5- class Sentiment Analysis

Using

Support Vector Machine,
Decision Tree,
Random Forest,
Adaboosting,
Gradient Boosting

Submitted by
Shristi Maskay

Abstract

The data set we have is a categorical data, consisting of Phrase and sentiment levels 0-5 as very sad, sad, neutral, happy, very happy. The task is to use various classification model to classify the dataset into categories and to used different param_grids and find the best parameters and best model that can be used

Required Libraries used

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

1. Import Dataset

Using Pandas library to import the dataset and store the data in dataframe and then performing replace action to replace NaN data with 0 in the dataframe

```
df = pd.read_csv('/home/shristi/Downloads/SVM/sentiment_5_class - sentiment_5_class.csv')
df.head()

df.replace('NaN', 0)
```

	Phrase	Sentiment
0	injects just enough freshness into the proceed...	3
1	that	2
2	never plays as dramatic even when dramatic thi...	0
3	None of this is very original , and it is n't ...	0
4	, Madonna gives her best performance since Abe...	3
...	...	...
18384	to balance pointed , often incisive satire and...	3
18385	have to be a most hard-hearted person not to b...	4
18386	could young romantics out on a date	3
18387	could be this good	3
18388	such a dungpile	0

18389 rows × 2 columns

2. Data exploration: Data analysis and visualization

The data set we have is a categorical data, consisting of columns: Phrase and sentiment. Sentiment column shows the level of sentiments range from 0-5 as very sad, sad, neutral, happy and very happy as per the contents in the phrases.
The below command helps to get the columns in the dataframe

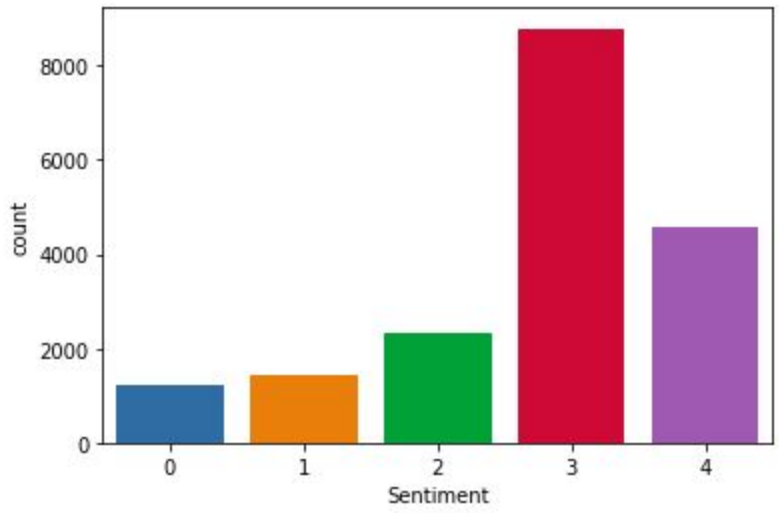
```
1 df.columns
```

```
Index(['Phrase', 'Sentiment'], dtype='object')
```

Now creating a countplot for Sentiment data got the graph of the phrase categorized into 5 classes. We can state that existing data seems to be imbalanced set as we have data of happy to be quite high in number than data of sad, very sad and neutral

```
1 sns.countplot(x="Sentiment", data=df)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f3b42eb9828>
```



3. Data Split

To get the model for proper classification now we need to split the data into train set and test set and used the train set to fit the model and use the test set to evaluate the model prediction and determine the accuracy of the model

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1, stratify=y
```

Splitting the data to train and test set considering 80% of data to train and 20% to test and using the data from the Phrase column need to predict the categories on which the phrase lies.

4. Features: Extraction and normalization

Since the data in Phrase Column is in a text form so, converting it to a matrix of token counts. Initially creating an a-priori dictionary using the words in phrases and then using analyzer for feature selection then the number of features will be equal to the vocabulary size found by analyzing the data.

Using Count Feature Vectorization

It is used to create vectors having dimensionality equal to the size of our vocabulary, and if the phrase consists of words available in vocab we put 1 in dimension and if not found then 0 for each and every word on the phrase. The result thus will be very large vector which will help to give accurate count

```
from sklearn.feature_extraction.text import CountVectorizer

def count_feature_vectorization(X_train, X_test):
    c_vectorizer = CountVectorizer()
    c_vectorizer.fit(X_train) #only use training data set to fit model
    c_vectorizer.get_feature_names() #vocab list
    print(c_vectorizer)

    #to vecorize the train and test data i.e encoding the phrases as per the dictionary created
    c_train_v =c_vectorizer.transform(X_train)
    c_test_v = c_vectorizer.transform(X_test)
    return c_train_v, c_test_v, c_vectorizer
```

```
c_train_v, c_test_v, c_vectorizer = count_feature_vectorization(X_train, X_test)
c_train_v.toarray()
c_test_v.toarray()
```

```
CountVectorizer(analyzer='word', binary=False, decode_error='strict',
                dtype=<class 'numpy.int64'>, encoding='utf-8', input='content',
                lowercase=True, max_df=1.0, max_features=None, min_df=1,
                ngram_range=(1, 1), preprocessor=None, stop_words=None,
                strip_accents=None, token_pattern='(?u)\\b\\w\\w+\\b',
                tokenizer=None, vocabulary=None)
```

Using Tf-IDF Feature Vectorization

This is used to transform a count matrix to tf-idf representation i.e term frequency times inverse document frequency. The purpose of using Tf-idf is to scale down the impact of tokens that occur very frequently in given data.

```
from sklearn.feature_extraction.text import TfidfVectorizer

def Tfidf_feature_vectorization(X_train, X_test):

    t_vectorizer = TfidfVectorizer()
    t_vectorizer.fit(X_train) #only use training data set to fit model
    t_vectorizer.get_feature_names() #vocab list
    print(t_vectorizer)

    #to vecorize the train and test data i.e encoding the phrases as per the dictionary created
    t_train_v =t_vectorizer.transform(X_train)
    t_test_v = t_vectorizer.transform(X_test)
    return t_train_v, t_test_v, t_vectorizer
```

```
t_train_v, t_test_v, t_vectorizer = Tfidf_feature_vectorization(X_train, X_test)
t_train_v.toarray()
t_test_v.toarray()
```

```
TfidfVectorizer(analyzer='word', binary=False, decode_error='strict',
                dtype=<class 'numpy.float64'>, encoding='utf-8',
                input='content', lowercase=True, max_df=1.0, max_features=None,
                min_df=1, ngram_range=(1, 1), norm='l2', preprocessor=None,
                smooth_idf=True, stop_words=None, strip_accents=None,
                sublinear_tf=False, token_pattern='(?u)\\b\\w\\w+\\b',
                tokenizer=None, use_idf=True, vocabulary=None)
```

5. Model Building

5.1. SVM (Support Vector Machine)

“Support Vector Machine” (SVM) is a supervised machine learning algorithm that can be used for both classification or regression. However, it is mostly used in classification problems. It uses sets of training points in the function thus is memory efficient and has different kernel functions can be used in decision functions. Sometimes SVM might lead to overfitting when features are greater than number of samples. So in this case 5fold cross validation might be used which might be expensive.

To determine the best parameters for SVM model, the parameters used in model training are:

```
grid_param = [{'kernel': ['rbf'], 'gamma': [1e-3, 1e-4, 1e-2],
               'C': [1, 10, 100, 1000]},
              {'kernel': ['linear'], 'C': [1, 10, 100, 1000]}]
```

And considering the score from the fitted model we get the best parameters that can be used in classification for the specific model and develop the model. Here,
Best_parametes :
Score:

Library: **from sklearn.svm import SVC**

```
def svm_model(X_train_v, y_train):
    model = SVC(random_state = 1)
    model.fit(X_train_v, y_train)
    return model
```

Now fitting the model using vectorized train data we get a model and now using the model

Model evaluation using the test set.

```
def prediction(model, feature_data):
    y_pred = model.predict(feature_data)
    return y_pred
```

Using model generated using tf-idf vectorizer train data and count vectorized data we predicted the output and generated a metric report

```
from sklearn import metrics
print(metrics.classification_report(y_test, y_pred)
```

	precision	recall	f1-score	support
0	0.79	0.45	0.57	247
1	0.62	0.30	0.40	291
2	0.78	0.20	0.32	469
3	0.60	0.91	0.73	1759
4	0.76	0.52	0.62	912
accuracy			0.64	3678
macro avg	0.71	0.48	0.53	3678
weighted avg	0.68	0.64	0.61	3678

Classification report generated from model using count vectorized train data

	precision	recall	f1-score	support
0	0.81	0.55	0.65	247
1	0.64	0.44	0.52	291
2	0.74	0.52	0.61	469
3	0.68	0.89	0.77	1759
4	0.78	0.58	0.66	912
accuracy			0.71	3678
macro avg	0.73	0.60	0.64	3678
weighted avg	0.72	0.71	0.70	3678

Classification report generated from model using TF-IDF vectorizer train data

This shows that the model generated using TF-IDF is more accurate than Count vectorization

Model Selection

For Model selection, GridSearchCV library is used for hyper parameter tuning process to determine optimal values for the given model as the model depends on the specified

hyperparameter values. Cross validation process is performed in order to determine the best score and parameters for the model.

```
from sklearn.model_selection import GridSearchCV
```

```
grid_param = {'kernel': ('linear', 'rbf'), 'C':(1,10, 0.5)}
grid_param
```

```
{'kernel': ('linear', 'rbf'), 'C': (1, 10, 0.5)}
```

```
grid_param = [{'kernel': ['rbf'], 'gamma': [1e-3, 1e-4, 1e-2],
                     'C': [1, 10, 100, 1000]},
                    {'kernel': ['linear'], 'C': [1, 10, 100, 1000]}]
grid_param
```

```
[{'kernel': ['rbf'], 'gamma': [0.001, 0.0001, 0.01], 'C': [1, 10, 100, 1000]},
 {'kernel': ['linear'], 'C': [1, 10, 100, 1000]}]
```

```
from sklearn.metrics import make_scorer, f1_score
scorer = make_scorer(f1_score, average='micro')
```

```
#using tfidf vectorised train data
scorer = make_scorer(f1_score, average='micro')
clf_tfidf = GridSearchCV(SVC(), grid_param,scoring=scorer)
clf_tfidf.fit(t_train_v, y_train)
print(clf_tfidf)
print(clf_tfidf.best_score_, clf_tfidf.best_params_)
```

Performing gridsearch with kernel parameters and getting the best_score result we got,

```
GridSearchCV(cv=None, error_score=nan,
             estimator=SVC(C=1.0, break_ties=False, cache_size=200,
                           class_weight=None, coef0=0.0,
                           decision_function_shape='ovr', degree=3,
                           gamma='scale', kernel='rbf', max_iter=-1,
                           probability=False, random_state=None, shrinking=True,
                           tol=0.001, verbose=False),
             iid='deprecated', n_jobs=None,
             param_grid=[{'C': [1, 10, 100, 1000],
                          'gamma': [0.001, 0.0001, 0.01], 'kernel': ['rbf']},
                         {'C': [1, 10, 100, 1000], 'kernel': ['linear']}],
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring=make_scorer(f1_score, average=micro), verbose=0)
0.6851333505653415 {'C': 100, 'gamma': 0.01, 'kernel': 'rbf'}
```

Best_score = 0.6851
Best parameters = C:100, gamma:0.01, kernel: rbf

Now using the best_parameters obtained from the gridsearch

Final Model using best parameter for SVM

```
model = SVC(random_state=1, kernel="rbf", C=100.0, gamma=0.01)
model.fit(t_train_v, y_train)
y_pred = model.predict(t_test_v)
print(metrics.classification_report(y_test, y_pred))
#FInal result obtained
```

```
              precision    recall  f1-score   support

           0       0.73      0.66      0.70       247
           1       0.60      0.53      0.56       291
           2       0.68      0.58      0.63       469
           3       0.72      0.84      0.78      1759
           4       0.72      0.61      0.66       912

    accuracy                           0.71      3678
   macro avg       0.69      0.64      0.66      3678
weighted avg       0.71      0.71      0.71      3678
```

Classification report generated for final model

5.2. Decision Tree

Decision tree is another non-linear model formed from a combination of linear boundaries. It is simply a tree formed by a series of yes/no questions asked about data leading to a predicted class. This is an interpretable model because it makes classifications much like we do: we ask a sequence of queries about the available data we have until we arrive at a decision.

Attributes in decision tree model:

__init__(self, *, criterion='gini', splitter='best', max_depth=None, min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features=None, random_state=None, max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, class_weight=None, presort='deprecated', ccp_alpha=0.0)

Parameters used in model training:

```
grid_params = {
    'max_depth': (5, 10, 20, 50, 100, 500),
   'min_samples_split': (2, 3, 4, 6, 8, 16, 20)
}
```

```
GridSearchCV(cv=3, error_score=nan,
             estimator=DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None,
                                              criterion='gini', max_depth=None,
                                              max_features=None,
                                              max_leaf_nodes=None,
                                              min_impurity_decrease=0.0,
                                              min_impurity_split=None,
                                              min_samples_leaf=1,
                                              min_samples_split=2,
                                              min_weight_fraction_leaf=0.0,
                                              presort='deprecated',
                                              random_state=None,
                                              splitter='best'),
             iid='deprecated', n_jobs=None,
             param_grid={'max_depth': (5, 10, 20, 50, 100, 500),
                         'min_samples_split': (2, 3, 4, 6, 8, 16, 20)},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring=make_scorer(f1_score, average=micro), verbose=0)
```

Best_parameters: {'max_depth': 500, 'min_samples_split': 6
Best Score: 0.5981915944749566}

Final Model using best parameter for Decision Tree

```
: import sklearn.metrics as metrics
  model =DecisionTreeClassifier(random_state=1,max_depth=500,min_samples_split=6)
  model.fit(c_train_v, y_train)
  preds = model.predict(c_test_v)
  print(metrics.classification_report(y_test, preds))
```

```
              precision    recall  f1-score   support

           0       0.59      0.53      0.56       247
           1       0.54      0.33      0.41       291
           2       0.51      0.29      0.37       469
           3       0.63      0.82      0.71      1759
           4       0.69      0.55      0.61       912

    accuracy                           0.63      3678
   macro avg       0.59      0.50      0.53      3678
weighted avg       0.62      0.63      0.61      3678
```

5.3. Random Forest

It technically is an ensemble method of decision trees generated on a randomly split dataset. This collection of decision tree classifiers is also known as the forest. The individual decision trees are generated using an attribute selection indicator such as information gain, gain ratio, and Gini index for each attribute. Each tree depends on an independent random sample. In a classification problem, each tree votes and the most popular class is chosen as the final result. In the case of regression, the average of all the tree outputs is considered as the final result. It is simpler and more powerful compared to the other non-linear classification algorithms. It is considered as an accurate and robust method because of the use of a number of decision trees.

Steps:

1. Select random samples from a given dataset.
2. Construct a decision tree for each sample and get a prediction result from trees.
3. Perform a vote for each predicted result.
4. Select the prediction result with the most votes as the final prediction.

Parameters used in model training:

```
grid_params = {
   'max_depth': (3,10, 13, 50),
   'min_samples_split': (2, 4, 8, 10),
   'n_estimators':(10, 20),
}
```

```
: scorer = make_scorer(f1_score, average='micro')
  clf=GridSearchCV(RandomForestClassifier(),grid_params,scoring=scorer,cv=3)
  clf.fit(c_train_v, y_train)
```

```
: GridSearchCV(cv=3, error_score=nan,
               estimator=RandomForestClassifier(bootstrap=True, ccp_alpha=0.0,
                                                class_weight=None,
                                                criterion='gini', max_depth=None,
                                                max_features='auto',
                                                max_leaf_nodes=None,
                                                max_samples=None,
                                                min_impurity_decrease=0.0,
                                                min_impurity_split=None,
                                                min_samples_leaf=1,
                                                min_samples_split=2,
                                                min_weight_fraction_leaf=0.0,
                                                n_estimators=100, n_jobs=None,
                                                oob_score=False,
                                                random_state=None, verbose=0,
                                                warm_start=False),
               iid='deprecated', n_jobs=None,
               param_grid={'max_depth': (3, 10, 13, 50),
                           'min_samples_split': (2, 4, 8, 10),
                           'n_estimators': (10, 20)},
               pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
               scoring=make_scorer(f1_score, average=micro), verbose=0)
```

Best_parameters: {'max_depth': 50, 'min_samples_split': 2, 'n_estimators': 20}
Best Score: 0.5466657283990215

Final Model Score

```
: from sklearn.model_selection import cross_val_score

  forest_scores = cross_val_score(model, c_train_v, y_train, cv=10)
  forest_scores.mean()
```

```
: 0.5574050760352319
```

5.4. Adaboosting

Adaboosting depends on classification problems and aims in converting weak classifiers to strong. It is a weighted combination of M weak classifiers. In the process of iteration if the accuracy is higher than 50% the weight is positive and more accurate the classifier, larger the weight while less than 50% accuracy gives negative weight.

Parameter used for model training:

```
grid_params = {
   'learning_rate':( 0.001,0.1, 0.5,0.8 ),
   'n_estimators':(10,50,100),
}
```

```
scorer = make_scorer(f1_score, average='micro')
clf=GridSearchCV(AdaBoostClassifier(),grid_params,scoring=scorer,cv=3)
clf.fit(c_train_v, y_train)
```

```
GridSearchCV(cv=3, error_score=nan,
             estimator=AdaBoostClassifier(algorithm='SAMME.R',
                                          base_estimator=None,
                                          learning_rate=1.0, n_estimators=50,
                                          random_state=None),
             iid='deprecated', n_jobs=None,
             param_grid={'learning_rate': (0.001, 0.1, 0.5, 0.8),
                         'n_estimators': (10, 50, 100)},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring=make_scorer(f1_score, average=micro), verbose=0)
```

Best_parametes: {'learning_rate': 0.8, 'n_estimators': 100}
F1_score: 0.512677440441354

Final Model using Best parameter

```
model =AdaBoostClassifier(learning_rate=0.1, n_estimators=100)
model.fit(c_train_v, y_train)
preds = model.predict(c_test_v)
print(metrics.classification_report(y_test, preds))
```

```
              precision    recall  f1-score   support

           0       0.84      0.09      0.15       247
           1       0.33      0.00      0.01       291
           2       0.20      0.00      0.00       469
           3       0.49      1.00      0.65      1759
           4       0.54      0.02      0.03       912

    accuracy                           0.49      3678
   macro avg       0.48      0.22      0.17      3678
weighted avg       0.47      0.49      0.33      3678
```

5.5. Gradient Boosting

This model is used for regression and classification models to predict models in the form of an ensemble of weak prediction models. It is similar to other boosting models allowing optimization by using differential loss function. It helps in combination of weak learners into one strong learner in iterative fashion. It is used with decision trees. It's performance depends on the size of trees, and number of leaves. In cases when training sets are too closed it can lead to overfitting and that can be reduced with regularization technique

Parameter used for model training:

```
grid_params = {
   'min_samples_split': (2, 4, 6, 8, 10, 20),
   'n_estimators':(50,100),
}
```

```
scorer = make_scorer(f1_score, average='micro')
clf=GridSearchCV(GradientBoostingClassifier(),grid_params,scoring=scorer,cv=3)
clf.fit(c_train_v, y_train)
```

```
GridSearchCV(cv=3, error_score=nan,
             estimator=GradientBoostingClassifier(ccp_alpha=0.0,
                                                  criterion='friedman_mse',
                                                  init=None, learning_rate=0.1,
                                                  loss='deviance', max_depth=3,
                                                  max_features=None,
                                                  max_leaf_nodes=None,
                                                  min_impurity_decrease=0.0,
                                                  min_impurity_split=None,
                                                  min_samples_leaf=1,
                                                  min_samples_split=2,
                                                  min_weight_fraction_leaf=0.0,
                                                  n_estimators=100,
                                                  n_iter_no_change=None,
                                                  presort='deprecated',
                                                  random_state=None,
                                                  subsample=1.0, tol=0.0001,
                                                  validation_fraction=0.1,
                                                  verbose=0, warm_start=False),
             iid='deprecated', n_jobs=None,
             param_grid={'min_samples_split': (2, 4, 6, 8, 10, 20),
                         'n_estimators': (50, 100)},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring=make_scorer(f1_score, average=micro), verbose=0)
```

Best_parametes: {'min_samples_split': 2, 'n_estimators': 100}
Best F1_score: 0.5529199171928897

Final Model using Best parameter

```
model =GradientBoostingClassifier(min_samples_split=2, n_estimators=100)
model.fit(c_train_v, y_train)
preds = model.predict(c_test_v)
print(metrics.classification_report(y_test, preds))
```

```
              precision    recall  f1-score   support

           0       0.84      0.23      0.37       247
           1       0.76      0.11      0.19       291
           2       0.78      0.06      0.11       469
           3       0.53      0.94      0.68      1759
           4       0.68      0.29      0.40       912

    accuracy                           0.55      3678
   macro avg       0.72      0.33      0.35      3678
weighted avg       0.64      0.55      0.48      3678
```

Model Comparison

Model	Parameter	F1_score (training)	F1_score final
SVM	C:100, gamma:0.01, kernel: rbf	0.6851	0.75
Decision Tree	'max_depth': 500, 'min_samples_split': 6	0.5819	0.63
Random Forest	{'max_depth': 50, 'min_samples_split': 2, 'n_estimators': 20}	0.54666	0.5574
Adaboosting	{'learning_rate': 0.8, 'n_estimators': 100}	0.512677	0.49
Gradient Boosting	{'min_samples_split': 2, 'n_estimators': 100}	0.5529	0.55

From the 5 model comparison SVM is found to be best model for the categorical data classification

Code link https://github.com/Kristiee/SVM_sentiment

SVM https://github.com/Kristiee/SVM_sentiment/blob/master/SVM-sentiment-final.ipynb

Other 4 models (Decision tree, Random forest, adaboosting, gradient boosting)
https://github.com/Kristiee/SVM_sentiment/blob/master/sentiment_analysis_other_models.ipynb