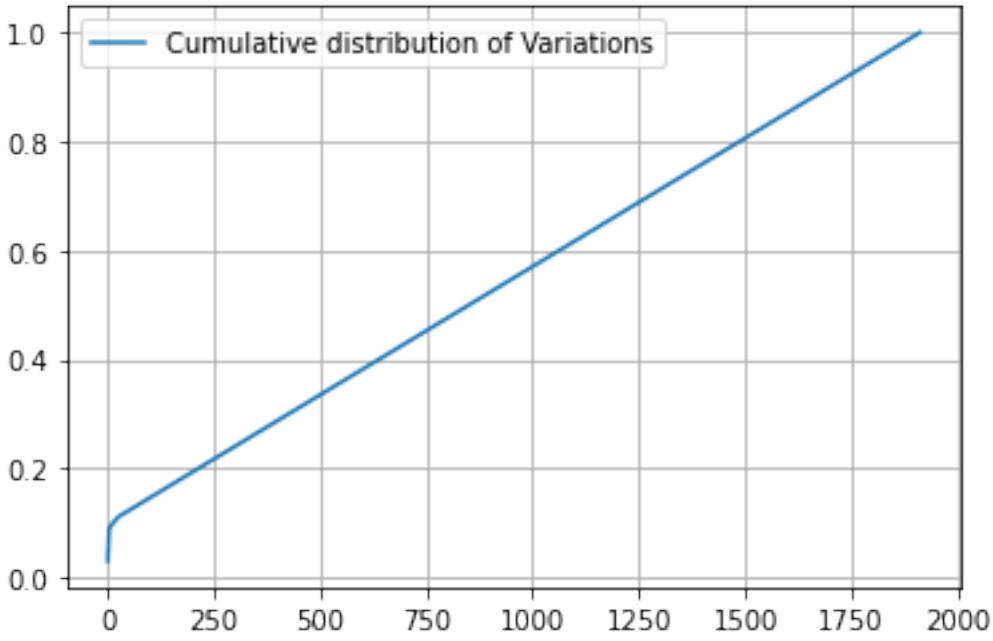
[ ]:
```
s = sum(unique_variations.values);
h = unique_variations.values/s;
plt.plot(h, label="Histrogram of Variations")
```

```
plt.xlabel('Index of a Variation')
plt.ylabel('Number of Occurances')
plt.legend()
plt.grid()
plt.show()
```



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

```
[0.03060264 0.05602637 0.08050847 … 0.99905838 0.99952919 1.        ]
```



Q9. How to featurize this Variation feature ?

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

One hot Encoding

Response coding

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

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

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

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

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

[ ]:
```

```

[ ]:
```
print("train_variation_feature_onehotEncoded is converted feature using the␣
 ↪onne-hot encoding method. The shape of Variation feature:",␣
 ↪train_variation_feature_onehotCoding.shape)
```

```
train_variation_feature_onehotEncoded is converted feature using the onne-hot
encoding method. The shape of Variation feature: (2124, 1947)
```

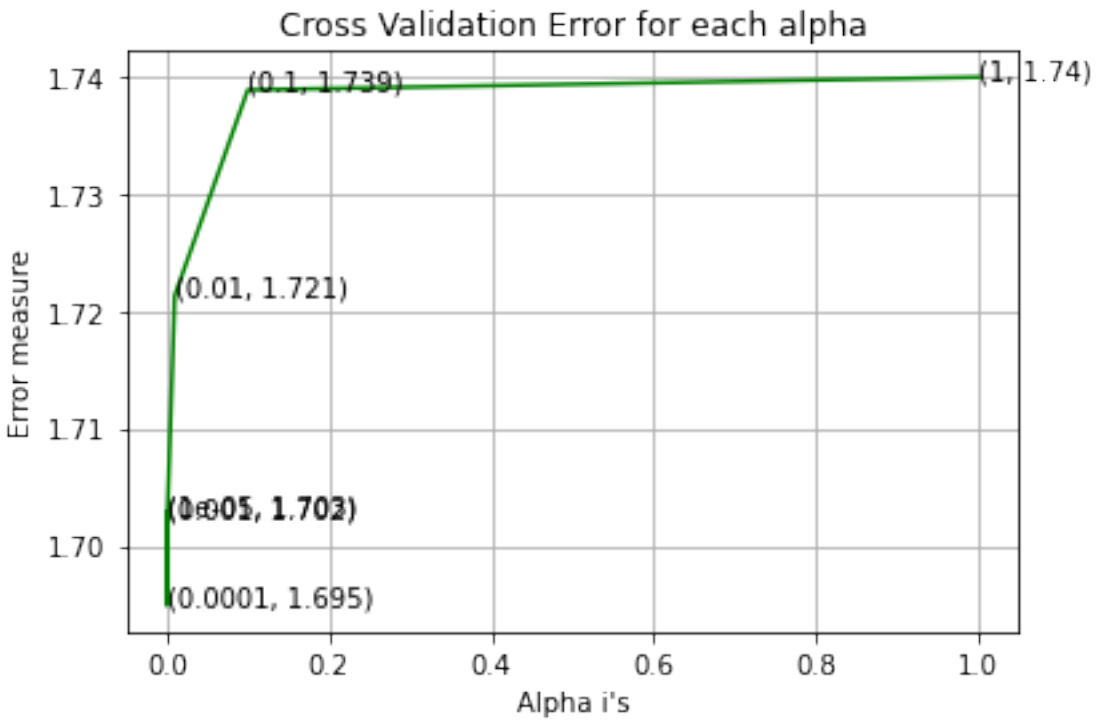
Q10. How good is this Variation feature in predicting y_i?

Let's build a model just like the earlier!

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

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

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

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


cv_log_error_array=[]
for i in alpha:
```

```
    clf = SGDClassifier(alpha=i, penalty='l2', loss='log', random_state=42)
    clf.fit(train_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))

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


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

predict_y = sig_clf.predict_proba(train_variation_feature_onehotCoding)
print('For values of best alpha = ', alpha[best_alpha], "The train log loss is:
 ↪",log_loss(y_train, predict_y, labels=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(cv_variation_feature_onehotCoding)
print('For values of best alpha = ', alpha[best_alpha], "The cross validation␣
 ↪log loss is:",log_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 alpha =  1e-05 The log loss is: 1.702805702164398
For values of alpha =  0.0001 The log loss is: 1.6949264299105384
For values of alpha =  0.001 The log loss is: 1.7024338884352959
For values of alpha =  0.01 The log loss is: 1.7212777172642189
For values of alpha =  0.1 The log loss is: 1.738801470419709
For values of alpha =  1 The log loss is: 1.7398888217754724
```




```
For values of best alpha =  0.0001 The train log loss is: 0.6748424691014371
For values of best alpha =  0.0001 The cross validation log loss is:
1.6949264299105384
For values of best alpha =  0.0001 The test log loss is: 1.7567093319317302
```

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

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

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

```
Q12. How many data points are covered by total  1913  genes in test and cross
validation data sets?
Ans
1. In test data 50 out of 665 : 7.518796992481203
2. In cross validation data 56 out of  532 : 10.526315789473683
```

3.2.3 Univariate Analysis on Text Feature

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

[ ]:
```
# 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
```

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

[ ]:
```
# 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 : 54185
```

[ ]:
```
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)
```

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

[ ]:
```
# 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
```

[ ]:
```
# 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)
```

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

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

```
Counter({3: 5196, 4: 4354, 5: 3139, 6: 2428, 8: 2178, 7: 2050, 9: 1732, 10:
1405, 12: 1234, 11: 1218, 13: 1014, 16: 966, 15: 960, 14: 869, 18: 713, 17: 605,
20: 572, 19: 572, 21: 559, 24: 512, 22: 504, 30: 386, 27: 380, 25: 360, 28: 358,
26: 347, 48: 343, 23: 343, 32: 326, 46: 315, 29: 278, 35: 270, 33: 270, 37: 248,
31: 246, 36: 243, 34: 242, 45: 232, 40: 232, 39: 209, 42: 192, 38: 191, 44: 181,
41: 176, 52: 175, 49: 175, 51: 172, 50: 159, 43: 158, 54: 156, 55: 154, 47: 142,
61: 139, 56: 136, 53: 130, 64: 126, 58: 124, 60: 119, 57: 117, 66: 111, 62: 111,
63: 107, 72: 106, 59: 106, 90: 99, 65: 95, 75: 90, 70: 90, 67: 90, 80: 89, 77:
87, 81: 86, 79: 86, 69: 86, 74: 85, 68: 85, 71: 84, 76: 83, 96: 75, 88: 74, 78:
73, 94: 72, 73: 70, 86: 69, 97: 68, 83: 68, 91: 67, 85: 65, 93: 64, 84: 64, 92:
63, 120: 62, 95: 62, 105: 61, 108: 60, 98: 59, 104: 57, 100: 56, 109: 55, 89:
54, 111: 53, 87: 52, 82: 52, 112: 50, 110: 50, 102: 50, 144: 48, 113: 48, 101:
48, 115: 47, 103: 47, 122: 46, 107: 46, 127: 45, 106: 45, 117: 44, 133: 42, 114:
42, 124: 41, 140: 40, 121: 40, 118: 40, 116: 40, 99: 40, 134: 39, 129: 39, 128:
38, 123: 38, 147: 37, 135: 37, 125: 37, 145: 36, 139: 36, 136: 36, 131: 36, 180:
35, 156: 34, 142: 33, 126: 33, 171: 32, 161: 32, 160: 32, 148: 32, 143: 32, 130:
32, 213: 31, 177: 31, 154: 30, 151: 30, 150: 30, 149: 30, 163: 29, 155: 29, 138:
29, 168: 28, 166: 28, 167: 27, 132: 27, 235: 26, 176: 26, 159: 26, 146: 26, 141:
26, 119: 26, 266: 25, 195: 25, 188: 25, 183: 25, 165: 25, 153: 25, 210: 24, 174:
24, 162: 24, 152: 24, 137: 24, 250: 23, 226: 23, 216: 23, 205: 23, 192: 23, 181:
23, 233: 22, 196: 22, 186: 22, 178: 22, 164: 22, 158: 22, 241: 21, 240: 21, 214:
21, 204: 21, 198: 21, 197: 21, 189: 21, 185: 21, 179: 21, 288: 20, 276: 20, 225:
20, 207: 20, 187: 20, 184: 20, 182: 20, 268: 19, 237: 19, 228: 19, 208: 19, 202:
19, 175: 19, 173: 19, 169: 19, 157: 19, 272: 18, 270: 18, 267: 18, 229: 18, 222:
18, 218: 18, 201: 18, 193: 18, 172: 18, 170: 18, 275: 17, 261: 17, 258: 17, 249:
```

17, 230: 17, 227: 17, 223: 17, 194: 17, 190: 17, 302: 16, 301: 16, 248: 16, 245:
16, 221: 16, 219: 16, 215: 16, 211: 16, 206: 16, 290: 15, 265: 15, 257: 15, 251:
15, 242: 15, 238: 15, 224: 15, 212: 15, 209: 15, 203: 15, 200: 15, 191: 15, 412:
14, 331: 14, 323: 14, 295: 14, 294: 14, 285: 14, 284: 14, 254: 14, 244: 14, 344:
13, 336: 13, 312: 13, 310: 13, 305: 13, 304: 13, 292: 13, 281: 13, 279: 13, 274:
13, 259: 13, 234: 13, 220: 13, 327: 12, 322: 12, 320: 12, 316: 12, 308: 12, 306:
12, 297: 12, 280: 12, 273: 12, 263: 12, 260: 12, 256: 12, 246: 12, 199: 12, 473:
11, 416: 11, 414: 11, 367: 11, 366: 11, 363: 11, 360: 11, 348: 11, 345: 11, 338:
11, 332: 11, 330: 11, 324: 11, 311: 11, 309: 11, 300: 11, 282: 11, 264: 11, 262:
11, 252: 11, 217: 11, 661: 10, 461: 10, 457: 10, 444: 10, 431: 10, 403: 10, 359:
10, 341: 10, 329: 10, 319: 10, 313: 10, 307: 10, 289: 10, 286: 10, 277: 10, 269:
10, 255: 10, 247: 10, 239: 10, 232: 10, 760: 9, 557: 9, 541: 9, 523: 9, 510: 9,
490: 9, 413: 9, 406: 9, 400: 9, 396: 9, 394: 9, 386: 9, 381: 9, 378: 9, 362: 9,
361: 9, 353: 9, 351: 9, 346: 9, 342: 9, 334: 9, 325: 9, 318: 9, 298: 9, 296: 9,
291: 9, 278: 9, 271: 9, 236: 9, 543: 8, 522: 8, 495: 8, 482: 8, 470: 8, 465: 8,
459: 8, 450: 8, 448: 8, 434: 8, 430: 8, 415: 8, 411: 8, 402: 8, 398: 8, 388: 8,
387: 8, 385: 8, 377: 8, 372: 8, 371: 8, 368: 8, 355: 8, 354: 8, 350: 8, 347: 8,
339: 8, 315: 8, 314: 8, 293: 8, 253: 8, 243: 8, 1262: 7, 763: 7, 634: 7, 559: 7,
532: 7, 528: 7, 515: 7, 513: 7, 508: 7, 456: 7, 446: 7, 429: 7, 420: 7, 419: 7,
418: 7, 401: 7, 395: 7, 391: 7, 389: 7, 382: 7, 379: 7, 376: 7, 369: 7, 352: 7,
349: 7, 287: 7, 283: 7, 1268: 6, 1047: 6, 1030: 6, 919: 6, 886: 6, 864: 6, 839:
6, 794: 6, 767: 6, 705: 6, 687: 6, 658: 6, 645: 6, 643: 6, 613: 6, 593: 6, 574:
6, 573: 6, 545: 6, 534: 6, 511: 6, 504: 6, 500: 6, 491: 6, 488: 6, 484: 6, 481:
6, 480: 6, 464: 6, 455: 6, 436: 6, 432: 6, 427: 6, 425: 6, 422: 6, 405: 6, 404:
6, 397: 6, 392: 6, 390: 6, 373: 6, 370: 6, 358: 6, 340: 6, 337: 6, 326: 6, 321:
6, 317: 6, 231: 6, 2433: 5, 1342: 5, 1056: 5, 1011: 5, 953: 5, 936: 5, 895: 5,
871: 5, 860: 5, 815: 5, 802: 5, 758: 5, 757: 5, 734: 5, 731: 5, 707: 5, 695: 5,
683: 5, 678: 5, 673: 5, 670: 5, 669: 5, 656: 5, 649: 5, 631: 5, 625: 5, 616: 5,
614: 5, 604: 5, 603: 5, 601: 5, 594: 5, 589: 5, 588: 5, 582: 5, 581: 5, 577: 5,
576: 5, 568: 5, 566: 5, 563: 5, 560: 5, 556: 5, 554: 5, 552: 5, 544: 5, 539: 5,
538: 5, 536: 5, 535: 5, 531: 5, 530: 5, 509: 5, 507: 5, 506: 5, 505: 5, 498: 5,
485: 5, 478: 5, 472: 5, 469: 5, 467: 5, 466: 5, 454: 5, 445: 5, 442: 5, 439: 5,
428: 5, 426: 5, 421: 5, 383: 5, 380: 5, 356: 5, 333: 5, 328: 5, 2258: 4, 1698:
4, 1622: 4, 1453: 4, 1340: 4, 1306: 4, 1301: 4, 1233: 4, 1112: 4, 1050: 4, 1040:
4, 1015: 4, 1010: 4, 1007: 4, 979: 4, 951: 4, 947: 4, 925: 4, 882: 4, 866: 4,
853: 4, 843: 4, 833: 4, 831: 4, 822: 4, 821: 4, 817: 4, 812: 4, 808: 4, 803: 4,
799: 4, 797: 4, 791: 4, 790: 4, 785: 4, 781: 4, 777: 4, 775: 4, 773: 4, 768: 4,
766: 4, 752: 4, 739: 4, 730: 4, 725: 4, 711: 4, 703: 4, 702: 4, 701: 4, 697: 4,
681: 4, 680: 4, 671: 4, 665: 4, 662: 4, 657: 4, 650: 4, 646: 4, 644: 4, 640: 4,
633: 4, 622: 4, 621: 4, 620: 4, 619: 4, 618: 4, 612: 4, 610: 4, 607: 4, 606: 4,
598: 4, 597: 4, 595: 4, 586: 4, 585: 4, 549: 4, 540: 4, 529: 4, 527: 4, 521: 4,
520: 4, 519: 4, 516: 4, 514: 4, 494: 4, 492: 4, 489: 4, 486: 4, 483: 4, 477: 4,
476: 4, 474: 4, 471: 4, 463: 4, 462: 4, 460: 4, 458: 4, 449: 4, 443: 4, 441: 4,
440: 4, 438: 4, 417: 4, 409: 4, 408: 4, 399: 4, 384: 4, 375: 4, 374: 4, 364: 4,
357: 4, 343: 4, 335: 4, 303: 4, 6087: 3, 3433: 3, 2761: 3, 2672: 3, 2512: 3,
2462: 3, 2440: 3, 2249: 3, 2212: 3, 2207: 3, 2176: 3, 2126: 3, 2110: 3, 2038: 3,
2027: 3, 2026: 3, 2017: 3, 1859: 3, 1839: 3, 1804: 3, 1796: 3, 1766: 3, 1755: 3,
1707: 3, 1702: 3, 1700: 3, 1677: 3, 1656: 3, 1640: 3, 1633: 3, 1629: 3, 1625: 3,

1613: 3, 1603: 3, 1598: 3, 1595: 3, 1584: 3, 1575: 3, 1565: 3, 1538: 3, 1530: 3,
1523: 3, 1511: 3, 1397: 3, 1391: 3, 1379: 3, 1378: 3, 1354: 3, 1350: 3, 1319: 3,
1316: 3, 1286: 3, 1274: 3, 1244: 3, 1241: 3, 1231: 3, 1230: 3, 1211: 3, 1205: 3,
1200: 3, 1196: 3, 1195: 3, 1188: 3, 1177: 3, 1174: 3, 1171: 3, 1165: 3, 1162: 3,
1161: 3, 1143: 3, 1137: 3, 1132: 3, 1130: 3, 1119: 3, 1117: 3, 1110: 3, 1107: 3,
1088: 3, 1080: 3, 1077: 3, 1069: 3, 1066: 3, 1058: 3, 1039: 3, 1017: 3, 1016: 3,
1003: 3, 1002: 3, 1000: 3, 996: 3, 995: 3, 994: 3, 985: 3, 982: 3, 977: 3, 975:
3, 969: 3, 923: 3, 917: 3, 915: 3, 914: 3, 907: 3, 902: 3, 901: 3, 896: 3, 892:
3, 890: 3, 888: 3, 887: 3, 875: 3, 873: 3, 857: 3, 848: 3, 844: 3, 840: 3, 836:
3, 832: 3, 829: 3, 825: 3, 824: 3, 801: 3, 798: 3, 796: 3, 795: 3, 788: 3, 779:
3, 778: 3, 776: 3, 769: 3, 756: 3, 744: 3, 741: 3, 737: 3, 736: 3, 728: 3, 727:
3, 726: 3, 724: 3, 723: 3, 722: 3, 721: 3, 714: 3, 712: 3, 710: 3, 708: 3, 706:
3, 698: 3, 694: 3, 689: 3, 686: 3, 679: 3, 675: 3, 668: 3, 664: 3, 663: 3, 660:
3, 655: 3, 653: 3, 652: 3, 651: 3, 648: 3, 642: 3, 638: 3, 629: 3, 624: 3, 608:
3, 605: 3, 602: 3, 600: 3, 596: 3, 587: 3, 584: 3, 579: 3, 572: 3, 571: 3, 569:
3, 567: 3, 565: 3, 558: 3, 553: 3, 550: 3, 537: 3, 526: 3, 525: 3, 524: 3, 518:
3, 503: 3, 502: 3, 501: 3, 499: 3, 496: 3, 493: 3, 447: 3, 435: 3, 433: 3, 424:
3, 410: 3, 365: 3, 299: 3, 11587: 2, 7176: 2, 6844: 2, 6286: 2, 6013: 2, 5900:
2, 5667: 2, 5613: 2, 4895: 2, 4747: 2, 4491: 2, 4175: 2, 4120: 2, 4034: 2, 4012:
2, 3945: 2, 3862: 2, 3650: 2, 3612: 2, 3583: 2, 3569: 2, 3567: 2, 3468: 2, 3465:
2, 3427: 2, 3417: 2, 3415: 2, 3389: 2, 3361: 2, 3350: 2, 3293: 2, 3287: 2, 3267:
2, 3242: 2, 3217: 2, 3147: 2, 3124: 2, 3067: 2, 3032: 2, 3030: 2, 3009: 2, 2966:
2, 2899: 2, 2715: 2, 2677: 2, 2662: 2, 2639: 2, 2617: 2, 2598: 2, 2568: 2, 2543:
2, 2518: 2, 2508: 2, 2485: 2, 2479: 2, 2474: 2, 2472: 2, 2458: 2, 2445: 2, 2434:
2, 2395: 2, 2343: 2, 2323: 2, 2293: 2, 2287: 2, 2263: 2, 2247: 2, 2246: 2, 2221:
2, 2208: 2, 2185: 2, 2179: 2, 2173: 2, 2167: 2, 2125: 2, 2121: 2, 2114: 2, 2107:
2, 2101: 2, 2099: 2, 2076: 2, 2075: 2, 2074: 2, 2053: 2, 2049: 2, 2035: 2, 2034:
2, 2030: 2, 2021: 2, 1997: 2, 1993: 2, 1974: 2, 1965: 2, 1957: 2, 1945: 2, 1927:
2, 1912: 2, 1906: 2, 1896: 2, 1895: 2, 1888: 2, 1887: 2, 1877: 2, 1874: 2, 1864:
2, 1855: 2, 1851: 2, 1849: 2, 1837: 2, 1824: 2, 1808: 2, 1794: 2, 1786: 2, 1784:
2, 1774: 2, 1771: 2, 1769: 2, 1763: 2, 1759: 2, 1757: 2, 1756: 2, 1752: 2, 1729:
2, 1714: 2, 1697: 2, 1676: 2, 1668: 2, 1660: 2, 1651: 2, 1646: 2, 1638: 2, 1635:
2, 1628: 2, 1614: 2, 1611: 2, 1601: 2, 1589: 2, 1571: 2, 1562: 2, 1556: 2, 1546:
2, 1540: 2, 1531: 2, 1510: 2, 1508: 2, 1501: 2, 1498: 2, 1495: 2, 1492: 2, 1491:
2, 1488: 2, 1484: 2, 1476: 2, 1474: 2, 1469: 2, 1468: 2, 1465: 2, 1464: 2, 1459:
2, 1451: 2, 1440: 2, 1430: 2, 1423: 2, 1421: 2, 1419: 2, 1418: 2, 1415: 2, 1413:
2, 1396: 2, 1393: 2, 1389: 2, 1384: 2, 1380: 2, 1376: 2, 1371: 2, 1365: 2, 1346:
2, 1345: 2, 1332: 2, 1331: 2, 1330: 2, 1329: 2, 1323: 2, 1321: 2, 1312: 2, 1311:
2, 1310: 2, 1302: 2, 1300: 2, 1299: 2, 1297: 2, 1294: 2, 1293: 2, 1292: 2, 1289:
2, 1285: 2, 1282: 2, 1276: 2, 1273: 2, 1270: 2, 1267: 2, 1266: 2, 1265: 2, 1263:
2, 1259: 2, 1257: 2, 1242: 2, 1236: 2, 1235: 2, 1234: 2, 1223: 2, 1219: 2, 1215:
2, 1213: 2, 1212: 2, 1209: 2, 1207: 2, 1206: 2, 1203: 2, 1202: 2, 1201: 2, 1191:
2, 1186: 2, 1180: 2, 1176: 2, 1175: 2, 1170: 2, 1169: 2, 1164: 2, 1159: 2, 1157:
2, 1154: 2, 1150: 2, 1146: 2, 1144: 2, 1141: 2, 1135: 2, 1133: 2, 1129: 2, 1122:
2, 1121: 2, 1115: 2, 1109: 2, 1106: 2, 1099: 2, 1094: 2, 1092: 2, 1090: 2, 1087:
2, 1085: 2, 1081: 2, 1079: 2, 1072: 2, 1070: 2, 1065: 2, 1061: 2, 1057: 2, 1049:
2, 1043: 2, 1041: 2, 1037: 2, 1028: 2, 1019: 2, 1014: 2, 1009: 2, 1008: 2, 1005:
2, 1004: 2, 998: 2, 989: 2, 983: 2, 970: 2, 967: 2, 964: 2, 962: 2, 959: 2, 949:

2, 948: 2, 946: 2, 937: 2, 934: 2, 933: 2, 930: 2, 927: 2, 926: 2, 924: 2, 916:
2, 911: 2, 906: 2, 905: 2, 904: 2, 903: 2, 894: 2, 884: 2, 880: 2, 876: 2, 872:
2, 869: 2, 867: 2, 862: 2, 861: 2, 858: 2, 855: 2, 854: 2, 847: 2, 846: 2, 845:
2, 841: 2, 838: 2, 837: 2, 834: 2, 828: 2, 826: 2, 823: 2, 818: 2, 811: 2, 806:
2, 800: 2, 786: 2, 782: 2, 780: 2, 771: 2, 770: 2, 762: 2, 761: 2, 755: 2, 754:
2, 753: 2, 748: 2, 747: 2, 746: 2, 743: 2, 733: 2, 729: 2, 716: 2, 715: 2, 713:
2, 709: 2, 699: 2, 693: 2, 691: 2, 688: 2, 677: 2, 676: 2, 674: 2, 672: 2, 666:
2, 654: 2, 647: 2, 641: 2, 639: 2, 637: 2, 636: 2, 635: 2, 630: 2, 617: 2, 615:
2, 611: 2, 591: 2, 590: 2, 580: 2, 570: 2, 564: 2, 562: 2, 555: 2, 551: 2, 548:
2, 546: 2, 542: 2, 533: 2, 517: 2, 512: 2, 497: 2, 487: 2, 475: 2, 468: 2, 453:
2, 452: 2, 451: 2, 437: 2, 423: 2, 407: 2, 393: 2, 151617: 1, 121974: 1, 83057:
1, 69674: 1, 69574: 1, 68674: 1, 67192: 1, 64057: 1, 63710: 1, 55156: 1, 53542:
1, 51548: 1, 49027: 1, 47056: 1, 46054: 1, 45220: 1, 43270: 1, 42635: 1, 42529:
1, 42143: 1, 40893: 1, 40857: 1, 39920: 1, 39419: 1, 38703: 1, 38236: 1, 36016:
1, 35907: 1, 35735: 1, 34923: 1, 33840: 1, 33723: 1, 32992: 1, 32611: 1, 32346:
1, 31957: 1, 29215: 1, 28645: 1, 27826: 1, 27316: 1, 26727: 1, 26045: 1, 25991:
1, 25231: 1, 25012: 1, 24968: 1, 24766: 1, 24577: 1, 24449: 1, 24429: 1, 24417:
1, 23923: 1, 23825: 1, 23456: 1, 22556: 1, 22111: 1, 22050: 1, 21563: 1, 21471:
1, 21423: 1, 21191: 1, 20653: 1, 20521: 1, 20219: 1, 19541: 1, 19521: 1, 19516:
1, 19500: 1, 19276: 1, 19135: 1, 19080: 1, 19015: 1, 18859: 1, 18621: 1, 18492:
1, 18468: 1, 18449: 1, 18301: 1, 18209: 1, 18087: 1, 17971: 1, 17947: 1, 17938:
1, 17805: 1, 17735: 1, 17604: 1, 17583: 1, 17501: 1, 17456: 1, 17364: 1, 17153:
1, 17049: 1, 16887: 1, 16811: 1, 16744: 1, 16671: 1, 16190: 1, 16039: 1, 15972:
1, 15898: 1, 15874: 1, 15807: 1, 15778: 1, 15646: 1, 15578: 1, 15550: 1, 15534:
1, 15429: 1, 15327: 1, 15251: 1, 15068: 1, 15028: 1, 14846: 1, 14813: 1, 14731:
1, 14628: 1, 14576: 1, 14507: 1, 14347: 1, 14343: 1, 14337: 1, 13871: 1, 13861:
1, 13854: 1, 13742: 1, 13578: 1, 13568: 1, 13525: 1, 13432: 1, 13391: 1, 13132:
1, 13099: 1, 13096: 1, 13068: 1, 13043: 1, 12988: 1, 12976: 1, 12916: 1, 12861:
1, 12860: 1, 12842: 1, 12817: 1, 12791: 1, 12742: 1, 12738: 1, 12733: 1, 12676:
1, 12588: 1, 12563: 1, 12523: 1, 12511: 1, 12408: 1, 12405: 1, 12402: 1, 12357:
1, 12321: 1, 12235: 1, 12203: 1, 12125: 1, 12113: 1, 12081: 1, 12045: 1, 12040:
1, 12009: 1, 12000: 1, 11957: 1, 11887: 1, 11815: 1, 11794: 1, 11793: 1, 11747:
1, 11690: 1, 11641: 1, 11603: 1, 11601: 1, 11435: 1, 11335: 1, 11313: 1, 11309:
1, 11243: 1, 11177: 1, 11171: 1, 11033: 1, 10935: 1, 10857: 1, 10748: 1, 10715:
1, 10660: 1, 10608: 1, 10539: 1, 10507: 1, 10457: 1, 10393: 1, 10280: 1, 10256:
1, 10218: 1, 10208: 1, 10193: 1, 10106: 1, 10093: 1, 10090: 1, 10086: 1, 10065:
1, 10013: 1, 9978: 1, 9889: 1, 9876: 1, 9871: 1, 9841: 1, 9789: 1, 9738: 1,
9695: 1, 9689: 1, 9649: 1, 9588: 1, 9582: 1, 9465: 1, 9425: 1, 9391: 1, 9372: 1,
9359: 1, 9352: 1, 9297: 1, 9284: 1, 9280: 1, 9217: 1, 9208: 1, 9175: 1, 9138: 1,
9116: 1, 9100: 1, 9071: 1, 9063: 1, 9059: 1, 9046: 1, 9030: 1, 9015: 1, 9011: 1,
9009: 1, 8989: 1, 8972: 1, 8970: 1, 8869: 1, 8841: 1, 8824: 1, 8785: 1, 8782: 1,
8763: 1, 8741: 1, 8675: 1, 8593: 1, 8587: 1, 8567: 1, 8558: 1, 8557: 1, 8555: 1,
8533: 1, 8490: 1, 8454: 1, 8446: 1, 8444: 1, 8388: 1, 8363: 1, 8291: 1, 8290: 1,
8285: 1, 8257: 1, 8229: 1, 8199: 1, 8164: 1, 8163: 1, 8156: 1, 8135: 1, 8102: 1,
8080: 1, 8078: 1, 8072: 1, 8038: 1, 8036: 1, 8035: 1, 8030: 1, 8028: 1, 8007: 1,
7954: 1, 7952: 1, 7888: 1, 7852: 1, 7797: 1, 7795: 1, 7768: 1, 7748: 1, 7739: 1,
7728: 1, 7692: 1, 7655: 1, 7601: 1, 7595: 1, 7565: 1, 7549: 1, 7546: 1, 7510: 1,
7506: 1, 7494: 1, 7477: 1, 7426: 1, 7418: 1, 7368: 1, 7332: 1, 7283: 1, 7248: 1,

7235: 1, 7234: 1, 7232: 1, 7226: 1, 7212: 1, 7191: 1, 7177: 1, 7154: 1, 7146: 1,
7129: 1, 7120: 1, 7113: 1, 7111: 1, 7087: 1, 7072: 1, 7066: 1, 7062: 1, 7054: 1,
7045: 1, 7024: 1, 7023: 1, 7016: 1, 7015: 1, 7009: 1, 6971: 1, 6963: 1, 6954: 1,
6912: 1, 6893: 1, 6869: 1, 6843: 1, 6824: 1, 6816: 1, 6799: 1, 6790: 1, 6789: 1,
6778: 1, 6769: 1, 6765: 1, 6749: 1, 6721: 1, 6703: 1, 6689: 1, 6659: 1, 6649: 1,
6640: 1, 6583: 1, 6582: 1, 6555: 1, 6545: 1, 6500: 1, 6478: 1, 6472: 1, 6454: 1,
6436: 1, 6422: 1, 6412: 1, 6391: 1, 6387: 1, 6345: 1, 6329: 1, 6327: 1, 6323: 1,
6308: 1, 6306: 1, 6289: 1, 6263: 1, 6243: 1, 6223: 1, 6217: 1, 6205: 1, 6200: 1,
6198: 1, 6197: 1, 6147: 1, 6146: 1, 6136: 1, 6126: 1, 6117: 1, 6107: 1, 6106: 1,
6076: 1, 6011: 1, 5983: 1, 5958: 1, 5957: 1, 5947: 1, 5906: 1, 5895: 1, 5890: 1,
5889: 1, 5887: 1, 5878: 1, 5874: 1, 5825: 1, 5824: 1, 5809: 1, 5797: 1, 5788: 1,
5778: 1, 5773: 1, 5745: 1, 5730: 1, 5683: 1, 5673: 1, 5655: 1, 5632: 1, 5584: 1,
5580: 1, 5579: 1, 5570: 1, 5547: 1, 5533: 1, 5508: 1, 5499: 1, 5484: 1, 5482: 1,
5480: 1, 5475: 1, 5461: 1, 5456: 1, 5450: 1, 5428: 1, 5426: 1, 5419: 1, 5407: 1,
5394: 1, 5387: 1, 5363: 1, 5349: 1, 5341: 1, 5338: 1, 5319: 1, 5316: 1, 5301: 1,
5289: 1, 5277: 1, 5251: 1, 5240: 1, 5238: 1, 5219: 1, 5213: 1, 5209: 1, 5155: 1,
5151: 1, 5140: 1, 5136: 1, 5135: 1, 5119: 1, 5115: 1, 5091: 1, 5058: 1, 5056: 1,
5049: 1, 5039: 1, 5036: 1, 5034: 1, 5027: 1, 5026: 1, 5019: 1, 5011: 1, 5009: 1,
5007: 1, 5004: 1, 4997: 1, 4990: 1, 4987: 1, 4974: 1, 4973: 1, 4956: 1, 4935: 1,
4933: 1, 4932: 1, 4919: 1, 4900: 1, 4879: 1, 4869: 1, 4867: 1, 4858: 1, 4856: 1,
4851: 1, 4850: 1, 4839: 1, 4835: 1, 4833: 1, 4829: 1, 4827: 1, 4825: 1, 4823: 1,
4816: 1, 4815: 1, 4790: 1, 4788: 1, 4784: 1, 4770: 1, 4765: 1, 4745: 1, 4735: 1,
4713: 1, 4710: 1, 4698: 1, 4695: 1, 4693: 1, 4680: 1, 4671: 1, 4666: 1, 4665: 1,
4656: 1, 4652: 1, 4651: 1, 4649: 1, 4623: 1, 4612: 1, 4608: 1, 4603: 1, 4598: 1,
4586: 1, 4566: 1, 4559: 1, 4540: 1, 4537: 1, 4535: 1, 4525: 1, 4524: 1, 4513: 1,
4508: 1, 4497: 1, 4483: 1, 4481: 1, 4456: 1, 4450: 1, 4436: 1, 4432: 1, 4421: 1,
4412: 1, 4409: 1, 4399: 1, 4390: 1, 4387: 1, 4375: 1, 4364: 1, 4359: 1, 4357: 1,
4353: 1, 4350: 1, 4344: 1, 4333: 1, 4328: 1, 4323: 1, 4322: 1, 4319: 1, 4318: 1,
4314: 1, 4309: 1, 4308: 1, 4301: 1, 4289: 1, 4284: 1, 4277: 1, 4273: 1, 4261: 1,
4259: 1, 4253: 1, 4243: 1, 4242: 1, 4241: 1, 4234: 1, 4230: 1, 4229: 1, 4220: 1,
4218: 1, 4212: 1, 4208: 1, 4186: 1, 4184: 1, 4165: 1, 4161: 1, 4159: 1, 4141: 1,
4132: 1, 4130: 1, 4119: 1, 4103: 1, 4099: 1, 4097: 1, 4084: 1, 4070: 1, 4059: 1,
4056: 1, 4049: 1, 4047: 1, 4040: 1, 4022: 1, 4020: 1, 4009: 1, 4007: 1, 4006: 1,
3997: 1, 3996: 1, 3993: 1, 3991: 1, 3989: 1, 3980: 1, 3975: 1, 3967: 1, 3962: 1,
3951: 1, 3948: 1, 3942: 1, 3941: 1, 3939: 1, 3936: 1, 3928: 1, 3913: 1, 3912: 1,
3899: 1, 3893: 1, 3892: 1, 3889: 1, 3883: 1, 3881: 1, 3877: 1, 3872: 1, 3866: 1,
3865: 1, 3863: 1, 3859: 1, 3857: 1, 3849: 1, 3846: 1, 3828: 1, 3814: 1, 3813: 1,
3807: 1, 3806: 1, 3795: 1, 3789: 1, 3788: 1, 3786: 1, 3784: 1, 3778: 1, 3753: 1,
3750: 1, 3743: 1, 3741: 1, 3737: 1, 3736: 1, 3733: 1, 3732: 1, 3725: 1, 3709: 1,
3702: 1, 3701: 1, 3697: 1, 3690: 1, 3687: 1, 3685: 1, 3683: 1, 3677: 1, 3670: 1,
3666: 1, 3662: 1, 3655: 1, 3644: 1, 3641: 1, 3637: 1, 3636: 1, 3634: 1, 3623: 1,
3611: 1, 3610: 1, 3605: 1, 3604: 1, 3593: 1, 3577: 1, 3575: 1, 3563: 1, 3560: 1,
3553: 1, 3550: 1, 3549: 1, 3546: 1, 3538: 1, 3530: 1, 3512: 1, 3509: 1, 3508: 1,
3502: 1, 3500: 1, 3484: 1, 3483: 1, 3482: 1, 3478: 1, 3476: 1, 3474: 1, 3473: 1,
3454: 1, 3452: 1, 3449: 1, 3443: 1, 3440: 1, 3439: 1, 3434: 1, 3430: 1, 3426: 1,
3424: 1, 3423: 1, 3421: 1, 3410: 1, 3407: 1, 3403: 1, 3388: 1, 3386: 1, 3384: 1,
3381: 1, 3378: 1, 3376: 1, 3374: 1, 3370: 1, 3348: 1, 3347: 1, 3344: 1, 3343: 1,
3340: 1, 3339: 1, 3336: 1, 3331: 1, 3326: 1, 3321: 1, 3319: 1, 3315: 1, 3311: 1,

3294: 1, 3280: 1, 3275: 1, 3271: 1, 3270: 1, 3269: 1, 3268: 1, 3259: 1, 3253: 1,
3243: 1, 3241: 1, 3239: 1, 3237: 1, 3229: 1, 3228: 1, 3224: 1, 3218: 1, 3215: 1,
3209: 1, 3203: 1, 3199: 1, 3196: 1, 3188: 1, 3186: 1, 3179: 1, 3174: 1, 3172: 1,
3160: 1, 3156: 1, 3154: 1, 3149: 1, 3133: 1, 3131: 1, 3129: 1, 3127: 1, 3119: 1,
3117: 1, 3115: 1, 3114: 1, 3110: 1, 3106: 1, 3103: 1, 3101: 1, 3099: 1, 3093: 1,
3092: 1, 3087: 1, 3079: 1, 3071: 1, 3069: 1, 3065: 1, 3060: 1, 3050: 1, 3029: 1,
3027: 1, 3026: 1, 3021: 1, 3020: 1, 3016: 1, 3011: 1, 3008: 1, 3006: 1, 3005: 1,
2999: 1, 2988: 1, 2985: 1, 2984: 1, 2977: 1, 2976: 1, 2972: 1, 2971: 1, 2969: 1,
2950: 1, 2947: 1, 2945: 1, 2942: 1, 2938: 1, 2937: 1, 2932: 1, 2925: 1, 2919: 1,
2913: 1, 2905: 1, 2896: 1, 2893: 1, 2890: 1, 2883: 1, 2872: 1, 2866: 1, 2864: 1,
2861: 1, 2858: 1, 2856: 1, 2855: 1, 2853: 1, 2844: 1, 2840: 1, 2830: 1, 2820: 1,
2819: 1, 2816: 1, 2808: 1, 2803: 1, 2795: 1, 2789: 1, 2781: 1, 2776: 1, 2769: 1,
2762: 1, 2760: 1, 2759: 1, 2754: 1, 2747: 1, 2745: 1, 2739: 1, 2735: 1, 2733: 1,
2728: 1, 2727: 1, 2726: 1, 2725: 1, 2719: 1, 2716: 1, 2710: 1, 2706: 1, 2704: 1,
2703: 1, 2701: 1, 2698: 1, 2695: 1, 2691: 1, 2685: 1, 2682: 1, 2681: 1, 2678: 1,
2674: 1, 2671: 1, 2665: 1, 2661: 1, 2654: 1, 2649: 1, 2646: 1, 2645: 1, 2644: 1,
2641: 1, 2640: 1, 2637: 1, 2636: 1, 2633: 1, 2632: 1, 2628: 1, 2623: 1, 2620: 1,
2614: 1, 2613: 1, 2610: 1, 2609: 1, 2607: 1, 2600: 1, 2594: 1, 2593: 1, 2588: 1,
2581: 1, 2578: 1, 2572: 1, 2571: 1, 2570: 1, 2569: 1, 2562: 1, 2559: 1, 2556: 1,
2555: 1, 2554: 1, 2547: 1, 2545: 1, 2534: 1, 2532: 1, 2530: 1, 2529: 1, 2509: 1,
2507: 1, 2504: 1, 2502: 1, 2501: 1, 2499: 1, 2496: 1, 2493: 1, 2492: 1, 2491: 1,
2490: 1, 2476: 1, 2470: 1, 2469: 1, 2467: 1, 2460: 1, 2456: 1, 2453: 1, 2452: 1,
2451: 1, 2437: 1, 2435: 1, 2427: 1, 2425: 1, 2424: 1, 2423: 1, 2422: 1, 2420: 1,
2417: 1, 2412: 1, 2408: 1, 2403: 1, 2400: 1, 2399: 1, 2393: 1, 2391: 1, 2386: 1,
2385: 1, 2384: 1, 2383: 1, 2379: 1, 2374: 1, 2371: 1, 2365: 1, 2364: 1, 2352: 1,
2351: 1, 2348: 1, 2344: 1, 2338: 1, 2336: 1, 2327: 1, 2324: 1, 2319: 1, 2315: 1,
2314: 1, 2312: 1, 2300: 1, 2295: 1, 2289: 1, 2284: 1, 2280: 1, 2278: 1, 2273: 1,
2271: 1, 2264: 1, 2260: 1, 2257: 1, 2256: 1, 2255: 1, 2251: 1, 2245: 1, 2244: 1,
2241: 1, 2237: 1, 2236: 1, 2233: 1, 2222: 1, 2220: 1, 2214: 1, 2210: 1, 2209: 1,
2206: 1, 2203: 1, 2202: 1, 2200: 1, 2198: 1, 2197: 1, 2196: 1, 2195: 1, 2194: 1,
2192: 1, 2190: 1, 2189: 1, 2182: 1, 2177: 1, 2174: 1, 2170: 1, 2168: 1, 2166: 1,
2163: 1, 2159: 1, 2157: 1, 2156: 1, 2147: 1, 2146: 1, 2145: 1, 2143: 1, 2133: 1,
2132: 1, 2123: 1, 2118: 1, 2117: 1, 2115: 1, 2112: 1, 2106: 1, 2100: 1, 2096: 1,
2093: 1, 2088: 1, 2087: 1, 2084: 1, 2081: 1, 2080: 1, 2073: 1, 2072: 1, 2069: 1,
2066: 1, 2060: 1, 2059: 1, 2057: 1, 2054: 1, 2052: 1, 2048: 1, 2043: 1, 2042: 1,
2041: 1, 2039: 1, 2031: 1, 2023: 1, 2018: 1, 2016: 1, 2014: 1, 2013: 1, 2004: 1,
2001: 1, 1999: 1, 1998: 1, 1995: 1, 1994: 1, 1988: 1, 1984: 1, 1979: 1, 1978: 1,
1975: 1, 1969: 1, 1964: 1, 1962: 1, 1961: 1, 1959: 1, 1955: 1, 1954: 1, 1952: 1,
1951: 1, 1949: 1, 1948: 1, 1946: 1, 1944: 1, 1942: 1, 1941: 1, 1939: 1, 1937: 1,
1936: 1, 1929: 1, 1926: 1, 1922: 1, 1915: 1, 1910: 1, 1907: 1, 1903: 1, 1902: 1,
1901: 1, 1899: 1, 1898: 1, 1897: 1, 1893: 1, 1891: 1, 1882: 1, 1880: 1, 1878: 1,
1876: 1, 1872: 1, 1870: 1, 1865: 1, 1862: 1, 1860: 1, 1858: 1, 1857: 1, 1856: 1,
1853: 1, 1850: 1, 1848: 1, 1847: 1, 1845: 1, 1843: 1, 1841: 1, 1840: 1, 1836: 1,
1835: 1, 1832: 1, 1829: 1, 1825: 1, 1823: 1, 1818: 1, 1817: 1, 1814: 1, 1810: 1,
1809: 1, 1795: 1, 1793: 1, 1789: 1, 1782: 1, 1781: 1, 1780: 1, 1778: 1, 1776: 1,
1775: 1, 1773: 1, 1770: 1, 1761: 1, 1760: 1, 1758: 1, 1753: 1, 1751: 1, 1750: 1,
1748: 1, 1747: 1, 1745: 1, 1742: 1, 1741: 1, 1737: 1, 1733: 1, 1732: 1, 1725: 1,
1720: 1, 1719: 1, 1718: 1, 1712: 1, 1708: 1, 1705: 1, 1694: 1, 1693: 1, 1689: 1,

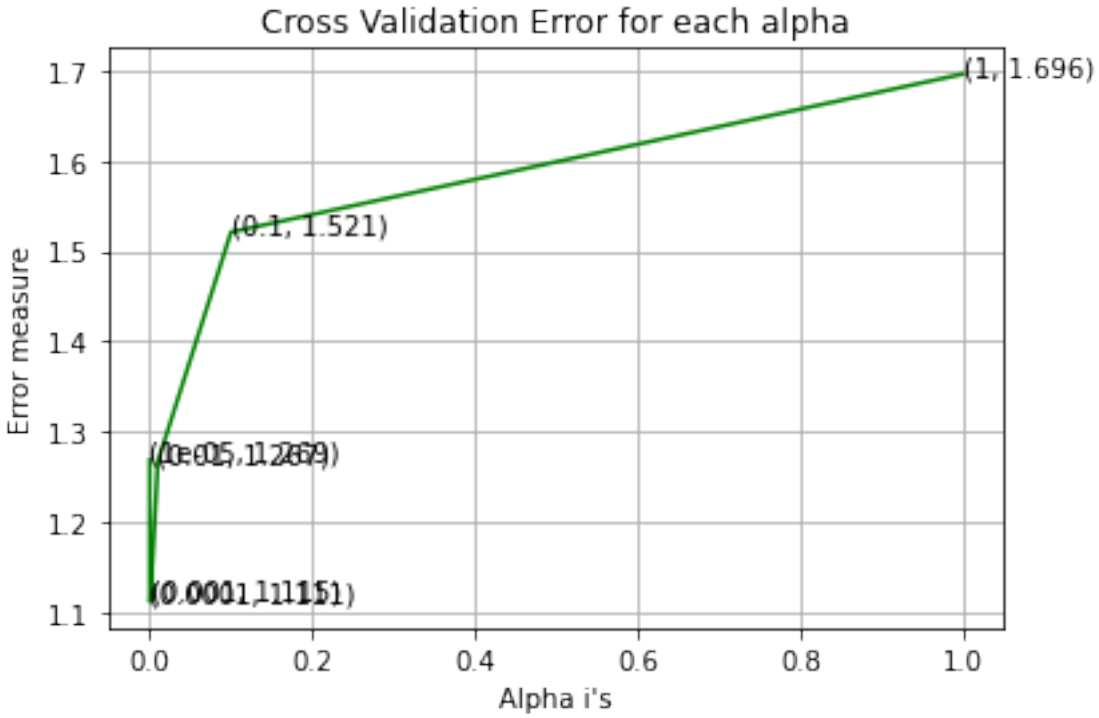
```
1688: 1, 1686: 1, 1685: 1, 1682: 1, 1681: 1, 1679: 1, 1675: 1, 1672: 1, 1671: 1,
1669: 1, 1667: 1, 1662: 1, 1659: 1, 1657: 1, 1654: 1, 1652: 1, 1649: 1, 1644: 1,
1642: 1, 1639: 1, 1636: 1, 1634: 1, 1631: 1, 1624: 1, 1617: 1, 1616: 1, 1609: 1,
1608: 1, 1607: 1, 1602: 1, 1599: 1, 1597: 1, 1596: 1, 1594: 1, 1592: 1, 1591: 1,
1587: 1, 1582: 1, 1579: 1, 1578: 1, 1570: 1, 1569: 1, 1568: 1, 1566: 1, 1564: 1,
1563: 1, 1560: 1, 1558: 1, 1557: 1, 1553: 1, 1547: 1, 1545: 1, 1542: 1, 1541: 1,
1539: 1, 1536: 1, 1535: 1, 1533: 1, 1524: 1, 1522: 1, 1521: 1, 1519: 1, 1516: 1,
1512: 1, 1509: 1, 1507: 1, 1500: 1, 1499: 1, 1497: 1, 1496: 1, 1493: 1, 1485: 1,
1480: 1, 1479: 1, 1477: 1, 1475: 1, 1473: 1, 1470: 1, 1466: 1, 1463: 1, 1462: 1,
1460: 1, 1454: 1, 1450: 1, 1449: 1, 1448: 1, 1446: 1, 1444: 1, 1443: 1, 1441: 1,
1438: 1, 1436: 1, 1435: 1, 1434: 1, 1432: 1, 1425: 1, 1424: 1, 1422: 1, 1420: 1,
1417: 1, 1409: 1, 1408: 1, 1406: 1, 1405: 1, 1403: 1, 1401: 1, 1400: 1, 1392: 1,
1390: 1, 1388: 1, 1385: 1, 1377: 1, 1375: 1, 1374: 1, 1372: 1, 1367: 1, 1366: 1,
1364: 1, 1363: 1, 1362: 1, 1361: 1, 1360: 1, 1359: 1, 1358: 1, 1357: 1, 1356: 1,
1355: 1, 1351: 1, 1347: 1, 1341: 1, 1339: 1, 1337: 1, 1334: 1, 1333: 1, 1326: 1,
1322: 1, 1320: 1, 1318: 1, 1317: 1, 1315: 1, 1313: 1, 1308: 1, 1307: 1, 1304: 1,
1303: 1, 1298: 1, 1288: 1, 1287: 1, 1284: 1, 1283: 1, 1281: 1, 1280: 1, 1275: 1,
1272: 1, 1271: 1, 1264: 1, 1256: 1, 1255: 1, 1254: 1, 1253: 1, 1252: 1, 1250: 1,
1249: 1, 1246: 1, 1245: 1, 1243: 1, 1240: 1, 1239: 1, 1238: 1, 1237: 1, 1232: 1,
1229: 1, 1228: 1, 1226: 1, 1225: 1, 1221: 1, 1214: 1, 1210: 1, 1208: 1, 1199: 1,
1197: 1, 1194: 1, 1193: 1, 1190: 1, 1189: 1, 1187: 1, 1185: 1, 1184: 1, 1183: 1,
1182: 1, 1179: 1, 1178: 1, 1172: 1, 1166: 1, 1160: 1, 1153: 1, 1151: 1, 1149: 1,
1148: 1, 1145: 1, 1140: 1, 1134: 1, 1131: 1, 1128: 1, 1127: 1, 1126: 1, 1125: 1,
1124: 1, 1118: 1, 1114: 1, 1113: 1, 1108: 1, 1105: 1, 1102: 1, 1101: 1, 1100: 1,
1098: 1, 1096: 1, 1091: 1, 1084: 1, 1083: 1, 1082: 1, 1078: 1, 1075: 1, 1074: 1,
1071: 1, 1068: 1, 1054: 1, 1052: 1, 1051: 1, 1048: 1, 1042: 1, 1036: 1, 1035: 1,
1034: 1, 1033: 1, 1029: 1, 1026: 1, 1023: 1, 1020: 1, 1018: 1, 1006: 1, 1001: 1,
997: 1, 993: 1, 992: 1, 991: 1, 988: 1, 987: 1, 986: 1, 980: 1, 978: 1, 976: 1,
973: 1, 972: 1, 971: 1, 965: 1, 963: 1, 961: 1, 958: 1, 955: 1, 952: 1, 945: 1,
943: 1, 940: 1, 939: 1, 932: 1, 931: 1, 920: 1, 918: 1, 913: 1, 912: 1, 910: 1,
900: 1, 893: 1, 891: 1, 885: 1, 883: 1, 881: 1, 878: 1, 877: 1, 874: 1, 870: 1,
865: 1, 863: 1, 859: 1, 856: 1, 852: 1, 850: 1, 835: 1, 830: 1, 827: 1, 820: 1,
819: 1, 814: 1, 810: 1, 809: 1, 807: 1, 805: 1, 792: 1, 789: 1, 787: 1, 784: 1,
783: 1, 774: 1, 765: 1, 764: 1, 751: 1, 750: 1, 745: 1, 742: 1, 740: 1, 738: 1,
735: 1, 732: 1, 719: 1, 717: 1, 700: 1, 696: 1, 685: 1, 684: 1, 682: 1, 667: 1,
659: 1, 632: 1, 628: 1, 626: 1, 623: 1, 609: 1, 599: 1, 592: 1, 578: 1, 575: 1,
561: 1, 547: 1})
```

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

# read more about SGDClassifier() at http://scikit-learn.org/stable/modules/
 ↪generated/sklearn.linear_model.SGDClassifier.html
# ------------------------------
# default parameters
```

```
# SGDClassifier(loss='hinge', penalty='l2', alpha=0.0001, l1_ratio=0.15,␣
 ↪fit_intercept=True, max_iter=None, tol=None,
# shuffle=True, verbose=0, epsilon=0.1, n_jobs=1, random_state=None,␣
 ↪learning_rate='optimal', eta0=0.0, power_t=0.5,
# class_weight=None, warm_start=False, average=False, n_iter=None)

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

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


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

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

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


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

```
predict_y = sig_clf.predict_proba(train_text_feature_onehotCoding)
print('For values of best alpha = ', alpha[best_alpha], "The train log loss is:
 ↪",log_loss(y_train, predict_y, labels=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 alpha =  1e-05 The log loss is: 1.268657097242295
For values of alpha =  0.0001 The log loss is: 1.1114245601411297
For values of alpha =  0.001 The log loss is: 1.1149128766871008
For values of alpha =  0.01 The log loss is: 1.2667068263268315
For values of alpha =  0.1 The log loss is: 1.5212307923568404
For values of alpha =  1 The log loss is: 1.6964467951923798
```




```
For values of best alpha =  0.0001 The train log loss is: 0.6751262655238751
For values of best alpha =  0.0001 The cross validation log loss is:
1.1114245601411297
For values of best alpha =  0.0001 The test log loss is: 1.142490293894683
```

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

Ans. Yes, it seems like!

[ ]:
```
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")
```

```
97.579 % of word of test data appeared in train data
98.663 % of word of Cross Validation appeared in train data
```

1 4. Machine Learning Models

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

#Misc. functionns for ML models


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

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

[ ]:
```
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 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")
```

Stacking the three types of features

[ ]:
```
# 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)
```

```
One hot encoding features :
(number of data points * number of features) in train data =  (2124, 3180)
(number of data points * number of features) in test data =  (665, 3180)
(number of data points * number of features) in cross validation data = (532,
3180)
```

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

```
 Response encoding features :
(number of data points * number of features) in train data =  (2124, 27)
(number of data points * number of features) in test data =  (665, 27)
(number of data points * number of features) in cross validation data = (532,
27)
```

4.1. Base Line Model

4.1.1. Naive Bayes

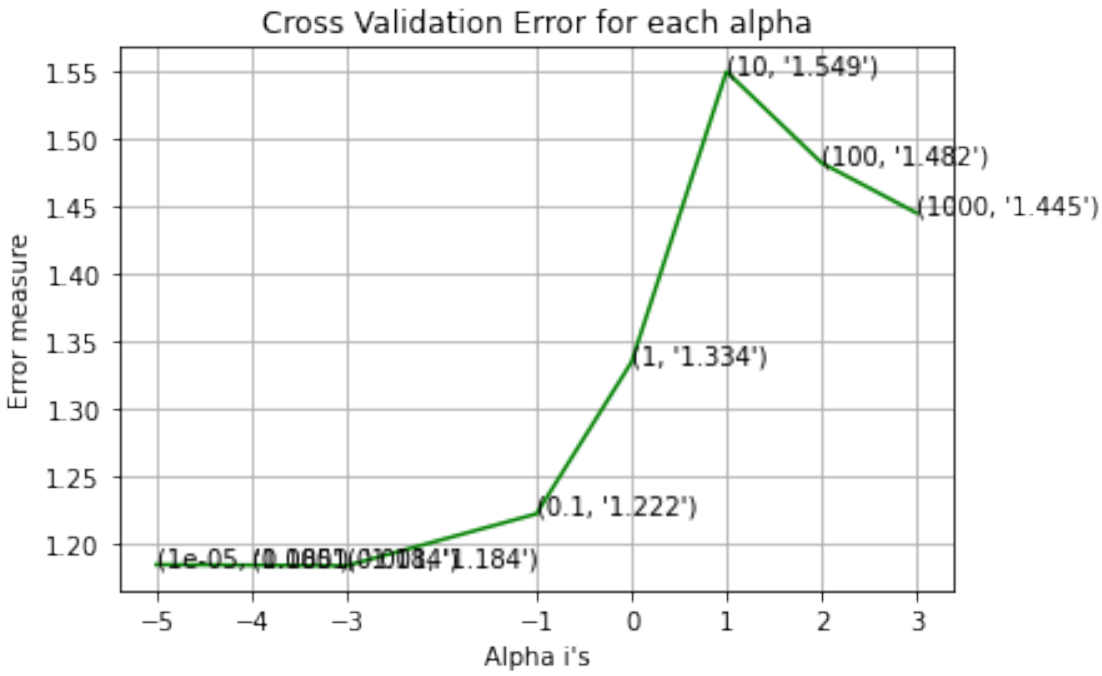
4.1.1.1. Hyper parameter tuning

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

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

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


alpha = [0.00001, 0.0001, 0.001, 0.1, 1, 10, 100,1000]
cv_log_error_array = []
for i in alpha:
    print("for alpha =", i)
    clf = MultinomialNB(alpha=i)
    clf.fit(train_x_onehotCoding, train_y)
    sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
    sig_clf.fit(train_x_onehotCoding, train_y)
    sig_clf_probs = sig_clf.predict_proba(cv_x_onehotCoding)
    cv_log_error_array.append(log_loss(cv_y, sig_clf_probs, labels=clf.
 ↪classes_, eps=1e-15))
    # to avoid rounding error while multiplying probabilites we use␣
 ↪log-probability estimates
    print("Log Loss :",log_loss(cv_y, sig_clf_probs))

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


best_alpha = np.argmin(cv_log_error_array)
```

```
clf = MultinomialNB(alpha=alpha[best_alpha])
clf.fit(train_x_onehotCoding, train_y)
sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
sig_clf.fit(train_x_onehotCoding, train_y)


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

```
for alpha = 1e-05
Log Loss : 1.184518102125294
for alpha = 0.0001
Log Loss : 1.1841543076609153
for alpha = 0.001
Log Loss : 1.1836655497136954
for alpha = 0.1
Log Loss : 1.2220690835705355
for alpha = 1
Log Loss : 1.3342444192151843
for alpha = 10
Log Loss : 1.5491188085094323
for alpha = 100
Log Loss : 1.4816525978387634
for alpha = 1000
Log Loss : 1.4445607952863126
```




```
For values of best alpha =  0.001 The train log loss is: 0.43959574697795245
For values of best alpha =  0.001 The cross validation log loss is:
1.1836655497136954
For values of best alpha =  0.001 The test log loss is: 1.2413172955570213
```

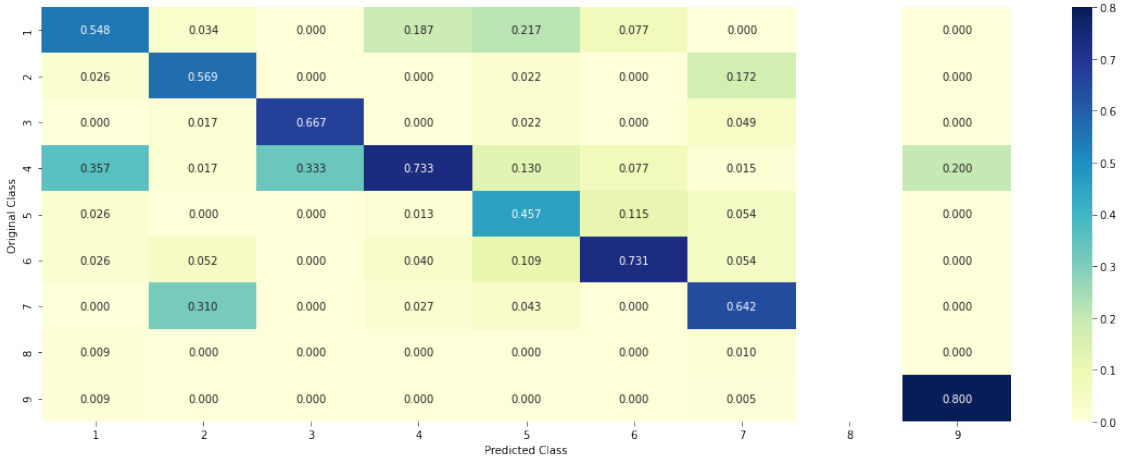
4.1.1.2. Testing the model with best hyper paramters

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

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

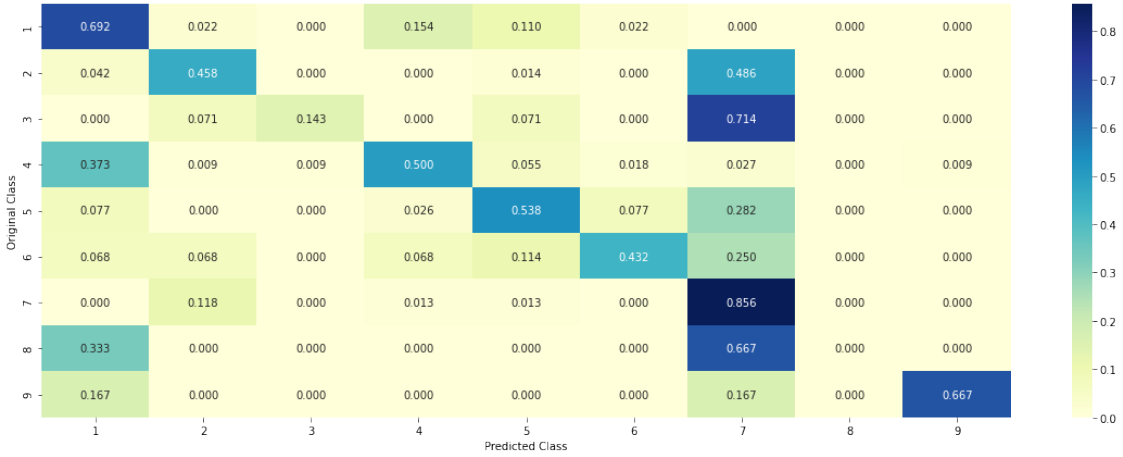
```
# find more about CalibratedClassifierCV here at http://scikit-learn.org/stable/
 ↪modules/generated/sklearn.calibration.CalibratedClassifierCV.html
# ----------------------------
# default paramters
# sklearn.calibration.CalibratedClassifierCV(base_estimator=None,␣
 ↪method='sigmoid', cv=3)
#
#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

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

```
Log Loss : 1.1836655497136954
Number of missclassified point : 0.38345864661654133
--------------------- Confusion matrix ----------------------
```


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



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



4.1.1.3. Feature Importance, Correctly classified point

[ ]:
```
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(-1*abs(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 : 1
Predicted Class Probabilities: [[0.7003 0.0593 0.013  0.064  0.0358 0.034
0.0865 0.004  0.0031]]
Actual Class : 1
--------------------------------------------------
Out of the top  100  features  0 are present in query point
```

4.1.1.4. Feature Importance, Incorrectly classified point

[ ]:
```
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(-1*abs(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 : 9
Predicted Class Probabilities: [[0.0702 0.051  0.0142 0.0703 0.0394 0.0374
0.0953 0.0044 0.6178]]
Actual Class : 9
--------------------------------------------------
Out of the top  100  features  0 are present in query point
```

[ ]:
```
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(-1*abs(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 : 9
Predicted Class Probabilities: [[0.0702 0.051  0.0142 0.0703 0.0394 0.0374
0.0953 0.0044 0.6178]]
Actual Class : 9
--------------------------------------------------
Out of the top  100  features  0 are present in query point
```

4.2. K Nearest Neighbour Classification

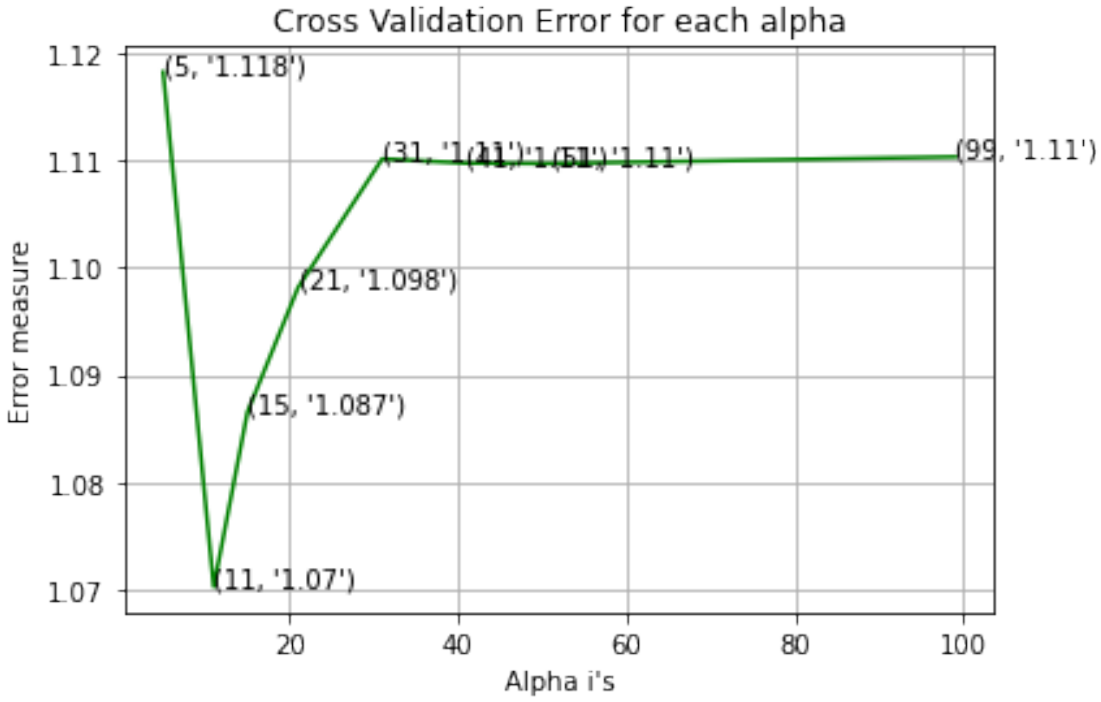
4.2.1. Hyper parameter tuning

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

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


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


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

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

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


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

predict_y = sig_clf.predict_proba(train_x_responseCoding)
print('For values of best alpha = ', alpha[best_alpha], "The train log loss is:
 ↪",log_loss(y_train, predict_y, labels=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(cv_x_responseCoding)
print('For values of best alpha = ', alpha[best_alpha], "The cross validation␣
 ↪log loss is:",log_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 alpha = 5
Log Loss : 1.1182561222059326
for alpha = 11
Log Loss : 1.0702193942785923
for alpha = 15
Log Loss : 1.0865023453430012
for alpha = 21
```

```
Log Loss : 1.0981244874850429
for alpha = 31
Log Loss : 1.1101471299169436
for alpha = 41
Log Loss : 1.1097465126181116
for alpha = 51
Log Loss : 1.1097123774494124
for alpha = 99
Log Loss : 1.1103337946585057
```



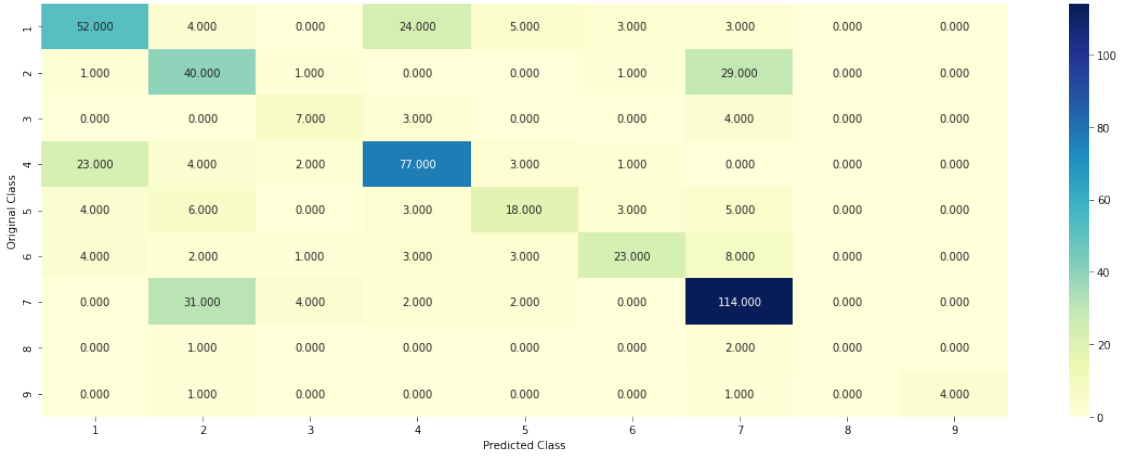

```
For values of best alpha =  11 The train log loss is: 0.5937297449040876
For values of best alpha =  11 The cross validation log loss is:
1.0702193942785923
For values of best alpha =  11 The test log loss is: 1.1026057011636197
```

4.2.2. Testing the model with best hyper paramters

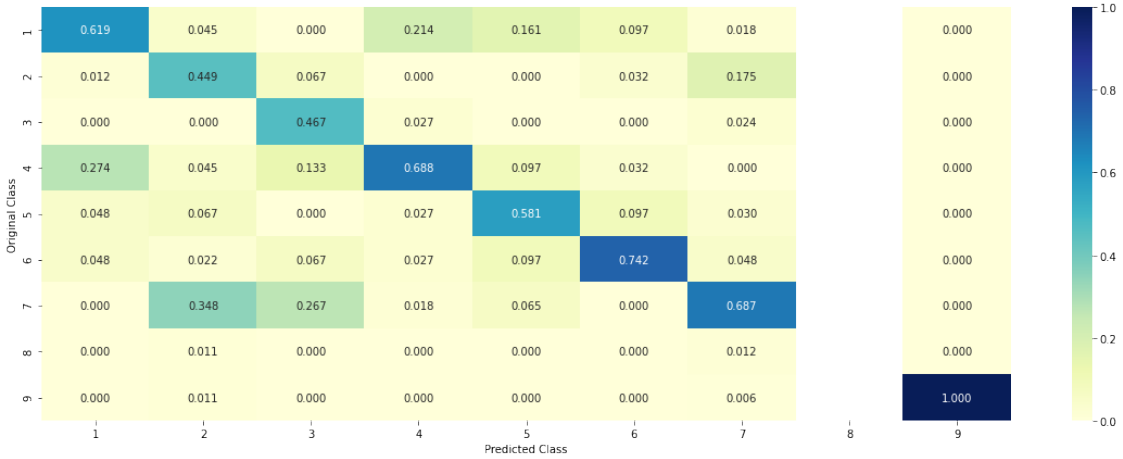
[ ]:
```
# find more about KNeighborsClassifier() here http://scikit-learn.org/stable/
 ↪modules/generated/sklearn.neighbors.KNeighborsClassifier.html
# ------------------------
# default parameter
# KNeighborsClassifier(n_neighbors=5, weights='uniform', algorithm='auto',␣
 ↪leaf_size=30, p=2,
# metric='minkowski',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

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

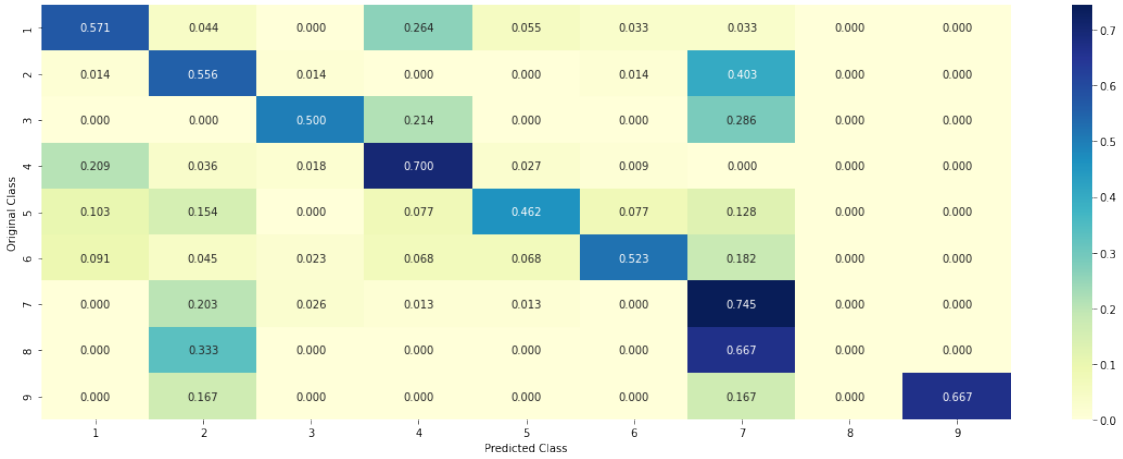
```
Log loss : 1.0702193942785923
Number of mis-classified points : 0.37030075187969924
-------------------- Confusion matrix --------------------
```




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



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



4.2.3.Sample Query point -1

[ ]:
```
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[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 ",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 : 1
The  11  nearest neighbours of the test points belongs to classes [1 1 1 1 1 1 1
1 1 1 1]
Fequency of nearest points : Counter({1: 11})
```

4.2.4. Sample Query Point-2

[ ]:
```
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 : 9
Actual Class : 9
the k value for knn is 11 and the nearest neighbours of the test points belongs
to classes [9 9 9 9 9 9 9 9 8 9 9]
Fequency of nearest points : Counter({9: 10, 8: 1})
```

4.3. Logistic Regression

4.3.1. With Class balancing

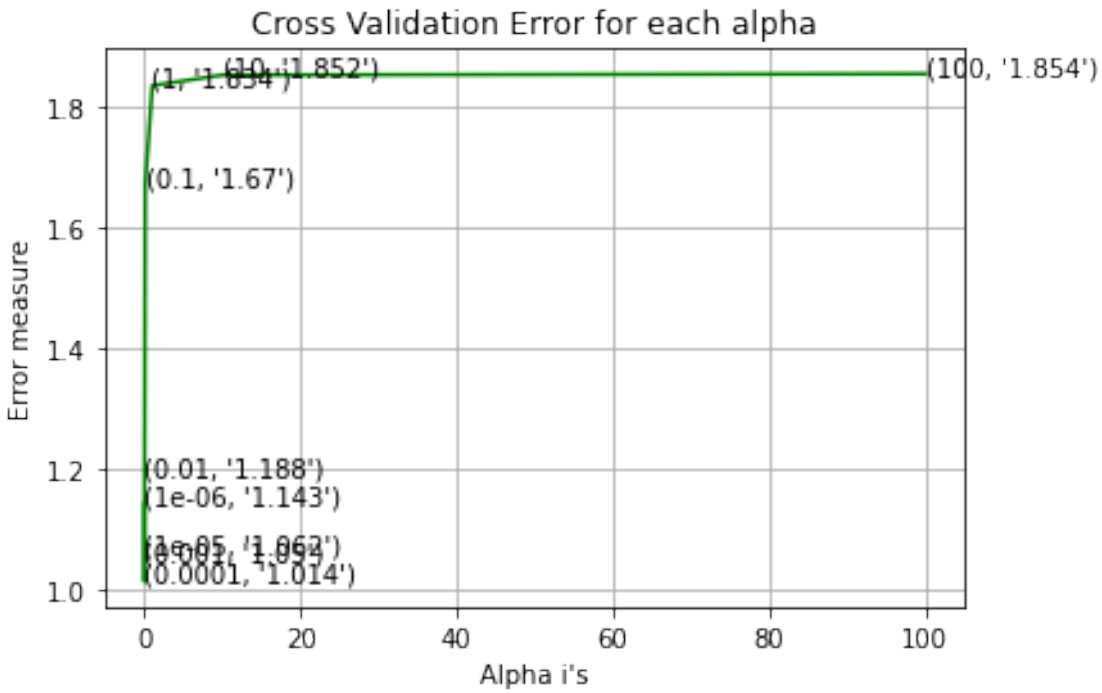
4.3.1.1. Hyper paramter tuning

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

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

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

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

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

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


best_alpha = np.argmin(cv_log_error_array)
clf = SGDClassifier(class_weight='balanced', alpha=alpha[best_alpha],␣
 ↪penalty='l2', loss='log', 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 alpha = 1e-06
Log Loss : 1.1429217562683396
for alpha = 1e-05
Log Loss : 1.062372836973558
for alpha = 0.0001
Log Loss : 1.0137604770733453
for alpha = 0.001
Log Loss : 1.0504856895500743
for alpha = 0.01
Log Loss : 1.1878018475744239
for alpha = 0.1
Log Loss : 1.6695084453961082
for alpha = 1
Log Loss : 1.8342129429870409
for alpha = 10
Log Loss : 1.8517060735871826
for alpha = 100
Log Loss : 1.8537519699432985
```



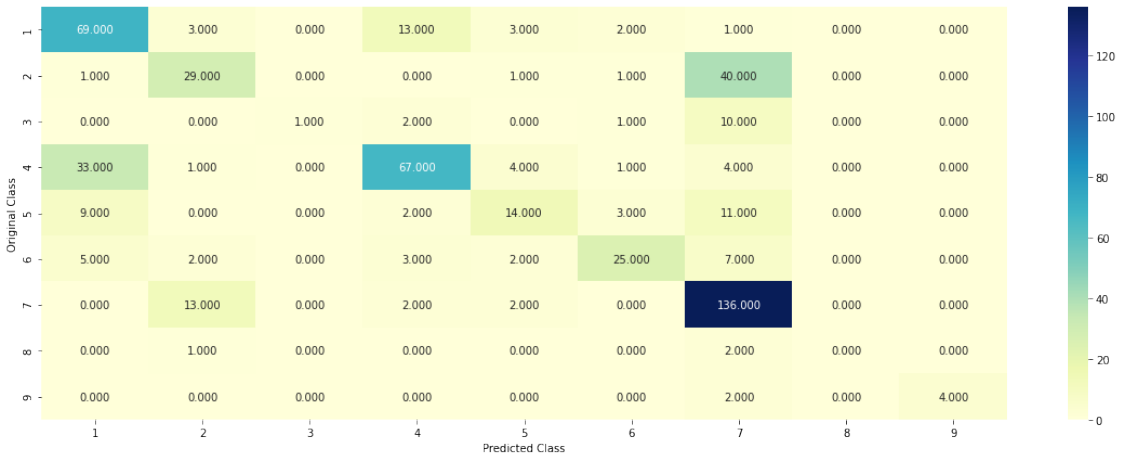

```
For values of best alpha =  0.0001 The train log loss is: 0.39702807145189284
For values of best alpha =  0.0001 The cross validation log loss is:
1.0137604770733453
For values of best alpha =  0.0001 The test log loss is: 1.0518764283278648
```

4.3.1.2. Testing the model with best hyper paramters

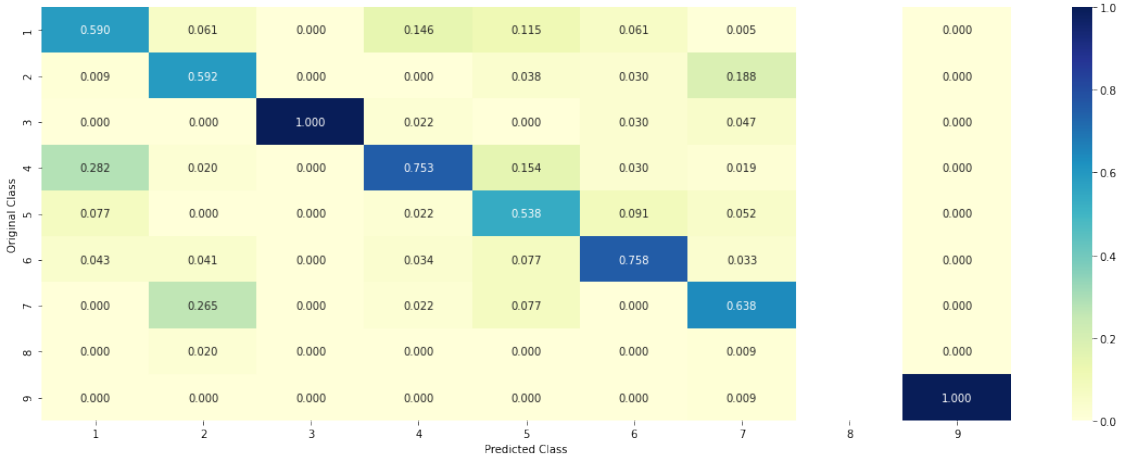
[ ]:
```
# read more about SGDClassifier() at http://scikit-learn.org/stable/modules/
 ↪generated/sklearn.linear_model.SGDClassifier.html
# ------------------------------
# default parameters
# SGDClassifier(loss='hinge', penalty='l2', alpha=0.0001, l1_ratio=0.15,␣
 ↪fit_intercept=True, max_iter=None,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

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

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

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

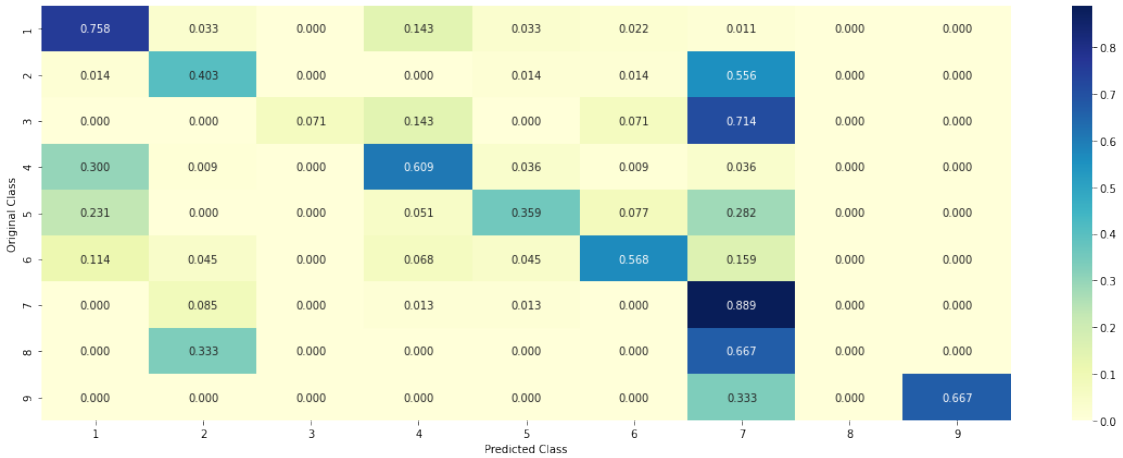
```
Log loss : 1.0137604770733453
Number of mis-classified points : 0.35150375939849626
-------------------- Confusion matrix --------------------
```




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




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



4.3.1.3. Feature Importance

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

4.3.1.3.1. Correctly Classified point

[ ]:
```
# 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(-1*abs(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 : 1
Predicted Class Probabilities: [[7.693e-01 1.878e-01 9.000e-04 1.190e-02
2.400e-03 1.500e-03 2.550e-02
  6.000e-04 2.000e-04]]
Actual Class : 1
--------------------------------------------------
292 Text feature [05] present in test data point [True]
401 Text feature [121] present in test data point [True]
Out of the top  500  features  2 are present in query point
```

4.3.1.3.2. Incorrectly Classified point

[ ]:
```
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(-1*abs(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 : 9
Predicted Class Probabilities: [[0.0534 0.0191 0.0014 0.0148 0.0056 0.0038
0.0208 0.0018 0.8792]]
Actual Class : 9
--------------------------------------------------
47 Text feature [1000] present in test data point [True]
93 Text feature [1010] present in test data point [True]
106 Text feature [10division] present in test data point [True]
```

```
116 Text feature [12q13] present in test data point [True]
180 Text feature [0005] present in test data point [True]
188 Text feature [0026] present in test data point [True]
193 Text feature [105] present in test data point [True]
225 Text feature [113] present in test data point [True]
232 Text feature [10] present in test data point [True]
234 Text feature [117456] present in test data point [True]
351 Text feature [001] present in test data point [True]
374 Text feature [0027] present in test data point [True]
388 Text feature [032] present in test data point [True]
404 Text feature [01] present in test data point [True]
409 Text feature [1177] present in test data point [True]
419 Text feature [108] present in test data point [True]
Out of the top  500  features  16 are present in query point
```

4.3.2. Without Class balancing

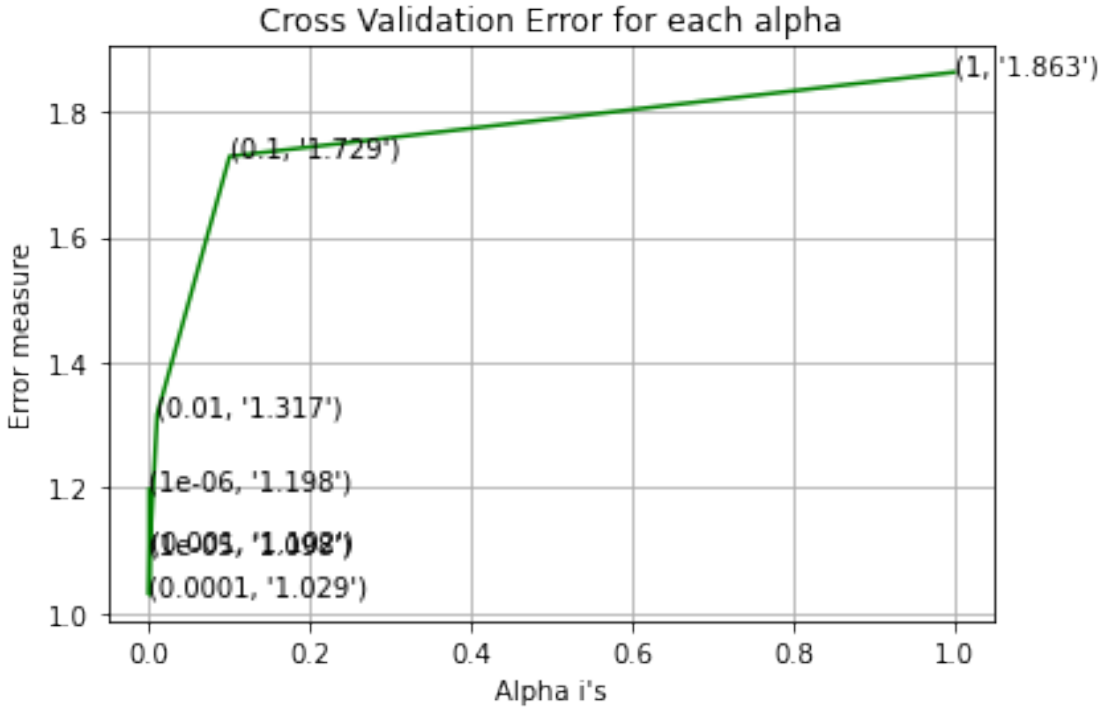
4.3.2.1. Hyper paramter tuning

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

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

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



# find more about CalibratedClassifierCV here at http://scikit-learn.org/stable/
 ↪modules/generated/sklearn.calibration.CalibratedClassifierCV.html
# ----------------------------
# default paramters
# sklearn.calibration.CalibratedClassifierCV(base_estimator=None,␣
 ↪method='sigmoid', cv=3)
#
```

```
# some of the methods of CalibratedClassifierCV()
# fit(X, y[, sample_weight])        Fit the calibrated model
# get_params([deep])        Get parameters for this estimator.
# predict(X)        Predict the target of new samples.
# predict_proba(X)        Posterior probabilities of classification
#--------------------------------------
# video link:
#--------------------------------------

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

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


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

predict_y = sig_clf.predict_proba(train_x_onehotCoding)
print('For values of best alpha = ', alpha[best_alpha], "The train log loss is:
 ↪",log_loss(y_train, predict_y, labels=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(cv_x_onehotCoding)
print('For values of best alpha = ', alpha[best_alpha], "The cross validation␣
 ↪log loss is:",log_loss(y_cv, predict_y, labels=clf.classes_, eps=1e-15))
predict_y = sig_clf.predict_proba(test_x_onehotCoding)
```

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

```
for alpha = 1e-06
Log Loss : 1.197680456163439
for alpha = 1e-05
Log Loss : 1.0978556690381653
for alpha = 0.0001
Log Loss : 1.0290123025403548
for alpha = 0.001
Log Loss : 1.1018219905261084
for alpha = 0.01
Log Loss : 1.3169593258814032
for alpha = 0.1
Log Loss : 1.7290494776201253
for alpha = 1
Log Loss : 1.8632635794998254
```

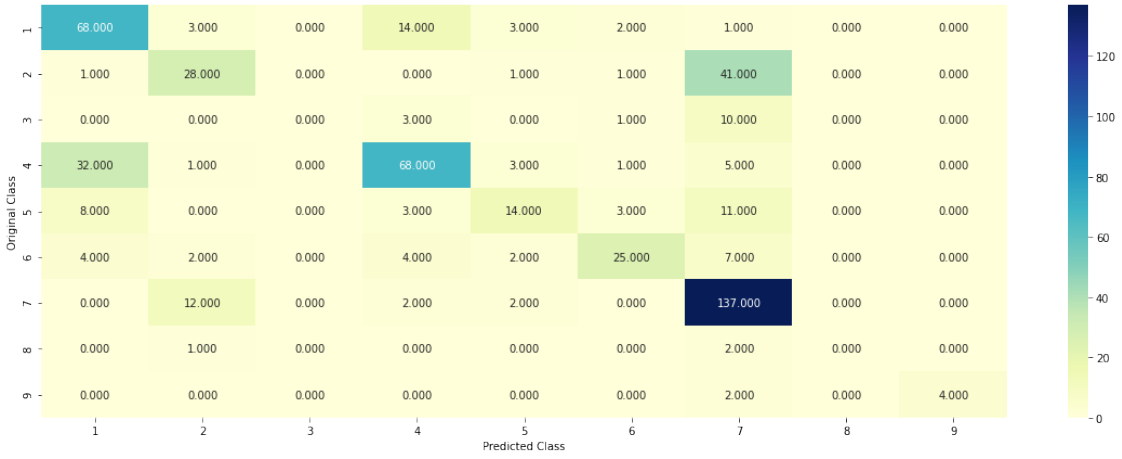



```
For values of best alpha =  0.0001 The train log loss is: 0.38525792542470305
For values of best alpha =  0.0001 The cross validation log loss is:
1.0290123025403548
For values of best alpha =  0.0001 The test log loss is: 1.0718839450366153
```

4.3.2.2. Testing model with best hyper parameters

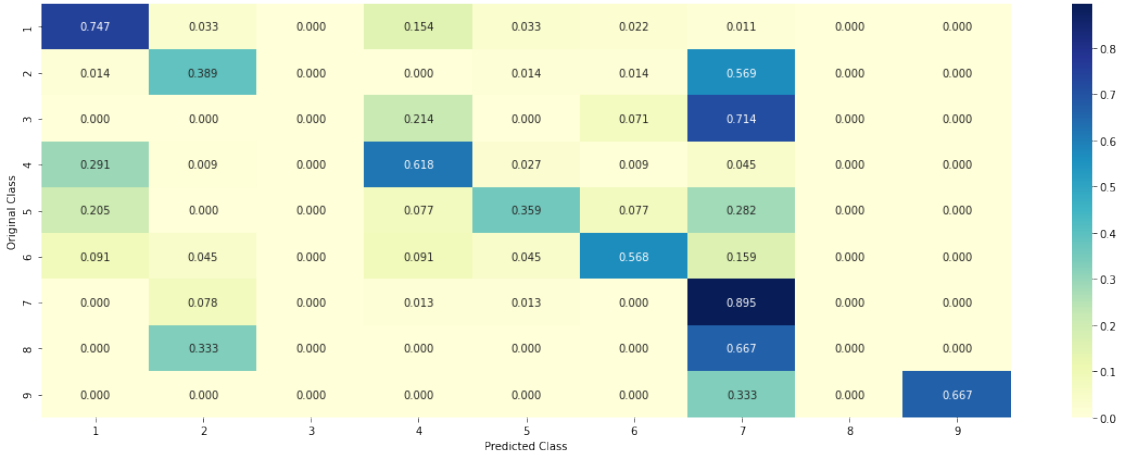
[ ]:
```
# read more about SGDClassifier() at http://scikit-learn.org/stable/modules/
 ↪generated/sklearn.linear_model.SGDClassifier.html
# ------------------------------
# default parameters
# SGDClassifier(loss='hinge', penalty='l2', alpha=0.0001, l1_ratio=0.15,␣
 ↪fit_intercept=True, max_iter=None,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

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

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

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

```
Log loss : 1.0290123025403548
Number of mis-classified points : 0.3533834586466165
-------------------- Confusion matrix --------------------
```




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

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



4.3.2.3. Feature Importance, Correctly Classified point

```
[ ]: 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(-1*abs(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 : 1
Predicted Class Probabilities: [[7.388e-01 2.121e-01 7.000e-04 1.510e-02
1.900e-03 1.400e-03 2.970e-02
  1.000e-04 1.000e-04]]
Actual Class : 1
--------------------------------------------------
285 Text feature [05] present in test data point [True]
359 Text feature [121] present in test data point [True]
Out of the top  500  features  2 are present in query point
```

4.3.2.4. Feature Importance, Inorrectly Classified point

[ ]:
```
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(-1*abs(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 : 9
Predicted Class Probabilities: [[0.0799 0.03   0.0017 0.0238 0.0076 0.0047
0.0385 0.0031 0.8108]]
Actual Class : 9
--------------------------------------------------
44 Text feature [1000] present in test data point [True]
97 Text feature [1010] present in test data point [True]
118 Text feature [113] present in test data point [True]
183 Text feature [10] present in test data point [True]
190 Text feature [032] present in test data point [True]
198 Text feature [0005] present in test data point [True]
200 Text feature [105] present in test data point [True]
249 Text feature [0026] present in test data point [True]
326 Text feature [10division] present in test data point [True]
328 Text feature [108] present in test data point [True]
336 Text feature [117456] present in test data point [True]
344 Text feature [02] present in test data point [True]
```

```
347 Text feature [12q13] present in test data point [True]
369 Text feature [01] present in test data point [True]
393 Text feature [001] present in test data point [True]
399 Text feature [101] present in test data point [True]
413 Text feature [11b] present in test data point [True]
433 Text feature [0027] present in test data point [True]
Out of the top  500  features  18 are present in query point
```

4.4. Linear Support Vector Machines

4.4.1. Hyper paramter tuning

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

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


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


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

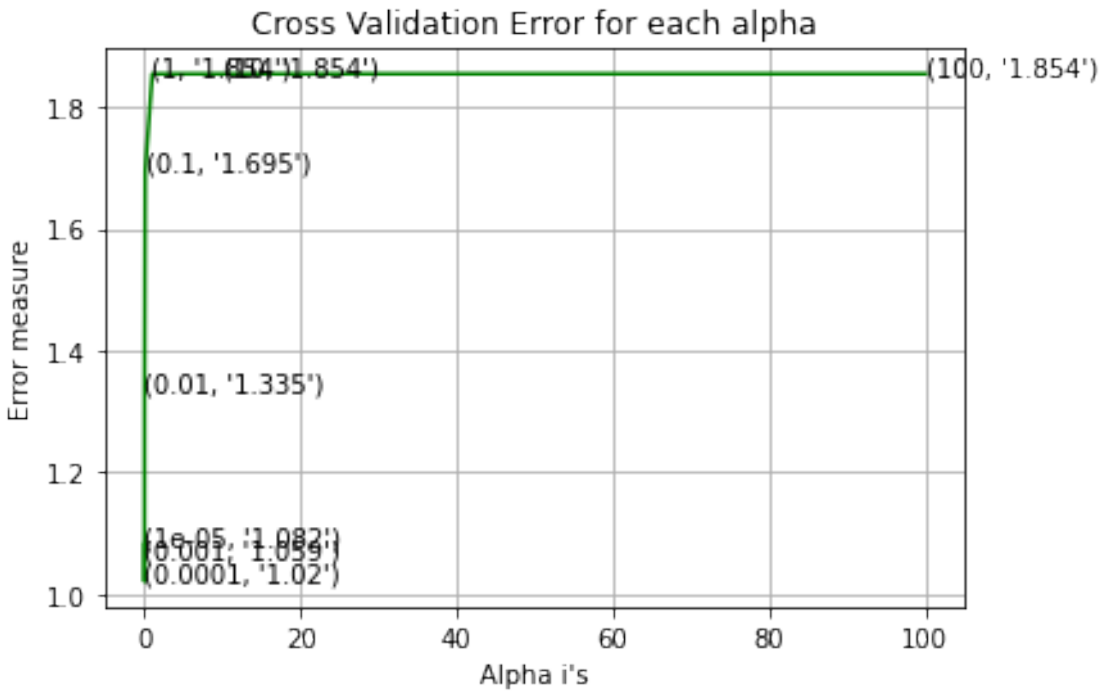
```
cv_log_error_array = []
for i in alpha:
    print("for alpha =", i)
    clf = KNeighborsClassifier(n_neighbors=i)
    clf.fit(train_x_responseCoding, train_y)
    sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
    sig_clf.fit(train_x_responseCoding, train_y)
    sig_clf_probs = sig_clf.predict_proba(cv_x_responseCoding)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

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


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

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

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

# --------------------------------
# default parameters
```

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

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



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

alpha = [10 ** x for x in range(-5, 3)]
cv_log_error_array = []
for i in alpha:
    print("for C =", i)
#     clf = SVC(C=i,kernel='linear',probability=True, class_weight='balanced')
    clf = SGDClassifier( class_weight='balanced', alpha=i, penalty='l2',␣
 ↪loss='hinge', random_state=42)
    clf.fit(train_x_onehotCoding, train_y)
    sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
    sig_clf.fit(train_x_onehotCoding, train_y)
    sig_clf_probs = sig_clf.predict_proba(cv_x_onehotCoding)
    cv_log_error_array.append(log_loss(cv_y, sig_clf_probs, labels=clf.
 ↪classes_, eps=1e-15))
    print("Log Loss :",log_loss(cv_y, sig_clf_probs))
```

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


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

predict_y = sig_clf.predict_proba(train_x_onehotCoding)
print('For values of best alpha = ', alpha[best_alpha], "The train log loss is:
 ↪",log_loss(y_train, predict_y, labels=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 C = 1e-05
Log Loss : 1.0815542001002818
for C = 0.0001
Log Loss : 1.0202566318521267
for C = 0.001
Log Loss : 1.0591312534862374
for C = 0.01
Log Loss : 1.3352352766190438
for C = 0.1
Log Loss : 1.6948938036813415
for C = 1
Log Loss : 1.8543729993880769
for C = 10
Log Loss : 1.8543626403036142
for C = 100
Log Loss : 1.8543631979514978
```



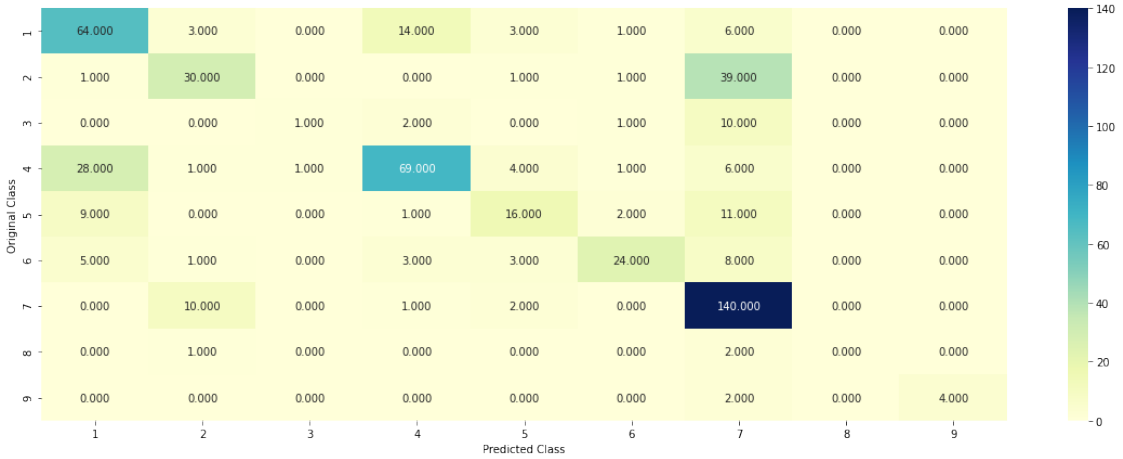

```
For values of best alpha =  0.0001 The train log loss is: 0.322238197884991
For values of best alpha =  0.0001 The cross validation log loss is:
1.0202566318521267
For values of best alpha =  0.0001 The test log loss is: 1.0578591107924888
```

4.4.2. Testing model with best hyper parameters

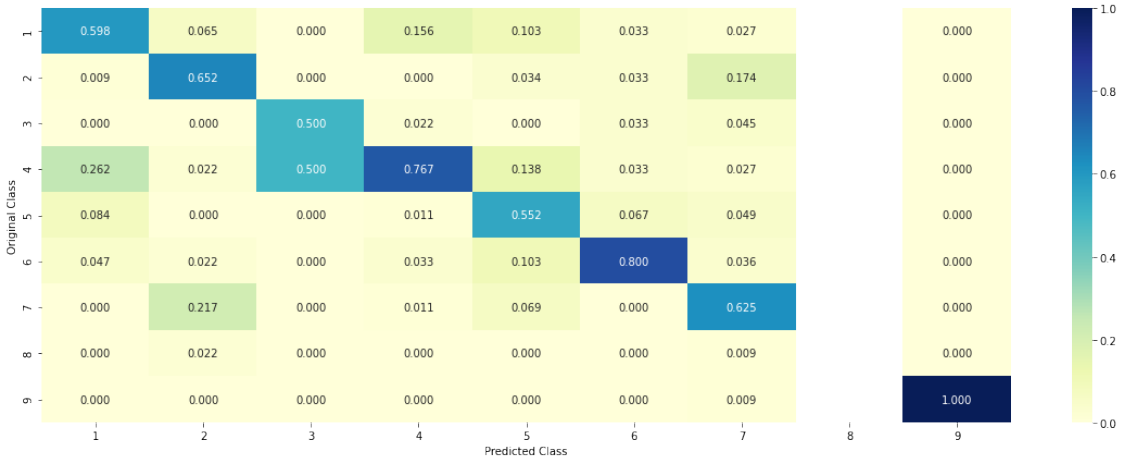
[ ]:
```
# read more about support vector machines with linear kernals here http://
 ↪scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

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

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

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

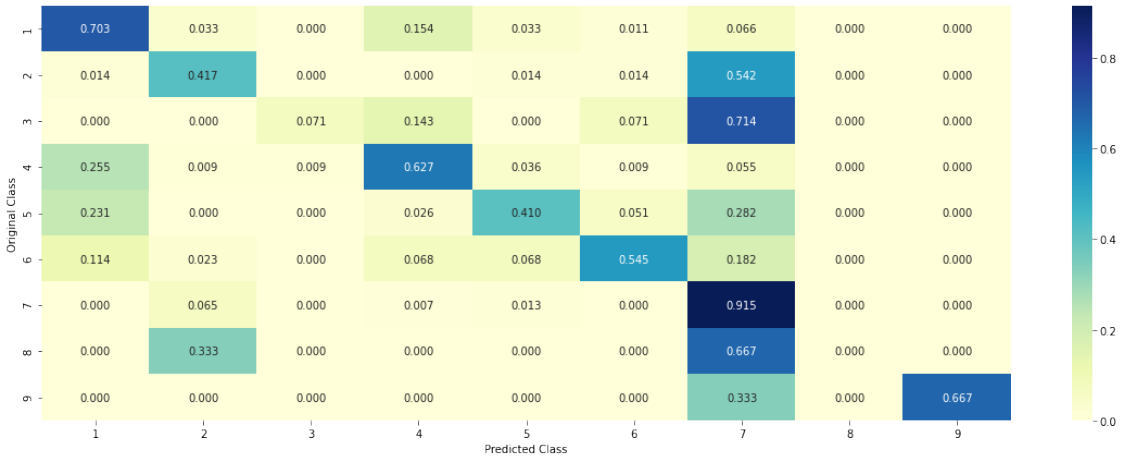
```
Log loss : 1.0202566318521267
Number of mis-classified points : 0.3458646616541353
-------------------- Confusion matrix --------------------
```




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




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



4.3.3. Feature Importance

4.3.3.1. For Correctly classified point

[ ]:
```
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(-1*abs(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 : 1
Predicted Class Probabilities: [[5.143e-01 3.376e-01 2.500e-03 6.720e-02
5.800e-03 2.700e-03 6.870e-02
  5.000e-04 8.000e-04]]
Actual Class : 1
--------------------------------------------------
317 Text feature [05] present in test data point [True]
469 Text feature [121] present in test data point [True]
Out of the top  500  features  2 are present in query point
```

4.3.3.2. For Incorrectly classified point

[ ]:
```
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(-1*abs(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 : 9
Predicted Class Probabilities: [[0.0995 0.0448 0.0059 0.0537 0.0335 0.0128
0.0617 0.0024 0.6857]]
Actual Class : 9
--------------------------------------------------
111 Text feature [1010] present in test data point [True]
116 Text feature [1000] present in test data point [True]
166 Text feature [10division] present in test data point [True]
176 Text feature [105] present in test data point [True]
204 Text feature [117456] present in test data point [True]
217 Text feature [108] present in test data point [True]
320 Text feature [101] present in test data point [True]
365 Text feature [01] present in test data point [True]
391 Text feature [113] present in test data point [True]
407 Text feature [0026] present in test data point [True]
Out of the top  500  features  10 are present in query point
```

4.5 Random Forest Classifier

4.5.1. Hyper paramter tuning (With One hot Encoding)

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

# Some of methods of RandomForestClassifier()
# fit(X, y, [sample_weight])        Fit the SVM model according to the given␣
 ↪training data.
```

```
# predict(X)        Perform classification on samples in X.
# predict_proba (X)        Perform classification on samples in X.

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

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


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

alpha = [100,200,500,1000,2000]
max_depth = [5, 10]
cv_log_error_array = []
for i in alpha:
    for j in max_depth:
        print("for n_estimators =", i,"and max depth = ", j)
        clf = RandomForestClassifier(n_estimators=i, criterion='gini',␣
 ↪max_depth=j, random_state=42, n_jobs=-1)
        clf.fit(train_x_onehotCoding, train_y)
        sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
        sig_clf.fit(train_x_onehotCoding, train_y)
        sig_clf_probs = sig_clf.predict_proba(cv_x_onehotCoding)
        cv_log_error_array.append(log_loss(cv_y, sig_clf_probs, labels=clf.
 ↪classes_, eps=1e-15))
        print("Log Loss :",log_loss(cv_y, sig_clf_probs))

'''fig, ax = plt.subplots()
features = np.dot(np.array(alpha)[:,None],np.array(max_depth)[None]).ravel()
```

```
ax.plot(features, cv_log_error_array,c='g')
for i, txt in enumerate(np.round(cv_log_error_array,3)):
    ax.annotate((alpha[int(i/2)],max_depth[int(i%2)],str(txt)),␣
 ↪(features[i],cv_log_error_array[i]))
plt.grid()
plt.title("Cross Validation Error for each alpha")
plt.xlabel("Alpha i's")
plt.ylabel("Error measure")
plt.show()
'''

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

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

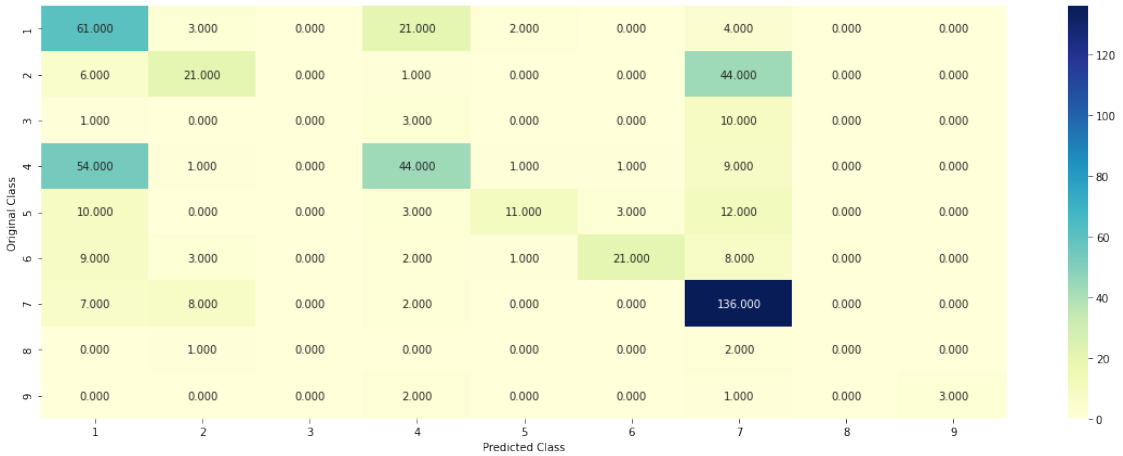
```
for n_estimators = 100 and max depth =  5
Log Loss : 1.2135988548483725
for n_estimators = 100 and max depth =  10
Log Loss : 1.224981713375827
for n_estimators = 200 and max depth =  5
Log Loss : 1.2014257595304836
for n_estimators = 200 and max depth =  10
Log Loss : 1.2171631082837233
for n_estimators = 500 and max depth =  5
Log Loss : 1.1925545565853508
for n_estimators = 500 and max depth =  10
Log Loss : 1.2073876872442846
for n_estimators = 1000 and max depth =  5
Log Loss : 1.1890709830907116
for n_estimators = 1000 and max depth =  10
Log Loss : 1.2061500131328826
for n_estimators = 2000 and max depth =  5
```

```
Log Loss : 1.186390819263271
for n_estimators = 2000 and max depth =  10
Log Loss : 1.2048246062613748
For values of best estimator =  2000 The train log loss is: 0.8597450010676068
For values of best estimator =  2000 The cross validation log loss is:
1.1863908192632708
For values of best estimator =  2000 The test log loss is: 1.1861404463913465
```

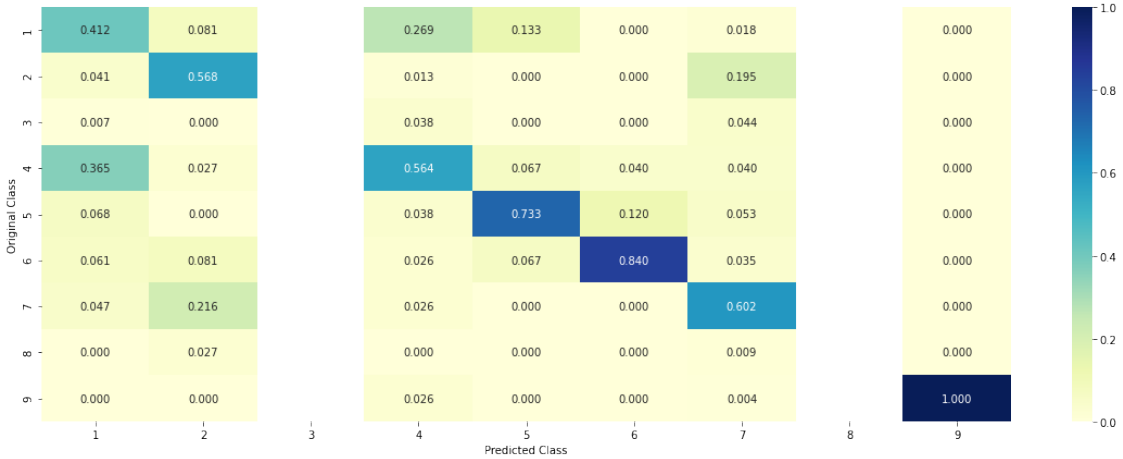
4.5.2. Testing model with best hyper parameters (One Hot Encoding)

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

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

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

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

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

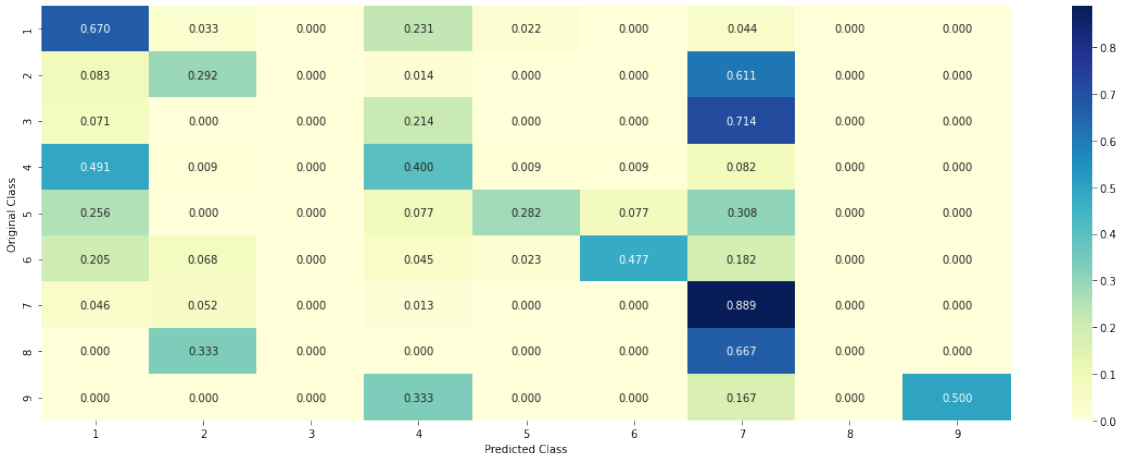
```
Log loss : 1.1863908192632708
Number of mis-classified points : 0.4417293233082707
-------------------- Confusion matrix --------------------
```



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



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



4.5.3. Feature Importance

4.5.3.1. Correctly Classified point

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

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

```
Predicted Class : 1
Predicted Class Probabilities: [[0.6437 0.0607 0.0126 0.0914 0.0466 0.0446
0.0499 0.0155 0.0349]]
Actual Class : 1
--------------------------------------------------
66 Text feature [11] present in test data point [True]
```

```
Out of the top  100  features  1 are present in query point
```

4.5.3.2. Incorrectly Classified point

[ ]:
```
test_point_index = 100
no_feature = 100
predicted_cls = sig_clf.predict(test_x_onehotCoding[test_point_index])
print("Predicted Class :", predicted_cls[0])
print("Predicted Class Probabilities:", np.round(sig_clf.
 ↪predict_proba(test_x_onehotCoding[test_point_index]),4))
print("Actuall Class :", test_y[test_point_index])
indices = np.argsort(-clf.feature_importances_)
print("-"*50)
get_impfeature_names(indices[:no_feature], test_df['TEXT'].
 ↪iloc[test_point_index],test_df['Gene'].
 ↪iloc[test_point_index],test_df['Variation'].iloc[test_point_index],␣
 ↪no_feature)
```

```
Predicted Class : 9
Predicted Class Probabilities: [[0.3241 0.0207 0.0067 0.0923 0.0296 0.025
0.0304 0.0181 0.4532]]
Actuall Class : 9
--------------------------------------------------
66 Text feature [11] present in test data point [True]
Out of the top  100  features  1 are present in query point
```

4.5.3. Hyper paramter tuning (With Response Coding)

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

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

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

# ------------------------------
```

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


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

alpha = [10,50,100,200,500,1000]
max_depth = [2,3,5,10]
cv_log_error_array = []
for i in alpha:
    for j in max_depth:
        print("for n_estimators =", i,"and max depth = ", j)
        clf = RandomForestClassifier(n_estimators=i, criterion='gini',␣
 ↪max_depth=j, random_state=42, n_jobs=-1)
        clf.fit(train_x_responseCoding, train_y)
        sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
        sig_clf.fit(train_x_responseCoding, train_y)
        sig_clf_probs = sig_clf.predict_proba(cv_x_responseCoding)
        cv_log_error_array.append(log_loss(cv_y, sig_clf_probs, labels=clf.
 ↪classes_, eps=1e-15))
        print("Log Loss :",log_loss(cv_y, sig_clf_probs))
'''
fig, ax = plt.subplots()
features = np.dot(np.array(alpha)[:,None],np.array(max_depth)[None]).ravel()
ax.plot(features, cv_log_error_array,c='g')
for i, txt in enumerate(np.round(cv_log_error_array,3)):
    ax.annotate((alpha[int(i/4)],max_depth[int(i%4)],str(txt)),␣
 ↪(features[i],cv_log_error_array[i]))
plt.grid()
plt.title("Cross Validation Error for each alpha")
plt.xlabel("Alpha i's")
```

```
plt.ylabel("Error measure")
plt.show()
'''

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

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

```
for n_estimators = 10 and max depth =  2
Log Loss : 2.1944745238871306
for n_estimators = 10 and max depth =  3
Log Loss : 1.7152750455136767
for n_estimators = 10 and max depth =  5
Log Loss : 1.483173592129834
for n_estimators = 10 and max depth =  10
Log Loss : 1.6138977698883985
for n_estimators = 50 and max depth =  2
Log Loss : 1.6685711618009886
for n_estimators = 50 and max depth =  3
Log Loss : 1.4306900392310882
for n_estimators = 50 and max depth =  5
Log Loss : 1.5042495687378523
for n_estimators = 50 and max depth =  10
Log Loss : 1.6196822997653033
for n_estimators = 100 and max depth =  2
Log Loss : 1.5522501466881198
for n_estimators = 100 and max depth =  3
Log Loss : 1.482446525987865
for n_estimators = 100 and max depth =  5
Log Loss : 1.3639106109608081
for n_estimators = 100 and max depth =  10
Log Loss : 1.666229430047524
```

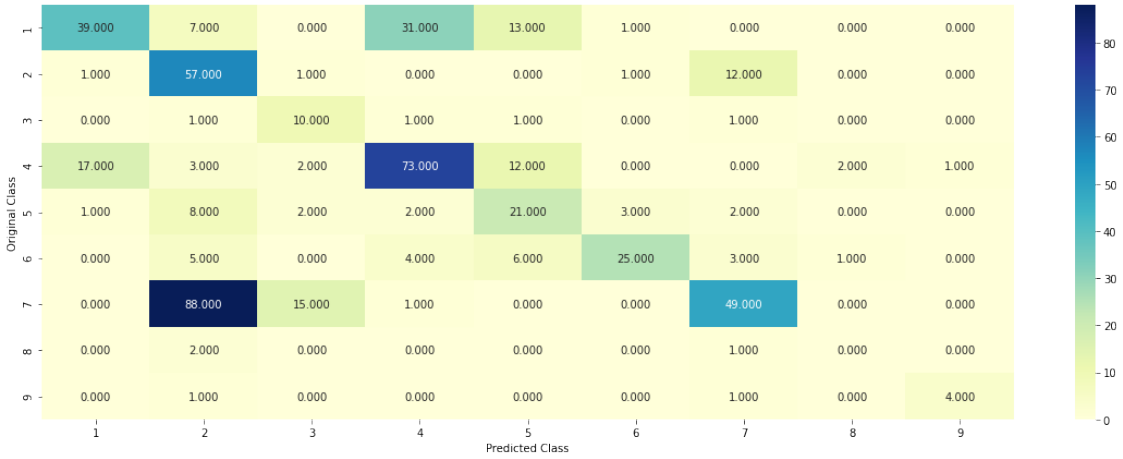
```
for n_estimators = 200 and max depth =  2
Log Loss : 1.6132804984288172
for n_estimators = 200 and max depth =  3
Log Loss : 1.4996520718123783
for n_estimators = 200 and max depth =  5
Log Loss : 1.3888803094075393
for n_estimators = 200 and max depth =  10
Log Loss : 1.6345210926456455
for n_estimators = 500 and max depth =  2
Log Loss : 1.6967876608954047
for n_estimators = 500 and max depth =  3
Log Loss : 1.5740559171169786
for n_estimators = 500 and max depth =  5
Log Loss : 1.3961240860348987
for n_estimators = 500 and max depth =  10
Log Loss : 1.6886712795861412
for n_estimators = 1000 and max depth =  2
Log Loss : 1.6742589269703652
for n_estimators = 1000 and max depth =  3
Log Loss : 1.5684128362394942
for n_estimators = 1000 and max depth =  5
Log Loss : 1.3799537093390601
for n_estimators = 1000 and max depth =  10
Log Loss : 1.6855943587384967
For values of best alpha =  100 The train log loss is: 0.0647860538149674
For values of best alpha =  100 The cross validation log loss is:
1.363910610960808
For values of best alpha =  100 The test log loss is: 1.3909830006365447
```

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

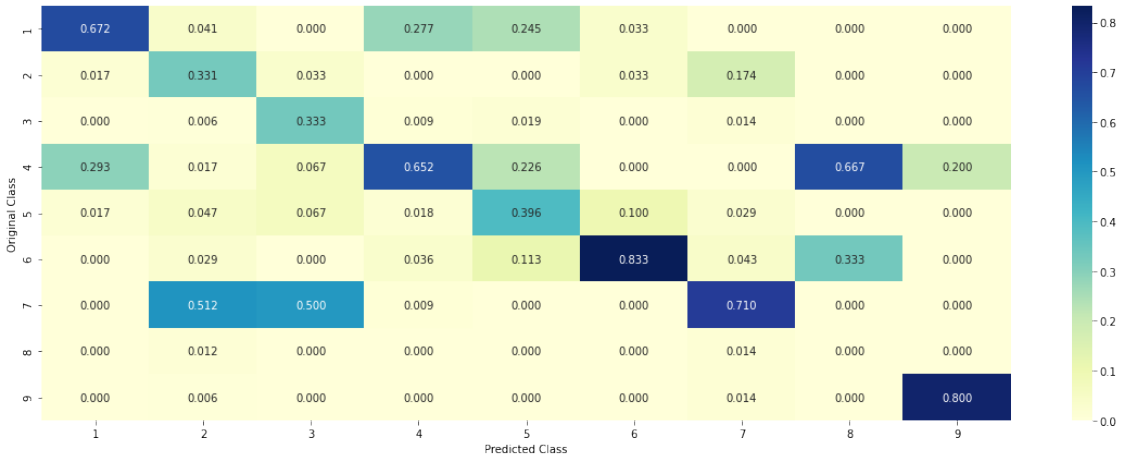
[ ]:
```
# ------------------------------
# default parameters
# sklearn.ensemble.RandomForestClassifier(n_estimators=10, criterion='gini',␣
 ↪max_depth=None, min_samples_split=2,
# min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features='auto',␣
 ↪max_leaf_nodes=None, min_impurity_decrease=0.0,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

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

# some of attributes of  RandomForestClassifier()
```

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

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

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

```
Log loss : 1.3639106109608081
Number of mis-classified points : 0.4774436090225564
-------------------- Confusion matrix --------------------
```




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



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

4.5.5. Feature Importance

4.5.5.1. Correctly Classified point

[ ]:
```
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 : 1
Predicted Class Probabilities: [[0.4331 0.1157 0.0405 0.0472 0.0399 0.0542
0.0194 0.1541 0.0959]]
Actual Class : 1
--------------------------------------------------
Variation is important feature
Variation is important feature
Variation is important feature
Variation is important feature
Variation is important feature
Variation is important feature
Text is important feature
Gene is important feature
Text is important feature
Text is important feature
Gene is important feature
Text is important feature
Text is important feature
Gene is important feature
Variation is important feature
Gene is important feature
Text is important feature
Gene is important feature
Gene is important feature
Variation is important feature
Gene is important feature
Text is important feature
Text is important feature
Variation is important feature
Text is important feature
Gene is important feature
```

```
Gene is important feature
```

4.5.5.2. Incorrectly Classified point

[ ]:
```
test_point_index = 100
predicted_cls = sig_clf.predict(test_x_responseCoding[test_point_index].
 ↪reshape(1,-1))
print("Predicted Class :", predicted_cls[0])
print("Predicted Class Probabilities:", np.round(sig_clf.
 ↪predict_proba(test_x_responseCoding[test_point_index].reshape(1,-1)),4))
print("Actual Class :", test_y[test_point_index])
indices = np.argsort(-clf.feature_importances_)
print("-"*50)
for i in indices:
    if i<9:
        print("Gene is important feature")
    elif i<18:
        print("Variation is important feature")
    else:
        print("Text is important feature")
```

```
Predicted Class : 9
Predicted Class Probabilities: [[0.0204 0.0199 0.0144 0.0129 0.0112 0.0178
0.0072 0.1826 0.7136]]
Actual Class : 9
--------------------------------------------------
Variation is important feature
Variation is important feature
Variation is important feature
Variation is important feature
Variation is important feature
Variation is important feature
Text is important feature
Gene is important feature
Text is important feature
Text is important feature
Gene is important feature
Text is important feature
Text is important feature
Gene is important feature
Variation is important feature
Gene is important feature
Text is important feature
Gene is important feature
Gene is important feature
Variation is important feature
Gene is important feature
Text is important feature
```

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

4.7 Stack the models

4.7.1 testing with hyper parameter tuning

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

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

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


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

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

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

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

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

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


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

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


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

```
sig_clf1.fit(train_x_onehotCoding, train_y)
print("Logistic Regression :  Log Loss: %0.2f" % (log_loss(cv_y, sig_clf1.
 ↪predict_proba(cv_x_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
```

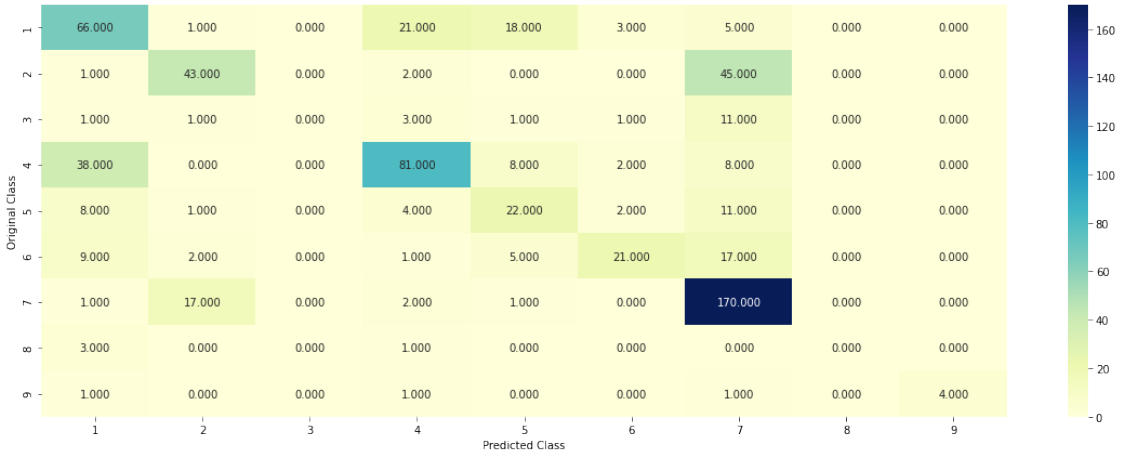
```
Logistic Regression :  Log Loss: 1.05
Support vector machines : Log Loss: 1.85
Naive Bayes : Log Loss: 1.18
--------------------------------------------------
Stacking Classifer : for the value of alpha: 0.000100 Log Loss: 1.817
Stacking Classifer : for the value of alpha: 0.001000 Log Loss: 1.711
Stacking Classifer : for the value of alpha: 0.010000 Log Loss: 1.324
Stacking Classifer : for the value of alpha: 0.100000 Log Loss: 1.261
Stacking Classifer : for the value of alpha: 1.000000 Log Loss: 1.592
Stacking Classifer : for the value of alpha: 10.000000 Log Loss: 1.978
```

4.7.2 testing the model with the best hyper parameters

[ ]:
```
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))
```

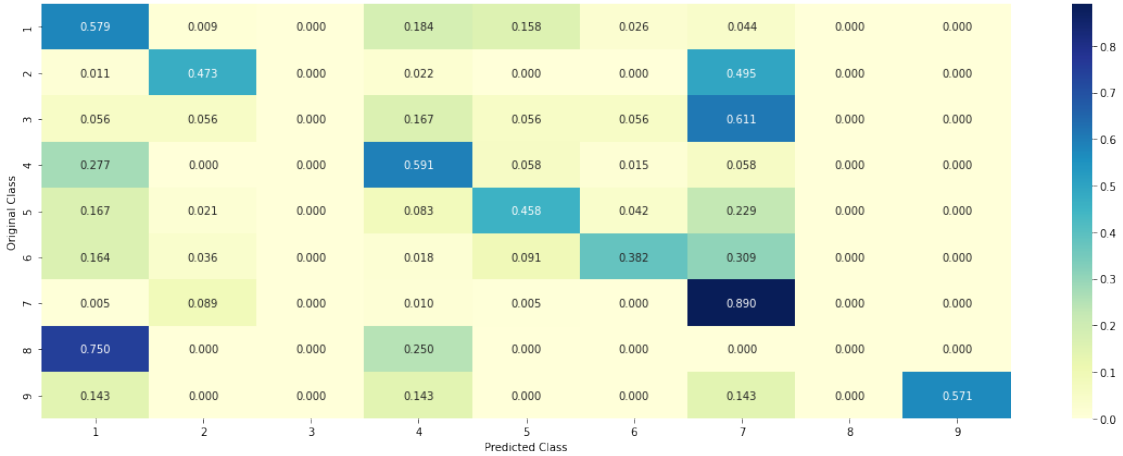
```
print("Log loss (test) on the stacking classifier :",log_error)

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 : 0.33709944840710554
Log loss (CV) on the stacking classifier : 1.2611882520861883
Log loss (test) on the stacking classifier : 1.2998271366490541
Number of missclassified point : 0.3879699248120301
-------------------- Confusion matrix --------------------
```




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

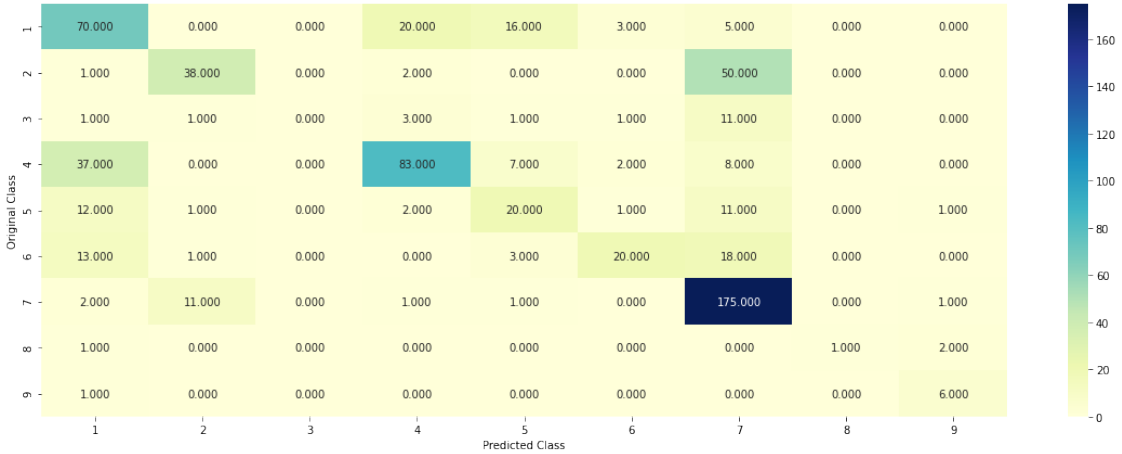

```
-------------------- Recall matrix (Row sum=1) --------------------
```



4.7.3 Maximum Voting classifier

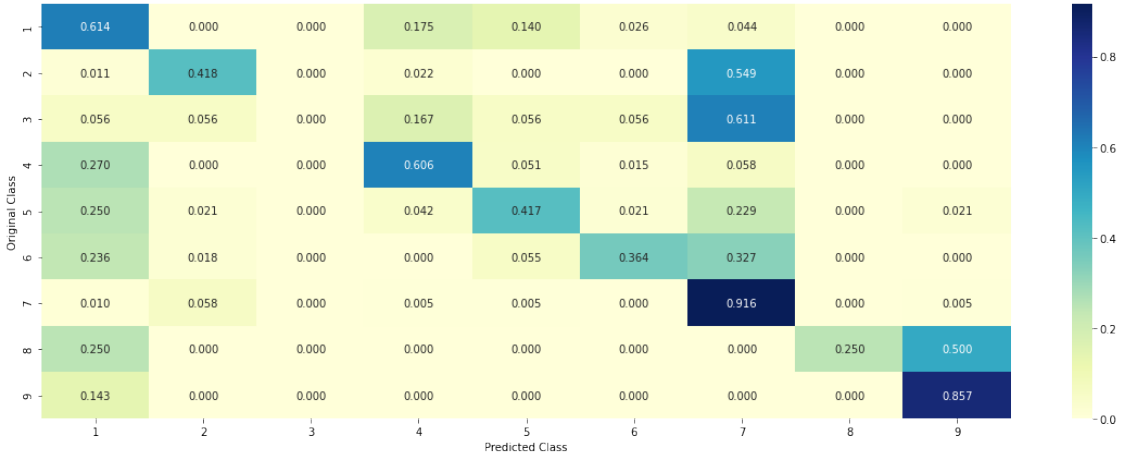
[ ]:
```
#Refer:http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.
 ↪VotingClassifier.html
from sklearn.ensemble import VotingClassifier
vclf = VotingClassifier(estimators=[('lr', sig_clf1), ('svc', sig_clf2), ('rf',␣
 ↪sig_clf3)], voting='soft')
vclf.fit(train_x_onehotCoding, train_y)
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 : 0.7789484750104647
Log loss (CV) on the VotingClassifier : 1.2036591142911721
Log loss (test) on the VotingClassifier : 1.2322549560846812
Number of missclassified point : 0.37894736842105264
-------------------- Confusion matrix --------------------
```



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

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



Tabular representation(BOW)

[ ]:
```
# https://zetcode.com/python/prettytable/

from prettytable import PrettyTable

x = PrettyTable()

x.field_names = ["Model", "Train_Log-loss", "Val_Log-loss", "Test_Los-loss",␣
 ↪'Misclassified points(%)']

x.add_row(["Multinomial-NB", 0.85, 1.27, 1.32, 0.40])
x.add_row(["KNN(Response Coding)", 0.60, 1.07, 1.10, 0.37])
x.add_row(["Logistic Reg.", 0.53, 1.07, 1.08, 0.36])
x.add_row(["Linear-SVM", 0.54, 1.10, 1.16, 0.36])
x.add_row(["Random Forest", 0.69, 1.15, 1.15, 0.40])
x.add_row(["Stacking", 0.51, 1.20, 1.19, 0.38])
x.add_row(["Voting", 0.87, 1.20, 1.23, .39])

print(x)
```

```
+---------------------+---------------+-------------+--------------+--------
-----------------+
|        Model        | Train_Log-loss | Val_Log-loss | Test_Los-loss |
Misclassified points(%) |
+---------------------+---------------+-------------+--------------+--------
-----------------+
|    Multinomial-NB   |      0.85      |     1.27     |      1.32     |
0.4           |
| KNN(Response Coding) |      0.6       |     1.07     |      1.1      |
0.37          |
|    Logistic Reg.    |      0.53      |     1.07     |      1.08     |
```

```
0.36          |
|      Linear-SVM      |      0.54      |     1.1      |      1.16     |
0.36          |
|    Random Forest     |      0.69      |     1.15     |      1.15     |
0.4           |
|       Stacking       |      0.51      |     1.2      |      1.19     |
0.38          |
|        Voting        |      0.87      |     1.2      |      1.23     |
0.39          |
+----------------------+----------------+--------------+---------------+--------
----------------+
```

2 5. Assignments

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

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

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

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

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

[ ]:
```
# building a TfidfVectorizer with all the words that occured minimum 3 times in␣
 ↪train data
text_vectorizer = TfidfVectorizer(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 : 54185
```

[ ]:
```
# 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)
```

[ ]:
```
x = PrettyTable()

x.field_names = ["Model", "Train_Log-loss", "Val_Log-loss", "Test_Los-loss",␣
 ↪'Misclassified points(%)']

x.add_row(["Multinomial-NB", 0.92, 1.21, 1.22, 0.43])
x.add_row(["Logistic Reg.", 0.48, 1.04, 1.06, 0.35])
x.add_row(["Linear-SVM", 0.52, 1.07, 1.14, 0.36])
x.add_row(["Random Forest", 0.63, 1.11, 1.13, 0.39])
x.add_row(["Stacking", 0.48, 1.20, 1.20, 0.37])
x.add_row(["Voting", 0.80, 1.13, 1.16, .38])

print(x)
```

```
+---------------+---------------+-------------+--------------+-------------
-----------+
|     Model     | Train_Log-loss | Val_Log-loss | Test_Los-loss | Misclassified
points(%) |
+---------------+---------------+-------------+--------------+-------------
-----------+
| Multinomial-NB |      0.92      |     1.21     |      1.22     |
0.43          |
| Logistic Reg.  |      0.48      |     1.04     |      1.06     |
0.35          |
|   Linear-SVM   |      0.52      |     1.07     |      1.14     |
0.36          |
| Random Forest  |      0.63      |     1.11     |      1.13     |
0.39          |
|    Stacking    |      0.48      |     1.2      |      1.2      |
0.37          |
|     Voting     |      0.8       |     1.13     |      1.16     |
0.38          |
+---------------+---------------+-------------+--------------+-------------
```

```
-----------+
```

2.2 Task 2 : Use top 1000 words based on tf-idf values

[ ]:
```
# building a TfidfVectorizer with all the words that occured minimum 3 times in␣
 ↪train data
n_top = 1000

text_vectorizer = TfidfVectorizer(min_df=3, max_features= n_top)
train_text_feature_onehotCoding = text_vectorizer.
 ↪fit_transform(train_df['TEXT'])

# 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

# getting the top 1000 feature names (words)
importance = np.argsort(np.asarray(train_text_feature_onehotCoding.sum(axis=0)).
 ↪ravel())[::-1]
tfidf_feature_names = np.array(text_vectorizer.get_feature_names())  ## ref.␣
 ↪link.: https://stackoverflow.com/questions/34232190/
 ↪scikit-learn-tfidfvectorizer-how-to-get-top-n-terms-with-highest-tf-idf-score
train_text_features = tfidf_feature_names[importance[:n_top]]

# 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))
print("Top 1000 features: ", text_fea_dict)
```

```
Total number of unique words in train data : 1000
Top 1000 features:  {'mutations': 12.671711410305852, 'cells':
14.431994126457067, 'fig': 9.294966648931013, 'cell': 8.661524705120554,
'brca1': 56.15317720334837, 'al': 21.942417410786348, 'et': 29.85088388210208,
'mutation': 31.45550347860939, 'figure': 24.611154545456674, 'variants':
26.661598095884184, 'cancer': 29.709219158400245, 'protein': 22.45384373997253,
'patients': 20.783330507566003, 'pten': 22.002566315618004, 'mutant':
19.743087185146997, 'p53': 8.916265435184362, 'expression': 10.946748680459804,
'activity': 10.443107922165916, 'tumor': 9.403329216329634, 'egfr':
10.383568061059114, 'kinase': 9.156983590280133, 'type': 31.843780770233224,
'domain': 7.792257740879259, 'gene': 9.311702239817016, 'dna':
9.664583402389857, 'using': 9.181735622858485, 'tumors': 10.021866640645554,
'mutants': 12.049937575639953, 'also': 10.95235952845175, 'data':
14.47020346146112, 'binding': 11.781477626364726, 'analysis':
11.710383155593554, 'wild': 11.154710136611172, 'table': 13.489240746764832,
'ras': 16.116452798851263, 'supplementary': 13.96601844485121, 'two':
13.319345376012, '10': 19.4020803941856, 'genes': 16.784546928789528, 'braf':
```

14.695346783688171, 'activation': 21.286379616370294, 'exon':
19.529091324536655, 'may': 14.587727089080998, 'kit': 12.393920400418763,
'identified': 12.77199185089032, 'wt': 13.2233751623155, 'vus':
10.619703241519892, 'results': 8.954554152716227, 'alk': 23.1846469228673,
'shown': 11.657178408448106, 'associated': 12.05770792916538, 'imatinib':
12.398436582674039, 'brct': 10.60532094085181, 'proteins': 12.816318926717678,
'deleterious': 10.580033680037015, 'function': 14.014410798866594, 'one':
9.047238562353735, 'raf': 10.388287003052508, 'signaling': 8.262894437541945,
'growth': 7.185629391301742, 'human': 15.159478914757397, 'variant':
15.833950115866513, 'used': 9.108225242734996, 'found': 10.057240407784798,
'lines': 8.295821657057678, 'cases': 7.742614943824458, 'clinical':
9.523494090943464, 'missense': 8.95940127204699, 'functional':
7.901238544529751, 'fusion': 10.84006675306378, 'pdgfra': 8.979531390228635,
'phosphorylation': 7.038638580797785, 'brca2': 24.104924990379608, 'study':
7.411807782696886, 'residues': 7.833201591452178, 'breast': 7.0048675365405,
'resistance': 8.60520176215801, 'melanoma': 9.180828244781095, 'samples':
8.190721966329903, 'observed': 11.818195534631377, 'sequence':
7.415004448484432, 'patient': 8.546638179290547, 'receptor': 9.331388206084538,
'levels': 7.481723288392417, 'three': 8.582751144738346, 'mice':
8.459068796423887, 'assay': 12.08685347011031, 'akt': 8.74850871495807, 'p110':
19.646962918180883, 'pcr': 17.265439077844075, 'sequencing': 8.08238269022353,
'family': 14.07746320569022, 'treatment': 14.368945553529764, 'studies':
23.907000228878935, 'structure': 14.588702825574979, 'showed':
9.064288058269018, 'pathway': 9.817270962925578, 'amino': 10.812092408260936,
'specific': 18.933651450116127, 'number': 22.22061794938157, 'mtor':
50.360458599406705, '20': 20.93534098817803, 'fgfr2': 12.89698595781551,
'previously': 78.78309824595594, 'erbb2': 7.551826154156001, '12':
19.90603695793107, 'reported': 18.884569689602827, 'well': 10.649943001993188,
'different': 6.8684057111947725, 'expressing': 13.732826299138, 'performed':
15.446449287074502, 'assays': 8.44446954186648, 'disease': 9.711775626480131,
'loss': 18.114938642504757, 'site': 34.3502984950584, 'however':
16.892054333568716, 'anti': 119.91376093707444, 'tyrosine': 44.6993537175033,
'11': 23.948224411856867, 'mm': 22.5140652945649, 'lung': 9.940718865738415,
'control': 64.34746614337718, '15': 23.968995766185675, 'described':
10.41845351927744, 'somatic': 11.02586314188418, 'flt3': 25.440758746606036,
'smad4': 7.956204994757866, 'kras': 32.20769049439441, 'high':
26.22161029530719, 'mutated': 18.7867159435338, 'transfected':
21.19533132779558, 'normal': 12.190416253389987, 'transcriptional':
14.938289103571552, 'region': 63.91562204195051, 'inhibitors':
21.192389680169065, 'tsc2': 10.053487709978032, 'response': 30.135031101764742,
'based': 15.566576540721258, 'fgfr3': 18.611499606669376, 'loop':
17.703895909838185, 'compared': 6.7049300441552555, 'neutral':
9.222073029912856, '14': 9.234706266869283, 'including': 9.240815121061098,
'cancers': 23.390802144364667, 'aml': 34.41213724045063, 'ret':
31.114604753252024, 'genetic': 9.324683446316422, 'increased':
10.060262860016136, 'erk': 9.26522698622272, 'transcription': 43.60562236586017,
'domains': 14.767346963498413, 'proliferation': 16.953236582304005, 'effect':
17.995860993302696, 'although': 18.827119027911348, 'met': 8.565601895776357,

'germline': 20.883183858394418, 'interaction': 8.933693809940637, 'risk':
27.172319185556535, 'phosphatase': 11.04339609365329, 'catalytic':
16.949822564175065, 'line': 11.211693555797643, '13': 64.1641994395094,
'inhibitor': 7.412575548305001, 'gefitinib': 8.763073297624016, 'survival':
8.061220865938216, 'similar': 13.681591761110612, 'classification':
12.501459984629173, 'within': 7.518740046065783, 'terminal': 13.076908127944066,
'jak2': 14.07816149441789, '50': 8.358186365641023, 'individuals':
14.949173126596786, 'complex': 54.50772339574224, 'known': 21.14232096774641,
'alterations': 130.06791598210256, 'allele': 37.98065761209775, 'acid':
43.44592084606362, 'vitro': 37.663582765277724, 'pi3k': 14.286361710035711,
'inhibition': 13.203763501869984, 'pik3ca': 9.21001742233703, 'mek':
106.51526766393916, 'ar': 26.221695769346542, 'primary': 15.568046417270029,
'detected': 10.08097244455966, 'induced': 9.507261791783614, '30':
9.134393468715139, 'role': 18.27233141651606, 'myc': 39.21180050106153,
'deletion': 24.721939260161136, 'non': 20.979785529119685, 'expressed':
10.486428817995542, 'families': 9.026050328755234, 'presence':
16.130937179687198, 'could': 7.876736925280683, 'rna': 16.301904201770586,
'alleles': 18.981174285285633, '16': 130.5891365813799, 'tp53':
181.119690878331, 'activating': 15.501152078718615, 'oncogenic':
11.102973987698935, 'molecular': 14.415724454613063, '18': 19.488179841684126,
'thus': 8.21426032387228, 'whereas': 11.60077969354286, 'pathogenic':
9.296015848419989, '100': 11.045684407773411, 'independent': 17.403473182446405,
'splicing': 9.516623821113717, 'tumour': 20.464263559366813, 'genomic':
24.33342443635859, 'significant': 14.312979415684925, 'ml': 8.359138625721654,
'containing': 38.98293295154695, 'tsc1': 7.951949895939316, 'positive':
12.911114275230043, '24': 18.42367862005344, 'amplification':
11.674008096841737, 'analyzed': 16.553990076346363, 'either':
13.567108711232555, 'mlh1': 9.253328224031094, 'brca': 9.79087819874791, 'her2':
9.88047591112091, 'likely': 15.100893694324062, 'several': 9.345561587282145,
'catenin': 9.489539943923162, 'active': 17.384767471949555, 'smad2':
27.10097470342265, 'ba': 10.143424580832132, 'leukemia': 10.226167193341123,
'17': 24.01309053510127, 'first': 11.846985887349168, 'kd': 9.872999309553606,
'effects': 10.542317276249364, 'class': 9.27941138247621, 'p85':
7.8868982808913835, 'dependent': 14.263227439007448, 'syndrome':
13.628858492698395, 'structural': 16.562776826645422, 'residue':
9.917425623588823, 'resistant': 18.12166058878376, 'notch': 11.390010772196284,
'total': 8.553012353256374, 'model': 8.931858318587174, 'vector':
13.404783118330963, 'small': 8.768579989383253, 'f3': 21.66615647691707,
'sensitivity': 9.597473545064137, 'addition': 8.632476827010638, 'negative':
15.822143492350088, 'mek1': 8.02495454818823, 'indicated': 29.712805700686637,
'19': 15.583642378599297, 'nih': 18.03853659209034, 'significantly':
15.100875391492655, 'ability': 6.727825170316909, 'group': 9.452661953661417,
'repair': 12.099795037281117, '25': 22.623200960802215, 'treated':
10.518790494546334, 'changes': 10.78409729659949, 'ovarian': 16.263002520593446,
'yeast': 10.081050183767998, '21': 5.872610127603128, 'whether':
9.986757607289253, 'p16ink4a': 8.433037880799503, 'low': 6.587909432867281,
'odds': 16.65163428594867, 'factor': 19.041123628825698, 'promoter':
12.29404346911511, 'four': 10.635517761594695, 'vhl': 64.32099576641127,

'cyclin': 12.571513908766592, 'present': 9.63536062554528, 'cdna':
11.671866290173512, 'activated': 8.862752898377575, 'nsclc': 6.803381421624925,
'target': 10.435923431829686, 'frequency': 8.717438592075672, 'additional':
11.7833122829766, 'therefore': 9.927506795523513, 'history': 11.417442012018563,
'nuclear': 9.423975935225428, 'available': 43.29396615253715, 'gtp':
22.892051965666642, 'among': 15.888723184593124, 'gfp': 8.298555359274475,
'respectively': 15.018935354543547, 'antibody': 20.386057091817637, 'formation':
16.153353217819365, 'exons': 29.423678132759097, 'co': 7.5596084386101365,
'single': 23.356791240221195, 'drug': 8.958165874895613, 'case':
12.88568409522977, 'smad3': 14.880560442568733, 'sites': 9.912630641956067,
'large': 17.814698908634085, 'consistent': 12.547958744487499, 'methods':
8.405970860601581, 'age': 12.074859971710278, 'level': 31.239105495550838,
'copy': 11.944227786948218, 'author': 10.601114780259577, 'tissue':
11.369739466210245, 'vivo': 7.575242731810618, 'mmr': 9.567075808479911,
'tested': 7.9977480337442755, 'development': 7.284050341778924, 'tumours':
30.733614259469615, 'related': 8.652454817526905, 'tgf': 13.143645892422322,
'apoptosis': 9.921106748748539, 'new': 67.99182852731788, 'substitution':
71.69386142370797, 'suppressor': 25.47523346054286, 'therapy':
11.120142814365808, 'mediated': 9.08514634103624, 'evidence': 8.759171054523732,
'obtained': 13.295701157190713, 'important': 13.821761036132237, 'mouse':
7.669036663294783, 'chromosome': 10.181183423261507, 'ii': 18.28230443918455,
'common': 14.25858378912669, 'information': 16.04468995914778, 'show':
10.648785951844218, 'experiments': 25.455703244543876, 'genome':
7.9587316065359355, 'full': 20.492371613252683, 'fold': 7.5720144460749905,
'abl': 12.127752693500664, 'luciferase': 77.63818174001096, 'interactions':
9.006863525449953, 'mapk': 21.17537031148898, 'set': 9.008776879256182, 'flag':
10.284283516675112, 'multiple': 12.612750797286965, 'revealed':
9.390497654846753, 'atp': 11.814723602890513, 'novel': 9.5982777593755, 'bcr':
10.882131722993408, 'akt1': 9.09615296293472, 'pathways': 14.837981129653084,
'fusions': 31.520945455007084, '22': 25.859315323398214, 'predicted':
12.74571277820364, 'test': 9.210401798444009, 'phenotype': 8.18235854377609,
'cycle': 119.74974839224001, 'min': 8.788401661843096, 'reporter':
9.60824817311804, 'nm': 10.862139698241602, 'conserved': 17.485912491108383,
'codon': 14.284505400144466, 'ligand': 10.443006667807202, 'substitutions':
8.868281947077506, 'go': 49.55762561416633, 'il': 18.46529718863291, 'nrf2':
10.92968908174222, 'reduced': 7.850455876834953, 'potential':
17.323296833269403, 'regions': 6.697703176070934, 'cdk4': 22.68985775751458,
'crizotinib': 31.18477454273903, 'ph': 79.63002655830952, 'derived':
11.84027369777673, 'cbl': 19.923842657741776, '2011': 7.95998078124519,
'suggesting': 19.10487875148807, 'hotspots': 13.829858410722473, 'spop':
22.68211107995955, 'e2': 33.46024641182146, 'splice': 12.24920636085617,
'stability': 31.72580655928678, 'progression': 27.170574375912402,
'phosphorylated': 131.3536579554221, 'suggest': 111.19197744775073, 'time':
12.639449518043795, 'western': 7.440445947963604, 'result': 7.417788382764895,
'surface': 13.125638006744449, 'deletions': 20.773658520880957, '40':
11.911817939240244, 'manuscript': 14.017532994685295, 'contrast':
17.04627503232411, 'transformation': 28.90910355602114, 'recurrent':
17.268885242266528, 'increase': 9.124175855255961, 'months': 14.898779065618104,

'nucleotide': 15.023509201595342, 'controls': 18.538716016679988, 'position':
7.265252753023313, 'carcinoma': 10.113166733300318, 'antibodies':
41.31770499873833, 'lapatinib': 19.049954644460367, 'cellular':
8.180642217260315, 'affected': 10.297069805913615, 'might': 9.896300484646666,
'would': 18.890708279582807, 'mrna': 9.182860204651917, 'point':
10.597354031576815, 'types': 17.285176847647374, 'kinases': 43.00003367375404,
'gists': 38.729543211017116, '3t3': 11.077261261203457, 'prostate':
11.052186149924939, 'like': 38.4237294294667, 'colorectal': 16.86478334954003,
'core': 13.597790251813334, 'highly': 9.604021238904961, 'status':
24.50334252145322, 'system': 9.185412370343588, 'form': 70.40696240152897,
'demonstrated': 6.826812262379588, 'located': 11.708090974116931, 'histone':
7.387957153525263, 'bp': 55.10058535450613, 'analyses': 25.94632468296633,
'without': 17.30209904257113, 'following': 21.766237772828735, 'melanomas':
25.393360136200354, 'determined': 18.735758067326024, 'er': 14.079508422295916,
'years': 15.312774359519066, 'helix': 12.518503535567135, 'sequences':
16.47598512330961, 'rt': 10.95650082484352, 'supplemental': 8.279597444904628,
'transfection': 7.944652438301464, 'association': 19.621467655987825,
'required': 9.446819489268597, 'hotspot': 42.32095601123731, 'many':
11.380087178228013, '23': 18.78853399378635, 'panel': 14.27390408978872,
'peptide': 6.887139390847796, 'involved': 11.685244595962406, 'acids':
14.828620798873331, '26': 21.084703286801105, 'substrate': 12.074852764758562,
'overall': 14.15335412204521, '2006': 28.403504661304474, 'hr':
14.239973299527238, '000': 15.065478018933415, 'change': 14.953386352353785,
'mechanism': 18.833771322640622, 'according': 10.271255944935692, 'classified':
14.726325141509529, 'sample': 16.05616823637866, 'buffer': 12.17944216588197,
'examined': 30.322018132658293, 'ha': 14.44369428973463, 'confirmed':
9.9933895070096, 'due': 42.28148907832441, 'higher': 10.20427679606897, 'hif':
7.996572900917535, 'methylation': 14.032747066702907, 'gist': 9.556505493102275,
'bound': 45.485499136330944, 'absence': 11.908596449940783, 'rare':
13.663395249688538, 'ic50': 17.389508301342296, 'five': 10.764858708120187,
'37': 16.466148306329323, 'see': 11.97881382689556, '2012': 43.52300109281833,
'subunit': 11.635652412011101, 'serum': 8.931753996837246, 'factors':
17.469885096061585, 'relative': 10.309033711375669, 'values':
11.083172221825253, 'downstream': 8.547172870458176, 'per': 13.373373029416701,
'sensitive': 26.25896252539514, 'notch1': 15.717217391409955, 'affect':
25.884719266940344, 'blood': 11.58741802988668, 'idh1': 8.06134747680614,
'conformation': 21.797380378058026, 'rate': 12.248086995762566, 'gain':
19.806945583940117, 'mechanisms': 7.56426203719104, 'cohort':
10.982491417584964, 'primers': 12.951725209400088, 'membrane':
24.062772873318753, '2010': 12.32176767263066, 'least': 23.301887313647775,
'second': 9.90744745687127, 'transforming': 17.35552368398127, 'constructs':
9.061220515690247, 'included': 10.309470399830715, 'targeted':
23.73543882426689, '2013': 24.591696674108494, 'mg': 27.864356443337503,
'download': 7.0603696187620635, 'stat3': 6.915989457895411, 'regulation':
8.533597769982318, '29': 9.134551886282651, 'pdgfr': 8.262157620675012, 'ca':
25.0820162114036, 'distinct': 17.12067302417389, 'findings': 11.29618853097178,
'provide': 10.313757959690898, 'bone': 8.883710901618977, 'less':
7.762470551807089, 'models': 7.9225715701771176, 'p16': 8.870919473704006,

'individual': 14.632899554553301, 'previous': 8.866227487986487, 'clones':
8.943112748636297, 'mutational': 24.14899117311404, 'activities':
20.50029198069315, 'research': 10.493539894655724, 'determine':
9.021270542408509, 'smo': 77.44798291734024, '35': 15.315353966435735, '28':
46.0548326421133, 'ros1': 11.737503926954417, 'secondary': 23.979560614872966,
'erlotinib': 28.466325540136356, 'length': 11.74043880680522, 'myeloid':
8.408727714368203, 'staining': 10.442788428124475, 'plasmid':
15.530476779477592, 'signal': 18.1455610834135, '00': 7.244890427474055,
'figures': 8.733067028077864, 'population': 8.89415984170693, 'encoding':
7.594614113318861, 'database': 13.446243398923345, 'diagnosis':
7.1898119797793445, 'lower': 8.269331987179832, 'stem': 12.740901757151043,
'given': 10.667697536222198, 'blot': 13.055059294992128, 'receptors':
20.85509316309879, 'pvhl': 18.062884755778036, 'use': 35.1377503063994, '33':
16.546774765008774, '27': 8.266270562499537, 'induce': 15.152675581700436,
'medium': 21.079725687113715, 'members': 7.439528985548912, 'lymphoma':
24.712063248480703, 'd1': 40.05566358559539, 'overexpression': 8.56441556347492,
'features': 11.79444693153622, 'indicate': 14.998047934120162, 'phospho':
9.934174778103875, 'phase': 7.854837843744959, 'motif': 27.1680963744244,
'recently': 30.47803013935152, 'amplified': 19.270728159259104, 'hours':
12.526279693872304, 'repeats': 17.142635593687274, 'transactivation':
29.72938974572679, 'therapeutic': 12.301693751537846, 'egf': 11.555074787276624,
'corresponding': 6.718157337140976, '95': 9.195229084073729, 'differences':
9.402789609445753, 'heterozygous': 8.855040013969754, 'ng': 9.408727855863173,
'32': 7.440873417347073, '2004': 8.31570651927311, 'six': 15.831926540632184,
'image': 14.722391914855956, 'differentiation': 17.050721855576622, 'fish':
7.800134318745527, 'identify': 48.84104610686225, 'complexes':
11.642094178453402, 'extracellular': 10.540873524344706, '60':
14.411858034327278, 'endogenous': 13.576128769660894, 'structures':
10.85164173238074, 'localization': 17.489237304563158, 'defects':
12.319738059304342, '2007': 23.625395400451932, 'various': 19.812199662901033,
'targeting': 36.55159395341311, 'l858r': 14.88357156425545, 'knockdown':
12.318480258641772, 's1': 13.498640737237267, '2008': 25.422197824470807,
'generated': 10.955151822041136, 'state': 9.45641780814387, 'harboring':
18.11971584431566, 'possible': 14.0828860341673, 'whole': 13.319020758997487,
'coding': 34.512199228828955, 'days': 15.443873638241092, '31':
16.640159154759463, 'mean': 38.74677166016697, 'impact': 21.67412286380406,
'characterized': 21.14930216174243, 'org': 29.836298472532086, 'incubated':
17.84740485136505, 'regulatory': 20.039137614191635, 'lysates':
13.053656275841758, 'size': 22.064520990894025, 'specimens': 8.546976427144985,
'primer': 8.452861833338655, 'nf1': 15.67978895033516, 'defined':
9.552621647620846, 'src': 12.215731256234454, 'constitutive':
17.409169018022272, 'sporadic': 15.406111158306736, 'gst': 31.89244010106968,
'side': 17.005050946554345, 'direct': 10.385147729320977, 'nras':
80.79900722196018, 'interestingly': 65.50526771183625, 'p21':
28.385624442252873, 'together': 115.06507024386433, 'since': 12.91018628781987,
'tagged': 250.28986919962847, 'tkis': 22.905835161780214, 'selected':
12.739379374422475, 'bind': 9.149027274312793, '2009': 19.883139024784818,
'suggests': 10.300407707098273, 'dominant': 26.699885825899973, 'chain':

17.617143473535364, 'sequenced': 10.911414367506094, 'inactive':
11.4413557181285, 'significance': 12.064809483881122, 'functions':
19.704945092587014, 'ratio': 16.573267488168042, 'furthermore':
22.80280663495555, 'chromosomal': 9.074159999114062, 'basis':
28.141361990631523, 'recent': 20.152466915598584, 'alternative':
13.737993293403663, 'resulting': 7.701638288954864, 'value': 16.952459547418123,
'indicating': 11.335438798231603, 'ubiquitin': 16.45845786390963, 'grade':
18.914480104551213, 'metastatic': 18.828428140593317, 's2': 15.629501372667184,
'2005': 7.9461461892407375, '1997': 10.540524521344464, 'expected':
31.93115246059017, 'next': 7.622925180275757, 'enzyme': 6.868665317836061,
'events': 7.955229360463489, 'median': 35.81239818697079, '48':
17.48096674848315, 'reaction': 10.515875626118381, 'powerpoint':
19.185179559805785, 'activate': 10.21741159916988, 'critical':
22.101200431898608, 'page': 42.550484136512154, 'iii': 7.610393705624332,
'seen': 11.592160464403594, 'pa': 19.48186812131603, 'tki': 14.536429007179397,
'lesions': 12.287560028316891, 'ref': 34.23954329131612, 'adenocarcinoma':
13.036552156668092, 'early': 19.272125541696116, 'damage': 11.255871490350335,
'2a': 80.46642538540976, '34': 20.445699504899952, 'dimerization':
10.71964215783337, 'frequently': 10.783894188168269, '36': 14.69517370433805,
'upon': 8.822458469175109, 'concentrations': 8.25943815574605, 'measured':
7.957316475031268, 'null': 21.969320371526393, 'range': 32.288382881546895,
'criteria': 16.877314518532433, 'occur': 35.364819025895294, 'sirna':
88.52298158262529, 'keap1': 9.037220996567394, 'cause': 33.654941142921246,
'1998': 13.204559771715218, 'example': 38.34349132814922, 'lane':
14.654097867690194, 'decreased': 13.787877962686945, 'altered':
7.482262374557212, 'pocket': 31.16953878427127, '39': 16.253010001862044,
'mutagenesis': 12.244640265501431, '1a': 16.73789781133059, 'strong':
24.851069563737173, 'resulted': 7.1388979597465, 'specificity':
12.247639023691669, 'interface': 15.974415874789367, 'purified':
38.00016211207249, 'inhibited': 23.790190904751906, 'inactivation':
23.649053533982663, 'tissues': 7.7470661394933815, 'neuroblastoma':
12.718893386209755, 'frame': 7.3246613864734, 'embryonic': 6.956795857709136,
'stimulation': 10.399311555279331, 'homozygous': 15.344253625328193,
'signalling': 12.616616438214479, 'terminus': 15.568265668598695, 'complete':
9.271008094986973, 'oncogene': 21.378955151445695, 'hydrophobic':
8.436870702840237, 'score': 11.685111893492545, 'driver': 16.369050308402024,
'products': 10.813404108261814, 'comparison': 16.77852088333296, 'targets':
22.63595891356575, 'reverse': 19.0215379441657, 'forms': 8.327241025011075,
'crystal': 12.936381608135083, 'carcinomas': 31.65366945286364, 'assessed':
23.361029789303245, '42': 11.477226873922515, 'another': 13.511835652615941,
'stage': 10.016040012871374, '2003': 7.070252558623259, 'prior':
9.006233169940613, 'http': 10.148033951072152, 'culture': 16.01085694170086,
'stable': 25.458780235636826, 'alone': 8.958358781046208, 'locus':
19.082114426207777, 'deficient': 8.397286098859837, 'strand': 8.21470546378448,
'distribution': 15.158306422649613, 'considered': 88.89268192042833,
'developed': 43.36953385789804, 'induction': 13.100449156365956, 'free':
7.875548055511839, 'colony': 82.89902229671343, 'concentration':
9.691817110024239, 'across': 10.309958652806888, 'colonies': 12.399075837980607,

'tumorigenesis': 42.46063887689866, 'affinity': 10.536606992257523, 'published':
14.055664344496607, '2001': 58.49413374581215, 'standard': 13.59960785281165,
'day': 8.196009379490466, 'even': 6.854885750549565, 'gap': 11.076728775044467,
'enhanced': 10.837701412863852, 'contains': 11.035787212851798, 'showing':
12.192129901254276, 'ring': 35.14460242234646, 'directly': 12.476081616799611,
'identification': 15.721572315041644, 'moreover': 8.937361720443773, 'shows':
16.44951286250578, '45': 8.5298569771355, 'ci': 10.661333618886298, 'carried':
27.94313001773178, 'series': 16.34251245792982, 'combined': 9.300575919813063,
'tables': 13.26604607541551, 'method': 11.582356470417706, 'correlation':
17.764663084408554, 'groups': 13.827424921062516, 'degradation':
7.8173616652987, 'mammalian': 8.266205523450793, '1999': 8.914032697614324,
'majority': 7.442315402628182, 'end': 19.561672852497907, 'subset':
8.372960119085393, 'combination': 12.16793149843093, 'weeks': 9.331020903858082,
'72': 31.415653148311808, 'report': 16.62287445100406, 'assess':
6.780290751828363, 'suggested': 8.914264335055586, 'unique': 7.578859437462698,
'2000': 8.161247175313282, 'top': 14.741079698636325, 'regulated':
12.888092542954798, 'chemotherapy': 20.234070299592275, '01': 37.85178746715768,
'conditions': 37.0099980965253, 'positions': 20.228865443595623, 'assessment':
18.63846628814113, 'colon': 27.689486418054038, 'www': 8.936288581693072,
'approximately': 15.93013692477779, 'approach': 10.36257712767887, 'applied':
11.01592447945658, 'essential': 44.760624335249425, 'calculated':
26.204180210875396, 'via': 16.958326429231718, 'major': 10.124930862409457,
'gel': 8.189306292806542, 'frequent': 9.569603576546443, '2002':
24.89091316609807, '57': 22.595259995885307, '1b': 22.999723219710877, 'nature':
12.76809829650797, 'insight': 14.779070485812415, 'carrying':
11.731857004117362, 'skin': 10.952797959229935, 'followed': 8.166020673622015,
'seven': 14.29509570897289, '41': 36.324556858195365, 'epithelial':
10.198049638637153, 'suppression': 8.553249647943542, 'transformed':
8.271628656385008, 'dose': 8.836943705359715, 'none': 13.436746476117431,
'acquired': 12.755919870192841, 'inhibit': 13.975551354486397, 'view':
10.734248056192996, '38': 11.216661126735366, 'pattern': 8.897193664063954,
'caused': 7.745994430344374, 'key': 13.772498602005868, 'elevated':
19.91761282792977, 'eight': 35.46513570618978, 'product': 11.102916158721493,
'49': 14.827349269420182, '46': 33.54795656922557, 'promote': 9.491583359633042,
'detection': 13.906532350808456, '2b': 17.04980026544283, 'isolated':
9.116987935961468, 'red': 21.018324823964228, 'responses': 8.04277131821871,
'basal': 17.349287798205584, 'construct': 32.4458675131042, 'impaired':
9.570831418766083, 'university': 44.13231115314956, '1996': 9.539013661475567,
'represent': 11.373953044855435, 'remaining': 12.677105268807471, 'selection':
42.457827512472974, 'leading': 10.239597613707225, 'studied': 8.893080003538236,
'signals': 11.080641049928076, 'intermediate': 21.714565494759544, 'invitrogen':
19.670538604450336, 'unknown': 24.355280274375904, 'exhibited':
7.414309988971154, 'involving': 11.248580822330355, 'death': 18.37371894429721,
'malignant': 10.495305721980538, 'sds': 30.410591228538113, 'part':
18.199402094292758, 'statistical': 12.015854117686402, 'evaluated':
11.533135117525049, 'contact': 9.133276625528838, 'biochemical':
20.93349995750777, 'double': 18.246156955769234, 'aberrant': 28.49381421863051,
'lead': 19.942706802543984, 'defective': 12.883858222936862, '05':

28.91741426684979, 'displayed': 32.171804188005666, 'context':
8.042610028978535, '56': 10.349121022535991, '80': 11.47925505507761, 'average':
16.01953769999162, 'linked': 21.77973697104649, 'screening': 16.049644639544322,
'constitutively': 11.388380479036151, 'include': 11.395064200602969, 'molecule':
16.016376155597715, '70': 9.947695780756943, 'insertion': 8.12241955621696,
'reduction': 10.030432733297895, 'times': 7.2829655241380005, '90':
12.718923929636885, 'molecules': 9.640020700547176, 'affecting':
13.291466821404498, 'possibility': 11.689066179995532, 'cultured':
8.788519551461436, 'lack': 15.060107009676873, 'testing': 12.522186689726091,
'difference': 7.098496486240542, 'properties': 7.651425902917529, 'repeat':
10.271960136809959, 'clear': 9.925847943472682, 'box': 10.38048047499208,
'presented': 7.902501028730332, 'manufacturer': 20.25263863527848,
'demonstrate': 32.522101876390764, '43': 11.812448233013637, 'greater':
8.893997894388729, 'sd': 32.61772240730971, 'left': 37.96140571021915, 'ligase':
7.722267469112084, 'relevant': 6.724449056805805, '3a': 9.37663138667399,
'interact': 17.517286283837343, 'partial': 16.493086803089977, 'proportion':
14.570549102963414, 'characteristics': 8.17789439940003, 'rates':
13.929396054359241, '59': 7.142953719960052, 'right': 15.961975364537865,
'established': 9.320965150009235, 'fragment': 16.11381481293202, 'substrates':
11.153739598108144, 's3': 14.772942369536995, 'require': 58.39444486585623,
'stably': 7.476630656905982, 'able': 9.085633594439122, 'indeed':
17.506608229550306, 'biological': 15.924268673808966, 'short':
24.387191700014267, 'specifically': 20.334713217226145, 'contribute':
15.05937631199273, 'unable': 58.565576034526124, 'discovery': 9.463816251706923,
'hypothesis': 11.241336055379803, 'fact': 6.8754901100938355, 'effective':
18.897218141910706, 'particular': 13.355229277108679, 'american':
11.740832508888012, 'observations': 10.136875741814812, 'clinically':
24.189933058791024, 'nucleus': 10.229251122584547, 'green': 16.75639242121478,
'washed': 17.82597971374615, 'investigated': 8.408489257512839, 'strongly':
17.760990191252358, '47': 12.130978461332866, 'confer': 17.49436808545959,
'cdk12': 18.878833930852274, 'provided': 34.98660869564752, 'long':
21.993420993152068, 'experimental': 15.939764601085216, '52': 8.497866387626134,
'versus': 17.9795306985899, 'relatively': 10.30627545864661, 'materials':
10.698200649144294, '200': 11.231771452730348, 'introduction':
11.249308087239305, 'plasma': 9.308466314309582, 'selective': 20.08975885580519,
'44': 22.299671369796037, 'subjected': 12.163050485048373, 'note':
25.824863545100275, 'driven': 28.068797325611165, 'stimulated':
28.330205482225477, 'numbers': 14.770607194852735, 'open': 15.747273997045504,
'leads': 9.085247444015696, 'representative': 13.425457401452945, 'directed':
19.523708425628808, 'efficacy': 33.12264446278075, 'trials': 7.56446625388432,
'indicates': 21.57849683483918, 'despite': 27.856786208325087, 'added':
78.25982371576923, 'blue': 9.731621477798535, 'percentage': 65.53460153257673,
'75': 21.770940492252322, 'support': 17.812256830238944, 'remains':
56.58033800903422, 'manner': 74.2407235274466, 'finally': 15.32498542172929,
'limited': 30.102650424429807, 'finding': 10.961123534490342, '61':
8.023689665076908, 'similarly': 9.315327332990291, 'binds': 8.931020975331355,
'51': 8.870675039975325, 'generation': 10.563199386330067, 'plates':
12.39905808501046, 'discussion': 42.056534689414825, 'stained': 67.647922022504,

```
'formed': 10.995978947688831, 'larger': 13.822424831492006, 'led':
42.28071447062444, '3b': 108.95614237306775, 'sufficient': 11.759649522399261,
'phosphate': 20.003894133642607, 'still': 7.821039389128981, 'process':
19.041846768852277, 'inhibitory': 9.19757398265941, '4a': 9.059709716475899,
'54': 23.853895433373555, 'play': 17.923665050515083, 'initial':
45.09194137537854, 'half': 7.92525685400908, 'taken': 9.340556593928133,
'observation': 31.405508411131613, 'advanced': 15.930519143042343, 'rather':
21.9830679021014, 'general': 19.351714138335083, 'decrease': 11.68183927131608,
'reports': 61.73100314409148, 'correlated': 24.27969166184704, 'subsequent':
14.936912581598774, 'made': 15.440451601967272, 'appears': 45.36045721997128,
'express': 9.25308834025208, 'yet': 14.834176336125395, 'current':
19.457027884555604, 'cul3': 6.693365651931919}
```

[ ]:
```
# 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)
```

[ ]:
```
x = PrettyTable()

x.field_names = ["Model", "Train_Log-loss", "Val_Log-loss", "Test_Los-loss",␣
↪'Misclassified points(%)']

x.add_row(["Multinomial-NB", 0.44, 1.18, 1.24, 0.38])
x.add_row(["Logistic Reg.", 0.40, 1.01, 1.05, 0.35])
x.add_row(["Linear-SVM", 0.32, 1.02, 1.05, 0.34])
x.add_row(["Random Forest", 0.86, 1.18, 1.18, 0.44])
x.add_row(["Stacking", 0.34, 1.26, 1.30, 0.39])
x.add_row(["Voting", 0.78, 1.20, 1.23, .38])

print(x)
```

```
+---------------+---------------+-------------+--------------+-------------
-----------+
|     Model     | Train_Log-loss | Val_Log-loss | Test_Los-loss | Misclassified
points(%) |
+---------------+---------------+-------------+--------------+-------------
-----------+
```

```
| Multinomial-NB |      0.44      |     1.18     |      1.24     |
0.38          |
| Logistic Reg.  |      0.4       |     1.01     |      1.05     |
0.35          |
|   Linear-SVM   |      0.32      |     1.02     |      1.05     |
0.34          |
| Random Forest  |      0.86      |     1.18     |      1.18     |
0.44          |
|    Stacking    |      0.34      |     1.26     |      1.3      |
0.39          |
|     Voting     |      0.78      |     1.2      |      1.23     |
0.38          |
+----------------+----------------+--------------+---------------+--------------
-----------+
```

2.3 Task 3 : Logistic Regression with BOW(unigrams and bigrams)

[ ]:
```
# building a CountVectorizer with all the words that occured minimum 3 times in␣
 ↪train data
text_vectorizer = CountVectorizer(min_df=3, ngram_range= (1,2))
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 : 785864
```

[ ]:
```
# 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)
```

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

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

```
Counter({3: 145147, 4: 114819, 5: 80743, 6: 53940, 7: 43636, 8: 36502, 9: 31687,
10: 24543, 11: 23403, 12: 18686, 13: 17094, 15: 14567, 14: 13932, 16: 12573, 17:
8221, 18: 8152, 20: 7011, 21: 6882, 19: 6137, 24: 5932, 22: 5010, 30: 4312, 23:
3724, 25: 3593, 26: 3346, 46: 3300, 28: 3194, 27: 3085, 37: 3000, 48: 2767, 29:
2417, 32: 2365, 31: 2314, 45: 2213, 33: 2040, 34: 1871, 35: 1859, 36: 1787, 44:
1655, 38: 1620, 40: 1551, 61: 1507, 39: 1490, 47: 1275, 42: 1262, 41: 1207, 49:
1186, 43: 1107, 50: 1056, 52: 980, 51: 964, 54: 876, 53: 833, 55: 803, 60: 795,
56: 773, 62: 726, 57: 713, 63: 637, 58: 637, 59: 618, 64: 601, 65: 598, 66: 565,
72: 528, 90: 511, 68: 505, 67: 486, 70: 480, 74: 479, 69: 458, 71: 433, 75: 422,
76: 404, 73: 402, 80: 390, 77: 380, 78: 378, 79: 365, 81: 345, 88: 343, 92: 339,
96: 314, 84: 299, 85: 298, 82: 286, 91: 284, 86: 284, 89: 272, 83: 272, 95: 271,
97: 265, 87: 265, 93: 256, 94: 245, 98: 238, 105: 222, 101: 220, 100: 215, 107:
211, 104: 211, 99: 202, 112: 201, 108: 200, 120: 193, 111: 193, 106: 189, 102:
185, 109: 183, 103: 181, 113: 177, 122: 174, 110: 174, 117: 162, 115: 162, 114:
161, 135: 153, 127: 152, 116: 148, 124: 145, 121: 145, 129: 139, 128: 138, 126:
138, 118: 137, 144: 135, 119: 133, 133: 132, 131: 132, 134: 130, 125: 127, 136:
126, 123: 126, 140: 125, 130: 119, 132: 118, 147: 113, 145: 109, 138: 109, 139:
108, 142: 107, 150: 104, 154: 101, 141: 100, 161: 99, 148: 99, 149: 98, 137: 96,
146: 95, 153: 94, 151: 93, 143: 93, 156: 92, 180: 88, 162: 88, 168: 86, 155: 86,
160: 83, 165: 81, 163: 77, 152: 77, 167: 76, 173: 75, 184: 73, 157: 73, 159: 72,
158: 72, 205: 70, 185: 70, 170: 70, 183: 69, 164: 69, 192: 68, 176: 68, 171: 67,
166: 67, 174: 66, 172: 66, 177: 64, 196: 62, 190: 61, 181: 61, 178: 61, 169: 61,
202: 59, 199: 59, 213: 58, 195: 58, 186: 58, 179: 58, 206: 56, 204: 56, 194: 56,
188: 56, 187: 56, 240: 55, 203: 53, 189: 53, 182: 53, 175: 53, 198: 52, 191: 52,
214: 51, 211: 51, 210: 51, 197: 49, 193: 49, 235: 48, 216: 48, 228: 46, 225: 46,
201: 45, 270: 44, 245: 44, 241: 44, 221: 44, 215: 44, 212: 44, 268: 43, 226: 43,
200: 43, 238: 42, 233: 42, 276: 40, 250: 40, 237: 40, 229: 40, 222: 40, 218: 40,
266: 39, 227: 39, 301: 38, 248: 38, 242: 38, 223: 38, 261: 37, 232: 37, 230: 37,
220: 37, 217: 37, 207: 37, 288: 36, 256: 36, 236: 36, 234: 35, 209: 35, 208: 35,
331: 34, 254: 34, 249: 34, 219: 34, 291: 33, 272: 33, 258: 33, 251: 33, 247: 33,
239: 33, 282: 32, 267: 32, 263: 32, 244: 32, 224: 32, 274: 31, 264: 31, 253: 31,
295: 30, 284: 30, 260: 30, 336: 29, 290: 29, 285: 29, 265: 29, 255: 29, 278: 28,
275: 28, 257: 28, 320: 27, 310: 27, 296: 27, 281: 27, 243: 27, 325: 26, 304: 26,
297: 26, 294: 26, 286: 26, 273: 26, 271: 26, 259: 26, 252: 26, 246: 26, 231: 26,
```

322: 25, 319: 25, 302: 25, 280: 25, 262: 25, 330: 24, 306: 24, 300: 24, 414: 23,
360: 23, 327: 23, 323: 23, 292: 23, 279: 23, 277: 23, 269: 23, 432: 22, 367: 22,
344: 22, 338: 22, 316: 22, 313: 22, 312: 22, 305: 22, 298: 22, 289: 22, 283: 22,
363: 21, 345: 21, 337: 21, 318: 21, 309: 21, 308: 21, 293: 21, 416: 20, 412: 20,
366: 20, 351: 20, 334: 20, 315: 20, 307: 20, 413: 19, 377: 19, 347: 19, 341: 19,
324: 19, 299: 19, 368: 18, 335: 18, 333: 18, 436: 17, 406: 17, 401: 17, 391: 17,
386: 17, 381: 17, 362: 17, 357: 17, 355: 17, 354: 17, 353: 17, 352: 17, 348: 17,
339: 17, 314: 17, 311: 17, 488: 16, 392: 16, 356: 16, 349: 16, 346: 16, 329: 16,
317: 16, 481: 15, 456: 15, 405: 15, 396: 15, 376: 15, 372: 15, 361: 15, 359: 15,
358: 15, 332: 15, 328: 15, 760: 14, 534: 14, 510: 14, 457: 14, 450: 14, 431: 14,
430: 14, 429: 14, 419: 14, 415: 14, 402: 14, 395: 14, 394: 14, 387: 14, 382: 14,
380: 14, 379: 14, 378: 14, 373: 14, 371: 14, 369: 14, 342: 14, 340: 14, 326: 14,
303: 14, 287: 14, 661: 13, 557: 13, 523: 13, 496: 13, 473: 13, 461: 13, 448: 13,
403: 13, 384: 13, 375: 13, 530: 12, 515: 12, 504: 12, 495: 12, 478: 12, 470: 12,
465: 12, 464: 12, 444: 12, 442: 12, 440: 12, 439: 12, 434: 12, 428: 12, 422: 12,
420: 12, 404: 12, 400: 12, 399: 12, 370: 12, 350: 12, 763: 11, 541: 11, 540: 11,
528: 11, 508: 11, 507: 11, 482: 11, 472: 11, 462: 11, 459: 11, 455: 11, 446: 11,
421: 11, 418: 11, 417: 11, 411: 11, 410: 11, 409: 11, 398: 11, 390: 11, 388: 11,
343: 11, 321: 11, 767: 10, 643: 10, 634: 10, 620: 10, 574: 10, 563: 10, 556: 10,
539: 10, 522: 10, 514: 10, 511: 10, 500: 10, 494: 10, 490: 10, 486: 10, 485: 10,
480: 10, 453: 10, 447: 10, 445: 10, 443: 10, 441: 10, 427: 10, 397: 10, 385: 10,
383: 10, 646: 9, 633: 9, 631: 9, 625: 9, 603: 9, 601: 9, 595: 9, 581: 9, 577: 9,
566: 9, 559: 9, 552: 9, 543: 9, 542: 9, 532: 9, 531: 9, 513: 9, 506: 9, 492: 9,
484: 9, 483: 9, 474: 9, 469: 9, 435: 9, 425: 9, 407: 9, 389: 9, 1045: 8, 1011:
8, 919: 8, 711: 8, 705: 8, 687: 8, 683: 8, 648: 8, 624: 8, 607: 8, 582: 8, 576:
8, 573: 8, 572: 8, 568: 8, 550: 8, 549: 8, 545: 8, 491: 8, 466: 8, 426: 8, 408:
8, 393: 8, 364: 8, 1268: 7, 1262: 7, 1056: 7, 1050: 7, 1047: 7, 839: 7, 803: 7,
796: 7, 794: 7, 785: 7, 768: 7, 757: 7, 732: 7, 728: 7, 727: 7, 680: 7, 673: 7,
670: 7, 664: 7, 658: 7, 657: 7, 649: 7, 645: 7, 636: 7, 619: 7, 614: 7, 606: 7,
602: 7, 600: 7, 594: 7, 593: 7, 587: 7, 586: 7, 585: 7, 584: 7, 565: 7, 560: 7,
555: 7, 554: 7, 548: 7, 537: 7, 536: 7, 516: 7, 509: 7, 505: 7, 503: 7, 498: 7,
493: 7, 489: 7, 477: 7, 476: 7, 467: 7, 463: 7, 460: 7, 458: 7, 449: 7, 374: 7,
1342: 6, 1112: 6, 1040: 6, 1030: 6, 953: 6, 936: 6, 917: 6, 895: 6, 886: 6, 864:
6, 860: 6, 833: 6, 817: 6, 808: 6, 802: 6, 797: 6, 781: 6, 777: 6, 765: 6, 752:
6, 746: 6, 741: 6, 739: 6, 721: 6, 710: 6, 709: 6, 707: 6, 689: 6, 681: 6, 669:
6, 665: 6, 662: 6, 656: 6, 655: 6, 651: 6, 650: 6, 640: 6, 637: 6, 621: 6, 618:
6, 616: 6, 613: 6, 608: 6, 589: 6, 588: 6, 580: 6, 570: 6, 569: 6, 567: 6, 538:
6, 535: 6, 527: 6, 521: 6, 520: 6, 519: 6, 512: 6, 499: 6, 471: 6, 454: 6, 452:
6, 438: 6, 437: 6, 433: 6, 365: 6, 2433: 5, 1530: 5, 1391: 5, 1378: 5, 1306: 5,
1301: 5, 1233: 5, 1195: 5, 1162: 5, 1080: 5, 1010: 5, 1007: 5, 951: 5, 937: 5,
925: 5, 907: 5, 894: 5, 892: 5, 888: 5, 887: 5, 882: 5, 872: 5, 871: 5, 866: 5,
855: 5, 850: 5, 821: 5, 815: 5, 812: 5, 811: 5, 798: 5, 791: 5, 790: 5, 779: 5,
776: 5, 766: 5, 758: 5, 755: 5, 754: 5, 737: 5, 734: 5, 731: 5, 716: 5, 712: 5,
708: 5, 703: 5, 702: 5, 701: 5, 700: 5, 699: 5, 697: 5, 695: 5, 694: 5, 686: 5,
678: 5, 675: 5, 638: 5, 622: 5, 612: 5, 610: 5, 609: 5, 604: 5, 598: 5, 597: 5,
591: 5, 579: 5, 571: 5, 562: 5, 553: 5, 544: 5, 529: 5, 526: 5, 524: 5, 517: 5,
502: 5, 501: 5, 497: 5, 487: 5, 475: 5, 451: 5, 424: 5, 423: 5, 2672: 4, 2445:
4, 2258: 4, 2049: 4, 2026: 4, 1766: 4, 1757: 4, 1707: 4, 1698: 4, 1625: 4, 1622:

4, 1453: 4, 1354: 4, 1340: 4, 1289: 4, 1266: 4, 1244: 4, 1230: 4, 1211: 4, 1202:
4, 1200: 4, 1177: 4, 1165: 4, 1160: 4, 1154: 4, 1117: 4, 1107: 4, 1088: 4, 1069:
4, 1066: 4, 1058: 4, 1043: 4, 1019: 4, 1017: 4, 1015: 4, 1008: 4, 1005: 4, 1002:
4, 1000: 4, 996: 4, 995: 4, 994: 4, 989: 4, 985: 4, 982: 4, 979: 4, 977: 4, 969:
4, 963: 4, 959: 4, 947: 4, 934: 4, 930: 4, 924: 4, 923: 4, 915: 4, 914: 4, 911:
4, 906: 4, 900: 4, 884: 4, 861: 4, 858: 4, 853: 4, 844: 4, 843: 4, 840: 4, 836:
4, 832: 4, 831: 4, 829: 4, 826: 4, 823: 4, 822: 4, 819: 4, 814: 4, 810: 4, 799:
4, 788: 4, 786: 4, 778: 4, 775: 4, 773: 4, 770: 4, 762: 4, 756: 4, 753: 4, 748:
4, 744: 4, 743: 4, 730: 4, 726: 4, 725: 4, 722: 4, 719: 4, 713: 4, 706: 4, 698:
4, 679: 4, 672: 4, 671: 4, 668: 4, 667: 4, 663: 4, 652: 4, 644: 4, 642: 4, 632:
4, 629: 4, 623: 4, 596: 4, 578: 4, 564: 4, 551: 4, 546: 4, 533: 4, 525: 4, 518:
4, 479: 4, 6087: 3, 5613: 3, 3433: 3, 3361: 3, 2761: 3, 2617: 3, 2512: 3, 2485:
3, 2462: 3, 2440: 3, 2249: 3, 2212: 3, 2207: 3, 2176: 3, 2173: 3, 2126: 3, 2110:
3, 2038: 3, 2027: 3, 2017: 3, 1993: 3, 1859: 3, 1839: 3, 1804: 3, 1796: 3, 1774:
3, 1763: 3, 1755: 3, 1725: 3, 1702: 3, 1700: 3, 1677: 3, 1656: 3, 1640: 3, 1633:
3, 1629: 3, 1613: 3, 1603: 3, 1598: 3, 1595: 3, 1587: 3, 1584: 3, 1575: 3, 1565:
3, 1562: 3, 1538: 3, 1531: 3, 1523: 3, 1511: 3, 1508: 3, 1498: 3, 1476: 3, 1474:
3, 1469: 3, 1464: 3, 1459: 3, 1451: 3, 1397: 3, 1393: 3, 1379: 3, 1367: 3, 1350:
3, 1346: 3, 1330: 3, 1319: 3, 1316: 3, 1312: 3, 1311: 3, 1310: 3, 1297: 3, 1292:
3, 1287: 3, 1286: 3, 1276: 3, 1275: 3, 1274: 3, 1270: 3, 1263: 3, 1241: 3, 1234:
3, 1232: 3, 1231: 3, 1223: 3, 1206: 3, 1205: 3, 1203: 3, 1196: 3, 1188: 3, 1182:
3, 1180: 3, 1174: 3, 1171: 3, 1161: 3, 1146: 3, 1143: 3, 1137: 3, 1132: 3, 1130:
3, 1122: 3, 1119: 3, 1114: 3, 1110: 3, 1106: 3, 1099: 3, 1090: 3, 1089: 3, 1079:
3, 1077: 3, 1065: 3, 1051: 3, 1039: 3, 1037: 3, 1036: 3, 1035: 3, 1028: 3, 1016:
3, 1014: 3, 1009: 3, 1004: 3, 1003: 3, 998: 3, 980: 3, 978: 3, 975: 3, 970: 3,
965: 3, 962: 3, 949: 3, 948: 3, 933: 3, 927: 3, 922: 3, 903: 3, 902: 3, 901: 3,
899: 3, 896: 3, 890: 3, 885: 3, 876: 3, 875: 3, 874: 3, 873: 3, 869: 3, 862: 3,
857: 3, 856: 3, 854: 3, 848: 3, 835: 3, 828: 3, 825: 3, 824: 3, 818: 3, 809: 3,
806: 3, 801: 3, 795: 3, 792: 3, 784: 3, 782: 3, 780: 3, 769: 3, 764: 3, 761: 3,
747: 3, 745: 3, 736: 3, 733: 3, 724: 3, 723: 3, 718: 3, 715: 3, 714: 3, 693: 3,
688: 3, 685: 3, 682: 3, 677: 3, 674: 3, 660: 3, 659: 3, 654: 3, 653: 3, 647: 3,
641: 3, 639: 3, 630: 3, 628: 3, 626: 3, 617: 3, 615: 3, 605: 3, 599: 3, 592: 3,
590: 3, 583: 3, 575: 3, 561: 3, 558: 3, 547: 3, 11587: 2, 7176: 2, 6844: 2,
6286: 2, 6013: 2, 5900: 2, 5667: 2, 5450: 2, 4895: 2, 4747: 2, 4491: 2, 4175: 2,
4120: 2, 4099: 2, 4034: 2, 4012: 2, 3989: 2, 3945: 2, 3862: 2, 3650: 2, 3612: 2,
3583: 2, 3569: 2, 3567: 2, 3468: 2, 3465: 2, 3427: 2, 3417: 2, 3415: 2, 3398: 2,
3389: 2, 3350: 2, 3339: 2, 3293: 2, 3287: 2, 3267: 2, 3242: 2, 3217: 2, 3147: 2,
3124: 2, 3067: 2, 3032: 2, 3030: 2, 3009: 2, 2976: 2, 2966: 2, 2899: 2, 2844: 2,
2840: 2, 2781: 2, 2715: 2, 2678: 2, 2677: 2, 2662: 2, 2646: 2, 2639: 2, 2598: 2,
2568: 2, 2562: 2, 2556: 2, 2545: 2, 2543: 2, 2518: 2, 2509: 2, 2508: 2, 2499: 2,
2479: 2, 2474: 2, 2472: 2, 2458: 2, 2434: 2, 2395: 2, 2386: 2, 2365: 2, 2343: 2,
2327: 2, 2323: 2, 2293: 2, 2287: 2, 2271: 2, 2263: 2, 2247: 2, 2246: 2, 2221: 2,
2208: 2, 2185: 2, 2181: 2, 2179: 2, 2168: 2, 2167: 2, 2145: 2, 2125: 2, 2121: 2,
2114: 2, 2107: 2, 2106: 2, 2101: 2, 2099: 2, 2076: 2, 2075: 2, 2074: 2, 2063: 2,
2053: 2, 2035: 2, 2034: 2, 2030: 2, 2021: 2, 2018: 2, 2012: 2, 2004: 2, 2001: 2,
1997: 2, 1974: 2, 1965: 2, 1957: 2, 1952: 2, 1951: 2, 1945: 2, 1939: 2, 1927: 2,
1915: 2, 1912: 2, 1906: 2, 1901: 2, 1896: 2, 1895: 2, 1890: 2, 1888: 2, 1887: 2,
1877: 2, 1874: 2, 1864: 2, 1858: 2, 1856: 2, 1855: 2, 1851: 2, 1850: 2, 1849: 2,

1841: 2, 1837: 2, 1824: 2, 1817: 2, 1808: 2, 1798: 2, 1794: 2, 1789: 2, 1786: 2,
1784: 2, 1780: 2, 1771: 2, 1769: 2, 1759: 2, 1756: 2, 1752: 2, 1745: 2, 1729: 2,
1714: 2, 1697: 2, 1695: 2, 1693: 2, 1690: 2, 1676: 2, 1668: 2, 1662: 2, 1660: 2,
1651: 2, 1646: 2, 1638: 2, 1637: 2, 1635: 2, 1628: 2, 1614: 2, 1611: 2, 1608: 2,
1601: 2, 1599: 2, 1589: 2, 1576: 2, 1571: 2, 1566: 2, 1556: 2, 1553: 2, 1546: 2,
1545: 2, 1540: 2, 1519: 2, 1510: 2, 1507: 2, 1501: 2, 1495: 2, 1492: 2, 1491: 2,
1488: 2, 1484: 2, 1473: 2, 1468: 2, 1465: 2, 1462: 2, 1460: 2, 1457: 2, 1446: 2,
1444: 2, 1441: 2, 1440: 2, 1435: 2, 1430: 2, 1423: 2, 1421: 2, 1419: 2, 1418: 2,
1415: 2, 1413: 2, 1408: 2, 1396: 2, 1392: 2, 1389: 2, 1388: 2, 1384: 2, 1380: 2,
1376: 2, 1375: 2, 1371: 2, 1370: 2, 1366: 2, 1365: 2, 1364: 2, 1363: 2, 1362: 2,
1356: 2, 1355: 2, 1345: 2, 1332: 2, 1331: 2, 1329: 2, 1323: 2, 1321: 2, 1318: 2,
1313: 2, 1308: 2, 1302: 2, 1300: 2, 1299: 2, 1294: 2, 1293: 2, 1285: 2, 1282: 2,
1281: 2, 1273: 2, 1267: 2, 1265: 2, 1259: 2, 1257: 2, 1256: 2, 1254: 2, 1252: 2,
1245: 2, 1242: 2, 1239: 2, 1236: 2, 1235: 2, 1226: 2, 1219: 2, 1215: 2, 1213: 2,
1212: 2, 1209: 2, 1207: 2, 1201: 2, 1199: 2, 1197: 2, 1191: 2, 1189: 2, 1186: 2,
1176: 2, 1175: 2, 1170: 2, 1169: 2, 1164: 2, 1159: 2, 1157: 2, 1155: 2, 1153: 2,
1151: 2, 1150: 2, 1144: 2, 1141: 2, 1135: 2, 1134: 2, 1133: 2, 1129: 2, 1125: 2,
1121: 2, 1116: 2, 1115: 2, 1109: 2, 1108: 2, 1101: 2, 1100: 2, 1096: 2, 1094: 2,
1092: 2, 1087: 2, 1085: 2, 1084: 2, 1081: 2, 1078: 2, 1076: 2, 1072: 2, 1071: 2,
1070: 2, 1068: 2, 1063: 2, 1061: 2, 1057: 2, 1049: 2, 1041: 2, 1033: 2, 1026: 2,
1025: 2, 1024: 2, 1013: 2, 1012: 2, 1001: 2, 997: 2, 991: 2, 987: 2, 983: 2,
974: 2, 973: 2, 971: 2, 967: 2, 964: 2, 958: 2, 946: 2, 945: 2, 943: 2, 942: 2,
940: 2, 926: 2, 921: 2, 920: 2, 918: 2, 916: 2, 912: 2, 908: 2, 905: 2, 904: 2,
881: 2, 880: 2, 877: 2, 867: 2, 849: 2, 847: 2, 846: 2, 845: 2, 842: 2, 841: 2,
838: 2, 837: 2, 834: 2, 827: 2, 820: 2, 816: 2, 805: 2, 800: 2, 789: 2, 783: 2,
772: 2, 771: 2, 751: 2, 750: 2, 749: 2, 742: 2, 738: 2, 729: 2, 720: 2, 717: 2,
704: 2, 691: 2, 684: 2, 676: 2, 666: 2, 635: 2, 611: 2, 468: 2, 151617: 1,
121974: 1, 83057: 1, 69674: 1, 69574: 1, 69029: 1, 68674: 1, 67192: 1, 64057: 1,
63710: 1, 55156: 1, 53542: 1, 51548: 1, 49027: 1, 47056: 1, 46054: 1, 45220: 1,
43270: 1, 42635: 1, 42529: 1, 42143: 1, 40893: 1, 40857: 1, 39920: 1, 39419: 1,
38703: 1, 38236: 1, 36016: 1, 35907: 1, 35735: 1, 35541: 1, 34923: 1, 33840: 1,
33723: 1, 32992: 1, 32611: 1, 32346: 1, 31957: 1, 29215: 1, 28645: 1, 27826: 1,
27316: 1, 26727: 1, 26045: 1, 25991: 1, 25231: 1, 25012: 1, 24968: 1, 24766: 1,
24577: 1, 24449: 1, 24429: 1, 24417: 1, 23923: 1, 23825: 1, 23456: 1, 22556: 1,
22111: 1, 22050: 1, 21563: 1, 21471: 1, 21423: 1, 21191: 1, 20653: 1, 20521: 1,
20219: 1, 19541: 1, 19521: 1, 19516: 1, 19500: 1, 19276: 1, 19135: 1, 19080: 1,
19015: 1, 19009: 1, 18859: 1, 18621: 1, 18492: 1, 18468: 1, 18449: 1, 18301: 1,
18209: 1, 18087: 1, 17971: 1, 17947: 1, 17938: 1, 17805: 1, 17735: 1, 17604: 1,
17583: 1, 17501: 1, 17456: 1, 17364: 1, 17153: 1, 17049: 1, 16887: 1, 16811: 1,
16744: 1, 16671: 1, 16190: 1, 16039: 1, 15972: 1, 15898: 1, 15874: 1, 15807: 1,
15778: 1, 15646: 1, 15578: 1, 15550: 1, 15534: 1, 15429: 1, 15327: 1, 15251: 1,
15068: 1, 15028: 1, 14846: 1, 14813: 1, 14731: 1, 14628: 1, 14576: 1, 14507: 1,
14347: 1, 14343: 1, 14337: 1, 13871: 1, 13861: 1, 13854: 1, 13742: 1, 13578: 1,
13568: 1, 13525: 1, 13432: 1, 13391: 1, 13132: 1, 13099: 1, 13096: 1, 13068: 1,
13043: 1, 12988: 1, 12976: 1, 12916: 1, 12861: 1, 12860: 1, 12842: 1, 12817: 1,
12791: 1, 12742: 1, 12738: 1, 12733: 1, 12676: 1, 12588: 1, 12563: 1, 12523: 1,
12511: 1, 12408: 1, 12405: 1, 12402: 1, 12357: 1, 12321: 1, 12235: 1, 12203: 1,
12125: 1, 12113: 1, 12081: 1, 12045: 1, 12040: 1, 12009: 1, 12000: 1, 11957: 1,

11887: 1, 11815: 1, 11794: 1, 11793: 1, 11747: 1, 11690: 1, 11641: 1, 11603: 1,
11601: 1, 11435: 1, 11335: 1, 11313: 1, 11309: 1, 11243: 1, 11177: 1, 11171: 1,
11033: 1, 10935: 1, 10857: 1, 10748: 1, 10715: 1, 10660: 1, 10608: 1, 10539: 1,
10507: 1, 10457: 1, 10393: 1, 10339: 1, 10280: 1, 10256: 1, 10218: 1, 10208: 1,
10193: 1, 10106: 1, 10093: 1, 10090: 1, 10086: 1, 10065: 1, 10013: 1, 9978: 1,
9889: 1, 9876: 1, 9871: 1, 9841: 1, 9789: 1, 9738: 1, 9695: 1, 9689: 1, 9649: 1,
9588: 1, 9582: 1, 9465: 1, 9425: 1, 9391: 1, 9372: 1, 9359: 1, 9352: 1, 9297: 1,
9284: 1, 9280: 1, 9217: 1, 9208: 1, 9175: 1, 9138: 1, 9116: 1, 9100: 1, 9071: 1,
9063: 1, 9059: 1, 9046: 1, 9030: 1, 9015: 1, 9011: 1, 9009: 1, 8989: 1, 8972: 1,
8970: 1, 8869: 1, 8841: 1, 8824: 1, 8785: 1, 8782: 1, 8763: 1, 8741: 1, 8675: 1,
8593: 1, 8587: 1, 8567: 1, 8558: 1, 8557: 1, 8555: 1, 8533: 1, 8490: 1, 8485: 1,
8454: 1, 8446: 1, 8444: 1, 8388: 1, 8363: 1, 8291: 1, 8290: 1, 8285: 1, 8257: 1,
8229: 1, 8199: 1, 8164: 1, 8163: 1, 8156: 1, 8135: 1, 8120: 1, 8102: 1, 8080: 1,
8078: 1, 8072: 1, 8038: 1, 8036: 1, 8035: 1, 8030: 1, 8028: 1, 8007: 1, 7983: 1,
7954: 1, 7952: 1, 7888: 1, 7852: 1, 7837: 1, 7797: 1, 7795: 1, 7768: 1, 7748: 1,
7739: 1, 7728: 1, 7692: 1, 7655: 1, 7601: 1, 7595: 1, 7565: 1, 7549: 1, 7546: 1,
7510: 1, 7506: 1, 7494: 1, 7477: 1, 7426: 1, 7418: 1, 7368: 1, 7332: 1, 7283: 1,
7248: 1, 7235: 1, 7234: 1, 7232: 1, 7226: 1, 7212: 1, 7191: 1, 7177: 1, 7154: 1,
7146: 1, 7129: 1, 7120: 1, 7113: 1, 7111: 1, 7087: 1, 7072: 1, 7066: 1, 7062: 1,
7054: 1, 7045: 1, 7024: 1, 7023: 1, 7016: 1, 7015: 1, 7009: 1, 6971: 1, 6963: 1,
6954: 1, 6912: 1, 6910: 1, 6893: 1, 6879: 1, 6869: 1, 6843: 1, 6824: 1, 6819: 1,
6816: 1, 6799: 1, 6790: 1, 6789: 1, 6778: 1, 6769: 1, 6765: 1, 6749: 1, 6721: 1,
6703: 1, 6689: 1, 6659: 1, 6649: 1, 6640: 1, 6627: 1, 6583: 1, 6582: 1, 6555: 1,
6545: 1, 6526: 1, 6500: 1, 6478: 1, 6472: 1, 6454: 1, 6436: 1, 6422: 1, 6412: 1,
6391: 1, 6387: 1, 6345: 1, 6329: 1, 6327: 1, 6323: 1, 6308: 1, 6306: 1, 6289: 1,
6263: 1, 6243: 1, 6223: 1, 6217: 1, 6205: 1, 6200: 1, 6198: 1, 6197: 1, 6193: 1,
6147: 1, 6146: 1, 6136: 1, 6128: 1, 6126: 1, 6117: 1, 6107: 1, 6106: 1, 6076: 1,
6011: 1, 5983: 1, 5958: 1, 5957: 1, 5947: 1, 5906: 1, 5895: 1, 5890: 1, 5889: 1,
5887: 1, 5878: 1, 5874: 1, 5825: 1, 5824: 1, 5809: 1, 5797: 1, 5788: 1, 5778: 1,
5773: 1, 5745: 1, 5730: 1, 5683: 1, 5673: 1, 5655: 1, 5632: 1, 5584: 1, 5580: 1,
5579: 1, 5570: 1, 5547: 1, 5533: 1, 5508: 1, 5499: 1, 5484: 1, 5482: 1, 5480: 1,
5475: 1, 5461: 1, 5456: 1, 5428: 1, 5426: 1, 5419: 1, 5407: 1, 5394: 1, 5387: 1,
5363: 1, 5349: 1, 5341: 1, 5338: 1, 5319: 1, 5316: 1, 5301: 1, 5289: 1, 5277: 1,
5266: 1, 5251: 1, 5240: 1, 5238: 1, 5219: 1, 5213: 1, 5209: 1, 5186: 1, 5176: 1,
5155: 1, 5151: 1, 5140: 1, 5136: 1, 5135: 1, 5119: 1, 5115: 1, 5091: 1, 5058: 1,
5056: 1, 5049: 1, 5039: 1, 5036: 1, 5034: 1, 5027: 1, 5026: 1, 5019: 1, 5011: 1,
5009: 1, 5007: 1, 5004: 1, 4997: 1, 4990: 1, 4989: 1, 4987: 1, 4978: 1, 4974: 1,
4973: 1, 4956: 1, 4952: 1, 4935: 1, 4933: 1, 4932: 1, 4919: 1, 4905: 1, 4900: 1,
4879: 1, 4869: 1, 4867: 1, 4858: 1, 4856: 1, 4851: 1, 4850: 1, 4839: 1, 4835: 1,
4833: 1, 4829: 1, 4827: 1, 4825: 1, 4823: 1, 4816: 1, 4815: 1, 4790: 1, 4788: 1,
4784: 1, 4770: 1, 4765: 1, 4745: 1, 4735: 1, 4713: 1, 4710: 1, 4698: 1, 4695: 1,
4693: 1, 4680: 1, 4671: 1, 4666: 1, 4665: 1, 4656: 1, 4652: 1, 4651: 1, 4649: 1,
4623: 1, 4612: 1, 4608: 1, 4603: 1, 4601: 1, 4598: 1, 4586: 1, 4566: 1, 4559: 1,
4540: 1, 4537: 1, 4535: 1, 4525: 1, 4524: 1, 4513: 1, 4508: 1, 4497: 1, 4483: 1,
4481: 1, 4456: 1, 4450: 1, 4436: 1, 4432: 1, 4421: 1, 4412: 1, 4409: 1, 4399: 1,
4390: 1, 4387: 1, 4375: 1, 4364: 1, 4359: 1, 4357: 1, 4353: 1, 4350: 1, 4344: 1,
4333: 1, 4328: 1, 4323: 1, 4322: 1, 4319: 1, 4318: 1, 4314: 1, 4313: 1, 4309: 1,
4308: 1, 4301: 1, 4289: 1, 4284: 1, 4277: 1, 4273: 1, 4261: 1, 4259: 1, 4253: 1,

4243: 1, 4242: 1, 4241: 1, 4234: 1, 4230: 1, 4229: 1, 4220: 1, 4218: 1, 4212: 1,
4208: 1, 4186: 1, 4184: 1, 4165: 1, 4161: 1, 4159: 1, 4141: 1, 4135: 1, 4132: 1,
4130: 1, 4119: 1, 4103: 1, 4097: 1, 4084: 1, 4070: 1, 4059: 1, 4056: 1, 4049: 1,
4047: 1, 4040: 1, 4022: 1, 4020: 1, 4009: 1, 4007: 1, 4006: 1, 3997: 1, 3996: 1,
3993: 1, 3991: 1, 3980: 1, 3975: 1, 3967: 1, 3962: 1, 3953: 1, 3951: 1, 3948: 1,
3942: 1, 3941: 1, 3939: 1, 3936: 1, 3929: 1, 3928: 1, 3913: 1, 3912: 1, 3908: 1,
3899: 1, 3893: 1, 3892: 1, 3889: 1, 3883: 1, 3881: 1, 3877: 1, 3872: 1, 3866: 1,
3865: 1, 3863: 1, 3859: 1, 3857: 1, 3849: 1, 3846: 1, 3828: 1, 3814: 1, 3813: 1,
3807: 1, 3806: 1, 3795: 1, 3789: 1, 3788: 1, 3786: 1, 3784: 1, 3778: 1, 3753: 1,
3750: 1, 3743: 1, 3741: 1, 3737: 1, 3736: 1, 3733: 1, 3732: 1, 3725: 1, 3709: 1,
3702: 1, 3701: 1, 3697: 1, 3690: 1, 3687: 1, 3685: 1, 3683: 1, 3677: 1, 3670: 1,
3666: 1, 3662: 1, 3655: 1, 3653: 1, 3645: 1, 3644: 1, 3641: 1, 3637: 1, 3636: 1,
3634: 1, 3623: 1, 3611: 1, 3610: 1, 3605: 1, 3604: 1, 3593: 1, 3591: 1, 3577: 1,
3575: 1, 3573: 1, 3563: 1, 3560: 1, 3553: 1, 3550: 1, 3549: 1, 3546: 1, 3540: 1,
3538: 1, 3530: 1, 3512: 1, 3509: 1, 3508: 1, 3502: 1, 3500: 1, 3484: 1, 3483: 1,
3482: 1, 3478: 1, 3476: 1, 3474: 1, 3473: 1, 3454: 1, 3452: 1, 3449: 1, 3443: 1,
3440: 1, 3439: 1, 3434: 1, 3430: 1, 3426: 1, 3424: 1, 3423: 1, 3421: 1, 3410: 1,
3407: 1, 3403: 1, 3388: 1, 3386: 1, 3384: 1, 3381: 1, 3378: 1, 3377: 1, 3376: 1,
3374: 1, 3370: 1, 3353: 1, 3348: 1, 3347: 1, 3344: 1, 3343: 1, 3340: 1, 3336: 1,
3331: 1, 3326: 1, 3324: 1, 3321: 1, 3319: 1, 3315: 1, 3311: 1, 3294: 1, 3280: 1,
3275: 1, 3272: 1, 3271: 1, 3270: 1, 3269: 1, 3268: 1, 3259: 1, 3253: 1, 3244: 1,
3243: 1, 3241: 1, 3239: 1, 3237: 1, 3235: 1, 3229: 1, 3228: 1, 3224: 1, 3223: 1,
3218: 1, 3215: 1, 3209: 1, 3203: 1, 3199: 1, 3196: 1, 3188: 1, 3186: 1, 3179: 1,
3175: 1, 3174: 1, 3172: 1, 3168: 1, 3164: 1, 3160: 1, 3156: 1, 3154: 1, 3149: 1,
3133: 1, 3132: 1, 3131: 1, 3129: 1, 3127: 1, 3119: 1, 3117: 1, 3115: 1, 3114: 1,
3110: 1, 3106: 1, 3103: 1, 3101: 1, 3099: 1, 3093: 1, 3092: 1, 3087: 1, 3079: 1,
3078: 1, 3071: 1, 3069: 1, 3065: 1, 3060: 1, 3056: 1, 3050: 1, 3042: 1, 3029: 1,
3027: 1, 3026: 1, 3021: 1, 3020: 1, 3016: 1, 3011: 1, 3008: 1, 3006: 1, 3005: 1,
2999: 1, 2988: 1, 2985: 1, 2984: 1, 2977: 1, 2974: 1, 2972: 1, 2971: 1, 2969: 1,
2950: 1, 2948: 1, 2947: 1, 2946: 1, 2945: 1, 2942: 1, 2938: 1, 2937: 1, 2932: 1,
2925: 1, 2919: 1, 2913: 1, 2905: 1, 2896: 1, 2893: 1, 2890: 1, 2886: 1, 2883: 1,
2872: 1, 2870: 1, 2866: 1, 2864: 1, 2861: 1, 2858: 1, 2856: 1, 2855: 1, 2853: 1,
2851: 1, 2830: 1, 2820: 1, 2819: 1, 2816: 1, 2808: 1, 2805: 1, 2803: 1, 2795: 1,
2789: 1, 2776: 1, 2769: 1, 2762: 1, 2760: 1, 2759: 1, 2754: 1, 2748: 1, 2747: 1,
2745: 1, 2739: 1, 2735: 1, 2733: 1, 2729: 1, 2728: 1, 2727: 1, 2726: 1, 2725: 1,
2719: 1, 2716: 1, 2710: 1, 2709: 1, 2708: 1, 2706: 1, 2704: 1, 2703: 1, 2701: 1,
2698: 1, 2695: 1, 2691: 1, 2685: 1, 2682: 1, 2681: 1, 2674: 1, 2671: 1, 2665: 1,
2661: 1, 2654: 1, 2649: 1, 2645: 1, 2644: 1, 2641: 1, 2640: 1, 2637: 1, 2636: 1,
2633: 1, 2632: 1, 2628: 1, 2623: 1, 2620: 1, 2619: 1, 2614: 1, 2613: 1, 2610: 1,
2609: 1, 2607: 1, 2600: 1, 2594: 1, 2593: 1, 2588: 1, 2581: 1, 2578: 1, 2572: 1,
2571: 1, 2570: 1, 2569: 1, 2561: 1, 2559: 1, 2555: 1, 2554: 1, 2552: 1, 2547: 1,
2546: 1, 2536: 1, 2534: 1, 2532: 1, 2530: 1, 2529: 1, 2522: 1, 2517: 1, 2507: 1,
2504: 1, 2503: 1, 2502: 1, 2501: 1, 2496: 1, 2493: 1, 2492: 1, 2491: 1, 2490: 1,
2476: 1, 2471: 1, 2470: 1, 2469: 1, 2467: 1, 2460: 1, 2456: 1, 2453: 1, 2452: 1,
2451: 1, 2437: 1, 2435: 1, 2427: 1, 2425: 1, 2424: 1, 2423: 1, 2422: 1, 2420: 1,
2417: 1, 2412: 1, 2410: 1, 2408: 1, 2403: 1, 2401: 1, 2400: 1, 2399: 1, 2393: 1,
2391: 1, 2385: 1, 2384: 1, 2383: 1, 2379: 1, 2374: 1, 2371: 1, 2364: 1, 2356: 1,
2352: 1, 2351: 1, 2348: 1, 2344: 1, 2341: 1, 2339: 1, 2338: 1, 2336: 1, 2330: 1,

2325: 1, 2324: 1, 2319: 1, 2315: 1, 2314: 1, 2312: 1, 2302: 1, 2300: 1, 2295: 1,
2294: 1, 2289: 1, 2284: 1, 2280: 1, 2278: 1, 2273: 1, 2264: 1, 2262: 1, 2260: 1,
2257: 1, 2256: 1, 2255: 1, 2251: 1, 2245: 1, 2244: 1, 2243: 1, 2241: 1, 2237: 1,
2236: 1, 2234: 1, 2233: 1, 2222: 1, 2220: 1, 2214: 1, 2210: 1, 2209: 1, 2206: 1,
2203: 1, 2202: 1, 2200: 1, 2198: 1, 2197: 1, 2196: 1, 2195: 1, 2194: 1, 2192: 1,
2190: 1, 2189: 1, 2188: 1, 2182: 1, 2180: 1, 2177: 1, 2174: 1, 2171: 1, 2170: 1,
2166: 1, 2163: 1, 2159: 1, 2157: 1, 2156: 1, 2149: 1, 2147: 1, 2146: 1, 2143: 1,
2142: 1, 2133: 1, 2132: 1, 2130: 1, 2123: 1, 2118: 1, 2117: 1, 2115: 1, 2112: 1,
2111: 1, 2108: 1, 2100: 1, 2096: 1, 2093: 1, 2089: 1, 2088: 1, 2087: 1, 2086: 1,
2084: 1, 2081: 1, 2080: 1, 2073: 1, 2072: 1, 2069: 1, 2066: 1, 2061: 1, 2060: 1,
2059: 1, 2057: 1, 2054: 1, 2052: 1, 2048: 1, 2044: 1, 2043: 1, 2042: 1, 2041: 1,
2039: 1, 2033: 1, 2032: 1, 2031: 1, 2023: 1, 2016: 1, 2014: 1, 2013: 1, 2009: 1,
1999: 1, 1998: 1, 1996: 1, 1995: 1, 1994: 1, 1992: 1, 1988: 1, 1984: 1, 1980: 1,
1979: 1, 1978: 1, 1975: 1, 1971: 1, 1969: 1, 1964: 1, 1962: 1, 1961: 1, 1959: 1,
1955: 1, 1954: 1, 1949: 1, 1948: 1, 1947: 1, 1946: 1, 1944: 1, 1942: 1, 1941: 1,
1937: 1, 1936: 1, 1933: 1, 1929: 1, 1926: 1, 1922: 1, 1910: 1, 1909: 1, 1907: 1,
1903: 1, 1902: 1, 1899: 1, 1898: 1, 1897: 1, 1893: 1, 1891: 1, 1882: 1, 1880: 1,
1878: 1, 1876: 1, 1872: 1, 1871: 1, 1870: 1, 1865: 1, 1862: 1, 1860: 1, 1857: 1,
1853: 1, 1848: 1, 1847: 1, 1845: 1, 1843: 1, 1840: 1, 1836: 1, 1835: 1, 1834: 1,
1832: 1, 1829: 1, 1825: 1, 1823: 1, 1822: 1, 1820: 1, 1818: 1, 1814: 1, 1810: 1,
1809: 1, 1795: 1, 1793: 1, 1782: 1, 1781: 1, 1778: 1, 1776: 1, 1775: 1, 1773: 1,
1770: 1, 1768: 1, 1761: 1, 1760: 1, 1758: 1, 1753: 1, 1751: 1, 1750: 1, 1748: 1,
1747: 1, 1742: 1, 1741: 1, 1737: 1, 1733: 1, 1732: 1, 1726: 1, 1722: 1, 1721: 1,
1720: 1, 1719: 1, 1718: 1, 1716: 1, 1715: 1, 1712: 1, 1711: 1, 1708: 1, 1705: 1,
1699: 1, 1694: 1, 1691: 1, 1689: 1, 1688: 1, 1686: 1, 1685: 1, 1682: 1, 1681: 1,
1679: 1, 1675: 1, 1673: 1, 1672: 1, 1671: 1, 1669: 1, 1667: 1, 1665: 1, 1659: 1,
1657: 1, 1654: 1, 1652: 1, 1649: 1, 1647: 1, 1644: 1, 1642: 1, 1639: 1, 1636: 1,
1634: 1, 1631: 1, 1624: 1, 1621: 1, 1617: 1, 1616: 1, 1609: 1, 1607: 1, 1602: 1,
1597: 1, 1596: 1, 1594: 1, 1592: 1, 1591: 1, 1583: 1, 1582: 1, 1579: 1, 1578: 1,
1577: 1, 1570: 1, 1569: 1, 1568: 1, 1564: 1, 1563: 1, 1561: 1, 1560: 1, 1558: 1,
1557: 1, 1551: 1, 1549: 1, 1547: 1, 1542: 1, 1541: 1, 1539: 1, 1536: 1, 1535: 1,
1533: 1, 1529: 1, 1524: 1, 1522: 1, 1521: 1, 1516: 1, 1512: 1, 1509: 1, 1502: 1,
1500: 1, 1499: 1, 1497: 1, 1496: 1, 1493: 1, 1490: 1, 1485: 1, 1482: 1, 1481: 1,
1480: 1, 1479: 1, 1477: 1, 1475: 1, 1472: 1, 1470: 1, 1466: 1, 1463: 1, 1454: 1,
1450: 1, 1449: 1, 1448: 1, 1445: 1, 1443: 1, 1438: 1, 1437: 1, 1436: 1, 1434: 1,
1432: 1, 1425: 1, 1424: 1, 1422: 1, 1420: 1, 1417: 1, 1416: 1, 1414: 1, 1412: 1,
1409: 1, 1407: 1, 1406: 1, 1405: 1, 1403: 1, 1402: 1, 1401: 1, 1400: 1, 1399: 1,
1395: 1, 1390: 1, 1385: 1, 1381: 1, 1377: 1, 1374: 1, 1372: 1, 1361: 1, 1360: 1,
1359: 1, 1358: 1, 1357: 1, 1351: 1, 1347: 1, 1341: 1, 1339: 1, 1338: 1, 1337: 1,
1334: 1, 1333: 1, 1326: 1, 1322: 1, 1320: 1, 1317: 1, 1315: 1, 1314: 1, 1309: 1,
1307: 1, 1305: 1, 1304: 1, 1303: 1, 1298: 1, 1296: 1, 1288: 1, 1284: 1, 1283: 1,
1280: 1, 1278: 1, 1272: 1, 1271: 1, 1264: 1, 1261: 1, 1260: 1, 1258: 1, 1255: 1,
1253: 1, 1250: 1, 1249: 1, 1246: 1, 1243: 1, 1240: 1, 1238: 1, 1237: 1, 1229: 1,
1228: 1, 1225: 1, 1224: 1, 1222: 1, 1221: 1, 1216: 1, 1214: 1, 1210: 1, 1208: 1,
1194: 1, 1193: 1, 1192: 1, 1190: 1, 1187: 1, 1185: 1, 1184: 1, 1183: 1, 1181: 1,
1179: 1, 1178: 1, 1173: 1, 1172: 1, 1167: 1, 1166: 1, 1163: 1, 1156: 1, 1149: 1,
1148: 1, 1145: 1, 1140: 1, 1131: 1, 1128: 1, 1127: 1, 1126: 1, 1124: 1, 1118: 1,
1113: 1, 1105: 1, 1102: 1, 1098: 1, 1093: 1, 1091: 1, 1083: 1, 1082: 1, 1075: 1,

```
1074: 1, 1055: 1, 1054: 1, 1052: 1, 1048: 1, 1046: 1, 1042: 1, 1034: 1, 1029: 1,
1023: 1, 1021: 1, 1020: 1, 1018: 1, 1006: 1, 993: 1, 992: 1, 990: 1, 988: 1,
986: 1, 981: 1, 976: 1, 972: 1, 968: 1, 961: 1, 956: 1, 955: 1, 954: 1, 952: 1,
950: 1, 944: 1, 939: 1, 938: 1, 935: 1, 932: 1, 931: 1, 913: 1, 910: 1, 909: 1,
893: 1, 891: 1, 883: 1, 878: 1, 870: 1, 865: 1, 863: 1, 859: 1, 852: 1, 851: 1,
830: 1, 813: 1, 807: 1, 804: 1, 793: 1, 787: 1, 774: 1, 740: 1, 735: 1, 696: 1,
692: 1, 627: 1})
```

2.3.1 stacking of all the three input features

[ ]:
```
# 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']))
```

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

```
One hot encoding features :
(number of data points * number of features) in train data =  (2124, 788044)
(number of data points * number of features) in test data =  (665, 788044)
(number of data points * number of features) in cross validation data = (532,
788044)
```

Logistic Regression

With Class balancing

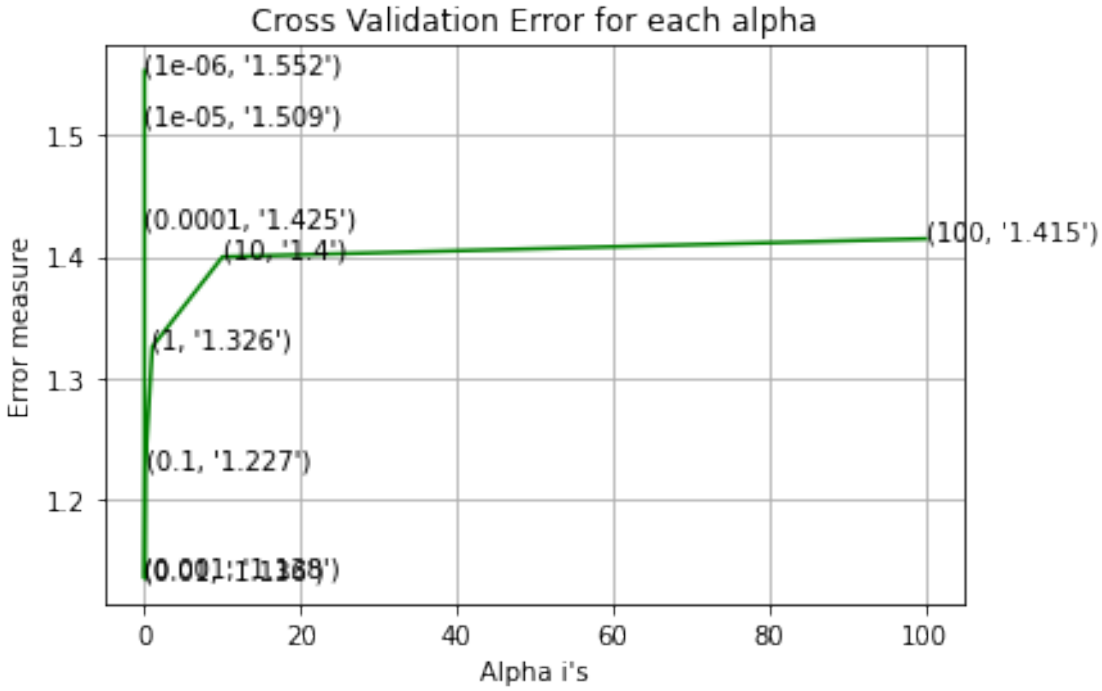
Hyper paramter tuning

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

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

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



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

```
alpha = [10 ** x for x in range(-6, 3)]
cv_log_error_array = []
for i in alpha:
    print("for alpha =", i)
    clf = SGDClassifier(class_weight='balanced', alpha=i, penalty='l2',␣
 ↪loss='log', random_state=42, 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))
    # to avoid rounding error while multiplying probabilites we use␣
 ↪log-probability estimates
    print("Log Loss :",log_loss(cv_y, sig_clf_probs))

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


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

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

```
for alpha = 1e-06
Log Loss : 1.552329991992755
```

```
for alpha = 1e-05
Log Loss : 1.5088426189704847
for alpha = 0.0001
Log Loss : 1.424863882632572
for alpha = 0.001
Log Loss : 1.1375305835300065
for alpha = 0.01
Log Loss : 1.1356452461453193
for alpha = 0.1
Log Loss : 1.2268135820289732
for alpha = 1
Log Loss : 1.3258661714962212
for alpha = 10
Log Loss : 1.399723605340151
for alpha = 100
Log Loss : 1.4145080893256086
```



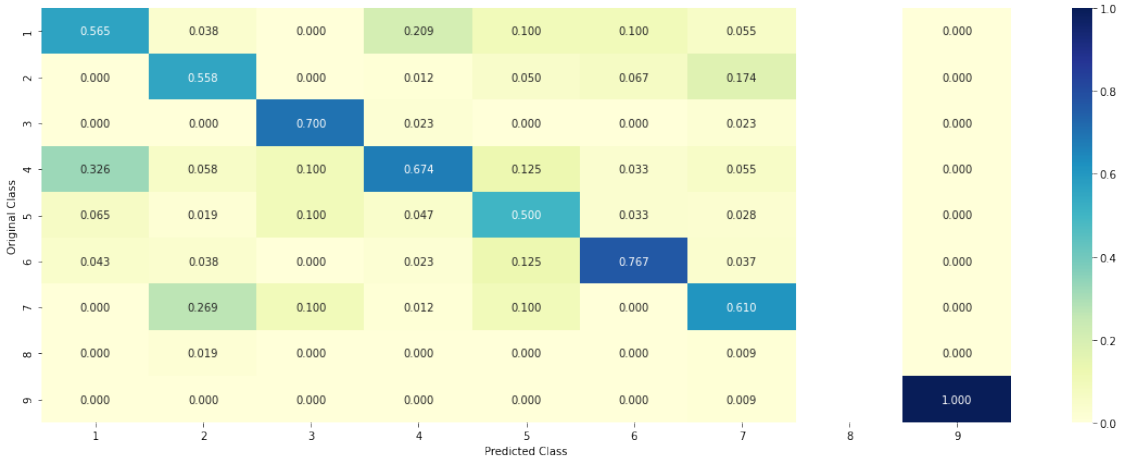

```
For values of best alpha =  0.01 The train log loss is: 0.6707974194294515
For values of best alpha =  0.01 The cross validation log loss is:
1.1356452461453193
For values of best alpha =  0.01 The test log loss is: 1.1239695839909265
```

Testing the model with best hyper paramters

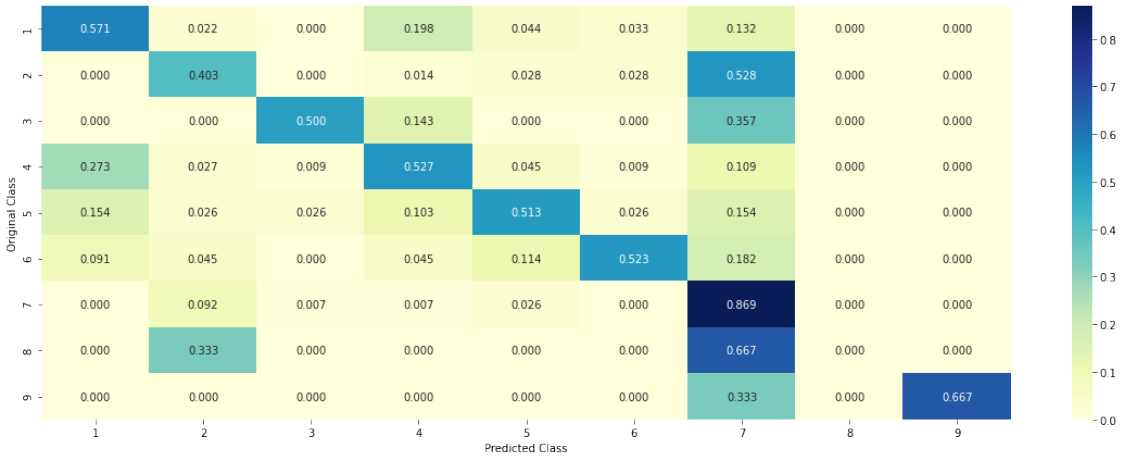
[ ]:
```
# read more about SGDClassifier() at http://scikit-learn.org/stable/modules/
 ↪generated/sklearn.linear_model.SGDClassifier.html
# ------------------------------
# default parameters
# SGDClassifier(loss='hinge', penalty='l2', alpha=0.0001, l1_ratio=0.15,␣
 ↪fit_intercept=True, max_iter=None,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

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

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

```
Log loss : 1.1356452461453193
Number of mis-classified points : 0.38721804511278196
-------------------- Confusion matrix --------------------
```


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



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



Feature Importance

Without Class balancing

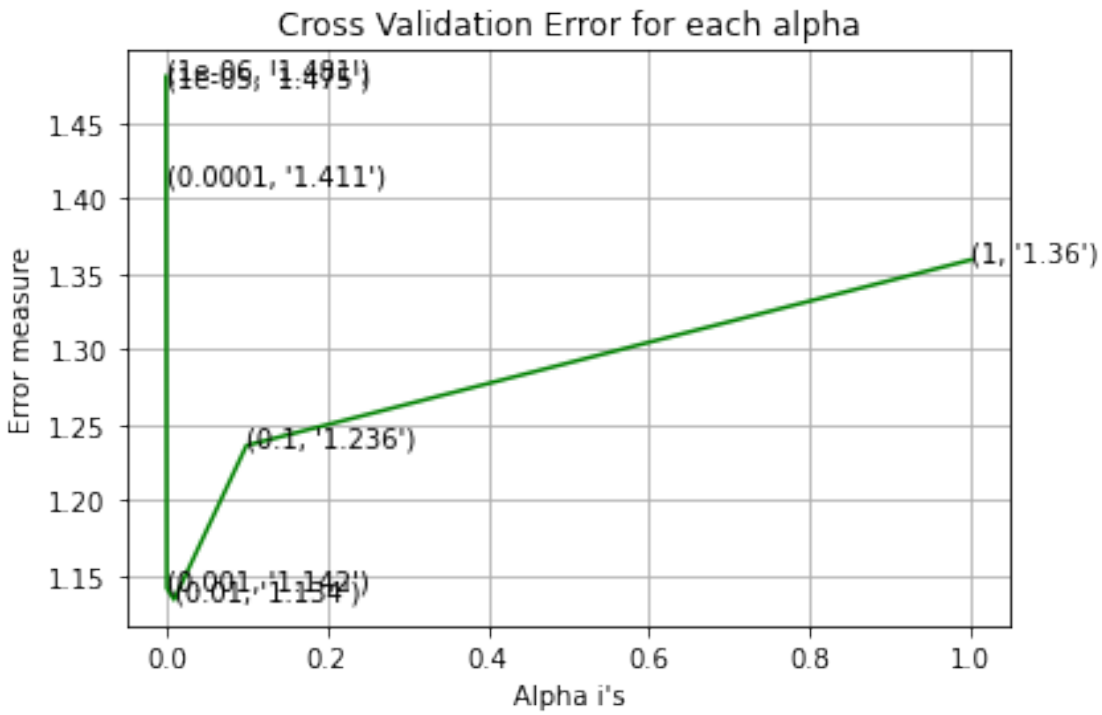
Hyper paramter tuning

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

```
# class_weight=None, warm_start=False, average=False, n_iter=None)

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

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



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

alpha = [10 ** x for x in range(-6, 1)]
cv_log_error_array = []
for i in alpha:
    print("for alpha =", i)
    clf = SGDClassifier(alpha=i, penalty='l2', loss='log', random_state=42,␣
 ↪n_jobs = -1)
    clf.fit(train_x_onehotCoding, train_y)
    sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
    sig_clf.fit(train_x_onehotCoding, train_y)
    sig_clf_probs = sig_clf.predict_proba(cv_x_onehotCoding)
    cv_log_error_array.append(log_loss(cv_y, sig_clf_probs, labels=clf.
 ↪classes_, eps=1e-15))
    print("Log Loss :",log_loss(cv_y, sig_clf_probs))

fig, ax = plt.subplots()
ax.plot(alpha, cv_log_error_array,c='g')
```

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


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

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

```
for alpha = 1e-06
Log Loss : 1.4811120488497715
for alpha = 1e-05
Log Loss : 1.474665666402785
for alpha = 0.0001
Log Loss : 1.4105469520954934
for alpha = 0.001
Log Loss : 1.1419164472697054
for alpha = 0.01
Log Loss : 1.1341243859033256
for alpha = 0.1
Log Loss : 1.2364004678264005
for alpha = 1
Log Loss : 1.3595048642259908
```




```
For values of best alpha =  0.01 The train log loss is: 0.6659518233634023
For values of best alpha =  0.01 The cross validation log loss is:
1.1341243859033256
For values of best alpha =  0.01 The test log loss is: 1.135344306609598
```

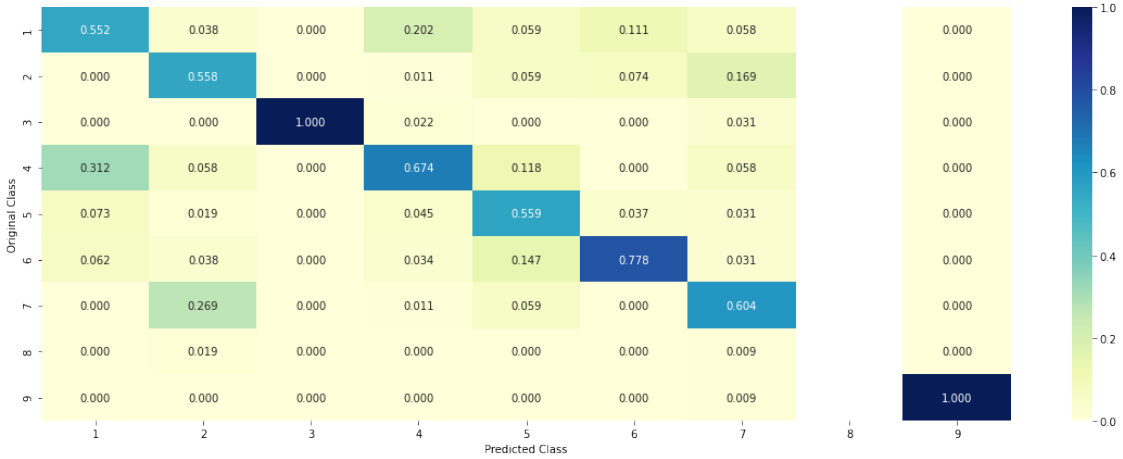
Testing model with best hyper parameters

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

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

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

```
#--------------------------------

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

```
Log loss : 1.1341243859033256
Number of mis-classified points : 0.38533834586466165
-------------------- Confusion matrix --------------------
```


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




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

[ ]:
```
x = PrettyTable()

x.field_names = ["Model", "Train_Log-loss", "Val_Log-loss", "Test_Los-loss",␣
 ↪'Misclassified points(%)']

x.add_row(["Logistic Reg.(weighted)", 0.67, 1.13, 1.12, 0.38])
x.add_row(["Logistic Reg.(non weighted)", 0.66, 1.13, 1.13, 0.38])

print(x)
```

```
+-----------------------------+---------------+-------------+--------------+-
------------------------+
|            Model            | Train_Log-loss | Val_Log-loss | Test_Los-loss |
Misclassified points(%) |
+-----------------------------+---------------+-------------+--------------+-
------------------------+
|   Logistic Reg.(weighted)   |      0.67      |     1.13     |      1.12     |
0.38          |
| Logistic Reg.(non weighted) |      0.66      |     1.13     |      1.13     |
0.38          |
+-----------------------------+---------------+-------------+--------------+-
------------------------+
```

2.4 Task 4 : FE to reduce the log loss of CV and Test below 1.0

[ ]:
```
train_y = y_train
cv_y = y_cv
test_y = y_test

train_df.head()
```

[ ]:

	ID	Gene	…	Class	TEXT
1125	1125	MET	…	7	assumption genes encoding tyrosine kinase rece…
403	403	TP53	…	4	mutation causes inactivation p53 tumor suppres…
3001	3001	KIT	…	7	analyze multi institutional series type c thym…
1575	1575	SDHB	…	4	pheochromocytomas pccs rare tumors arise chrom…
1963	1963	MAPK1	…	7	introduction epidermal growth factor receptor …

[5 rows x 5 columns]

2.5 Univariate analysis

Q3. How to featurize this Gene feature ?

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

One hot Encoding

Response coding

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

T1. One hot encoded features

[ ]:
```
# 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'])
```

[ ]:
```
print("train_gene_feature_onehotCoding is converted feature using one-hot␣
 ↪encoding method. The shape of gene feature:",␣
 ↪train_gene_feature_onehotCoding.shape)
```

```
train_gene_feature_onehotCoding is converted feature using one-hot encoding
method. The shape of gene feature: (2124, 233)
```

[ ]:
```
# one-hot encoding of variation feature.
var_vectorizer = CountVectorizer()

train_variation_feature_onehotCoding  = var_vectorizer.
 ↪fit_transform(train_df['Variation'])
test_variation_feature_onehotCoding  = var_vectorizer.
 ↪transform(test_df['Variation'])
cv_variation_feature_onehotCoding  = var_vectorizer.
 ↪transform(cv_df['Variation'])
```

```
[ ]: print("train_variation_feature_onehotCoding is converted feature using respone␣
     ↪coding method. The shape of gene feature:",␣
     ↪train_variation_feature_onehotCoding.shape)
```

```
train_variation_feature_onehotCoding is converted feature using respone coding
method. The shape of gene feature: (2124, 1947)
```

T2. Response coding

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

```
[ ]: print("train_gene_feature_responseCoding is converted feature using respone␣
     ↪coding method. The shape of gene feature:",␣
     ↪train_gene_feature_responseCoding.shape)
```

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

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

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

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

```
[ ]: #response coding of text features
     train_text_feature_responseCoding  = get_text_responsecoding(train_df)
```

```
test_text_feature_responseCoding  = get_text_responsecoding(test_df)
cv_text_feature_responseCoding  = get_text_responsecoding(cv_df)
```

[ ]:
```
# 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
```

T3. Count/frequency encoding

[ ]:
```
#gene
train_gene_frequency_map = train_df['Gene'].value_counts().to_dict()

train_gene_count_encoding = pd.DataFrame(train_df['Gene'].
 ↪map(train_gene_frequency_map))
test_gene_count_encoding = pd.DataFrame(test_df['Gene'].
 ↪map(train_gene_frequency_map))
cv_gene_count_encoding = pd.DataFrame(cv_df['Gene'].
 ↪map(train_gene_frequency_map))
```

[ ]:
```
test_gene_count_encoding = test_gene_count_encoding.fillna(0)
cv_gene_count_encoding = cv_gene_count_encoding.fillna(0)
```

[ ]:
```
train_gene_count_encoding = train_gene_count_encoding/train_gene_count_encoding.
 ↪max()
test_gene_count_encoding = test_gene_count_encoding/test_gene_count_encoding.
 ↪max()
cv_gene_count_encoding = cv_gene_count_encoding/cv_gene_count_encoding.max()
```

[ ]:
```
train_gene_count_encoding.head()
```

[ ]:
```
          Gene
1125  0.155844
403   0.623377
3001  0.415584
1575  0.006494
1963  0.012987
```

[ ]:
```
# variation
train_variation_frequency_map = train_df['Variation'].value_counts().to_dict()

train_variation_count_encoding = pd.DataFrame(train_df['Variation'].
 ↪map(train_variation_frequency_map))
```

```
test_variation_count_encoding = pd.DataFrame(test_df['Variation'].
 ↪map(train_variation_frequency_map))
cv_variation_count_encoding = pd.DataFrame(cv_df['Variation'].
 ↪map(train_variation_frequency_map))
```

[ ]:
```
test_variation_count_encoding = test_variation_count_encoding.fillna(0)
cv_variation_count_encoding = cv_variation_count_encoding.fillna(0)
```

[ ]:
```
train_variation_count_encoding = train_variation_count_encoding/
 ↪train_variation_count_encoding.max()
test_variation_count_encoding = test_variation_count_encoding/
 ↪test_variation_count_encoding.max()
cv_variation_count_encoding = cv_variation_count_encoding/
 ↪cv_variation_count_encoding.max()

train_variation_count_encoding.head()
```

[ ]:
```
      Variation
1125   0.015385
403    0.015385
3001   0.015385
1575   0.015385
1963   0.830769
```

T4. Length of Text

[ ]:
```
def doc_len(doc):
  text_len = list()
  for i in doc:
    text_len.append(len(i))
  return text_len
```

[ ]:
```
train_text_len = pd.DataFrame(doc_len(train_df.TEXT), columns = ['Text_length'])
test_text_len = pd.DataFrame(doc_len(test_df.TEXT), columns = ['Text_length'])
cv_text_len = pd.DataFrame(doc_len(cv_df.TEXT), columns = ['Text_length'])
```

[ ]:
```
train_text_len = train_text_len/ train_text_len.max()
test_text_len = test_text_len/ test_text_len.max()
cv_text_len = cv_text_len/ cv_text_len.max()

print((train_text_len.shape))
train_text_len.head()
```

```
(2124, 1)
```

[ ]:
```
   Text_length
0     0.099144
1     0.101958
```

```
2     0.716293
3     0.081802
4     0.080302
```

T5. Regex on gene and variation

[ ]:
```
import re

def fetch_numeric(text):
  x = re.findall('[0-9]+', text) # https://pythonexamples.org/
 ↪python-regex-extract-find-all-the-numbers-in-string/
  x = list(map(int, x))
  if(not x):
    return len(text)
  elif(len(x) == 1):
    return (x[0])
  else:
    return sum(x)
```

[ ]:
```
train_variation_numeric_info = np.array(train_df['Variation'].
 ↪map(fetch_numeric)).reshape(-1,1)
train_variation_numeric_info = normalize(train_variation_numeric_info, axis = 0)

test_variation_numeric_info = np.array(test_df['Variation'].map(fetch_numeric)).
 ↪reshape(-1,1)
test_variation_numeric_info = normalize(test_variation_numeric_info, axis = 0)

cv_variation_numeric_info = np.array(cv_df['Variation'].map(fetch_numeric)).
 ↪reshape(-1,1)
cv_variation_numeric_info = normalize(cv_variation_numeric_info, axis = 0)
```

[ ]:
```
train_gene_numeric_info = np.array(train_df['Gene'].map(fetch_numeric)).
 ↪reshape(-1,1)
train_gene_numeric_info = normalize(train_gene_numeric_info, axis = 0)

test_gene_numeric_info = np.array(test_df['Gene'].map(fetch_numeric)).
 ↪reshape(-1,1)
test_gene_numeric_info = normalize(test_gene_numeric_info, axis = 0)

cv_gene_numeric_info = np.array(cv_df['Gene'].map(fetch_numeric)).reshape(-1,1)
cv_gene_numeric_info = normalize(cv_gene_numeric_info, axis = 0)
```

2.5.1 Textual data

[ ]:
```
# building a TfidfVectorizer with all the words that occured minimum 3 times in␣
 ↪train data
text_vectorizer = TfidfVectorizer(min_df=3, max_features= 1000)
```

```
train_text_feature_onehotCoding = text_vectorizer.
 ↪fit_transform(train_df['TEXT'])
# getting all the feature names (words)
train_text_features= text_vectorizer.get_feature_names()

# train_text_feature_onehotCoding.sum(axis=0).A1 will sum every row and returns␣
 ↪(1*number of 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 : 1000
```

[ ]:
```
# 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)
```

2.5.2 Stacking all the features

[ ]:
```
train_x_onehotCoding = hstack((train_text_feature_responseCoding␣
 ↪,train_variation_count_encoding ,train_gene_count_encoding, train_text_len,␣
 ↪train_gene_feature_responseCoding, train_variation_feature_responseCoding,␣
 ↪train_text_feature_onehotCoding)).tocsr()
test_x_onehotCoding = hstack((test_text_feature_responseCoding,␣
 ↪test_variation_count_encoding, test_gene_count_encoding, test_text_len,␣
 ↪test_gene_feature_responseCoding, test_variation_feature_responseCoding,␣
 ↪test_text_feature_onehotCoding)).tocsr()
cv_x_onehotCoding = hstack((cv_text_feature_responseCoding,␣
 ↪cv_variation_count_encoding, cv_gene_count_encoding, cv_text_len,␣
 ↪cv_gene_feature_responseCoding, cv_variation_feature_responseCoding,␣
 ↪cv_text_feature_onehotCoding)).tocsr()
```

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

```
One hot encoding features :
(number of data points * number of features) in train data =  (2124, 1030)
(number of data points * number of features) in test data =  (665, 1030)
(number of data points * number of features) in cross validation data = (532,
1030)
```

2.5.3 Modeling and evaluation

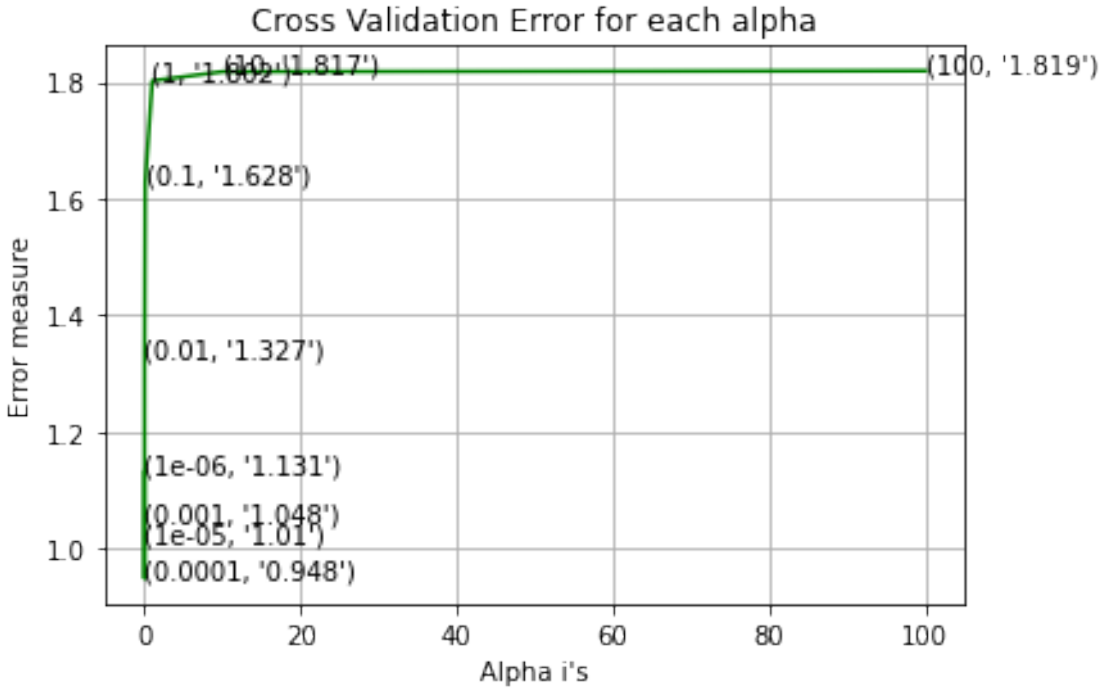
Logistic Regression

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

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

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


# find more about CalibratedClassifierCV here at http://scikit-learn.org/stable/
 ↪modules/generated/sklearn.calibration.CalibratedClassifierCV.html
# ----------------------------
# default paramters
# sklearn.calibration.CalibratedClassifierCV(base_estimator=None,␣
 ↪method='sigmoid', cv=3)
#
# some of the methods of CalibratedClassifierCV()
```

```
# fit(X, y[, sample_weight])        Fit the calibrated model
# get_params([deep])        Get parameters for this estimator.
# predict(X)        Predict the target of new samples.
# predict_proba(X)        Posterior probabilities of classification
#--------------------------------------
# video link:
#--------------------------------------

alpha = [10 ** x for x in range(-6, 3)]
cv_log_error_array = []
for i in alpha:
    print("for alpha =", i)
    clf = SGDClassifier(class_weight='balanced', alpha=i, penalty='l2',␣
 ↪loss='log', random_state=42, n_jobs = -1)
    clf.fit(train_x_onehotCoding, y_train)
    sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
    sig_clf.fit(train_x_onehotCoding, y_train)
    sig_clf_probs = sig_clf.predict_proba(cv_x_onehotCoding)
    cv_log_error_array.append(log_loss(y_cv, 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(y_cv, sig_clf_probs))

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


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

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

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

```
for alpha = 1e-06
Log Loss : 1.1307036310938268
for alpha = 1e-05
Log Loss : 1.0097895072879794
for alpha = 0.0001
Log Loss : 0.9481302110183609
for alpha = 0.001
Log Loss : 1.0476203943212696
for alpha = 0.01
Log Loss : 1.326815839430294
for alpha = 0.1
Log Loss : 1.6277222552170105
for alpha = 1
Log Loss : 1.8016735431547155
for alpha = 10
Log Loss : 1.817246656727863
for alpha = 100
Log Loss : 1.8187550519903541
```

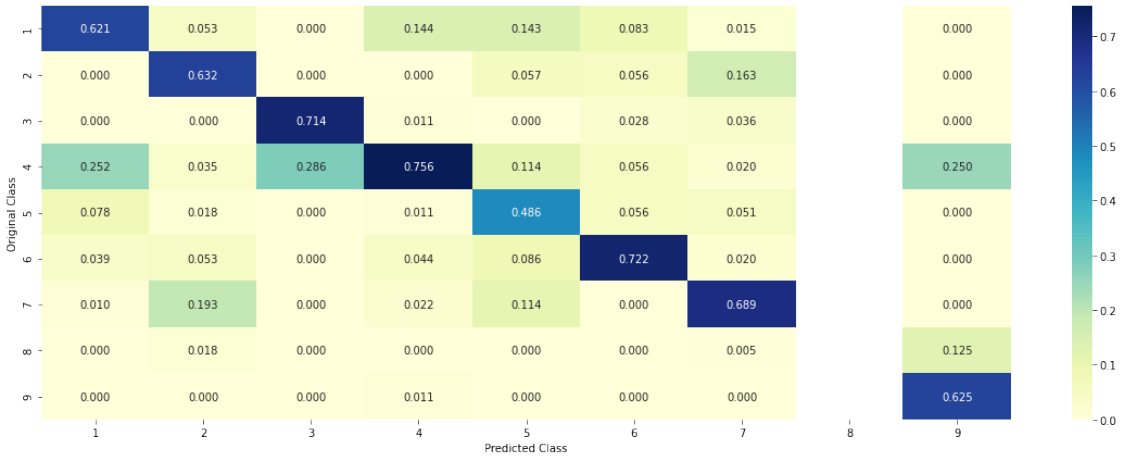


```
For values of best alpha =  0.0001 The train log loss is: 0.6685864734426902
For values of best alpha =  0.0001 The cross validation log loss is:
0.9481302110183609
For values of best alpha =  0.0001 The test log loss is: 0.9850008847439474
```

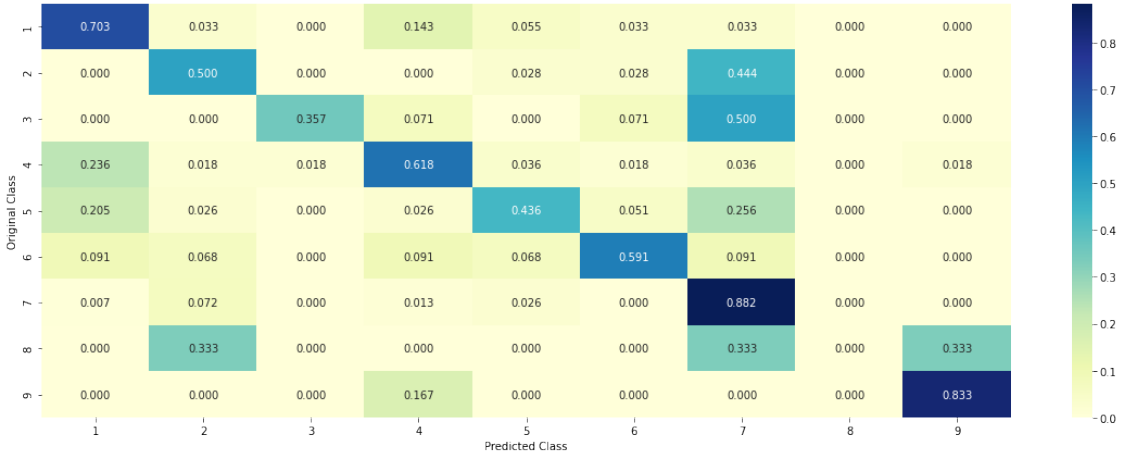
[ ]:
```
# read more about SGDClassifier() at http://scikit-learn.org/stable/modules/
 ↪generated/sklearn.linear_model.SGDClassifier.html
# ------------------------------
# default parameters
# SGDClassifier(loss='hinge', penalty='l2', alpha=0.0001, l1_ratio=0.15,␣
 ↪fit_intercept=True, max_iter=None,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

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

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

```
Log loss : 0.9481302110183609
Number of mis-classified points : 0.3308270676691729
-------------------- Confusion matrix --------------------
```

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




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



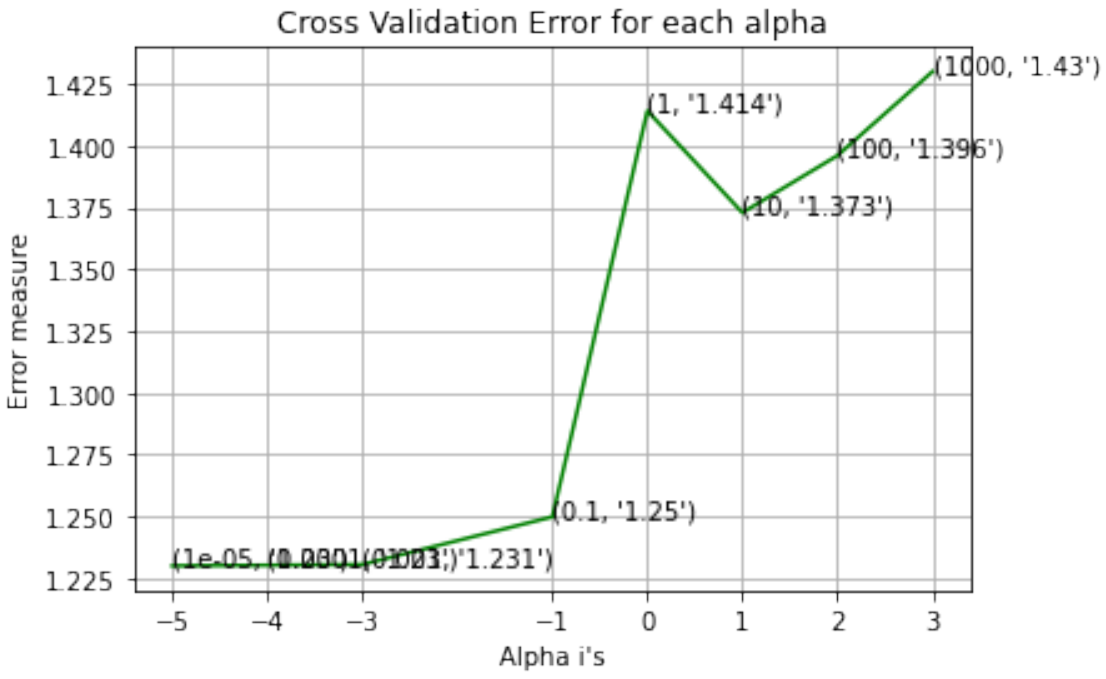
NB

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

# some of methods of MultinomialNB()
# fit(X, y[, sample_weight])        Fit Naive Bayes classifier according to X, y
# predict(X)        Perform classification on an array of test vectors X.
```

```
# predict_log_proba(X)        Return log-probability estimates for the test␣
 ↪vector X.
# ----------------------
# video link: https://www.appliedaicourse.com/course/applied-ai-course-online/
 ↪lessons/naive-bayes-algorithm-1/
# ----------------------


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


alpha = [0.00001, 0.0001, 0.001, 0.1, 1, 10, 100,1000]
cv_log_error_array = []
for i in alpha:
    print("for alpha =", i)
    clf = MultinomialNB(alpha=i)
    clf.fit(train_x_onehotCoding, train_y)
    sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
    sig_clf.fit(train_x_onehotCoding, train_y)
    sig_clf_probs = sig_clf.predict_proba(cv_x_onehotCoding)
    cv_log_error_array.append(log_loss(cv_y, sig_clf_probs, labels=clf.
 ↪classes_, eps=1e-15))
    # to avoid rounding error while multiplying probabilites we use␣
 ↪log-probability estimates
    print("Log Loss :",log_loss(cv_y, sig_clf_probs))

fig, ax = plt.subplots()
ax.plot(np.log10(alpha), cv_log_error_array,c='g')
for i, txt in enumerate(np.round(cv_log_error_array,3)):
    ax.annotate((alpha[i],str(txt)), (np.log10(alpha[i]),cv_log_error_array[i]))
plt.grid()
```

```
plt.xticks(np.log10(alpha))
plt.title("Cross Validation Error for each alpha")
plt.xlabel("Alpha i's")
plt.ylabel("Error measure")
plt.show()


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


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

```
for alpha = 1e-05
Log Loss : 1.2299806687189234
for alpha = 0.0001
Log Loss : 1.2302275015650737
for alpha = 0.001
Log Loss : 1.2305004353369358
for alpha = 0.1
Log Loss : 1.249790047461321
for alpha = 1
Log Loss : 1.4140086299803183
for alpha = 10
Log Loss : 1.3728352596834288
for alpha = 100
Log Loss : 1.3959268573428045
for alpha = 1000
Log Loss : 1.429888684594597
```



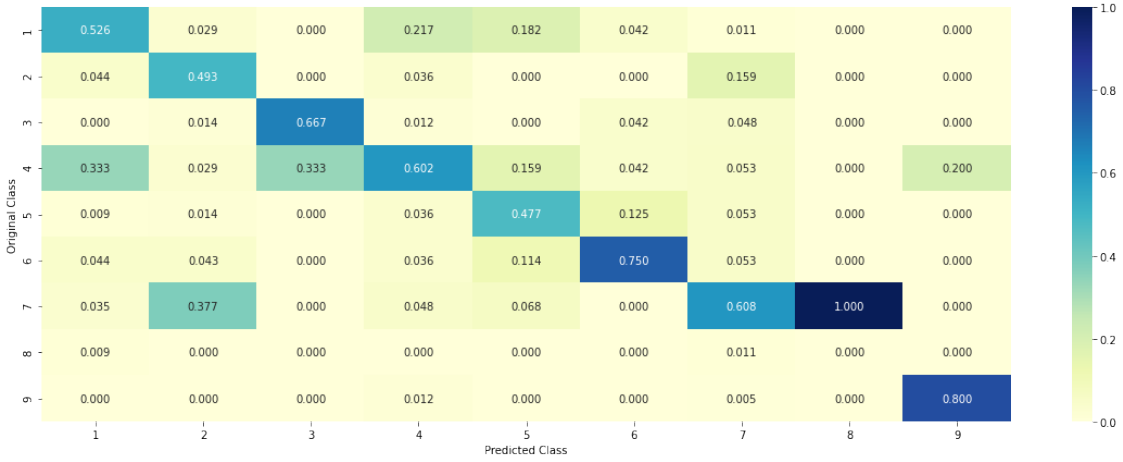

```
For values of best alpha =  1e-05 The train log loss is: 1.1156501777493955
For values of best alpha =  1e-05 The cross validation log loss is:
1.2299806687189234
For values of best alpha =  1e-05 The test log loss is: 1.2821371472684568
```

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

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


# find more about CalibratedClassifierCV here at http://scikit-learn.org/stable/
 ↪modules/generated/sklearn.calibration.CalibratedClassifierCV.html
# ----------------------------
```

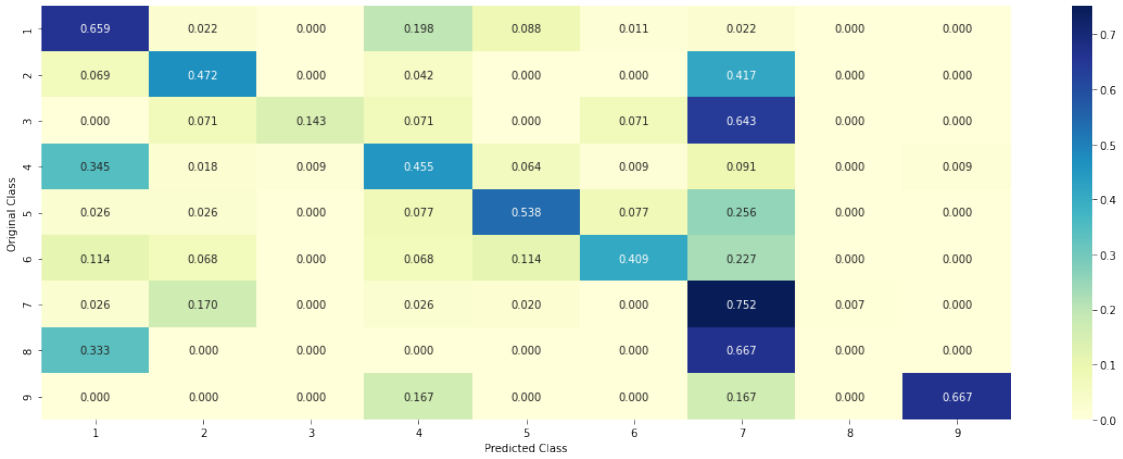
```
# default paramters
# sklearn.calibration.CalibratedClassifierCV(base_estimator=None,␣
 ↪method='sigmoid', cv=3)
#
# some of the methods of CalibratedClassifierCV()
# fit(X, y[, sample_weight])        Fit the calibrated model
# get_params([deep])        Get parameters for this estimator.
# predict(X)        Predict the target of new samples.
# predict_proba(X)        Posterior probabilities of classification
# ----------------------------

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

```
Log Loss : 1.2299806687189234
Number of missclassified point : 0.42857142857142855
-------------------- Confusion matrix --------------------
```


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



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



Random Forest

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

# Some of methods of RandomForestClassifier()
```

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

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

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


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

alpha = [100,200,500,1000]
max_depth = [5, 10]
cv_log_error_array = []
for i in alpha:
    for j in max_depth:
        print("for n_estimators =", i,"and max depth = ", j)
        clf = RandomForestClassifier(n_estimators=i, criterion='gini',␣
 ↪max_depth=j, random_state=42, n_jobs=-1)
        clf.fit(train_x_onehotCoding, train_y)
        sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
        sig_clf.fit(train_x_onehotCoding, train_y)
        sig_clf_probs = sig_clf.predict_proba(cv_x_onehotCoding)
        cv_log_error_array.append(log_loss(cv_y, sig_clf_probs, labels=clf.
 ↪classes_, eps=1e-15))
        print("Log Loss :",log_loss(cv_y, sig_clf_probs))
```

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

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

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

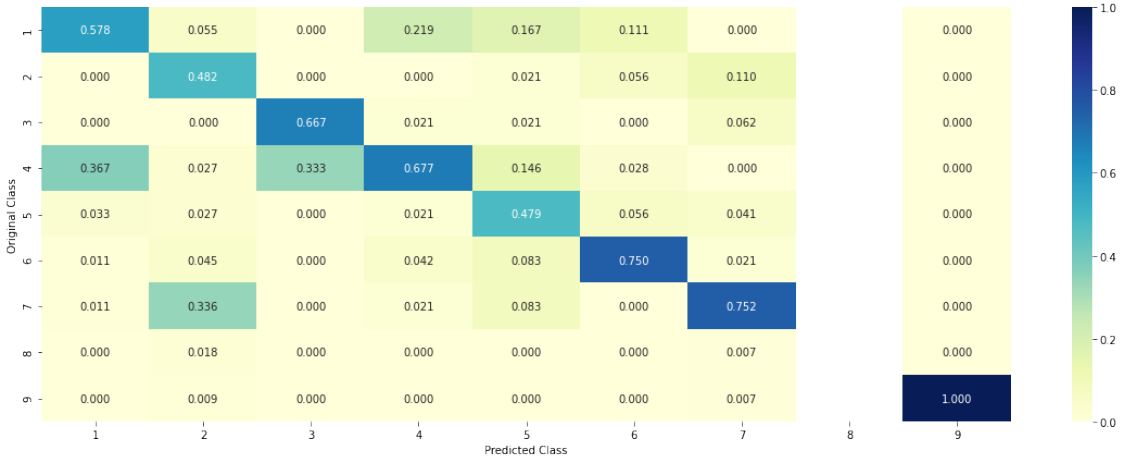
```
for n_estimators = 100 and max depth =  5
Log Loss : 1.0748109965225723
for n_estimators = 100 and max depth =  10
Log Loss : 1.0120018649953078
for n_estimators = 200 and max depth =  5
Log Loss : 1.0639253826714763
for n_estimators = 200 and max depth =  10
Log Loss : 1.004175981377222
for n_estimators = 500 and max depth =  5
Log Loss : 1.0485846751381485
for n_estimators = 500 and max depth =  10
Log Loss : 0.9911662753635907
for n_estimators = 1000 and max depth =  5
Log Loss : 1.0500540447219016
```

```
for n_estimators = 1000 and max depth =  10
Log Loss : 0.9963585287511596
For values of best estimator =  500 The train log loss is: 0.09972923854128404
For values of best estimator =  500 The cross validation log loss is:
0.9911662753635907
For values of best estimator =  500 The test log loss is: 0.9972950218180476
```

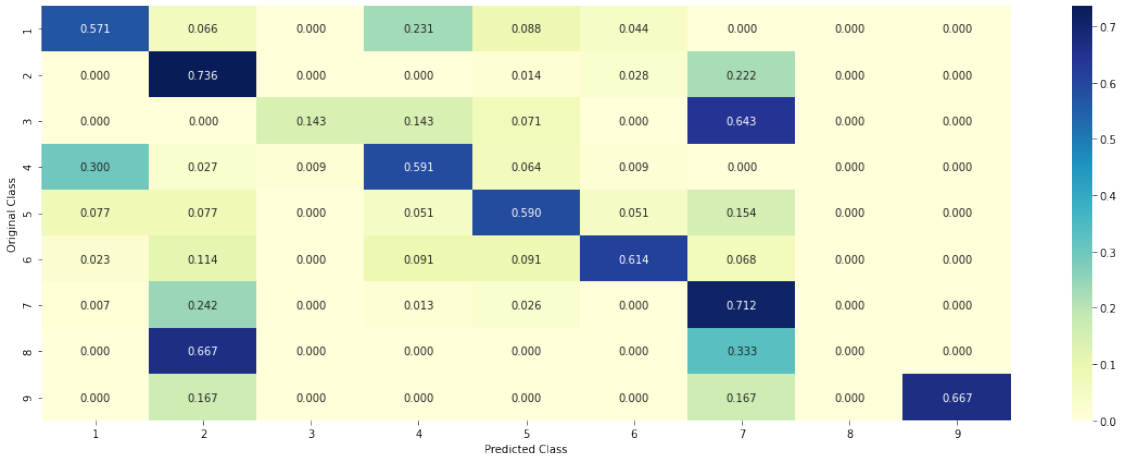
[ ]:
```
# ---------------------------------
# default parameters
# sklearn.ensemble.RandomForestClassifier(n_estimators=10, criterion='gini',␣
 ↪max_depth=None, min_samples_split=2,
# min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features='auto',␣
 ↪max_leaf_nodes=None, min_impurity_decrease=0.0,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

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

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

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

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

```
Log loss : 0.9911662753635907
Number of mis-classified points : 0.37030075187969924
-------------------- Confusion matrix --------------------
```

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



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



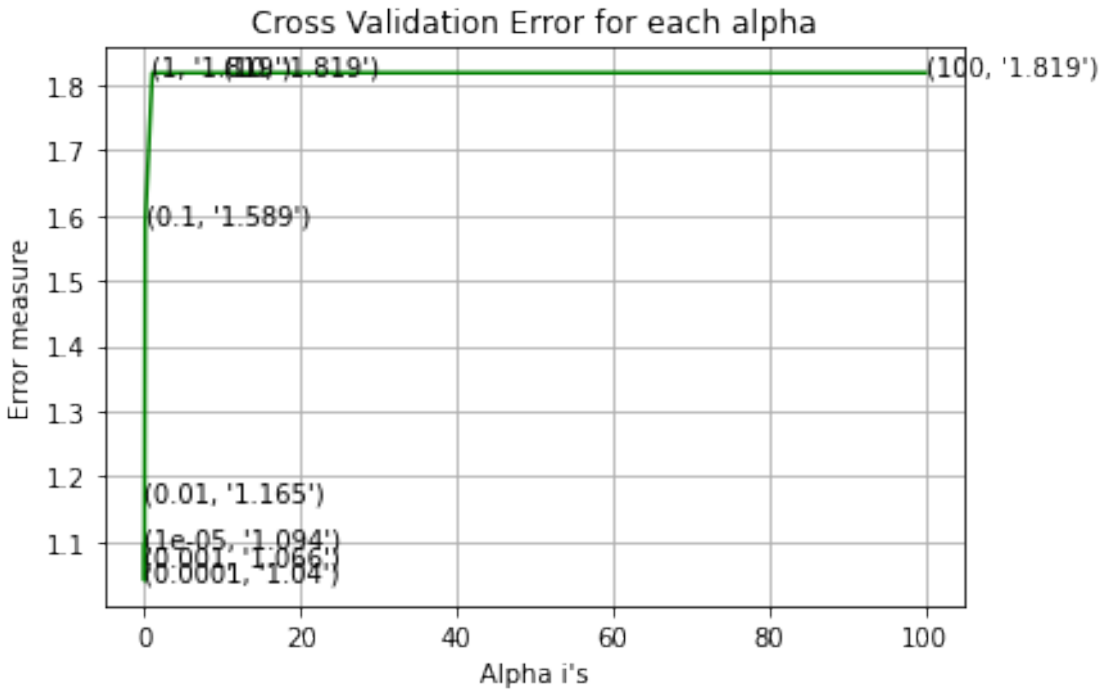
SVM

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

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

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



# find more about CalibratedClassifierCV here at http://scikit-learn.org/stable/
↪modules/generated/sklearn.calibration.CalibratedClassifierCV.html
# ----------------------------
# default paramters
# sklearn.calibration.CalibratedClassifierCV(base_estimator=None,␣
↪method='sigmoid', cv=3)
#
# some of the methods of CalibratedClassifierCV()
```

```
# fit(X, y[, sample_weight])        Fit the calibrated model
# get_params([deep])        Get parameters for this estimator.
# predict(X)        Predict the target of new samples.
# predict_proba(X)        Posterior probabilities of classification
#--------------------------------------
# video link:
#--------------------------------------

alpha = [10 ** x for x in range(-5, 3)]
cv_log_error_array = []
for i in alpha:
    print("for C =", i)
#     clf = SVC(C=i,kernel='linear',probability=True, class_weight='balanced')
    clf = SGDClassifier( class_weight='balanced', alpha=i, penalty='l2',␣
 ↪loss='hinge', random_state=42)
    clf.fit(train_x_onehotCoding, train_y)
    sig_clf = CalibratedClassifierCV(clf, method="sigmoid")
    sig_clf.fit(train_x_onehotCoding, train_y)
    sig_clf_probs = sig_clf.predict_proba(cv_x_onehotCoding)
    cv_log_error_array.append(log_loss(cv_y, sig_clf_probs, labels=clf.
 ↪classes_, eps=1e-15))
    print("Log Loss :",log_loss(cv_y, sig_clf_probs))

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


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

predict_y = sig_clf.predict_proba(train_x_onehotCoding)
print('For values of best alpha = ', alpha[best_alpha], "The train log loss is:
 ↪",log_loss(y_train, predict_y, labels=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 C = 1e-05
Log Loss : 1.0941274574727036
for C = 0.0001
Log Loss : 1.040369880722636
for C = 0.001
Log Loss : 1.0656076496575875
for C = 0.01
Log Loss : 1.164820982256052
for C = 0.1
Log Loss : 1.5894910213769726
for C = 1
Log Loss : 1.818999684619598
for C = 10
Log Loss : 1.8190005022065758
for C = 100
Log Loss : 1.819000216097014
```



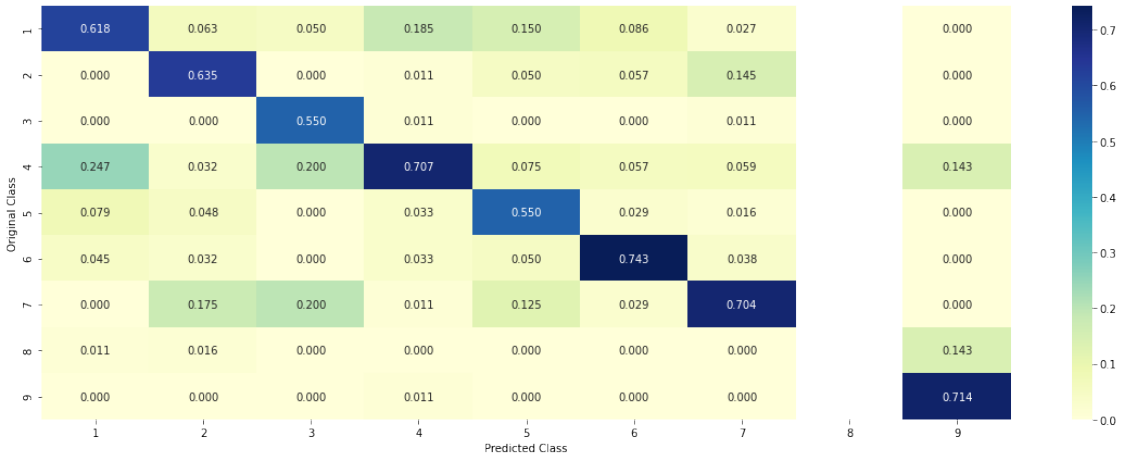

```
For values of best alpha =  0.0001 The train log loss is: 0.6219930162993779
For values of best alpha =  0.0001 The cross validation log loss is:
```

```
1.040369880722636
For values of best alpha =  0.0001 The test log loss is: 1.0758762577480732
```

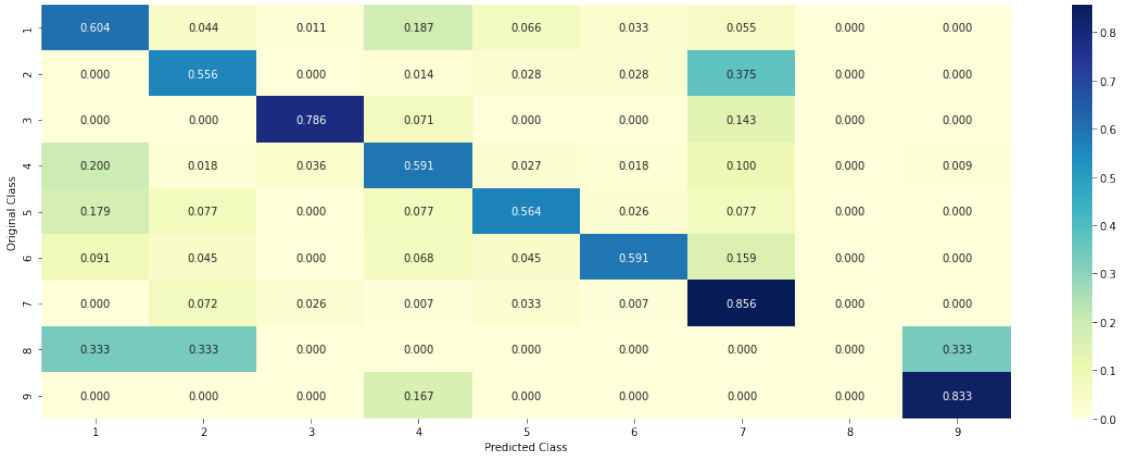
[ ]:
```
# read more about support vector machines with linear kernals here http://
 ↪scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

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

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


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

```
Log loss : 1.040369880722636
Number of mis-classified points : 0.33270676691729323
-------------------- Confusion matrix --------------------
```

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




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



Stacking

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

```
# class_weight=None, warm_start=False, average=False, n_iter=None)

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

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


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

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


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

# Some of methods of RandomForestClassifier()
```

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

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

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


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

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

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))))

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],␣
 ↪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
```

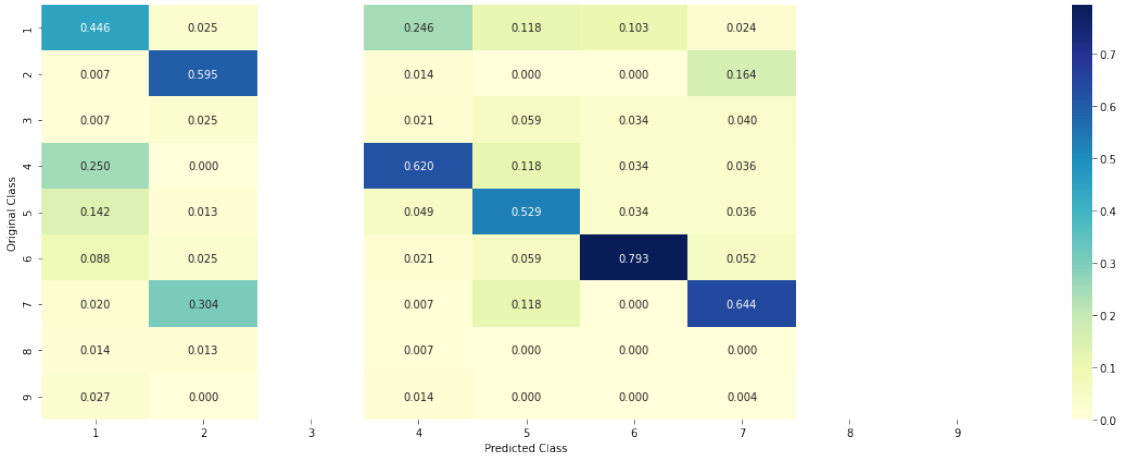
```
Logistic Regression :  Log Loss: 1.05
```

```
Support vector machines : Log Loss: 1.82
--------------------------------------------------
Stacking Classifer : for the value of alpha: 0.000100 Log Loss: 1.824
Stacking Classifer : for the value of alpha: 0.001000 Log Loss: 1.772
Stacking Classifer : for the value of alpha: 0.010000 Log Loss: 1.492
Stacking Classifer : for the value of alpha: 0.100000 Log Loss: 1.154
Stacking Classifer : for the value of alpha: 1.000000 Log Loss: 1.085
Stacking Classifer : for the value of alpha: 10.000000 Log Loss: 1.115
```

[ ]:
```
lr = LogisticRegression(C=0.1)
sclf = StackingClassifier(classifiers=[sig_clf1, sig_clf2], 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 : 1.0775791597411297
Log loss (CV) on the stacking classifier : 1.1535208349939163
Log loss (test) on the stacking classifier : 1.162203364703595
Number of missclassified point : 0.4075187969924812
-------------------- Confusion matrix --------------------
```

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




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

Maximum Voting classifier

[ ]:
```
clf1 = SGDClassifier(alpha=0.0001, penalty='l2', loss='log',␣
 ↪class_weight='balanced', random_state=0, n_jobs = -1)
clf1.fit(train_x_onehotCoding, y_train)
sig_clf1 = CalibratedClassifierCV(clf1, method="sigmoid")

clf2 = MultinomialNB(alpha=0.00001)
clf2.fit(train_x_onehotCoding, y_train)
sig_clf2 = CalibratedClassifierCV(clf2, method="sigmoid")
```

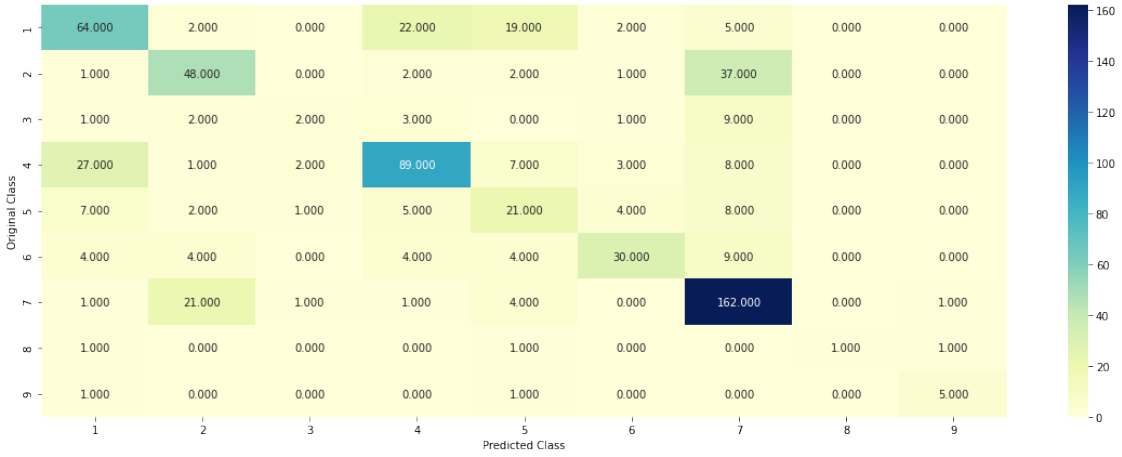
```
clf3 = SGDClassifier(alpha=0.0001, penalty='l2', loss='hinge',␣
 ↪random_state=42,class_weight='balanced')
clf3.fit(train_x_onehotCoding, y_train)
sig_clf3 = CalibratedClassifierCV(clf3, method="sigmoid")

clf4 = RandomForestClassifier(n_estimators=500, criterion='gini', max_depth=10,␣
 ↪n_jobs=-1)
clf4.fit(train_x_onehotCoding, y_train)
sig_clf4 = CalibratedClassifierCV(clf4, method="sigmoid")

sig_clf1.fit(train_x_onehotCoding, y_train)
print("Logistic Regression :  Log Loss: %0.2f" % (log_loss(y_cv, sig_clf1.
 ↪predict_proba(cv_x_onehotCoding))))
sig_clf2.fit(train_x_onehotCoding, y_train)
print("NB : Log Loss: %0.2f" % (log_loss(y_cv, sig_clf2.
 ↪predict_proba(cv_x_onehotCoding))))
sig_clf3.fit(train_x_onehotCoding, y_train)
print("SVM : Log Loss: %0.2f" % (log_loss(y_cv, sig_clf3.
 ↪predict_proba(cv_x_onehotCoding))))
sig_clf4.fit(train_x_onehotCoding, y_train)
print("RF : Log Loss: %0.2f" % (log_loss(y_cv, sig_clf4.
 ↪predict_proba(cv_x_onehotCoding))))
```

```
Logistic Regression :  Log Loss: 0.95
NB : Log Loss: 1.23
SVM : Log Loss: 1.04
RF : Log Loss: 0.98
```

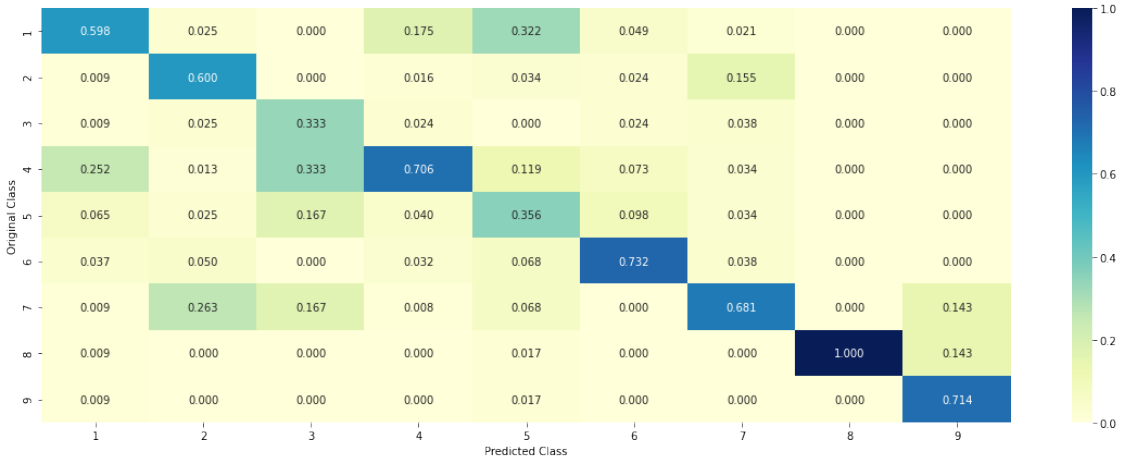
[ ]:
```
#Refer:http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.
 ↪VotingClassifier.html
from sklearn.ensemble import VotingClassifier
vclf = VotingClassifier(estimators=[('lr', sig_clf1), ('nb', sig_clf2), ('svm',␣
 ↪sig_clf3)], voting='soft')
vclf.fit(train_x_onehotCoding, y_train)
print("Log loss (train) on the VotingClassifier :", log_loss(y_train, vclf.
 ↪predict_proba(train_x_onehotCoding)))
print("Log loss (CV) on the VotingClassifier :", log_loss(y_cv, vclf.
 ↪predict_proba(cv_x_onehotCoding)))
print("Log loss (test) on the VotingClassifier :", log_loss(y_test, vclf.
 ↪predict_proba(test_x_onehotCoding)))
print("Number of missclassified point :", np.count_nonzero((vclf.
 ↪predict(test_x_onehotCoding)- y_test))/y_test.shape[0])
plot_confusion_matrix(test_y=y_test, predict_y=vclf.
 ↪predict(test_x_onehotCoding))
```

```
Log loss (train) on the VotingClassifier : 0.7407640399050727
Log loss (CV) on the VotingClassifier : 0.9751029837217141
```

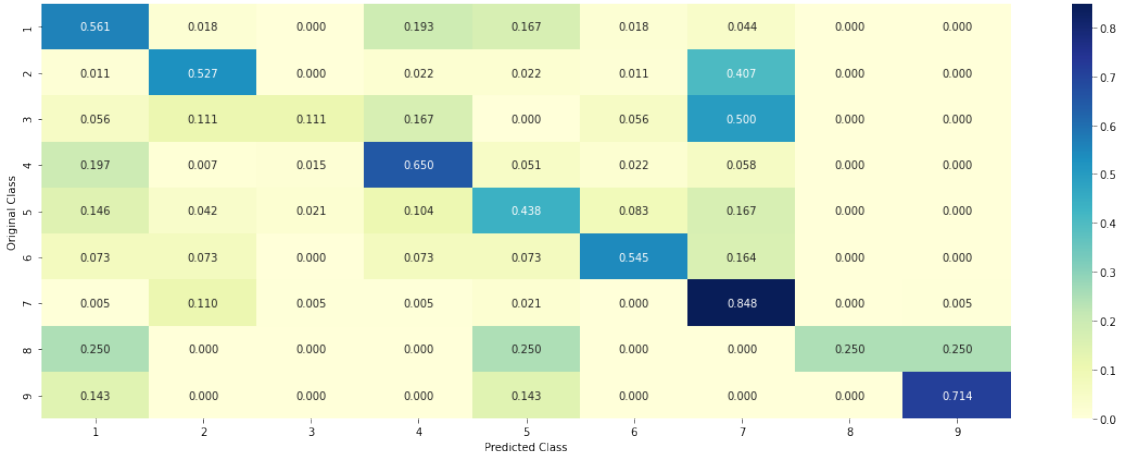
```
Log loss (test) on the VotingClassifier : 1.0140471844895755
Number of missclassified point : 0.36541353383458647
-------------------- Confusion matrix --------------------
```




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




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



Conclusion

[ ]:
```
from prettytable import PrettyTable

x = PrettyTable()

x.field_names = ['Model' ,"Train_Log-loss", "Val_Log-loss", "Test_Los-loss",␣
 ↪'Misclassified points(%)']

x.add_row([ "Logistic Reg", 0.67, 0.95, 0.98, 0.33])

print(x)
```

```
+--------------+----------------+--------------+---------------+----------------
---------+
|    Model     | Train_Log-loss | Val_Log-loss | Test_Los-loss | Misclassified
points(%) |
+--------------+----------------+--------------+---------------+----------------
---------+
| Logistic Reg |      0.67      |     0.95     |      0.98     |           0.33
|
+--------------+----------------+--------------+---------------+----------------
---------+
```