Consumer financial protection bureau is responsible for consumer protection in the financial sector. Consumer complaint database is a collection of complaints about consumer financial products and services that is sent to companies for response.

The whole database in CSV format can be downloaded here. <https://www.consumerfinance.gov/data-research/consumer-complaints/#download-the-data>

The text data which would be used for analysis resides in “Consumer Complaint Narrative” column and classifier target would be “Product” column. Few more columns can be used if analysis has to be focused on particular bank/financial sector.

1)   First Step: Fetch Data using read\_csv into a Pandas Dataframe. Prepare training & test data set.

# Fetch CFPB Data , Filter with small samples

df\_data = pd.read\_csv('complaints.csv',usecols=["Product","Consumer complaint narrative","Company"],header=0,low\_memory=False).dropna()

df\_data = df\_data.loc[df\_data['Company'] == 'BANK OF AMERICA, NATIONAL ASSOCIATION']

df = df\_data.head(100)

#print(len(df),df.size)

#Prepare training matrix for further steps.

df\_attr = df['Consumer complaint narrative']

df\_target = df['Product']

# Prepare test matrix data set for later prediction

df\_t = df\_data.tail(100)

df\_t\_attr = df\_t['Consumer complaint narrative']

df\_t\_target = df\_t['Product']

2) Tuning hyperparameters through Grid Search method

The below code snippet basically search for the most optimum parameters for all methods used for vectorizing, transforming & classifier model. The method works by first declaring a pipeline. In this case, Countvectorizer() transform the complaint text data in a document term frequency matrix which is further converted into a TF-IDF value which is a short form for term frequency-inverse document frequency. This transformer helps in weeding out higher frequency words with frequent use in all the documents such as stop words. The last step is to use a classifier such as SGDClassifier. It is a form of linear classifer with SGD training. Stochastic Gradient Descent is one of the popular algorithm used in machine learning.

All of the methods has large numbers of parameters which can be optimized. But due to increased processing time, it is important to choose few important parameters where the search study needs to be focused.

a)max\_df (CountVectorizer) - When building vocabulary, ignore document frequency higher than the threshold value.

b)ngram\_range (CountVectorizer)- Unigram or Bigrams as features?

c) max\_iter (Classifier) - Maximum number of passes over training data (epochs)

d) Alpha (Classifier) - Constant that multiplies the regularization term. The higher the value, the stronger the regularization.

e) Penalty (Classifier) - The penalty (aka regularization term) to be used. Defaults to ‘l2’ which is the standard regularizer for linear SVM models. ‘l1’ and ‘elasticnet’ brings sparsity to the model.

# Tune hyperparameters through gridsearch methods

# #############################################################################

# Define a pipeline combining a text feature extractor with a simple

# classifier

pipeline = Pipeline([

    ('vect', CountVectorizer()),

    ('tfidf', TfidfTransformer()),

    ('clf', SGDClassifier()),

])

# uncommenting more parameters will give better exploring power but will

# increase processing time in a combinatorial way

parameters = {

    'vect\_\_max\_df': (0.5, 0.75, 1.0),

    # 'vect\_\_max\_features': (None, 5000, 10000, 50000),

    'vect\_\_ngram\_range': ((1, 1), (1, 2)),  # unigrams or bigrams

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

    # 'tfidf\_\_norm': ('l1', 'l2'),

    'clf\_\_max\_iter': (20,),

    'clf\_\_alpha': (0.00001, 0.000001),

    'clf\_\_penalty': ('l2', 'elasticnet'),

    # 'clf\_\_max\_iter': (10, 50, 80),

}

if \_\_name\_\_ == "\_\_main\_\_":

    # multiprocessing requires the fork to happen in a \_\_main\_\_ protected

    # block

    # find the best parameters for both the feature extraction and the

    # classifier

    grid\_search = GridSearchCV(pipeline, parameters, n\_jobs=-1, verbose=1,cv=5)

    print("Performing grid search...")

    print("pipeline:", [name for name, \_ in pipeline.steps])

    print("parameters:")

    pprint(parameters)

    t0 = time()

    #grid\_search.fit(data.data, data.target)

    grid\_search.fit(df\_attr,df\_target)

    print("done in %0.3fs" % (time() - t0))

    print()

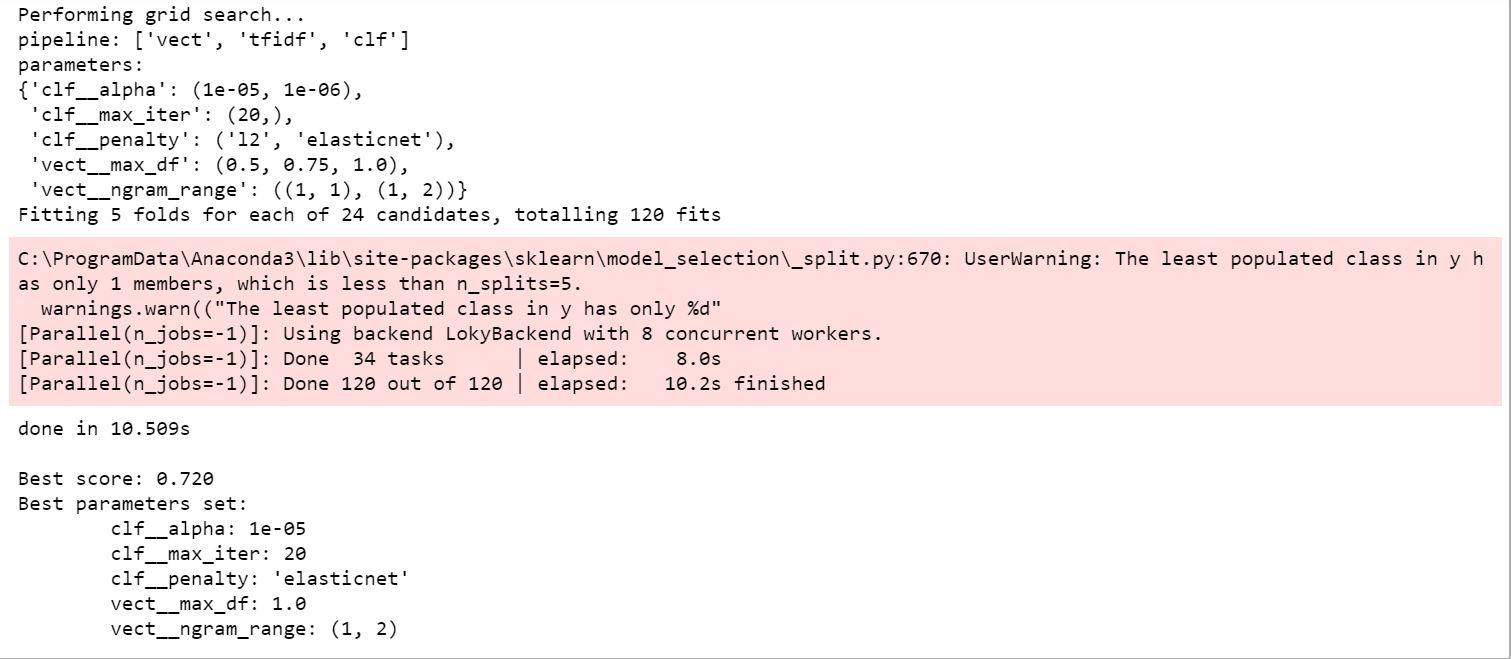
    print("Best score: %0.3f" % grid\_search.best\_score\_)

    print("Best parameters set:")

    best\_parameters = grid\_search.best\_estimator\_.get\_params()

    for param\_name in sorted(parameters.keys()):

        print("\t%s: %r" % (param\_name, best\_parameters[param\_name]))

Add alt text

3) The above method has provided us with best parameters set for the selected small sample of data. Now, we will use the same pipeline method to build our classifier model with few more filters.

a)   Stop\_words = ‘English’ : Remove common English stop words

b)   token\_pattern = r'(?u)\b[A-Za-z]+\b') – Regular expression to only select words. Eliminate all digits and other special, foreign characters.

# Build classifier model using best parameters set. Remove english stop words & only select words token

# This method uses pipeline..

text\_clf = Pipeline([

    ('vect', CountVectorizer(stop\_words = 'english',max\_df = 0.5,ngram\_range = (1,2),token\_pattern = r'(?u)\b[A-Za-z]+\b')),

    ('tfidf', TfidfTransformer()),

    ('clf', SGDClassifier(alpha = 1e-5,max\_iter = 20,penalty = 'elasticnet')),

])

text\_clf.fit(df\_attr,df\_target)

predictions = text\_clf.predict(df\_t\_attr)

from sklearn.metrics import accuracy\_score

print('Test accuracy is {}'.format(accuracy\_score(df\_t\_target, predictions)))

4) Pipeline method in above steps takes care of vectorizing and transforming matrices. But if you are interested in knowing intermediate steps and how this really works, you can test using following code snippet.

a) Vectorizer - The output of vectorizer.fit\_transform is a sparse matrix which excludes cells with zero values. This sometimes can be difficult to understand and hence the preferred method is to have full matrix using toarray() to preserve the shape.

b) TfidfTransformer - Similar transformation as above

c) Label Encoder - Label encoder basically encode unique integer number for all unique entries in target text column.

d) SGDClassifier - Finally, fit the model using this transformed matrices.

# Building same model without pipeline method to know intermediate steps..

vectorizer = CountVectorizer(stop\_words = 'english',max\_df=0.5,ngram\_range=(1,2),token\_pattern = r'(?u)\b[A-Za-z]+\b')

X\_v = vectorizer.fit\_transform(df\_attr).toarray()

#print(vectorizer.get\_feature\_names())

#print(X\_v)

transformer=TfidfTransformer()

X\_t = transformer.fit\_transform(X\_v).toarray()

#print(X\_t)

from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()

y\_t = le.fit\_transform(df\_target)

#print(y\_t)

classifier = SGDClassifier(penalty = 'elasticnet',alpha = 1e-6,max\_iter = 20,random\_state = 3000)

classifier.fit(X\_t,y\_t)

predictions = classifier.predict(X\_t)

from sklearn.metrics import accuracy\_score

print('Test accuracy is {}'.format(accuracy\_score(y\_t, predictions)))

This is just an attempt to show you how to use NLP techniques on text data to build a classifier model. In order to get good model accuracy, the training has to be performed on large number of records. Additionally, gridsearch also has to be applied on more number of parameters with wider range.

Reference:

1) <https://www.consumerfinance.gov/>

2) <https://en.wikipedia.org/wiki/Stochastic_gradient_descent>

3) <https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html#sklearn.linear_model.SGDClassifier>