**Tweet Classification & Prediction based on PPI words**

**Task :**

Classification and Prediction of twitter data based on Personal Identifiable information (PPI) using SVM algorithm and categorized them with high risk, low risk and medium risk based on the number of feature (PII) words in the tweets.

**Description :**

Personal information, described in United States legal fields as either personally identifiable information (PII), or sensitive personal information (SPI), as used in information security and privacy laws, is information that can be used on its own or with other information to identify, contact, or locate a single person, or to identify an individual in context.

In today’s world lot’s of people in social media post lot’s of personal information about them which can be used to exploit them. So we need to identify the tweet in which

PPI words are present.

**Features Taken** :

|  |
| --- |
| * Personal Information   + Middle name, Father's /Mother's first name, Mother's maiden name,Child's name, Son's name, Daughter's name, Pet's name, Birthday, Age, Email, Address, House Address, Home, Gender, current location, Best holiday, Social security number, Car license number/ Registration no, Zip Code, phone number, Blood type, Place of birth, passport, Invoice, drivers license      * Financial Information   + Bank, Account number, Debit/ Credit card number, Salary, Earning, Paypal |
| * Marital status   + Married, Divorced,Single |
| * Medical information   + biometric records, Genotype, Blood type, |
| * Religious preference |
| * Legal status |
| * Sexuality Information   + Gay, lesbian, Transgender, homophobic |

**Data Preprocessing :**

To process the tweet data, the data needs to be pre-processed first for the cleaning.

* HTML decoding
* @mention
* URL link
* Non-Ascii character (UTF-8 BOM)
* special characters

**HTML decoding :**

Sometimes HTML encoding has not been converted to text, and ended up in text field as '&amp','&quot',etc.

Example : "Whinging. My client&amp;boss don't understand English well. Rewrote some text unreadable. It's written by v. good writer&amp;reviewed correctly. "

Decoding HTML to general text will be my first step of data preparation. I have used **BeautifulSoup** library for this.

from bs4 import BeautifulSoup

soup = BeautifulSoup(tweet, 'lxml')

print (example1.get\_text())

In first line we imported the library. In 2nd line we parse the text in ‘lxml’ format (use for parsing both xml,HTML). In 3rd line we extract the text.

**@mention:**

The second part of the preparation is dealing with @mention.

Even though @mention carries a certain information (which another user that the tweet mentioned), this information doesn't add value to any PPI words.

Ex- @TheLeagueSF Not Fun !

For this we use regex to extract @mention

**import** **re**  
 re.sub(r'@[A-Za-z0-9]+','',tweet)

In 2nd line @mention is replaced. For this we use this expression ‘@[A-Za-z0-9]+’ .It will select all the words start with @ and replace by with None.

**Remove HTTP links:**

The third part is to remove http website links. As lot’s of people use http links. So this information is irrelevant to us. So we will remove it.

re.sub('https?://[A-Za-z0-9./]+', '' ,tweet)

It will extract all links with http tag and replace with None.

**Remove non-ASCII character :**

Ex- 'Tuesdayï¿½ll start with reflection ï¿½n then a lecture in Stress reducing techniques. That sure might become very useful for us accompaniers '

In above example some non-ascii characters are present. It is present due to emoji, small icon text. So we can subtract this by only taking known ascii character. We can implement this by following code :

''.join([i if ord(i) < 128 else ' ' for i in tweet])

In this code we are allowing only character which has ascii number less than 128.

**Remove #tag :**

Similar to @mention we don’t need #tag. So we will remove those.

re.sub(r'(\s)#\w+', r'\1', tweet)

Similar to @mention.

Regex Reference :

<https://regex101.com/>

<http://www.ntu.edu.sg/home/ehchua/programming/howto/regexe.html>

Used for testing regex pattern.

* Then we collectively apply all the processing to the tweet and store it in processed\_tweet column. So that we don’t have to processed every time.
* We remove white spaces by tokenize the string and join them.

**Feature Extraction :**

If any features in these are present we increase the “PPI count” column by 1.

* **Email:**

re.findall("[a-zA-Z0-9\_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+",tweet)

If some word is in format of (\*\*\*@\*\*\*.\*\*\*) it will find it.

## **Sexuality**

(re.findall("(?: gay | transgender | lesbian | homophobic )",tweet))

It will find if any 4 of these words present in an tweet.

* **Paypal :**

re.findall("(?:paypal|my paypal|has paypal|have paypal|via paypal|use paypal|her paypal)",tweet))

We will match the different context of conversion to extract the match.

* **Phone number :**

re.findall("[a-z. ]+ \+?1?\s\*\(?-\*\.\*(\d{3})\)?\.\*-\*\s\*(\d{3})\.\*-\*\s\*(\d{4})$",tweet))

<https://stackoverflow.com/questions/8634139/phone-validation-regex>

* **Legal status :**

For this I scrap all the citizen name in each country.

<https://www.myenglishpages.com/site_php_files/vocabulary-lesson-countries-nationalities.php>

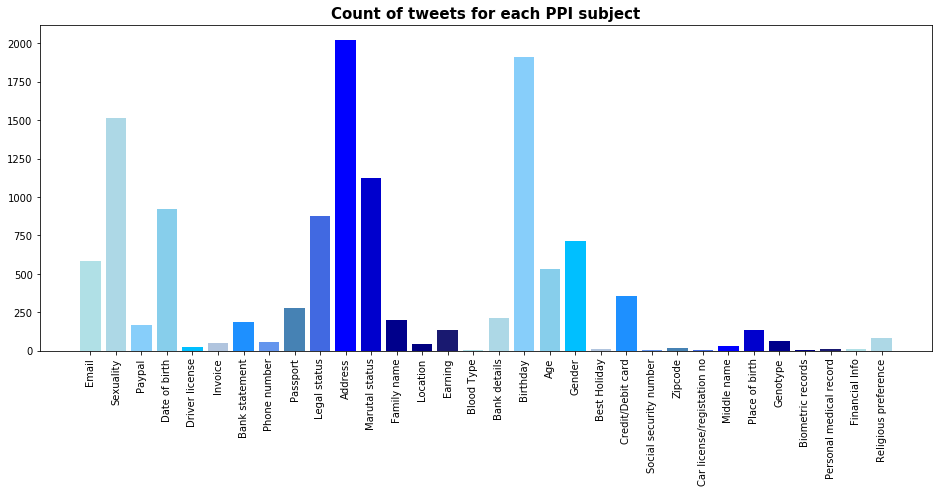
And match the all the pattern in tweet.

# **religious preference:**

Similarly for religion I get all the religion name from wikipedia and match the pattern.

<https://en.wikipedia.org/wiki/List_of_religions_and_spiritual_traditions>

* Then we will visualize each count for each PPI subject.



* **Modeling :**

After count all the PPI words in each tweet we observe that a tweet has a maximum number 3 PPI words and minimum is 0.

We categorize PPI words with high risk, medium risk, low risk.

0 - low risk

1 - medium risk

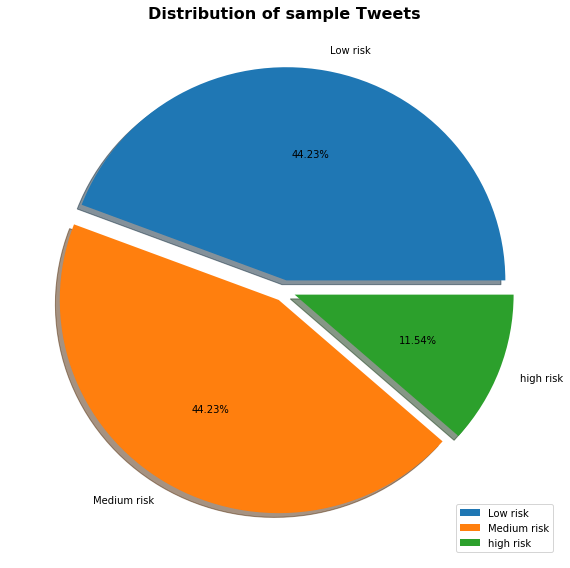
Greater than 2 : High risk

The distribution of all the category is uneven. For example :

|  |  |
| --- | --- |
| **PPI\_word\_count** | **total tweet** |
| 0 | 1587977 |
| 1 | 11759 |
| 2 | 261 |
| 3 | 3 |

* Less than 1% tweet has PPI words. So in this distribution we can’t build the model. Otherwise it will be always biased towards prediction 0 PPI\_words.
* So we take a sample of tweets data with approximately even distribution.

|  |  |
| --- | --- |
| **PPI\_words** | **Sample size** |
| 0 | 1000 |
| 1 | 1000 |
| 2 | 261 |



* This distribution will be used for building the model.
* We will build the model using **scikit-learn** library in python.
* After we load the data , we split the data to train set and test set by using the sklearn’s train\_test\_split function.

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

X\_train, X\_test, y\_train, y\_test = train\_test\_split(result['processed\_tweet'], result['PPI count'], test\_size=0.25, random\_state=42)

* We pass the tweet as a feature and PPI\_word to label. And we take 25% test.
* **Extracting features from text files** :

Text files are actually series of words (ordered). In order to run machine learning algorithms we need to convert the text files into numerical feature vectors. We will be using bag of words model for our example. Briefly, we segment each text file into words (for English splitting by space), and count total times each word occurs in each document and finally assign each word an integer id. Each unique word in our dictionary will correspond to a feature (descriptive feature).

Scikit-learn has a high level component which will create feature vectors for us ‘CountVectorizer’. More about it [here](http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html).

from sklearn.feature\_extraction.text import CountVectorizer

count\_vect = CountVectorizer()

X\_train\_counts = count\_vect.fit\_transform(X\_train)

Here by doing ‘count\_vect.fit\_transform(X\_train)’, we are learning the vocabulary dictionary and it returns a Document-Term matrix. [n\_samples, n\_features].

**TF:** Just counting the number of words in each document has 1 issue: it will give more weightage to longer documents than shorter documents. To avoid this, we can use frequency (**TF - Term Frequencies**) i.e. #count(word) / #Total words, in each document.

**TF-IDF:** Finally, we can even reduce the weightage of more common words like (the, is, an etc.) which occurs in all document. This is called as **TF-IDF i.e Term Frequency times inverse document frequency.**

We can achieve both using below line of code:

from sklearn.feature\_extraction.text import TfidfTransformer

tfidf\_transformer = TfidfTransformer()

X\_train\_tfidf = tfidf\_transformer.fit\_transform(X\_train\_counts)

X\_train\_tfidf.shape

The last line will output the dimension of the Document-Term matrix ->

**Building the Model :**

There are various algorithms which can be used for text classification. We will use SVM algorithm in this project.

**Support Vector Machines (SVM):** Let’s try using a algorithm SVM. More about it [here](http://scikit-learn.org/stable/modules/svm.html).

>>> from sklearn.linear\_model import SGDClassifier

>>> text\_clf\_svm = Pipeline([('vect', CountVectorizer()),

... ('tfidf', TfidfTransformer()),

... ('clf-svm', SGDClassifier(loss='hinge', penalty='l2',

... alpha=1e-3, n\_iter=5, random\_state=42)),

... ])

>>> \_ = text\_clf\_svm.fit(twenty\_train.data, twenty\_train.target)

>>> predicted\_svm = text\_clf\_svm.predict(twenty\_test.data)

>>> np.mean(predicted\_svm == twenty\_test.target)

The accuracy we get is**~86.74%.**

#### 

#### 

#### **Grid Search**

Almost all the classifiers will have various parameters which can be tuned to obtain optimal performance. Scikit gives an extremely useful tool ‘GridSearchCV’.

**>>> from sklearn.model\_selection import GridSearchCV**

**>>> parameters = {'vect\_\_ngram\_range': [(1, 1), (1, 2)],**

**... 'tfidf\_\_use\_idf': (True, False),**

**... 'clf\_\_alpha': (1e-2, 1e-3),**

**... }**

Here, we are creating a list of parameters for which we would like to do performance tuning. All the parameters name start with the classifier name (remember the arbitrary name we gave). E.g. vect\_\_ngram\_range; here we are telling to use unigram and bigrams and choose the one which is optimal.

Next, we create an instance of the grid search by passing the classifier, parameters and n\_jobs=-1 which tells to use multiple cores from user machine.

**gs\_clf = GridSearchCV(text\_clf, parameters, n\_jobs=-1)**

**gs\_clf = gs\_clf.fit(twenty\_train.data, twenty\_train.target)**

This might take few minutes to run depending on the machine configuration.

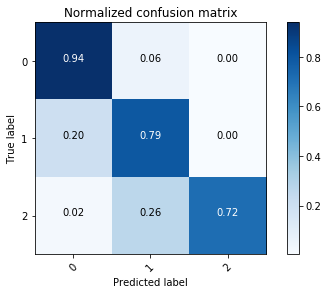
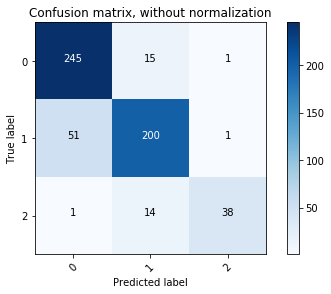
Lastly, to see the best mean score and the params, run the following code:

**gs\_clf.best\_score\_**

**gs\_clf.best\_params\_**

In this project are getting good accuracy score with default parameter so we don’t need to run grid search here.

* Finally we predict for test set and get the accuracy. And get the confusion matrix.

****

**Conclusion:** We have learned the classic problem in NLP, text classification. We learned about important concepts like bag of words, TF-IDF and SVM. We saw that for our data set, *Sometimes*, if we have enough data set, choice of algorithm can make hardly any difference. We also saw, how to perform grid search for performance tuning