文本分类或回归方法

文本问题是我的最爱。一般来说，这些问题也被称为**自然语言处理（NLP）问题**。NLP 问题与图像问题也有很大不同。你需要创建以前从未为表格问题创建过的数据管道。你需要了解商业案例，才能建立一个好的模型。顺便说一句，机器学习中的任何事情都是如此。建立模型会让你达到一定的水平，但要想改善和促进你所建立模型的业务，你必须了解它对业务的影响。

NLP 问题有很多种，其中最常见的是字符串分类。很多时候，我们会看到人们在处理表格数据或图像时表现出色，但在处理文本时，他们甚至不知道从何入手。文本数据与其他类型的数据集没有什么不同。对于计算机来说，一切都是数字。

假设我们从情感分类这一基本任务开始。我们将尝试对电影评论进行情感分类。因此，您有一个文本，并有与之相关的情感。你将如何处理这类问题？是应用深度神经网络？ 不，绝对错了。你要从最基本的开始。让我们先看看这些数据是什么样子的。

我们从**IMDB 电影评论数据集**开始，该数据集包含 25000 篇正面情感评论和 25000 篇负面情感评论。

我将在此讨论的概念几乎适用于任何文本分类数据集。

这个数据集非常容易理解。一篇评论对应一个目标变量。请注意，我写的是评论而不是句子。评论就是一堆句子。所以，到目前为止，你一定只看到了对单句的分类，但在这个问题中，我们将对多个句子进行分类。简单地说，这意味着不仅一个句子会对情感产生影响，而且情感得分是多个句子得分的组合。数据简介如图 1 所示。

	review	sentiment
0	One of the other reviewers has mentioned that ...	positive
1	A wonderful little production. The...	positive
2	I thought this was a wonderful way to spend ti...	positive
3	Basically there's a family where a little boy ...	negative
4	Petter Mattei's "Love in the Time of Money" is...	positive

如何着手解决这样的问题？一个简单的方法就是手工制作两份单词表。一个列表包含你能想象到的所有正面词汇，例如好、棒、好等；另一个列表包含所有负面词汇，例如坏、恶等。我们先不要举例说明坏词，否则这本书就只能供 18 岁以上的人阅读了。一旦你有了这些列表，你甚至不需要一个模型来进行预测。这些列表也被称为情感词典。你可以用一个简单的计数器来计算句子

中正面和负面词语的数量。如果正面词语的数量较多，则表示该句子具有正面情感；如果负面词语的数量较多，则表示该句子具有负面情感。如果句子中没有这些词，则可以说该句子具有中性情感。这是最古老的方法之一，现在仍有人在使用。它也不需要太多代码。

```
def find_sentiment(sentence, pos, neg):
    sentence = sentence.split()
    sentence = set(sentence)
    num_common_pos = len(sentence.intersection(pos))
    num_common_neg = len(sentence.intersection(neg))
    if num_common_pos > num_common_neg:
        return "positive"
    if num_common_pos < num_common_neg:
        return "negative"
    return "neutral"
```

不过，这种方法考虑的因素并不多。正如你所看到的，我们的 split() 也并不完美。如果使用 split()，就会出现这样的句子：

"hi, how are you?"

经过分割后变为：

["hi,", "how","are","you?"]

这种方法并不理想，因为单词中包含了逗号和问号，它们并没有被分割。因此，如果没有在分割前对这些特殊字符进行预处理，不建议使用这种方法。将字符串拆分为单词列表称为标记化。最流行的标记化方法之一来自 NLTK（自然语言工具包）。

```
In [X]: from nltk.tokenize import word_tokenize
In [X]: sentence = "hi, how are you?"
In [X]: sentence.split()
Out[X]: ['hi,', 'how', 'are', 'you?']
In [X]: word_tokenize(sentence)
Out[X]: ['hi', ',', 'how', 'are', 'you', '?']
```

正如您所看到的，使用 NLTK 的单词标记化功能，同一个句子的拆分效果要好得多。使用单词列表进行对比的效果也会更好！这就是我们将应用于第一个情感检测模型的方法。

在处理 NLP 分类问题时，您应该经常尝试的基本模型之一是词袋模型（bag of words）。在词袋模型中，我们创建一个巨大的稀疏矩阵，存储语料库（语料库=所有文档=所有句子）中所有单词的计数。为此，我们将使用 scikit-learn 中的 CountVectorizer。让我们看看它是如何工作的。

```
from sklearn.feature_extraction.text import CountVectorizer

corpus = [
    "hello, how are you?",
    "im getting bored at home. And you? What do you think?",
    "did you know about counts",
    "let's see if this works!",
    "YES!!!!"
]
ctv = CountVectorizer()
ctv.fit(corpus)
corpus_transformed = ctv.transform(corpus)
```

如果我们打印 corpus_transformed，就会得到类似下面的结果：

```
(0, 2)      1
(0, 9)      1
(0, 11)     1
(0, 22)     1
(1, 1)      1
(1, 3)      1
(1, 4)      1
(1, 7)      1
(1, 8)      1
(1, 10)     1
(1, 13)     1
(1, 17)     1
(1, 19)     1
(1, 22)     2
(2, 0)      1
(2, 5)      1
(2, 6)      1
(2, 14)     1
(2, 22)     1
(3, 12)     1
(3, 15)     1
(3, 16)     1
(3, 18)     1
(3, 20)     1
(4, 21)     1
```

在前面的章节中，我们已经见识过这种表示法。即稀疏表示法。因此，语料库现在是一个稀疏矩阵，其中第一个样本有 4 个元素，第二个样本有 10 个元素，以此类推，第三个样本有 5 个元素，以此类推。我们还可以看到，这些元素都有相关的计数。有些元素会出现两次，有些则只有一次。例如，在样本 2（第 1 行）中，我们看到第 22 列的数值是 2。这是为什么呢？第 22 列是什么？

CountVectorizer 的工作方式是首先对句子进行标记化处理，然后为每个标记赋值。因此，每个标记都由一个唯一索引表示。这些唯一索引就是我们看到的列。CountVectorizer 会存储这些信息。

```
print(ctv.vocabulary_)
{'hello': 9, 'how': 11, 'are': 2, 'you': 22, 'im': 13, 'getting': 8,
'bored': 4, 'at': 3, 'home': 10, 'and': 1, 'what': 19, 'do': 7, 'think':
17, 'did': 6, 'know': 14, 'about': 0, 'counts': 5, 'let': 15, 'see': 16,
'if': 12, 'this': 18, 'works': 20, 'yes': 21}
```

我们看到，索引 22 属于 "you"，而在第二句中，我们使用了两次 "you"。我希望大家现在已经清楚什么是词袋了。但是我们还缺少一些特殊字符。有时，这些特殊字符也很有用。例如，"? "在大多数句子中表示疑问句。让我们把 scikit-learn 的 word_tokenize 整合到 CountVectorizer 中，看看会发生什么。

```
from sklearn.feature_extraction.text import CountVectorizer
from nltk.tokenize import word_tokenize

corpus = [
    "hello, how are you?",
    "im getting bored at home. And you? What do you think?",
    "did you know about counts",
    "let's see if this works!",
    "YES!!!!"
]
ctv = CountVectorizer(tokenizer=word_tokenize, token_pattern=None)
ctv.fit(corpus)
corpus_transformed = ctv.transform(corpus)
print(ctv.vocabulary_)
```

这样，我们的词袋就变成了：

```
{'hello': 14, ',': 2, 'how': 16, 'are': 7, 'you': 27, '?': 4, 'im': 18,
'getting': 13, 'bored': 9, 'at': 8, 'home': 15, '.': 3, 'and': 6, 'what':
24, 'do': 12, 'think': 22, 'did': 11, 'know': 19, 'about': 5, 'counts':
10, 'let': 20, "'s": 1, 'see': 21, 'if': 17, 'this': 23, 'works': 25,
'!': 0, 'yes': 26}
```

我们现在可以利用 IMDB 数据集中的所有句子创建一个稀疏矩阵，并建立一个模型。该数据集中正负样本的比例为 1:1，因此我们可以使用准确率作为衡量标准。我们将使用 StratifiedKFold 并创建一个脚本来训练5个折叠。你会问使用哪个模型？对于高维稀疏数据，哪个模型最快？逻辑回归。我们将首先使用逻辑回归来处理这个数据集，并创建第一个基准模型。

让我们看看如何做到这一点。

```
import pandas as pd
from nltk.tokenize import word_tokenize
from sklearn import linear_model
from sklearn import metrics
from sklearn import model_selection
from sklearn.feature_extraction.text import CountVectorizer
if __name__ == "__main__":
    df = pd.read_csv("../input/imdb.csv")
    df.sentiment = df.sentiment.apply(
        lambda x: 1 if x == "positive" else 0
    )
    df["kfold"] = -1
    df = df.sample(frac=1).reset_index(drop=True)
    y = df.sentiment.values
    kf = model_selection.StratifiedKFold(n_splits=5)
    for f, (t_, v_) in enumerate(kf.split(X=df, y=y)):
        df.loc[v_, 'kfold'] = f

    for fold_ in range(5):
        train_df = df[df.kfold != fold_].reset_index(drop=True)
        test_df = df[df.kfold == fold_].reset_index(drop=True)
        count_vec = CountVectorizer(
            tokenizer=word_tokenize,
            token_pattern=None
        )
        count_vec.fit(train_df.review)
        xtrain = count_vec.transform(train_df.review)
        xtest = count_vec.transform(test_df.review)
        model = linear_model.LogisticRegression()
        model.fit(xtrain, train_df.sentiment)
        preds = model.predict(xtest)
```

```
        accuracy = metrics.accuracy_score(test_df.sentiment, preds)
        print(f"Fold: {fold_}")
        print(f"Accuracy = {accuracy}")
        print("")
```

这段代码的运行需要一定的时间，但可以得到以下输出结果：

```
Fold: 0
Accuracy = 0.8903

Fold: 1
Accuracy = 0.897

Fold: 2
Accuracy = 0.891

Fold: 3
Accuracy = 0.8914

Fold: 4
Accuracy = 0.8931
```

哇，准确率已经达到 89%，而我们所做的只是使用词袋和逻辑回归！这真是太棒了！不过，这个模型的训练花费了很多时间，让我们看看能否通过使用朴素贝叶斯分类器来缩短训练时间。朴素贝叶斯分类器在 NLP 任务中相当流行，因为稀疏矩阵非常庞大，而朴素贝叶斯是一个简单的模型。要使用这个模型，需要更改一个导入和模型的行。让我们看看这个模型的性能如何。我们将使用 scikit-learn 中的 MultinomialNB。

```
import pandas as pd
from nltk.tokenize import word_tokenize
from sklearn import naive_bayes
from sklearn import metrics
from sklearn import model_selection
from sklearn.feature_extraction.text import CountVectorizer


model = naive_bayes.MultinomialNB()
model.fit(xtrain, train_df.sentiment)
```

得到如下结果：

```
Fold: 0
```

```
Accuracy = 0.8444

Fold: 1
Accuracy = 0.8499

Fold: 2
Accuracy = 0.8422

Fold: 3
Accuracy = 0.8443

Fold: 4
Accuracy = 0.8455
```

我们看到这个分数很低。但朴素贝叶斯模型的速度非常快。

NLP 中的另一种方法是 TF-IDF，如今大多数人都倾向于忽略或不屑于了解这种方法。TF 是术语频率，IDF 是反向文档频率。从这些术语来看，这似乎有些困难，但通过 TF 和 IDF 的计算公式，事情就会变得很明显。

$$TF(t) = \frac{Number\ of\ times\ a\ term\ t\ appears\ in\ a\ document}{Total\ number\ of\ terms\ in\ the\ document}$$

$$IDF(t) = LOG\left(\frac{Total\ number\ of\ documents}{Number\ of\ documentswith\ term\ t\ in\ it}\right)$$

术语 t 的 TF-IDF 定义为：

$$TF - IDF(t) = TF(t) \times IDF(t)$$

与 scikit-learn 中的 CountVectorizer 类似，我们也有 TfidfVectorizer。让我们试着像使用 CountVectorizer 一样使用它。

```
from sklearn.feature_extraction.text import TfidfVectorizer
from nltk.tokenize import word_tokenize

corpus = [
    "hello, how are you?",
    "im getting bored at home. And you? What do you think?",
    "did you know about counts",
    "let's see if this works!",
    "YES!!!!"
]
tfv = TfidfVectorizer(tokenizer=word_tokenize, token_pattern=None)
tfv.fit(corpus)
corpus_transformed = tfv.transform(corpus)
print(corpus_transformed)
```

输出结果如下：

```
(0, 27)     0.2965698850220162
(0, 16)     0.4428321995085722
(0, 14)     0.4428321995085722
(0, 7)      0.4428321995085722
(0, 4)      0.35727423026525224
(0, 2)      0.4428321995085722
(1, 27)     0.35299699146792735
(1, 24)     0.2635440111190765
(1, 22)     0.2635440111190765
(1, 18)     0.2635440111190765
(1, 15)     0.2635440111190765
(1, 13)     0.2635440111190765
(1, 12)     0.2635440111190765
(1, 9)      0.2635440111190765
(1, 8)      0.2635440111190765
(1, 6)      0.2635440111190765
(1, 4)      0.42525129752567803
(1, 3)      0.2635440111190765
(2, 27)     0.31752680284846835
(2, 19)     0.4741246485558491
(2, 11)     0.4741246485558491
(2, 10)     0.4741246485558491
(2, 5)      0.4741246485558491
(3, 25)     0.38775666010579296
(3, 23)     0.38775666010579296
(3, 21)     0.38775666010579296
(3, 20)     0.38775666010579296
(3, 17)     0.38775666010579296
(3, 1)      0.38775666010579296
(3, 0)      0.3128396318588854
(4, 26)     0.2959842226518677
(4, 0)      0.9551928286692534
```

可以看到，这次我们得到的不是整数值，而是浮点数。用 TfidfVectorizer 代替 CountVectorizer 也是小菜一碟。Scikit-learn 还提供了 TfidfTransformer。如果你使用的是计数值，可以使用 TfidfTransformer 并获得与 TfidfVectorizer 相同的效果。

```
import pandas as pd
from nltk.tokenize import word_tokenize
from sklearn import linear_model
from sklearn import metrics
```

```
from sklearn import model_selection
from sklearn.feature_extraction.text import TfidfVectorizer

for fold_ in range(5):
    train_df = df[df.kfold != fold_].reset_index(drop=True)
    test_df = df[df.kfold == fold_].reset_index(drop=True)
    tfidf_vec = TfidfVectorizer(
        tokenizer=word_tokenize,
        token_pattern=None
    )
    tfidf_vec.fit(train_df.review)
    xtrain = tfidf_vec.transform(train_df.review)
    xtest = tfidf_vec.transform(test_df.review)
    model = linear_model.LogisticRegression()
    model.fit(xtrain, train_df.sentiment)
    preds = model.predict(xtest)
    accuracy = metrics.accuracy_score(test_df.sentiment, preds)
    print(f"Fold: {fold_}")
    print(f"Accuracy = {accuracy}")
    print("")
```

我们可以看看 TF-IDF 在逻辑回归模型上的表现如何。

```
Fold: 0
Accuracy = 0.8976

Fold: 1
Accuracy = 0.8998

Fold: 2
Accuracy = 0.8948

Fold: 3
Accuracy = 0.8912

Fold: 4
Accuracy = 0.8995
```

我们看到，这些分数都比 CountVectorizer 高一些，因此它成为了我们想要击败的新基准。

NLP 中另一个有趣的概念是 N-gram。N-grams 是按顺序排列的单词组合。N-grams 很容易创建。您只需注意顺序即可。为了让事情变得更简单，我们可以使用 NLTK 的 N-gram 实现。

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

N = 3
sentence = "hi, how are you?"
tokenized_sentence = word_tokenize(sentence)
n_grams = list(ngrams(tokenized_sentence, N))
print(n_grams)
```

由此得到：

```
[('hi', ',', 'how'),
(',', 'how', 'are'),
('how', 'are', 'you'),
('are', 'you', '?')]
```

同样，我们还可以创建 2-gram 或 4-gram 等。现在，这些 n-gram 将成为我们词汇表的一部分，当我们计算计数或 tf-idf 时，我们会将一个 n-gram 视为一个全新的标记。因此，在某种程度上，我们是在结合上下文。scikit-learn 的 CountVectorizer 和 TfidfVectorizer 实现都通过 ngram_range 参数提供 n-gram，该参数有最小和最大限制。默认情况下，该参数为（1, 1）。当我们将其改为 (1, 3) 时，我们将看到单字元、双字元和三字元。代码改动很小。

由于到目前为止我们使用 tf-idf 得到了最好的结果，让我们来看看包含 n-grams 直至 trigrams 是否能改进模型。唯一需要修改的是 TfidfVectorizer 的初始化。

```
tfidf_vec = TfidfVectorizer(
    tokenizer=word_tokenize,
    token_pattern=None,
    ngram_range=(1, 3)
)
```

让我们看看是否会有改进。

```
Fold: 0
Accuracy = 0.8931

Fold: 1
Accuracy = 0.8941

Fold: 2
Accuracy = 0.897
```

```
Fold: 3
Accuracy = 0.8922

Fold: 4
Accuracy = 0.8847
```

看起来还行，但我们看不到任何改进。 也许我们可以通过多使用 bigrams 来获得改进。 我不会在这里展示这一部分。也许你可以自己试着做。

NLP 的基础知识还有很多。你必须知道的一个术语是词干提取（strmming）。另一个是词形还原（lemmatization）。**词干提取和词形还原**可以将一个词减少到最小形式。在词干提取的情况下，处理后的单词称为词干单词，而在词形还原情况下，处理后的单词称为词形。必须指出的是，词形还原比词干提取更激进，而词干提取更流行和广泛。词干和词形都来自语言学。如果你打算为某种语言制作词干或词型，需要对该语言有深入的了解。如果要过多地介绍这些知识，就意味着要在本书中增加一章。使用 NLTK 软件包可以轻松完成词干提取和词形还原。让我们来看看这两种方法的一些示例。有许多不同类型的词干提取和词形还原器。我将用最常见的 Snowball Stemmer 和 WordNet Lemmatizer 来举例说明。

```
from nltk.stem import WordNetLemmatizer
from nltk.stem.snowball import SnowballStemmer

lemmatizer = WordNetLemmatizer()
stemmer = SnowballStemmer("english")
words = ["fishing", "fishes", "fished"]
for word in words:
    print(f"word={word}")
    print(f"stemmed_word={stemmer.stem(word)}")
    print(f"lemma={lemmatizer.lemmatize(word)}")
    print("")
```

这将打印：

```
word=fishing
stemmed_word=fish
lemma=fishing
word=fishes
stemmed_word=fish
lemma=fish
word=fished
stemmed_word=fish
lemma=fished
```

正如您所看到的，词干提取和词形还原是截然不同的。当我们进行词干提取时，我们得到的是一个词的最小形式，它可能是也可能不是该词所属语言词典中的一个词。但是，在词形还原情况下，这将是一个词。现在，您可以自己尝试添加词干和词素化，看看是否能改善结果。

您还应该了解的一个主题是主题提取。**主题提取**可以使用非负矩阵因式分解（NMF）或潜在语义分析（LSA）来完成，后者也被称为奇异值分解或 SVD。这些分解技术可将数据简化为给定数量的成分。 您可以在从 CountVectorizer 或 TfidfVectorizer 中获得的稀疏矩阵上应用其中任何一种技术。

让我们把它应用到之前使用过的 TfidfVetorizer 上。

```
import pandas as pd
from nltk.tokenize import word_tokenize
from sklearn import decomposition
from sklearn.feature_extraction.text import TfidfVectorizer
corpus = pd.read_csv("../input/imdb.csv", nrows=10000)
corpus = corpus.review.values
tfv = TfidfVectorizer(tokenizer=word_tokenize, token_pattern=None)
tfv.fit(corpus)
corpus_transformed = tfv.transform(corpus)
svd = decomposition.TruncatedSVD(n_components=10)
corpus_svd = svd.fit(corpus_transformed)
sample_index = 0
feature_scores = dict(
    zip(
        tfv.get_feature_names(),
        corpus_svd.components_[sample_index]
    )
)
N = 5
print(sorted(feature_scores, key=feature_scores.get, reverse=True)[:N])
```

您可以使用循环来运行多个样本。

```
N = 5
for sample_index in range(5):
    feature_scores = dict(
        zip(
            tfv.get_feature_names(),
            corpus_svd.components_[sample_index]
        )
    )
    print(
        sorted(
```

```
            feature_scores,
            key=feature_scores.get,
            reverse=True
        )[:N]
    )
```

输出结果如下：

```
['the', ',', '.', 'a', 'and']
['br', '<', '>', '/', '-']
['i', 'movie', '!', 'it', 'was']
[',', '!', "'", '``', 'you']
['!', 'the', '...', "'", '``']
```

你可以看到，这根本说不通。怎么办呢？让我们试着清理一下，看看是否有意义。要清理任何文本数据，尤其是 pandas 数据帧中的文本数据，可以创建一个函数。

```
import re
import string

def clean_text(s):
    s = s.split()
    s = " ".join(s)
    s = re.sub(f'[{re.escape(string.punctuation)}]', '', s)
    return s
```

该函数会将 "hi, how are you????" 这样的字符串转换为 "hi how are you"。让我们把这个函数应用到旧的 SVD 代码中，看看它是否能给提取的主题带来提升。使用 pandas，你可以使用 apply 函数将清理代码 "应用 "到任意给定的列中。

```
import pandas as pd
corpus = pd.read_csv("../input/imdb.csv", nrows=10000)
corpus.loc[:, "review"] = corpus.review.apply(clean_text)
```

请注意，我们只在主 SVD 脚本中添加了一行代码，这就是使用函数和 pandas 应用的好处。这次生成的主题如下。

```
['the', 'a', 'and', 'of', 'to']
['i', 'movie', 'it', 'was', 'this']
['the', 'was', 'i', 'were', 'of']
['her', 'was', 'she', 'i', 'he']
['br', 'to', 'they', 'he', 'show']
```

呼！至少这比我们之前好多了。但你知道吗？你可以通过在清理功能中删除停止词（stopwords）来使它变得更好。什么是stopwords？它们是存在于每种语言中的高频词。例如，在英语中，这些词包括 "a"、"an"、"the"、"for "等。删除停止词并非总是明智的选择，这在很大程度上取决于业务问题。像 "I need a new dog"这样的句子，去掉停止词后会变成 "need new dog"，此时我们不知道谁需要new dog。

如果我们总是删除停止词，就会丢失很多上下文信息。你可以在 NLTK 中找到许多语言的停止词，如果没有，你也可以在自己喜欢的搜索引擎上快速搜索一下。

现在，让我们转到大多数人都喜欢使用的方法：深度学习。但首先，我们必须知道什么是词嵌入（embedings for words）。你已经看到，到目前为止，我们已经将标记转换成了数字。因此，如果某个语料库中有 N 个唯一的词块，它们可以用 0 到 N-1 之间的整数来表示。现在，我们将用向量来表示这些整数词块。这种将单词表示成向量的方法被称为单词嵌入或单词向量。谷歌的 Word2Vec 是将单词转换为向量的最古老方法之一。此外，还有 Facebook 的 FastText 和斯坦福大学的 GloVe（用于单词表示的全局向量）。这些方法彼此大相径庭。

其基本思想是建立一个浅层网络，通过重构输入句子来学习单词的嵌入。因此，您可以通过使用周围的所有单词来训练网络预测一个缺失的单词，在此过程中，网络将学习并更新所有相关单词的嵌入。这种方法也被称为连续词袋或 CBoW 模型。您也可以尝试使用一个单词来预测上下文中的单词。这就是所谓的跳格模型。Word2Vec 可以使用这两种方法学习嵌入。

FastText 可以学习字符 n-gram 的嵌入。和单词 n-gram 一样，如果我们使用的是字符，则称为字符 n-gram，最后，GloVe 通过共现矩阵来学习这些嵌入。因此，我们可以说，所有这些不同类型的嵌入最终都会返回一个字典，其中键是语料库（例如英语维基百科）中的单词，值是大小为 N（通常为 300）的向量。

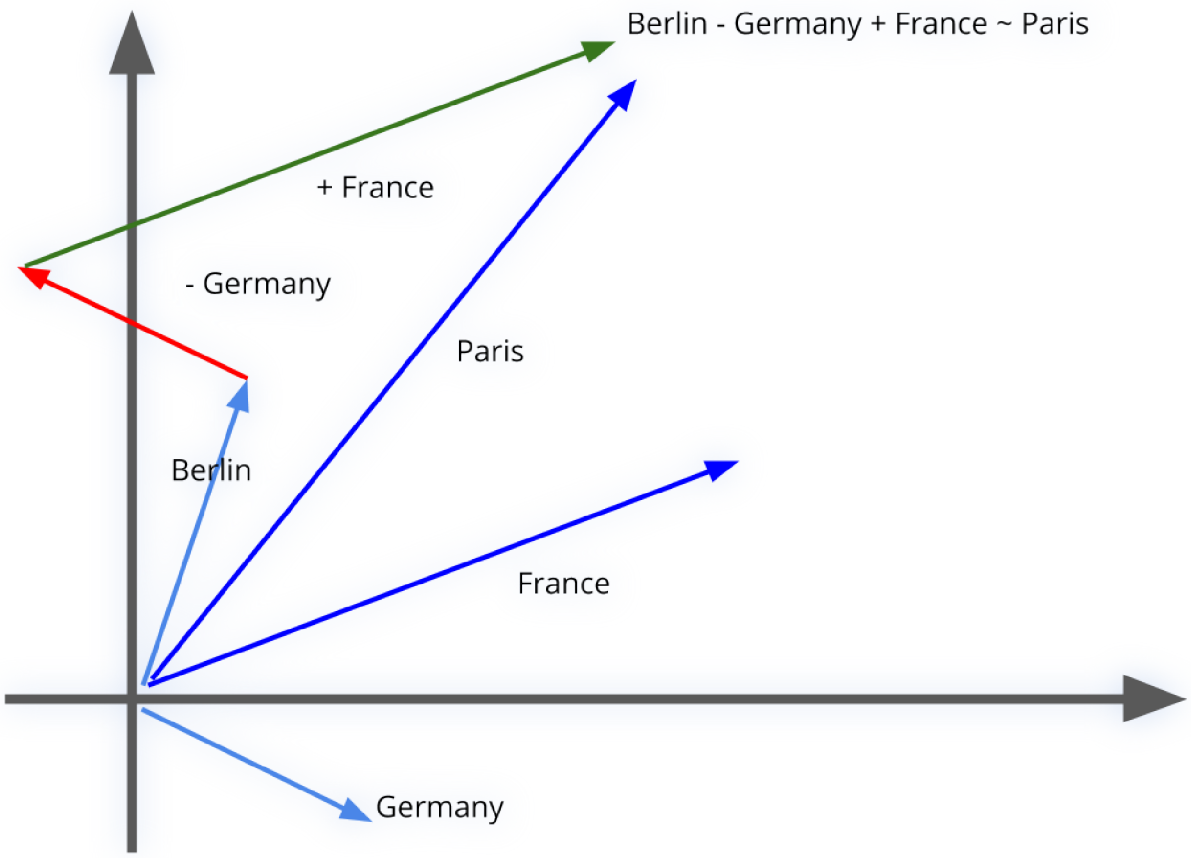

图 1：可视化二维单词嵌入。

图 1 显示了二维单词嵌入的可视化效果。假设我们以某种方式完成了词语的二维表示。图 1 显示，如果从Berlin（德国首都）的向量中减去德国（Germany）的向量，再加上法国（france）的向量，就会得到一个接近Paris（法国首都）的向量。由此可见，嵌入式也能进行类比。 这并不总是正确的，但这样的例子有助于理解单词嵌入的作用。像 "嗨，你好吗 "这样的句子可以用下面的一堆向量来表示。

hi —> [vector (v1) of size 300]

, —> [vector (v2) of size 300]

how —> [vector (v3) of size 300]

are —> [vector (v4) of size 300]

you —> [vector (v5) of size 300]

? —> [vector (v6) of size 300]

使用这些信息有多种方法。最简单的方法之一就是使用嵌入向量。如上例所示，每个单词都有一个 1x300 的嵌入向量。利用这些信息，我们可以计算出整个句子的嵌入。计算方法有多种。其中一种方法如下所示。在这个函数中，我们将给定句子中的所有单词向量提取出来，然后从所有标记词的单词向量中创建一个归一化的单词向量。这样就得到了一个句子向量。

```
import numpy as np
def sentence_to_vec(s, embedding_dict, stop_words, tokenizer):
    words = str(s).lower()
    words = tokenizer(words)
    words = [w for w in words if not w in stop_words]
    words = [w for w in words if w.isalpha()]
    M = []
    for w in words:
        if w in embedding_dict:
            M.append(embedding_dict[w])
    if len(M) == 0:
        return np.zeros(300)
    M = np.array(M)
    v = M.sum(axis=0)
    return v / np.sqrt((v ** 2).sum())
```

我们可以用这种方法将所有示例转换成一个向量。我们能否使用 fastText 向量来改进之前的结果？每篇评论都有 300 个特征。

```
import io
import numpy as np
import pandas as pd
from nltk.tokenize import word_tokenize
from sklearn import linear_model
from sklearn import metrics
from sklearn import model_selection
from sklearn.feature_extraction.text import TfidfVectorizer
def load_vectors(fname):
    fin = io.open(
        fname,
        'r',
        encoding='utf-8',
        newline='\n',
        errors='ignore'
    )
    n, d = map(int, fin.readline().split())
    data = {}
    for line in fin:
        tokens = line.rstrip().split(' ')
        data[tokens[0]] = list(map(float, tokens[1:]))
    return data

def sentence_to_vec(s, embedding_dict, stop_words, tokenizer):

if __name__ == "__main__":
```

```
    df = pd.read_csv("../input/imdb.csv")
    df.sentiment = df.sentiment.apply(
        lambda x: 1 if x == "positive" else 0
    )
    df = df.sample(frac=1).reset_index(drop=True)
    print("Loading embeddings")
    embeddings = load_vectors("../input/crawl-300d-2M.vec")
    print("Creating sentence vectors")
    vectors = []
    for review in df.review.values:
        vectors.append(
            sentence_to_vec(
                s = review,
                embedding_dict = embeddings,
                stop_words = [],
                tokenizer = word_tokenize
            )
        )
        vectors = np.array(vectors)
        y = df.sentiment.values
        kf = model_selection.StratifiedKFold(n_splits=5)
        for fold_, (t_, v_) in enumerate(kf.split(X=vectors, y=y)):
            print(f"Training fold: {fold_}")
        xtrain = vectors[t_, :]
        ytrain = y[t_]
        xtest = vectors[v_, :]
        ytest = y[v_]
        model = linear_model.LogisticRegression()
        model.fit(xtrain, ytrain)
        preds = model.predict(xtest)
        accuracy = metrics.accuracy_score(ytest, preds)
        print(f"Accuracy = {accuracy}")
        print("")
```

这将得到如下结果：

```
Loading embeddings
Creating sentence vectors

Training fold: 0
Accuracy = 0.8619

Training fold: 1
Accuracy = 0.8661
```

```
Training fold: 2
Accuracy = 0.8544

Training fold: 3
Accuracy = 0.8624

Training fold: 4
Accuracy = 0.8595
```

Wow！真是出乎意料。我们所做的一切都是为了使用 FastText 嵌入。试着把嵌入式换成 GloVe，看看会发生什么。我把它作为一个练习留给大家。

当我们谈论文本数据时，我们必须牢记一件事。文本数据与时间序列数据非常相似。如图 2 所示，我们评论中的任何样本都是在不同时间戳上按递增顺序排列的标记序列，每个标记都可以表示为一个向量/嵌入。

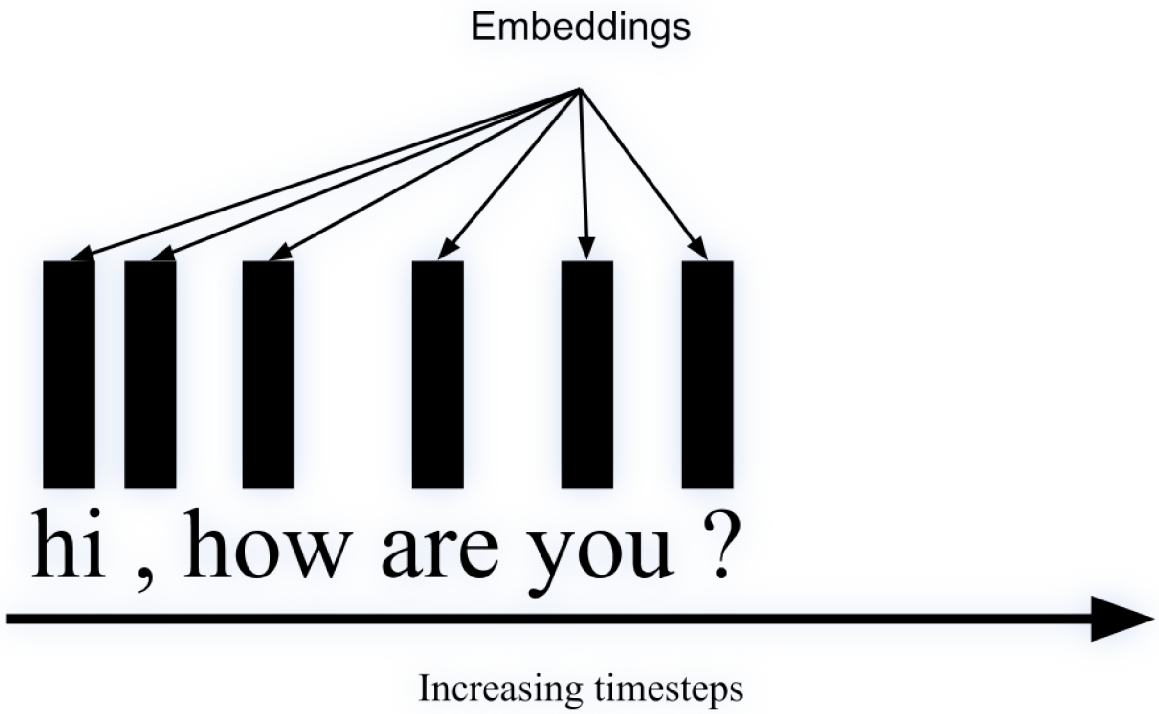

图 2：将标记表示为嵌入，并将其视为时间序列

这意味着我们可以使用广泛用于时间序列数据的模型，例如长短期记忆（LSTM）或门控递归单元（GRU），甚至卷积神经网络（CNN）。让我们看看如何在该数据集上训练一个简单的双向 LSTM 模型。

首先，我们将创建一个项目。你可以随意给它命名。然后，我们的第一步将是分割数据进行交叉验证。

```
import pandas as pd
from sklearn import model_selection
if __name__ == "__main__":
    df = pd.read_csv("../input/imdb.csv")
    df.sentiment = df.sentiment.apply(
        lambda x: 1 if x == "positive" else 0
    )
    df["kfold"] = -1
    df = df.sample(frac=1).reset_index(drop=True)
    y = df.sentiment.values
    kf = model_selection.StratifiedKFold(n_splits=5)
    for f, (t_, v_) in enumerate(kf.split(X=df, y=y)):
        df.loc[v_, 'kfold'] = f
        df.to_csv("../input/imdb_folds.csv", index=False)
```

将数据集划分为多个折叠后，我们就可以在 dataset.py 中创建一个简单的数据集类。数据集类会返回一个训练或验证数据样本。

```
import torch
class IMDBDataset:
def __init__(self, reviews, targets):
    self.reviews = reviews
    self.target = targets
def __len__(self):
    return len(self.reviews)
def __getitem__(self, item):
    review = self.reviews[item, :]
    target = self.target[item]
    return {
        "review": torch.tensor(review, dtype=torch.long),
        "target": torch.tensor(target, dtype=torch.float)
    }
```

完成数据集分类后，我们就可以创建 lstm.py，其中包含我们的 LSTM 模型

```
import torch
import torch.nn as nn
class LSTM(nn.Module):
    def __init__(self, embedding_matrix):
        super(LSTM, self).__init__()
```

```
        num_words = embedding_matrix.shape[0]
        embed_dim = embedding_matrix.shape[1]
        self.embedding = nn.Embedding(
            num_embeddings=num_words,
            embedding_dim=embed_dim)
        self.embedding.weight = nn.Parameter(
            torch.tensor(
                embedding_matrix,
                dtype=torch.float32
            )
        )
        self.embedding.weight.requires_grad = False
        self.lstm = nn.LSTM(
            embed_dim,
            128,
            bidirectional=True,
            batch_first=True,
        )
        self.out = nn.Linear(512, 1)

        def forward(self, x):
            x = self.embedding(x)
            x, _ = self.lstm(x)
            avg_pool = torch.mean(x, 1)
            max_pool, _ = torch.max(x, 1)
            out = torch.cat((avg_pool, max_pool), 1)
            out = self.out(out)
            return out
```

现在，我们创建 engine.py，其中包含训练和评估函数。

```
import torch
import torch.nn as nn
def train(data_loader, model, optimizer, device):
    model.train()
    for data in data_loader:
        reviews = data["review"]
        targets = data["target"]
        reviews = reviews.to(device, dtype=torch.long)
        targets = targets.to(device, dtype=torch.float)
        optimizer.zero_grad()
        predictions = model(reviews)
        loss = nn.BCEWithLogitsLoss()(
            predictions,
            targets.view(-1, 1)
```

```
            )
            loss.backward()
            optimizer.step()

    def evaluate(data_loader, model, device):
        final_predictions = []
        final_targets = []
        model.eval()
        with torch.no_grad():
            for data in data_loader:
                reviews = data["review"]
                targets = data["target"]
                reviews = reviews.to(device, dtype=torch.long)
                targets = targets.to(device, dtype=torch.float)
                predictions = model(reviews)
                predictions = predictions.cpu().numpy().tolist()
                targets = data["target"].cpu().numpy().tolist()
                final_predictions.extend(predictions)
                final_targets.extend(targets)
                return final_predictions, final_targets
```

这些函数将在 train.py 中为我们提供帮助，该函数用于训练多个折叠。

```
import io
import torch
import numpy as np
import pandas as pd
import tensorflow as tf
from sklearn import metrics
import config
import dataset
import engine
import lstm

def load_vectors(fname):
    fin = io.open(
        fname,
        'r',
        encoding='utf-8',
        newline='\n',
        errors='ignore'
    )
    n, d = map(int, fin.readline().split())
    data = {}
    for line in fin:
```

```
            tokens = line.rstrip().split(' ')
            data[tokens[0]] = list(map(float, tokens[1:]))
    return data

def create_embedding_matrix(word_index, embedding_dict):
    embedding_matrix = np.zeros((len(word_index) + 1, 300))
    for word, i in word_index.items():
        if word in embedding_dict:
            embedding_matrix[i] = embedding_dict[word]
    return embedding_matrix

def run(df, fold):
    train_df = df[df.kfold != fold].reset_index(drop=True)
    valid_df = df[df.kfold == fold].reset_index(drop=True)
    print("Fitting tokenizer")
    tokenizer = tf.keras.preprocessing.text.Tokenizer()
    tokenizer.fit_on_texts(df.review.values.tolist())
    xtrain = tokenizer.texts_to_sequences(train_df.review.values)
    xtest = tokenizer.texts_to_sequences(valid_df.review.values)
    xtrain = tf.keras.preprocessing.sequence.pad_sequences(
        xtrain, maxlen=config.MAX_LEN
    )
    xtest = tf.keras.preprocessing.sequence.pad_sequences(
        xtest, maxlen=config.MAX_LEN
    )
    train_dataset = dataset.IMDBDataset(
        reviews=xtrain,
        targets=train_df.sentiment.values
    )
    train_data_loader = torch.utils.data.DataLoader(
        train_dataset,
        batch_size=config.TRAIN_BATCH_SIZE,
        num_workers=2
    )
    valid_dataset = dataset.IMDBDataset(
        reviews=xtest,
        targets=valid_df.sentiment.values
    )
    valid_data_loader = torch.utils.data.DataLoader(
        valid_dataset,
        batch_size=config.VALID_BATCH_SIZE,
        num_workers=1
    )
    print("Loading embeddings")
    embedding_dict = load_vectors("../input/crawl-300d-2M.vec")
    embedding_matrix = create_embedding_matrix(
```

```
            tokenizer.word_index, embedding_dict
        )
        device = torch.device("cuda")
        model = lstm.LSTM(embedding_matrix)
        model.to(device)
        optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
        print("Training Model")
        best_accuracy = 0
        early_stopping_counter = 0
        for epoch in range(config.EPOCHS):
            engine.train(train_data_loader, model, optimizer, device)
            outputs, targets = engine.evaluate(
                valid_data_loader, model, device
            )
            outputs = np.array(outputs) >= 0.5
            accuracy = metrics.accuracy_score(targets, outputs)
            print(f"FOLD:{fold}, Epoch: {epoch}, Accuracy Score = {accuracy}")
            if accuracy > best_accuracy:
                best_accuracy = accuracy
            else:
                early_stopping_counter += 1
                if early_stopping_counter > 2:
                    break
    if __name__ == "__main__":
        df = pd.read_csv("../input/imdb_folds.csv")
        run(df, fold=0)
        run(df, fold=1)
        run(df, fold=2)
        run(df, fold=3)
        run(df, fold=4)
```

最后是 config.py。

```
MAX_LEN = 128
TRAIN_BATCH_SIZE = 16
VALID_BATCH_SIZE = 8
EPOCHS = 10
```

让我们看看输出：

```
FOLD:0, Epoch: 3, Accuracy Score = 0.9015
FOLD:1, Epoch: 4, Accuracy Score = 0.9007
FOLD:2, Epoch: 3, Accuracy Score = 0.8924
FOLD:3, Epoch: 2, Accuracy Score = 0.9
FOLD:4, Epoch: 1, Accuracy Score = 0.878
```

这是迄今为止我们获得的最好成绩。 请注意，我只显示了每个折叠中精度最高的Epoch。

你一定已经注意到，我们使用了预先训练的嵌入和简单的双向 LSTM。 如果你想改变模型，你可以只改变 lstm.py 中的模型并保持一切不变。 这种代码只需要很少的实验改动，并且很容易理解。 例如，您可以自己学习嵌入而不是使用预训练的嵌入，您可以使用其他一些预训练的嵌入，您可以组合多个预训练的嵌入，您可以使用GRU，您可以在嵌入后使用空间dropout，您可以添加GRU LSTM 层之后，您可以添加两个 LSTM 层，您可以进行 LSTM-GRU-LSTM 配置，您可以用卷积层替换 LSTM 等，而无需对代码进行太多更改。 我提到的大部分内容只需要更改模型类。

当您使用预训练的嵌入时，尝试查看有多少单词无法找到嵌入以及原因。 预训练嵌入的单词越多，结果就越好。 我向您展示以下未注释的 (!) 函数，您可以使用它为任何类型的预训练嵌入创建嵌入矩阵，其格式与 glove 或 fastText 相同（可能需要进行一些更改）。

```
def load_embeddings(word_index, embedding_file, vector_length=300):
    max_features = len(word_index) + 1
    words_to_find = list(word_index.keys())
    more_words_to_find = []
    for wtf in words_to_find:
        more_words_to_find.append(wtf)
        more_words_to_find.append(str(wtf).capitalize())
    more_words_to_find = set(more_words_to_find)
def get_coefs(word, *arr):
    return word, np.asarray(arr, dtype='float32')

embeddings_index = dict(
    get_coefs(*o.strip().split(" "))
    for o in open(embedding_file)
    if o.split(" ")[0]
    in more_words_to_find
    and len(o) > 100
)

embedding_matrix = np.zeros((max_features, vector_length))
for word, i in word_index.items():
    if i >= max_features:
        continue
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is None:
```

```
            embedding_vector = embeddings_index.get(
                str(word).capitalize()
            )
        if embedding_vector is None:
            embedding_vector = embeddings_index.get(
                str(word).upper()
            )
        if (embedding_vector is not None
            and len(embedding_vector) == vector_length):
            embedding_matrix[i] = embedding_vector
        return embedding_matrix
```

阅读并运行上面的函数，看看发生了什么。 该函数还可以修改为使用词干词或词形还原词。 最后，您希望训练语料库中的未知单词数量最少。 另一个技巧是学习嵌入层，即使其可训练，然后训练网络。

到目前为止，我们已经为分类问题构建了很多模型。 然而，现在是布偶时代，越来越多的人转向基于变形金刚的模型。 基于 Transformer 的网络能够处理本质上长期的依赖关系。 LSTM 仅当它看到前一个单词时才查看下一个单词。 变压器的情况并非如此。 它可以同时查看整个句子中的所有单词。 因此，另一个优点是它可以轻松并行化并更有效地使用 GPU。

Transformers 是一个非常广泛的话题，有太多的模型：BERT、RoBERTa、XLNet、XLM-RoBERTa、T5 等。我将向您展示一种可用于所有这些模型（T5 除外）进行分类的通用方法 我们一直在讨论的问题。 请注意，这些变压器需要训练它们所需的计算能力。 因此，如果您没有高端系统，与基于 LSTM 或 TF-IDF 的模型相比，训练模型可能需要更长的时间。

我们要做的第一件事是创建一个配置文件。

```
import transformers
MAX_LEN = 512
TRAIN_BATCH_SIZE = 8
VALID_BATCH_SIZE = 4
EPOCHS = 10

BERT_PATH = "../input/bert_base_uncased/"
MODEL_PATH = "model.bin"
TRAINING_FILE = "../input/imdb.csv"
TOKENIZER = transformers.BertTokenizer.from_pretrained(
    BERT_PATH,
    do_lower_case=True
)
```

这里的配置文件是我们定义分词器和其他我们想要经常更改的参数的唯一地方 —— 这样我们就可以做很多实验而不需要进行大量更改。

下一步是构建数据集类。

```
import config
import torch
class BERTDataset:
    def __init__(self, review, target):
        self.review = review
        self.target = target
        self.tokenizer = config.TOKENIZER
        self.max_len = config.MAX_LEN
    def __len__(self):
        return len(self.review)
    def __getitem__(self, item):
        review = str(self.review[item])
        review = " ".join(review.split())
        inputs = self.tokenizer.encode_plus(
            review,
            None,
            add_special_tokens=True,
            max_length=self.max_len,
            pad_to_max_length=True,
        )
        ids = inputs["input_ids"]
        mask = inputs["attention_mask"]
        token_type_ids = inputs["token_type_ids"]
        return {
            "ids": torch.tensor(
                ids, dtype=torch.long
            ),
            "mask": torch.tensor(
                mask, dtype=torch.long
            ),
            "token_type_ids": torch.tensor(
                token_type_ids, dtype=torch.long
            ),
            "targets": torch.tensor(
                self.target[item], dtype=torch.float
            )
        }
```

现在我们来到了该项目的核心，即模型。

```
import config
import transformers
import torch.nn as nn
```

```
class BERTBaseUncased(nn.Module):
    def __init__(self):
        super(BERTBaseUncased, self).__init__()
        self.bert = transformers.BertModel.from_pretrained(
            config.BERT_PATH
        )
        self.bert_drop = nn.Dropout(0.3)
        self.out = nn.Linear(768, 1)
    def forward(self, ids, mask, token_type_ids):
        hidden state
        _, o2 = self.bert(
            ids,
            attention_mask=mask,
            token_type_ids=token_type_ids
        )
        bo = self.bert_drop(o2)
        output = self.out(bo)
        return output
```

该模型返回单个输出。 我们可以使用带有 logits 的二元交叉熵损失，它首先应用 sigmoid，然后计算损失。 这是在engine.py 中完成的。

```
import torch
import torch.nn as nn
def loss_fn(outputs, targets):
    return nn.BCEWithLogitsLoss()(outputs, targets.view(-1, 1))
def train_fn(data_loader, model, optimizer, device, scheduler):
    model.train()
    for d in data_loader:
        ids = d["ids"]
        token_type_ids = d["token_type_ids"]
        mask = d["mask"]
        targets = d["targets"]
        ids = ids.to(device, dtype=torch.long)
        token_type_ids = token_type_ids.to(device, dtype=torch.long)
        mask = mask.to(device, dtype=torch.long)
        targets = targets.to(device, dtype=torch.float)
        optimizer.zero_grad()
        outputs = model(
            ids=ids,
            mask=mask,
            token_type_ids=token_type_ids
        )
        loss = loss_fn(outputs, targets)
        loss.backward()
```

```
            optimizer.step()
            scheduler.step()

    def eval_fn(data_loader, model, device):
        model.eval()
        fin_targets = []
        fin_outputs = []
        with torch.no_grad():
            for d in data_loader:
                ids = d["ids"]
                token_type_ids = d["token_type_ids"]
                mask = d["mask"]
                targets = d["targets"]
                ids = ids.to(device, dtype=torch.long)
                token_type_ids = token_type_ids.to(device, dtype=torch.long)
                mask = mask.to(device, dtype=torch.long)
                targets = targets.to(device, dtype=torch.float)
                outputs = model(
                    ids=ids,
                    mask=mask,
                    token_type_ids=token_type_ids
                )
                targets = targets.cpu().detach()
                fin_targets.extend(targets.numpy().tolist())
                outputs = torch.sigmoid(outputs).cpu().detach()
                fin_outputs.extend(outputs.numpy().tolist())
        return fin_outputs, fin_targets
```

最后，我们准备好训练了。 我们来看看训练脚本吧！

```
    import config
    import dataset
    import engine
    import torch
    import pandas as pd
    import torch.nn as nn
    import numpy as np
    from model import BERTBaseUncased
    from sklearn import model_selection
    from sklearn import metrics
    from transformers import AdamW
    from transformers import get_linear_schedule_with_warmup
    def train():
        dfx = pd.read_csv(config.TRAINING_FILE).fillna("none")
        dfx.sentiment = dfx.sentiment.apply(
```

```
        lambda x: 1 if x == "positive" else 0
    )
    df_train, df_valid = model_selection.train_test_split(
        dfx,
        test_size=0.1,
        random_state=42,
        stratify=dfx.sentiment.values
    )
    df_train = df_train.reset_index(drop=True)
    df_valid = df_valid.reset_index(drop=True)
    train_dataset = dataset.BERTDataset(
        review=df_train.review.values,
        target=df_train.sentiment.values
    )
    train_data_loader = torch.utils.data.DataLoader(
        train_dataset,
        batch_size=config.TRAIN_BATCH_SIZE,
        num_workers=4
    )
    valid_dataset = dataset.BERTDataset(
        review=df_valid.review.values,
        target=df_valid.sentiment.values
    )
    valid_data_loader = torch.utils.data.DataLoader(
        valid_dataset,
        batch_size=config.VALID_BATCH_SIZE,
        num_workers=1
    )
    device = torch.device("cuda")
    model = BERTBaseUncased()
    model.to(device)
    param_optimizer = list(model.named_parameters())
    no_decay = ["bias", "LayerNorm.bias", "LayerNorm.weight"]
    optimizer_parameters = [
        {
            "params": [
                p for n, p in param_optimizer if
                not any(nd in n for nd in no_decay)
            ],
            "weight_decay": 0.001,
        },
        {
            "params": [
                p for n, p in param_optimizer if
                any(nd in n for nd in no_decay)
            ],
```

```
            "weight_decay": 0.0,
        }]
    num_train_steps = int(
        len(df_train) / config.TRAIN_BATCH_SIZE * config.EPOCHS
    )
    optimizer = AdamW(optimizer_parameters, lr=3e-5)
    scheduler = get_linear_schedule_with_warmup(
        optimizer,
        num_warmup_steps=0,
        num_training_steps=num_train_steps
    )
    model = nn.DataParallel(model)
    best_accuracy = 0
    for epoch in range(config.EPOCHS):
        engine.train_fn(
            train_data_loader, model, optimizer, device, scheduler
        )
        outputs, targets = engine.eval_fn(
            valid_data_loader, model, device
        )

        outputs = np.array(outputs) >= 0.5
        accuracy = metrics.accuracy_score(targets, outputs)
        print(f"Accuracy Score = {accuracy}")
        if accuracy > best_accuracy:
            torch.save(model.state_dict(), config.MODEL_PATH)
            best_accuracy = accuracy

if __name__ == "__main__":
    train()
```

乍一看可能看起来很多，但一旦您了解了各个组件，就不再那么简单了。 您只需更改几行代码即可轻松将其更改为您想要使用的任何其他变压器模型。

该模型的准确率为 93%！ 哇！ 这比任何其他模型都要好得多。 但是这值得吗？

我们使用 LSTM 能够实现 90% 的目标，而且它们更简单、更容易训练并且推理速度更快。 通过使用不同的数据处理或调整层、节点、dropout、学习率、更改优化器等参数，我们可以将该模型改进一个百分点。然后我们将从 BERT 中获得约 2% 的收益。 另一方面，BERT 的训练时间要长得多，参数很多，而且推理速度也很慢。 最后，您应该审视自己的业务并做出明智的选择。 不要仅仅因为 BERT“酷”而选择它。

必须注意的是，我们在这里讨论的唯一任务是分类，但将其更改为回归、多标签或多类只需要更改几行代码。 例如，多类分类设置中的同一问题将有多个输出和交叉熵损失。 其他一切都应该保持不变。 自然语言处理非常庞大，我们只讨论了其中的一小部分。 显然，这是一个很大的比例，因为大多数工业模型都是分类或回归模型。 如果我开始详细写所有内容，我最终可能会写几百页，这就是为什么我决定将所有内容包含在一本单独的书中：接近（几乎）任何 NLP 问题！