电子商务数值数据与文本分析案例

🔒 项目流程:

- 特征变量可视化: sns.histplot pieplot
- 特征工程:关于review text构建新特征:word count/character count
- 多变量:百分比发生率透视表
- 因子相关性分析
- 利用Bert预训练模型为text分词,词嵌入
- 利用NLTK库中的VADER遍历给出综合极性分数
- 根据分数“Polarity Score”并相应地分配情感标签
- 利用ngrams统计不同类别的出现频率得N-字节片段
- Naive Bayes是一个概率模型，
  它依赖于贝叶斯定理，通过观察一个词在不同类别中的出现情况来计算其类别的概率。
  由于同时考察了好的和坏的评论，它能够提取出最能使类别两极化的单格标记。
  使用这个模型，我有可能预测
  没有标签的评论的积极或消极情绪。

🔒 一些数据预处理操作:

1. 统一大小写
2. 分词
3. 去除停用词
4. 词干提取

🔒 Naive Bayes Classifier
简单来说:

1. 特征字典准备

在朴素贝叶斯分类器中，每个样本（在您的案例中是一段文本）都通过一个特征字典来表示。这个字典的键是特征（通常是单词），而值是表示该特征是否出现在样本中的布尔值

2. 特征提取

在您的代码中，特征提取是通过 `find_features` 函数完成的。这个函数接受一个文档（一系列单词），并返回一个字典，其中包含了您选择的所有特征词（`word_features`）及其在该文档中是否出现。

```
'''
使用朴素贝叶斯分类器来预测文本的情感倾向（在这个案例中是用户是否推荐某个产品）。
这种方法适用于基于词频的文本分类任务，并且在处理小型到中型数据集时效果良好。
'''
from nltk.stem.porter import PorterStemmer
ps = PorterStemmer()
df['tokenized'] = df["Review Text"].astype(str).str.lower() # Turn into lower case text
df['tokenized'] = df.apply(lambda row: tokenizer.tokenize(row['tokenized']), axis=1) # Apply tok
df['tokenized'] = df['tokenized'].apply(lambda x: [w for w in x if not w in stopwords]) # Remove
df['tokenized'] = df['tokenized'].apply(lambda x: [ps.stem(w) for w in x]) # Apply stemming to e
all_words = nltk.FreqDist(preprocessing(df['Review Text'])) # Calculate word occurrence from who

vocab_count = 200
word_features= list(all_words.keys())[:vocab_count] # 2000 most recurring unique words
print("Number of words columns (One Hot Encoding): {}".format(len(all_words)))

# Tuple
labtext= list(zip(df.tokenized, (df["Recommended IND"])))

# Function to create model features
# for each review, records which unique words out of the whole text body are present
def find_features(document):
    words = set(document)
    features = {}
    for w in word_features:
        features[w] = (w in words)

    return features
# Apply function to data
featuresets = [(find_features(text), LABEL) for (text, LABEL) in labtext]
len(featuresets)

# Train/Test
training_set = featuresets[:15000]
testing_set = featuresets[15000:]

# Posterior = prior_occurrence * likelihood / evidence
classifier = nltk.NaiveBayesClassifier.train(training_set)

# Output
print("Classifier accuracy percent:",(nltk.classify.accuracy(classifier, testing_set))*100)
print(classifier.show_most_informative_features(40))
```

```
# 由于NLP算法的计算较为耗时，我们仅仅取其中的3000条数据作为训练集
train_set = df[:3000]
print(train_set['Label'].value_counts())
model_class, tokenizer_class, pretrained_weights = (tfs.BertModel, tfs.BertTokenizer, 'bert-base
tokenizer = tokenizer_class.from_pretrained(pretrained_weights)
model = model_class.from_pretrained(pretrained_weights)
train_tokenized = train_set['Review Text'].apply((lambda x: tokenizer.encode(x, add_special_toke

from keras.preprocessing.sequence import pad_sequences
MAX_LEN = 128  # 定义序列最大长度
input_ids = pad_sequences(train_tokenized, maxlen=MAX_LEN, dtype="long",
                          value=0, truncating="post", padding="post")

attention_masks = []
for seq in input_ids:
    seq_mask = [float(i>0) for i in seq]
    attention_masks.append(seq_mask)

from torch.utils.data import TensorDataset, DataLoader, RandomSampler, SequentialSampler
import torch

# 将数据转换为torch tensors
inputs = torch.tensor(input_ids)
labels = torch.tensor(train_set['Label'].values)
masks = torch.tensor(attention_masks)

# 创建TensorDataset
data = TensorDataset(inputs, masks, labels)

# 创建DataLoader
batch_size = 32
data_loader = DataLoader(data, sampler=RandomSampler(data), batch_size=batch_size)

from transformers import AdamW, BertConfig

# 定义优化器
optimizer = AdamW(model.parameters(), lr=2e-5, eps=1e-8)

# 总的训练步骤 = 数据批次数 * 轮数
epochs = 4
total_steps = len(data_loader) * epochs

# 创建学习率调度器
from transformers import get_linear_schedule_with_warmup

scheduler = get_linear_schedule_with_warmup(optimizer, num_warmup_steps=0, num_training_steps=to

import numpy as np

# 用于计算整个训练过程的准确度
def flat_accuracy(preds, labels):
    pred_flat = np.argmax(preds, axis=1).flatten()
    labels_flat = labels.flatten()
    return np.sum(pred_flat == labels_flat) / len(labels_flat)

# 训练循环
for epoch in range(0, epochs):
    # 训练模式
    model.train()

    # 用于跟踪每一轮的损失
    total_loss = 0

    # 对于数据加载器中的每个批次
    for step, batch in enumerate(data_loader):
        # 解压数据，并将其放置到GPU上
        b_input_ids = batch[0].to(device)
        b_input_mask = batch[1].to(device)
        b_labels = batch[2].to(device)

        # 清除之前的梯度
        model.zero_grad()

        # 前向传播
        outputs = model(b_input_ids, token_type_ids=None, attention_mask=b_input_mask, labels=b_
        loss = outputs[0]

        # 累计损失
        total_loss += loss.item()

        # 后向传播
        loss.backward()

        # 防止梯度爆炸
        torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)

        # 更新参数
        optimizer.step()
        scheduler.step()

    # 计算这一轮的平均损失
    avg_train_loss = total_loss / len(data_loader)

#-----------根据NLTK.VADER直接对文本情感分析-----------------------------------------------------
from nltk.sentiment.vader import SentimentIntensityAnalyzer
SIA = SentimentIntensityAnalyzer()
df["Review Text"]= df["Review Text"].astype(str)

# Applying Model, Variable Creation
df['Polarity Score']=df["Review Text"].apply(lambda x:SIA.polarity_scores(x)['compound'])
df['Neutral Score']=df["Review Text"].apply(lambda x:SIA.polarity_scores(x)['neu'])
df['Negative Score']=df["Review Text"].apply(lambda x:SIA.polarity_scores(x)['neg'])
df['Positive Score']=df["Review Text"].apply(lambda x:SIA.polarity_scores(x)['pos'])

# Converting 0 to 1 Decimal Score to a Categorical Variable
df['Sentiment']=''
df.loc[df['Polarity Score']>0,'Sentiment']='Positive'
df.loc[df['Polarity Score']==0,'Sentiment']='Neutral'
df.loc[df['Polarity Score']<0,'Sentiment']='Negative'
```

🔒 N-gram提取高频词

```
from nltk.util import ngrams
from collections import Counter
def get_ngrams(text, n):
    n_grams = ngrams((text), n)
    return [ ' '.join(grams) for grams in n_grams]

def gramfreq(text,n,num):
    # Extracting bigrams
    result = get_ngrams(text,n)
    # Counting bigrams
    result_count = Counter(result)
    # Converting to the result to a data frame
    df = pd.DataFrame.from_dict(result_count, orient='index')
    df = df.rename(columns={'index':'words', 0:'frequency'}) # Renaming index column name
    return df.sort_values(["frequency"],ascending=[0])[:num]

def gram_table(data, gram, length):
    out = pd.DataFrame(index=None)
    for i in gram:
        table = pd.DataFrame(gramfreq(preprocessing(data),i,length).reset_index())
        table.columns = ["{}-Gram".format(i),"Occurrence"]
        out = pd.concat([out, table], axis=1)
    return out
print("Non-Recommended Items")

gram_table(data= df['Review Text'][df["Recommended IND"].astype(int) == 0], gram=[1,2,3,4,5], le
```

🔒 利用Keras搭建基于bert预训练得分类任务

```
from transformers import TFBertModel, BertTokenizer

#加载预训练的BERT模型和分词器
bert_model_name = 'bert-base-uncased'
tokenizer = BertTokenizer.from_pretrained(bert_model_name)
bert_model = TFBertModel.from_pretrained(bert_model_name)

#使用BERT的分词器处理文本数据，包括分词、添加特殊令牌、截断和填充
def encode_sentences(sentences, tokenizer, max_length=512):
    input_ids, attention_masks = [], []

    for sentence in sentences:
        encoded_dict = tokenizer.encode_plus(
            sentence,                      # 输入文本
            add_special_tokens = True,     # 添加特殊令牌
            max_length = max_length,       # 填充并截断长度
            pad_to_max_length = True,
            return_attention_mask = True,  # 返回attention mask
            return_tensors = 'tf',         # 返回tensorflow tensor
        )

        input_ids.append(encoded_dict['input_ids'])
        attention_masks.append(encoded_dict['attention_mask'])

    return tf.concat(input_ids, axis=0), tf.concat(attention_masks, axis=0)
```

```
#创建一个Keras模型，该模型包括BERT层和一个用于分类的输出层
import tensorflow as tf
from tensorflow.keras.layers import Dense, Input
from tensorflow.keras.models import Model

def build_model(bert_model, num_labels):
    input_ids = Input(shape=(None,), dtype=tf.int32, name="input_ids")
    attention_masks = Input(shape=(None,), dtype=tf.int32, name="attention_masks")

    output = bert_model([input_ids, attention_masks])[0]
    output = output[:, 0, :]  # 取CLS令牌的输出
    output = Dense(num_labels, activation='softmax')(output)

    model = Model(inputs=[input_ids, attention_masks], outputs=output)
    return model

num_labels = 2  # 根据您的任务自定义
model = build_model(bert_model, num_labels)

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

# 准备数据
sentences = ...  # 输入句子的列表
labels = ...     # 标签列表

input_ids, attention_masks = encode_sentences(sentences, tokenizer)
train_dataset = tf.data.Dataset.from_tensor_slices(({'input_ids': input_ids, 'attention_masks':

# 训练
model.fit(train_dataset, epochs=3, batch_size=32)
```