Multi-class Classification for Cancer Diagnosis

When a doctor suspects a cancer in a patient, probably the first thing he does is to collect the tumor sample from the patient. He then puts the sample through genetic sequencing of DNA. A tumor can have thousands of genetic mutations. Here briefly, a ‘mutation’ is a small change in gene which causes cancer. One more important thing is that for every gene, there is a variation associated with it. The process for cancer diagnosis may be briefed as follows:

1. A molecular pathologist selects a list of genetic variations of interest.
2. He searches for evidence in the medical literature that somehow are relevant to the genetic variations of interest.
3. Finally the molecular pathologist spends a huge amount of time analyzing the evidence related to each of the variations to classify them.

The goal of our project is to replace step 3 by a machine learning model. The molecular pathologist will still have to decide which variations are of interest, and also collect the relevant evidence for them. But the last step, which is also the most time consuming, will be fully automated. So, let us learn how to apply the multi-class classification technique of machine learning to aid the doctor in his analysis of the cancer patient.

# Cancer diagnosis Project

Create a Colab notebook and rename it to CancerDiagnosis. Import the following libraries:

import pandas as pd

import matplotlib.pyplot as plt

import re

import warnings

import numpy as np

import nltk

from sklearn.calibration import CalibratedClassifierCV

from nltk.corpus import stopwords

from sklearn.preprocessing import normalize

from sklearn.feature\_extraction.text import CountVectorizer

import seaborn as sns

from sklearn.metrics import confusion\_matrix

from sklearn.metrics.classification import accuracy\_score, log\_loss

from sklearn.linear\_model import SGDClassifier

from scipy.sparse import hstack

from sklearn.model\_selection import train\_test\_split

import math

from sklearn.linear\_model import LogisticRegression

## Downloading Data

You can download the full dataset from Kaggle [site](https://www.kaggle.com/c/msk-redefining-cancer-treatment). Optionally, you may download the two text files needed for our project from our github. These files are the partial data taken from the Kaggle competition site.

Download the following two zip files and unzip them into your project folder:

!wget <http://abcom.com/article/training_text.zip>

!unzip training\_text.zip

!wget http://abcom.com/article/training\_variants.zip

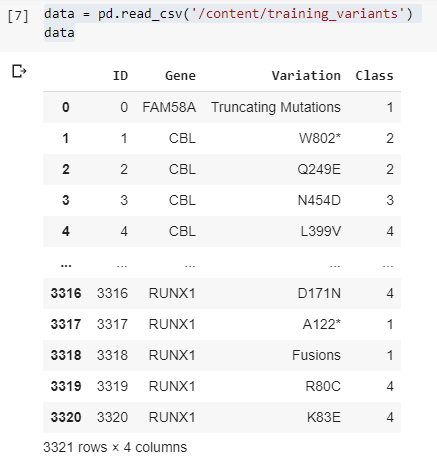
!unzip training\_variants.zip

Examine the contents of training\_variants file.

data = pd.read\_csv('/content/training\_variants')

data

The output is shown below:



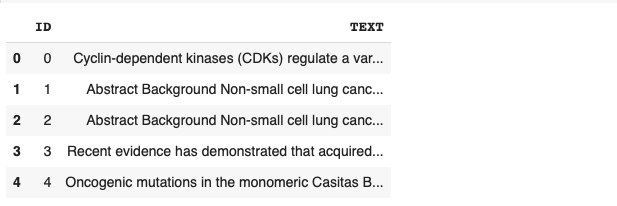
As you can see there are 3321 records, each containing four fields. For each Gene, variations and the corresponding class are listed. For example, we observe that a gene named CBL has four variations - W802, Q249E, N454D and L399V. The corresponding class for these four variations is 2,2,3 and 4. These classes indicate the type of cancer. In the dataset, the classification is provided for 9 classes all of which do not indicate a presence of cancer. Let the doctors decide on this.

Next, load the contents of training\_text file.

data\_text = pd.read\_csv("/content/training\_text", sep = "\|\|", engine = "python", names = ["ID","TEXT"], skiprows = 1)

data\_text.head()

The output is shown below:

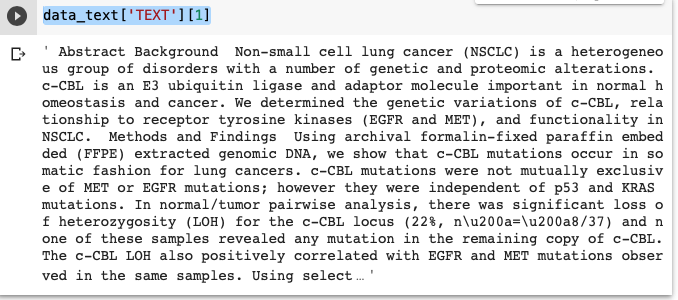


As you can see, for each record (ID) in the training-variants datafile, some text related to cancer research is provided. For doctors, this text would make lots of sense in taking their decisions.

Let us examine one of the records:

data\_text['TEXT'][1]

The output is shown below:



Now, to give you some idea about how the doctors would use the above two files to diagnose a patient, I will take a single record from the above created data\_text and explain to you the whole logic. Let us take the same data text displayed above and split it into a bag of words:

lu = data\_text['TEXT'][1].split()

print(lu)

The output is shown below:



So, we now have a bag of words from the original text item. Let us vectorize them using the following code fragment:

gene\_vectorize = CountVectorizer()

train\_gene\_feature\_onehotCodin = gene\_vectorize.fit\_transform(lu)

lop=gene\_vectorize.vocabulary\_

print(lop)

This is the output:



As you can see, each word in our bag has some unique index associated with it.

Next, we will merge the two text files on the ID field:

result = pd.merge(data, data\_text, on ='ID', how = 'left')

Now, we can search this text for the presence of “gene” and “variation” from our training\_variants datafile.

lst = []

for k in lop.keys():

if(k=="cbl" or k=="w802\*" or k=="q249e" or k=="n454d"):

lst.append(k)

print(lst)

In the above code, we check for the presence of “cbl”, “w802\*”, and so on and print the result. Having found the desired text, we can look up the cancer “Class” from the training\_variants file. This is how the doctors would use the two files to determine the type of cancer.

As you can see, scanning all the text items would be a tremendous task for a human-being. So, we will develop a ML model to provide us a multi-class classification on this dataset. Let us start now with pre-processing the entire dataset.

# Pre-processing Data

To remove the digits and the special characters from the text, we define a function as follows:

import string

regex1 = re.compile('[%s]' % re.escape(string.digits))

regex2 = re.compile('[%s]' % re.escape(string.punctuation))

def remove(sentence):

reg = regex1.sub('',sentence)

reg = regex2.sub('',reg)

return reg

Apply this function on the dataset:

data\_text['TEXT'] = data\_text['TEXT'].apply(lambda x: remove(str(x)))

Replace all null values with gene value and variations.

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

Our data is pre-processed. We will now create training and testing datasets.

# Creating Datasets

We define our target variable as follows:

y\_true = result['Class'].values

Split the dataset into training and testing by maintaining the distribution of output variable y\_true. This is achieved by setting the stratify parameter to y\_true.

X\_train, X\_test, Y\_train, Y\_test = train\_test\_split(result,

y\_true,

stratify = y\_true,

test\_size = 0.2)

Likewise, create a cross validation dataset.

X\_train\_cv, X\_test\_cv, Y\_train\_cv, Y\_test\_cv = train\_test\_split(X\_train, Y\_train,

stratify = Y\_train, test\_size = 0.2)

# Model Choice

From the dataset that we have, you easily understand that it is a high dimensionality data with several keywords in the text acting as features and the 9 class values as output targets. As this is a multi-class classification problem, we will use logistic regression with class balancing as our model choice. For the model training, we will use **Log Loss** (Logarithmic Loss) as a performance measurement function. In Log Loss the prediction input is a probability value between 0 and 1. The goal is to minimize this value, with a log loss value of 0 indicating a perfect model. If you are curious about the mathematical model behind this, the logarithmic loss function Fi is written with the following formula.

Where summation of log loss values is taken across classes. The yo,c is the binary variable with expected values and po,c is the classification probability output for the oth instance and c the cth label. To have a benchmark value for our model’s performance, we will create some random data and generate a log-loss. Then, we will calculate the log-loss on our training and test data. If the log-loss on our test data shows improvement over the random data, then we can safely assume that our model is performing well and can be used on unseen data. So, let us first create some random data and a model based on it.

# Random Model for Benchmarking

We will generate 9 random numbers in the range 1 to 9 representing the 9 output classes that we want for this application. We will sum up these 9 numbers and divide each one by the sum to represent probabilities for each class. Note that the sum of all these probabilities would be 1. We will pick the highest probability value to make the prediction and then plot a confusion matrix and also compute the total log-loss.

First, we define a function for plotting confusion matrix:

## Function for Confusion Matrix

We define the function for plotting a confusion matrix as follows:

def plot\_confusion\_matrix(test\_y, predict\_y):

C = confusion\_matrix(test\_y, predict\_y)

labels = [1,2,3,4,5,6,7,8,9]

# representing A in heatmap format

print("-"\*20, "Confusion matrix", "-"\*20)

plt.figure(figsize = (20,7))

sns.heatmap(C, annot = True, cmap = "YlGnBu", fmt = ".3f",

xticklabels = labels, yticklabels = labels)

plt.xlabel('Predicted Class')

plt.ylabel('Original Class')

plt.show()

We will now create some random target values for the test dataset that we created earlier and use the above function to plot the confusion and print its log-loss value.

## Creating Random Model

The following code generates some random values for the output classes and assigns them to each row of the test dataset. It then computes the log loss between the actual and predicted values and prints its value.

# Test dataset

test\_data\_len = X\_test.shape[0]

# Number of rows equals number of test datapoints

test\_predicted\_y = np.zeros((test\_data\_len,9))

for i in range(test\_data\_len):

# generate 9 random numbers for our random classes.

rand\_probs = np.random.rand(1,9)

# divide each with the sum of all

test\_predicted\_y[i] = ((rand\_probs/sum(sum(rand\_probs)))[0])

print("Log loss on Test Data using Random Model",

log\_loss(Y\_test, test\_predicted\_y, eps=1e-15))

The output is shown below:

Log loss on Test Data using Random Model 2.487717672120489

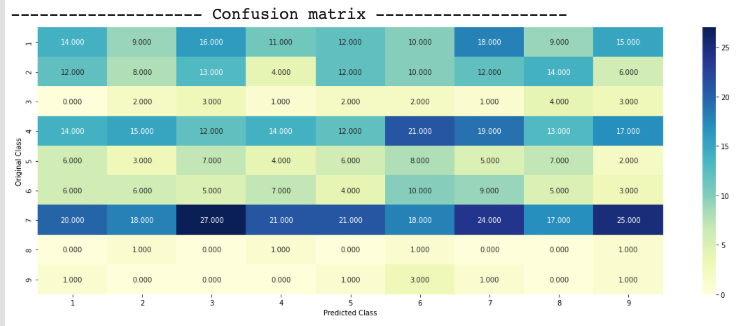
Note that the output will vary on each run due to the randomness of the input data. Finally, we plot a confusion matrix representing all 9 classes.

# get the max probability

predicted\_y = np.argmax(test\_predicted\_y, axis = 1)

plot\_confusion\_matrix(Y\_test, predicted\_y+1)

The output is shown below:



In the above confusion matrix, you see that three are 14000 data points for which the original and the predicted classes match. There are 11000 data points for which both original and predicted classes match, and so on. Thus, using the confusion matrix, you get a better visualization of the model’s performance than looking at a single value of log loss. As you are aware, if all diagonal values are high, the model is considered to be best tuned.

Just the way you printed the log loss on a test data, you could do the same on the cross validation data using the following code:

cv\_data\_len = X\_test\_cv.shape[0]

# Cross validation dataset

cv\_predicted\_y = np.zeros((cv\_data\_len,9))

for i in range(cv\_data\_len):

rand\_probs = np.random.rand(1,9)

cv\_predicted\_y[i] = ((rand\_probs/sum(sum(rand\_probs)))[0])

print("Log loss on Cross Validation Data using Random Model",

log\_loss(Y\_test\_cv, cv\_predicted\_y, eps = 1e-15))

The output is shown below:

Log loss on Cross Validation Data using Random Model 2.5192218893929446

# Encoding Text

For model fitting, we need to convert our textual features into numerical values. We will use Bag-of-Words (BOW) for tokenization and use one hot encoding on the tokens.

<https://machinelearningmastery.com/gentle-introduction-bag-words-model/>

We encode the **gene** using the following code:

gene\_vectorizer = CountVectorizer()

train\_gene\_feature\_onehotCoding = gene\_vectorizer.fit\_transform(X\_train['Gene'])

test\_gene\_feature\_onehotCoding = gene\_vectorizer.transform(X\_test['Gene'])

cv\_gene\_feature\_onehotCoding = gene\_vectorizer.transform(X\_train\_cv['Gene'])

Note that the above code encodes gene features in all three datasets. Likewise, we encode variation features using the following code:

# one-hot encoding of variation feature.

variation\_vectorizer = CountVectorizer()

train\_variation\_feature\_onehotCoding = variation\_vectorizer.fit\_transform(X\_train['Variation'])

test\_variation\_feature\_onehotCoding = variation\_vectorizer.transform(X\_test['Variation'])

cv\_variation\_feature\_onehotCoding = variation\_vectorizer.transform(X\_train\_cv['Variation'])

Lastly, we encode the text feature using the following code. In the CountVectorizer constructor we specify the minimum frequency of occurrence to be 3 and discard all English stop words. We do this for text features only as the text feature has lots of words while the gene and variation has only one word in it. If we do not restrict the tokenization to words having a minimum frequency of 3, then the number of features for training would turn out to be very large and unmanageable in terms of required resources for training. We also normalize each word as in case of the text feature a particular word may have a high frequency. In case of gene and variation, most of the values were 0 with a max value of 1, so normalization was not required in those cases.

# one-hot encoding of text feature.

# Minimum frequency for words = 3 and remove all stop words

text\_vectorizer = CountVectorizer(min\_df = 3, stop\_words = 'english')

train\_text\_feature\_onehotCoding = text\_vectorizer.fit\_transform(X\_train['TEXT'])

train\_text\_feature\_onehotCoding = normalize(train\_text\_feature\_onehotCoding, axis = 0)

test\_text\_feature\_onehotCoding = text\_vectorizer.transform(X\_test['TEXT'])

test\_text\_feature\_onehotCoding = normalize(test\_text\_feature\_onehotCoding, axis = 0)

cv\_text\_feature\_onehotCoding = text\_vectorizer.transform(X\_train\_cv['TEXT'])

cv\_text\_feature\_onehotCoding = normalize(cv\_text\_feature\_onehotCoding, axis = 0)

## Merging Features

For our analysis, all features - gene, variation and text are equally important. So we merge them in a single list before feeding it to our logistic regression algorithm. We use hstack function to stack the various data points and finally convert them into numpy arrays for inputting them to our model.

# merging gene, variation and text features

train\_gene\_var\_onehotCoding = hstack((train\_gene\_feature\_onehotCoding,

train\_variation\_feature\_onehotCoding))

test\_gene\_var\_onehotCoding = hstack((test\_gene\_feature\_onehotCoding,

test\_variation\_feature\_onehotCoding))

cv\_gene\_var\_onehotCoding = hstack((cv\_gene\_feature\_onehotCoding,

cv\_variation\_feature\_onehotCoding))

train\_x\_onehotCoding = hstack((train\_gene\_var\_onehotCoding,

train\_text\_feature\_onehotCoding)).tocsr()

train\_y = np.array(list(X\_train['Class']))

test\_x\_onehotCoding = hstack((test\_gene\_var\_onehotCoding,

test\_text\_feature\_onehotCoding)).tocsr()

test\_y = np.array(list(X\_test['Class']))

cv\_x\_onehotCoding = hstack((cv\_gene\_var\_onehotCoding,

cv\_text\_feature\_onehotCoding)).tocsr()

cv\_y = np.array(list(X\_train\_cv['Class']))

Now, we are ready to build our model.

# Logistic Regression with Class Balancing

As mentioned before, we will use logistic regression for modeling our dataset. The logistic regression performs well for high dimension data and it is also interpretable. First, we will identify the best alpha value for training the model.

## Selecting Best Alpha for Model Training

The model training code is shown here. After the code, I have explained the relevant statements.

alpha = [10 \*\* x for x in range(-6, 3)]

cv\_log\_error\_array = []

for i in alpha:

print("for alpha =", i)

clf = SGDClassifier(class\_weight='balanced',

alpha = i, penalty = 'l2',

loss = 'log', random\_state = 42)

clf.fit(train\_x\_onehotCoding, train\_y)

sig\_clf = CalibratedClassifierCV(clf, method = "sigmoid")

sig\_clf.fit(train\_x\_onehotCoding, train\_y)

sig\_clf\_probs = sig\_clf.predict\_proba(cv\_x\_onehotCoding)

cv\_log\_error\_array.append(log\_loss(cv\_y, sig\_clf\_probs,

labels = clf.classes\_, eps = 1e-15))

print("Log Loss :",log\_loss(cv\_y, sig\_clf\_probs))

We use SGDClassifier function for logistic regression. The class\_weight parameter is set to balanced so that it can take care of an imbalance dataset. In our dataset, some classes occur more frequently than others. We set the l2 regularization to avoid overfitting. The loss parameter is set to log indicating that we want to apply logistic regression. Also, we use the calibrated classifier that predicts the class probabilities rather than the absolute values. Predicting probabilities provides more nuanced ways to evaluate the model’s skills. Running the code gives following output:

for alpha = 1e-06

Log Loss : 0.800011752988093

for alpha = 1e-05

Log Loss : 0.7109059689512729

for alpha = 0.0001

Log Loss : 0.4364909854786977

for alpha = 0.001

Log Loss : 0.48798880250248494

for alpha = 0.01

Log Loss : 0.7112861981789363

for alpha = 0.1

Log Loss : 1.3174625469050127

for alpha = 1

Log Loss : 1.5946220303762801

for alpha = 10

Log Loss : 1.6277595686104878

for alpha = 100

Log Loss : 1.6312474225418718

Now, we need to select the best alpha value and redo the training with it.

## Train with Best Alpha

We select the best alpha value by using argmin function and then redo the training using this alpha value.

best\_alpha = np.argmin(cv\_log\_error\_array)

clf = SGDClassifier(class\_weight = 'balanced',

alpha = alpha[best\_alpha], penalty = 'l2',

loss='log', random\_state = 42)

clf.fit(train\_x\_onehotCoding, train\_y)

sig\_clf = CalibratedClassifierCV(clf, method = "sigmoid")

sig\_clf.fit(train\_x\_onehotCoding, train\_y)

## Inference

After the training is over, we do inference on the three datasets.

predict\_y = sig\_clf.predict\_proba(train\_x\_onehotCoding)

print("The train log loss is:",

log\_loss(Y\_train, predict\_y, labels = clf.classes\_, eps = 1e-15))

predict\_y = sig\_clf.predict\_proba(cv\_x\_onehotCoding)

print("The cross validation log loss is:",

log\_loss(Y\_train\_cv, predict\_y, labels = clf.classes\_, eps = 1e-15))

predict\_y = sig\_clf.predict\_proba(test\_x\_onehotCoding)

print("The test log loss is:",

log\_loss(Y\_test, predict\_y, labels = clf.classes\_, eps = 1e-15))

You will see the following output:

The train log loss is: 0.4640617141872244

The cross validation log loss is: 0.4364909854786977

The test log loss is: 1.0651077770959576

Compare these values with our earlier results of the random model, which are reproduced below for your quick reference.

Log loss on Cross Validation Data using Random Model 2.484133730408836

Log loss on Test Data using Random Model 2.522241936357013

As you can see, in both cases of validation and test datasets, the log-loss is reduced considerably from about 2.5 to less than 1.0. Thus, we can conclude that our model is performing well.

Now, we will do some model testing with the best hyper parameters that we have obtained above.

# Model Evaluation

For model testing and plotting the confusion matrix on the results we define the following function:

def predict\_and\_plot\_confusion\_matrix(train\_x, train\_y,

test\_x, test\_y, clf):

clf.fit(train\_x, train\_y)

sig\_clf = CalibratedClassifierCV(clf, method = "sigmoid")

sig\_clf.fit(train\_x, train\_y)

pred\_y = sig\_clf.predict(test\_x)

# Display number of data points that are misclassified

print("Number of mis-classified points :",

np.count\_nonzero((pred\_y- test\_y))/test\_y.shape[0])

plot\_confusion\_matrix(test\_y, pred\_y)

We pass the classifier instance in the above method, train the model with the training dataset and do a prediction on the test dataset. We print the number of mis-classified points and the confusion matrix on the terminal.

The above method itself is called by first creating the classifier instance with the best alpha parameter.

clf = SGDClassifier(class\_weight = 'balanced',

alpha = alpha[best\_alpha], penalty = 'l2',

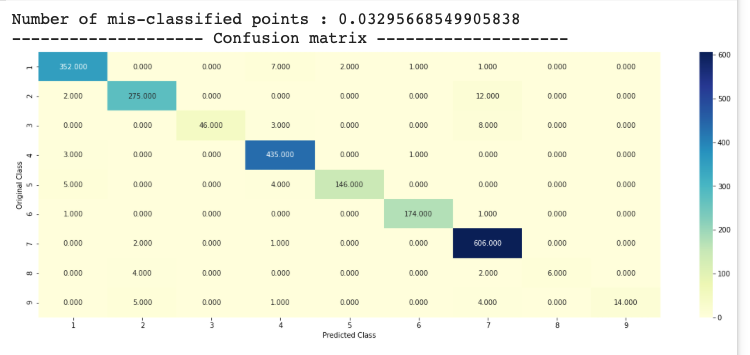
loss = 'log', random\_state = 42)

predict\_and\_plot\_confusion\_matrix(train\_x\_onehotCoding,

train\_y, cv\_x\_onehotCoding,

cv\_y, clf)

The output is shown below:



As you can see, the number of mis-classified points is just 3.2%. The confusion matrix too shows most of the diagonal cells have high values. Thus, our model is doing very well.

Now, we are going to put our trained model for some real use.

# Real Use by Doctors

Consider that a doctor sees a patient with a tumor. He samples the tumor and gets the gene and variation details on the sample. Then he uses our model to find out the cancer class (a number in the range 1 to 9) and also the number of places in the text where the important keywords (features) are found. Based on this information, he does the further diagnosis on the patient. To demonstrate this use, I will pick up a record from our test dataset that gives us the gene and its variations. To determine the detected features in the dataset, we first write a small function called getImportantFeatures as follows:

def getImportantFeatures(indices, gene, variation, text, noOfFeatures):

gene\_features = gene\_vectorizer.get\_feature\_names()

variation\_features = variation\_vectorizer.get\_feature\_names()

text\_features = text\_vectorizer.get\_feature\_names()

gene\_feat\_len = len(gene\_features)

var\_feat\_len = len(variation\_features)

text\_features\_len =len(text\_features)

word\_present = 0

for i, v in enumerate(indices):

if v < gene\_feat\_len:

word = gene\_features[v]

if word == gene:

word\_present += 1

print("{}st Gene feature [{}] is present in query point [{}]".format(i+1, word,bool\_var))

elif (v < gene\_feat\_len + var\_feat\_len):

word = variation\_features[v - gene\_feat\_len]

if word == variation:

word\_present += 1

print("{}th Variation feature [{}] is present in query point".format(i+1, word))

else:

word = text\_features[v - (gene\_feat\_len + var\_feat\_len)]

if word in text.split():

word\_present += 1

print("{}th Text feature [{}] is present in query point".format(i+1, word))

print("-"\*63)

print("Out of the top "+str(noOfFeatures)+" features "+

str(word\_present)+" are present in query point")

print("-"\*63)

The function simply looks for the gene, variation and top text features and prints the index values of the text items where those are found. Let us now take a sample data point, say at index 500, and print the analysis results given by our model. This is done using the following code:

testDataPoint = 500

top\_features = 1000

predicted\_cls = sig\_clf.predict(test\_x\_onehotCoding[testDataPoint])

print("Predicted Class :", predicted\_cls[0])

print("Predicted Class Probabilities:", np.round(sig\_clf.predict\_proba(test\_x\_onehotCoding[testDataPoint]),4))

print("Actual Class :", test\_y[testDataPoint])

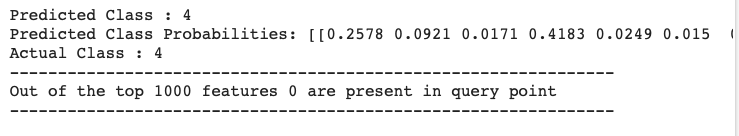
indices = np.argsort(-1\*abs(clf.coef\_))[predicted\_cls-1][:,:top\_features]

getImportantFeatures(indices[0], X\_test.iloc[testDataPoint]["Gene"],

X\_test.iloc[testDataPoint]["Variation"],

X\_test.iloc[testDataPoint]["TEXT"], top\_features)

The output is shown below:



From the above output, you can see that the predicted class is 4 with the highest probability of 0.4711 amongst all classes. This indicates to the doctor that the probability of getting cancer of class 4 is about 47%. The actual class as known from our dataset too is 4. Note that in the real world example, this actual class is not obviously available to the doctor. The output also shows that the two features “labeling” and “activating” are present in the text at index 781 and 995 respectively. The doctor can now study the text at these indices to further firm up his decision. You can easily make out that the doctor is now saving his tremendous valuable time in going through all the texts in the dataset and focussing only on those text items pointed out in the prediction output. You may test the model with another index value (the test datapoint) and see the results for yourself.

Finally, to make the model really useful to the doctors in diagnosing unseen cases, you will need to provide an application level user interface to the doctor where he enters the three fields - gene, variations and some text of his own. This would create a record similar to the test record at index 500. The new record has to undergo the entire preprocessing that we did on our test dataset. Feed the pre-processed record to the predict method and wow it will spill out the predictions indicating the probable type of cancer class and some text to read to further up the claim.

# Conclusion

In this tutorial, you learned how to apply a multi-class classification using linear regression to help the doctors diagnose a probable cancer patient. We used the SGDClassifier of sklearn and trained it for classifying 9 cancer types. You used the data supplied in the Kaggle competition for training the model. The use of confusion matrix helped us in visualizing the model’s performance for a multi-class problem. You may now use the technique that you learned here in solving other multi-class classification problems with a large number of target values.

# 

# References

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

<https://stackoverflow.com/a/18662466/4084039>

<http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html>

<http://scikit-learn.org/stable/modules/generated/sklearn.calibration.CalibratedClassifierCV.html>

<https://medium.com/@tulasiram11729/personalized-cancer-diagnosis-3d6f09a6b8c9>