

# text\_sentiment\_wordvec

November 26, 2021

## 1 Classification de phrases avec word2vec

Le texte est toujours délicat à traiter. La langue est importante et plus le vocabulaire est étendu, plus il faut de données. Le problème qui suit est classique, on cherche à catégoriser des phrases en sentiment positif ou négatif. Ce pourrait être aussi classer des spams. Le problème le plus simple : une phrase, un label.

```
[1]: from jupyterhelper import add_notebook_menu
      add_notebook_menu()
```

```
[1]: <IPython.core.display.HTML object>
```

```
[2]: from jupyterhelper import add_notebook_menu
      add_notebook_menu()
```

```
[2]: <IPython.core.display.HTML object>
```

```
[3]: %matplotlib inline
```

### 1.1 Les données

Elles proviennent de [Sentiment Labelled Sentences Data Set](#).

```
[4]: from papierstat.datasets import load_sentiment_dataset
      df = load_sentiment_dataset()
      df.head()
```

```
[4]:
```

	sentence	sentiment	\
0	So there is no way for me to plug it in here i...	0	
1	Good case, Excellent value.	1	
2	Great for the jawbone.	1	
3	Tied to charger for conversations lasting more...	0	
4	The mic is great.	1	

	source
0	amazon_cells_labelled
1	amazon_cells_labelled
2	amazon_cells_labelled
3	amazon_cells_labelled
4	amazon_cells_labelled

```
[5]: df.groupby(['source', 'sentiment']).count()
```

```
[5]:
```

	source	sentiment	sentence
	amazon_cells_labelled	0	500
		1	500
	imdb_labelled	0	500
		1	500
	yelp_labelled	0	500
		1	500

On découpe en train and test.

```
[6]: from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(df[["sentence"]], df['sentiment'])
```

## 1.2 L'approche classique

TF-IDF est une approche très répandue lorsqu'il s'agit de convertir des phrases en features.

```
[7]: from sklearn.feature_extraction.text import TfidfTransformer, CountVectorizer
from sklearn.pipeline import make_pipeline
pipe = make_pipeline(CountVectorizer(), TfidfTransformer())
pipe.fit(X_train['sentence'])
feat_train = pipe.transform(X_train['sentence'])
feat_train.shape
```

```
[7]: (2250, 4324)
```

```
[8]: feat_train.min(), feat_train.max()
```

```
[8]: (0.0, 1.0)
```

```
[9]: feat_test = pipe.transform(X_test['sentence'])
feat_test.shape
```

```
[9]: (750, 4324)
```

```
[10]: from sklearn.ensemble import RandomForestClassifier
clf = RandomForestClassifier(n_estimators=50)
clf.fit(feat_train, y_train)
```

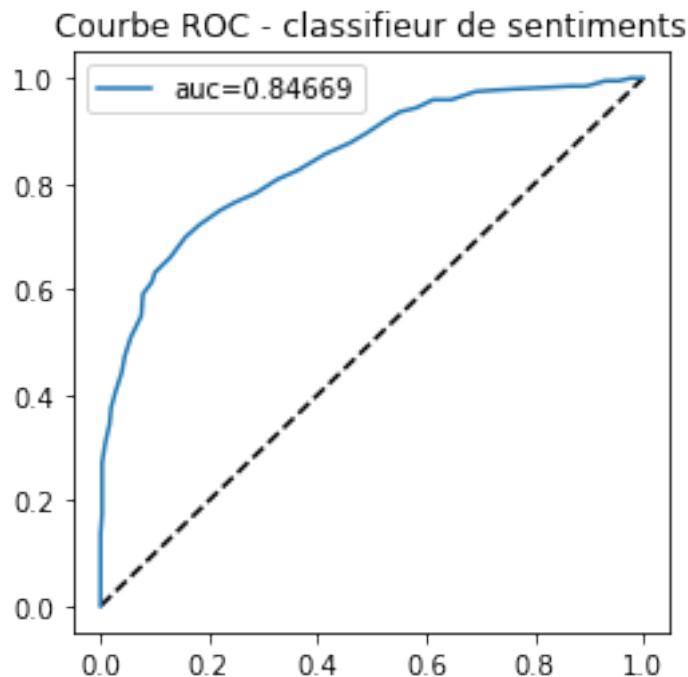
```
[10]: RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
criterion='gini', max_depth=None, max_features='auto',
max_leaf_nodes=None, max_samples=None,
min_impurity_decrease=0.0, min_impurity_split=None,
min_samples_leaf=1, min_samples_split=2,
min_weight_fraction_leaf=0.0, n_estimators=50,
n_jobs=None, oob_score=False, random_state=None,
verbose=0, warm_start=False)
```

```
[11]: clf.score(feat_test, y_test)
```

```
[11]: 0.768
```

```
[12]: from sklearn.metrics import roc_auc_score, roc_curve, auc
score = clf.predict_proba(feet_test)
fpr, tpr, th = roc_curve(y_test, score[:, 1])
```

```
[13]: import matplotlib.pyplot as plt
fig, ax = plt.subplots(1, 1, figsize=(4,4))
ax.plot([0, 1], [0, 1], 'k--')
aucf = auc(fpr, tpr)
ax.plot(fpr, tpr, label='auc=%1.5f' % aucf)
ax.set_title('Courbe ROC - classifieur de sentiments')
ax.legend();
```



### 1.3 Autres modèles que la RandomForest

La forêt aléatoire n'est pas nécessairement le meilleur modèle, comme l'affirme [#25 - Choix des classificateurs dans Classification de phrases avec word2vect](#). La transformation du texte en feature génère beaucoup de variables et un arbre de décision n'exploite quasiment que le fait qu'elles soient non nulles. Un arbre de décision consiste à prendre des décisions sur des seuils puis retourne une constante tirée d'une feuille de l'arbre. Un modèle linéaire ferait tout aussi bien l'affaire avec en plus la possibilité de tenir compte de la valeur de la variable.

L'autre aspect à prendre en compte est la profondeur de l'arbre. Par défaut, elle est de 10, soit  $2^{10} = 1024$  décisions de seuils, soit au mieux  $2^{10}$  variables ce qui est loin du nombre de variables total.

```
[14]: feat_test.shape
```

```
[14]: (750, 4324)
```

Essayons d'abord avec un modèle linéaire.

```
[15]: from sklearn.linear_model import LogisticRegression
lr = LogisticRegression()
lr.fit(feats_train, y_train)
lr.score(feats_test, y_test)
```

[15]: 0.8013333333333333

```
[16]: from sklearn.naive_bayes import MultinomialNB
mnb = MultinomialNB()
mnb.fit(feats_train, y_train)
mnb.score(feats_test, y_test)
```

[16]: 0.8106666666666666

On vérifie ensuite qu'en augmentant la profondeur de l'arbre, la forêt aléatoire est plus performante.

```
[17]: from sklearn.ensemble import RandomForestClassifier
clf20 = RandomForestClassifier(n_estimators=50, max_depth=20)
clf20.fit(feats_train, y_train)
clf20.score(feats_test, y_test)
```

[17]: 0.7466666666666667

```
[18]: from sklearn.ensemble import RandomForestClassifier
clf40 = RandomForestClassifier(n_estimators=50, max_depth=40)
clf40.fit(feats_train, y_train)
clf40.score(feats_test, y_test)
```

[18]: 0.7693333333333333

```
[19]: from sklearn.ensemble import RandomForestClassifier
clf60 = RandomForestClassifier(n_estimators=50, max_depth=60)
clf60.fit(feats_train, y_train)
clf60.score(feats_test, y_test)
```

[19]: 0.7693333333333333

Ainsi qu'en augmentant le nombre d'arbre puisque l'algorithme balayera plus de variables.

```
[20]: clf40 = RandomForestClassifier(n_estimators=200, max_depth=40)
clf40.fit(feats_train, y_train)
clf40.score(feats_test, y_test)
```

[20]: 0.7693333333333333

Un seul arbre de décision produira des résultats plutôt pauvres.

```
[21]: from sklearn.tree import DecisionTreeClassifier
dt = DecisionTreeClassifier()
dt.fit(feats_train, y_train)
dt.score(feats_test, y_test)
```

[21]: 0.7053333333333334

Enfin, un modèle de gradient boosting devrait sans doute dépasser les forêts aléatoires puisque les arbres ne sont plus appris indépendamment les uns des autres.

```
[22]: from sklearn.ensemble import GradientBoostingClassifier
gbc40 = GradientBoostingClassifier(n_estimators=200, max_depth=40)
gbc40.fit(feats_train, y_train)
gbc40.score(feats_test, y_test)
```

[22]: 0.772

On garde la régression logistique pour la suite.

## 1.4 Les n-grammes

L'approche présentée ci-dessus ne tient pas compte de l'ordre des mots. Chaque phrase est convertie en un sac de mots (ou *bag of words*). Il est néanmoins possible de tenir compte de séquence plus ou moins longue.

```
[23]: # s'il faut télécharger des données
if False:
    import nltk
    nltk.download('punkt')
```

```
[24]: from nltk.util import ngrams
from nltk.tokenize import word_tokenize
generated_ngrams = ngrams(word_tokenize(X_train.iloc[0,0]), 3, pad_left=True,
↳pad_right=True)
list(generated_ngrams)[:7]
```

```
[24]: [(None, None, 'Their'),
      (None, 'Their', 'chow'),
      ('Their', 'chow', 'mein'),
      ('chow', 'mein', 'is'),
      ('mein', 'is', 'so'),
      ('is', 'so', 'good'),
      ('so', 'good', '!')]
```

scikit-learn permet d'essayer cette idée simplement.

```
[25]: pipe2 = make_pipeline(CountVectorizer(ngram_range=(1, 2)),
                          TfidfTransformer())
pipe2.fit(X_train['sentence'])
feats_train2 = pipe2.transform(X_train['sentence'])
feats_train2.shape
```

[25]: (2250, 20067)

Il y a plus de colonnes, on vérifie malgré tout que les features ressemblent à des couples de mots.

```
[26]: c1 = pipe2.steps[0]
c1[1].get_feature_names()[:10]
```

```
[26]: ['00',
      '10',
      '10 10',
      '10 feet',
      '10 for',
      '10 grade',
      '10 minutes',
      '10 on',
```

```
'10 out',  
'10 plus']
```

C'est le cas.

```
[27]: feat_test2 = pipe2.transform(X_test['sentence'])
```

```
[28]: from sklearn.linear_model import LogisticRegression  
clf2 = LogisticRegression()  
clf2.fit(feet_train2, y_train)
```

```
[28]: LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,  
intercept_scaling=1, l1_ratio=None, max_iter=100,  
multi_class='auto', n_jobs=None, penalty='l2',  
random_state=None, solver='lbfgs', tol=0.0001, verbose=0,  
warm_start=False)
```

```
[29]: clf2.score(feet_test2, y_test)
```

```
[29]: 0.8
```

Cela n'améliore pas de façon significative. Il faudrait faire une cross-validation pour s'en assurer.

## 1.5 Réduire les dimensions avec une ACP

C'est un moyen fréquemment utilisé pour réduire les dimensions. On choisit le modèle [TruncatedSVD](#) plutôt que l'[ACP](#) dont l'implémentation ne supporte pas les features sparses.

```
[30]: from sklearn.decomposition import TruncatedSVD  
pipe_svd = make_pipeline(CountVectorizer(), TruncatedSVD(n_components=300))  
pipe_svd.fit(X_train['sentence'])  
feat_train_svd = pipe_svd.transform(X_train['sentence'])  
feat_train_svd.shape
```

```
[30]: (2250, 300)
```

```
[31]: clf_svd = RandomForestClassifier(n_estimators=50)  
clf_svd.fit(feet_train_svd, y_train)
```

```
[31]: 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=50,  
n_jobs=None, oob_score=False, random_state=None,  
verbose=0, warm_start=False)
```

```
[32]: feat_test_svd = pipe_svd.transform(X_test['sentence'])  
clf_svd.score(feet_test_svd, y_test)
```

```
[32]: 0.684
```

```
[33]: lr_svd = LogisticRegression()  
lr_svd.fit(feet_train_svd, y_train)
```

```
lr_svd.score(feats_test_svd, y_test)
```

[33]: 0.7533333333333333

Et si on repart de TF-IDF :

```
[34]: pipe_svd_tfidf = make_pipeline(CountVectorizer(),
                                   TfidfTransformer(),
                                   TruncatedSVD(n_components=300))
pipe_svd_tfidf.fit(X_train['sentence'])
feats_train_svd_tfidf = pipe_svd_tfidf.transform(X_train['sentence'])

clf_svd_tfidf = LogisticRegression()
clf_svd_tfidf.fit(feats_train_svd_tfidf, y_train)

feats_test_svd_tfidf = pipe_svd_tfidf.transform(X_test['sentence'])
clf_svd_tfidf.score(feats_test_svd_tfidf, y_test)
```

[34]: 0.7786666666666666

C'est mieux mais cela reste moins bien que le *tf-idf* sans réduction de dimensions. Cela veut dire qu'il faut garder plus de dimensions.

## 1.6 word2vec

*word2vec* est une sorte d'ACP non linéaire en ce sens qu'il réduit les dimensions. Il faut lire [Analyse en composantes principales \(ACP\) et Auto Encoders](#) pour comprendre le lien entre ACP, ACP non linéaire, réseaux de neurones diabolos et compression. *word2vec* est plus d'une ACP non linéaire car il prend en compte le contexte mais ne s'en éloigne pas tant que ça.

```
[35]: from gensim.utils import tokenize
sentence = [list(tokenize(s, deacc=True, lower=True)) for s in X_train['sentence']]
sentence[0]
```

[35]: ['their', 'chow', 'mein', 'is', 'so', 'good']

Les paramètres d'apprentissage du modèle *Word2Vec* ne sont pas toujours décrit de façon explicite.

```
[36]: from gensim.models import word2vec
model = word2vec.Word2Vec(sentence, size=300, window=20,
                          min_count=2, workers=1, iter=100)
model.corpus_count
```

[36]: 2250

```
[37]: vocab = model.wv.vocab
list(vocab)[:5]
```

[37]: ['their', 'chow', 'is', 'so', 'good']

```
[38]: model.save('trained_word2vec.bin')
```

Les dix premières coordonnées du vecteur associé au mot *after*.

```
[39]: model.wv['after'].shape, model.wv['after'][:10]
```

```
[39]: ((300,),
       array([-0.35271704, -0.7440025 , -0.71504265,  0.5013653 , -0.8039807 ,
              -1.1835612 , -0.24699177,  0.03716084,  0.04742139, -0.31049225],
       dtype=float32))
```

Lorsque le mot est inconnu :

```
[40]: try:
       model.wv['rrrrrrrr']
       except KeyError as e:
           print(e)
```

```
"word 'rrrrrrrr' not in vocabulary"
```

Pour chaque phrase, on fait la somme des vecteurs associés aux mots qui la composent ou pas si le mot n'est pas dans le vocabulaire. Il y a probablement des fonctions déjà prêtes à l'emploi mais la documentation de [gensim](#) n'était pas assez explicite et lire l'article [Efficient Estimation of Word Representations in Vector Space](#) puis celui-ci [Distributed Representations of Words and Phrases and their Compositionality](#).

```
[41]: import numpy

def get_vect(word, model):
    try:
        return model.wv[word]
    except KeyError:
        return numpy.zeros((model.vector_size,))

def sum_vectors(phrase, model):
    return sum(get_vect(w, model) for w in phrase)

def word2vec_features(X, model):
    feats = numpy.vstack([sum_vectors(p, model) for p in X])
    return feats

wv_train_feat = word2vec_features(X_train["sentence"], model)
wv_train_feat.shape
```

```
[41]: (2250, 300)
```

```
[42]: clfwv = LogisticRegression(solver='lbfgs', max_iter=1000)
       clfwv.fit(wv_train_feat, y_train)
```

```
[42]: LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                          intercept_scaling=1, l1_ratio=None, max_iter=1000,
                          multi_class='auto', n_jobs=None, penalty='l2',
                          random_state=None, solver='lbfgs', tol=0.0001, verbose=0,
                          warm_start=False)
```

```
[43]: wv_test_feat = word2vec_features(X_test["sentence"], model)
```

```
[44]: clfwv.score(wv_test_feat, y_test)
```

```
[44]: 0.5653333333333334
```



La performance est nettement moindre et notamment moindre que la performance obtenue avec l'ACP. Il faudrait sans doute jouer avec les hyperparamètres de l'apprentissage ou réutiliser un modèle appris sur un corpus similaire aux données initiales mais nettement plus grand. On peut constater que la fonction de similarités ne retourne pas des résultats très intéressants.

```
[45]: words = list(sorted(model.wv.vocab))
      words[:10]
```

```
[45]: ['a',
      'ability',
      'able',
      'about',
      'above',
      'absolutely',
      'accept',
      'accidentally',
      'accused',
      'achievement']
```

```
[46]: subset = ['after', 'before', words[3], words[4], words[5]]
      rows = []
      for w in subset:
          for ww in subset:
              rows.append(dict(w1=w, w2=ww, d=model.wv.similarity(w, ww)))
      import pandas
      pandas.DataFrame(rows).pivot("w1", "w2", "d")
```

```
[46]: w2          about      above  absolutely      after      before
      w1
      about      1.000000  0.193730  -0.248471  0.426798  0.081973
      above      0.193730  1.000000   0.215933  0.232336  0.081327
      absolutely -0.248471  0.215933   1.000000  0.062538  0.059916
      after      0.426798  0.232336   0.062538  1.000000  0.216423
      before      0.081973  0.081327   0.059916  0.216423  1.000000
```

## 1.7 word2vec pré-entraînés

Ce modèle est plus performant avec plus de données. On peut télécharger des modèles pré-entraînés sur des données plus volumineuses : [Pre-Trained Word2Vec Models](#) ou encore [Pre-trained word vectors of 30+ languages](#). Ceux-ci sont plutôt gros (> 600 Mo). Le module *spacy* propose une version plus légère et mieux documentée [Word Vectors and Semantic Similarity](#) avec les données `en_core_web_md`.

```
[47]: import spacy
      from spacy.cli import download
      # download("en_core_web_md") # ça ne marche pas toujours
```

```
[48]: import os
      version = "2.1.0"
      unzip_dest = 'en_core_web_md-{}-tar/dist/en_core_web_md-{}-en_core_web_md/
      ↪en_core_web_md-{}'.format(version)
      if not os.path.exists(unzip_dest):
          from pyquickhelper.pycode import is_travis_or_appveyor
          if not is_travis_or_appveyor():
              # On le fait seulement si ce n'est pas un test d'intégration continue.
```

```

url = "https://github.com/explosion/spacy-models/releases/download/
→en_core_web_md-%s/" % version
name = "en_core_web_md-%s.tar.gz" % version
print("Téléchargement de ", name)
from pyensae.datasource import download_data
unzipped = download_data(name, url=url, fLOG=print)
unzip_dest = os.path.split(unzipped[0])[0]
unzip_dest = "en_core_web_md-{0}/en_core_web_md/en_core_web_md-{0}".
→format(version)
print("Found", unzip_dest)

if os.path.exists(unzip_dest):
    print("Chargement des données par spacy.")
    nlp = spacy.load(unzip_dest)
    continue_wv = True
else:
    continue_wv = False
    print('Pas de données on passe la suite.')

```

Téléchargement de en\_core\_web\_md-2.1.0.tar.gz  
Found en\_core\_web\_md-2.1.0/en\_core\_web\_md/en\_core\_web\_md-2.1.0  
Chargement des données par spacy.

```

[49]: if continue_wv:
    tokens = nlp('after before films about above absolutely')
    rows = []
    for token1 in tokens:
        for token2 in tokens:
            sim = model.wv.similarity(token1.text, token2.text)
            rows.append(dict(w1=token1.text, w2=token2.text, d=sim))
pandas.DataFrame(rows).pivot("w1", "w2", "d")

```

```

[49]: w2          about      above  absolutely    after    before    films
w1
about          1.000000  0.193730  -0.248471  0.426798  0.081973  0.175330
above          0.193730  1.000000   0.215933  0.232336  0.081327 -0.226075
absolutely    -0.248471  0.215933   1.000000  0.062538  0.059916 -0.033369
after          0.426798  0.232336   0.062538  1.000000  0.216423  0.116398
before         0.081973  0.081327   0.059916  0.216423  1.000000  0.147834
films          0.175330 -0.226075  -0.033369  0.116398  0.147834  1.000000

```

```

[50]: if continue_wv:
    print(tokens[0].vector.shape, tokens[0].vector[:10])

```

```

(300,) [ 0.2069  0.44321 -0.12522 -0.017724 -0.064277 -0.44308  0.014019
-0.10119  0.22699  3.1689 ]

```

```

[51]: import numpy

def spacy_sum_vectors(phrase, nlp):
    dec = nlp(phrase)
    return sum(w.vector for w in dec)

```

```

def spacy_word2vec_features(X, nlp):
    feats = numpy.vstack([spacy_sum_vectors(p, nlp) for p in X])
    return feats

if continue_wv:
    try:
        wv_train_feat2 = spacy_word2vec_features(X_train["sentence"], nlp)
        print(wv_train_feat2.shape)
    except ValueError as e:
        print(e)
        continue_wv = False

```

(2250, 300)

```

[52]: if continue_wv:
        clfwv2 = LogisticRegression(solver='lbfgs', max_iter=1000)
        clfwv2.fit(wv_train_feat2, y_train)

```

```

[53]: if continue_wv:
        wv_test_feat2 = spacy_word2vec_features(X_test["sentence"], nlp)

```

```

[54]: if continue_wv:
        print(clfwv2.score(wv_test_feat2, y_test))

```

0.8053333333333333

C'est un peu mieux mais un peu plus coûteux en temps de calcul mais même sans entraînement, le modèle obtenu est plus performant avec 300 dimensions que celui obtenu avec l'ACP. Le corpus extérieur au problème apporte de la valeur.

[55]:

[56]: