# TP 1: Sentiment analysis in textual movie reviews

CAO Anh Quan

Natural Language and Speech Processing
AI-VIS

September 30, 2019

## Contents

# 1 Implementation of the classifier

## 1.1 Complete the count_words function. Delete the punctuation. Give the vocabulary size.

### 1.1.1 Code

```
def tokenize(s):
    s = s.lower().strip()
    s = re.sub(r"[^a-z]", " ", s) # remove punctuation
    ws = re.compile(r"\s+").split(s)
    return [w for w in ws if len(w) > 0]

def count_words(texts, stop_words=[]):
    """Vectorize text : return count of each word in the text snippets

    Parameters
    ----------
    texts: list of str
        The texts

    Returns
    -------
    vocabulary : dict
        A dictionary that points to an index in counts for each word.
    counts: ndarray, shape (n_samples, n_features)
        The counts of each word in each text.
        n_samples == number of documents.
        n_features == number of words in vocabulary.
    """

    words = set()
    for text in texts:
        ws = tokenize(text)
        for w in ws:
            if w not in stop_words:
                words.add(w)

    vocabulary = dict()
    for i, word in enumerate(words):
        vocabulary[word] = i
```

```
35
    n_features = len(vocabulary)
37  print("Number of words in vocabulary:", n_features)
    counts = np.zeros((len(texts), n_features))
39  for i, text in enumerate(texts):
        ws = tokenize(text)
41      for word in ws:
            if word in vocabulary:
43              word_index = vocabulary[word]
                counts[i, word_index] += 1
45  return vocabulary, counts
```

### 1.1.2 Discussion

There are many stop words such as "don't" and "couldn't" which contain punctuation. Hence, I deleted punctuation in both stop words and the tokens of the texts to make them consistent with each other. We had the following **vocabulary size:**

- Without removing stop words: 38,911 words
- Remove stop words: 38,395 words

## 1.2 Explain how positive and negative classes have been assigned to movie reviews

The movie reviews are assigned to positive or negative classes based on the rating system.

- **Five-star system:** Reviews with rating larger than three-and-a-half stars are considered positive. Reviews with a rating below or equal to two stars are considered negative.
- **Four-star system:** Reviews with rating larger than three stars are considered positive. Reviews with a rating below or equal to one-and-a-half are considered negative.
- **Letter grade system:** Reviews with a rating above or equal to B are considered as positive. Reviews with a rating below or equal to C- are considered as negative.

## 1.3 Complete the NB class

### 1.3.1 Code

```
class NB(BaseEstimator, ClassifierMixin):
2   def __init__(self):
        self.prior = []
4       self.likelihood = []
        self.classes = []
6
```

```
    def fit(self, X, y):
        self.classes = np.unique(y)
        prior = [0] * len(self.classes)
        N = len(y)
        likelihood = np.zeros((X.shape[1], len(self.classes)))
        for i, c in enumerate(self.classes):
            # compute prior
            prior[i] = np.sum(y == c) / N

            # count number of occurences of token in each class
            X_class = np.sum(X[y == c], axis=0)
            likelihood[:, i] = X_class

        # compute conditional probability
        likelihood = likelihood + 1
        likelihood = likelihood / np.sum(likelihood, axis=0).reshape(1, -1)

        self.prior = prior
        self.likelihood = likelihood

        return self

    def predict(self, X):
        scores = X @ np.log(self.likelihood) + np.log(self.prior).reshape(1, -1)
        y_pred = np.argmax(scores, axis=1)
        return [self.classes[i] for i in y_pred]

    def score(self, X, y):
        return np.mean(self.predict(X) == y)
```

### 1.3.2 Discussion

First, I implemented the Naive Bayes algorithm following the pseudo-code, but it was very slow because it used for loop. Thus, I modified and reimplemented the algorithm using matrix operations. My modified implementation is much faster and shorter than the original implementation of the pseudo-code.

## 1.4 Evaluate the performance of your classifier in cross-validation 5-folds

### 1.4.1 Code

```
def accuracy(y, y_pred):
    return np.mean(y == y_pred)

def cross_validation(clf, X, y, n_folds=5):
    interval = len(y) // n_folds
    scores = []
```

```
    for i in range(n_folds):
        start = int(i * interval)
        end = int((i + 1) * interval)

        X_test = X[start:end]
        y_test = y[start:end]
        X_train = np.concatenate([X[:start], X[end:]])
        y_train = np.concatenate([y[:start], y[end:]])

        clf.fit(X_train, y_train)
        y_pred = clf.predict(X_test)

        score = accuracy(y_pred, y_test)
        scores.append(score)

    return scores

nb = NB()
scores = cross_validation(nb, X, y)
print(scores, np.mean(scores))
# [0.8275, 0.7825, 0.8025, 0.8375, 0.8175] 0.8135


mnb = MultinomialNB()
scores = cross_validation(mnb, X, y)
print(scores, np.mean(scores))
# [0.8275, 0.7825, 0.8025, 0.8375, 0.8175] 0.8135
```

### 1.4.2 Discussion

The accuracy without removing stop words:

- My implementation of Naive Bayes:
  - In each fold: [0.8275, 0.7825, 0.8025, 0.8375, 0.8175]
  - On average of all folds: 0.8135
- MultinomialNB from sklearn:
  - In each fold: [0.8275, 0.7825, 0.8025, 0.8375, 0.8175]
  - On average of all folds: 0.8135

My implementation of Naive Bayes has the same accuracy in each fold and on average as the MultinomialNB.

## 1.5 Remove stop words. Are performances improved?

### 1.5.1 Without removing stop words

```
# without removing stop words
texts, y = load_data()
texts, y = shuffle_data(np.array(texts), y)
vocabulary, X = count_words(texts, stop_words=[])
print("X shape", X.shape)
nb = NB()
scores = cross_validation(nb, X_removed, y)
print("My NB", scores, np.mean(scores))

mnb = MultinomialNB()
scores = cross_validation(mnb, X_removed, y)
print("MultinomialNB", scores, np.mean(scores))
# My NB [0.8275, 0.7825, 0.8025, 0.8375, 0.8175] 0.8135
# MultinomialNB [0.8275, 0.7825, 0.8025, 0.8375, 0.8175] 0.8135
```

### 1.5.2 Remove stop words

```
# remove stop words
texts, y = load_data()
texts, y = shuffle_data(np.array(texts), y)
stop_words = read_stop_words(remove_punc=True)
vocabulary_removed, X_removed = count_words(texts, stop_words=stop_words)
print("X shape", X_removed.shape)

nb = NB()
scores = cross_validation(nb, X_removed, y)
print("My NB", scores, np.mean(scores))

mnb = MultinomialNB()
scores = cross_validation(mnb, X_removed, y)
print("MultinomialNB", scores, np.mean(scores))
# My NB [0.8, 0.7975, 0.795, 0.8125, 0.8175] 0.8045
# MultinomialNB [0.8, 0.7975, 0.795, 0.8125, 0.8175] 0.8045
```

### 1.5.3 Discussion

If we did not remove stop words, both classifiers had the following performance:

- In each fold: [0.8275, 0.7825, 0.8025, 0.8375, 0.8175]
- On average of all folds: 0.8135

If we removed stop words, both classifiers had the following performance:

- In each fold: [0.8, 0.7975, 0.795, 0.8125, 0.8175]
- On average of all folds: 0.8045

The performances decreased when we removed the stop words. I think the reason is that the stop words contain reasonable information in sentiment analysis problem. When we removed the stop words, we remove some important information which changes the meaning of the sentences. For example, the sentence "I do not feel happy" has a negative meaning. After removing the stop words, the result is ["feel", "happy"]. The overall meaning is positive, which is not correct in this case.

# 2 Scikit-learn use

## 2.1 Compare your implementation with scikitlearn. We will use CountVectorizer and Pipeline. You will experiment by allowing words and bigrams or by working on substrings of characters

### 2.1.1 Code

```
def evaluate_pipeline(clf_name, clf, texts, y, analyzer):
    # Create pipeline
    mnb_pipeline = Pipeline([
        ('vect', CountVectorizer(analyzer=analyzer,
                                 ngram_range=(1, 2))), # allow bigrams
        ('clf', clf),
    ])
    scores = cross_validation(mnb_pipeline, texts, y)
    print(clf_name + ":", scores,"- mean accuracy:", np.mean(scores))


def evaluate(analyzer, texts, y):

    nb = NB()
    evaluate_pipeline("My NB", nb, texts, y, analyzer)

    mnb = MultinomialNB()
    evaluate_pipeline("Multinomial NB", mnb, texts, y, analyzer)

    lsvc = LinearSVC(max_iter=500)
    evaluate_pipeline("Linear SVC", lsvc, texts, y, analyzer)

    reg = LogisticRegression(solver='lbfgs')
    evaluate_pipeline("Logistic Regression", reg, texts, y, analyzer)

    rf = RandomForestClassifier(n_estimators=20)
    evaluate_pipeline("Random forest", rf, texts, y, analyzer)

## Use only words, no bigram
evaluate('word', texts, y, ngram_range=(1,1))
# Result
# My NB: [0.785, 0.79, 0.825, 0.84, 0.8225] - mean accuracy: 0.8125
# Multinomial NB: [0.785, 0.79, 0.825, 0.84, 0.8225] - mean accuracy: 0.8125
```

```
# Linear SVC: [0.8375, 0.8175, 0.83, 0.835, 0.82] - mean accuracy: 0.828
# Logistic Regression: [0.8375, 0.825, 0.8375, 0.84, 0.8325] - mean accuracy:
    0.8344999999999999
# Random forest: [0.6925, 0.7325, 0.7375, 0.7175, 0.7375] - mean accuracy:
    0.7234999999999999


## Words and bigram
evaluate('word', texts, y)
# Result
# My NB: [0.855, 0.81, 0.8275, 0.825, 0.85] - mean accuracy:
    0.8334999999999999
# Multinomial NB: [0.855, 0.81, 0.8275, 0.825, 0.85] - mean accuracy:
    0.8334999999999999
# Linear SVC: [0.8525, 0.8225, 0.855, 0.845, 0.8775] - mean accuracy:
    0.8504999999999999
# Logistic Regression: [0.85, 0.825, 0.8375, 0.8525, 0.8725] - mean accuracy:
    0.8474999999999999
# Random forest: [0.7125, 0.6675, 0.7325, 0.73, 0.7675] - mean accuracy: 0.722

## Characters and bigram
evaluate('char', texts, y)
# Result
# My NB: [0.68, 0.6375, 0.6875, 0.6525, 0.7075] - mean accuracy:
    0.6729999999999999
# Multinomial NB: [0.68, 0.6375, 0.6875, 0.6525, 0.7075] - mean accuracy:
    0.6729999999999999
# Linear SVC: [0.5775, 0.51, 0.6675, 0.5925, 0.7175] - mean accuracy:
    0.6130000000000001
# Logistic Regression: [0.73, 0.6725, 0.7425, 0.7175, 0.77] - mean accuracy:
    0.7264999999999999
# Random forest: [0.65, 0.6775, 0.605, 0.655, 0.6175] - mean accuracy:
    0.6410000000000001
```

### 2.1.2 Discussion

My implementation of Naive Bayes had the same performance as the MultinomialNB. The performances of all classifiers were improved significantly by using bigram. It makes sense because with bigram, we can capture pairs of words that have a different meaning when they go together such as "not happy" and "pretty bad".

When we considered only characters and bigram of characters, the performances were worse because we looked at the distribution of characters instead of looking at the distribution of words. Hence, it is much harder to determine the meaning of the sentences when we only look at the characters of the sentence. For example, if we look at the distribution of a sentence, and we notice many words: "good", "happy", "pretty", we can determine that the sentence is positive. However, if we only look at the distribution of the characters, we cannot determine if the sentence is positive or negative.

## 2.2 Test another classification method scikitlearn

### 2.2.1 Code

```
def evaluate(analyzer, texts, y):

    nb = NB()
    evaluate_pipeline("My NB", nb, texts, y, analyzer)

    mnb = MultinomialNB()
    evaluate_pipeline("Multinomial NB", mnb, texts, y, analyzer)

    lsvc = LinearSVC(max_iter=500)
    evaluate_pipeline("Linear SVC", lsvc, texts, y, analyzer)

    reg = LogisticRegression(solver='lbfgs')
    evaluate_pipeline("Logistic Regression", reg, texts, y, analyzer)

    rf = RandomForestClassifier(n_estimators=20)
    evaluate_pipeline("Random forest", rf, texts, y, analyzer)


## Use only words, no bigram
evaluate('word', texts, y, ngram_range=(1,1))
# Result
# My NB: [0.785, 0.79, 0.825, 0.84, 0.8225] - mean accuracy: 0.8125
# Multinomial NB: [0.785, 0.79, 0.825, 0.84, 0.8225] - mean accuracy: 0.8125
# Linear SVC: [0.8375, 0.8175, 0.83, 0.835, 0.82] - mean accuracy: 0.828
# Logistic Regression: [0.8375, 0.825, 0.8375, 0.84, 0.8325] - mean accuracy:
    0.8344999999999999
# Random forest: [0.6925, 0.7325, 0.7375, 0.7175, 0.7375] - mean accuracy:
    0.7234999999999999


## Words and bigram
evaluate('word', texts, y)
# Result
# My NB: [0.855, 0.81, 0.8275, 0.825, 0.85] - mean accuracy:
    0.8334999999999999
# Multinomial NB: [0.855, 0.81, 0.8275, 0.825, 0.85] - mean accuracy:
    0.8334999999999999
# Linear SVC: [0.8525, 0.8225, 0.855, 0.845, 0.8775] - mean accuracy:
    0.8504999999999999
# Logistic Regression: [0.85, 0.825, 0.8375, 0.8525, 0.8725] - mean accuracy:
    0.8474999999999999
# Random forest: [0.7125, 0.6675, 0.7325, 0.73, 0.7675] - mean accuracy: 0.722

## Characters and bigram
evaluate('char', texts, y)
# Result
# My NB: [0.68, 0.6375, 0.6875, 0.6525, 0.7075] - mean accuracy:
    0.6729999999999999
# Multinomial NB: [0.68, 0.6375, 0.6875, 0.6525, 0.7075] - mean accuracy:
    0.6729999999999999
```

```
# Linear SVC: [0.5775, 0.51, 0.6675, 0.5925, 0.7175] - mean accuracy:
    0.6130000000000001
# Logistic Regression: [0.73, 0.6725, 0.7425, 0.7175, 0.77] - mean accuracy:
    0.7264999999999999
# Random forest: [0.65, 0.6775, 0.605, 0.655, 0.6175] - mean accuracy:
    0.6410000000000001
```

### 2.2.2 Discussion

We tests four classification algorithms: Naive Bayes, Linear SVC, Logistic Regression, and Random Forest. In our problem, the Linear SVC has the best performance. The second one is Logistic Regression and following by Naive Bayes. The worst algorithm is the Random Forest. Thus, we can improve our classifier by using the Linear SVC algorithm, which had much better performance than the Naive Bayes.

## 2.3 Use NLTK library in order to process a stemming. You will use the class SnowballStemmer

### 2.3.1 Discussion

In this question, I first tokenized each sentence. Then, I stemmed the tokens of each sentence using SnowballStemmer. After that, for each sentence, I rejoined the tokens to create a new sentence so that I could fit the pipeline to the stemmed texts.

By stemming the words, we increased the performance of all classifiers slightly. Stemming has both good and bad effects for sentiment analysis problem. On the good side, it reduces noise in our data because it converts inflectional forms and related forms to the common form. On the other hand, the disadvantage of it is that it could change the meaning of a word. For example, "universal" and "university" have different meaning but are both stemmed to "univers". Furthermore, we cannot apply syntactic and semantic processing such as pos-tagging on the sentence after stemming. In our problem, it seems that the good effect of stemming is higher than the bad effect.

### 2.3.2 Code

```
from nltk import SnowballStemmer

stemmer = SnowballStemmer(language='english')

texts, y = load_data()
texts, y = shuffle_data(np.array(texts), y)

new_texts = []
for i, text in enumerate(texts):
    tokens = tokenize(text)
    token_str = " ".join([stemmer.stem(token) for token in tokens if len(token
    ) > 0])
```

```
    new_texts.append(token_str)

# Evaluate stemmed texts
evaluate('word', new_texts, y)
# Result
# My NB: [0.81, 0.825, 0.8675, 0.8225, 0.825] - mean accuracy:
    0.8300000000000001
# Multinomial NB: [0.81, 0.825, 0.8675, 0.8225, 0.825] - mean accuracy:
    0.8300000000000001
# Linear SVC: [0.86, 0.84, 0.8625, 0.8575, 0.835] - mean accuracy: 0.851
# Logistic Regression: [0.8625, 0.8425, 0.8475, 0.875, 0.8525] - mean accuracy
    : 0.8560000000000001
# Random forest: [0.725, 0.705, 0.6825, 0.7375, 0.705] - mean accuracy: 0.711



# Evaluate original texts
evaluate('word', texts, y)
# Result
# My NB: [0.81, 0.8375, 0.865, 0.8275, 0.805] - mean accuracy:
    0.8290000000000001
# Multinomial NB: [0.81, 0.8375, 0.865, 0.8275, 0.805] - mean accuracy:
    0.8290000000000001
# Linear SVC: [0.855, 0.84, 0.855, 0.8525, 0.845] - mean accuracy:
    0.8494999999999999

# Logistic Regression: [0.8575, 0.83, 0.855, 0.8575, 0.8525] - mean accuracy:
    0.8504999999999999
# Random forest: [0.7075, 0.715, 0.655, 0.7425, 0.6825] - mean accuracy:
    0.7005
```

## 2.4 Filter words by grammatical category (POS : Part Of Speech) and keep only nouns, verbs, adverbs and adjectives for classification

### 2.4.1 Discussion

In this question, similar to the previous one, I first tokenized the texts and then applied pos-tagging on the resulting tokens. After that, I kept only tokens that are nouns (NN, NNS, NNP, NNPS), verbs (VB, VBD, VBG, VBN, VBP, VBZ), adverbs (RB, RBR, RBS) and adjectives (JJ, JJR, JJS). For each sentence, I rejoined its tokens to create a new sentence. Finally, I fit the pipeline on the list of new sentences.

This process increased the performance of the Naive Bayes classifier slightly, but it also decreased the performances of other classifiers: Linear SVC, Random Forest, and Logistic Regression slightly. It seems that the removed words contain some important information. In my opinion, we should use the pos-tag at the same time with the words as features for prediction instead of deleting the words.

### 2.4.2 Code

```
texts, y = load_data()
texts, y = shuffle_data(np.array(texts), y)

keeps = ["NN", "NNS", 'NNP', 'NNPS', 'VB', 'VBD', 'VBG', 'VBN', 'VBP', 'VBZ',
    'RB', 'RBR', 'RBS', 'JJ', 'JJR', 'JJS']

new_texts = []
for i, text in enumerate(texts):
    tokens = tokenize(text)
    token_with_tags = nltk.pos_tag(tokens)
    filtered_token_with_tags = filter(lambda x: x[1] in keeps, token_with_tags
    ) # keep only noun, adverbs, verb, adj
    filtered_tokens = map(lambda x: x[0], filtered_token_with_tags)
    new_texts.append(" ".join(filtered_tokens))

# pos_tag - keep only noun, adverbs, verb, adj
evaluate('word', new_texts, y)
# Result
# My NB: [0.845, 0.775, 0.785, 0.7775, 0.82] - mean accuracy:
    0.8005000000000001
# Multinomial NB: [0.845, 0.775, 0.785, 0.7775, 0.82] - mean accuracy:
    0.8005000000000001
# Linear SVC: [0.825, 0.83, 0.83, 0.825, 0.845] - mean accuracy:
    0.8309999999999998
# Logistic Regression: [0.835, 0.835, 0.8425, 0.81, 0.86] - mean accuracy:
    0.8365
# Random forest: [0.7475, 0.74, 0.74, 0.715, 0.7025] - mean accuracy: 0.729


# Without pos tag
evaluate('word', texts, y)
# Result
# My NB: [0.8425, 0.7675, 0.7925, 0.7775, 0.82] - mean accuracy:
    0.7999999999999999
# Multinomial NB: [0.8425, 0.7675, 0.7925, 0.7775, 0.82] - mean accuracy:
    0.7999999999999999

# Linear SVC: [0.825, 0.8325, 0.8475, 0.82, 0.8525] - mean accuracy: 0.8355
# Logistic Regression: [0.835, 0.8325, 0.8475, 0.8175, 0.86] - mean accuracy:
    0.8385
# Random forest: [0.725, 0.7325, 0.73, 0.7275, 0.735] - mean accuracy: 0.73
```

# 3 Conclusion

We observed a significant improvement on the speed of the algorithm when we implemented the algorithm using matrix operations instead of the for loop.

In this lab, we tackled a sentiment analysis problem using bag of words approach. Then, we apply many techniques to see whether we can improve the performance of our classifier:

- Remove stop words.
- Use bigram.
- Use character instead of word.
- Try different classification algorithm.
- Stemming
- POS tagging - keep only nouns, verbs, adverbs and adjectives.

The most effective technique is the bigram which increases the performance of all classifiers significantly because we could capture pairs of words that have a different meaning when they go together such as "not happy" and "pretty bad". The second most effective technique is to try different classification algorithm as we can switch from Naive Bayes to LinearSVC algorithm, which had much better performance. The third one is stemming, which increase the performance of the classifier slightly as we discussed in section 2.3. The pos-tagging improved performance of Naive Bayes but decreased the performances of other classifiers. The two technique remove stop words and use characters instead of words did not work in our case. Therefore, we need to test different techniques to select the best combination of techniques for our problem.