***Note:****We will be using****accuracy\_score****function to evaluate all our models in this project.*

**Part-3: Exploratory Data Analysis (EDA):**

*Exploratory Data Analysis is one of the important steps in the data analysis process. Here, the focus is on making sense of the data in hand — things like formulating the correct questions to ask to your dataset, how to manipulate the data sources to get the required answers, and others.*

* First let us import the necessary libraries.

import os  
import csv  
import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
import seaborn as sns

* Next we load the data from csv files into a pandas dataframe and check its attributes.

data\_path = "/Users/kartik/Desktop/AAIC/Projects/jigsaw-toxic-comment-classification-challenge/data/train.csv"data\_raw = pd.read\_csv(data\_path)print("Number of rows in data =",data\_raw.shape[0])  
print("Number of columns in data =",data\_raw.shape[1])  
print("\n")  
print("\*\*Sample data:\*\*")  
data\_raw.head()





Fig-8: Data Attributes

* Now we count the number of comments under each label. (For detailed code, please refer to the GitHub link of this project.)

categories = list(data\_raw.columns.values)  
sns.set(font\_scale = 2)  
plt.figure(figsize=(15,8))ax= sns.barplot(categories, data\_raw.iloc[:,2:].sum().values)plt.title("Comments in each category", fontsize=24)  
plt.ylabel('Number of comments', fontsize=18)  
plt.xlabel('Comment Type ', fontsize=18)#adding the text labels  
rects = ax.patches  
labels = data\_raw.iloc[:,2:].sum().values  
for rect, label in zip(rects, labels):  
 height = rect.get\_height()  
 ax.text(rect.get\_x() + rect.get\_width()/2, height + 5, label, ha='center', va='bottom', fontsize=18)plt.show()



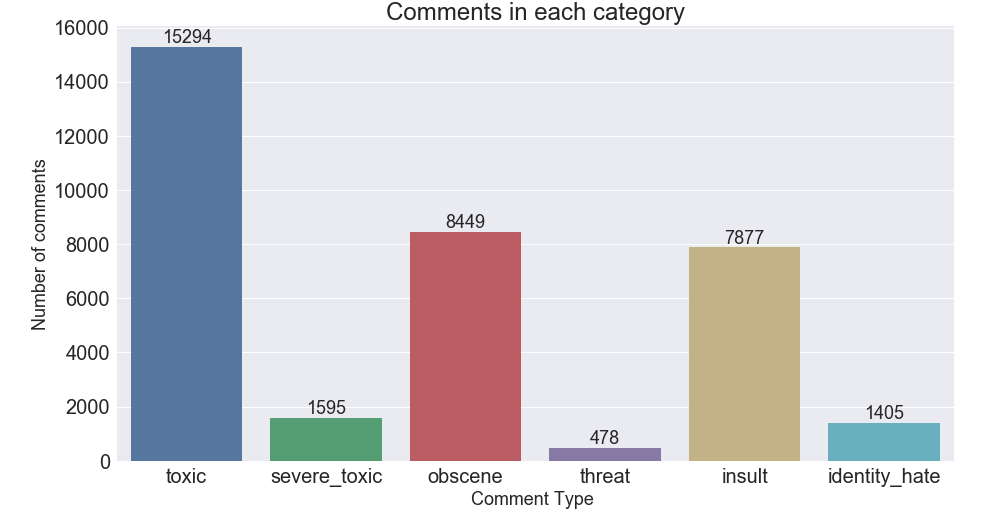


Fig-9: Count of comments under each label

* Counting the number of comments having multiple labels.

rowSums = data\_raw.iloc[:,2:].sum(axis=1)  
multiLabel\_counts = rowSums.value\_counts()  
multiLabel\_counts = multiLabel\_counts.iloc[1:]sns.set(font\_scale = 2)  
plt.figure(figsize=(15,8))ax = sns.barplot(multiLabel\_counts.index, multiLabel\_counts.values)plt.title("Comments having multiple labels ")  
plt.ylabel('Number of comments', fontsize=18)  
plt.xlabel('Number of labels', fontsize=18)#adding the text labels  
rects = ax.patches  
labels = multiLabel\_counts.values  
for rect, label in zip(rects, labels):  
 height = rect.get\_height()  
 ax.text(rect.get\_x() + rect.get\_width()/2, height + 5, label, ha='center', va='bottom')plt.show()



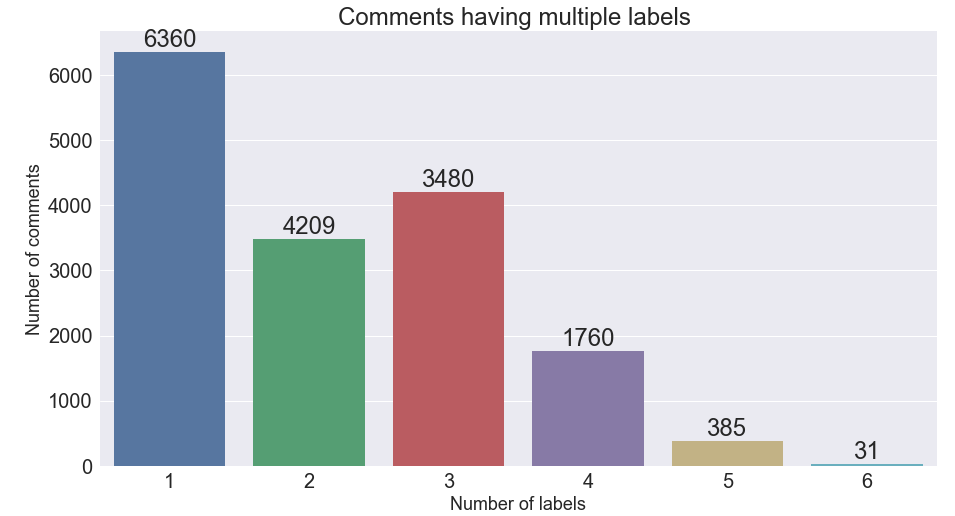


Fig-10: Count of comments with multiple labels.

* WordCloud representation of most used words in each category of comments.

from wordcloud import WordCloud,STOPWORDSplt.figure(figsize=(40,25))# clean  
subset = data\_raw[data\_raw.clean==True]  
text = subset.comment\_text.values  
cloud\_toxic = WordCloud(  
 stopwords=STOPWORDS,  
 background\_color='black',  
 collocations=False,  
 width=2500,  
 height=1800  
 ).generate(" ".join(text))  
plt.axis('off')  
plt.title("Clean",fontsize=40)  
plt.imshow(cloud\_clean)# Same code can be used to generate wordclouds of other categories.



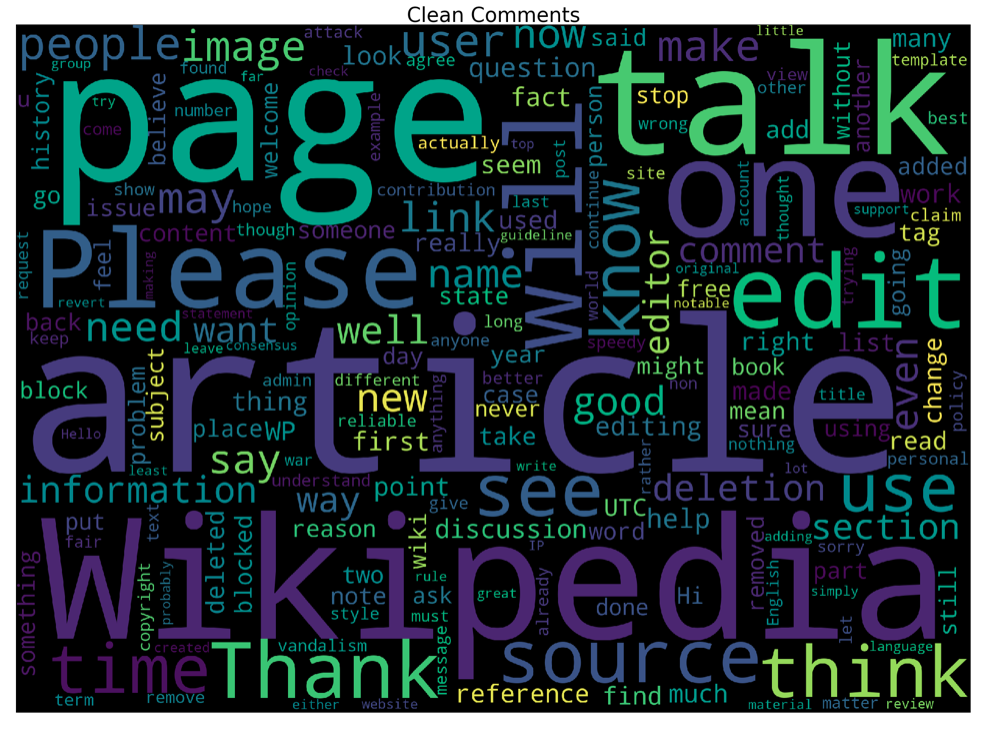


Fig-1: Word-cloud Representation of Clean Comments

**Part-4: Data Pre-Processing:**

* We first convert the comments to lower-case and then use custom made functions to remove *html-tags, punctuation and non-alphabetic characters* from the comments.

import nltk  
from nltk.corpus import stopwords  
from nltk.stem.snowball import SnowballStemmer  
import re  
import sys  
import warningsdata = data\_rawif not sys.warnoptions:  
 warnings.simplefilter("ignore")def cleanHtml(sentence):  
 cleanr = re.compile('<.\*?>')  
 cleantext = re.sub(cleanr, ' ', str(sentence))  
 return cleantextdef cleanPunc(sentence): #function to clean the word of any punctuation or special characters  
 cleaned = re.sub(r'[?|!|\'|"|#]',r'',sentence)  
 cleaned = re.sub(r'[.|,|)|(|\|/]',r' ',cleaned)  
 cleaned = cleaned.strip()  
 cleaned = cleaned.replace("\n"," ")  
 return cleaneddef keepAlpha(sentence):  
 alpha\_sent = ""  
 for word in sentence.split():  
 alpha\_word = re.sub('[^a-z A-Z]+', ' ', word)  
 alpha\_sent += alpha\_word  
 alpha\_sent += " "  
 alpha\_sent = alpha\_sent.strip()  
 return alpha\_sentdata['comment\_text'] = data['comment\_text'].str.lower()  
data['comment\_text'] = data['comment\_text'].apply(cleanHtml)  
data['comment\_text'] = data['comment\_text'].apply(cleanPunc)  
data['comment\_text'] = data['comment\_text'].apply(keepAlpha)

* Next we remove all the ***stop-words*** present in the comments using the default set of stop-words that can be downloaded from *NLTK* library. We also add few stop-words to the standard list.
* Stop words are basically a set of commonly used words in any language, not just English. The reason why stop words are critical to many applications is that, if we remove the words that are very commonly used in a given language, we can focus on the important words instead.

stop\_words = set(stopwords.words('english'))  
stop\_words.update(['zero','one','two','three','four','five','six','seven','eight','nine','ten','may','also','across','among','beside','however','yet','within'])  
re\_stop\_words = re.compile(r"\b(" + "|".join(stop\_words) + ")\\W", re.I)  
def removeStopWords(sentence):  
 global re\_stop\_words  
 return re\_stop\_words.sub(" ", sentence)data['comment\_text'] = data['comment\_text'].apply(removeStopWords)

* Next we do ***stemming****.*There exist different kinds of stemming which basically transform words with roughly the same semantics to one standard form. For example, for amusing, amusement, and amused, the stem would be amus.

stemmer = SnowballStemmer("english")  
def stemming(sentence):  
 stemSentence = ""  
 for word in sentence.split():  
 stem = stemmer.stem(word)  
 stemSentence += stem  
 stemSentence += " "  
 stemSentence = stemSentence.strip()  
 return stemSentencedata['comment\_text'] = data['comment\_text'].apply(stemming)

* After splitting the dataset into train & test sets, we want to summarize our comments and convert them into numerical vectors.
* One technique is to pick the most frequently occurring terms (words with high ***term frequency*** or *tf*). However, the most frequent word is a less useful metric since some words like ‘*this*’, ‘*a*’ occur very frequently across all documents.
* Hence, we also want a measure of how unique a word is i.e. how infrequently the word occurs across all documents (i***nverse document frequency*** or *idf*).
* So, the product of tf & idf (***TF-IDF***) of a word gives a product of how frequent this word is in the document multiplied by how unique the word is w.r.t. the entire corpus of documents.
* Words in the document with a high tfidf score occur frequently in the document and provide the most information about that specific document.

from sklearn.model\_selection import train\_test\_splittrain, test = train\_test\_split(data, random\_state=42, test\_size=0.30, shuffle=True)from sklearn.feature\_extraction.text import TfidfVectorizer  
vectorizer = TfidfVectorizer(strip\_accents='unicode', analyzer='word', ngram\_range=(1,3), norm='l2')  
vectorizer.fit(train\_text)  
vectorizer.fit(test\_text)x\_train = vectorizer.transform(train\_text)  
y\_train = train.drop(labels = ['id','comment\_text'], axis=1)x\_test = vectorizer.transform(test\_text)  
y\_test = test.drop(labels = ['id','comment\_text'], axis=1)

* ***TF-IDF***is easy to compute but its disadvantage is that it does not capture position in text, semantics, co-occurrences in different documents, etc.

**Part-5: Multi-Label Classification Techniques:**

*Most traditional learning algorithms are developed for single-label classification problems. Therefore a lot of approaches in the literature transform the multi-label problem into multiple single-label problems, so that the existing single-label algorithms can be used.*

**1. OneVsRest**

* Traditional two-class and multi-class problems can both be cast into multi-label ones by restricting each instance to have only one label. On the other hand, the generality of multi-label problems inevitably makes it more difficult to learn. An intuitive approach to solving multi-label problem is to decompose it into multiple independent binary classification problems (one per category).
* In an “one-to-rest” strategy, one could build multiple independent classifiers and, for an unseen instance, choose the class for which the confidence is maximized.
* The main assumption here is that the labels are *mutually exclusive*. You do not consider any underlying correlation between the classes in this method.
* For instance, it is more like asking simple questions, say, “*is the comment toxic or not*”, “*is the comment threatening or not?*”, etc. Also there might be an extensive case of overfitting here, since most of the comments are unlabeled, i,e., most of the comments are clean comments.

from sklearn.linear\_model import LogisticRegression  
from sklearn.pipeline import Pipeline  
from sklearn.metrics import accuracy\_score  
from sklearn.multiclass import OneVsRestClassifier# Using pipeline for applying logistic regression and one vs rest classifier  
LogReg\_pipeline = Pipeline([  
 ('clf', OneVsRestClassifier(LogisticRegression(solver='sag'), n\_jobs=-1)),  
 ])for category in categories:  
 print('\*\*Processing {} comments...\*\*'.format(category))  
   
 # Training logistic regression model on train data  
 LogReg\_pipeline.fit(x\_train, train[category])  
   
 # calculating test accuracy  
 prediction = LogReg\_pipeline.predict(x\_test)  
 print('Test accuracy is {}'.format(accuracy\_score(test[category], prediction)))  
 print("\n")



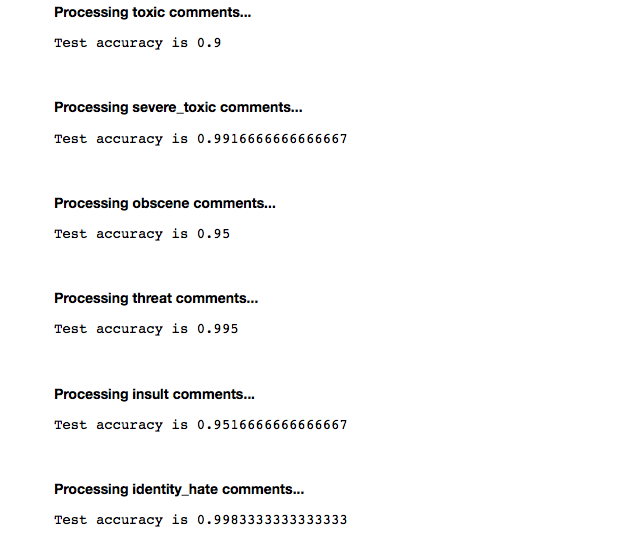


Fig-12: OneVsRest

**2. Binary Relevance**

* In this case an ensemble of single-label binary classifiers is trained, one for each class. Each classifier predicts either the membership or the non-membership of one class. The union of all classes that were predicted is taken as the multi-label output. This approach is popular because it is easy to implement, however it also ignores the possible correlations between class labels.
* In other words, if there’s *q* labels, the binary relevance method create *q* new data sets from the images, one for each label and train single-label classifiers on each new data set. One classifier may answer yes/no to the question “does it contain trees?”, thus the “binary” in “binary relevance”. This is a simple approach but does not work well when there’s dependencies between the labels.
* *OneVsRest & Binary Relevance*seem very much alike. If multiple classifiers in OneVsRest answer *“yes”* then you are back to the binary relevance scenario.

# using binary relevance  
from skmultilearn.problem\_transform import BinaryRelevance  
from sklearn.naive\_bayes import GaussianNB# initialize binary relevance multi-label classifier  
# with a gaussian naive bayes base classifier  
classifier = BinaryRelevance(GaussianNB())# train  
classifier.fit(x\_train, y\_train)# predict  
predictions = classifier.predict(x\_test)# accuracy  
print("Accuracy = ",accuracy\_score(y\_test,predictions))  
***Output:****Accuracy = 0.856666666667*

**3. Classifier Chains**

* A chain of binary classifiers C0, C1, . . . , Cn is constructed, where a classifier Ci uses the predictions of all the classifier Cj , where j < i. This way the method, also called classifier chains (CC), can take into account label correlations.
* The total number of classifiers needed for this approach is equal to the number of classes, but the training of the classifiers is more involved.
* Following is an illustrated example with a classification problem of three categories {C1, C2, C3} chained in that order.



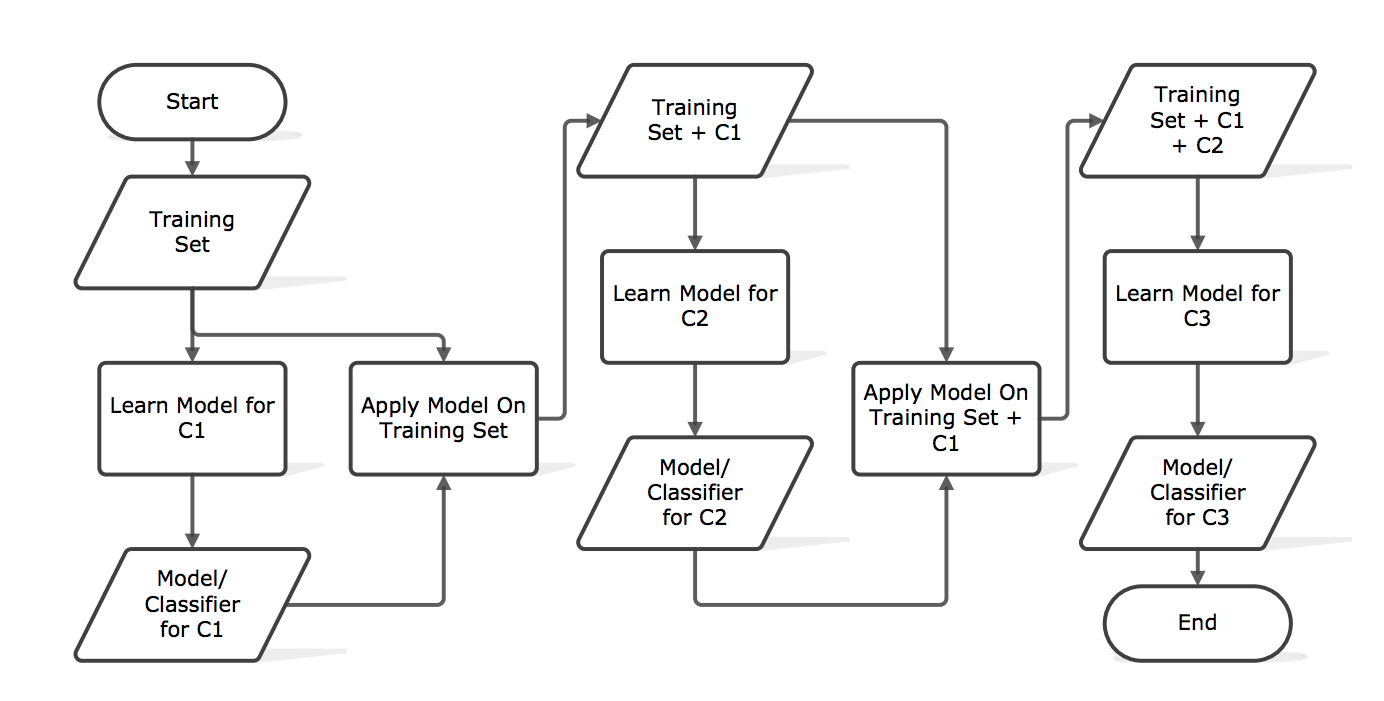


Fig-13: Classifier Chains

# using classifier chains  
from skmultilearn.problem\_transform import ClassifierChain  
from sklearn.linear\_model import LogisticRegression# initialize classifier chains multi-label classifier  
classifier = ClassifierChain(LogisticRegression())# Training logistic regression model on train data  
classifier.fit(x\_train, y\_train)# predict  
predictions = classifier.predict(x\_test)# accuracy  
print("Accuracy = ",accuracy\_score(y\_test,predictions))  
print("\n")***Output:****Accuracy = 0.893333333333*

**4. Label Powerset**

* This approach does take possible correlations between class labels into account. More commonly this approach is called the label-powerset method, because it considers each member of the power set of labels in the training set as a single label.
* This method needs worst case (2^|C|) classifiers, and has a high computational complexity.
* However when the number of classes increases the number of distinct label combinations can grow exponentially. This easily leads to combinatorial explosion and thus computational infeasibility. Furthermore, some label combinations will have very few positive examples.

# using Label Powerset  
from skmultilearn.problem\_transform import LabelPowerset# initialize label powerset multi-label classifier  
classifier = LabelPowerset(LogisticRegression())# train  
classifier.fit(x\_train, y\_train)# predict  
predictions = classifier.predict(x\_test)# accuracy  
print("Accuracy = ",accuracy\_score(y\_test,predictions))  
print("\n")***Output:****Accuracy = 0.893333333333*

**5. Adapted Algorithm**

* Algorithm adaptation methods for multi-label classification concentrate on adapting single-label classification algorithms to the multi-label case usually by changes in cost/decision functions.
* Here we use a multi-label lazy learning approach named ***ML-KNN*** which is derived from the traditional K-nearest neighbor (KNN) algorithm.
* The **[skmultilearn.adapt](http://scikit.ml/api/api/skmultilearn.adapt.html" \l "module-skmultilearn.adapt" \t "_blank)** module implements algorithm adaptation approaches to multi-label classification, including but not limited to ***ML-KNN.***

from skmultilearn.adapt import MLkNN  
from scipy.sparse import csr\_matrix, lil\_matrixclassifier\_new = MLkNN(k=10)# Note that this classifier can throw up errors when handling sparse matrices.x\_train = lil\_matrix(x\_train).toarray()  
y\_train = lil\_matrix(y\_train).toarray()  
x\_test = lil\_matrix(x\_test).toarray()# train  
classifier\_new.fit(x\_train, y\_train)# predict  
predictions\_new = classifier\_new.predict(x\_test)# accuracy  
print("Accuracy = ",accuracy\_score(y\_test,predictions\_new))  
print("\n")***Output:****Accuracy = 0.88166666667*

**Conclusion:**

**Results:**

* There are two main methods for tackling a multi-label classification problem: **problem transformation methods**and**algorithm adaptation methods**.
* Problem transformation methods transform the multi-label problem into a set of [binary classification](https://en.wikipedia.org/wiki/Binary_classification) problems, which can then be handled using single-class classifiers.
* Whereas algorithm adaptation methods adapt the algorithms to directly perform multi-label classification. In other words, rather than trying to convert the problem to a simpler problem, they try to address the problem in its full form.
* In an extensive comparison with other approaches, label-powerset method scores best, followed by the one-against-all method.
* Both ML-KNN and label-powerset take considerable amount of time when run on this dataset, so experimentation was done on a random sample of the train data.

**Further improvements:**

* The same problem can be solved using LSTMs in deep learning.
* For more speed we could use decision trees and for a reasonable trade-off between speed and accuracy we could also opt for ensemble models.
* Other frameworks such as MEKA can be used to deal with multi-label classification problems.