在第一个实践课（使用 XGB 实现酒店信息消歧）中其实没有涉及太多的代码，主要是以介绍思路为主。在这一课时中，我将提供一个较为完整的代码，带领你亲自实践一下。

理解业务

在旅行场景下，城市——我们通常称为目的地，是一个很重要的信息。根据用户对于目的地的偏好，我们既可以把目的地作为一个特征用于推荐系统中，也可以把目的地当作一个被推荐的信息直接推荐给用户。所以，我们有一个需求，就是把相似的目的地整理出来，然后可以通过这些相似目的地做相关推荐，或者是相关目的地的推荐。

理解数据

可以想到，这是一个比较典型的聚类问题，我们只要能够把相似的城市按照一定的相关性聚在一起，就可以完成我们的需求，当然具体的效果要根据结果不断地进行调整。

那我们就来看一下我们的数据。

思来想去我们只有很多目的地的名字，但这些目的地并没有什么统一标准的特征可以给我们做向量，那么该怎么去给这些目的地计算相关性呢？

这时不禁想到，我们有很多用户写过游记，这些游记里总会出现各种各样的目的地名字，对于相似的目的地，那用户所写的内容也会有一定的相似性，不管是地理位置接近，还是消费价位类似，或者是可以玩的内容存在一定的相似性。

总之，我们可以靠这些内容把这些城市的名字关联起来，而且不同于结构化的信息，游记是用户自己来写的内容，里面对于目的地的认知也是用户的认知，所以如果我们能够从中发现关联性，再应用到用户身上也是比较合理的。比如说 “三亚” 如果只是按客观属性来划分，那应该是“海边”，但是很多用户去三亚，除了看海本身，还有家庭出游等，这些是只能从用户的角度才会产生的认知。

这里，我们就要用到一个 Word2Vec 算法，它可以学习输入的文本，并输出一个词向量模型，经过 Word2Vec 算法处理之后，每一个词都会变成一个预设长度的数值向量。这个算法会在后面的章节进行更详细的讲解，这里我们大概知道它的功能就可以了。下面我们进入到具体代码实现的环节，看看如何训练一个这样的模型。

准备数据与模型训练

准备数据

我们获取所有需要用到的文本数据，在这里使用了全量的游记文本数据。**我们首先要对数据进行清洗，去除掉异常的数据**，比如内容过短、获取失败，或者是存在特殊字符、使用纯英文 / 泰语写的游记，等等。

完成了这个步骤之后我们要对文本内容进行分词，因为我期望 Word2Vec 最终构建的向量是词级别的。**完成分词之后，我们把数据存储在文本文件中**，其中每一行是一篇内容。

接下来就要训练我们的 Word2Vec 模型了。

训练 Word2Vec 模型

这里我们使用了一个新的算法包：Gensim。不知道你是否还记得我在之前介绍过这个工具包，它主要用于从原始的非结构化文本信息中，通过无监督算法学习文本向量表达。这里面支持 TF-IDF、LSA、LDA 和 Word2Vec 等多种算法模型。来看一下代码。

```
import gensim


import os


import re


import sys


import multiprocessing


from time import time


class getSentence(object):


def __init__(self, dirname):


        self.dirname = dirname
```

文本可以存储在多个文本文件中，存放在一个文件目录下，这里构建了一个迭代方法，循环读取目录下的所有文件。

我这里使用的文件目录为 traindata，在 traindata 下面有 31 个语料文件，其中每个有 1G 左右，如下图所示。

traindata_00 traindata_03 traindata_06 traindata_09 traindata_12 traindata_15 traindata_18 traindata_21 traindata_24 traindata_27 traindata_30
traindata_01 traindata_04 traindata_07 traindata_10 traindata_13 traindata_16 traindata_19 traindata_22 traindata_25 traindata_28 traindata_31
traindata_02 traindata_05 traindata_08 traindata_11 traindata_14 traindata_17 traindata_20 traindata_23 traindata_26 traindata_29

```
def __iter__(self):

for root, dirs, files in os.walk(self.dirname):

for filename in files:

                file_path = root + '/' + filename

for line in open(file_path):

try:

                        s_line = line.strip()

if s_line== "":

continue

                        word_line = [word for word in s_line.split( )]

yield word_line

except Exception:

                        print("catch exception")

yield ""

if __name__ == '__main__':

    begin = time()

    sentences = getSentence("traindata")

    model = gensim.models.Word2Vec(sentences,size=200,window=15,min_count=10,
workers=multiprocessing.cpu_count())
```

```
    model.save("model/word2vec_gensim")

    model.wv.save_word2vec_format("model/word2vec_org",

"model/vocabulary",

                                  binary=False)

    end = time()

print ("Total procesing time: %d seconds" % (end - begin))
```

在正常的情况下，我们会在 model 路径下看到几个文件。其中比较重要的两个，一个 vocabulary 是词典文件，记录了出现过的词汇以及词汇出现的次数；一个 word2vec_gensim 是生成的向量文件。

通过上面的方法，我们成功获取到了很多词汇的向量，这里我的词汇量大概有 1000w 左右。但是我们这次所需要的是寻找相似城市，所以对于那些非城市名字的词汇就没有什么价值了。

于是我们这里使用我们自己的城市词库与词汇表进行匹配，对于没有在词汇表中出现过的城市名称也没有办法计算，要把这部分剔除掉。不用担心，如果这么多的语料都没有出现过的城市也一定是没有人去过的城市。

训练 K-means 模型

下面我们就可以开始训练我们的 K-means 模型了。像我们前面用过的一样，K-means 是在 sklearn 里面的一个模块。具体步骤如下所示。

```
import gensim

from sklearn.cluster import KMeans

from sklearn.externals import joblib

from time import time

def load_model():

    model = gensim.models.Word2Vec.load('../word2vec/model/word2vec_gensim')

return model

def load_filterword():
```

```
    fd = open("mddwords.txt","r")

    filterword=[]

for line in fd.readlines():

        line=line.strip()

        filterword.append(line)

return filterword

if __name__=="__main__":

    start = time()

    model = load_model()

    filterword = load_filterword()

    print(len(filterword))

    wordvector = []

    filterkey={}

for word in filterword:

        wordvector.append(model[word])

        filterkey[word]=model[word]

    print(len(wordvector))

    clf = KMeans(n_clusters=2000,max_iter=100,n_jobs=10)

    s = clf.fit_predict(wordvector)

    joblib.dump(clf,"kmeans_mdd2000.pkl")
```

```
    labels = clf.labels_

    labellist = labels.tolist()

    print(clf.inertia_)

    fp = open("label_mdd2000",'w')

    fp.write(str(labellist))

    fp.close()

    fp1 = open("keys_mdd2000",'w')

for key in filterkey:

        fp1.write(key+'\n')

    print("over")

    end = time()

    print("use time")

    print(end-start)
```

经过上面的步骤，我们就训练好了 K-means 模型，当然，经过反复尝试，最终确定的不是 2000 这个簇数量，而是使用了 100 个簇的结果。我们尝试了 50、100、200、500、1000、2000 等多个聚类的结果，经过我们最后的对比评估，100 个簇的时候效果较好，于是我们最终选择了这个模型。

下图是我从结果中抽了一些簇的 TOP 结果生成的图片，可以看到聚类的效果还是很不错的。比如右下角那一簇基本都是日本关西的城市名字，左下角基本都是川藏线上的地点。

有了已经训练好的模型，我们就知道了这些相似城市的名称以及它们所属的簇。接下来我们要做的，就是**把这些数据存储到数据库中，并在具体的业务中进行应用**了。

当然，随着时间的推移，在积累了一段时间的数据之后，我们还要对模型进行重新迭代，以期望获得更好的结果。

总结

在这一节实践课程中，我着重介绍了整个模型训练环节的代码，其中主要写了两段代码，分别训练了 Word2Vec 模型和 K-means 模型。除了数据部分，这些代码几乎可以复制即运行。

到这一课时，关于聚类问题的内容就告一段落了，在数据缺少标注的时候，聚类算法是十分常用的，它可以帮助我们了解数据情况。当然，聚类方法也存在一些局限，还需要在日常的工作中多加练习，不断积累自己的经验。