Text Classification Project

jupyter tt_project_review_text_ONLY Last Checkpoint: 38 minutes ago (autosaved)

Logout

File Edit View Insert Cell Kernel Widgets Help

Trusted | Python 3

Run | Markdown

1. Read in json, get only the text data and labels cols; change labels to 'neg', 'mixed', and 'pos' only

In [2]:

```
%%time
path=''
file='kindle_reviews.json'
df = pd.read_json(path_or_buf=path+file, lines=True, encoding='utf-8')     #, orient=None, typ='frame', dtype=True, convert_axes=
print('Length of text: {}'.format(len(df)))
```

```
Length of text: 982619
Wall time: 13.1 s
```

In [3]:

```
df.head()
```

Out[3]:

	asin	helpful	overall	reviewText	reviewTime	reviewerID	reviewerName	summary	unixReviewTime
0	B000F83SZQ	[0, 0]	5	I enjoy vintage books and movies so I enjoyed ...	05 5, 2014	A1F6404F1VG29J	Avidreader	Nice vintage story	1399248000
1	B000F83SZQ	[2, 2]	4	This book is a reissue of an old one; the auth...	01 6, 2014	AN0N05A9LIJEQ	critters	Different...	1388966400
2	B000F83SZQ	[2, 2]	4	This was a fairly interesting read. It had ol...	04 4, 2014	A795DMNCJILA6	dot	Oldie	1396569600
3	B000F83SZQ	[1, 1]	5	I'd never read any of the Amy Brewster mysteri...	02 19, 2014	A1FV0SX13TWVXQ	Elaine H. Turley "Montana Songbird"	I really liked it.	1392768000
4	B000F83SZQ	[0, 1]	4	If you like period pieces - clothing, lingo, y...	03 19, 2014	A3SPTOKDG7WBLN	Father Dowling Fan	Period Mystery	1395187200

In [4]:

```
len(df.asin.unique())
```

Out[4]: 61934

In [5]:

```
df_text = df[['reviewText', 'overall']].copy()          # copy only certain columns to another df
#df = None                                               # release memory
```

Code repository: https://github.com/agnedil/Portfolio/tree/master/01-NLP/02-Sklearn-Text-Classifier

DATASET: Amazon Product Reviews (Kindle)
http://snap.stanford.edu/data/amazon/productGraph/

Deduplicated dataset in a one-review-per-line json file. Example of one review:


```
{
  "reviewerID": "A2SUAM1J3GNN3B",
  "asin": "0000013714",
  "reviewerName": "J. McDonald",
  "helpful": [2, 3],
  "reviewText": "I bought this for my husband who plays the piano.  He is having a wonderful time playing
                 these old hymns.  The music  is at times hard to read because we think the book was published
                 for singing from more than playing from.  Great purchase though!",
  "overall": 5.0,
  "summary": "Heavenly Highway Hymns",
  "unixReviewTime": 1252800000,
  "reviewTime": "09 13, 2009"
}
```


TASK: develop a classifier first based solely on text, then include additional features. The labels should be converted from 1-5 stars to 3 labels: positive (4 and 5 stars), negative (1 and 2), mixed (3).
Avoid overfitting (by using cross-validation)

CHALLENGES: a huge dataset, about a million reviews, json file is almost 1 GB in size; limited time for completion

SOLUTION:

- Using sklearn classifiers after TFIDF vectorization which allows to pass text to a classifier as a sparse matrix of terms
- Stopword removal after selecting the most efficient list of stopwords
- Feature engineering – unigram, bigram, and trigram language models with a cross-validated parameter grid search on classifier arguments
- Grid search is time-consuming – conduct it on a partial dataset, then use the best classifier on the full dataset
- Use F-1 measure as a metric since the dataset is imbalanced

jupyter tt_project_review_text_ONLY Last Checkpoint: an hour ago (autosaved)

Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

Run Markdown

2. Reduce size of data, get reduced-size dataset and labels

In [10]:
```
# get a random sample from the dataframe whose size is manageable for cross-validation and grid search
# with more computing resources and/or time, this can be done on a larger data set
length = len(df_text)
df_text = df_text.sample(n=length)
df_text_short = df_text.sample(n=int(length/23))
print('Length of the sample is {}'.format(len(df_text_short)))
```
```
Length of the sample is 42722
```

In [11]:
```
# count of unique labels
df_text_short.overall.value_counts()
```
Out[11]:
```
pos      35988
mixed     4158
neg       2576
Name: overall, dtype: int64
```

In [12]:
```
# this is our train set and labels
data = df_text_short['reviewText'].values
labels = df_text_short['overall'].values
```

3. Experimenting with different lists of stopwords and selecting the most efficient one

In [13]:
```
# GENERATING A LIST OF STOPWORDS
# these various stopword lists and the combined joint list were tested on the same classifier (MultinomialNB) with the same
# parameters, and it was found that the lemur list and the combined list, the latter includes the former, were the most
# efficient ones

from sklearn.feature_extraction import stop_words
from nltk.corpus import stopwords

print('Sklearn:')
stopwords_sklearn = list(stop_words.ENGLISH_STOP_WORDS)        # 318 words
```

The solution may seem trivial, but again – the dataset was huge and the time was limited.
A potentially better solution would be renting a GPU instance in the cloud and trying PyTorch or fast.ai, but this would require more time and resources.

Even for this solution, one cross-validated grid search run could take up to 2 hours on a 32 GB machine and I had to make a lot of them:

```
In [25]: %%time
         best_SVM = clf_GridSearchCV(svc, data, labels, svc_param_grid)    # parameter grid search SVM

0.868508 (0.001322) with: {'clf__C': 1.0, 'vect__max_df': 0.75, 'vect__min_df': 15, 'vect__ngram_range': (1, 1)}
0.869298 (0.000826) with: {'clf__C': 1.0, 'vect__max_df': 0.75, 'vect__min_df': 15, 'vect__ngram_range': (1, 2)}
0.869503 (0.000682) with: {'clf__C': 1.0, 'vect__max_df': 0.75, 'vect__min_df': 15, 'vect__ngram_range': (1, 3)}
0.869181 (0.001742) with: {'clf__C': 1.0, 'vect__max_df': 0.75, 'vect__min_df': 25, 'vect__ngram_range': (1, 1)}
0.868245 (0.001156) with: {'clf__C': 1.0, 'vect__max_df': 0.75, 'vect__min_df': 25, 'vect__ngram_range': (1, 2)}
0.868157 (0.001023) with: {'clf__C': 1.0, 'vect__max_df': 0.75, 'vect__min_df': 25, 'vect__ngram_range': (1, 3)}
0.869181 (0.000799) with: {'clf__C': 1.0, 'vect__max_df': 0.75, 'vect__min_df': 50, 'vect__ngram_range': (1, 1)}
0.868011 (0.001459) with: {'clf__C': 1.0, 'vect__max_df': 0.75, 'vect__min_df': 50, 'vect__ngram_range': (1, 2)}
0.867894 (0.001566) with: {'clf__C': 1.0, 'vect__max_df': 0.75, 'vect__min_df': 50, 'vect__ngram_range': (1, 3)}
0.869123 (0.000339) with: {'clf__C': 1.0, 'vect__max_df': 0.75, 'vect__min_df': 100, 'vect__ngram_range': (1, 1)}
0.868040 (0.000688) with: {'clf__C': 1.0, 'vect__max_df': 0.75, 'vect__min_df': 100, 'vect__ngram_range': (1, 2)}
0.868011 (0.000607) with: {'clf__C': 1.0, 'vect__max_df': 0.75, 'vect__min_df': 100, 'vect__ngram_range': (1, 3)}

LinearSVC cross-validated F-1 score with grid search: 0.8734
Confusion matrix:
[[ 196   68  616]
 [  96  207  197]
 [  92   13 7060]]

Wall time: 2h 10min 42s
```

Step 1. Classifier selection

- Several classifiers were tested on a limited dataset with cross-validation (see on the right)
- **Naïve Bayes** and **SVM** were selected based on reviewing mean train and test scores and past experience with their efficiency.
- They were built into a **Pipeline** along with the **TfidfVectorizer**
- A number of Pipeline parameters were selected to be tackled during the **grid search** (see below): the max and min document frequency of words, a choice of a unigram, bigram, or trigram model, and classifier parameters

```
In [14]: # SELECTING THE MOST EFFICIENT CLASSIFIER ON A LIMITED DATASET

         # potential candidates
         clfs = [MultinomialNB(),
                 svm.LinearSVC(),
                 LogisticRegression(),
                 KNeighborsClassifier(n_neighbors=3),
                 GradientBoostingClassifier(),
                 DecisionTreeClassifier(),
                 RandomForestClassifier(),
                 SGDClassifier()]

         # vectorize data
         vectorizer = TfidfVectorizer(analyzer='word', stop_words=stopwords_combined, min_df=5, max_df=0.25, ngram_range=(1, 2))
         matrix = vectorizer.fit_transform(data)

         # try each classifier on the data
         for clf in clfs:
             #scoring = ['precision_macro', 'recall_macro']        # if using this, add scoring=scoring to cross_validate()
             scores = cross_validate(clf, matrix, labels, cv=3)
             print('--------------------------------')
             print(str(clf))
             print('----------------------------------')
             for key, values in scores.items():
                     print(key,' mean ', values.mean())
                     print(key,' std ', values.std())
```

```
----------------------------------
MultinomialNB(alpha=1.0, class_prior=None, fit_prior=True)
-----------------------------------
fit_time  mean  0.10594948132832845
fit_time  std  0.0036903773744526307
score_time  mean  0.015373150507609049
score_time  std  0.0018912696447948786
test_score  mean  0.842446517243 1659
test_score  std  8.267247775353001e-05
train_score  mean  0.8425869594139265
train_score  std  0.00011325272724816022
---------------------------------
LinearSVC(C=1.0, class_weight=None, dual=True, fit_intercept=True,
     intercept_scaling=1, loss='squared_hinge', max_iter=1000,
     multi_class='ovr', penalty='l2', random_state=None, tol=0.0001,
     verbose=0)
-----------------------------------
```

```
In [19]: %%time
         # MultinomialNB()
         nb = MultinomialNB()
         nb_param_grid = {
                 'vect__max_df':[0.25, 0.5,0.75],
                 'vect__min_df':[5,15,25,50,100],
                 'vect__ngram_range':[(1, 1),(1, 2),(1, 3)],
                 'clf__alpha':[0.1,0.25,0.5,0.75,1.0]
             }
         clf_simple(nb, data, labels)    # straightforward NB
```

Step 2. Cross-validated parameter grid search

In [18]:

```
# GridSearchCV
def clf_GridSearchCV(classifier, data, labels, param_grid):

    # split data into train and test sets; create pipeline
    trainX, testX, trainY, testY = train_test_split(data, labels, test_size = 0.2, random_state = 43)
    clf = Pipeline([('vect', CountVectorizer(analyzer='word', stop_words=stopwords_combined, min_df=5, max_df=0.5, ngram_range=(
                    ('tfidf', TfidfTransformer()),
                    ('clf', classifier),
                  ])

    # get classifier's name to print results; otherwise, this function needs another argument
    clf_string = str(classifier)
    idx = clf_string.find("(")
    classifier_name = clf_string[:idx]

    # do 3-fold cross validation for each of the possible combinations of the parameter values above
    grid = GridSearchCV(clf, cv=3, param_grid=param_grid, scoring='f1_micro')
    grid.fit(trainX, trainY)

    # summarize results
    print("Best: %f using %s" % (grid.best_score_,
        grid.best_params_))
    means = grid.cv_results_['mean_test_score']
    stds = grid.cv_results_['std_test_score']
    params = grid.cv_results_['params']
    for mean, stdev, param in zip(means, stds, params):
        print("%f (%f) with: %r" % (mean, stdev, param))

    # train and predict on test instances using the best configs found in the CV step

    #predictions = grid.best_estimator_.predict(testX)                # this is how to find the best estimator
    #testX = grid.best_estimator_.named_steps['tfidf'].transform(testX)  # this is how to find indiv. components (same for pipel
    predictions=grid.predict(testX)                                   # called on the best estimator by default
    score = metrics.f1_score(testY, predictions, average='micro')
    cm    = metrics.confusion_matrix(testY, predictions)
    print('{} cross-validated F-1 score with grid search: {:0.4f}'.format(classifier_name, score))
    print('Confusion matrix:')
```

RESULTS

Classification F1 score = approx. 0.9

Text only:

10. Running Naive Bayes and SVM with the Best Parameters from Grid Search on the full dataset

In [26]:
```
# create full dataset and labels
full_data = df_text['reviewText'].values
full_labels = df_text['overall'].values
```

In [27]:
```
%%time
# run the two best classifier on it
for best_clf in [best_NB, best_SVM]:

    # split into train and test sets
    trainX, testX, trainY, testY = train_test_split(full_data, full_labels, test_size = 0.2, random_state = 43)
    clf = best_clf.fit(trainX, trainY)

    # predict and compute metrics
    predictions = clf.predict(testX)
    score = metrics.f1_score(testY, predictions, average='micro')
    cm    = metrics.confusion_matrix(testY, predictions)
    print('The best {} F-1 score on full dataset: {:0.4f}'.format('Naive Bayes' if best_clf==best_NB else 'SVM', score))
    print('Confusion matrix:')
    print(cm)
    print()
```

```
The best Naive Bayes F-1 score on full dataset: 0.8744
Confusion matrix:
[[  3956   1338  13950]
 [  1598   4652   5200]
 [  2153    442 163235]]

The best SVM F-1 score on full dataset: 0.8845
Confusion matrix:
[[  5304   2006  11934]
 [  2069   6118   3263]
 [  2703    726 162401]]

Wall time: 17min 30s
```

Text + additional data
(summary, product ID, reviewer name)

10. Running Naive Bayes and SVM with the Best Parameters from Grid Search on the full dataset

In [26]:
```
# create full dataset and labels
full_data = df_text['reviewText'].values
full_labels = df_text['overall'].values
```

In [27]:
```
%%time
# run the two best classifier on it
for best_clf in [best_NB, best_SVM]:

    # split into train and test sets
    trainX, testX, trainY, testY = train_test_split(full_data, full_labels, test_size = 0.2, random_state = 43)
    clf = best_clf.fit(trainX, trainY)

    # predict and compute metrics
    predictions = clf.predict(testX)
    score = metrics.f1_score(testY, predictions, average='micro')
    cm    = metrics.confusion_matrix(testY, predictions)
    print('The best {} F-1 score on full dataset: {:0.4f}'.format('Naive Bayes' if best_clf==best_NB else 'SVM', score))
    print('Confusion matrix:')
    print(cm)
    print()
```

```
The best Naive Bayes F-1 score on full dataset: 0.8832
Confusion matrix:
[[  6189   1657  11385]
 [  2036   5718   3598]
 [  3742    543 161656]]

The best SVM F-1 score on full dataset: 0.8971
Confusion matrix:
[[  6591   1997  10643]
 [  2089   6884   2379]
 [  2577    532 162832]]

Wall time: 16min 46s
```

In an attempt to improve the score, a library different than sklearn was attempted. It is specifically designed for text-related tasks:

Metapy - a Python wrapper for the **MeTa toolkit** (modern text analysis in C++):

- Text tokenization (incl. deep semantic features like parse trees);
- Inverted and forward indexes with compression and various caching strategies built from text corpora;
- Ranking functions for searching the indexes;
- Text classification (based on indexes saved on disk);
- Language and topic modeling;

References:

Metapy - https://github.com/meta-toolkit/metapy

MeTa - https://meta-toolkit.org/

Save `ForwardIndex` to disk.

In [8]:
```
%%time
fidx = metapy.index.make_forward_index('ceeaus-config.toml')
```
```
Wall time: 59.9 s
```

In [9]:
```
# inverted index - not needed for these classifiers
#iidx = metapy.index.make_inverted_index('ceeaus-config.toml')
```

The feature set used for classification depends on the settings in the configuration file *at the time of indexing*. Thus, if you change your `analyzer` pipeline (or other settings) - **reindex** your documents!

Decide what kind of dataset we're using - for binary classification (MeTA's `BinaryDataset`) or multi-class classification (`MulticlassDataset`). To see the number of labels:

In [10]:
```
fidx.num_labels()
```
Out[10]:
```
3
```

Looks like we need a `MulticlassDataset` to predict which of these three labels a document should have (but if we are interested in one particular class only, we might use a `BinaryDataset`).

For now, let's focus on the multi-class case, as that likely makes the most sense for this kind of data. Since the dataset is small enough, we can load all documents into memory at once like this.

In [11]:
```
dset = metapy.classify.MulticlassDataset(fidx)
len(dset)
```
Out[11]:
```
982619
```

Since datasets may be large, it's beneficial to avoid creating copies of them (e.g. to shuffle them) => you can operate with a `DatasetView`

In [19]:
```
mtrx = metapy.classify.cross_validate(lambda fold: metapy.classify.NaiveBayes(fold), view, 5)
```

`Cross_validate()` also returns a `ConfusionMatrix` ; arguments - function to create the trained classifiers for each fold, a dataset view (all documents), and the number of folds.

In [20]:
```
print(mtrx)
mtrx.print_stats()
```
```
             mixed     neg       pos
           ------------------------------
    mixed | 0.448     0.259     0.293
      neg | 0.176     0.722     0.103
      pos | 0.094     0.0369    0.869


------------------------------------------------------------
Class       F1 Score    Precision   Recall      Class Dist
------------------------------------------------------------
mixed       0.379       0.329       0.448       0.0979
neg         0.536       0.426       0.722       0.0582
pos         0.91        0.955       0.869       0.844
------------------------------------------------------------
Total       0.84        0.863       0.819
------------------------------------------------------------
982615 predictions attempted, overall accuracy: 0.819
```

In [29]:
```
mtrx = metapy.classify.cross_validate(lambda fold: metapy.classify.OneVsAll(fold, metapy.classify.SGD, loss_id='hinge'), view, 5
print(mtrx)
mtrx.print_stats()
```
```
             mixed     neg       pos
           ------------------------------
    mixed | 0.108     0.104     0.789
      neg | 0.041     0.469     0.49
      pos | 0.00412   0.00278   0.993


------------------------------------------------------------
Class       F1 Score    Precision   Recall      Class Dist
------------------------------------------------------------
mixed       0.184       0.643       0.108       0.0979
neg         0.557       0.686       0.469       0.0582
pos         0.938       0.888       0.993       0.844
------------------------------------------------------------
Total       0.864       0.852       0.876
------------------------------------------------------------
982615 predictions attempted, overall accuracy: 0.876
```

Figure 1. Cross-validated F1 score for Metapy Naïve Bayes (left) and SVM (right)