[ ]:
```
from google.colab import drive
drive.mount("/content/gdrive")
```

[ ]:
```
import pandas as pd
import numpy as np
from nltk.corpus import stopwords
import nltk
nltk.download('stopwords')
import warnings
warnings.filterwarnings('ignore')
import re
import time
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix , log_loss
import seaborn as sns
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.linear_model import SGDClassifier
from sklearn.calibration import CalibratedClassifierCV
import math
from collections import Counter , defaultdict
from sklearn.preprocessing import normalize
from scipy.sparse import hstack
from scipy.sparse import csr_matrix
from sklearn.naive_bayes import MultinomialNB
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier
import sys   # for stacking
import six   # for stacking
sys.modules['sklearn.externals.six'] = six
from mlxtend.classifier import StackingClassifier
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data…
[nltk_data]   Package stopwords is already up-to-date!
```

Importing Gene, Variation & Class data

[ ]:
```
df_gene_var = pd.read_csv("/content/gdrive/My Drive/Cancer Prediction/
  ↪training_variants")
print("Number of data points: ",df_gene_var.shape[0])
print("Number of Features: ",df_gene_var.shape[1])
print("Features: ", df_gene_var.columns.values)
df_gene_var.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

Importing Text data

[ ]:
```
df_text = pd.read_csv("/content/gdrive/MyDrive/Cancer Prediction/
 ↪training_text", sep='\|\|', engine='python', names=['ID', 'TEXT'] ,␣
 ↪skiprows=1)
print("Number of data points: ",df_text.shape[0])
print("Number of Features: ",df_text.shape[1])
print("Features: ",df_text.columns.values)
df_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…

Some little preprocessing

[ ]:
```
# importing stopwords
stop_words = set(stopwords.words('english'))
# creating function for preprocessing text
def nlp_preprocessing(total_text, index, column):
  if type(total_text) is not int:
    string = ""
    # replace every special chracter with splace
    total_text = re.sub('[^a-zA-Z0-9\n]',' ',total_text)
    # replace multiple spaces with single space
    total_text = re.sub('\s+',' ', total_text)
    # lower case the text
    total_text = total_text.lower()

    # Take the sentence and change the o/p to words
    # split() function working---> I/P "welcome to the jungle"    O/
 ↪P['welcome', 'to', 'the', 'jungle']
    for word in total_text.split():
      if word not in stop_words:
```

```
        string += word + ' '
    df_text[column][index] = string
```

[ ]:
```
#
start_time = time.clock()
for index, row in df_text.iterrows():
  if type(row['TEXT']) is str:
    nlp_preprocessing(row['TEXT'],index,'TEXT')
  else:
    print("There is no Description for Index: ",index)
print("Total Time Taken for nlp preprocessing: ",time.clock() - start_time,"␣
 ↪seconds")
```

```
There is no Description for Index:  1109
There is no Description for Index:  1277
There is no Description for Index:  1407
There is no Description for Index:  1639
There is no Description for Index:  2755
Total Time Taken for nlp preprocessing:  41.555978  seconds
```

Merging both gene, Variation and Text dataset

[ ]:
```
result = pd.merge(df_gene_var, df_text, on='ID', how='left')
print("Shape of resultant data: ",result.shape)
result.head()
```

```
Shape of resultant data:  (3321, 5)
```

[ ]:

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

[ ]:
```
#Finding the null values
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
```

[ ]:
```
# Now adding gene and variation inplace of null text
result.loc[result['TEXT'].isnull(),'TEXT'] = result['Gene'] + " " +␣
 ↪result['Variation']
result[result['ID'] == 1109]
# result['ID'] == 1109  df inside df
# 0       False
# 1       False
# 2       False
# 3       False
# 4       False

#       ...
# 3316    False
# 3317    False
# 3318    False
# 3319    False
# 3320    False
# Name: ID, Length: 3321, dtype: bool
```

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

Splitting of data for Train, Cross Validate and Test.

[ ]:
```
y = result['Class']
#result = result.drop('Class',axis=1)
# 80% Train data and 20% Test data
x_train1, x_test, y_train1, y_test = train_test_split(result , y , stratify= y␣
 ↪, test_size= 0.2)
# 80% Train data and 20% Test data
x_train, x_cv, y_train, y_cv = train_test_split(x_train1, y_train1, stratify=␣
 ↪y_train1, test_size=0.2)
```

[ ]:
```
x_train.head(2)
```

[ ]:
```
        ID   Gene Variation  Class  \
1661  1661   FLT3     R834Q      7
1847  1847  PPP6C     S270L      4

                                                   TEXT
1661  mutations juxtamembrane kinase domains flt3 co…
1847  mutations ppp6c catalytic subunit protein phos…
```

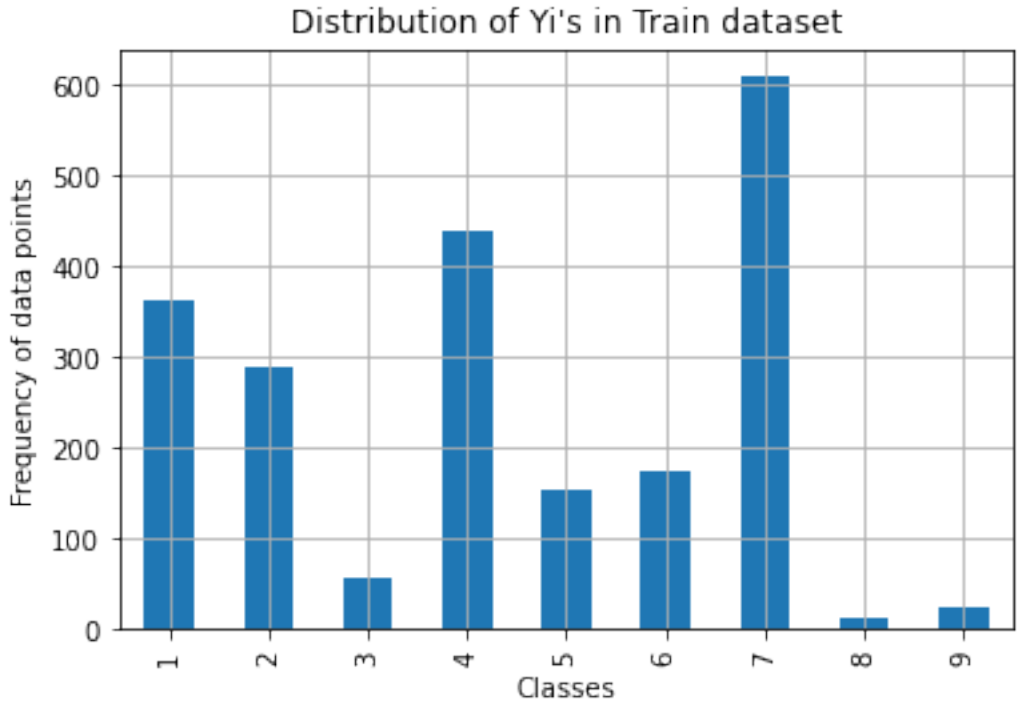
[ ]:
```
print("Number of data point in train set: ",x_train.shape[0])
print("Number of data point in Cross Validate set: ",x_cv.shape[0])
print("Number of data point in Test set: ",x_test.shape[0])
```

```
Number of data point in train set:  2124
Number of data point in Cross Validate set:  532
Number of data point in Test set:  665
```

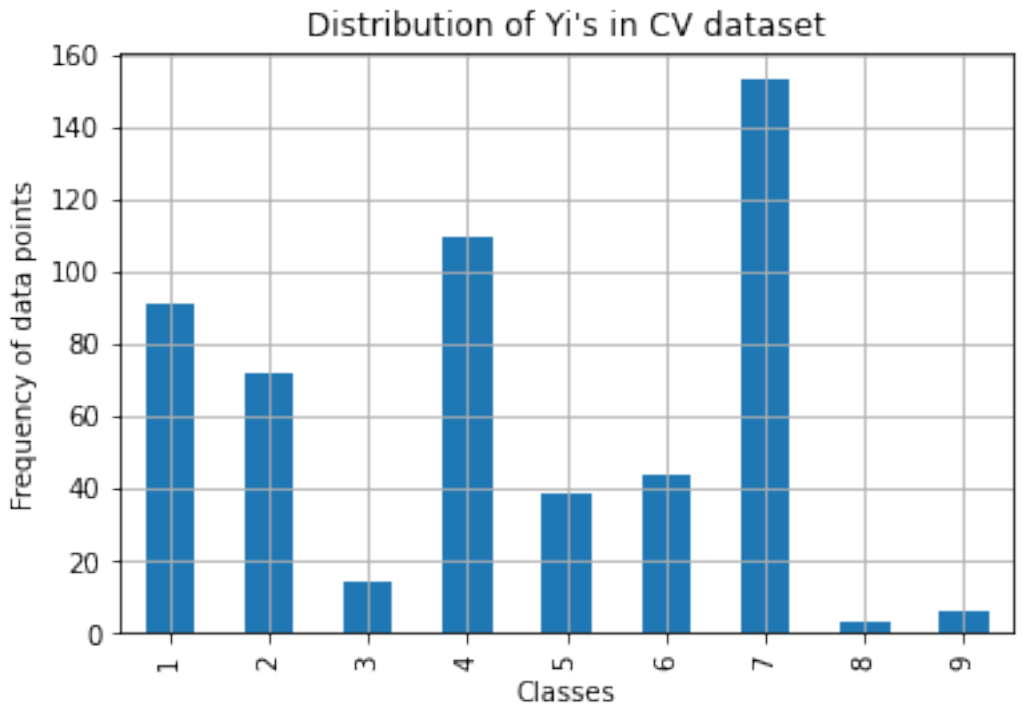
Now it's time to analyse the Distribution of data.

[ ]:
```
train_class_distribution = x_train['Class'].value_counts().sort_index()
cv_class_distribution    = x_cv['Class'].value_counts().sort_index()
test_class_distribution  = x_test['Class'].value_counts().sort_index()


train_class_distribution.plot(kind='bar')
plt.xlabel('Classes')
plt.ylabel('Frequency of data points')
plt.title("Distribution of Yi's in Train dataset")
plt.grid()
plt.show()

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


print("-"*150)
cv_class_distribution.plot(kind='bar')
plt.xlabel("Classes")
plt.ylabel("Frequency of data points")
plt.title("Distribution of Yi's in CV dataset")
plt.grid()
plt.show()

sorted_y = np.argsort(-cv_class_distribution.values)
for i in sorted_y:
  print("Number of data points in class ",i+1,":",cv_class_distribution.
 ↪values[i],"(",np.round(cv_class_distribution.values[i]/y_cv.
 ↪shape[0]*100,3),"%)")


print("-"*150)
cv_class_distribution.plot(kind='bar')
plt.xlabel("Classes")
plt.ylabel("Frequency of data points")
```

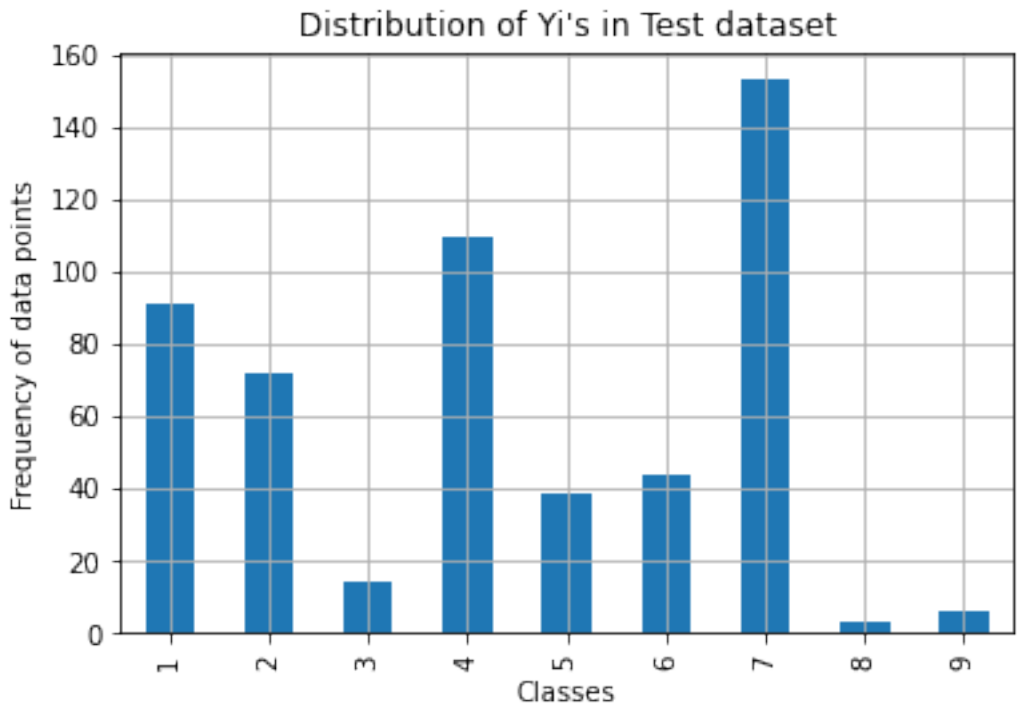
```
plt.title("Distribution of Yi's in Test dataset")
plt.grid()
plt.show()

sorted_y = np.argsort(-cv_class_distribution.values)
for i in sorted_y:
  print("Number of data points in class ",i+1,":",test_class_distribution.
  ↪values[i],"(",np.round(test_class_distribution.values[i]/y_test.
  ↪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 : 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 %)
--------------------------------------------------------------------------------
--------------------------------------------------------------------
```




```
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 %)
```

By observing Gene feature we observe that their train, cv, and test dataset follows similar distribution which is good.

[ ]:
```
def plot_confusion_matrix(y_test , predicted_y):
  C = confusion_matrix(y_test , predicted_y)
  A = (((C.T)/(C.sum(axis=1))).T)    # precison column sum
  B = (C/C.sum(axis=0))        # recall   row sum

  label = [1,2,3,4,5,6,7,8,9]
  plt.figure(figsize=(20,7))
  sns.heatmap(C, annot=True, cmap="YlGnBu", fmt="0.3f", xticklabels=label,␣
 ↪yticklabels=label)
  print("-"*60, "Confusion matrix", "-"*60)
  plt.xlabel("Predicted Class")
```

```
  plt.ylabel("Original Class")
  plt.show()

  plt.figure(figsize=(20,7))
  sns.heatmap(A, fmt='0.3f', annot=True, cmap="YlGnBu" , xticklabels=label ,␣
 ↪yticklabels=label)
  print("-"*60,"Precision Martix axis=1","-"*60)
  plt.xlabel("Predicted Class")
  plt.ylabel("Original Class")
  plt.show()

  plt.figure(figsize=(20,7))
  sns.heatmap(B, fmt='0.3f', annot=True, cmap="YlGnBu" , xticklabels=label ,␣
 ↪yticklabels=label)
  print("-"*60,"Recall Martix axis=0","-"*60)
  plt.xlabel("Predicted Class")
  plt.ylabel("Original Class")
  plt.show()
```

[ ]:
```
train_data_len = x_train.shape[0]  # length is 2124
cv_data_len    = x_cv.shape[0]     # length is 532
test_data_len  = x_test.shape[0]   # length is 665
# we need to generate random 9 random numbers and their sum should be 1.
# One solution is to generate 9 numbers and divide each number with their sum
train_predict_y = np.zeros((train_data_len,9))
for i in range(train_data_len):
  rand_probs = np.random.rand(1,9)
  train_predict_y[i] = ((rand_probs/sum(sum(rand_probs)))[0])
print("logloss for train dataset is: ", log_loss(y_train , train_predict_y,␣
 ↪eps=1e-15))

cv_predict_y = np.zeros((cv_data_len,9))
for i in range(cv_data_len):
  rand_probs = np.random.rand(1,9)
  cv_predict_y[i] = ((rand_probs/sum(sum(rand_probs)))[0])
print("Logloss for Cross Validate dataset is: ",log_loss(y_cv , cv_predict_y,␣
 ↪eps=1e-15))

test_predict_y = np.zeros((test_data_len,9))
for i in range(test_data_len):
  rand_probs = np.random.rand(1,9)
  test_predict_y[i] = ((rand_probs/sum(sum(rand_probs)))[0])
print("Logloss for test dataset is: ",log_loss(y_test , test_predict_y ,␣
 ↪eps=1e-15))
predicted_y = np.argmax(test_predict_y,axis=1)
plot_confusion_matrix(y_test , predicted_y+1)
```

```
logloss for train dataset is:  2.470811804229691
Logloss for Cross Validate dataset is:  2.476160510248127
Logloss for test dataset is:  2.4610290188282096
------------------------------------------------------------ Confusion matrix
------------------------------------------------------------
```


```
------------------------------------------------------------ Precision Martix
axis=1 ------------------------------------------------------------
```


```
------------------------------------------------------------ Recall Martix
axis=0 ------------------------------------------------------------
```

[ ]:
```
# Function for response coding
def get_gv_feat_dict(alpha, feature,df):
  value_count = x_train[feature].value_counts()
  gv_dict=dict()
  for i, denominator in value_count.items():
    vec = []
    for k in range(1,10):
      cls_cnt = x_train.loc[(x_train['Class']==k) & (x_train[feature]==i)]
      vec.append((cls_cnt.shape[0] + alpha*10) / (denominator + alpha*90))
    gv_dict[i] = vec
  return gv_dict

def get_gv_feature(alpha, feature, df):
  gv_dict     = get_gv_feat_dict(alpha, feature, df)
  value_count = x_train[feature].value_counts()
  gv_feat=[]
  for index,row in df.iterrows():
    if row[feature] in dict(value_count).keys():
      gv_feat.append(gv_dict[row[feature]])
    else:
      gv_feat.append([1/9, 1/9, 1/9, 1/9, 1/9, 1/9, 1/9, 1/9, 1/9])
  return gv_feat
```

Univarient Analysis of Gene Feature

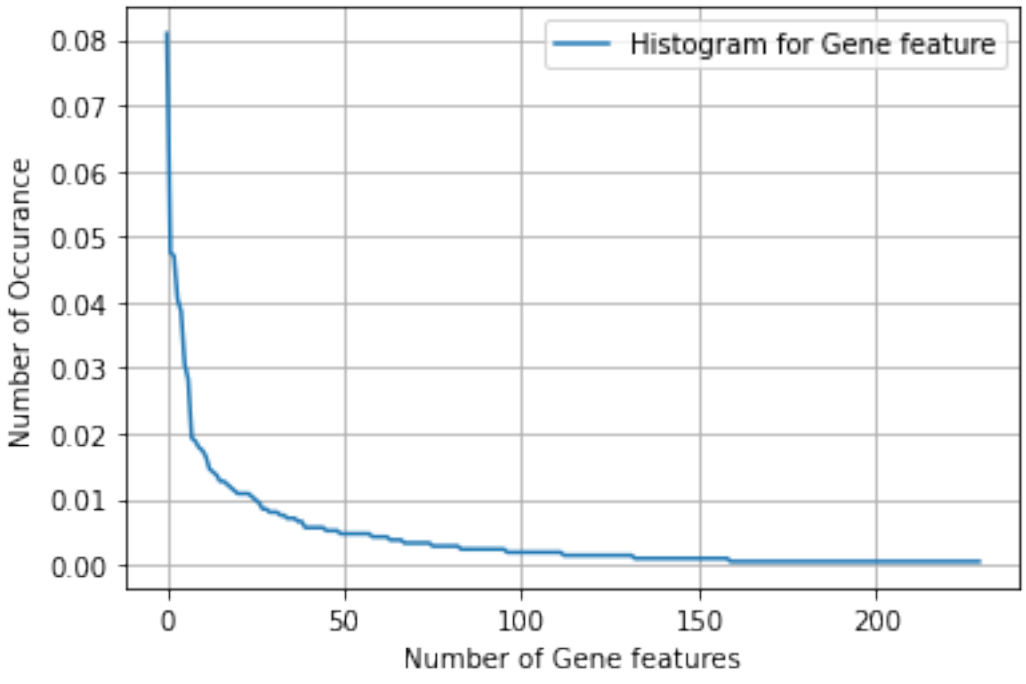
[ ]:
```
unique_gene = x_train['Gene'].value_counts()
print("Number of unique genes: ",unique_gene.shape[0])
print("Top 10 Gene features in train dataset: ")
top_10_g = x_train['Gene'].value_counts()
top_10_g.head(10)
```

```
Number of unique genes:  230
```

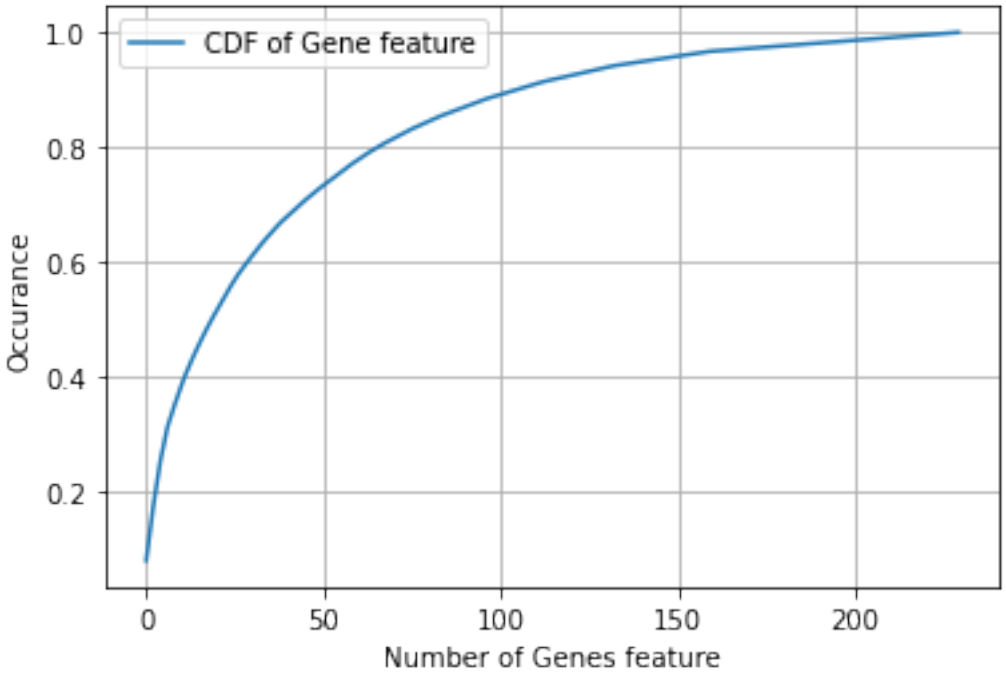
Top 10 Gene features in train dataset:

[ ]:
```
BRCA1     172
EGFR      101
TP53      100
PTEN       86
BRCA2      82
KIT        65
BRAF       60
PDGFRA     41
ALK        40
ERBB2      38
Name: Gene, dtype: int64
```

[ ]:
```
# plotting histogram for Gene feature
s = sum(unique_gene.values)
h = unique_gene.values/s
plt.plot(h,label="Histogram for Gene feature")
plt.xlabel("Number of Gene features")
plt.ylabel("Number of Occurance")
plt.grid()
plt.legend()
plt.show()
```



[ ]:
```
c = np.cumsum(h)
plt.plot(c, label="CDF of Gene feature")
plt.legend()
plt.xlabel("Number of Genes feature")
plt.ylabel("Occurance")
plt.grid()
plt.show()
```



[ ]:
```
alpha = 1
train_gene_ResponseCoding = np.array(get_gv_feature(alpha,'Gene', x_train))
cv_gene_ResponseCoding    = np.array(get_gv_feature(alpha,'Gene', x_cv))
test_gene_ResponseCoding  = np.array(get_gv_feature(alpha, 'Gene', x_test))
print("train_gene_feature_responseCoding is converted feature using respone␣
 ↪coding method. The shape of gene feature:", train_gene_ResponseCoding.shape)
print("cv_gene_feature_responseCoding is converted feature using respone coding␣
 ↪method. The shape of gene feature:", cv_gene_ResponseCoding.shape)
print("testn_gene_feature_responseCoding is converted feature using respone␣
 ↪coding method. The shape of gene feature:", test_gene_ResponseCoding.shape)
```

```
train_gene_feature_responseCoding is converted feature using respone coding
method. The shape of gene feature: (2124, 9)
cv_gene_feature_responseCoding is converted feature using respone coding method.
The shape of gene feature: (532, 9)
testn_gene_feature_responseCoding is converted feature using respone coding
method. The shape of gene feature: (665, 9)
```

[ ]:
```
gene_encoding = CountVectorizer()
train_gene_onehotencode  = gene_encoding.fit_transform(x_train['Gene'])  # One␣
 ↪time fit
cv_gene_onehotencode     = gene_encoding.transform(x_cv['Gene'])    # No need␣
 ↪to fit
test_gene_onehotencode = gene_encoding.transform(x_test['Gene'])
print("train_gene_onehotencoding is converted feature using onehot encoding␣
 ↪method. The shape of gene feature:", train_gene_onehotencode.shape)
print("cv_gene_onehotencoding is converted feature using onehot encoding method.
 ↪ The shape of gene feature:", cv_gene_onehotencode.shape)
print("test_gene_onehotencoding is converted feature using onehot encoding␣
 ↪method. The shape of gene feature:", test_gene_onehotencode.shape)
```

```
train_gene_onehotencoding is converted feature using onehot encoding method. The
shape of gene feature: (2124, 229)
cv_gene_onehotencoding is converted feature using onehot encoding method. The
shape of gene feature: (532, 229)
test_gene_onehotencoding is converted feature using onehot encoding method. The
shape of gene feature: (665, 229)
```

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

[ ]:
```
['abl1',
 'acvr1',
 'ago2',
 'akt1',
 'akt2',
 'akt3',
 'alk',
 'apc',
 'ar',
 'araf',
 'arid1b',
 'arid2',
 'arid5b',
 'asxl1',
 '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',
'egfr',
'elf3',
'ep300',
'epas1',
'erbb2',
'erbb3',
'erbb4',
'ercc2',
'ercc3',
'ercc4',
'erg',
```

```
'esr1',
'etv1',
'etv6',
'ewsr1',
'ezh2',
'fanca',
'fat1',
'fbxw7',
'fgf3',
'fgf4',
'fgfr1',
'fgfr2',
'fgfr3',
'fgfr4',
'flt3',
'foxa1',
'foxl2',
'foxp1',
'fubp1',
'gata3',
'gli1',
'gna11',
'gnas',
'hist1h1c',
'hla',
'hnf1a',
'hras',
'idh1',
'idh2',
'igf1r',
'ikbke',
'il7r',
'jak1',
'jak2',
'jun',
'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',
'msh2',
'msh6',
'mtor',
'myc',
'mycn',
'myd88',
'myod1',
'nf1',
'nf2',
'nfe2l2',
'nfkbia',
'nkx2',
'notch1',
'notch2',
'npm1',
'nras',
'nsd1',
'ntrk1',
'ntrk2',
'ntrk3',
'nup93',
'pak1',
'pax8',
'pbrm1',
'pdgfra',
'pdgfrb',
'pik3ca',
'pik3cb',
'pik3cd',
'pik3r1',
'pik3r2',
'pik3r3',
'pim1',
```

```
'pms1',
'pms2',
'pole',
'ppm1d',
'ppp2r1a',
'ppp6c',
'prdm1',
'ptch1',
'pten',
'ptpn11',
'ptprd',
'ptprt',
'rab35',
'rac1',
'rad21',
'rad50',
'rad51b',
'rad51c',
'rad51d',
'rad54l',
'raf1',
'rasa1',
'rb1',
'rbm10',
'ret',
'rheb',
'rhoa',
'rit1',
'ros1',
'runx1',
'rxra',
'sdhb',
'setd2',
'sf3b1',
'shoc2',
'smad2',
'smad3',
'smad4',
'smarca4',
'smarcb1',
'smo',
'sos1',
'sox9',
'spop',
'src',
'srsf2',
'stag2',
```

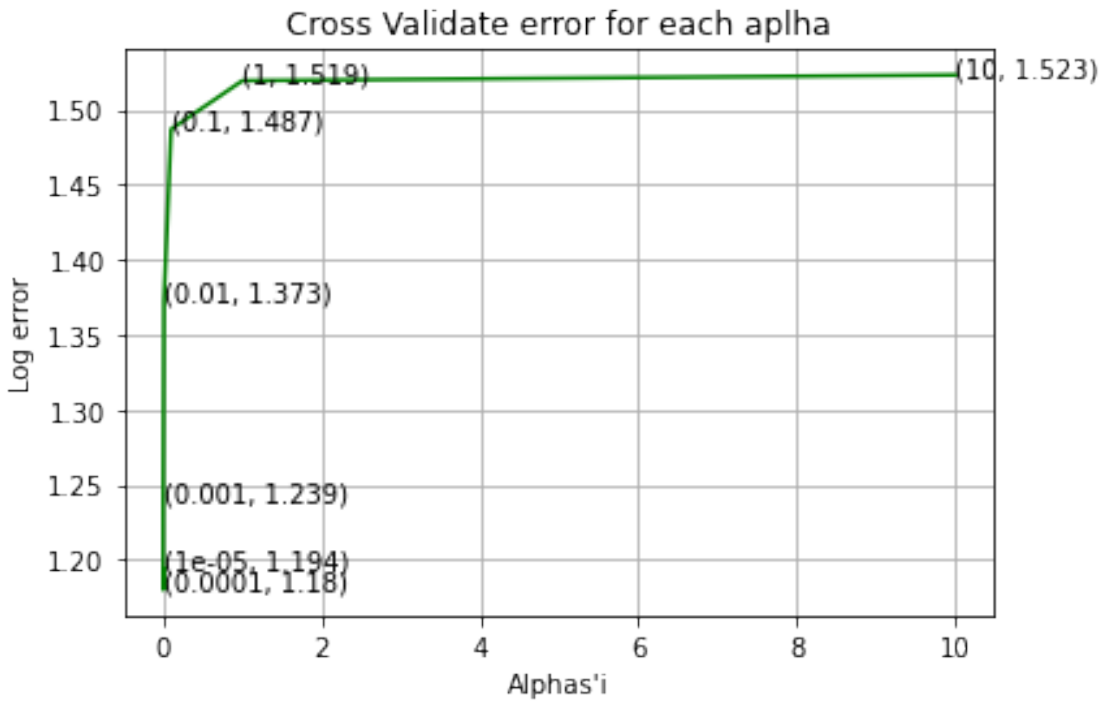
```
 'stat3',
 'stk11',
 'tert',
 'tet2',
 'tgfbr1',
 'tgfbr2',
 'tmprss2',
 'tp53',
 'tsc1',
 'tsc2',
 'u2af1',
 'vegfa',
 'vhl',
 'whsc1l1',
 'xrcc2',
 'yap1']
```

Modelling using Logistic Regression

[ ]:
```
cv_log_error = []
alpha = [10 ** x for x in range(-5 , 2 )]
for i in alpha:
  clf   = SGDClassifier(alpha= i, penalty= 'l2' , loss='log', random_state= 42,␣
 ↪n_jobs= -1)
  clf.fit(train_gene_onehotencode, y_train)
  sig_clf = CalibratedClassifierCV(clf, method='sigmoid')
  sig_clf.fit(train_gene_onehotencode, y_train)
  y_predicted = sig_clf.predict_proba(cv_gene_onehotencode)
  cv_log_error.append(log_loss(y_cv, y_predicted, labels= clf.classes_ ,␣
 ↪eps=1e-15))
  print("LogLoss for alpha",i,"is: ",log_loss(y_cv, y_predicted, labels= clf.
 ↪classes_, eps=1e-15))

fig,ax = plt.subplots()
ax.plot(alpha, cv_log_error, c='g')
for i, txt in enumerate(np.round(cv_log_error,3)):
  ax.annotate((alpha[i],np.round(txt,3)), (alpha[i], cv_log_error[i]))
plt.title("Cross Validate error for each aplha")
plt.xlabel("Alphas'i")
plt.ylabel("Log error")
plt.grid()
plt.show()

best_alpha= np.argmin(cv_log_error)
clf = SGDClassifier(alpha = alpha[best_alpha], penalty='l2', random_state=42,␣
 ↪loss='log', n_jobs=-1)
clf.fit(train_gene_onehotencode, y_train)
```

```
sig_clf = CalibratedClassifierCV(clf, method = 'sigmoid')
# sig_clf = CalibratedClassifierCV(clf, method = 'isotonic')
# For value of best alpha 1  the logloss is:  0.9549938260739086
# For value of best alpha 1  the logloss is:  1.3962269548727062
# For value of best alpha 1  the logloss is:  1.3576217932980164

sig_clf.fit(train_gene_onehotencode, y_train)

y_predicted = sig_clf.predict_proba(train_gene_onehotencode)
print("For value of best alpha", alpha[best_alpha] , " the logloss is:␣
 ↪",log_loss(y_train, y_predicted, labels=clf.classes_, eps=1e-15))

y_predicted = sig_clf.predict_proba(cv_gene_onehotencode)
print("For value of best alpha", alpha[best_alpha], " the logloss is: ",␣
 ↪log_loss(y_cv, y_predicted , labels= clf.classes_, eps=1e-15))

y_predicted = sig_clf.predict_proba(test_gene_onehotencode)
print("For value of best alpha", alpha[best_alpha], " the logloss is:␣
 ↪",log_loss(y_test, y_predicted, labels=clf.classes_, eps=1e-15))
```

```
LogLoss for alpha 1e-05 is:  1.1938429418727896
LogLoss for alpha 0.0001 is:  1.1799408115063472
LogLoss for alpha 0.001 is:  1.239097250191419
LogLoss for alpha 0.01 is:  1.3726607421328307
LogLoss for alpha 0.1 is:  1.4870232172431697
LogLoss for alpha 1 is:  1.51937134805121
LogLoss for alpha 10 is:  1.5231969989072416
```




```
For value of best alpha 0.0001  the logloss is:  1.0040492075534777
For value of best alpha 0.0001  the logloss is:  1.1799408115063472
For value of best alpha 0.0001  the logloss is:  1.1980153162458622
```

Hence by observing Train, Cross Validate and Test logloss, it is clear that our Logistic Regression Model is not Overfitting and Underfitting.

[ ]:
```
print("Finding how many datapoint of Test and Cross Validate are present in␣
 ↪Train dataset.")
test_coverage = x_test[x_test['Gene'].isin(list(set(x_train['Gene'])))].shape[0]
cv_coverage   = x_cv[x_cv['Gene'].isin(list(set(x_train['Gene'])))].shape[0]
print("Out of ",x_test.shape[0] , " datapoints", test_coverage , " are present␣
 ↪in Train dataset.", np.round((test_coverage/x_test.shape[0])*100 , 3),"%")
print("Out of ",x_cv.shape[0], " datapoints", cv_coverage , " are present in␣
 ↪Train dataset.", np.round((cv_coverage/x_cv.shape[0])*100 , 3),"%")
```

```
Finding how many datapoint of Test and Cross Validate are present in Train
dataset.
Out of  665  datapoints 641  are present in Train dataset. 96.391 %
Out of  532  datapoints 513  are present in Train dataset. 96.429 %
```

Hence, it is clear that all the three (Train, CrossValidate and Test) datasets follows similar distribution.

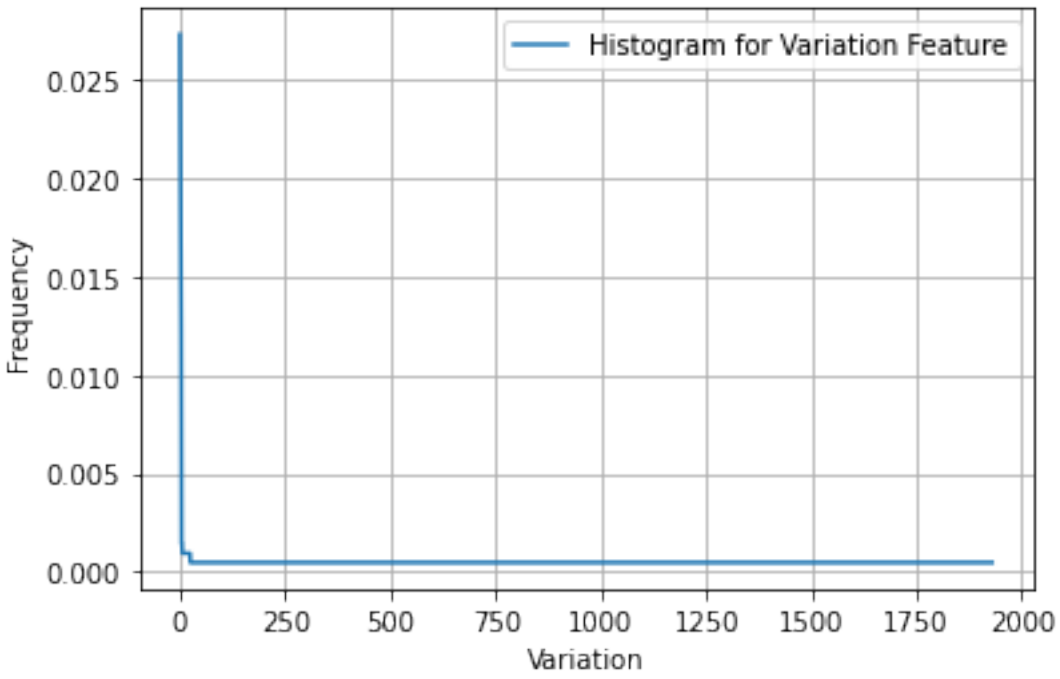
Univarient Analysis of Variation Feature

[ ]:
```
unique_var = x_train['Variation'].value_counts()
top_10_var = x_train.Variation.value_counts()
print("Top 10 Variation")
top_10_var.head(10)
```

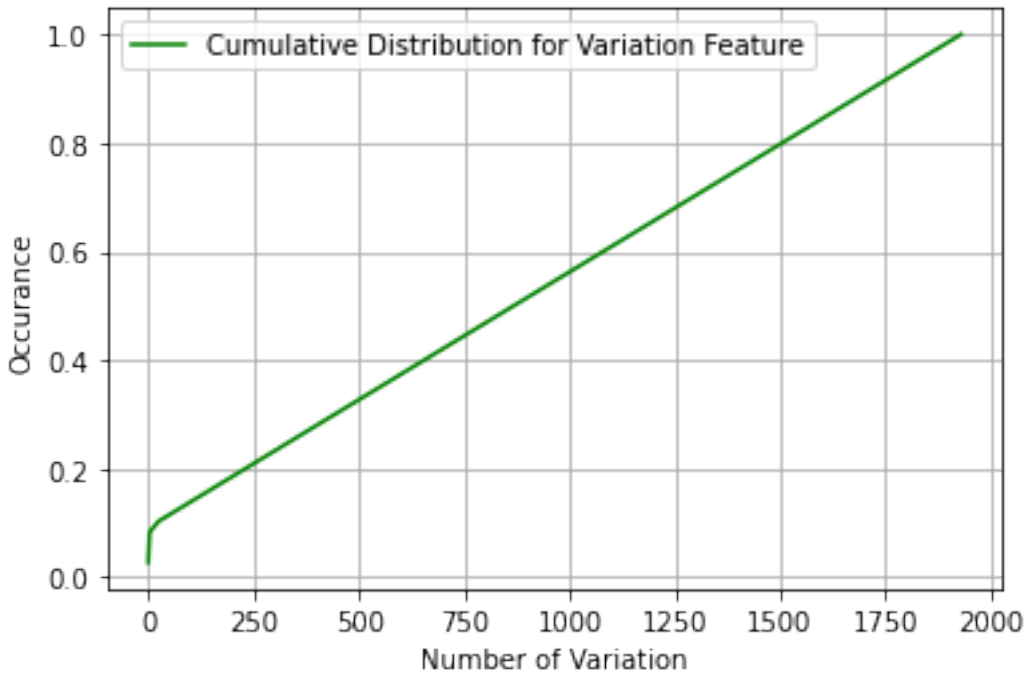
```
Top 10 Variation
```

[ ]:
```
Truncating Mutations         58
Amplification                49
Deletion                     42
Fusions                      26
G12V                          3
Overexpression                3
T167A                         2
A146V                         2
Promoter Hypermethylation     2
S222D                         2
Name: Variation, dtype: int64
```

[ ]:
```
# plotting histogram
s = sum(unique_var.values)
h = unique_var.values/s
plt.plot(h, label="Histogram for Variation Feature")
plt.xlabel('Variation')
plt.ylabel('Frequency')
plt.legend()
plt.grid()
plt.show()
```



[ ]:
```
# plotting Cumulative Distribution
c = np.cumsum(h)
plt.plot(c, label="Cumulative Distribution for Variation Feature", c='g')
plt.xlabel("Number of Variation")
plt.ylabel('Occurance')
plt.legend()
plt.grid()
plt.show()
```



Calling Response Coding for Variation feature

[ ]:
```
alpha =1
train_var_ResponseCode = np.array(get_gv_feature(alpha, 'Variation', x_train))
cv_var_ResponseCode     = np.array(get_gv_feature(alpha, 'Variation', x_cv))
test_var_ResponseCode   = np.array(get_gv_feature(alpha, 'Variation', x_test))
print("After Response Coding Train dataset looks like: ",train_var_ResponseCode.
 ↪shape)
print("After Response Coding Cross Validate dataset looks like:␣
 ↪",cv_var_ResponseCode.shape)
print("After Response Coding Test dataset looks like: ",test_var_ResponseCode.
 ↪shape)
```

```
After Response Coding Train dataset looks like:  (2124, 9)
After Response Coding Cross Validate dataset looks like:  (532, 9)
After Response Coding Test dataset looks like:  (665, 9)
```

Variation Feature using OneHot Encoding

[ ]:
```
var_encoding = CountVectorizer()
train_var_onehotencode = var_encoding.fit_transform(x_train['Variation'])
cv_var_onehotencode    = var_encoding.transform(x_cv['Variation'])
test_var_onehotencode  = var_encoding.transform(x_test['Variation'])
print("Train dataset after applying One Hot Encoding looks like:␣
 ↪",train_var_onehotencode.shape)
```

```
print("Cross Validate dataset after applying One Hot Encoding looks like:␣
  ↪",cv_var_onehotencode.shape)
print("Test dataset after applying One Hot Encoding looks like:␣
  ↪",test_var_onehotencode.shape)
```

```
Train dataset after applying One Hot Encoding looks like:  (2124, 1961)
Cross Validate dataset after applying One Hot Encoding looks like:  (532, 1961)
Test dataset after applying One Hot Encoding looks like:  (665, 1961)
```

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

[ ]:
```
['126',
 '13',
 '17',
 '19',
 '1_2009trunc',
 '20',
 '256_286trunc',
 '422_605trunc',
 '51',
 '6a',
 '963_d1010splice',
 'a1022e',
 'a1065t',
 'a111p',
 'a1170v',
 'a1200v',
 'a120s',
 'a121e',
 'a122',
 'a1234t',
 'a126d',
 'a126g',
 'a126v',
 'a134d',
 'a146t',
 'a146v',
 'a148t',
 'a149p',
 'a1519t',
 'a151t',
 'a1685s',
 'a1701p',
 'a1708e',
 'a1708v',
 'a1752p',
 'a1752v',
```

```
'a1823t',
'a1830t',
'a1843p',
'a1843t',
'a18d',
'a197t',
'a2034v',
'a205t',
'a209t',
'a2351g',
'a2425t',
'a246p',
'a263v',
'a2643g',
'a2717s',
'a2770t',
'a290t',
'a298t',
'a339v',
'a347t',
'a349p',
'a34d',
'a389t',
'a39p',
'a41p',
'a41t',
'a459v',
'a504_y505ins',
'a530t',
'a530v',
'a532h',
'a598t',
'a598v',
'a59t',
'a60v',
'a614d',
'a633t',
'a633v',
'a634d',
'a634v',
'a636p',
'a677g',
'a707t',
'a717g',
'a723d',
'a727v',
'a728v',
```

```
'a72s',
'a750_e758del',
'a750p',
'a75p',
'a763_y764insfqea',
'a767_v769dup',
'a77p',
'a77s',
'a77t',
'a829p',
'a859_l883delinsv',
'a864t',
'a883t',
'a889p',
'a8s',
'a919v',
'a95d',
'acpp',
'agk',
'ahcyl1',
'akap9',
'akt3',
'alk',
'amplification',
'ar',
'arv567es',
'atf1',
'atg7',
'bcan',
'bcor',
'bcr',
'binding',
'braf',
'brd4',
'c1156f',
'c1156y',
'c124s',
'c134w',
'c135r',
'c135s',
'c135y',
'c1365y',
'c136r',
'c136y',
'c1385',
'c1483f',
'c1483r',
```

```
'c1483w',
'c1697r',
'c1767s',
'c18y',
'c2060g',
'c229r',
'c242f',
'c248t',
'c24r',
'c24y',
'c250t',
'c275s',
'c277w',
'c278f',
'c27a',
'c378r',
'c396r',
'c39r',
'c39s',
'c41y',
'c420r',
'c443y',
'c44f',
'c44y',
'c450_k451insmiewmi',
'c456_n468del',
'c456_r481del',
'c47s',
'c481s',
'c482r',
'c554w',
'c569y',
'c582f',
'c609y',
'c611y',
'c618r',
'c61g',
'c620r',
'c620y',
'c628y',
'c630r',
'c634w',
'c634y',
'c64g',
'c706f',
'c712r',
'c71y',
```

```
'c77f',
'c91a',
'c91s',
'cad',
'ccdc170',
'cd74',
'cep110',
'cep85l',
'chtop',
'cic',
'cpeb1',
'creb1',
'cux1',
'd1010h',
'd1010y',
'd1067a',
'd1067v',
'd1067y',
'd1071n',
'd108h',
'd108n',
'd1203n',
'd1280v',
'd130a',
'd1349h',
'd1352y',
'd1399y',
'd140g',
'd153v',
'd162g',
'd1709a',
'd1709e',
'd171g',
'd171n',
'd1733g',
'd1739g',
'd1739v',
'd1739y',
'd1778g',
'd1778n',
'd1778y',
'd1810a',
'd2312v',
'd245v',
'd24y',
'd2512y',
'd252g',
```

```
'd254n',
'd257n',
'd2665g',
'd2723h',
'd277h',
'd2870a',
'd287h',
'd29y',
'd3170g',
'd323h',
'd325a',
'd326n',
'd32a',
'd32h',
'd32n',
'd32y',
'd331g',
'd350g',
'd357y',
'd387v',
'd390y',
'd399n',
'd401n',
'd402y',
'd404g',
'd408e',
'd419del',
'd422n',
'd423n',
'd450h',
'd473g',
'd473h',
'd520n',
'd537y',
'd544h',
'd560y',
'd572a',
'd579del',
'd594e',
'd594n',
'd594v',
'd594y',
'd595v',
'd600_l601insfreyeyd',
'd603n',
'd609e',
'd60n',
```

```
'd617g',
'd61y',
'd631g',
'd641n',
'd646y',
'd65n',
'd661y',
'd67n',
'd67y',
'd717v',
'd737n',
'd74n',
'd761y',
'd769a',
'd769h',
'd769y',
'd770_n771insd',
'd770_n771insnpg',
'd770_n771insvdsvdnp',
'd770_p772dup',
'd806h',
'd808n',
'd816f',
'd816g',
'd816h',
'd816n',
'd816v',
'd816y',
'd820a',
'd820g',
'd821n',
'd835a',
'd835del',
'd835h',
'd835n',
'd835y',
'd837n',
'd83v',
'd842_h845del',
'd842i',
'd842v',
'd84g',
'd84h',
'd84n',
'd86n',
'd927g',
'd92a',
```

```
'd92e',
'd92n',
'd92v',
'd935n',
'd96n',
'ddit3',
'deletion',
'deletions',
'dna',
'dnmt3b7',
'domain',
'dux4',
'e1021k',
'e102_i103del',
'e1060a',
'e106g',
'e1071w',
'e116k',
'e120q',
'e1214k',
'e124q',
'e1250k',
'e127g',
'e1282v',
'e1286v',
'e1356g',
'e135k',
'e1384k',
'e139d',
'e14',
'e142d',
'e143k',
'e144k',
'e1552del',
'e157g',
'e1586g',
'e1644g',
'e1682k',
'e168d',
'e1705a',
'e1705k',
'e172k',
'e1794d',
'e17k',
'e1836k',
'e1935g',
'e218',
```

```
'e221q',
'e239a',
'e23fs',
'e255k',
'e258v',
'e267g',
'e279k',
'e281k',
'e2856a',
'e285k',
'e285v',
'e29v',
'e3002k',
'e311_k312del',
'e317k',
'e321g',
'e321k',
'e326l',
'e330g',
'e330k',
'e35',
'e355a',
'e365k',
'e40k',
'e40n',
'e40q',
'e40t',
'e40w',
'e41a',
'e453a',
'e459k',
'e462g',
'e466k',
'e475k',
'e49k',
'e501g',
'e50k',
'e518a',
'e518k',
'e541k',
'e542g',
'e542q',
'e542v',
'e545g',
'e545k',
'e554_k558del',
'e554_v559del',
```

```
'e563k',
'e565g',
'e579k',
'e580',
'e598_y599insdvdfreye',
'e598_y599insglvqvtgssdneyfyvdfreye',
'e5k',
'e606g',
'e60l',
'e622d',
'e626k',
'e627d',
'e633k',
'e664k',
'e685v',
'e69g',
'e69k',
'e709_t710delinsd',
'e709a',
'e709g',
'e709k',
'e709q',
'e709v',
'e719k',
'e731k',
'e734q',
'e746_a750del',
'e746_a750delinsq',
'e746_s752delinsa',
'e746_s752delinsi',
'e746_t751delinsva',
'e746_t751insip',
'e746g',
'e746v',
'e75g',
'e768d',
'e76k',
'e77k',
'e78k',
'e812k',
'e81k',
'e82d',
'e82g',
'e82v',
'e836k',
'e839k',
'e846k',
```

```
'e884k',
'e88k',
'e946',
'e996k',
'ebf1',
'egfr',
'egfrvii',
'egfrviv',
'egfrvv',
'ep300',
'erbb4',
'erg',
'erlin2',
'esr1',
'esrp1',
'etv1',
'etv4',
'etv5',
'etv6',
'evi1',
'ewsr1',
'exon',
'ezr',
'f102c',
'f1061w',
'f1088lfs',
'f1088sfs',
'f119s',
'f1200i',
'f1245v',
'f129l',
'f12l',
'f133l',
'f133v',
'f1524v',
'f154l',
'f158c',
'f1592s',
'f1704s',
'f1734s',
'f1761s',
'f1888l',
'f1888v',
'f212y',
'f21a',
'f241s',
'f248s',
```

```
'f28l',
'f311l',
'f317l',
'f346v',
'f347l',
'f354l',
'f359c',
'f367s',
'f384l',
'f384v',
'f468c',
'f53c',
'f53l',
'f53s',
'f568fs',
'f57c',
'f57l',
'f57v',
'f594_r595inssdneyfyvdf',
'f71i',
'f74s',
'f808l',
'f81v',
'f876l',
'f893l',
'fam118b',
'fam131b',
'fam76a',
'fgfr1',
'fgfr1op1',
'fgfr2',
'fgfr3',
'fig',
'fus',
'fusion',
'fusions',
'g101s',
'g106_r108del',
'g1123d',
'g1123s',
'g1125a',
'g1128a',
'g1128s',
'g114r',
'g1194d',
'g1202r',
'g1232d',
```

```
'g123r',
'g123s',
'g1269a',
'g127e',
'g127n',
'g1286r',
'g128v',
'g129e',
'g129r',
'g12a',
'g12c',
'g12r',
'g12v',
'g13d',
'g13e',
'g14v',
'g1596v',
'g161v',
'g165e',
'g1706a',
'g1738e',
'g1743r',
'g1770v',
'g1788d',
'g1788v',
'g17a',
'g17v',
'g1803a',
'g1809k',
'g1809r',
'g186r',
'g1971e',
'g199r',
'g2032r',
'g2101a',
'g2274v',
'g23d',
'g244d',
'g244r',
'g244s',
'g245a',
'g245d',
'g245s',
'g248v',
'g251c',
'g253c',
'g264s',
```

```
'g266e',
'g271e',
'g2748d',
'g284r',
'g305r',
'g305w',
'g311d',
'g31a',
'g31r',
'g31v',
'g325r',
'g328e',
'g334r',
'g34e',
'g34v',
'g35a',
'g35r',
'g370c',
'g373r',
'g375c',
'g375p',
'g380r',
'g382d',
'g39e',
'g419v',
'g423r',
'g423v',
'g44s',
'g464a',
'g464r',
'g466a',
'g466v',
'g469a',
'g469del',
'g469e',
'g469v',
'g478c',
'g480w',
'g503v',
'g505s',
'g591v',
'g596c',
'g596r',
'g602r',
'g60d',
'g60r',
'g660d',
```

```
'g665a',
'g67r',
'g67s',
'g67w',
'g701s',
'g719a',
'g719c',
'g719d',
'g719s',
'g724s',
'g735s',
'g75r',
'g774v',
'g776s',
'g778_p780dup',
'g785s',
'g796s',
'g81r',
'g81s',
'g857e',
'g863d',
'g863s',
'g87r',
'g936r',
'g93w',
'golga4',
'h1047r',
'h1047y',
'h105r',
'h1094r',
'h1106d',
'h114y',
'h115n',
'h118p',
'h123d',
'h132y',
'h1382y',
'h1421y',
'h168n',
'h1746q',
'h179q',
'h179r',
'h1918y',
'h191d',
'h193n',
'h193p',
'h2074n',
```

```
'h214n',
'h214q',
'h214r',
'h233n',
'h284n',
'h284p',
'h297n',
'h36p',
'h396p',
'h398y',
'h410r',
'h412y',
'h492r',
'h538q',
'h570r',
'h597y',
'h61r',
'h643d',
'h650q',
'h65y',
'h662q',
'h662r',
'h68y',
'h694r',
'h697y',
'h773_v774insh',
'h773dup',
'h773inslgnp',
'h773l',
'h845_n848delinsp',
'h845y',
'h870r',
'h875y',
'h876q',
'h878y',
'h93d',
'h93r',
'hmga2',
'hypermethylation',
'i1018f',
'i103n',
'i111a',
'i111n',
'i111r',
'i1170n',
'i1171n',
'i1183t',
```

```
'i122l',
'i122s',
'i122v',
'i1250t',
'i151s',
'i157t',
'i1616n',
'i1616t',
'i162m',
'i1680n',
'i168f',
'i1766s',
'i1807s',
'i18v',
'i195t',
'i204t',
'i21v',
'i2500f',
'i2500m',
'i255f',
'i2627f',
'i26n',
'i279p',
'i28t',
'i290a',
'i290r',
'i31m',
'i33del',
'i347m',
'i391m',
'i42v',
'i448v',
'i463s',
'i47f',
'i491m',
'i49s',
'i538v',
'i562m',
'i563_l576del',
'i642v',
'i653t',
'i668v',
'i68k',
'i744_k745delinskipvai',
'i834v',
'i843_d846del',
'i843del',
```

```
'i852m',
'i853t',
'i90t',
'igh',
'igl',
'in',
'insertion',
'insertions',
'jak2',
'k101m',
'k1026e',
'k111e',
'k111n',
'k117r',
'k11r',
'k120e',
'k125e',
'k125l',
'k125m',
'k128n',
'k128q',
'k128t',
'k1434i',
'k1436q',
'k162d',
'k1702e',
'k179m',
'k181m',
'k189n',
'k2472t',
'k2729n',
'k289e',
'k291e',
'k291q',
'k292i',
'k2950n',
'k310r',
'k335i',
'k341a',
'k375a',
'k379e',
'k382e',
'k38n',
'k398a',
'k409q',
'k428a',
'k442nfs',
```

```
'k45n',
'k45q',
'k45t',
'k467t',
'k483e',
'k499e',
'k4e',
'k507a',
'k507q',
'k50e',
'k513r',
'k517r',
'k525e',
'k526e',
'k52r',
'k539l',
'k550_k558del',
'k558_e562del',
'k558delinsnp',
'k558n',
'k575m',
'k57e',
'k57t',
'k590r',
'k5n',
'k601e',
'k601q',
'k603q',
'k607t',
'k62r',
'k641r',
'k642e',
'k648n',
'k650e',
'k650n',
'k650r',
'k656e',
'k659n',
'k666m',
'k666n',
'k700r',
'k745_a750del',
'k745m',
'k753a',
'k753m',
'k765r',
'k78a',
```

```
'k78i',
'k79e',
'k82t',
'k830r',
'k83n',
'k935i',
'k975e',
'kdd',
'kdr',
'kiaa1509',
'kiaa1549',
'kiaa1967',
'kif5b',
'l1026f',
'l108p',
'l1122v',
'l112p',
'l112r',
'l1152p',
'l115r',
'l1195v',
'l1198f',
'l1198p',
'l1204f',
'l122r',
'l1407p',
'l145r',
'l147f',
'l1574p',
'l1584r',
'l158v',
'l1593p',
'l1657p',
'l165p',
'l1678p',
'l1705p',
'l1764p',
'l180p',
'l181p',
'l1844r',
'l1904v',
'l191h',
'l1947r',
'l1951r',
'l202f',
'l209f',
'l2230v',
```

```
'l225li',
'l239r',
'l23f',
'l246v',
'l248v',
'l265p',
'l2721h',
'l272f',
'l283_d294del',
'l28p',
'l30f',
'l321a',
'l325f',
'l330r',
'l344p',
'l344r',
'l345q',
'l348s',
'l358r',
'l388m',
'l399v',
'l412f',
'l424i',
'l424v',
'l43v',
'l448p',
'l455m',
'l461v',
'l469v',
'l46r',
'l481f',
'l485_p490del',
'l485_p490delinsf',
'l485_p490delinsy',
'l493p',
'l493v',
'l507p',
'l52r',
'l535p',
'l536h',
'l536q',
'l536r',
'l550p',
'l576del',
'l576p',
'l57del',
'l57v',
```

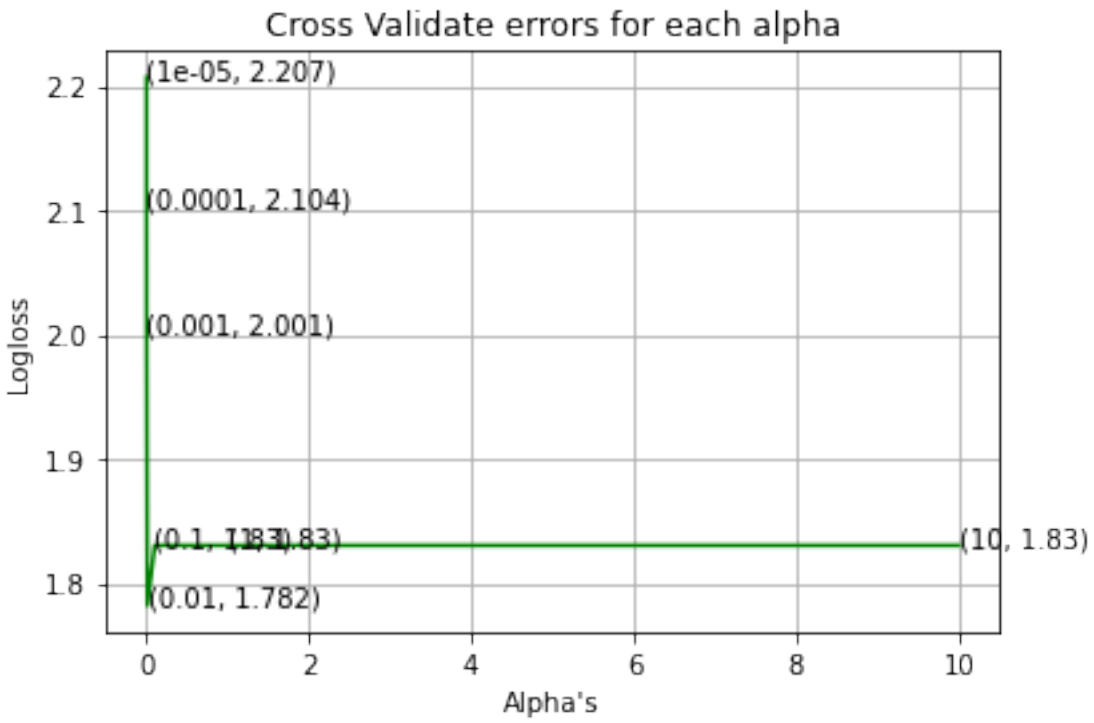
```
 'l584f',
 'l585i',
 'l597q',
 'l597r',
 'l607i',
 'l611v',
 'l617f',
 'l622h',
 'l63f',
 'l63v',
 'l668f',
 'l67p',
 'l692f',
 'l703p',
 'l704n',
 'l708p',
 'l726i',
 'l747_a750del',
 'l747_a750delinsp',
 'l747_e749del',
 'l747_p753del',
 'l747_t751del',
 'l749p',
 'l755s',
 …]
```

Modeling Variation feature with Logistic Regression using Response Coding

[ ]:
```
alpha = [10 ** x for x in range(-5,2)]
cv_log_error = []
for i in alpha:
  clf = SGDClassifier(alpha= i, penalty='l1',loss='log', n_jobs= -1,␣
 ↪random_state=42)
  clf.fit(train_var_ResponseCode, y_train)
  sig_clf = CalibratedClassifierCV(clf, method='sigmoid')
  sig_clf.fit(train_var_ResponseCode, y_train)
  y_predicted = sig_clf.predict_proba(cv_var_ResponseCode)
  cv_log_error.append(log_loss(y_cv, y_predicted, labels=clf.classes_,␣
 ↪eps=1e-15))
  print("For alpha ", i," logloss is: ",log_loss(y_cv, y_predicted, labels=clf.
 ↪classes_, eps=1e-15))

fig, ax = plt.subplots()
ax.plot(alpha, cv_log_error, c='g')
for i, txt in enumerate(np.round(cv_log_error,3)):
  ax.annotate((alpha[i], np.round(txt,3)), (alpha[i], cv_log_error[i]))
plt.title("Cross Validate errors for each alpha")
```

```
plt.xlabel("Alpha's")
plt.ylabel("Logloss")
plt.grid()
plt.show()


best_alpha = np.argmin(cv_log_error)

clf = SGDClassifier(alpha = alpha[best_alpha], penalty='l1',loss='log', n_jobs=␣
 ↪-1, random_state=42)
clf.fit(train_var_ResponseCode, y_train)
sig_clf = CalibratedClassifierCV(clf, method='sigmoid')
sig_clf.fit(train_var_ResponseCode, y_train)

y_predicted = sig_clf.predict_proba(test_var_ResponseCode)
print("For alpha ", alpha[best_alpha]," the logloss on Test dataset is:␣
 ↪",log_loss(y_test, y_predicted, labels=clf.classes_, eps=1e-15))

y_predicted = sig_clf.predict_proba(train_var_ResponseCode)
print("For alpha ", alpha[best_alpha]," the logloss on Train dataset is:␣
 ↪",log_loss(y_train, y_predicted, labels=clf.classes_, eps=1e-15))

y_predicted = sig_clf.predict_proba(cv_var_ResponseCode)
print("For alpha ", alpha[best_alpha]," the logloss on Cross Validate dataset␣
 ↪is: ",log_loss(y_cv, y_predicted, labels=clf.classes_, eps=1e-15))
```

```
For alpha  1e-05  logloss is:  2.207098108919576
For alpha  0.0001  logloss is:  2.1038324782577975
For alpha  0.001  logloss is:  2.0014741445749826
For alpha  0.01  logloss is:  1.781997649498826
For alpha  0.1  logloss is:  1.8303536242315765
For alpha  1  logloss is:  1.8303536242674052
For alpha  10  logloss is:  1.8303536255236392
```



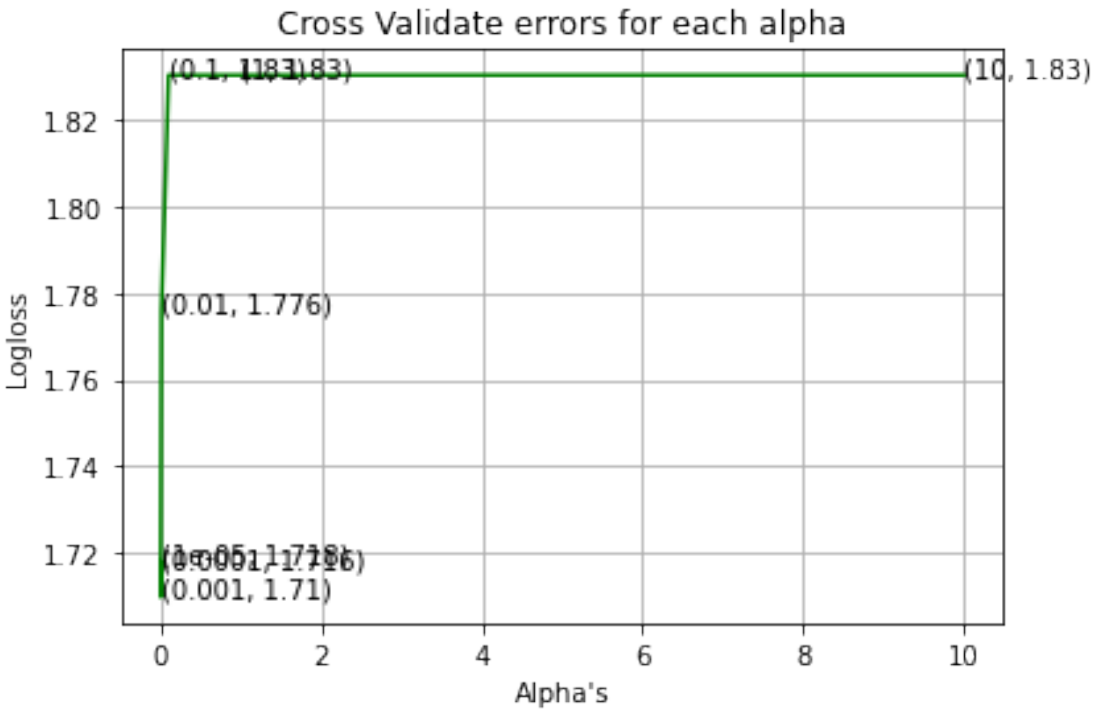

```
For alpha  0.01  the logloss on Test dataset is:  1.789670113085929
For alpha  0.01  the logloss on Train dataset is:  1.765256915743442
For alpha  0.01  the logloss on Cross Validate dataset is:  1.781997649498826
```

Modeling Variation feature with Logistic Regression using OneHot Encoding

[ ]:
```
alpha = [10 ** x for x in range(-5,2)]
cv_log_error = []
for i in alpha:
  clf = SGDClassifier(alpha= i, penalty='l1',loss='log', n_jobs= -1,␣
 ↪random_state=42)
  clf.fit(train_var_onehotencode, y_train)
  sig_clf = CalibratedClassifierCV(clf, method='sigmoid')
  sig_clf.fit(train_var_onehotencode, y_train)
  y_predicted = sig_clf.predict_proba(cv_var_onehotencode)
  cv_log_error.append(log_loss(y_cv, y_predicted, labels=clf.classes_,␣
 ↪eps=1e-15))
  print("For alpha ", i," logloss is: ",log_loss(y_cv, y_predicted, labels=clf.
 ↪classes_, eps=1e-15))

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

```
plt.title("Cross Validate errors for each alpha")
plt.xlabel("Alpha's")
plt.ylabel("Logloss")
plt.grid()
plt.show()


best_alpha = np.argmin(cv_log_error)

clf = SGDClassifier(alpha = alpha[best_alpha], penalty='l1',loss='log', n_jobs=␣
 ↪-1, random_state=42)
clf.fit(train_var_onehotencode, y_train)
sig_clf = CalibratedClassifierCV(clf, method='sigmoid')
sig_clf.fit(train_var_onehotencode, y_train)

y_predicted = sig_clf.predict_proba(test_var_onehotencode)
print("For alpha ", alpha[best_alpha]," the logloss on Test dataset is:␣
 ↪",log_loss(y_test, y_predicted, labels=clf.classes_, eps=1e-15))

y_predicted = sig_clf.predict_proba(train_var_onehotencode)
print("For alpha ", alpha[best_alpha]," the logloss on Train dataset is:␣
 ↪",log_loss(y_train, y_predicted, labels=clf.classes_, eps=1e-15))

y_predicted = sig_clf.predict_proba(cv_var_onehotencode)
print("For alpha ", alpha[best_alpha]," the logloss on Cross Validate dataset␣
 ↪is: ",log_loss(y_cv, y_predicted, labels=clf.classes_, eps=1e-15))

# Logistic reg With L1 regularization
# For alpha  0.001  the logloss on Test dataset is:  1.7134085717233352
# For alpha  0.001  the logloss on Train dataset is:  1.687610047601688
# For alpha  0.001  the logloss on Cross Validate dataset is:  1.
 ↪6984516054770864

# Logistic reg With L2 regularization
# For alpha  0.0001  the logloss on Test dataset is:  1.7029377326637025
# For alpha  0.0001  the logloss on Train dataset is:  0.6604824786838963
# For alpha  0.0001  the logloss on Cross Validate dataset is:  1.
 ↪689454970455003
```

```
For alpha  1e-05  logloss is:  1.717835396121652
For alpha  0.0001  logloss is:  1.7164042239012698
For alpha  0.001  logloss is:  1.7097048769561436
For alpha  0.01  logloss is:  1.7757880800119366
For alpha  0.1  logloss is:  1.8303536247739272
For alpha  1  logloss is:  1.8303536245954006
For alpha  10  logloss is:  1.830353624593319
```




```
For alpha  0.001  the logloss on Test dataset is:  1.6838697482967115
For alpha  0.001  the logloss on Train dataset is:  1.6917125249451703
For alpha  0.001  the logloss on Cross Validate dataset is:  1.7097048769561436
```

Hence by observing Train, Cross Validate and Test logloss, it is clear that our Logistic Regression Model for Variation feature is not Overfitting and Underfitting.

[ ]:
```
print("Finding how many data points for Test and Cross Validate are present in␣
 ↪Train dataset.")
test_coverage = x_test[x_test['Variation'].
 ↪isin(list(set(x_train['Variation'])))].shape[0]
cv_coverage = x_cv[x_cv['Variation'].isin(list(set(x_train['Variation'])))].
 ↪shape[0]
print("Out of ",x_test.shape[0] ,"(Test) datapoints",  test_coverage, " are␣
 ↪present in Train dataset.", "(",np.round((test_coverage/x_test.
 ↪shape[0])*100,3),"%)")
print("Out of ",x_cv.shape[0] ,"(Cross Validate) datapoints",  cv_coverage, "␣
 ↪are present in Train dataset.", "(",np.round((cv_coverage/x_cv.
 ↪shape[0])*100,3),"%)")
```

```
Finding how many data points for Test and Cross Validate are present in Train
dataset.
Out of  665 (Test) datapoints 75  are present in Train dataset. ( 11.278 %)
Out of  532 (Cross Validate) datapoints 47  are present in Train dataset. (
8.835 %)
```

Hence, it is clear that all the three (Train, CrossValidate and Test) datasets shares little amount of same distribution.

Univerant analysis of Text feature

[ ]:
```
def extract_dictonary_paddle(cls_text):
  dictionary = defaultdict(int)   #defaultdict does not raise keyvalue error
  for index, row in cls_text.iterrows():
    for word in row['TEXT'].split():
      dictionary[word] += 1
  return dictionary
```

[ ]:
```
def get_response_code(df):
  text_feature_response_coding = 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 words in row['TEXT'].split():
        sum_prob += math.log(((dict_list[i].get(words,0)+10) / (total_dict.
 ↪get(words,0)+90)))
      text_feature_response_coding[row_index][i] = math.exp(sum_prob/
 ↪len(row['TEXT'].split()))
      row_index += 1
    return text_feature_response_coding
```

[ ]:
```
# buid a countvector for word occur minimum 3 times.
text_vectorizer = CountVectorizer(min_df=3)
train_text_onehotencode = text_vectorizer.fit_transform(x_train['TEXT'])
cv_text_onehotencode    = text_vectorizer.transform(x_cv['TEXT'])
test_text_onehotencode  = text_vectorizer.transform(x_test['TEXT'])
# get feature names
train_text_feat_names   = text_vectorizer.get_feature_names()

train_text_feature_count= train_text_onehotencode.sum(axis=0).A1

text_feat_dict          = dict(zip(list(train_text_feat_names),␣
 ↪train_text_feature_count))
print("Number of unique words in dictionary: ",len(train_text_feat_names))
```

```
Number of unique words in dictionary:  53280
```

[ ]:
```
dict_list = []
for i in range(1,10):
  cls_text = x_train[x_train["Class"] == i]
  dict_list.append(extract_dictonary_paddle(cls_text))   #

total_dict = extract_dictonary_paddle(x_train)
```

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

Calling Response Coding function for Text dataset

[ ]:
```
train_text_responsecode = get_response_code(x_train)
cv_text_responsecode    = get_response_code(x_cv)
test_text_responsecode  = get_response_code(x_test)
```

Normalize Response Coding

[ ]:
```
train_text_feat_responsecode = ((train_text_responsecode.T) /␣
 ↪(train_text_responsecode.sum(axis=1))).T
cv_text_feat_responsecode    = ((cv_text_responsecode.T) /␣
 ↪(cv_text_responsecode.sum(axis=1))).T
test_text_feat_responsecode  = ((test_text_responsecode.T) /␣
 ↪(test_text_responsecode.sum(axis=1))).T
```

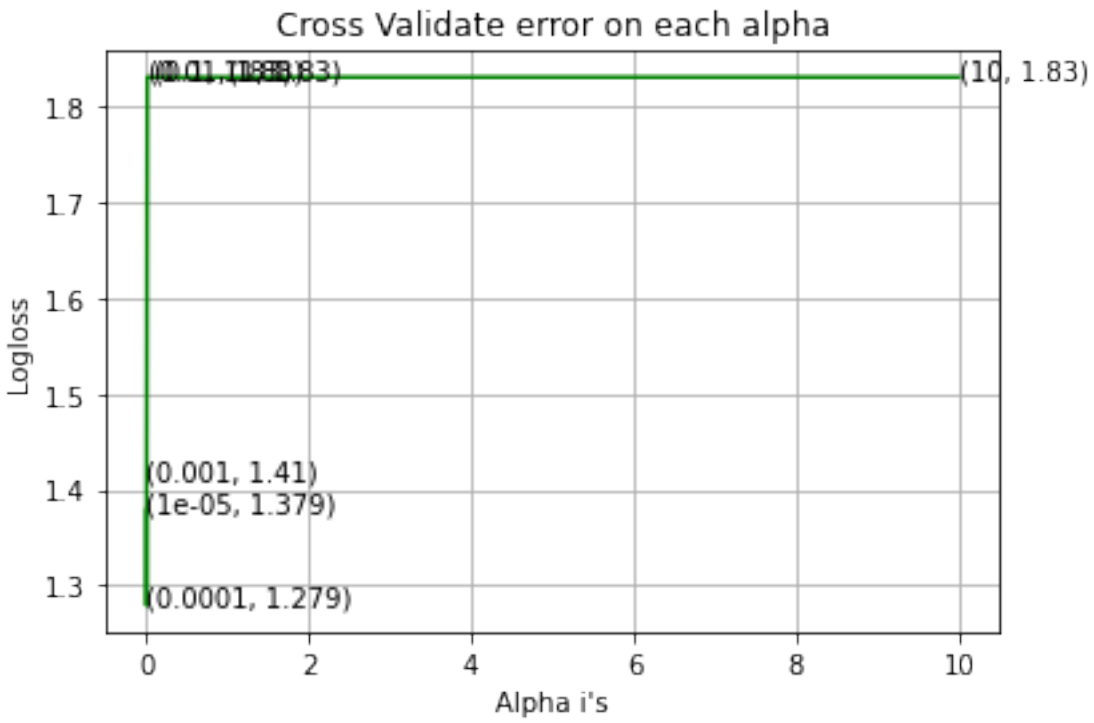
Normalize OneHot Encoding

[ ]:
```
train_text_onehotencode = normalize(train_text_onehotencode , axis=0)
cv_text_onehotencode    = normalize(cv_text_onehotencode , axis =0)
test_text_onehotencode  = normalize(test_text_onehotencode , axis =0)
```

[ ]:
```
alpha = [10 ** x for x in range(-5,2)]
cv_log_error = []
for i in alpha:
  clf = SGDClassifier(alpha = i , penalty = 'l1' , loss= 'log' , random_state =␣
 ↪42 , n_jobs=-1)
  clf.fit(train_text_onehotencode , y_train)
  sig_clf = CalibratedClassifierCV(clf , method='sigmoid')
  sig_clf.fit(train_text_onehotencode , y_train)
  y_predicted = sig_clf.predict_proba(cv_text_onehotencode)
  cv_log_error.append(log_loss(y_cv , y_predicted , labels = clf.classes_ ,␣
 ↪eps=1e-15))
  print("For alpha ",i," the log loss is: ", log_loss(y_cv , y_predicted ,␣
 ↪labels= clf.classes_ , eps= 1e-15))

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

```
for i , txt in enumerate(np.round(cv_log_error , 3)):
  ax.annotate((alpha[i] , np.round(txt,3)) , (alpha[i] , cv_log_error[i]))
plt.title("Cross Validate error on each alpha")
plt.xlabel("Alpha i's")
plt.ylabel("Logloss")
plt.grid()
plt.show()

best_alpha = np.argmin(cv_log_error)

clf = SGDClassifier(alpha = alpha[best_alpha] , penalty = 'l1', loss = 'log',␣
 ↪random_state=42, n_jobs=-1)
clf.fit(train_text_onehotencode , y_train)
sig_clf = CalibratedClassifierCV(clf , method = 'sigmoid')
sig_clf.fit(train_text_onehotencode , y_train)
y_predicted = sig_clf.predict_proba(test_text_onehotencode)
print("For alpha ",alpha[best_alpha],"the test logloss is: ", log_loss(y_test ,␣
 ↪y_predicted , labels= clf.classes_ , eps=1e-15))
y_predicted = sig_clf.predict_proba(cv_text_onehotencode)
print("For alpha ",alpha[best_alpha],"the cross validate logloss is: ",␣
 ↪log_loss(y_cv , y_predicted , labels= clf.classes_ , eps=1e-15))
y_predicted = sig_clf.predict_proba(train_text_onehotencode)
print("For alpha ",alpha[best_alpha],"the train logloss is: ", log_loss(y_train␣
 ↪, y_predicted , labels= clf.classes_ , eps=1e-15))
```

```
For alpha  1e-05  the log loss is:  1.378626835538475
For alpha  0.0001  the log loss is:  1.2792247252062663
For alpha  0.001  the log loss is:  1.410122876749546
For alpha  0.01  the log loss is:  1.8303536243464869
For alpha  0.1  the log loss is:  1.8303536247734493
For alpha  1  the log loss is:  1.8303536245954006
For alpha  10  the log loss is:  1.830353624593319
```




```
For alpha  0.0001 the test logloss is:  1.303263808809831
For alpha  0.0001 the cross validate logloss is:  1.2792247252062663
For alpha  0.0001 the train logloss is:  1.0742960444253586
```

[ ]:
```
def get_intersect_text(df_text):
  text_vectorizer = CountVectorizer()
  text_feat       = text_vectorizer.fit_transform(df_text['TEXT'])
  text_feat_names = text_vectorizer.get_feature_names()
  text_feat_count = text_feat.sum(axis=0).A1
  text_feat_dict  = dict(zip(list(text_feat_names),text_feat_count))

  len1 = len(set(text_feat_names))
  len2 = len(set(train_text_feat_names) & set(text_feat_names))
  return len1 , len2
```

[ ]:
```
len1 , len2 = get_intersect_text(x_cv)
print(np.round((len2/len1) *100,3), "%  of cross_validate dataset are present␣
 ↪in train dataset.")
len1 , len2 = get_intersect_text(x_test)
print(np.round((len2/len1) *100 , 3), "%  of test dataset are present in train␣
 ↪dataset.")
```

```
64.872 %  of cross_validate dataset are present in train dataset.
57.834 %  of test dataset are present in train dataset.
```

Heap Stack Train, Cross Validate and Test dataset (OneHotEncoding).

[ ]:
```
train_gene_var_onehotencode = hstack((train_gene_onehotencode ,␣
 ↪train_var_onehotencode))
cv_gene_var_onehotencode    = hstack((cv_gene_onehotencode ,␣
 ↪cv_var_onehotencode))
test_gene_var_onehotencode  = hstack((test_gene_onehotencode ,␣
 ↪test_var_onehotencode))

train_onehotencode = hstack((train_gene_var_onehotencode ,␣
 ↪train_text_onehotencode)).tocsr()
cv_onehotencode    = hstack((cv_gene_var_onehotencode , cv_text_onehotencode)).
 ↪tocsr()
test_onehotencode  = hstack((test_gene_var_onehotencode ,␣
 ↪test_text_onehotencode)).tocsr()

y_train_onehotencode = np.array(list(x_train['Class']))
y_cv_onehotencode    = np.array(list(x_cv['Class']))
y_test_onehotencode  = np.array(list(x_test['Class']))

print("Train Y onehotencode",y_train_onehotencode.shape)
print("Train X onehotencode",train_onehotencode.shape)
```

```
Train Y onehotencode (2124,)
Train X onehotencode (2124, 55470)
```

Stacking after OneHot Encoding

[ ]:
```
print("Train dataset after stacking:",train_onehotencode.shape)
print("Cross Validate dataset after stacking:",cv_onehotencode.shape)
print("Test dataset after stacking:",test_onehotencode.shape)
```

```
Train dataset after stacking: (2124, 55470)
Cross Validate dataset after stacking: (532, 55470)
Test dataset after stacking: (665, 55470)
```

Heap Stack Train, Cross Validate and Test dataset (Response Coding).

[ ]:
```
train_gene_var_ResponseCode = hstack((csr_matrix(train_gene_ResponseCoding) ,␣
 ↪train_var_ResponseCode))
cv_gene_var_ResponseCode    = hstack((csr_matrix(cv_gene_ResponseCoding) ,␣
 ↪cv_var_ResponseCode))
test_gene_var_ResponseCode  = hstack((csr_matrix(test_gene_ResponseCoding) ,␣
 ↪test_var_ResponseCode))
# without csr_matrix, atleast one dataset must have sparse array.
# ValueError: blocks must be 2-D
```

```
train_ResponseCode = hstack((train_gene_var_ResponseCode ,␣
 ↪train_text_feat_responsecode))
cv_ResponseCode    = hstack((cv_gene_var_ResponseCode ,␣
 ↪cv_text_feat_responsecode))
test_ResponseCode  = hstack((test_gene_var_ResponseCode ,␣
 ↪test_text_feat_responsecode))
```

Stacking after Response Coding

[ ]:
```
print("Train dataset after stacking:",train_ResponseCode.shape)
print("Cross Validate dataset after stacking:",cv_ResponseCode.shape)
print("Test dataset after stacking:",test_ResponseCode.shape)
```

```
Train dataset after stacking: (2124, 27)
Cross Validate dataset after stacking: (532, 27)
Test dataset after stacking: (665, 27)
```

Function that is use for feature importance for Naive Bayes.

[ ]:
```
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(x_train['Gene'])
    var_vec  = var_count_vec.fit(x_train['Variation'])
    text_vec = text_count_vec.fit(x_train['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")
```

Function to predict and plot confusion matrix.

[ ]:
```
def predict_and_plot_confusion_matrix(x_train, y_train, x_test, y_test, clf):
  clf.fit(x_train , y_train)
  sig_clf = CalibratedClassifierCV(clf , method='sigmoid')
  sig_clf.fit(x_train , y_train)
  y_predicted_class = sig_clf.predict(x_test)
  y_predicted = sig_clf.predict_proba(x_test)

  print("Logloss is: ", log_loss(y_test , y_predicted , labels=clf.classes_,␣
 ↪eps=1e-15))
  #print("Number of misclassified:", np.count_nonzero((y_predicted-y_test))/
 ↪y_test.shape[0])
  print("Number of mis-classified points :", np.
 ↪count_nonzero((y_predicted_class- y_test))/y_test.shape[0])
  plot_confusion_matrix(y_test , y_predicted_class)
```

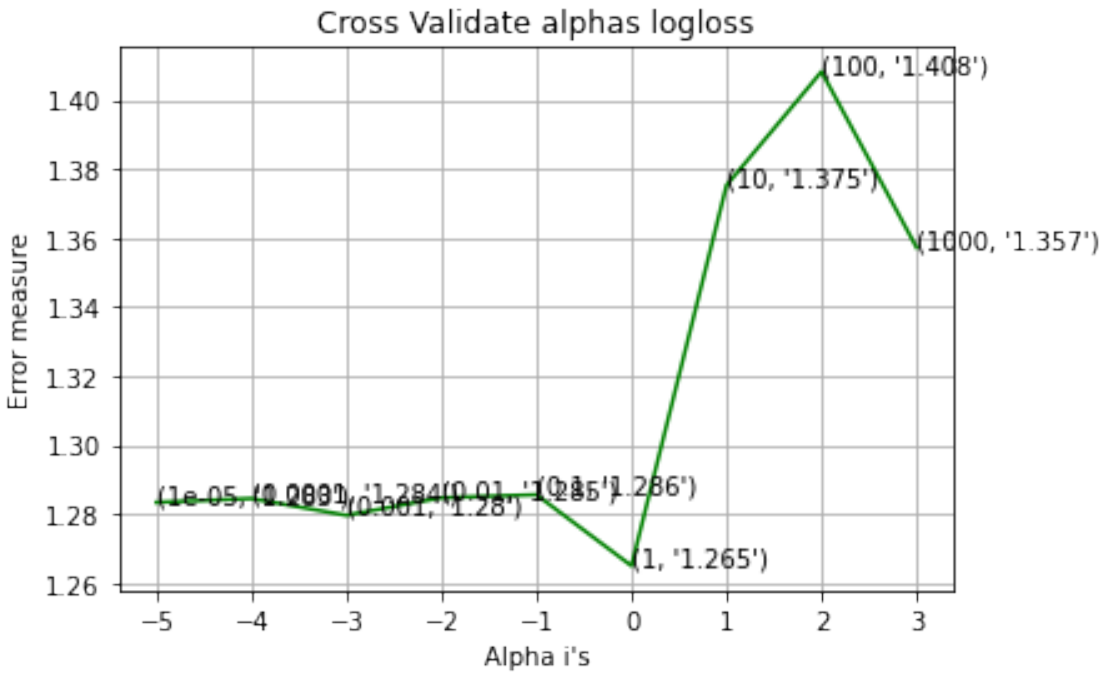
Function to report logloss.

[ ]:
```
def report_logloss(x_train, y_train, x_test, y_test, clf):
  clf.fit(x_train , y_train)
  sig_clf = CalibratedClassifierCV(clf , method='sigmoid')
  sig_clf.fit(x_train , y_train)
  y_predicted = sig_clf.predict_proba(x_test)
  return log_loss(y_test, y_predicted, label=clf.classes_ , eps=1e-15)
```

[ ]:
```
alpha = [0.00001, 0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000]
cv_log_error = []

for i in alpha:
  clf = MultinomialNB(alpha = i)
  clf.fit(train_onehotencode , y_train_onehotencode)
  sig_clf = CalibratedClassifierCV(clf , method='sigmoid')
  sig_clf.fit(train_onehotencode, y_train_onehotencode)
  y_predicted = sig_clf.predict_proba(cv_onehotencode)
```

```
  print("logloss for alpha",i,"is:",log_loss(y_cv_onehotencode , y_predicted,␣
 ↪labels= clf.classes_, eps=1e-15))
  cv_log_error.append(log_loss(y_cv_onehotencode , y_predicted, labels= clf.
 ↪classes_, eps=1e-15))

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

best_alpha = np.argmin(cv_log_error)
clf = MultinomialNB(alpha= alpha[best_alpha])
clf.fit(train_onehotencode , y_train_onehotencode)
sig_clf = CalibratedClassifierCV(clf, method = 'sigmoid')
sig_clf.fit(train_onehotencode, y_train_onehotencode)

y_predicted = sig_clf.predict_proba(cv_onehotencode)
print("Logloss for best alpha",alpha[best_alpha], "on cross validate data is:
 ↪",log_loss(y_cv_onehotencode, y_predicted, labels=clf.classes_, eps=1e-15))

y_predicted = sig_clf.predict_proba(train_onehotencode)
print("Logloss for best alpha", alpha[best_alpha], "on train data is:",␣
 ↪log_loss(y_train_onehotencode, y_predicted, labels= clf.classes_, eps=1e-15))

y_predicted = sig_clf.predict_proba(test_onehotencode)
print("Logloss for best alpha",alpha[best_alpha], "on test data is:",␣
 ↪log_loss(y_test_onehotencode, y_predicted, labels=clf.classes_, eps=1e-15))
```

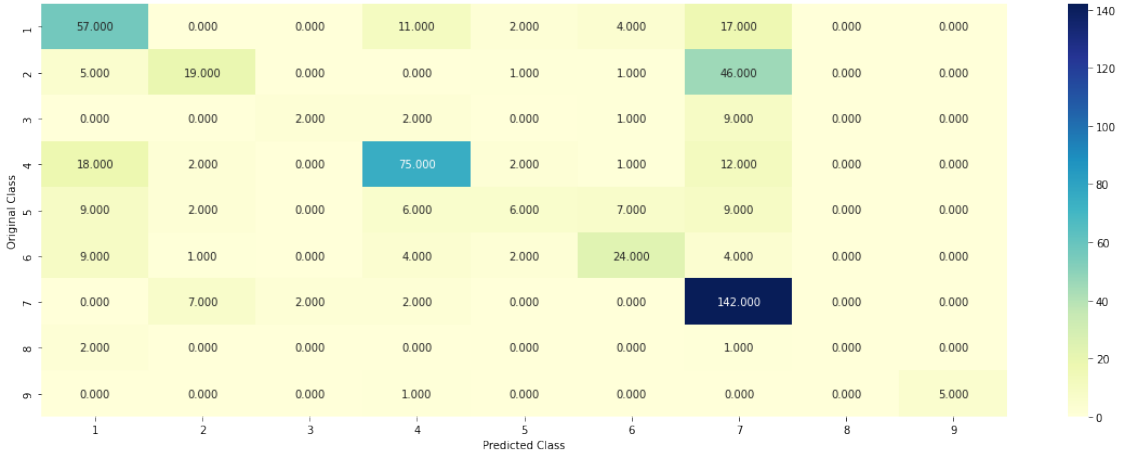
```
logloss for alpha 1e-05 is: 1.2833181370970674
logloss for alpha 0.0001 is: 1.2844286077024358
logloss for alpha 0.001 is: 1.2795595062874388
logloss for alpha 0.01 is: 1.2846549167632473
logloss for alpha 0.1 is: 1.2855336332082272
logloss for alpha 1 is: 1.2649179491814637
logloss for alpha 10 is: 1.3752816756612873
logloss for alpha 100 is: 1.40827194538743
logloss for alpha 1000 is: 1.3572584268295924
```




```
Logloss for best alpha 1 on cross validate data is: 1.2649179491814637
Logloss for best alpha 1 on train data is: 0.9302775084598898
Logloss for best alpha 1 on test data is: 1.279367004825177
```

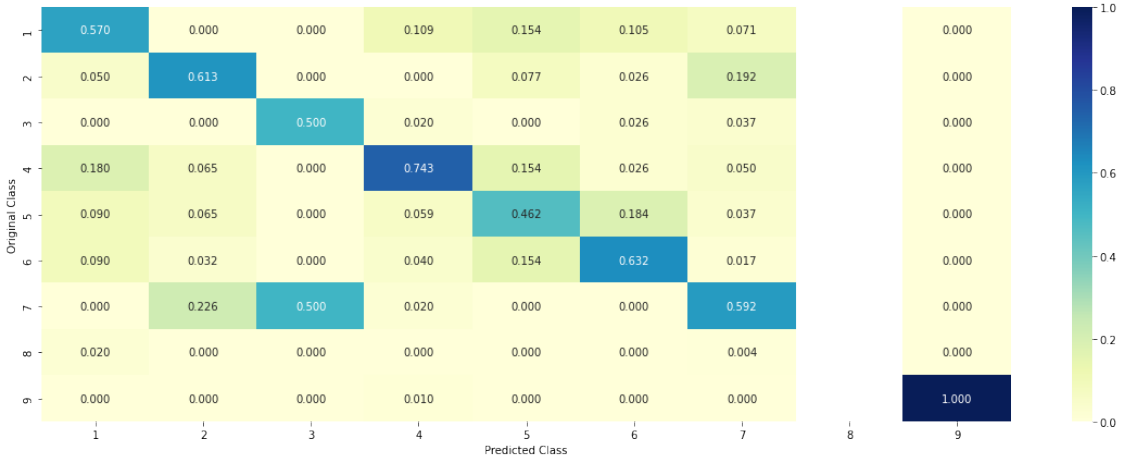
[ ]:
```
clf = MultinomialNB(alpha = alpha[best_alpha])
clf.fit(train_onehotencode, y_train_onehotencode)
sig_clf = CalibratedClassifierCV(clf , method='sigmoid')
sig_clf.fit(train_onehotencode, y_train_onehotencode)
y_predicted = sig_clf.predict_proba(cv_onehotencode)    # use sig_clf.
 ↪predict_proba
print("Logloss:", log_loss(y_cv_onehotencode , y_predicted))
print("Number of misclassified points: ",np.count_nonzero((sig_clf.
 ↪predict(cv_onehotencode) - y_cv_onehotencode)) / y_cv_onehotencode.shape[0])␣
 ↪    # use sig_clf.predict
plot_confusion_matrix(y_cv_onehotencode, sig_clf.predict(cv_onehotencode.
 ↪toarray()))
```

```
Logloss: 1.2649179491814637
Number of misclassified points:  0.37969924812030076
------------------------------------------------------------ Confusion matrix
------------------------------------------------------------
```



---------- Precision Martix axis=1 ----------

---------- Recall Martix axis=0 ----------



Testing the MultinomialNB model.

[ ]:
```
clf = MultinomialNB(alpha= alpha[best_alpha])
clf.fit(train_onehotencode,y_train_onehotencode)
test_point_index = 1
no_features = 100
predicted_cls = sig_clf.predict(test_onehotencode[test_point_index])
print("Predicted Class :", predicted_cls[0])
print("Predicted Class Probabilities:", np.round(sig_clf.
 ↪predict_proba(test_onehotencode[test_point_index]),4))
print("Actual Class :", y_test_onehotencode[test_point_index])
indices = np.argsort(-1*abs(clf.coef_))[predicted_cls-1][:,:no_features]
print("-"*50)
get_impfeature_names(indices[0], x_test['TEXT'].
 ↪iloc[test_point_index],x_test['Gene'].
 ↪iloc[test_point_index],x_test['Variation'].iloc[test_point_index],␣
 ↪no_features)
```

```
Predicted Class : 1
Predicted Class Probabilities: [[0.5951 0.0958 0.0242 0.1018 0.053  0.0384
0.0798 0.007  0.0049]]
Actual Class : 1
--------------------------------------------------
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-53-24efaa5195fa> in <module>()
      9 indices = np.argsort(-1*abs(clf.coef_))[predicted_cls-1][:,:no_features]
     10 print("-"*50)
---> 11 get_impfeature_names(indices[0], x_test['TEXT'].
 ↪iloc[test_point_index],x_test['Gene'].
 ↪iloc[test_point_index],x_test['Variation'].iloc[test_point_index], no_features)
```

```
<ipython-input-48-2c8844db3b15> in get_impfeature_names(indices, text, gene,␣
 ↪var, no_features)
     27                 print(i, "variation feature [{}] present in test data␣
 ↪point [{}]".format(word,yes_no))
     28         else:
---> 29             word = text_vec.get_feature_names()[v-(fea1_len+fea2_len)]
     30             yes_no = True if word in text.split() else False
     31             if yes_no:

/usr/local/lib/python3.7/dist-packages/sklearn/utils/deprecation.py in␣
 ↪wrapped(*args, **kwargs)
     86         def wrapped(*args, **kwargs):
     87             warnings.warn(msg, category=FutureWarning)
---> 88             return fun(*args, **kwargs)
     89
     90         wrapped.__doc__ = self._update_doc(wrapped.__doc__)

/usr/local/lib/python3.7/dist-packages/sklearn/feature_extraction/text.py in␣
 ↪get_feature_names(self)
   1429         self._check_vocabulary()
   1430
-> 1431         return [t for t, i in sorted(self.vocabulary_.items(),␣
 ↪key=itemgetter(1))]
   1432
   1433     def get_feature_names_out(self, input_features=None):

KeyboardInterrupt:
```

Modelling with KNN using Response Coding.

[ ]:
```
cv_log_error = []
alpha = [3, 5, 7, 9, 13, 17, 33, 53, 77, 99, 103, 109]
for i in alpha:
  clf = KNeighborsClassifier(n_jobs=-1, n_neighbors= i)
  clf.fit(train_ResponseCode, y_train_onehotencode)
  sig_clf = CalibratedClassifierCV(clf , method="sigmoid")
  sig_clf.fit(train_ResponseCode, y_train_onehotencode)
  predicted_y = sig_clf.predict_proba(cv_ResponseCode)
  cv_log_error.append(log_loss(y_cv_onehotencode, predicted_y, labels = clf.
 ↪classes_ , eps=1e-15))
  print("Logloss is: ", log_loss(y_cv_onehotencode, predicted_y, labels = clf.
 ↪classes_ , eps=1e-15))

fig,ax = plt.subplots()
ax.plot(alpha , cv_log_error , c='g')
for i , txt in enumerate(np.round(cv_log_error,3)):
```

```
  ax.annotate((alpha[i], str(txt)) , (alpha[i] , np.round(cv_log_error[i],3)))
plt.xlabel("Alpha i's")
plt.ylabel("Error loss")
plt.title("Cross Validate Log loss")
plt.grid()
plt.show()

best_alpha = np.argmin(cv_log_error)
clf = KNeighborsClassifier(n_jobs= -1, n_neighbors= alpha[best_alpha])
clf.fit(train_ResponseCode, y_train_onehotencode)
sig_clf = CalibratedClassifierCV(clf, method='sigmoid')
sig_clf.fit(train_ResponseCode, y_train_onehotencode)

predicted_y = sig_clf.predict_proba(test_ResponseCode)
print("Logloss for best alpha" , alpha[best_alpha],"Test data:␣
 ↪",log_loss(y_test_onehotencode , predicted_y , labels = clf.classes_,␣
 ↪eps=1e-15))

predicted_y = sig_clf.predict_proba(train_ResponseCode)
print("Logloss for " , alpha[best_alpha],"Train data:␣
 ↪",log_loss(y_train_onehotencode , predicted_y , labels = clf.classes_,␣
 ↪eps=1e-15))

predicted_y = sig_clf.predict_proba(cv_ResponseCode)
print("Logloss for " , alpha[best_alpha]," Cross Validate data:␣
 ↪",log_loss(y_cv_onehotencode , predicted_y , labels = clf.classes_,␣
 ↪eps=1e-15))
```

[ ]:
```
clf = KNeighborsClassifier(n_neighbors = alpha[best_alpha] , n_jobs= -1)
predict_and_plot_confusion_matrix(train_ResponseCode, y_train_onehotencode ,␣
 ↪cv_ResponseCode, y_cv_onehotencode , clf)
```

Modelling with KNN using OneHot Encoding.

[ ]:
```
cv_log_error = []
alpha = [3, 5, 7, 9, 13, 17, 33, 53, 77, 99, 103, 109]
for i in alpha:
  clf = KNeighborsClassifier(n_jobs=-1, n_neighbors= i)
  clf.fit(train_onehotencode, y_train_onehotencode)
  sig_clf = CalibratedClassifierCV(clf , method="sigmoid")
  sig_clf.fit(train_onehotencode, y_train_onehotencode)
  predicted_y = sig_clf.predict_proba(cv_onehotencode)
  cv_log_error.append(log_loss(y_cv, predicted_y, labels = clf.classes_ ,␣
 ↪eps=1e-15))
  print("Logloss is: ", log_loss(y_cv, predicted_y, labels = clf.classes_ ,␣
 ↪eps=1e-15))
```

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

best_alpha = np.argmin(cv_log_error)
clf = KNeighborsClassifier(n_jobs= -1, n_neighbors= alpha[best_alpha])
clf.fit(train_onehotencode, y_train_onehotencode)
sig_clf = CalibratedClassifierCV(clf, method='sigmoid')
sig_clf.fit(train_onehotencode, y_train_onehotencode)

predicted_y = sig_clf.predict_proba(test_onehotencode)
print("Logloss for best alpha" , alpha[best_alpha],"Test data:␣
 ↪",log_loss(y_test_onehotencode , predicted_y , labels = clf.classes_,␣
 ↪eps=1e-15))

predicted_y = sig_clf.predict_proba(train_onehotencode)
print("Logloss for " , alpha[best_alpha],"Train data:␣
 ↪",log_loss(y_train_onehotencode , predicted_y , labels = clf.classes_,␣
 ↪eps=1e-15))

predicted_y = sig_clf.predict_proba(cv_onehotencode)
print("Logloss for " , alpha[best_alpha]," Cross Validate data:␣
 ↪",log_loss(y_cv_onehotencode , predicted_y , labels = clf.classes_,␣
 ↪eps=1e-15))
```

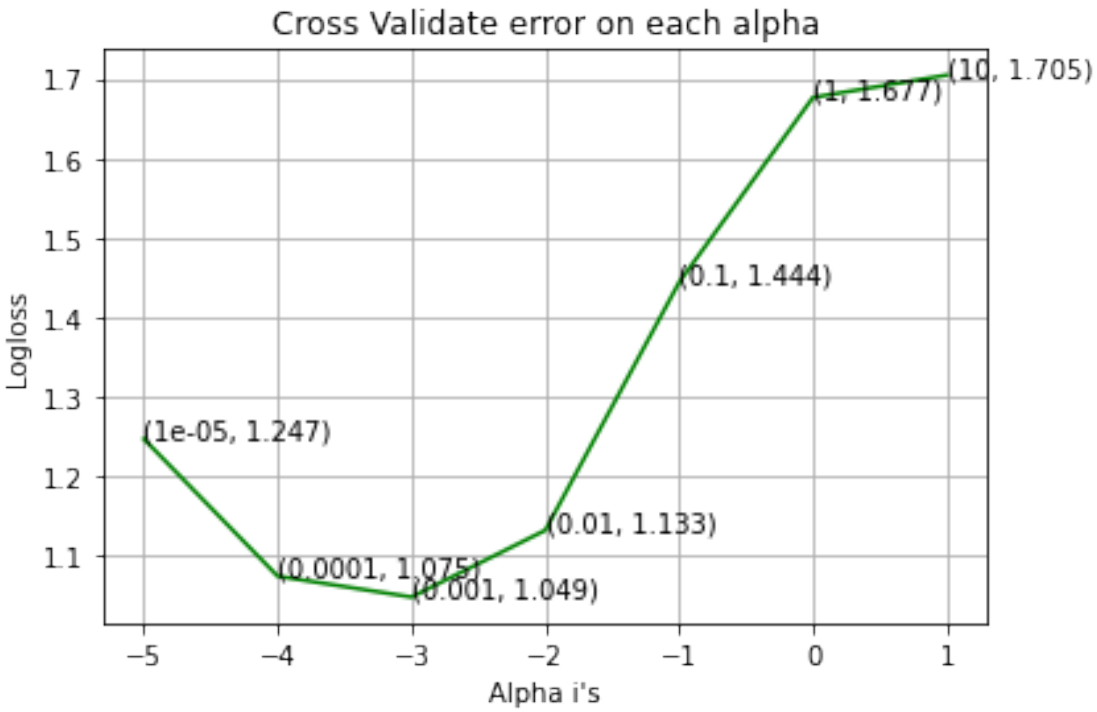
[ ]:
```
test_datapoint = 1
clf = KNeighborsClassifier(n_jobs=-1, n_neighbors=alpha[best_alpha])
clf.fit(train_ResponseCode, y_train_onehotencode)
sig_clf = CalibratedClassifierCV(clf, method= 'sigmoid')
predicted_class = sig_clf.predict(test_ResponseCode[test_datapoint])
```

Modelling by Logistic Regression with balance class weights using OneHot Encoding.

[ ]:
```
alpha = [10 ** x for x in range(-5,2)]
cv_log_error = []
for i in alpha:
  clf = SGDClassifier(alpha = i , class_weight = 'balanced', penalty = 'l2' ,␣
 ↪loss= 'log' , random_state = 42 , n_jobs=-1)
  clf.fit(train_onehotencode , y_train_onehotencode)
  sig_clf = CalibratedClassifierCV(clf , method='sigmoid')
  sig_clf.fit(train_onehotencode , y_train_onehotencode)
```

```
  y_predicted = sig_clf.predict_proba(cv_onehotencode)
  cv_log_error.append(log_loss(y_cv_onehotencode , y_predicted , labels = clf.
 ↪classes_ , eps=1e-15))
  print("For alpha ",i," the log loss is: ", log_loss(y_cv_onehotencode ,␣
 ↪y_predicted , labels= clf.classes_ , eps= 1e-15))

fig , ax = plt.subplots()
ax.plot(np.log10(alpha) , cv_log_error , c='g')
for i , txt in enumerate(np.round(cv_log_error , 3)):
  ax.annotate((alpha[i] , np.round(txt,3)) , (np.log10(alpha[i]) ,␣
 ↪cv_log_error[i]))
plt.title("Cross Validate error on each alpha")
plt.xticks(np.log10(alpha))
plt.xlabel("Alpha i's")
plt.ylabel("Logloss")
plt.grid()
plt.show()

best_alpha = np.argmin(cv_log_error)

clf = SGDClassifier(alpha = alpha[best_alpha] ,class_weight='balanced' ,␣
 ↪penalty = 'l2', loss = 'log', random_state=42, n_jobs=-1)
clf.fit(train_text_onehotencode , y_train_onehotencode)
sig_clf = CalibratedClassifierCV(clf , method = 'sigmoid')
sig_clf.fit(train_onehotencode , y_train_onehotencode)
y_predicted = sig_clf.predict_proba(test_onehotencode)
print("For alpha ",alpha[best_alpha],"the test logloss is: ",␣
 ↪log_loss(y_test_onehotencode , y_predicted , labels= clf.classes_ ,␣
 ↪eps=1e-15))
y_predicted = sig_clf.predict_proba(cv_onehotencode)
print("For alpha ",alpha[best_alpha],"the cross validate logloss is: ",␣
 ↪log_loss(y_cv_onehotencode , y_predicted , labels= clf.classes_ , eps=1e-15))
y_predicted = sig_clf.predict_proba(train_onehotencode)
print("For alpha ",alpha[best_alpha],"the train logloss is: ",␣
 ↪log_loss(y_train_onehotencode , y_predicted , labels= clf.classes_ ,␣
 ↪eps=1e-15))
```

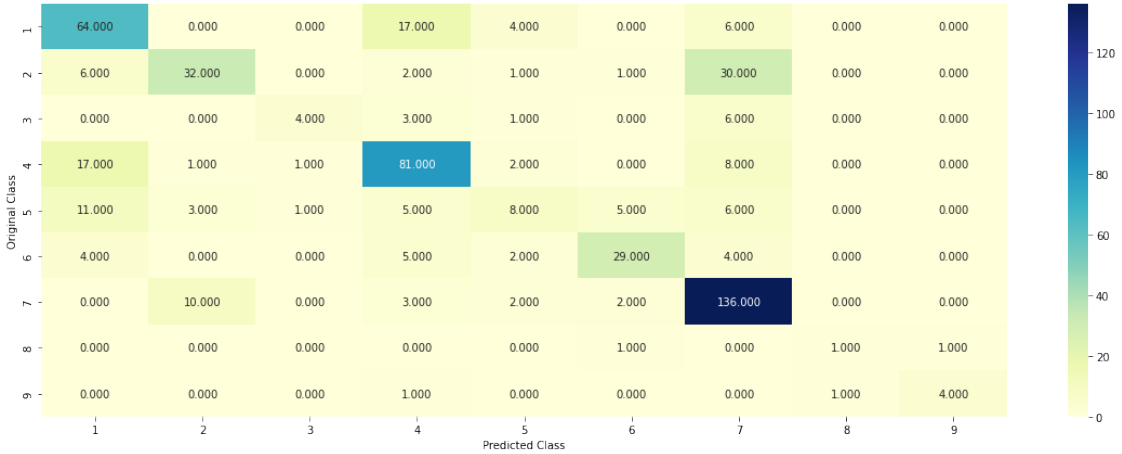
```
For alpha  1e-05  the log loss is:  1.2473875785582018
For alpha  0.0001  the log loss is:  1.0748413470926341
For alpha  0.001  the log loss is:  1.0485951395432134
For alpha  0.01  the log loss is:  1.1325258669671847
For alpha  0.1  the log loss is:  1.444243997320998
For alpha  1  the log loss is:  1.6766154984589832
For alpha  10  the log loss is:  1.7046724931949284
```




```
For alpha  0.001 the test logloss is:  1.0707726300779963
For alpha  0.001 the cross validate logloss is:  1.0485951395432134
For alpha  0.001 the train logloss is:  0.5288145361128199
```

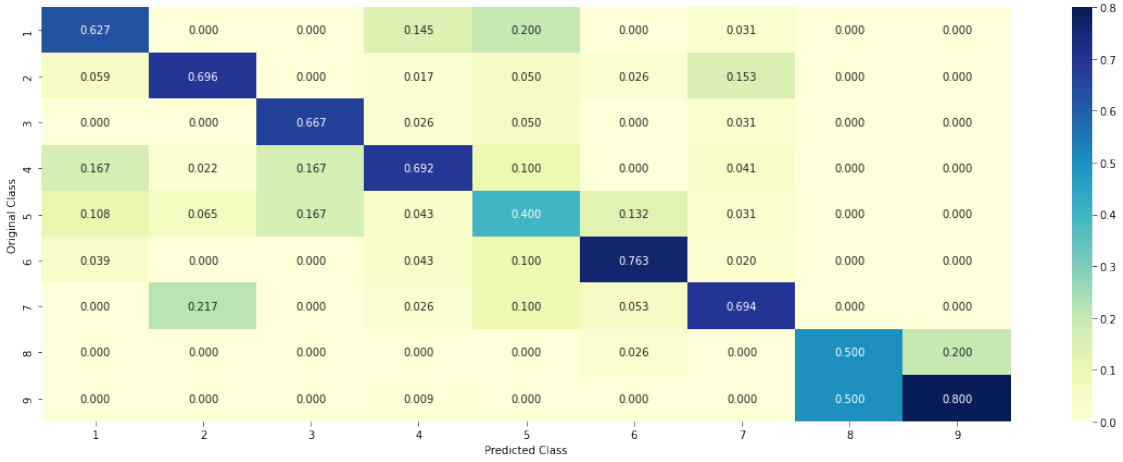
[ ]:
```
clf = SGDClassifier(alpha = alpha[best_alpha] , class_weight = 'balanced',␣
 ↪penalty = 'l2' , loss= 'log' , random_state = 42 , n_jobs=-1)
predict_and_plot_confusion_matrix(train_onehotencode , y_train_onehotencode,␣
 ↪cv_onehotencode, y_cv_onehotencode, clf)
```

```
Logloss is:  1.0485951395432134
Number of mis-classified points : 0.325187969924812
------------------------------------------------------------ Confusion matrix
------------------------------------------------------------
```



---------- Precision Martix axis=1 ----------

---------- Recall Martix axis=0 ----------

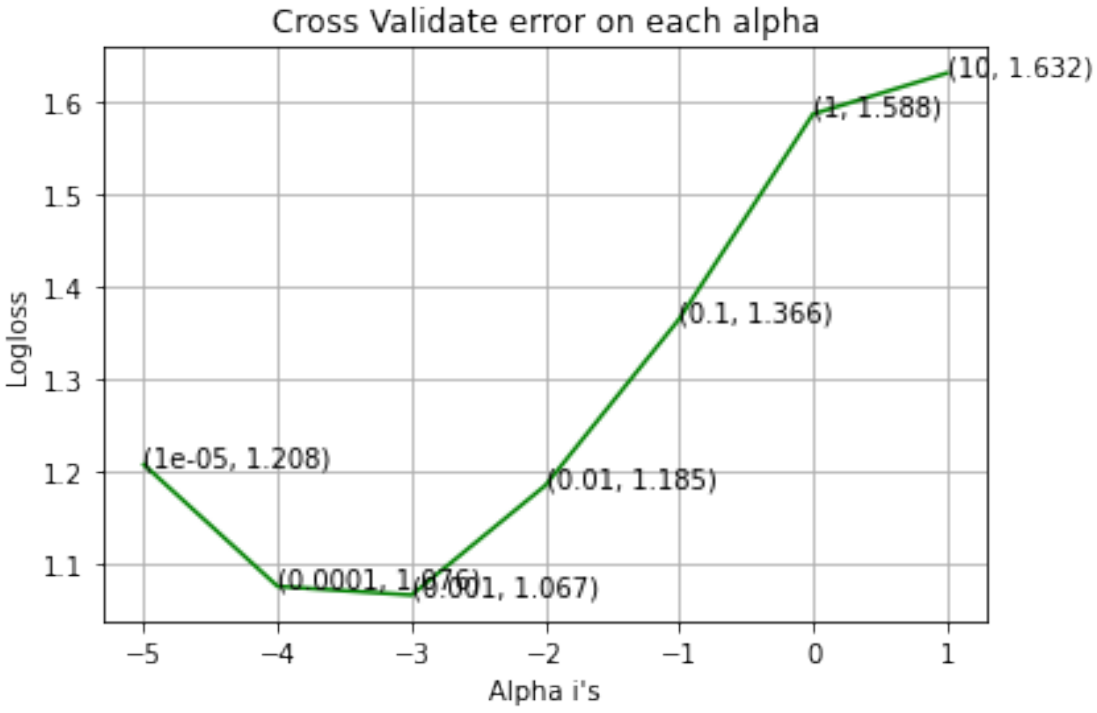


Modelling by Logistic Regression without balance class weights using OneHot Encoding.

[ ]:
```
alpha = [10 ** x for x in range(-5,2)]
cv_log_error = []
for i in alpha:
  clf = SGDClassifier(alpha = i , penalty = 'l2' , loss= 'log' , random_state =␣
 ↪42 , n_jobs=-1)
  clf.fit(train_onehotencode , y_train_onehotencode)
  sig_clf = CalibratedClassifierCV(clf , method='sigmoid')
  sig_clf.fit(train_onehotencode , y_train_onehotencode)
  y_predicted = sig_clf.predict_proba(cv_onehotencode)
  cv_log_error.append(log_loss(y_cv_onehotencode , y_predicted , labels = clf.
 ↪classes_ , eps=1e-15))
  print("For alpha ",i," the log loss is: ", log_loss(y_cv_onehotencode ,␣
 ↪y_predicted , labels= clf.classes_ , eps= 1e-15))

fig , ax = plt.subplots()
ax.plot(np.log10(alpha) , cv_log_error , c='g')
for i , txt in enumerate(np.round(cv_log_error , 3)):
  ax.annotate((alpha[i] , np.round(txt,3)) , (np.log10(alpha[i]) ,␣
 ↪cv_log_error[i]))
plt.title("Cross Validate error on each alpha")
plt.xticks(np.log10(alpha))
plt.xlabel("Alpha i's")
plt.ylabel("Logloss")
plt.grid()
plt.show()

best_alpha = np.argmin(cv_log_error)
```

```
clf = SGDClassifier(alpha = alpha[best_alpha] , penalty = 'l2', loss = 'log',␣
 ↪random_state=42, n_jobs=-1)
clf.fit(train_text_onehotencode , y_train_onehotencode)
sig_clf = CalibratedClassifierCV(clf , method = 'sigmoid')
sig_clf.fit(train_onehotencode , y_train_onehotencode)
y_predicted = sig_clf.predict_proba(test_onehotencode)
print("For alpha ",alpha[best_alpha],"the test logloss is: ",␣
 ↪log_loss(y_test_onehotencode , y_predicted , labels= clf.classes_ ,␣
 ↪eps=1e-15))
y_predicted = sig_clf.predict_proba(cv_onehotencode)
print("For alpha ",alpha[best_alpha],"the cross validate logloss is: ",␣
 ↪log_loss(y_cv_onehotencode , y_predicted , labels= clf.classes_ , eps=1e-15))
y_predicted = sig_clf.predict_proba(train_onehotencode)
print("For alpha ",alpha[best_alpha],"the train logloss is: ",␣
 ↪log_loss(y_train_onehotencode , y_predicted , labels= clf.classes_ ,␣
 ↪eps=1e-15))
```

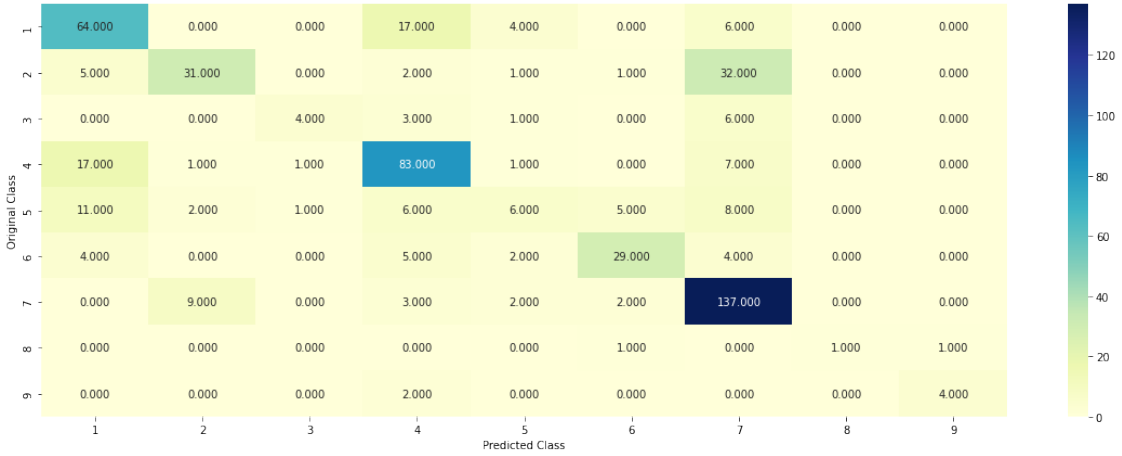
```
For alpha  1e-05  the log loss is:  1.2081297923398033
For alpha  0.0001  the log loss is:  1.0762558401778575
For alpha  0.001  the log loss is:  1.066821961771742
For alpha  0.01  the log loss is:  1.185345300279447
For alpha  0.1  the log loss is:  1.3655363881095843
For alpha  1  the log loss is:  1.5875084802850978
For alpha  10  the log loss is:  1.6317954052105024
```



```
For alpha  0.001 the test logloss is:  1.0913634713092308
For alpha  0.001 the cross validate logloss is:  1.066821961771742
For alpha  0.001 the train logloss is:  0.5316078936690953
```

[ ]:
```
clf = SGDClassifier(alpha = alpha[best_alpha] , penalty = 'l2' , loss= 'log' ,␣
 ↪random_state = 42 , n_jobs=-1)
predict_and_plot_confusion_matrix(train_onehotencode , y_train_onehotencode,␣
 ↪cv_onehotencode, y_cv_onehotencode, clf)
```

```
Logloss is:  1.066821961771742
Number of mis-classified points : 0.325187969924812
------------------------------------------------------------ Confusion matrix
------------------------------------------------------------
```




```
------------------------------------------------------------ Precision Martix
axis=1 ------------------------------------------------------------
```

```
---------------------------------------------------------- Recall Martix
axis=0 ----------------------------------------------------------
```

[ ]:
```
clf = SGDClassifier(alpha=alpha[best_alpha], penalty='l2', loss='log',␣
 ↪random_state=42)
clf.fit(train_onehotencode,y_train_onehotencode)
test_point_index = 1
no_features = 100
predicted_cls = sig_clf.predict(test_onehotencode[test_point_index])
print("Predicted Class :", predicted_cls[0], '\n',np.max(np.round(sig_clf.
 ↪predict_proba(test_onehotencode[test_point_index]),4)*100), "% ")
print("Predicted Class Probabilities:", np.round(sig_clf.
 ↪predict_proba(test_onehotencode[test_point_index]),4))
print("Actual Class :", y_test_onehotencode[test_point_index] )
indices = np.argsort(-1*abs(clf.coef_))[predicted_cls-1][:,:no_features]
print("-"*50)
get_impfeature_names(indices[0], x_test['TEXT'].
 ↪iloc[test_point_index],x_test['Gene'].
 ↪iloc[test_point_index],x_test['Variation'].iloc[test_point_index],␣
 ↪no_features)
```

```
Predicted Class : 4
 69.27 % sure that the
Predicted Class Probabilities: [[0.06   0.0778 0.0207 0.6927 0.0457 0.0384
0.0549 0.0055 0.0043]]
Actual Class : 4
--------------------------------------------------
29 Text feature [identified] present in test data point [True]
30 Text feature [well] present in test data point [True]
```

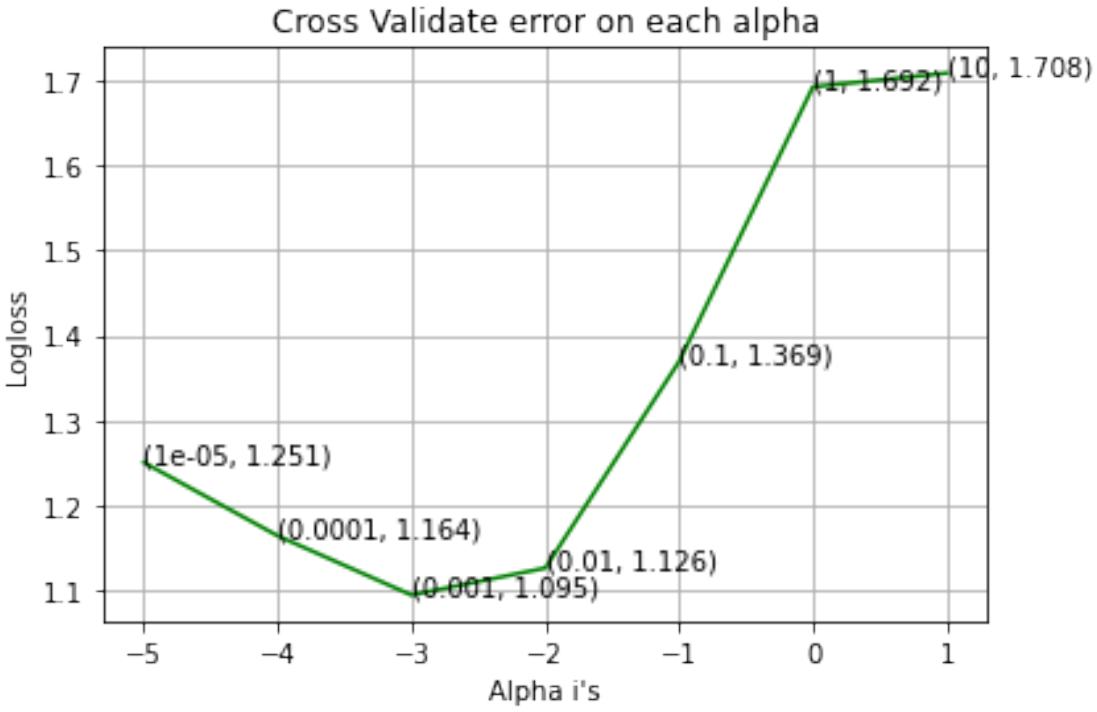
31 Text feature [observed] present in test data point [True]
33 Text feature [additional] present in test data point [True]
34 Text feature [recently] present in test data point [True]
37 Text feature [previously] present in test data point [True]
38 Text feature [using] present in test data point [True]
41 Text feature [mutation] present in test data point [True]
42 Text feature [found] present in test data point [True]
43 Text feature [one] present in test data point [True]
44 Text feature [studies] present in test data point [True]
45 Text feature [also] present in test data point [True]
46 Text feature [however] present in test data point [True]
47 Text feature [10] present in test data point [True]
49 Text feature [compared] present in test data point [True]
50 Text feature [table] present in test data point [True]
51 Text feature [showed] present in test data point [True]
52 Text feature [three] present in test data point [True]
54 Text feature [may] present in test data point [True]
55 Text feature [used] present in test data point [True]
56 Text feature [respectively] present in test data point [True]
57 Text feature [presence] present in test data point [True]
58 Text feature [data] present in test data point [True]
59 Text feature [addition] present in test data point [True]
60 Text feature [analysis] present in test data point [True]
61 Text feature [expected] present in test data point [True]
63 Text feature [independent] present in test data point [True]
64 Text feature [12] present in test data point [True]
65 Text feature [mutations] present in test data point [True]
66 Text feature [different] present in test data point [True]
67 Text feature [higher] present in test data point [True]
68 Text feature [present] present in test data point [True]
69 Text feature [two] present in test data point [True]
70 Text feature [identify] present in test data point [True]
71 Text feature [including] present in test data point [True]
72 Text feature [known] present in test data point [True]
73 Text feature [15] present in test data point [True]
74 Text feature [confirmed] present in test data point [True]
75 Text feature [identification] present in test data point [True]
76 Text feature [reported] present in test data point [True]
77 Text feature [highly] present in test data point [True]
79 Text feature [cancer] present in test data point [True]
80 Text feature [similar] present in test data point [True]
82 Text feature [clinical] present in test data point [True]
84 Text feature [25] present in test data point [True]
85 Text feature [selected] present in test data point [True]
86 Text feature [small] present in test data point [True]
88 Text feature [obtained] present in test data point [True]
89 Text feature [cell] present in test data point [True]
90 Text feature [confirm] present in test data point [True]

```
91 Text feature [significant] present in test data point [True]
92 Text feature [discussion] present in test data point [True]
93 Text feature [total] present in test data point [True]
95 Text feature [increased] present in test data point [True]
96 Text feature [16] present in test data point [True]
97 Text feature [1b] present in test data point [True]
98 Text feature [shown] present in test data point [True]
99 Text feature [single] present in test data point [True]
Out of the top  100  features  58 are present in query point
```

[ ]:
```
alpha = [10 ** x for x in range(-5,2)]
cv_log_error = []
for i in alpha:
  clf = SGDClassifier(alpha = i , class_weight = 'balanced', penalty = 'l2' ,␣
 ↪loss= 'hinge' , random_state = 42 , n_jobs=-1)
  clf.fit(train_onehotencode , y_train_onehotencode)
  sig_clf = CalibratedClassifierCV(clf , method='sigmoid')
  sig_clf.fit(train_onehotencode , y_train_onehotencode)

  y_predicted = sig_clf.predict_proba(cv_onehotencode)
  cv_log_error.append(log_loss(y_cv_onehotencode , y_predicted , labels = clf.
 ↪classes_ , eps=1e-15))
  print("For C= ",i," the log loss is: ", log_loss(y_cv_onehotencode ,␣
 ↪y_predicted , labels= clf.classes_ , eps= 1e-15))

fig , ax = plt.subplots()
ax.plot(np.log10(alpha) , cv_log_error , c='g')
for i , txt in enumerate(np.round(cv_log_error , 3)):
  ax.annotate((alpha[i] , np.round(txt,3)) , (np.log10(alpha[i]) ,␣
 ↪cv_log_error[i]))
plt.title("Cross Validate error on each alpha")
plt.xticks(np.log10(alpha))
plt.xlabel("Alpha i's")
plt.ylabel("Logloss")
plt.grid()
plt.show()

best_alpha = np.argmin(cv_log_error)

clf = SGDClassifier(alpha = alpha[best_alpha] ,class_weight='balanced' ,␣
 ↪penalty = 'l2', loss = 'hinge', random_state=42, n_jobs=-1)
clf.fit(train_text_onehotencode , y_train_onehotencode)
sig_clf = CalibratedClassifierCV(clf , method = 'sigmoid')
sig_clf.fit(train_onehotencode , y_train_onehotencode)
y_predicted = sig_clf.predict_proba(test_onehotencode)
```

```
print("For C= ",alpha[best_alpha],"the test logloss is: ",␣
 ↪log_loss(y_test_onehotencode , y_predicted , labels= clf.classes_ ,␣
 ↪eps=1e-15))
y_predicted = sig_clf.predict_proba(cv_onehotencode)
print("For C= ",alpha[best_alpha],"the cross validate logloss is: ",␣
 ↪log_loss(y_cv_onehotencode , y_predicted , labels= clf.classes_ , eps=1e-15))
y_predicted = sig_clf.predict_proba(train_onehotencode)
print("For C= ",alpha[best_alpha],"the train logloss is: ",␣
 ↪log_loss(y_train_onehotencode , y_predicted , labels= clf.classes_ ,␣
 ↪eps=1e-15))
```

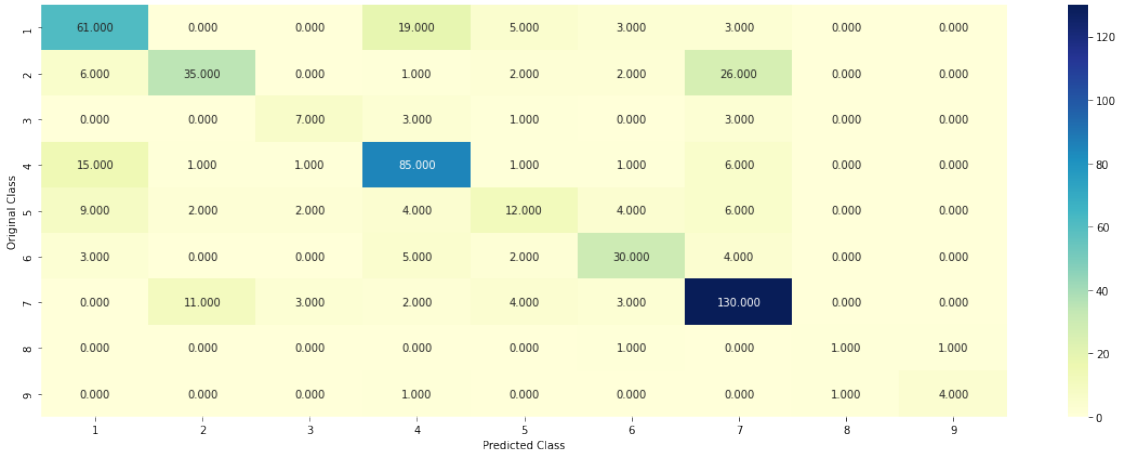
```
For C=  1e-05  the log loss is:  1.2506158250899744
For C=  0.0001  the log loss is:  1.1643879914712791
For C=  0.001  the log loss is:  1.0947252970329808
For C=  0.01  the log loss is:  1.1264490587268021
For C=  0.1  the log loss is:  1.3692839515352673
For C=  1  the log loss is:  1.6920532196555274
For C=  10  the log loss is:  1.7081851551357823
```



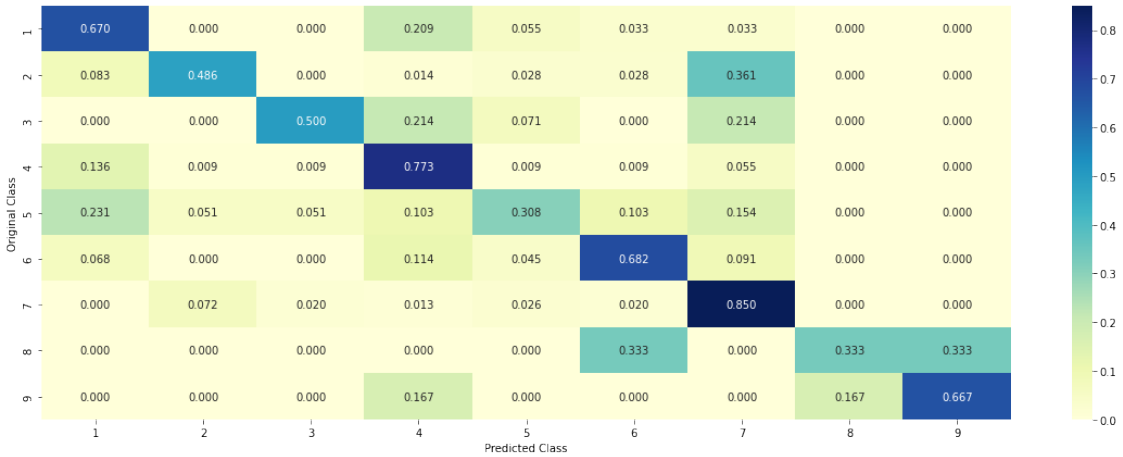

```
For C=  0.001 the test logloss is:  1.1529109431637767
For C=  0.001 the cross validate logloss is:  1.0947252970329808
For C=  0.001 the train logloss is:  0.5481320621722389
```

```
[ ]: clf = SGDClassifier(alpha = alpha[best_alpha] ,class_weight='balanced' ,␣
      ↪penalty = 'l2', loss = 'hinge', random_state=42, n_jobs=-1)
     predict_and_plot_confusion_matrix(train_onehotencode, y_train_onehotencode ,␣
      ↪cv_onehotencode, y_cv_onehotencode, clf)
```

```
Logloss is:  1.0947252970329808
Number of mis-classified points : 0.31390977443609025
------------------------------------------------------------ Confusion matrix
-----------------------------------------------------------
```




```
------------------------------------------------------------ Precision Martix
axis=1 -------------------------------------------------------------
```




```
------------------------------------------------------------ Recall Martix
axis=0 -------------------------------------------------------------
```

[ ]:
```
clf = SGDClassifier(alpha=alpha[best_alpha], class_weight='balanced',␣
 ↪penalty='l2', loss='hinge', random_state=42)
clf.fit(train_onehotencode,y_train_onehotencode)
test_point_index = 1
no_features = 500
predicted_cls = sig_clf.predict(test_onehotencode[test_point_index])
print("Predicted Class :", predicted_cls[0], '\n',np.max(np.round(sig_clf.
 ↪predict_proba(test_onehotencode[test_point_index]),4)*100), "% ")
print("Predicted Class Probabilities:", np.round(sig_clf.
 ↪predict_proba(test_onehotencode[test_point_index]),4))
print("Actual Class :", y_test_onehotencode[test_point_index] )
indices = np.argsort(-1*abs(clf.coef_))[predicted_cls-1][:,:no_features]
print("-"*50)
get_impfeature_names(indices[0], x_test['TEXT'].
 ↪iloc[test_point_index],x_test['Gene'].
 ↪iloc[test_point_index],x_test['Variation'].iloc[test_point_index],␣
 ↪no_features)
```

```
Predicted Class : 4
 78.10000000000001 %
Predicted Class Probabilities: [[0.0165 0.0576 0.005  0.781  0.0321 0.0155
0.0837 0.004  0.0045]]
Actual Class : 4
--------------------------------------------------
195 Text feature [spell] present in test data point [True]
254 Text feature [n239s] present in test data point [True]
330 Text feature [foa] present in test data point [True]
332 Text feature [y234c] present in test data point [True]
379 Text feature [dbs] present in test data point [True]
400 Text feature [r158l] present in test data point [True]
468 Text feature [stressgen] present in test data point [True]
```

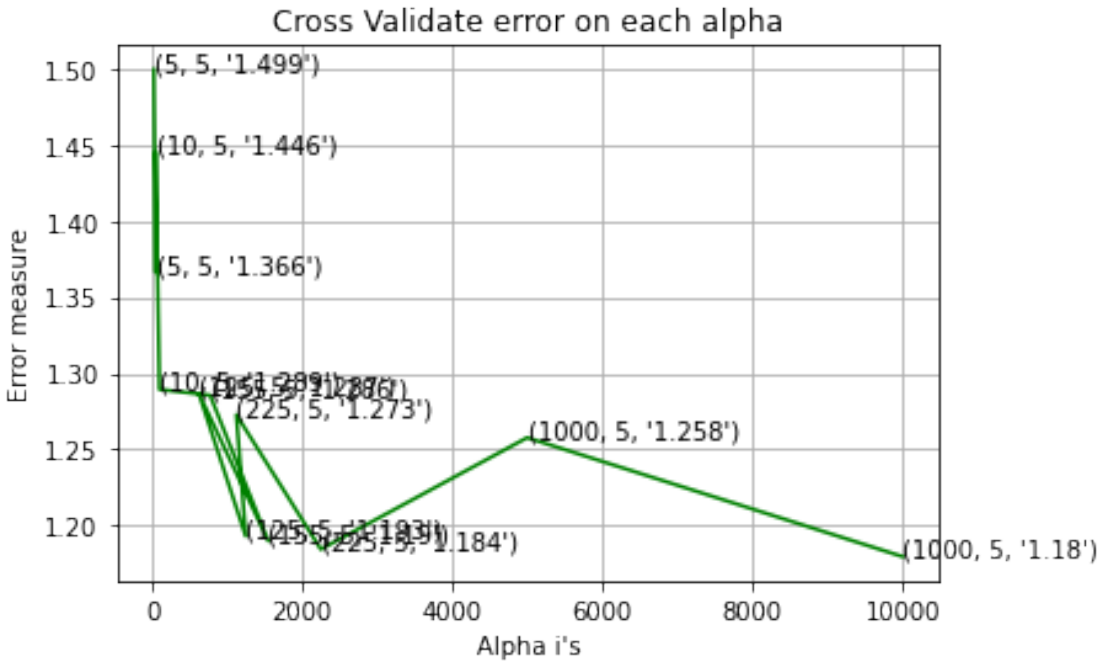
```
Out of the top  500  features  7 are present in query point
```

Modelling with Random Forest using OneHot Encoding.

[ ]:
```
alpha = [5,10,155,125,225,1000]
max_depth = [5,10]
cv_log_error = []
for i in alpha:
  for j in max_depth:
    clf = RandomForestClassifier(n_estimators = i, max_depth=j,␣
 ↪criterion='gini' , random_state = 42 , n_jobs=-1)
    clf.fit(train_onehotencode , y_train_onehotencode)
    sig_clf = CalibratedClassifierCV(clf , method='sigmoid')
    sig_clf.fit(train_onehotencode , y_train_onehotencode)

    y_predicted = sig_clf.predict_proba(cv_onehotencode)
    cv_log_error.append(log_loss(y_cv_onehotencode , y_predicted , labels = clf.
 ↪classes_ , eps=1e-15))
    print("For n_estimators= ",i," and the max_depth ",j, "logloss is: ",␣
 ↪log_loss(y_cv_onehotencode , y_predicted , labels= clf.classes_ , eps=␣
 ↪1e-15))

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

best_alpha = np.argmin(cv_log_error)

clf = RandomForestClassifier(n_estimators = alpha[int(best_alpha/2)]␣
 ↪,max_depth=max_depth[int(j%2)], criterion = 'gini', random_state=42,␣
 ↪n_jobs=-1)
clf.fit(train_onehotencode , y_train_onehotencode)
sig_clf = CalibratedClassifierCV(clf , method = 'sigmoid')
sig_clf.fit(train_onehotencode , y_train_onehotencode)
y_predicted = sig_clf.predict_proba(test_onehotencode)
print("For value of best estimator ",alpha[int(best_alpha/2)],"the test logloss␣
 ↪is: ", log_loss(y_test_onehotencode , y_predicted , labels= clf.classes_ ,␣
 ↪eps=1e-15))
y_predicted = sig_clf.predict_proba(cv_onehotencode)
```

```
print("For value of best estimator ",alpha[int(best_alpha/2)],"the cross␣
 ↪validate logloss is: ", log_loss(y_cv_onehotencode , y_predicted , labels=␣
 ↪clf.classes_ , eps=1e-15))
y_predicted = sig_clf.predict_proba(train_onehotencode)
print("For value of best estimator ",alpha[int(best_alpha/2)],"the train␣
 ↪logloss is: ", log_loss(y_train_onehotencode , y_predicted , labels= clf.
 ↪classes_ , eps=1e-15))
```

```
For n_estimators=  5  and the max_depth  5 logloss is:  1.4994687320531899
For n_estimators=  5  and the max_depth  10 logloss is:  1.3663067577448709
For n_estimators=  10  and the max_depth  5 logloss is:  1.445546235693778
For n_estimators=  10  and the max_depth  10 logloss is:  1.2894108103488713
For n_estimators=  155  and the max_depth  5 logloss is:  1.285538856272138
For n_estimators=  155  and the max_depth  10 logloss is:  1.1898160739367794
For n_estimators=  125  and the max_depth  5 logloss is:  1.2867590923292425
For n_estimators=  125  and the max_depth  10 logloss is:  1.192998090005966
For n_estimators=  225  and the max_depth  5 logloss is:  1.2727958979247436
For n_estimators=  225  and the max_depth  10 logloss is:  1.1844112130579874
For n_estimators=  1000  and the max_depth  5 logloss is:  1.2577234208946462
For n_estimators=  1000  and the max_depth  10 logloss is:  1.1796728464443378
```

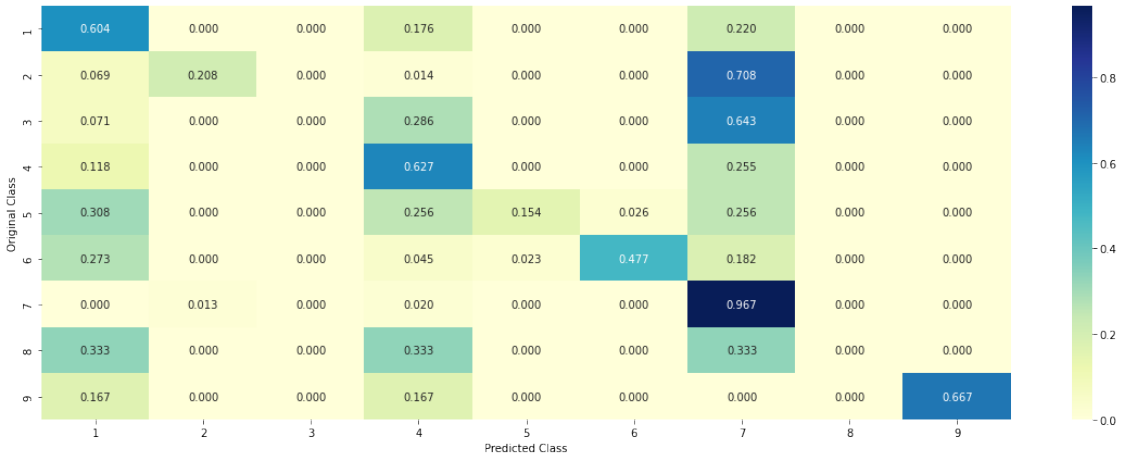



```
For value of best estimator  1000 the test logloss is:  1.2542306850241645
For value of best estimator  1000 the cross validate logloss is:
1.2577234208946462
For value of best estimator  1000 the train logloss is:  1.0308767091863429
```

[ ]:
```
clf = RandomForestClassifier(n_estimators = alpha[int(best_alpha/2)],␣
 ↪,max_depth=max_depth[int(j%2)], criterion = 'gini', random_state=42,␣
 ↪n_jobs=-1)
predict_and_plot_confusion_matrix(train_onehotencode, y_train_onehotencode,␣
 ↪cv_onehotencode, y_cv_onehotencode, clf)
```

```
Logloss is:  1.2577234208946462
Number of mis-classified points : 0.40225563909774437
------------------------------------------------------------ Confusion matrix
-----------------------------------------------------------
```


```
------------------------------------------------------------ Precision Martix
axis=1 -------------------------------------------------------------
```



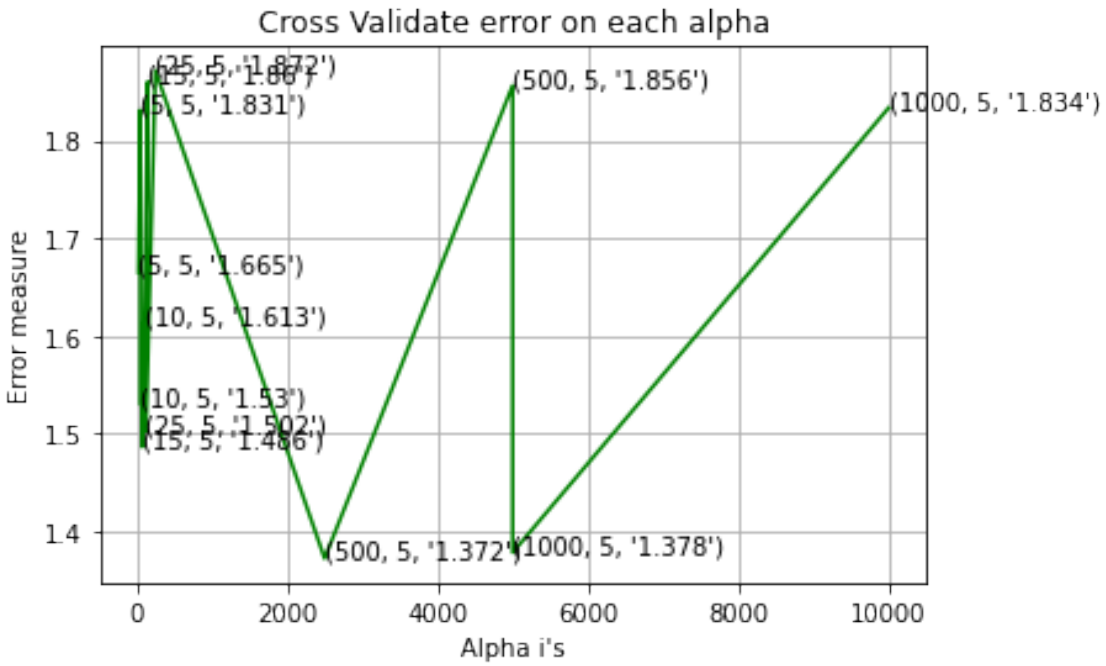

```
------------------------------------------------------------ Recall Martix
axis=0 -------------------------------------------------------------
```

Modelling with Random Forest using Response Coding.

[ ]:
```
alpha = [5,10,15,25,500,1000]
max_depth = [5,10]
cv_log_error = []
for i in alpha:
  for j in max_depth:
    clf = RandomForestClassifier(n_estimators = i, max_depth=j,␣
 ↪criterion='gini' , random_state = 42 , n_jobs=-1)
    clf.fit(train_ResponseCode , y_train_onehotencode)
    sig_clf = CalibratedClassifierCV(clf , method='sigmoid')
    sig_clf.fit(train_ResponseCode , y_train_onehotencode)

    y_predicted = sig_clf.predict_proba(cv_ResponseCode)
    cv_log_error.append(log_loss(y_cv_onehotencode , y_predicted , labels = clf.
 ↪classes_ , eps=1e-15))
    print("For n_estimators= ",i," and the max_depth ",j, "logloss is: ",␣
 ↪log_loss(y_cv_onehotencode , y_predicted , labels= clf.classes_ , eps=␣
 ↪1e-15))

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

```
best_alpha = np.argmin(cv_log_error)

clf = RandomForestClassifier(n_estimators = alpha[int(best_alpha/2)]␣
 ↪,max_depth=max_depth[int(j%2)], criterion = 'gini', random_state=42,␣
 ↪n_jobs=-1)
clf.fit(train_ResponseCode , y_train_onehotencode)
sig_clf = CalibratedClassifierCV(clf , method = 'sigmoid')
sig_clf.fit(train_ResponseCode , y_train_onehotencode)
y_predicted = sig_clf.predict_proba(test_ResponseCode)
print("For value of best estimator ",alpha[int(best_alpha/2)],"the test logloss␣
 ↪is: ", log_loss(y_test_onehotencode , y_predicted , labels= clf.classes_ ,␣
 ↪eps=1e-15))
y_predicted = sig_clf.predict_proba(cv_ResponseCode)
print("For value of best estimator ",alpha[int(best_alpha/2)],"the cross␣
 ↪validate logloss is: ", log_loss(y_cv_onehotencode , y_predicted , labels=␣
 ↪clf.classes_ , eps=1e-15))
y_predicted = sig_clf.predict_proba(train_ResponseCode)
print("For value of best estimator ",alpha[int(best_alpha/2)],"the train␣
 ↪logloss is: ", log_loss(y_train_onehotencode , y_predicted , labels= clf.
 ↪classes_ , eps=1e-15))
```

```
For n_estimators=  5  and the max_depth  5 logloss is:  1.664747350090387
For n_estimators=  5  and the max_depth  10 logloss is:  1.831229176069789
For n_estimators=  10  and the max_depth  5 logloss is:  1.5295563002001502
For n_estimators=  10  and the max_depth  10 logloss is:  1.6134640703047571
For n_estimators=  15  and the max_depth  5 logloss is:  1.485906035960822
For n_estimators=  15  and the max_depth  10 logloss is:  1.8602628145419957
For n_estimators=  25  and the max_depth  5 logloss is:  1.5015272260678922
For n_estimators=  25  and the max_depth  10 logloss is:  1.871753233574049
For n_estimators=  500  and the max_depth  5 logloss is:  1.3717526896858698
For n_estimators=  500  and the max_depth  10 logloss is:  1.8563348451787494
For n_estimators=  1000  and the max_depth  5 logloss is:  1.3777292550486167
For n_estimators=  1000  and the max_depth  10 logloss is:  1.8335882151797402
```

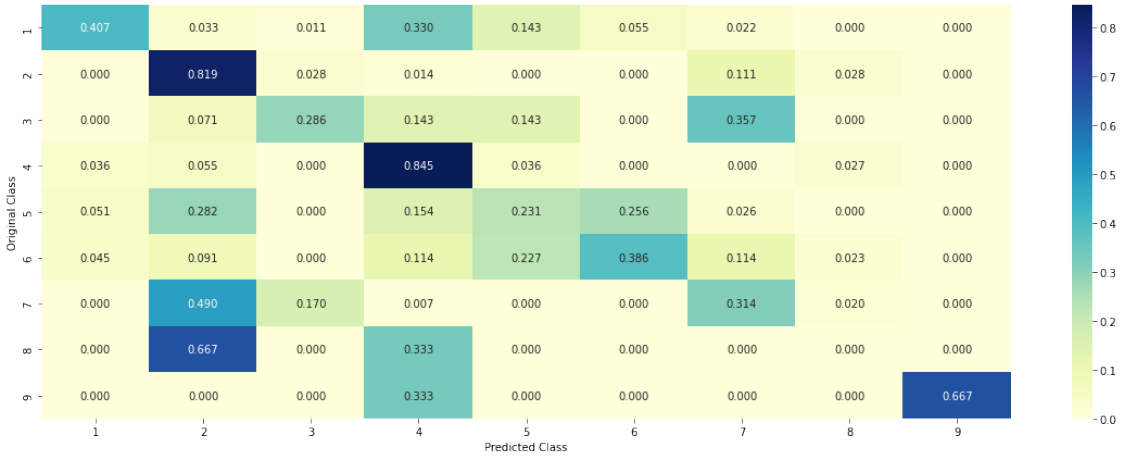



```
For value of best estimator  500 the test logloss is:  1.341510687550331
For value of best estimator  500 the cross validate logloss is:
1.3717526896858698
For value of best estimator  500 the train logloss is:  0.06273645975884622
```

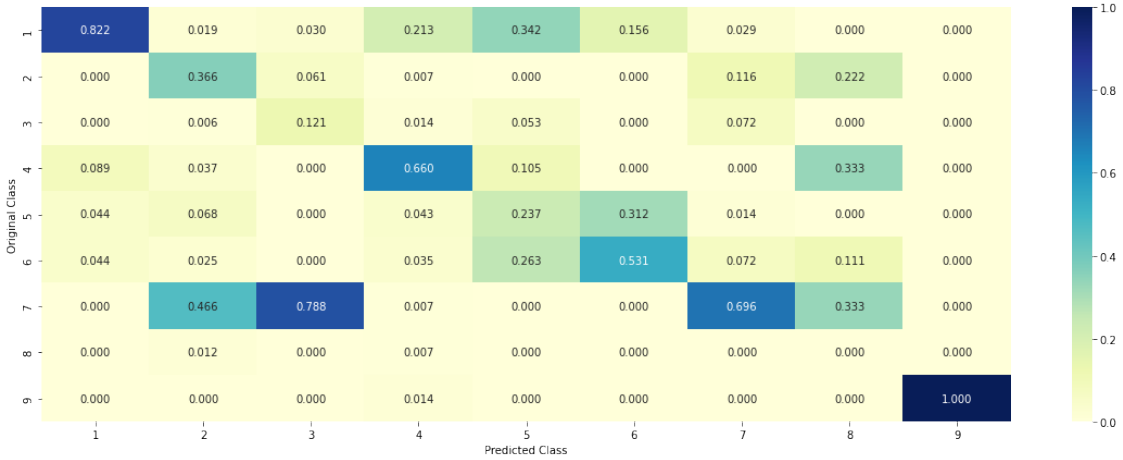
[ ]:
```
clf = RandomForestClassifier(n_estimators = alpha[int(best_alpha/2)]␣
 ↪,max_depth=max_depth[int(j%2)], criterion = 'gini', random_state=42,␣
 ↪n_jobs=-1)
predict_and_plot_confusion_matrix(train_ResponseCode, y_train_onehotencode,␣
 ↪cv_ResponseCode, y_cv_onehotencode, clf)
```

```
Logloss is:  1.3717526896858698
Number of mis-classified points : 0.4906015037593985
------------------------------------------------------------ Confusion matrix
------------------------------------------------------------
```

---------- Precision Martix axis=1 ----------



---------- Recall Martix axis=0 ----------


Original Class
Predicted Class
1 0.822 0.019 0.030 0.213 0.342 0.156 0.029 0.000 0.000
2 0.000 0.366 0.061 0.007 0.000 0.000 0.116 0.222 0.000
3 0.000 0.006 0.121 0.014 0.053 0.000 0.072 0.000 0.000
4 0.089 0.037 0.000 0.660 0.105 0.000 0.000 0.333 0.000
5 0.044 0.068 0.000 0.043 0.237 0.312 0.014 0.000 0.000
6 0.044 0.025 0.000 0.035 0.263 0.531 0.072 0.111 0.000
7 0.000 0.466 0.788 0.007 0.000 0.000 0.696 0.333 0.000
8 0.000 0.012 0.000 0.007 0.000 0.000 0.000 0.000 0.000
9 0.000 0.000 0.000 0.014 0.000 0.000 0.000 0.000 1.000
1 2 3 4 5 6 7 8 9
1.0 0.8 0.6 0.4 0.2 0.0