lecture04

March 5, 2025

1 DATA 607 - Machine Learning

1.1 Class 4 — 2025.05.03 — Text Day

1.1.1 Document embedding, classification, and retrieval

[28]:
```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
from icecream import ic
from sklearn.pipeline import make_pipeline
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.linear_model import LogisticRegression
```

1.1.2 20newsgroups

> The 20 newsgroups dataset comprises around 18000 newsgroups posts on 20 topics split in two subsets: one for training (or development) and the other one for testing (or for performance evaluation). The split between the train and test set is based upon a messages posted before and after a specific date.
>
> — the Scikit Learn docs

- The posts include headers, footers, and quotes. As it turns out, this really helps with classification! We'll work without them, though.
- Since we'll be focusing on model building, we won't touch the test set. We'll draw validation sets from the training data.

[29]:
```
# 20newsgroups, a real-world dataset

from sklearn.datasets import fetch_20newsgroups
from sklearn.utils import Bunch


bunch = fetch_20newsgroups(subset="train", remove=("headers", "footers",␣
↪"quotes"))
assert isinstance(bunch, Bunch)
```

```
X = bunch.data
y = bunch.target

print(f"y[0] = {bunch.target_names[y[0]]}\n\nX[0] = {X[0]}")
```

```
y[0] = rec.autos

X[0] = I was wondering if anyone out there could enlighten me on this car I saw
the other day. It was a 2-door sports car, looked to be from the late 60s/
early 70s. It was called a Bricklin. The doors were really small. In addition,
the front bumper was separate from the rest of the body. This is
all I know. If anyone can tellme a model name, engine specs, years
of production, where this car is made, history, or whatever info you
have on this funky looking car, please e-mail.
```

[30]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5)
print(len(X_train), len(X_test))
```

```
5657 5657
```

1.2 Sparse embeddings with CountVectorizer

[31]:
```
from sklearn.feature_extraction.text import CountVectorizer

counter = CountVectorizer()
counts = counter.fit_transform(X_train)
ic(counts)

print(
    f"proportion of nonzero entries = {len(counts.indices)/(counts.
 ↪shape[0]*counts.shape[1]):.4f}"
)
```

```
ic| counts: <Compressed Sparse Row sparse matrix of dtype 'int64'
                with 564342 stored elements and shape (5657, 71912)>

proportion of nonzero entries = 0.0014
```

1.2.1 MultinomialNaiveBayes

[295]:
```
from sklearn.naive_bayes import MultinomialNB

model = make_pipeline(CountVectorizer(), MultinomialNB())
model.fit(X_train, y_train)
ic(accuracy_score(y_train, model.predict(X_train)))
ic(accuracy_score(y_test, model.predict(X_test)))
```

```
ic| accuracy_score(y_train, model.predict(X_train)): 0.7187555241293972
ic| accuracy_score(y_test, model.predict(X_test)): 0.5016793353367509
```

[295]:
```
0.5016793353367509
```

- Can we improve predictive performance by tuning the `alpha` parameter of the `MultinomialNB` model?

[ ]:
```
param_grid = {"multinomialnb__alpha": [0.001, 0.01, 0.1, 1, 10]}
model = make_pipeline(CountVectorizer(), MultinomialNB())
search = GridSearchCV(model, param_grid, scoring="accuracy")
best_model = search.fit(X_train, y_train).best_estimator_  # returns the best␣
↪estimator
display(best_model)

ic(search.best_params_)
ic(search.best_score_)
ic(accuracy_score(y_test, best_model.predict(X_test)))
```

```
Pipeline(steps=[('countvectorizer', CountVectorizer()),
                ('multinomialnb', MultinomialNB(alpha=0.1))])
```

```
ic| search.best_params_: {'multinomialnb__alpha': 0.1}
ic| search.best_score_: np.float64(0.6694394716205366)
ic| accuracy_score(y_test, best_model.predict(X_test)): 0.6818101467208768
```

1.2.2 Tuning `CountVectorizer` for `MultinomialNaiveBayes`

- `CountVectorizer` also has knobs we can twiddle. See its documentation for details.

[ ]:
```
param_grid = {
    "countvectorizer__stop_words": [None, "english"],
    "countvectorizer__strip_accents": [None, "ascii"],
}
model = make_pipeline(CountVectorizer(), MultinomialNB(alpha=0.1))
search = GridSearchCV(model, param_grid, scoring="accuracy")
best_model = search.fit(X_train, y_train).best_estimator_  # returns the best␣
↪estimator
display(best_model)

ic(search.best_params_)
ic(search.best_score_)
ic(accuracy_score(y_test, best_model.predict(X_test)))
```

```
Pipeline(steps=[('countvectorizer', CountVectorizer(stop_words='english')),
                ('multinomialnb', MultinomialNB(alpha=0.1))])
```

```
ic| search.best_params_: {'countvectorizer__stop_words': 'english',
                          'countvectorizer__strip_accents': None}
```

```
ic| search.best_score_: np.float64(0.6777461704048764)
ic| accuracy_score(y_test, best_model.predict(X_test)): 0.694891285133463
```

[ ]:
```
param_grid = {"countvectorizer__max_df": [0.2, 0.4, 0.6, 0.8, 1.0]}
model = make_pipeline(CountVectorizer(stop_words="english"),␣
↪MultinomialNB(alpha=0.1))
search = GridSearchCV(model, param_grid, scoring="accuracy")
best_model = search.fit(X_train, y_train).best_estimator_  # returns the best␣
↪estimator
display(best_model)

ic(search.best_params_)
ic(search.best_score_)
ic(accuracy_score(y_test, best_model.predict(X_test)))
```

```
Pipeline(steps=[('countvectorizer',
                 CountVectorizer(max_df=0.2, stop_words='english')),
                ('multinomialnb', MultinomialNB(alpha=0.1))])
```

```
ic| search.best_params_: {'countvectorizer__max_df': 0.2}
ic| search.best_score_: np.float64(0.6782760495262018)
ic| accuracy_score(y_test, best_model.predict(X_test)): 0.695775145837016
```

Exercise

- Can you improve performance by tuning `CountVectorizer`'s `min_df` parameter?

1.2.3 `LogisticRegression`

[ ]:
```
model = make_pipeline(
    CountVectorizer(stop_words="english", max_df=0.2), LogisticRegression()
)
model.fit(X_train, y_train)
ic(accuracy_score(y_train, model.predict(X_train)))
ic(accuracy_score(y_test, model.predict(X_test)))
```

```
ic| accuracy_score(y_train, model.predict(X_train)): 0.9703022803606152
ic| accuracy_score(y_test, model.predict(X_test)): 0.6558246420364151
```

[ ]:
```
param_grid = {"logisticregression__C": [0.001, 0.01, 0.1, 1, 10]}
model = make_pipeline(
    CountVectorizer(stop_words="english", max_df=0.2), LogisticRegression()
)
search = GridSearchCV(model, param_grid, scoring="accuracy")
best_model = search.fit(X_train, y_train).best_estimator_  # returns the best␣
↪estimator
display(best_model)
```

```
ic(search.best_params_)
ic(search.best_score_)
ic(accuracy_score(y_test, best_model.predict(X_test)))
```

```
Pipeline(steps=[('countvectorizer',
                 CountVectorizer(max_df=0.2, stop_words='english')),
                ('logisticregression', LogisticRegression(C=0.1))])
```

```
ic| search.best_params_: {'logisticregression__C': 0.1}
ic| search.best_score_: np.float64(0.6581190853336583)
ic| accuracy_score(y_test, best_model.predict(X_test)): 0.6740321725296093
```

1.2.4 SGDClassifier

[266]:
```
from sklearn.linear_model import SGDClassifier

param_grid = {"sgdclassifier__alpha": [0.0001, 0.001, 0.01, 0.1, 1.0]}
model = make_pipeline(
    CountVectorizer(stop_words="english", max_df=0.2), SGDClassifier()
)
search = GridSearchCV(model, param_grid, scoring="accuracy")
best_model = search.fit(X_train, y_train).best_estimator_  # returns the best
 ↪estimator
display(best_model)

ic(search.best_params_)
ic(search.best_score_)
ic(accuracy_score(y_test, best_model.predict(X_test)))
```

```
Pipeline(steps=[('countvectorizer',
                 CountVectorizer(max_df=0.2, stop_words='english')),
                ('sgdclassifier', SGDClassifier(alpha=0.01))])
```

```
ic| search.best_params_: {'sgdclassifier__alpha': 0.01}
ic| search.best_score_: np.float64(0.6788007735735284)
ic| accuracy_score(y_test, best_model.predict(X_test)): 0.6947145129927523
```

[266]: 0.6947145129927523

Normalizer to normalize rows

- Contrast with `StandardScaler` that operates on columns.

[40]:
```
from sklearn.preprocessing import Normalizer

A = np.random.normal(size=(2, 4))

normalizer = Normalizer()
```

```
assert np.allclose(
    normalizer.fit_transform(A), A / np.linalg.norm(A, axis=1, keepdims=True)
)
```

- Default parameter values usually work better with normalized data.

[41]:
```
# Normalize the count data, can use the default value for alpha in SGDClassifier

from sklearn.linear_model import SGDClassifier


model = make_pipeline(
    CountVectorizer(stop_words="english", max_df=0.2), Normalizer(),␣
  ↪SGDClassifier()
)
display(model)

model.fit(X_train, y_train)
ic(accuracy_score(y_test, model.predict(X_test)))
```

```
Pipeline(steps=[('countvectorizer',
                 CountVectorizer(max_df=0.2, stop_words='english')),
                ('normalizer', Normalizer()),
                ('sgdclassifier', SGDClassifier())])
```

```
ic| accuracy_score(y_test, model.predict(X_test)): 0.699133816510518
```

[41]:
```
0.699133816510518
```

1.2.5 SVC (Support Vector Classifier)

[ ]:
```
from sklearn.svm import SVC
from sklearn.preprocessing import Normalizer

model = make_pipeline(
    CountVectorizer(stop_words="english", max_df=0.2),
    Normalizer(),  # SVC is sensitive to normalization!
    SVC(kernel="linear", C=1.0),
)

model.fit(X_train, y_train)

ic(accuracy_score(y_test, model.predict(X_test)))
```

```
ic| accuracy_score(y_test, model.predict(X_test)): 0.6618348948205762
```

[ ]:
```python
from sklearn.svm import LinearSVC

model = make_pipeline(
    CountVectorizer(stop_words="english", max_df=0.2),
    Normalizer(),  # LinearSVC is sensitive to normalization!
    LinearSVC(loss="hinge", max_iter=10000),
)

model.fit(X_train, y_train)

ic(accuracy_score(y_test, model.predict(X_test)))
```

```
ic| accuracy_score(y_test, model.predict(X_test)): 0.7037298921689942
```

1.2.6 IDF (Inverse Document Frequency) weighting

- Words that apear in lots of documents, are "less informative".
- ***Document frequency*** of the term t:

$$\mathrm{df}(t) = \text{proportion of documents containing } t$$

- ***Inverse document frequency*** of the term t:

$$\mathrm{idf}(t) = \log \frac{1}{\mathrm{df}(t)}$$

  Even though it's not reflected in the name, the logarithmic scaling is standard.

- Scikit Learn does some extra smoothing by default, so these aren't the exact quantities it computes.

1.2.7 `TfidfVectorizer`

- Weights each term-count by the corresponding inverse document frequency.
- Concretely, `TfidfVectorizer` multiplies the j-th column of the count matrix returned by `CountVectorizer.transform` by the inverse document frequency of the j-th term. Each row of the resulting matrix is then normalized to have length 1.
- This is a bit trick in practice because of *sparse matrices*, but here it is explicitly:

[9]:
```python
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.preprocessing import normalize

count_vectorizer = CountVectorizer().fit(X_train)
counts = count_vectorizer.transform(X_train)

tfidf_vectorizer = TfidfVectorizer().fit(X_train)
assert tfidf_vectorizer.vocabulary_ == count_vectorizer.vocabulary_
```

```
document_indices, term_indices = counts.nonzero()
smoothed_counts = counts.astype(float)
idf_weights = tfidf_vectorizer.idf_[term_indices]
smoothed_counts.data *= idf_weights

assert np.allclose(
    normalize(smoothed_counts).data, tfidf_vectorizer.transform(X_train).data
)
assert np.all(
    normalize(smoothed_counts).indices == tfidf_vectorizer.transform(X_train).
 ↪indices
)
```

- Let's try it out with LinearSVC

[6]:
```
from sklearn.svm import LinearSVC
from sklearn.feature_extraction.text import TfidfVectorizer


model = make_pipeline(
    TfidfVectorizer(stop_words="english"),
    LinearSVC(loss="hinge", max_iter=10000),
)

model.fit(X_train, y_train)

ic(accuracy_score(y_test, model.predict(X_test)))
```

```
ic| accuracy_score(y_test, model.predict(X_test)): 0.7399681810146721
```

[6]:
```
0.7399681810146721
```

1.2.8 Back to MultinomialNaiveBayes

[ ]:
```
model = make_pipeline(
    TfidfVectorizer(
        stop_words="english"
    ),  # Try letting stop_words revert to the default!
    MultinomialNB(),
)

model.fit(X_train, y_train)

ic(accuracy_score(y_test, model.predict(X_test)))
```

```
ic| accuracy_score(y_test, model.predict(X_test)): 0.6860526780979318
```

Retune alpha…

[ ]:
```python
param_grid = {"multinomialnb__alpha": np.logspace(-2, -1, 20)}

# With a higher value of alpha, stop-words don't help anymore.
# Regularization can often be used in place of feature selection.

model = make_pipeline(TfidfVectorizer(), MultinomialNB())

search = GridSearchCV(model, param_grid, scoring="accuracy")
best_model = search.fit(X_train, y_train).best_estimator_
display(best_model)

ic(search.best_params_)
ic(search.best_score_)
ic(accuracy_score(y_test, best_model.predict(X_test)))
```

```
Pipeline(steps=[('tfidfvectorizer', TfidfVectorizer()),
                ('multinomialnb', MultinomialNB(alpha=np.float64(0.01)))])
```

```
ic| search.best_params_: {'multinomialnb__alpha': np.float64(0.01)}
ic| search.best_score_: np.float64(0.7240573244228661)
ic| accuracy_score(y_test, best_model.predict(X_test)): 0.7392610924518296
```

[ ]:
```
0.7392610924518296
```

1.3 Pretrained embeddings

1.3.1 GLoVe embeddings

- Pennington, Socher, Manning (2014). **GloVe: Global Vectors for Word Representation**.
- https://nlp.stanford.edu/projects/glove/

```bash
wget http://nlp.stanford.edu/data/glove.6B.zip
unzip -l glove.6B.zip
```

```
Archive:  glove.6B.zip
  Length      Date    Time    Name
---------  ---------- -----   ----
171350079  08-04-2014 14:15   glove.6B.50d.txt
347116733  08-04-2014 14:14   glove.6B.100d.txt
693432828  08-04-2014 14:14   glove.6B.200d.txt
1037962819  08-27-2014 13:19   glove.6B.300d.txt
---------                     -------
2249862459                     4 files
```

[7]:
```python
EMB_DIM = 300

embeddings = np.zeros((400000, 300))
vocabulary = []
```

```python
with open("glove.6B.300d.txt") as f:
    for i, line in enumerate(f):
        word, coeffs = line.split(maxsplit=1)
        vocabulary.append(word)
        embeddings[i] = np.fromstring(coeffs, sep=" ")

print(len(vocabulary))
```

400000

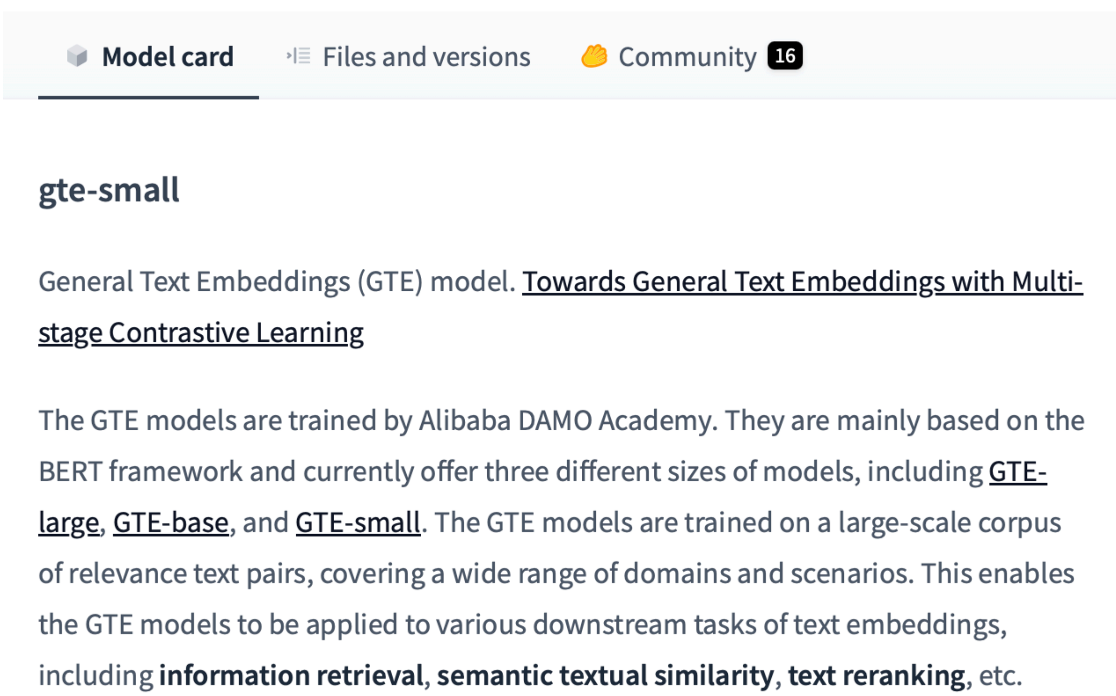
[10]:
```python
counter = CountVectorizer(vocabulary=vocabulary, stop_words="english")
counts_train = counter.fit_transform(X_train)
counts_test = counter.transform(X_test)

W_train = normalize(counts_train @ embeddings)
W_test = normalize(counts_test @ embeddings)
```

[ ]:
```python
model = LinearSVC()
model.fit(W_train, y_train)
ic(accuracy_score(y_test, model.predict(W_test)))
```

ic| accuracy_score(y_test, model.predict(W_test)): 0.6506982499558069

1.3.2 GTE (General Text Embeddings)

- See https://huggingface.co/thenlper/gte-small.



Model card | Files and versions | Community 16

gte-small

General Text Embeddings (GTE) model. Towards General Text Embeddings with Multi-stage Contrastive Learning

The GTE models are trained by Alibaba DAMO Academy. They are mainly based on the BERT framework and currently offer three different sizes of models, including GTE-large, GTE-base, and GTE-small. The GTE models are trained on a large-scale corpus of relevance text pairs, covering a wide range of domains and scenarios. This enables the GTE models to be applied to various downstream tasks of text embeddings, including **information retrieval**, **semantic textual similarity**, **text reranking**, etc.

[32]:
```
from sentence_transformers import SentenceTransformer

model = SentenceTransformer("thenlper/gte-small")
X_train_gte_small = model.encode(X_train)
X_test_gte_small = model.encode(X_test)

# Takes a few minutes...
```

[33]:
```
np.savez(
    "20newsgroups_gte_small.npz",
    X_train_gte_small=X_train_gte_small,
    X_test_gte_small=X_test_gte_small,
    y_train=y_train,
    y_test=y_test,
)
```

[ ]:
```
data = np.load("20newsgroups_gte_small.npz")
X_train_gte_small = data["X_train_gte_small"]
X_test_gte_small = data["X_test_gte_small"]
y_train = data["y_train"]
y_test = data["y_test"]

ic(X_train_gte_small.shape, X_test_gte_small.shape, y_train.shape, y_test.shape)
```

```
ic| X_train_gte_small.shape: (5657, 384)
    X_test_gte_small.shape: (5657, 384)
    y_train.shape: (5657,)
    y_test.shape: (5657,)
```

[45]:
```
model = SGDClassifier()
model.fit(X_train_gte_small, y_train)
accuracy_score(y_test, model.predict(X_test_gte_small))
```

[45]: 0.7192858405515291

[46]:
```
model = LinearSVC()
model.fit(X_train_gte_small, y_train)
accuracy_score(y_test, model.predict(X_test_gte_small))
```

[46]: 0.7392610924518296

1.3.3 Proximity in embedding space reflects semantic similarity

- Euclidean distance in embedding space:

$$\mathrm{S}(x, x') = \|\,\mathrm{embedding}(x) - \mathrm{embedding}(x')\|$$

- Cosine similarity in embedding space:

$$\mathrm{S}(x, x') = \cos\left(\text{angle between embedding}(x) \text{ and embedding}(x')\right)$$

```
[ ]: from sklearn.metrics.pairwise import cosine_similarity
from sklearn.neighbors import NearestNeighbors
from collections import Counter

i = 1234

x = X_test_gte_small[i]
ic(y_test[i])

I = (
    cosine_similarity(x.reshape(1, -1), X_train_gte_small)
    .squeeze()
    .argsort()[::-1][:50]
)

ic(Counter(y_train[I]))

nns = NearestNeighbors()
nns.fit(X_train_gte_small)
distances, J = nns.kneighbors(x.reshape(1, -1), 50)
J = J.squeeze()
ic(Counter(y_train[J]))
```

```
ic| y_test[i]: np.int64(5)
ic| Counter(y_train[I]): Counter({np.int64(5): 48, np.int64(1): 1, np.int64(2):
1})
ic| Counter(y_train[J]): Counter({np.int64(5): 48, np.int64(1): 1, np.int64(2):
1})
```

1.3.4 Relevance of a document to a query

- We want to retrieve documents from a collection that are most relevant to a query.
- There are many ways to assign a **relevance score** $\mathrm{score}(D, Q)$ indicating the relevance of a document D to a query Q.
- BM25:

$$\mathrm{S}(D, Q) = \sum_{t \in Q} \mathrm{idf}(t) \frac{f(t, D)(k_1 + 1)}{f(t, D) + k_1 \left(1 - b + b \frac{\mathrm{len}(D)}{\mathrm{av.doc.len.}}\right)}$$

  Here, $f(t, D)$ be the frequency of occurence of term t in document D, i.e., how many times it appears.

Vector search

- Euclidean distance in embedding space:

$$\mathrm{S}(D, Q) = \|\mathrm{embedding}(D) - \mathrm{embedding}(Q)\|$$

- Cosine similarity in embedding space:

$$\mathrm{S}(D, Q) = \cos\left(\text{angle between embedding}(D) \text{ and embedding}(Q)\right)$$

- Used for semantic search, recommendation/ranking, …
- The R in RAG (Retrieval Augmented Generation)

[ ]:
```python
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.neighbors import NearestNeighbors
from collections import Counter

i = 1234

x = X_test_gte_small[i]
ic(y_test[i])

I = (
    cosine_similarity(x.reshape(1, -1), X_train_gte_small)
    .squeeze()
    .argsort()[::-1][:50]
)

ic(Counter(y_train[I]))

nns = NearestNeighbors()
nns.fit(X_train_gte_small)
distances, J = nns.kneighbors(x.reshape(1, -1), 50)
J = J.squeeze()
ic(Counter(y_train[J]))
```

```
ic| y_test[i]: np.int64(5)
ic| Counter(y_train[I]): Counter({np.int64(5): 48, np.int64(1): 1, np.int64(2):
1})
ic| Counter(y_train[J]): Counter({np.int64(5): 48, np.int64(1): 1, np.int64(2):
1})
```