NATURAL LANGUAGE PROCESSING COURSEWORK 1 REPORT
By Watchara Sirinaovakul 170779274

Files: **ex1_FULL.ipynb** = full implementaion
ex1A_1-3.ipynb = for Part A question 1 to 3

Part A: Deception detection (60 points)

The template file contains some functions to load in the dataset, but there are some missing parts that you are going to fill in.

1. (5 points) Start by implementing the parseReview and the preProcess functions. Given a line of a tab-separated text file, parseReview should return a triple containing the identifier of the review (as an integer), the review text itself, and the label (either 'fake' or 'real'). The preProcess function should turn a review text (a string) into a list of tokens. *Hint: you can start by tokenising on white space; but you might want to think about some simple normalisation too.*

parseReview

```
def parseReview(reviewLine):
    # Should return a triple of an integer, a string containing the review, and a string indicating the label
    if reviewLine[1] == '__label1__':
        label = 'fake'
    else:
        label = 'real'
    return (reviewLine[0], reviewLine[8], label)
```

The order of the reviewLine is (0)DOC_ID, (1)LABEL, (2)RATING, (3)VERIFIED_PURCHASE, (4)PRODUCT_CATEGORY, (5)PRODUCT_ID, (6)PRODUCT_TITLE, (7)REVIEW_TITLE, (8)REVIEW_TEXT and it is passed as a list.

So, output have to be (identifier, review text, label). I just extracted those information according to the index. Also, the label is written as __label1__ and __label2__. As a result, I decoded it as the code.

Refactored parseReview

```
import pandas as pd
data = pd.read_csv('amazon_reviews.txt', delimiter='\t')
data['LABEL'] = data['LABEL'].map({'__label1__': 'fake', '__label2__': 'real'})
selected_data = data[['DOC_ID', 'REVIEW_VOCTOR', 'LABEL']]
```

I rewrote the parseReview with pandas. As you can see the code is cleaner and easier to understand.

preProcess V1

```
from nltk.tokenize import word_tokenize
```

```
from nltk.stem import SnowballStemmer

stemmer = SnowballStemmer('english')

# Input: a string of one review
def preProcess(text):
    # Should return a list of tokens
    tokens = word_tokenize(text)
    result = [stemmer.stem(t) for t in tokens]
    return result
```

```
{'accuracy': 0.77355592927729488,
 'f1': 0.77352992997243142,
 'precision': 0.77386831296115577,
 'recall': 0.77355592927729488}
```

In my first version of pre-processing step. I didn't remove stopwords and use word_tokenize from nltk. This give me worse performance.

Updated preProcess

```
from nltk.tokenize import RegexpTokenizer
from nltk.corpus import stopwords
from nltk.stem import SnowballStemmer

stemmer = SnowballStemmer('english')
stopWords = set(stopwords.words('english'))
tokenizer = RegexpTokenizer(r'\w+')

def preProcess(text):
    tokens = tokenizer.tokenize(text)
    result = [stemmer.stem(t) for t in tokens if t not in stopWords]
    return result
```

```
{'accuracy': 0.85535714285714282,
 'f1': 0.81908931399004381,
 'precision': 0.90402229582435156,
 'recall': 0.85535714285714282}
```

I used 3 libraries: RegexpTokenizer, stopwords, SnowballStemmer. Firstly, I tokenize the review text and filter stopwords out and then stem them. This improved the accuracy significantly.

2. (10 points) The next step is to implement the toFeatureVector function. Given a preprocessed review (that is, a list of tokens), it will return a Python dictionary that has as its keys the tokens, and as values the weight of those tokens in the preprocessed reviews. The weight could be simply the number of occurrences of a token in the preprocessed review, or it could give more weight to specific words. While building up this feature vector, you may want to incrementally build up a global featureDict, which should be a list or dictionary that keeps track of all the tokens in the whole review dataset. While a global feature dictionary is not strictly required for this coursework, it will help you understand which features (and how many!) you are using to train your classifier and can help understand possible performance issues you encounter on the way. *Hint: start by using binary feature values; 1 if the feature is present, 0 if it's not.*

```
featureDict = {} # A global dictionary of features

def toFeatureVector(row):
    # Should return a dictionary containing features as keys, and weights as values
    tokens = row[0]
    feature_vector = {}
    for token in tokens:
        if token in feature_vector:
            feature_vector[token] += 1
        else:
            feature_vector[token] = 1

        if token in featureDict:
            featureDict[token] += 1
        else:
            featureDict[token] = 1
    return feature_vector
```

row is the input features, which are tokens of review text and other features such as verified user.

In this step, the weight is the number of occurrences.

3. (15 points) Using the loadData function already present in the template file, you are now ready to process the review data from amazon_reviews.txt. In order to train a good classifier, finish the implementation of the crossValidate function to do a 10-fold cross validation on the training data. Make use of the given functions trainClassifier and predictLabels to do the cross-validation. Make sure that your program stores the (average) precision, recall, f1 score, and accuracy of your classifier in a variable cv_results. *Hint: the package sklearn.metrics contains many utilities for evaluation metrics - you could try precision recall fscore support to start with.*

Cross-validation

```
def trainClassifier(trainData):
    print("Training Classifier...")
    pipeline =  Pipeline([('svc', LinearSVC(loss='hinge', max_iter=3000, C=1))])
    return SklearnClassifier(pipeline).train(trainData)

def predictLabels_cv(reviewSamples, classifier):
    return classifier.classify_many(map(lambda t: t[0], reviewSamples))

def crossValidate(dataset, folds):
    shuffle(dataset)
    cv_results = []
    foldSize = math.ceil(len(dataset)/folds)

    kf = KFold(n_splits=folds)

    scores = np.array([0,0,0,0])

    for train_index, test_index in kf.split(dataset):
        X_train, X_test = dataset[train_index], dataset[test_index]
        classifier = trainClassifier(X_train)
        y_pred = predictLabels_cv(X_test, classifier)
        y_true = X_test[:, 1]

        acc = accuracy(classifier, X_test)
        prfs = precision_recall_fscore_support(y_true, y_pred, average='weighted')
        scores = scores + np.array([prfs[0], prfs[1], prfs[2], acc])

    scores = scores / folds
    cv_results = {'precision': scores[0], 'recall': scores[1],
                  'f1': scores[2], 'accuracy': scores[3]}
    return cv_results
```

I used KFold class from sklearn to do K-fold cross validation and used sklearn.metrics to calculate precision, recall, f score. However, Accuracy is the function from NLTK. I calculated those score for each loop and average them after.

Running the function

```
data['TOKEN'] = data['REVIEW_TEXT'].apply(preProcess)
data['REVIEW_VOCTOR'] = data[['TOKEN']].apply(toFeatureVector, axis=1)
selected_data = data[['REVIEW_VOCTOR', 'LABEL']]

train_data = selected_data.values
crossValidate(train_data, 10)
```

Result

```
{'accuracy': 0.89299999999999979,
 'f1': 0.88774792602602193,
 'precision': 0.92186068822221523,
```

```
'recall': 0.89299999999999979}
```

I refactored the template to use pandas. First of all, I preprocess the review text and then vectorize them. Then I called 10 fold cross validation. The results are as shown above.

4. (15 points) Now that you have the numbers for accuracy of your classifier, think of ways to improve this score. Things to consider:
- Improve the preprocessing. Which tokens might you want to throw out or preserve?
- What about punctuation? Do not forget normalisation and lemmatization - what aspects of this might be useful?
- Think about the features: what could you use other than unigram tokens from the review texts? It may be useful to look beyond single words to combinations of words or characters. Also the feature weighting scheme: what could you do other than using binary values?
- You could consider playing with the parameters of the SVM (cost parameter? per-class weighting?)

  Report what methods you tried and what the effect was on the classifier performance.

Preprocessing

```
stemmer = SnowballStemmer('english')
stopWords = set(stopwords.words('english'))
tokenizer = RegexpTokenizer(r'\w+')
lmtzr = WordNetLemmatizer()

def preProcess(text):
    tokens = tokenizer.tokenize(text)
    result = [lmtzr.lemmatize(t) for t in tokens if t not in stopWords]
    return result
```

As discussed in question 1, I filtered out stop words as well as when using RegexpTokenizer, it should remove punctuation. Doing this step improved my accuracy almost 10%.

The library WordNetLemmatizer does normalization, lemmatization and stemming.

Vectorization

```
def toFeatureVector(row):
    # Should return a dictionary containing features as keys, and weights as values

    tokens = row[0]

    feature_vector = {}
    for token in tokens:
        if token in feature_vector:
            feature_vector[token] += 1
```

```
        else:
            feature_vector[token] = 1

        if token in featureDict:
            featureDict[token] += 1
        else:
            featureDict[token] = 1

    for i in range(len(tokens)-1):
        token = (tokens[i] + ' ' + tokens[i+1])
        if token in feature_vector:
            feature_vector[token] += 1
        else:
            feature_vector[token] = 1

        if token in featureDict:
            featureDict[token] += 1
        else:
            featureDict[token] = 1

    return feature_vector
```

The second for in the function is to do bigram tokens. I used number of occurrences for the weights for both unigram and bigram. From my observation, this doesn't improve performance much.

SVM

```
def trainClassifier(trainData):
    pipeline =  Pipeline([('svc', LinearSVC(loss='hinge', max_iter=3000, C=1))])
    return SklearnClassifier(pipeline).train(trainData)
```

I changed the loss function to hinge and increase max_iter to 3000. This does not improve performance much.

Result

```
{'accuracy': 0.8877142857142859,
 'f1': 0.8822069897089136,
 'precision': 0.9170828108526644,
 'recall': 0.8877142857142859}
```

The performance improved significantly from the old plain preprocessing! The most significant factors for the improvement are mostly from preprocessing step.

5. (15 points) Now look beyond textual features of the review. The data set contains a number of other features for each review (rating, verified purchase, product category, product ID, product title, review title). How can the inclusion of these features improve

your classifier’s performance? Pick three of these metadata types to use as additional features and report how they improve the classifier performance.

Vectorization

```
def toFeatureVector(row):
    tokens = row[0]

    feature_vector = {}
    for token in tokens:
        if token in feature_vector:
            feature_vector[token] += 1
        else:
            feature_vector[token] = 1

        if token in featureDict:
            featureDict[token] += 1
        else:
            featureDict[token] = 1

    for i in range(len(tokens)-1):
        token = (tokens[i] + ' ' + tokens[i+1])
        if token in feature_vector:
            feature_vector[token] += 1
        else:
            feature_vector[token] = 1

        if token in featureDict:
            featureDict[token] += 1
        else:
            featureDict[token] = 1

    if(len(row)>1):
        feature_vector['RATING'] = row[1]
        feature_vector['VERIFIED_PURCHASE'] = row[2]
        feature_vector['PRODUCT_CATEGORY'] = row[3]

    return feature_vector

review_vector = data[['TOKEN', 'RATING', 'VERIFIED_PURCHASE',
'PRODUCT_CATEGORY']].apply(toFeatureVector, axis=1)
```

In this step, I added more features to the vector, ie, the features’ name as a key and features’ value as value.

Result

```
{'accuracy': 0.9264761904761904 5,
 'f1': 0.92873474269461354,
```

```
'precision': 0.94122184107524076,
'recall': 0.92647619047619045}
```

After adding more features, the result increased significantly!

We can conclude that having good features is very substantial to getting higher accuracy.

Implementing TF-IDF

According to this TF-IDF formula

$$w_{t,d} = (1 + \log \mathrm{tf}_{t,d}) \times \log_{10}(N / \mathrm{df}_t)$$

https://www.youtube.com/watch?v=PhunzHqhKoQ&t=125s

Some of the codes are modified to support TF-IDF.

```
def toFeatureVector(row):
    # Should return a dictionary containing features as keys, and weights as values

    tokens = row[0]

    feature_vector = {}
    for token in tokens:
        if token in feature_vector:
            feature_vector[token] += 1
        else:
            feature_vector[token] = 1

        # implement TFIDF
        if feature_vector[token] == 1:
            if token in featureDict:
                featureDict[token] += 1
            else:
                featureDict[token] = 1

    for i in range(len(tokens)-1):
        token = (tokens[i] + ' ' + tokens[i+1])
        if token in feature_vector:
            feature_vector[token] += 1
        else:
            feature_vector[token] = 1

        if feature_vector[token] == 1:
            if token in featureDict:
                featureDict[token] += 1
            else:
                featureDict[token] = 1
```

```
    return feature_vector
```

The variable featureDict collects how review the tokens appear in, not counting as previously.

```
review_count = data.shape[0]

def tfidf(row):
    result = {}

    featureVector = row[0]
    for token, count in featureVector.items():
        result[token] = (1 + math.log10(count))*math.log10(review_count/featureDict[token])


    if(len(row)>1):
        result['RATING'] = row[1]*2
        result['VERIFIED_PURCHASE'] = row[2]
        result['PRODUCT_CATEGORY'] = row[3]

    return result
```

Tfidf function is added to calculate tfidf score according to the formula given. Other features is add in this function as well.

The result improves by 2% after TF-IDF implementation.

```
{'accuracy': 0.94166666666666676,
 'f1': 0.94166719211353234,
 'precision': 0.941769709481315,
 'recall': 0.94166666666666676}
```

FIXING DIFFERENT ACCURACY

Before this, the performance in test dataset and cross validation are significantly different, 81% and 93% respectively. After removing function shuffle(dataset) in cross validation, the accuracy reduces to 81%, which makes more sense compared to the accuracy on test dataset.

Cross-validation performance

```
{'accuracy': 0.81101190476190477,
 'f1': 0.81088772667044129,
 'precision': 0.81212075505930503,
 'recall': 0.81101190476190477}
```

Testset performance

```
{'accuracy': 0.8173809523809524,
 'f1': 0.81725965974301473,
 'precision': 0.81777819872823443,
 'recall': 0.81738095238095243}
```

Part B: Data Exploration (40 points)

1. (5 points) When you are convinced your classifier works well (think about what are acceptable accuracy scores!), spend some time exploring the data. Are there any interesting correlations between the review class (fake or real) and product category? What about review rating? Is verified purchase a useful indicator for whether the review is genuine?

product category

```
data['LABEL_INT'] = data['LABEL'].map({'fake': 1, 'real': 0})
```

I gave fake label as 1 and real as 0 in order be easy to calculate statistics.

```
data[['LABEL_INT',
'PRODUCT_CATEGORY']].groupby('PRODUCT_CATEGORY').agg(['sum', 'count',
'mean'])
```

	LABEL_INT		
	sum	count	mean
PRODUCT_CATEGORY			
Apparel	350	700	0.5
Automotive	350	700	0.5
Baby	350	700	0.5
Beauty	350	700	0.5
Books	350	700	0.5
Camera	350	700	0.5
Electronics	350	700	0.5
Furniture	350	700	0.5
Grocery	350	700	0.5
Health & Personal Care	350	700	0.5

Each product category has the same number of fake and real classes.

review rating

```
data['LABEL_INT'].corr(data['RATING'])
```

Result -0.0097972205512207866

```
data[['LABEL_INT', 'RATING']].groupby('RATING').agg(['sum', 'count', 'mean'])
```

	LABEL_INT		
	sum	count	mean
RATING			
1	889	1757	0.505976
2	627	1192	0.526007
3	926	1868	0.495717
4	1999	3973	0.503146
5	6059	12210	0.496233

We could conclude that review class has no correlation with rating and is equally splitted among each rating.

verified purchase

```
data['LABEL_INT'].corr(data['VERIFIED_PURCHASE_INT'])
```

Result -0.56981624262119279

Verified purchase has negative correlation with the review class. This means that verified purchase tends to be real review.

2. (15 points) It is also interesting to consider whether there is anything intrinsically different about the ways in which the fake reviews are written when compared with the genuine ones. To examine this, you will now explore some of the linguistic and stylistic traits of the reviews and compare the two classes. Think about the following areas (and use the original raw data set to preserve the stylistic features):

- On average, how long are the reviews for each class? Does one class use more complex vocabulary than the other (consider word length, but also more complex measures of reading level e.g. the Flesch-Kincaid readability test)?

Length

```
data['REVIEW_COUNT'] = data['REVIEW_TEXT'].str.count(r'\w+')
data[['LABEL', 'REVIEW_COUNT']].groupby('LABEL').agg(['count', 'mean', 'std'])
```

	REVIEW_COUNT		
	count	mean	std
LABEL			
fake	10500	61.050476	60.870686
real	10500	81.653810	109.870801

The real review tends to be longer and more spread in length.

Flesch-Kincaid readability test

```
data['READABILITY'] = data['REVIEW_TEXT'].apply(textstat.flesch_reading_ease)
data[['LABEL', 'READABILITY']].groupby('LABEL').agg(['count', 'mean', 'std'])
```

	READABILITY		
	count	mean	std
LABEL			
fake	10500	79.759028	13.044638
real	10500	79.029707	13.187130

Surprisingly, Flesh-Kincaid readability test give the same score for real and fake review. This means the fake review made by taking readability test into consideration.

- Does one class contain more stopwords than the others? What about use of capslock and punctuation?

Stopwords

```
from nltk.corpus import stopwords
stopWords = set(stopwords.words('english'))

def count_stopwords(text):
   c = 0
   for word in text.split():
      if word in stopWords:
         c += 1
   return c

data['STOPWORDS_COUNT'] = data['REVIEW_TEXT'].apply(count_stopwords)
data[['LABEL', 'STOPWORDS_COUNT']].groupby('LABEL').agg(['count', 'mean', 'std'])

data['STOPWORDS_RATIO'] = data['STOPWORDS_COUNT'] / data['REVIEW_COUNT']
data[['LABEL', 'STOPWORDS_RATIO']].groupby('LABEL').agg(['count', 'mean', 'std'])
```

	STOPWORDS_COUNT		
	count	mean	std
LABEL			
fake	10500	24.696190	24.325351
real	10500	32.519048	43.813539

	STOPWORDS_RATIO		
	count	mean	std
LABEL			
fake	10500	0.401980	0.076370
real	10500	0.393652	0.077381

Real review tends to have more stopwords due to longer length. However, when divided by length, ratio for fake and real review are the same.

Punctuation

```
from string import punctuation

def count_punctuation(text):
    c = 0
    for word in text:
        if word in punctuation:
            c += 1
    return c
data['PUNCTUATION_COUNT'] = data['REVIEW_TEXT'].apply(count_punctuation)
data[['LABEL', 'PUNCTUATION_COUNT']].groupby('LABEL').agg(['count', 'mean', 'std'])

data['PUNCTUATION_RATIO'] = data['PUNCTUATION_COUNT'] /
data['REVIEW_COUNT']
data[['LABEL', 'PUNCTUATION_RATIO']].groupby('LABEL').agg(['count', 'mean', 'std'])
```

	PUNCTUATION_COUNT		
	count	mean	std
LABEL			
fake	10500	10.182571	15.482145
real	10500	15.571524	25.888301

	PUNCTUATION_RATIO		
	count	mean	std
LABEL			
fake	10500	0.157543	0.091244
real	10500	0.178093	0.144681

Real review tends to have more punctuation as both count and ratio indicate the same things.

Uppercase

```
def count_upper(text):
    return sum(1 for char in text if char.isupper())

data['UPPER_COUNT'] = data['REVIEW_TEXT'].apply(count_upper)
data[['LABEL', 'UPPER_COUNT']].groupby('LABEL').agg(['count', 'mean', 'std'])

data['UPPER_RATIO'] = data['UPPER_COUNT'] / data['REVIEW_COUNT']
data[['LABEL', 'UPPER_RATIO']].groupby('LABEL').agg(['count', 'mean', 'std'])
```

	UPPER_COUNT		
	count	mean	std
LABEL			
fake	10500	8.712667	24.175636
real	10500	12.099905	27.639396

	UPPER_RATIO		
	count	mean	std
LABEL			
fake	10500	0.136546	0.222843
real	10500	0.150311	0.262332

Real reviews tend to have more uppercases.

- Does the product name appear in the review? If so, does this happen more or less often in genuine and fake reviews? *Hint: You could play around with regular expressions to search for (variations of) the product name in a review.*

```
data['IS_NAME_IN_TEXT'] = data[['REVIEW_TEXT',
'PRODUCT_TITLE']].apply(name_in_text, axis=1)
data[['LABEL', 'IS_NAME_IN_TEXT']].groupby('LABEL').agg(['count', 'mean', 'std'])
```

	IS_NAME_IN_TEXT		
	count	mean	std
LABEL			
fake	10500	0.456000	0.498084
real	10500	0.448381	0.497352

I splitted the product titles by space since the product titles are quite long and consisted of more than 1 words. It may be very difficult to see the whole title of the product in the review. So, each splitted part of the title are searched on the review text. If it occurs in the review, return 1, 0 otherwise.

The result show the proportion of title appears on the review and the total reviews. It shows that real and fake reviews have roughly the same result.

- Are there any other surface features you can think of that may be interesting to compare?
  Write up your findings in your report.

3. (20 points) You will now look at the sentiment of the reviews in your data set. Build a sentiment classifier using the review rating as the sentiment gold standard. Treating this as a binary classification task, you can consider a 1-2 star review as negative, and a 4-5 star review as positive. In order to achieve a roughly balanced data set, you may want to remove some of the positive reviews. *Hint: Try retraining your deception classifier for this task - how does it perform? Think of which features may be better suited for detecting sentiment instead of deception.*

```
data['SENTIMENT_FROM_RATING'] = data['RATING'].map({5: 'positive', 4: 'positive', 3:
'neutral', 2: 'negative', 1: 'negative'})
selected_data2 = data[['REVIEW_VOCTOR', 'SENTIMENT_FROM_RATING']]
crossValidate(selected_data2.values, 3)
```

Result

```
{'accuracy': 0.9154761904761904,
 'f1': 0.88869568851928615,
 'precision': 0.90283824662183854,
 'recall': 0.91547619047619044}
```

I did almost the same as the previous real/fake classifier except changing the label to sentiment. The results are pretty great, giving 91% accuracy, which is higher than the real/fake classifier as shown.