# 机器学习与数据挖掘期末作业

# IMDB电影评论文本表征学习与情感分类

19335015 陈恩婷

## 概述

表征学习与文本分类都是目前在机器学习领域的常见应用。本实验通过尝试不同的文本表征学习算法，并将所得的文本嵌入应用于深度神经网络的训练，对IMDB电影评论文本进行情感分类，探究与比较了不同的表征学习算法与他们的情感分类效果，其中采用Word2vec模型时得到了最大测试集正确率88.49%。

关键词：表征学习、TF-IDF、Word2vec、文本情感分类、多层感知机、机器学习、数据挖掘

## 研究问题的背景与动机

### 表征学习

在机器学习领域，表征学习（Representation Learning）是一种将原始数据转换成为更容易被机器学习应用的数据的过程。表征学习的目的是对复杂的原始数据化繁为简，把原始数据提炼成更好的数据表达，使后续的任务事半功倍。表征学习可以用于很多下游的机器学习应用，包括分类、聚类、异常检测等等。

### 文本情感分类

在机器学习中，文本情感分类是一类有监督学习，它主要的任务是对输入的文本，确定它情感态度上是积极还是消极（或者中立）。文本情感分类任务有很多应用，包括用于调查公众对于某一作品或事件的态度，用于电商平台等的推荐系统，用于预测未来市场的走向等等。文本情感分类主要借助于文本的表征学习，然后把学到的文本嵌入传入深度神经网络进行训练，得到文本分类模型。

本实验采用IMDB电影评论的数据集，利用多种不同的文本表征学习的方法，生成了文本的向量化表示，并通过多层感知机的训练出文本情感分类模型，通过模型在测试集上的结果分析和探究各文本生成算法的优劣。

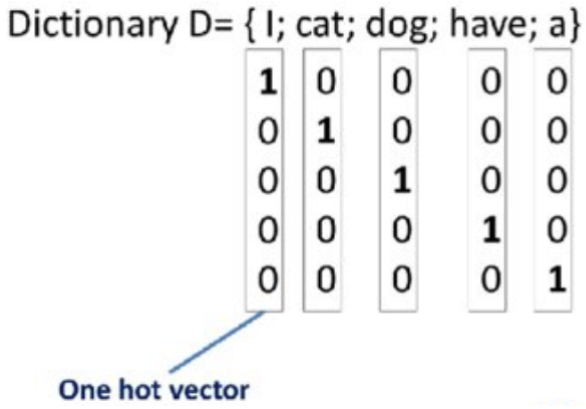
## 现有的主要方法

### 文本的表征学习

在机器学习中，目前文本的表征学习主要包括以下几种方法：

#### One-Hot Representation

最简单的表示文本数据的方法就是用一个one-hot vector来表示单词集中的每一个单词，例如



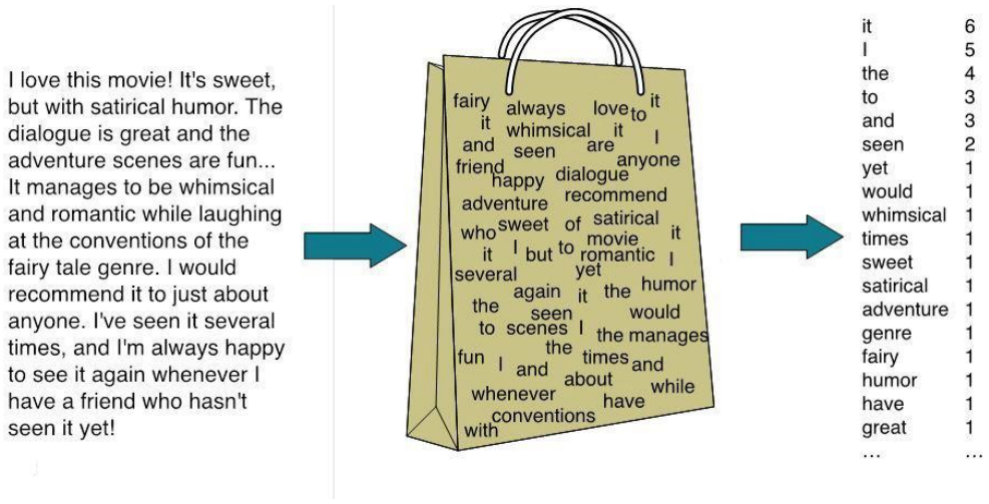

接着，每一个句子就可以表示为这些单词的one-hot编码按顺序拼接成的矩阵，比如句子I have a dot可以表示为

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

one-hot编码的主要缺陷在于，它无法反映单词之前语义的相似性，而且使用起来非常麻烦，维度也很高，文本的矩阵也非常稀疏。所以就有了如下的其他表征学习方法。

## TF (Term Frequency, or BOW, Bag of Words)

Bag-of-Word (BOW) 方法忽略了单词在文本中的出现顺序，只保留了每个单词在当前文本中出现的次数：




这种方法的弊端是假设每个单词都有同等的重要性，而实际上不同的单词所携带的信息量是不同的，例如英文中the, a之类的单词(称为stop words)通常就比classroom, football之类的单词要携带的信息要少。这个问题可以通过在文本预处理中，去除stop words来改善，或者采用TF-IDF的方法。

## TF-IDF

TF-IDF在TF的基础上，尝试给每个单词赋予不同的权重，具体来说，TF-IDF又两个统计结果的值相乘而得：

$$tfidf(w,d,D) = tf(w,d) \times idf(w,D)$$

其中 $w$ 为当前单词，$d$ 为第d篇文本，$D$ 为语料库（所有文本的集合）。

$tf(w,d)$ 代表了单词$w$在第d篇文本中的频率

$$tf(w,d) = \frac{\#\ of\ word\ w\ in\ d}{\#\ of\ all\ words\ in\ d}$$

$idf(w,D)$ 则为如下表达式

$$idf(w,D) = log\frac{\#\ of\ documents\ in\ D}{\#\ of\ documents\ containing\ word\ w\ in\ D}$$

$idf(w,D)$ 代表了单词$w$携带的信息量大小，$idf(w,D)$值越大，单词$w$携带的信息量越大。

所以，TF-IDF值$tfidf(w,d,D) = tf(w,d) \times idf(w,D)$ 涵盖了单词$w$在文本$d$的频率，以及它在语料库$D$中携带的信息量。

$$\texttt{tfidf}(t,d,D) = \texttt{tf}(t,d) \cdot \texttt{idf}(t,D)$$

| | | blue | bright | can | see | shining | sky | sun | today |
|---|---|---|---|---|---|---|---|---|---|
| Document 1 | 1 | **0.301** | 0 | 0 | 0 | 0 | 0.151 | 0 | 0 |
| Document 2 | 2 | 0 | 0.0417 | 0 | 0 | 0 | 0 | 0.0417 | **0.201** |
| Document 3 | 3 | 0 | 0.0417 | 0 | 0 | 0 | **0.100** | 0.0417 | 0 |
| Document 4 | 4 | 0 | 0.0209 | **0.100** | **0.100** | **0.100** | 0 | 0.0417 | 0 |

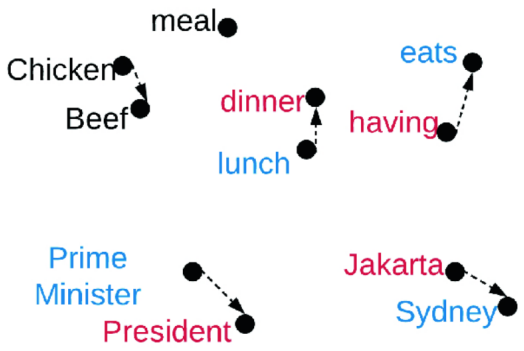
如图所示，每一行代表了每个文本的tf-idf值，每一列代表了每个单词的tf-idf值。

tf-idf的主要缺陷在于，它同样无法反映单词之间语义的相似性，也没有包含任何关于单词在文本中的顺序的信息，同样它非常稀疏，维度也非常高。

## Word2Vec

Word2vec是一种更加新兴的文本表征学习方法。它的目标是通过机器学习方法，获得单词的数值向量表示，而且这些向量具有如下特征

1. 稠密（低维向量）
2. 能反映单词之间的语义相似性，如下图所示




文本的嵌入则可以在单词的词嵌入的基础上计算而得。

Word2Vec的主要思想为：单词之间词义的相似性就隐含在现有的文本当中。例如，如果两个单词频繁地同时出现，那么他们的词义可能非常相近

A cup of **tea**
A cup of **coffee**
**Tea** or **coffee**?
**Coffee** and **tea** have caffeine
Let's go for a **coffee**
Let's get a **tea**
**Coffee** vs **Tea**: Which is Best?
I avoid adding sugar to my **tea**
I drink **coffee** with two spoons of sugar

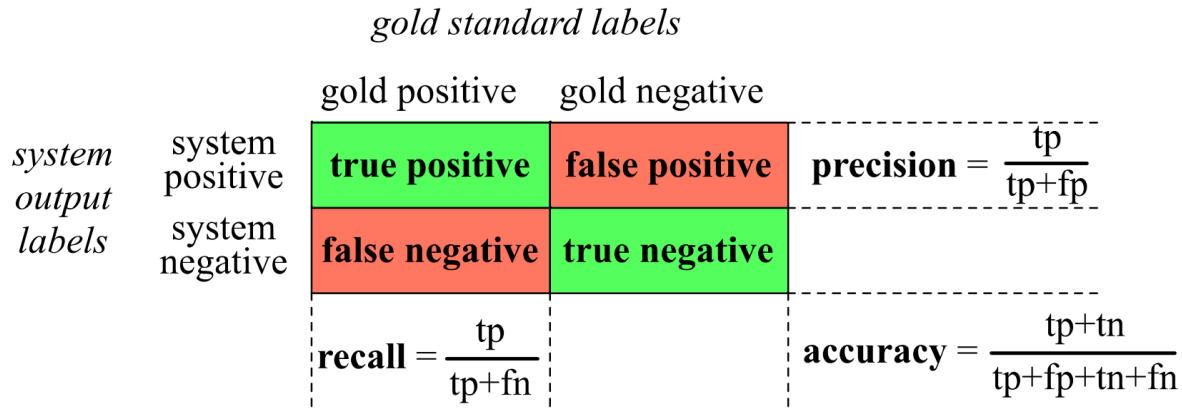
Word2Vec的具体方法主要有两种，一种是Skip-gram，即用中心的单词预测周围的单词；另一种是CBOW(Continuous bag of words)，即用周围的单词预测中心的单词。

## 文本的分类

文本分类的常用方法是将文本嵌入传入深度神经网络进行训练，从而得到一个文本分类器。本实验采用多层感知机来训练一个文本分类模型。根据Lecture6-Neural Networks中的介绍，多层感知机（MLP，multiplayer perception)就是全连接神经网络（fully connected Neural Network）。按照一般的PyTorch神经网络设计方法可以很方便实现一个简单的MLP，在接下来的部分会有详细的介绍。

## 分类结果的Accuracy, Precision, Recall, F measure

对于二分类的任务，主要的模型性能度量包括accuracy, precision, recall, F measure等。其中accuracy, precision, recall的示意图如下：




对于F measure，本实验采用的是平衡的F1，公式如下：

$$F_1 = \frac{2PR}{P+R}$$

其中P和R分别代表precision和recall。

# 本实验采用的模型与算法

## 读入与预处理数据

读入预处理数据的代码存于preprocess.py中，代码如下：

先导入需要用到的库：

```
import re
import csv
import string
import numpy as np
import pandas as pd
import unicodedata
from bs4 import BeautifulSoup
from nltk import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
```

读入和预处理的主要函数如下：

```
def read_and_preprocess():
    df = pd.read_csv('IMDB Dataset.csv', encoding = 'utf-8')
    df['review'] = df['review'].apply(denoise_text)
    df['sentiment'] = df['sentiment'].apply(convert_string_to_true_false)
    X = df['review'].values
    Y = df['sentiment'].values
    return X, Y
```

首先用pandas中的read_csv函数按utf-8编码读入数据集，然后对影评文本进行清洗，将影评的情感分类转换为布尔矩阵，最后将影评部分和分类部分分别赋值给X，Y数组。

其中denoise_text函数的代码如下：

```
def denoise_text(text):
    # Remove stopwords, punctuation, and convert to lowercase etc
    text = remove_non_utf8_characters(text)
    text = remove_html_tags(text)
    text = remove_URL(text)
    text = remove_punctuation(text)
    text = remove_single_characters(text)
    text = remove_digits(text)
    text = convert_to_lowercase(text)
    text = word_tokenize(text)
    text = remove_stopwords(text)
    text = lemmatize(text)
    return text
```

最后将处理好的数据用numpy.save放入npy文件中：

```
X, Y = read_and_preprocess()
with open('X_Y.npy', 'wb') as f:
    np.save(f, X)
    np.save(f, Y)
```

## 分出训练集、测试集和验证集

这一部分比较简单，先读入清洗好的数据集：

```
import numpy as np

with open('X_Y.npy', 'rb') as f:
    X = np.load(f, allow_pickle=True)
    Y = np.load(f, allow_pickle=True)
```

分割数据集：

```
X_train = X[:30000]
Y_train = Y[:30000]
X_test = X[30000:40000]
Y_test = Y[30000:40000]
X_valid = X[40000:]
Y_valid = Y[40000:]
```

保存：

```
with open('X_splitted.npy', 'wb') as f:
    np.save(f, X_train)
    np.save(f, X_test)
    np.save(f, X_valid)

with open('Y_splitted.npy', 'wb') as f:
    np.save(f, Y_train)
    np.save(f, Y_test)
    np.save(f, Y_valid)
```

接下来就到了向量化的步骤。

## TF

首先导入需要的库：

```
# Importing required modules
import pickle
import numpy as np
from nltk.tokenize import word_tokenize
```

读入训练集、测试集和验证集后，先遍历一遍训练集中的所有影评，将所有的单词放入word_set，把它们在所有影评中出现的次数放入collection_freq_dict：

```
# get word index dictionary, collection frequency dictionary from X_train
def get_dicts(X_train):
    word_set = set()
    collection_freq_dict = {}
    for review in X_train:
        for word in review:
            if word not in word_set:
                word_set.add(word)
                collection_freq_dict[word] = 1
            else:
                collection_freq_dict[word] += 1
    return word_set, collection_freq_dict
```

再根据单词出现的频率过滤掉低频词：

```
def remove_low_freq_words(X_train, threshold, word_set):
    word_set_copy = word_set.copy()
    for word in word_set_copy:
        if collection_freq_dict[word] < threshold:
            word_set.remove(word)
    return word_set
```

然后就给用剩余的每个单词创建索引值了，存在word_index_dict中：

```
def get_word_index_dict(word_set):
    word_set = list(word_set)
    word_set.sort()
    word_index_dict = {}
    for i, word in enumerate(word_set):
        word_index_dict[word] = i
    return word_index_dict
```

为了实现Tf向量的计算，需要实现对每个影评进行词频的计算。先遍历一遍影评中的所有单词得到每个单词出现的次数，再除以影评的长度得到频率：

```
# Term Frequency (TF)
def get_tf_dict(review):
    tf_dict = {}
    for word in review:
        if word in tf_dict:
            tf_dict[word] += 1
        else:
            tf_dict[word] = 1
    return tf_dict

def tf(word, review, tf_dict):
    return tf_dict[word] / len(review)
```

最后就可以为每篇影评计算tf向量了：

```
def tf_vec(review, N, df_dict):
    tf_vector = np.zeros(len(word_index_dict))
    tf_dict = get_tf_dict(review)
    for word in review:
        if word in word_index_dict:
            tf_vector[word_index_dict[word]] = tf(word, review, tf_dict)
    return tf_vector
```

## TF-IDF

tf-idf向量的计算流程和tf类似，但要多乘一项idf，仿照tf的写法，先遍历数据集（训练集、测试集或验证集）得到每个单词出现的影评个数，存在df_dict中，再获取每个单词的idf：

```
def get_df_dict(X):
    df_dict = {}
    for review in X:
        for word in review:
            if word in df_dict:
                df_dict[word] += 1
            else:
                df_dict[word] = 1
    return df_dict

# Inverse Document Frequency (IDF)
def idf(word, N, df_dict):
    return np.log10(N / df_dict[word])
```

然后就可以获取每篇影评的tf-idf向量了：

```
def tf_idf_vec(review, N, df_dict):
    tf_idf_vector = np.zeros(len(word_index_dict))
    tf_dict = get_tf_dict(review)
    for word in review:
        if word in word_index_dict:
            tf_idf_vector[word_index_dict[word]] = tf(word, review, tf_dict) *
idf(word, N, df_dict)
    return tf_idf_vector
```

## word2vec

word2vec的实现思路和前两种不太一样，主要是调用库函数来获取向量。首先导入需要的库：

```
import pickle
import numpy as np
from gensim.test.utils import common_texts
from gensim.models import Word2Vec
```

创建并训练模型：

```
model = Word2Vec(X_train, min_count = min_count, vector_size=100, workers=1,
seed=1)
# 以上采用的word2vec参数在实验中有调整
```

这样还只是得到了每个单词的向量，要得到每篇影评的向量，我参考了stackoverflow.com上面的问题 How to get vector for a sentence from the word2vec of tokens in sentence：

> There are different methods to get the sentence vectors :
>
> 1. **Doc2Vec** : you can train your dataset using Doc2Vec and then use the sentence vectors.
> 2. **Average of Word2Vec vectors** : You can just take the average of all the word vectors in a sentence. This average vector will represent your sentence vector.
> 3. **Average of Word2Vec vectors with TF-IDF** : this is one of the best approach which I will recommend. Just take the word vectors and multiply it with their TF-IDF scores. Just take the average and it will represent your sentence vector.

这里我选择了第二种办法，将影评中的每个单词的向量计算平均值得到影评的向量（第三种方法尝试过但正确率不如第二种方法高），其中保留的单词最小出现次数为50，与tf和tf-idf最小为100略有不同。

代码如下：

```
X_train_word2vec = np.array([
    np.mean([X_train_tf_idf[i][word_index_dict[word]] * model.wv[word]
             for word in X_train[i] if word in word_index_dict], axis=0)
    for i in range(len(X_train))
])

X_test_word2vec = np.array([
    np.mean([X_test_tf_idf[i][word_index_dict[word]] * model.wv[word]
             for word in X_test[i] if word in word_index_dict], axis=0)
    for i in range(len(X_test))
])

X_valid_word2vec = np.array([
    np.mean([X_valid_tf_idf[i][word_index_dict[word]] * model.wv[word]
             for word in X_valid[i] if word in word_index_dict], axis=0)
    for i in range(len(X_valid))
])
```

# 前馈神经网络

## 设计

分类方法我选用了前馈神经网络，主要借助pytorch中的库函数来搭建，包括两层节点，每层100个节点，使用ReLU和sigmoid等函数。代码如下：

```
import pickle
import numpy as np
import torch.nn as nn
import torch.optim as optim
import torch

class FFNN(nn.Module):
    def __init__(self, input_size, hidden_size_1, hidden_size_2, output_size):
        super(FFNN, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size_1)
        self.relu_1 = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size_1, hidden_size_2)
        self.relu_2 = nn.ReLU()
        self.fc3 = nn.Linear(hidden_size_2, output_size)

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu_1(out)
        out = self.fc2(out)
        out = self.relu_2(out)
        out = self.fc3(out)
        return torch.squeeze(torch.sigmoid(out))

    def predict(self, x):
        self.eval()
        torch.no_grad()
        return self.forward(x)

    def evaluate(self, output, y):
```

```
        output = torch.round(output)
        correct = (output == y).float()
        return correct.sum() / len(correct)
```

## 训练

模型的训练主要用train_model成员函数完成，每次循环先计算出结果再求出与标准结果之间的损失，再用损失对神经网络中的权值进行调整。由于想要观察训练过程中损失和正确率的变化，每二十次循环都记录一次训练集和测试集上的损失和正确率，用于作图。

训练函数的代码如下：

```
    def train_model(self, x, y, x_test, y_test, epochs):
        self.train()
        epoch_list = []
        train_loss = []
        test_loss = []
        train_acc = []
        test_acc = []
        for epoch in range(epochs):
            output = self.forward(x)
            loss = criterion(output, y)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            if epoch % 20 == 0:
                epoch_list.append(epoch)
                accuracy = self.evaluate(output, y)
                train_loss.append(loss.item())
                train_acc.append(accuracy.item())
                output = self.forward(x_test)
                loss = criterion(output, y_test)
                accuracy = self.evaluate(output, y_test)
                test_loss.append(loss.item())
                test_acc.append(accuracy.item())
        return train_loss, train_acc, test_loss, test_acc, epoch_list
```

作图的代码如下：

```
import matplotlib.pyplot as plt
from matplotlib.pyplot import figure

figure(figsize=(8, 8))
plt.plot(epoch_list, train_loss)
plt.plot(epoch_list, test_loss)
plt.title(feature + ' Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['train', 'val'], loc='upper right')
plt.show()

figure(figsize=(8, 8))
plt.plot(epoch_list, train_acc)
plt.plot(epoch_list, test_acc)
plt.title(feature + ' Accuracy')
plt.ylabel('Accuracy')
```

```
plt.xlabel('Epoch')
plt.legend(['train', 'val'], loc='upper right')
plt.show()
```

最后保存checkpoint:

```
model.save(feature + '.pth')
```

### 验证训练结果

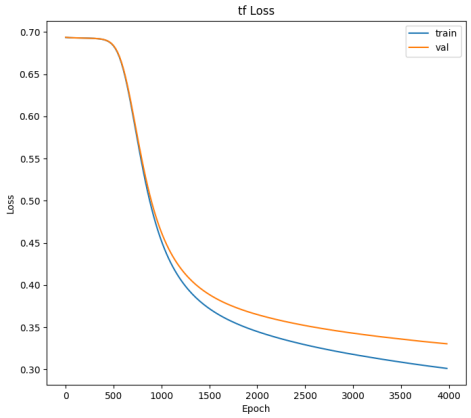
从保存的checkpoint文件中恢复模型，在验证集上进行结果的预测并计算准确率，代码如下：

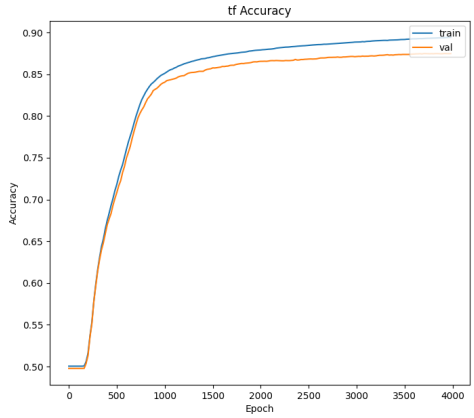
```
model.load(feature+'.pth')

print("validation set: ", model.evaluate(
    model.predict(X_valid), Y_valid).item().__round__(4))
```

# 实验结果与分析

## 训练集和验证集上的损失与正确率

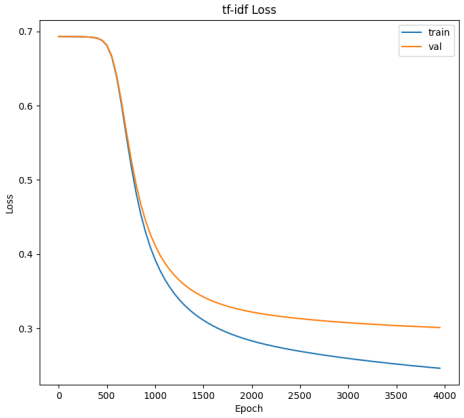
### TF

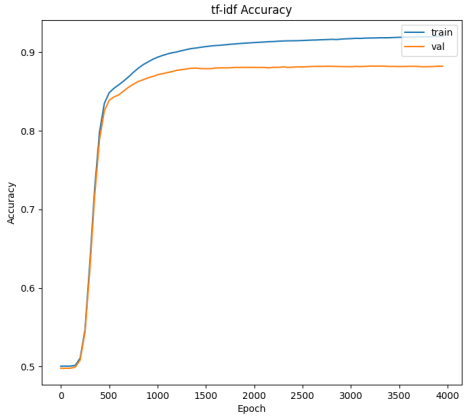



如图所示，TF的文本表示用于训练文本情感分类模型时，正确率上升和损失下降都很快，到后面趋于平稳，逐渐拟合到最优解，最终在验证集上达到87%左右的正确率。

### TF-IDF



如图所示，TF-IDF的训练过程也是呈先快后慢的趋势，最终在验证机上收敛到88%左右的正确率。

## word2vec

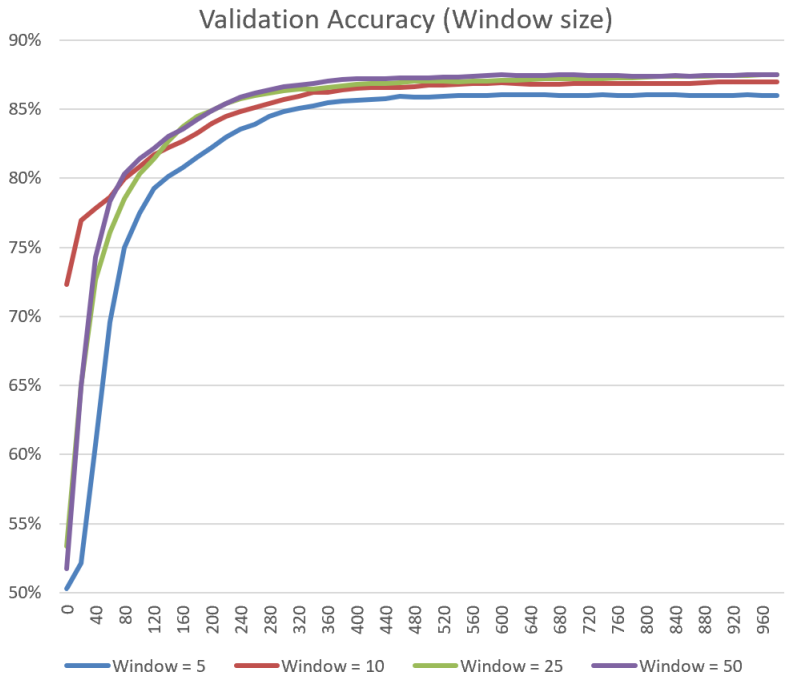
本实验探究了word2vec和前馈神经网络中的一些超参数对于模型最终正确率的影响，列表如下：

| # | alpha | window | vector_size | epochs | training_iterations | accuracy |
|---|---|---|---|---|---|---|
| 1 | 0.025 | 5 | 100 | 5 | 3000 | 0.8611 |
| 2 | 0.025 | 5 | 100 | 5 | 5000 | 0.8406 |
| 3 | 0.025 | 5 | 100 | 5 | 5000 | 0.8591 |
| 4 | 0.025 | 5 | 100 | 5 | 3000 | 0.8565 |
| 5 | 0.025 | 5 | 100 | 5 | 3000 | 0.8542 |
| 6 | 0.025 | 5 | 100 | 5 | 3000 | 0.8588 |
| 7 | 0.025 | 5 | 100 | 5 | 3000 | 0.8586 |
| 8 | 0.01 | 5 | 100 | 5 | 3000 | 0.8293 |
| 9 | 0.05 | 5 | 100 | 5 | 1500 | 0.8601 |
| 10 | 0.04 | 5 | 100 | 5 | 1000 | 0.8573 |
| 11 | 0.05 | 10 | 100 | 5 | 1000 | 0.8697 |
| 12 | 0.05 | 25 | 100 | 5 | 1000 | 0.8749 |
| 13 | 0.05 | 50 | 100 | 5 | 1000 | 0.8751 |
| 14 | 0.05 | 50 | 200 | 5 | 1000 | 0.8775 |
| 15 | 0.05 | 50 | 500 | 5 | 1000 | 0.8792 |
| 16 | 0.05 | 50 | 500 | 5 | 1000 | 0.8811 |
| 17 | 0.05 | 50 | 500 | 5 | 1000 | 0.8812 |
| 18 | 0.05 | 50 | 500 | 10 | 1000 | 0.8821 |
| 19 | 0.05 | 50 | 500 | 10 | 1000 | 0.8809 |
| 20 | 0.05 | 50 | 1000 | 10 | 1000 | 0.8837 |
| 21 | 0.05 | 50 | 1000 | 15 | 1000 | 0.8799 |

其中前四列为训练得到Word2vec向量表示时的参数，第五列training iteration为训练二分类的神经网络时的迭代次数。IMDB电影评论文本corpus的min_count均设为100。

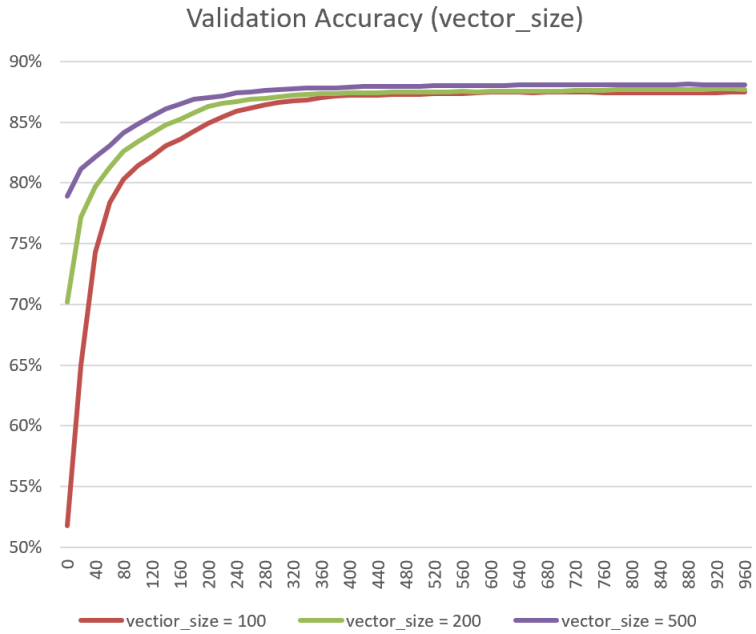
在表格中可以看到，增大滑动窗口的大小、word2vec向量长度，以及增大训练生成word2vec的epoch数都有助于提高文本表征学习的效果。

对比表格中的第9、11、12、13次训练，可以看到增大滑动窗口的大小可以提升二分类的正确率，这是因为增大窗口有利于发掘距离较远的单词之间的关系，尤其是带有感情色彩的单词与其他单词的关系。下图是第9、11、12、13次训练的Validation Accuracy的对比：



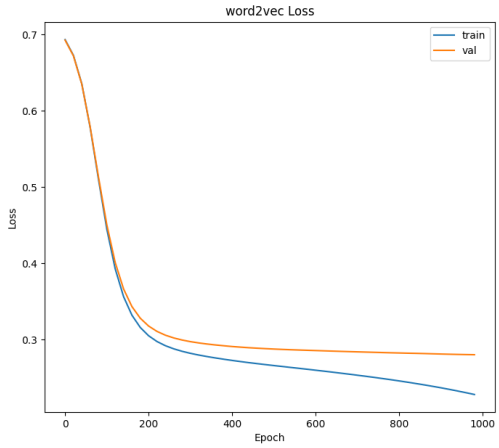

如图所示，可以看到从window = 5增大到window = 10有比较大的改进，而从window = 10增大到25或50的改进就没那么明显了。

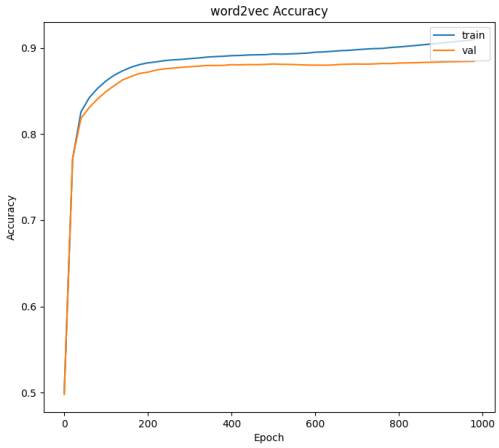
对比第13、14、15次训练，可以得出增大vector size提升了文本分类的正确率，这是因为增大向量长度可以增加文本表征学习的结果所携带的信息。下图是第13、14、15次训练的Validation Accuracy的对比：




对比第17、18次训练和第20、21次训练，可以得出一定程度增大训练轮数有利于提升实验的正确率，这是因为增大训练轮数有助于在多次迭代中发掘单词蕴含的信息，但训练轮数过大时，word2vec模型可能会学习到一些噪音，产生过拟合的现象，反而在分类任务中的性能下降了。

实验中最好的一次训练（第20次）的loss和accuracy图表如下：




## 验证集上的正确率

```
C:\Users\豹豹\OneDrive - 中山大学\大三下\机器学习与数据挖掘\19335015-陈恩婷\劳train最优>python -u "c:\Users\豹豹\OneDrive - 中山大学\大三下\机器学习与数据挖掘\19335015-陈恩婷\劳train最优\test.py"
finished loading
created model
tf
test set: 0.8752 precision = 0.8633 recall = 0.8903 f_score = 0.8766
valid set: 0.8738 precision = 0.871 recall = 0.878 f_score = 0.8745

C:\Users\豹豹\OneDrive - 中山大学\大三下\机器学习与数据挖掘\19335015-陈恩婷\劳train最优>python -u "c:\Users\豹豹\OneDrive - 中山大学\大三下\机器学习与数据挖掘\19335015-陈恩婷\劳train最优\test.py"
finished loading
created model
tf-idf
test set: 0.882 precision = 0.8695 recall = 0.8978 f_score = 0.8834
valid set: 0.8829 precision = 0.8806 recall = 0.8864 f_score = 0.8834

C:\Users\豹豹\OneDrive - 中山大学\大三下\机器学习与数据挖掘\19335015-陈恩婷\劳train最优>python -u "c:\Users\豹豹\OneDrive - 中山大学\大三下\机器学习与数据挖掘\19335015-陈恩婷\劳train最优\test.py"
finished loading
created model
word2vec
test set: 0.883 precision = 0.8744 recall = 0.8933 f_score = 0.8837
valid set: 0.8849 precision = 0.8875 recall = 0.882 f_score = 0.8847
```

如图所示，三种向量化方法都达到了85%以上的正确率和F score，其中Word2vec的性能最好，在测试集达到了88.3%的正确率，TF-IDF次之，TF的正确率最低。这是因为Word2vec作为一个深度学习模型，相对于传统的TF-IDF等方法能挖掘出文本蕴含的信息。

# 实验中遇到的一些问题和讨论

## 计算TF-IDF时一直算不出结果

解决方法：检查代码发现，原先的实现方法不是很好，每次计算tf或者idf都把影评遍历一遍，非常浪费时间，其实只遍历一次影评就可以得到tf和idf，改进之后性能明显提高了。

## 使用Matplotlib时出现报错

实验的代码中使用matplotlib进行作图，但是一调用其库函数就会报如下错误：

> Warning: QT_DEVICE_PIXEL_RATIO is deprecated. Instead use:
> QT_AUTO_SCREEN_SCALE_FACTOR to enable platform plugin controlled per-screen factors.
> QT_SCREEN_SCALE_FACTORS to set per-screen DPI.
> QT_SCALE_FACTOR to set the application global scale factor.
>
> This application failed to start because no Qt platform plugin could be initialized

解决方法：查阅资料发现这其实是两个不同的报错。关于第二个报错，我参考这篇文章，补充了相应环境变量再重启电脑，但是还有前面的报错，于是按照这篇文章的建议补充了一下代码：

```
def suppress_qt_warnings():
    environ["QT_DEVICE_PIXEL_RATIO"] = "0"
    environ["QT_AUTO_SCREEN_SCALE_FACTOR"] = "1"
    environ["QT_SCREEN_SCALE_FACTORS"] = "1"
    environ["QT_SCALE_FACTOR"] = "1"

suppress_qt_warnings()
```

这样就不会有原来的报错了，可以正常使用matplotlib。

## 训练好模型以后，一旦重新计算每篇影评的向量，模型就会失效

训练好模型后，如果删除了一开始生成的存有文本表征的.bin文件，重新运行代码来生成这几个文件后，验证集上面的结果就会出现反弹回了很低的正确率的情况

解决方法：仔细检查代码每一步的结果并查阅资料后，发现是这两个问题：

### tf和tf-idf向量中的set顺序问题

Python中的set里面使用的是hash结构，即使存储的数据内容相同，每次运行也会按不同顺序存放，就会导致下面这段代码遍历word_set中每个单词的顺序不同，相同的单词就会每次在word_index_dict中都获得不同id：

```
def get_word_index_dict(word_set):
    word_index_dict = {}
    for i, word in enumerate(word_set):
        word_index_dict[word] = i
    return word_index_dict
```

由于生成向量的函数会根据每个单词在word_index_dict中的id对向量的各维度进行赋值，影评向量化的结果就会不一样：

```
def tf_vec(review, N, df_dict):
    tf_vector = np.zeros(len(word_index_dict))
    tf_dict = get_tf_dict(review)
    for word in review:
        if word in word_index_dict:
            tf_vector[word_index_dict[word]] = tf(word, review, tf_dict)
    return tf_vector
```

解决这个问题的一种方法是修改PYTHONHASHSEEED（见这个问题），但是我在尝试了在代码中修改的方式（`os.environ['PYTHONHASHSEED '] ='0'`）好像没有效果，为了简便起见，我直接将set转换为list，再给list排序（见这篇文章）：

```
    word_set = list(word_set)
    word_set.sort()
```

这样就可以避免每次计算出的向量不同的问题。

### word2vec训练出的模型的随机性

word2vec情况类似，也是PYTHONHASHSEED导致的问题，虽然找到的文章中说python3除了修改代码还要配置环境变量，但是我尝试后发现修改代码就已经可以解决问题了：

```
model = Word2Vec(X_train, min_count = min_count, vector_size=100, workers=1,
seed=1)
```

这里加上`workers=1, seed=1`即可。

# 实验结论与总结

本实验通过尝试了TF(CBOW)、TF-IDF和Word2vec等不同的文本表征学习算法于模型，复习于巩固了关于表征学习部分的相关知识，并将所得的文本嵌入应用于深度神经网络的训练，对IMDB电影评论文本进行情感分类，探究与比较了不同的表征学习算法与他们的情感分类效果，其中采用TF-IDF时得到了最大测试集正确率88.49%。对于Word2Vec模型，实验发现增大滑动窗口的大小、word2vec向量长度，以及增大训练生成word2vec的epoch数都有助于提高文本表征学习的效果。

在实现模型的过程中，我还遇到了一些关于python的matplotlib、set和word2vec等等的问题，经过详细查阅资料和不断试验都成功解决了问题，也收获了很多知识与技能。

# 参考链接


1. Pattern Recognition and Machine Learning
2. https://www.tensorflow.org/tutorials/text/word2vec
3. https://stackoverflow.com/questions/29760935/how-to-get-vector-for-a-sentence-from-the-word2vec-of-tokens-in-sentence
4. https://blog.csdn.net/qq_43480604/article/details/117906343
5. https://blog.csdn.net/QinZheng7575/article/details/108980162
6. https://stackoverflow.com/questions/3812429/is-pythons-set-stable
7. https://blog.xiaoquankong.ai/python-set-%E8%BF%AD%E4%BB%A3%E8%BF%94%E5%9B%9E%E5%85%83%E7%B4%A0%E9%A1%BA%E5%BA%8F%E9%9A%8F%E6%9C%BA%E5%8F%98%E5%8C%96%E9%97%AE%E9%A2%98/
8. https://blog.csdn.net/zdm_0301/article/details/120013915