期中作业：IMDB电影评论文本分类

姓名	学号	专业（方向）
李雪堃	19335109	计算机科学与技术（超算）

---

(0) 项目简介

目录结构：

```
.
├── assets
│   ├── IMDB Dataset.csv
│   ├── processed_dataset.csv
│   └── stopwords-en.txt
├── checkpoint
│   ├── tf-idf.hdf5
│   ├── tf.hdf5
│   └── w2v.hdf5
├── images
│   ├── tf_accuracy.png
│   ├── tf_loss.png
│   ├── tf-idf_accuracy.png
│   ├── tf-idf_loss.png
│   ├── w2v_accuracy.backup.png
│   ├── w2v_accuracy.png
│   ├── w2v_loss.backup.png
│   └── w2v_loss.png
├── models
│   ├── GoogleNews-vectors-negative300.bin.gz
│   └── idf.txt
├── dataset.py
├── main.py
```

```
├── model.py
├── word2vec.py
├── tfidf.py
└── requirements.txt
```

- assets 目录下存放原始数据集、清洗后的数据集、以及英文停用词文件（来源：https://github.com/stopwords-iso/stopwords-en）
- checkpoint 目录下存放的是训练好的 checkpoint
- images 目录存放的是模型训练好后对 accuracy 和 loss 的评估
- models 目录下存放的是预训练的词向量 word2vec（来源：https://code.google.com/archive/p/word2vec，由于不要提交预训练的词向量文件，所以目录下没有 GoogleNews-vectors-negative300.bin.gz），以及 idf.txt，这是为了加快计算 tf-idf 的速度，所以预先计算好并保存，可以删除掉，这样程序在运行时会自动重新计算 idf
- 源代码在项目根目录下
    - dataset.py：对原始数据集进行清洗和预处理
    - tfidf.py：计算 TF、IDF、TF-IDF，并生成词嵌入矩阵
    - word2vec.py：从 GoogleNews-vectors-negative300.bin.gz 加载预训练的词向量，并生成词嵌入矩阵
    - model.py：定义神经网络模型（包括一个 1D CNN、一个普通的 NN）
    - main.py：对数据集进行清洗和预处理，加载词嵌入矩阵，训练神经网络并评估结果
    - requirements.txt：conda 导出的所需要的包，可以使用 conda 直接安装里面的包 `conda install --file requirements.txt`

(1) 运行和测试方法

main.py 提供一个参数 `--feed / -f`，用于指定将何种词向量（特征）表示作为神经网络的输入，提供下面三个选项：

- `w2v`：使用预训练的词向量 word2vec
- `tf`：使用 tf
- `tf-idf`：使用 tf-idf

比如，下面的命令是用 word2vec 来作为神经网络的输入：

```
python3 main.py --feed w2v
```

word2vec 会用来训练一个一维的卷积神经网络，而 tf 和 tf-idf 会用来训练一个简单的 NN。

(2) 核心代码解释

(2.0) 数据清洗和处理

```python
def strip_html_tags(review: str) -> str:
  return BeautifulSoup(review, "lxml").get_text()

def remove_special_chars(review: str) -> str:
  # replace all the special chars and digits in the review
  return re.sub(r"[^A-Za-z]+" , ' ', review)

def tokenize(review: str) -> str:
  # just tokenize by white space
  token_list = review.strip().split()
  return ' '.join(token_list)

def remove_stopwords(review: str) -> str:
  # load stopwords
  stopwords_list = []
  file = open(stopwords_file, 'r')
  for line in file:
    line = line.strip()
    stopwords_list.append(line)
  file.close()
  # remove stopwords from review
  review_filtered = []
  for word in review.split():
    if word not in stopwords_list:
      review_filtered.append(word)
  # join words by space
  return ' '.join(review_filtered)

def preprocess(review):
  review = strip_html_tags(review) # strip html tags
  review = remove_special_chars(review) # remove all the special chars including digits and punctuation, leaving letters only
  review = review.lower() # to lower case
  review = tokenize(review) # tokenize
  review = remove_stopwords(review) # remove stopwords
  return review

def clean_dataset():
  df = pd.read_csv(imdb_dataset)
  print(df)
  for i in tqdm(range(len(df.index)), desc="Loading and preprocessing raw dataset", bar_format='{l_bar}{bar:10}{r_bar}{bar:-10b}'):
    df['review'].iloc[i] = preprocess(df['review'].iloc[i])
    df['sentiment'].iloc[i] = POSITIVE if df['sentiment'].iloc[i] == 'positive' else NEGATIVE
  df.columns = ['review', 'label']
  df.to_csv("./assets/processed_dataset.csv", index=False)
  return df
```

- `strip_html_tag` 使用 `BeautifulSoup` 来去除文本中的 HTML 标签
- `remove_sepcial_chars` 使用正则去除文本中所有的非字母字符
- `tokenize` 简单地将文本按照空格分词
- `remove_stop_words` 从 `assets/stopwords-en.txt` 加载停用词，去除评论中的停用词
- `preprocess` 接收一条评论，依次调用上面 4 个函数，中间还要把单词转换为小写，最后返回处理后的单条评论
- `clean_dataset` 读取原始的数据集，依次对每条评论进行预处理，并将标签转换为 1/0（positive 为 1、negative 为 0），最后处理后的文本保存为 .csv 文件，存放在 `assets` 文件夹下面

```python
def get_word_index(X):
  '''
    Establish word-freq dict wordbag, remove low-frequency (< 50) words
    Then, build the mapping from each word to its unique index
  '''
  word_bag = {}  # mapping from word to its total frequency in all the reviews
  for review in X:
    for word in review:
      if word not in word_bag.keys():
        word_bag[word] = 0
      else:
        word_bag[word] += 1

  word_bag_sorted = sorted(word_bag.items(), key=lambda x: x[1], reverse=True)
  # print(word_bag_sorted[:100])

  word_index = {} # mapping from word to its index
  index = 1
  for word, freq in word_bag_sorted:
    if freq < 50:
      break
    word_index[word] = index
    index += 1

  return word_index
```

`get_word_index` 函数首先建立一个 `word_bag`，`word_bag` 是一个字典，存储了文本所有的词到它出现的总频数之间的映射。然后将 `word_bag` 按照频数从大到小排序。最后从排序后的列表中，建立 `word_index`，舍弃掉低频词，不对低频词建立索引（由于评论有 5W 条，如果出现次数小于 50，说明这个词在一条评论中出现的概率大约不超过 1%，可以舍弃掉）。注意到索引是从 1 而不是从 0 开始的，我们将 0 号索引保留，用于表示低频词或未出现的词。

```python
def load_dataset(sequence_length):
  '''
    X is reviews list, X[i] is the token list of the i-th review
    Y is the corresponding labels list of reviews, Y[i] is the label of review X[i]
  '''
  # read X, Y from processed data set, convert them to numpy ndarray
  dataset = pd.read_csv("./assets/processed_dataset.csv")
  X = dataset['review'].iloc[:].to_numpy()
  Y = dataset['label'].iloc[:].to_numpy()

  for i in range(len(X)):
    X[i] = X[i].split()

  word_index = get_word_index(X)

  for i in range(len(X)):
    for j in range(len(X[i])):
      if X[i][j] in word_index.keys():
        X[i][j] = word_index[X[i][j]]
      else:
        X[i][j] = 0

  X = pad_sequences(X, maxlen=sequence_length, dtype=np.int32, value=0, padding='post', truncating='post')

  return X, Y, word_index
```

`load_dataset` 首先读取处理后的文本，X 为分词后的评论列表，Y 为标签。然后根据 X 来建立 `word_index`。

接着，我们需要对 X 进行词到索引的映射，遍历每条评论的每个词，将它替换为对应的索引（低频词则令其索引为 0）。

最后，需要对每条评论进行 padding 操作（这时每条评论已经表示为词索引了），注意到 `load_dataset` 接收一个参数 `sequence_length`，作为 padding 的固定长度。这里使用 `keras.prerprocessing.sequence` 中的 `pad_sequences` 函数，对每条评论，长度不足则补 0 到 `sequence_length`，长度超过则截断后面的词索引。

(2.1) 词嵌入矩阵的输入

首先是 TF、TF-IDF 的计算。

```
def get_tf(X, word_index):
  TF = {}
  for review_index in tqdm(range(len(X)), desc="Calculating tf", bar_format='{l_bar}{bar:10}{r_bar}{bar:-10b}'):
    TF[review_index] = {}
    review = X[review_index]
    for index_of_word in review:
      if index_of_word != 0:
        review = list(review)
        TF[review_index][index_of_word] = review.count(index_of_word) / len(review)
  return TF
```

`get_tf` 函数返回一个嵌套的字典，每条评论的索引（0 - 49999）对应一个字典，这个字典中评论的每个单词的索引对应着它的 TF。

```
def get_idf(X, word_index):
  IDF = {}
  if not os.path.exists("./models/idf.txt"):
    for index_of_word in tqdm(word_index.values(), desc="Calculating idf", bar_format='{l_bar}{bar:10}{r_bar}{bar:-10b}'):
      IDF[index_of_word] = 0
      for review in X:
        if index_of_word in review:
          IDF[index_of_word] += 1
    for index_of_word in IDF.keys():
      IDF[index_of_word] = np.log10(len(X) / (1 + IDF[index_of_word]))
    idf_file = open("./models/idf.txt", 'w')
    for index_of_word, idf in IDF.items():
      idf_file.write("{}\t{}\n".format(index_of_word, idf))
    idf_file.close()
  else:
    idf_file = open("./models/idf.txt", 'r')
    for line in idf_file:
      line = line.strip().split()
      index_of_word = int(line[0])
      idf = float(line[1])
      IDF[index_of_word] = idf
    idf_file.close()
  return IDF
```

`get_idf` 函数首先判断 `models` 文件夹下有没有计算好的 idf.txt，没有则计算并保存到 idf.txt。

```
def get_tfidf(TF, IDF):
  TF_IDF = {}
  for review_index in tqdm(TF.keys(), desc="Calculating tf-idf", bar_format='{l_bar}{bar:10}{r_bar}{bar:-10b}'):
    TF_IDF[review_index] = {}
    for index_of_word in TF[review_index].keys():
      TF_IDF[review_index][index_of_word] = TF[review_index][index_of_word] * IDF[index_of_word]
  return TF_IDF
```

`get_tfidf` 函数直接根据 TF 和 IDF 来计算 TF-IDF，返回的是一个嵌套的字典，字典的键分别是评论的索引，以及评论中单词的索引。

```
def get_tf_embedding(X, word_index):
  TF = get_tf(X, word_index)
  tf_embedding_matrix = np.zeros((len(TF), len(word_index)+1))
  for review_index in TF.keys():
    for index_of_word in TF[review_index].keys():
      tf_embedding_matrix[review_index][index_of_word] = TF[review_index][index_of_word]
  normalize(X=tf_embedding_matrix, norm='l2', copy=False, return_norm=False)
  return tf_embedding_matrix
```

`get_tf_embedding` 函数将根据 TF 生成一个词嵌入矩阵，每条评论对应着一个向量，向量的维度是总的词的个数，元素是对应的词的 TF，如果是低频词则为 0。最后还要标准化每个向量。

`get_tfidf_embedding` 函数与之相同，这里不赘述了。

```
def get_w2v_embedding(word_index):
  w2v_model = KeyedVectors.load_word2vec_format("./models/GoogleNews-vectors-negative300.bin.gz", binary=True, limit=2000000)
  w2v_embedding_matrix = np.zeros((len(word_index) + 1, w2v_model.vector_size))
  for word, index in word_index.items():
    try:
      w2v_embedding_matrix[index] = w2v_model[word]
    except:
      continue  # if word is not in w2v_model, then it's embedding is set to zero
  normalize(X=w2v_embedding_matrix, norm='l2', copy=False, return_norm=False)
  return w2v_embedding_matrix
```

`get_w2v_embedding` 首先 load 预训练的词向量，选择频率最高的两百万个。然后遍历所有的单词，获得它对应的 word2vec，放到词嵌入矩阵中，如果在加载的 word2vec 模型中没有这个单词，那么它的词向量设置为 0。最后，还需要标准化整个词嵌入矩阵。

(2.2) 神经网络的结构

首先是普通的前馈神经网络，一个 Dense 层作为输入，中间有两层全连接层，最后是一层输出一维结果的 Dense 层，因为我们的任务是二分类，所以输出的 units 是 1。Dense 层之间有 Dropout，用于丢弃部分神经元，每次只有部分神经网络结构得到更新，防止过拟合。

```
def nn(vocab_size):
  model = models.Sequential()
  model.add(Dense(256, activation='relu', input_shape=(vocab_size,)))
  model.add(Dropout(0.2))
  model.add(Dense(128, activation='relu'))
  model.add(Dropout(0.2))
  model.add(Dense(64, activation='relu'))
  model.add(Dense(1, activation='sigmoid'))
  model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['accuracy'])
  model.summary()
  return model
```

然后是 1 维的卷积神经网络。我们这里使用 Embedding 层接收输入，`input_dim` 是输入的词向量的个数，即训练集总的词的个数（去掉低频词）；`output_dim` 是 Embedding 层输出的向量的维度，也就是 word2vec 的维度；`input_length` 是每条评论的词的个数，我们前面做了 padding 操作，所以这里设置为 padding 的长度 `sequence_length`，在代码中，我设置为 200；`weights` 输入的是权重矩阵，也就是我们的词嵌入矩阵；最后，由于我们使用预训练的词向量，所以不希望 Embedding 层在训练过程中更新词向量，所以 `trainable` 参数设置为 False。

因此，Embedding 层就只是用来接收输入的参数，而不会在训练过程中进行更新。

然后，添加两个 Conv1D 的卷积层，并其后做 MaxPooling 操作。在最后的全连接层前，设置 Flatten 层，用于拼接 MaxPooling 后的 feature_map。最后是两个全连接层，输出 1 个 unit。

```
def cnn(embedding_matrix, embedding_dim, sequence_length, vocab_size):
  model = Sequential()
  model.add(Embedding(input_dim=vocab_size, output_dim=embedding_dim, input_length=sequence_length, \
                      weights=[embedding_matrix], trainable=False))

  model.add(Conv1D(embedding_dim, 3, activation='relu', padding='same'))
  model.add(MaxPool1D(2))
  model.add(Dropout(0.2))

  model.add(Conv1D(embedding_dim // 2, 3, activation='relu', padding='same'))
  model.add(MaxPool1D(2))
  model.add(Dropout(0.2))

  model.add(Flatten())
  model.add(Dense(256, activation='relu'))
  model.add(Dense(1, activation='sigmoid'))

  model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['accuracy'])
  model.summary()
  return model
```

(2.3) 神经网络的训练

```
# split dataset
# 0     - 29999: train set
# 30000 - 39999: validation set
# 40000 - 50000: test set
X_train, Y_train = X[0:30000], Y[0:30000]
X_valid, Y_valid = X[30000:40000], Y[30000:40000]
X_test, Y_test   = X[40000:50000], Y[40000:50000]

# get w2v embedding matrix
w2v_embedding_matrix = get_w2v_embedding(word_index)

# create text cnn model
model = cnn(w2v_embedding_matrix, w2v_embedding_matrix.shape[1], sequence_length, len(word_index)+1)
checkpoint = ModelCheckpoint('./checkpoint/w2v.hdf5', monitor='val_accuracy', verbose=1, save_best_only=True, mode='max')

# compile and train model
print("Traning Model...")
history = model.fit(X_train, Y_train, batch_size=30, epochs=4, verbose=1, callbacks=[checkpoint], validation_data=(X_valid, Y_valid))

# evaluate model
print("Evaluate on test data")
results = model.evaluate(X_test, Y_test, batch_size=50)
print("test loss, test acc:", results)

# visualize training history
history_plot(history, 'w2v')
```

训练时，先对读取的数据集 X 和标签 Y 进行分割，分割为训练集、验证集和测试集。

在加载 word2vec 词嵌入矩阵后，获取我们定义的 cnn 模型，注意到最后一个参数是 `len(word_index)+1` 而不是 `len(word_index)`，因为我们预留 0 号索引是低频词，但在 X 中没有去除低频词，只是将低频词的索引设置为 0。

然后设置 checkpoint，选取在验证集上准确率最高的模型进行保留。

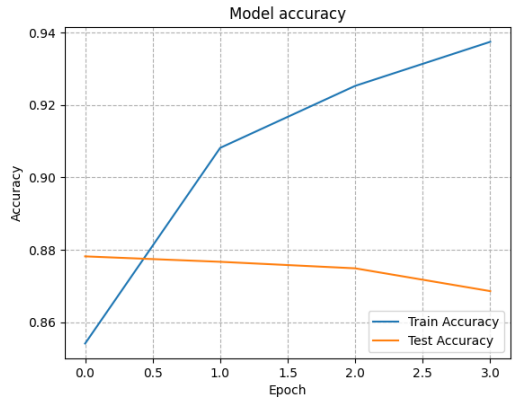
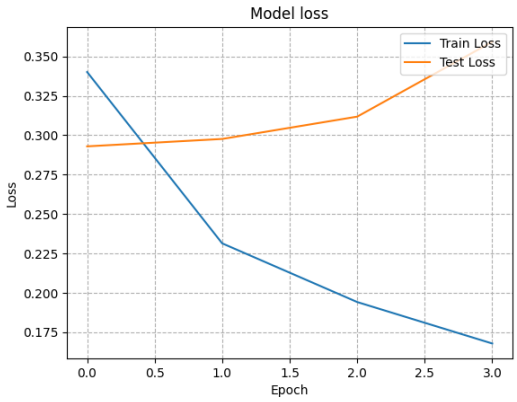
然后，对模型进行训练，batch_size 设置为 30，epochs 设置为 4 轮。输入测试集、验证集、以及它们对应的标签。

最后是模型评估，先在测试集上测试我们的模型，输出 loss 和 accuracy，再将训练过程中在训练集和测试集上的 loss 和 accuracy 画出来，保存为折线图。

(3) 结果和评估

(3.0) TF

训练集和测试集上的表现：

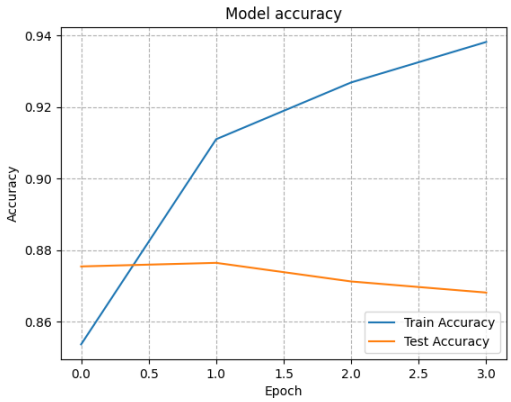
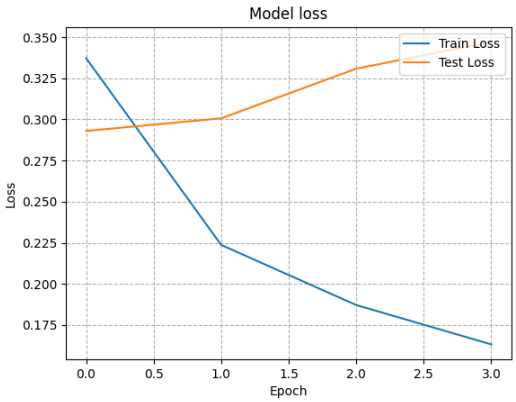
准确率	损失
	

测试集上的表现：

```
Epoch 1/4
469/469 [==============================] - 15s 30ms/step - loss: 0.4054 - accuracy: 0.8161 - val_loss: 0.2979 - val_accuracy: 0.8748
Epoch 2/4
469/469 [==============================] - 15s 31ms/step - loss: 0.2211 - accuracy: 0.9138 - val_loss: 0.3185 - val_accuracy: 0.8687
Epoch 3/4
469/469 [==============================] - 13s 28ms/step - loss: 0.1843 - accuracy: 0.9303 - val_loss: 0.3066 - val_accuracy: 0.8762
Epoch 4/4
469/469 [==============================] - 13s 27ms/step - loss: 0.1575 - accuracy: 0.9433 - val_loss: 0.3354 - val_accuracy: 0.8731
Evaluate on test data
200/200 [==============================] - 2s 9ms/step - loss: 0.3295 - accuracy: 0.8781
test loss, test acc: [0.3294684886932373, 0.8780999779701233]
```

(3.1) TF-IDF

训练集和测试集上的表现：

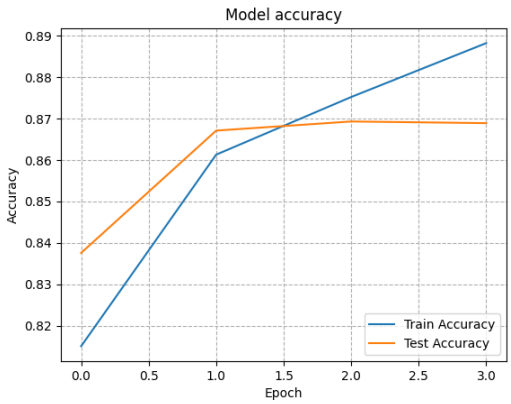
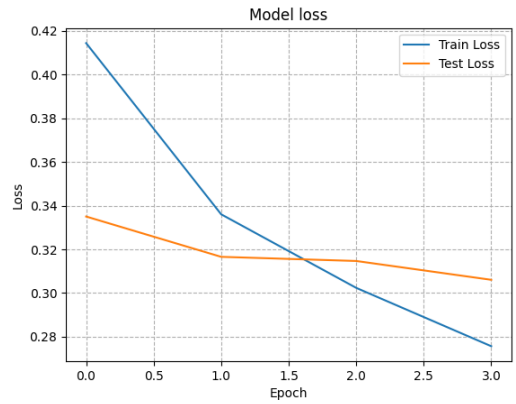
准确率	损失
	

测试集上的表现：

```
Epoch 1/4
469/469 [==============================] - 15s 30ms/step - loss: 0.4075 - accuracy: 0.8110 - val_loss: 0.3050 - val_accuracy: 0.8693
Epoch 2/4
469/469 [==============================] - 14s 29ms/step - loss: 0.2134 - accuracy: 0.9139 - val_loss: 0.3118 - val_accuracy: 0.8727
Epoch 3/4
469/469 [==============================] - 13s 27ms/step - loss: 0.1765 - accuracy: 0.9322 - val_loss: 0.3262 - val_accuracy: 0.8720
Epoch 4/4
469/469 [==============================] - 13s 28ms/step - loss: 0.1500 - accuracy: 0.9434 - val_loss: 0.3322 - val_accuracy: 0.8678
Evaluate on test data
200/200 [==============================] - 1s 7ms/step - loss: 0.3172 - accuracy: 0.8768
test loss, test acc: [0.3172127306461334, 0.876800000667572]
```

(3.2) word2vec

训练集和测试集上的表现：

准确率	损失
	

测试集上的表现：

```
Epoch 1/4
1000/1000 [==============================] - 125s 124ms/step - loss: 0.4827 - accuracy: 0.7572 - val_loss: 0.3734 - val_accuracy: 0.8376

Epoch 00001: val_accuracy improved from -inf to 0.83760, saving model to ./checkpoint/w2v.hdf5
Epoch 2/4
1000/1000 [==============================] - 122s 122ms/step - loss: 0.3305 - accuracy: 0.8626 - val_loss: 0.3285 - val_accuracy: 0.8671

Epoch 00002: val_accuracy improved from 0.83760 to 0.86710, saving model to ./checkpoint/w2v.hdf5
Epoch 3/4
1000/1000 [==============================] - 119s 119ms/step - loss: 0.3071 - accuracy: 0.8738 - val_loss: 0.3124 - val_accuracy: 0.8693

Epoch 00003: val_accuracy improved from 0.86710 to 0.86930, saving model to ./checkpoint/w2v.hdf5
Epoch 4/4
1000/1000 [==============================] - 120s 120ms/step - loss: 0.2671 - accuracy: 0.8922 - val_loss: 0.3186 - val_accuracy: 0.8689

Epoch 00004: val_accuracy did not improve from 0.86930
Evaluate on test data
200/200 [==============================] - 13s 63ms/step - loss: 0.3058 - accuracy: 0.8779
test loss, test acc: [0.3057701289653778, 0.8779000043869019]
```

可以看到，进行多轮训练之后，TF 和 TF-IDF 虽然在训练集上的 accuracy 在增加、 loss 在减少（最后减少到比较小的值），但是在验证集上的 accuracy 缓慢下降、loss 不断增加，一部分可能是过拟合的结果，一部分是因为 TF、TF-IDF 作为特征来说文本的关联性不够强。

对于 word2vec，在测试集和验证集上的 accuracy 都在不断增加，loss 不断减少（训练集最后的 loss 比 TF 和 TF-IDF 要高，这说明防止了过拟合），因为 word2vec 用来表示词语的话，词语之间的相关性较强，尤其是我们使用了预训练的词向量，可以防止过拟合，而且还能带入数据集之外的语义信息。

在测试集上，三种特征的 loss 都在 0.31 ~ 0.32，accuracy 在 0.87 ~ 0.88 之间，表现都比较好。

总的来说，word2vec 的表现比 TF 和 TF-IDF 更为优秀。

(4) 心得与体会

关于停用词的使用，我在网上看到了几篇国外的博客，说某些停用词例如 can't、don't、like 等，这些有可能会带有表达文本情感的关键信息，所以去除掉可能会对最后预测结果产生影响，尤其是二分类的情感分析，所以在情感分析中停用词需要按实际情况取舍一部分，我这里找的是 github 上一个停用词列表。

词向量最好使用预训练的，而且最好是用在大规模数据集上训练的词向量如 Glove、word2vec 等，词语的相关性比较准确，而且包含的语义信息更为丰富，往往可以取得更好的效果。自己训练的语义信息少，而且关联性不准确。

去除文本中的低频词也是必要的，否则词语数量太大，而且对于文本分类的问题，低频词一般对于文本类型的贡献非常小。

另外，神经网络的搭建令我非常头疼，刚开始各种奇怪的 API，然后在网上看了很多的资料和博客，查询了很多官方文档，对于之前害怕的神经网络，现在也基本上搞懂它的结构和原理了，并不是特别困难。卷积神经网络部分原本是想实现 TextCNN 的，但搭建起来之后，跑得非常慢，于是换成 1D 的卷积层，调整了网络的结构。

说一遍不如做一遍，这次大作业不仅帮我复习巩固了前面学到的基本的理论知识，还学习了神经网络相关的知识，掌握了基本的数据清洗和处理、搭建神经网络进行文本分类的流程，收获非常大，看到最后的准确率在 87%～88% 的时候，我非常开心、也非常满足，我也对 NLP 兴趣倍增。