ML 2022 Fall

Project: 'Fit Feedback' Prediction

Xiao Li 34%,Yibo Chen 33%,Jiacheng Xu 33%

Completion Date .... Jan 11, 2023
Mentor .... Jie Wang

Abstract

Our group have tried three approaches to predict 'fit feedback' based on multi-attributes given. These methods include K-Nearest Neighbor, Naïve Bayes and TextCNN[1].

1 Introduction

These days, electronic commerce have been thriving for a long time. For platforms that provide such services, there is a realistic problem. How to recommend suitable clothes to potential customers? In this project, we are provided with a dataset containing 87,766 samples, each holds 15 attributes including 'item_name', 'review', 'fit' and etc. Our task is to predict 'fit'('True to Size', 'Small', 'Large') based on other 14 attributes. In practice, we don't have to make use of all features. Actually, KNN only use 10 features and TextCNN[1] only use the 'review' attribute whereas Naïve Bayes takes all features as input.

2 KNN Classifier

2.1 Formulation and Theory

The full name of KNN algorithm is the k-nearest neighbors algorithm, a supervised learning classifier, which uses proximity to make classifications or predictions about the grouping of an individual data point based on the fact that similar points are close to one another.

The reason why I choose this algorithm is that I believe people with similar figures, like 'weight','height','bust_size' and so on, are likely to wear the same size of the clothes. In addition, when they buy a similar cloth, they may send identical feedback.

After we choose the algorithm applied to this problem, we need to deal with the data set we got to train our model.

Steps of data processing:

1) Drop out data with empty 'fit' feature because they are useless in supervised learning.

2) Then we convert all string features to integer numbers. Fist, 'Small','True to Size','Large' is assigned to 1,2,3 as required. Next, We build a list without duplicated strings for each feature which contains all its values. And we replace the values with its index in the list.

3) We nee to find a metric for the similarity between different data. In geometric meaning, we call it distance. Euclidean distance is most commonly used, which is defined as $d(x, y) = \sqrt{\sum_{i=1}^{n}(y_i - x_i)^2}$. In this question, n equals 10, that is the graph is 10 dimensions.

2.2 Model and Code

- data_proc.py
  Clean the dataset and convert the type of the data
- kNN.py
  Implement KNN algorithm
- PB20000178.py
  Include main function
- Store_data.npy
  Store the supporting data
- Store_label.npy
  Store the supporting label
- train_proc.txt
  Remaining data after we dropout the empty feature 'fit'

2.3 Results and Analysis

The first generation of the code is the most common implementation of KNN algorithm. But it comes out that the result is terrible. I first realized that the data is extremely imbalanced. Label 1 has about 2500 pieces of data, label 2 has 16000 pieces and label 3 has 3000 pieces after data processing. We have to deal with the imbalanced data.

In the second generation of the code, I use random downsampling on label 2 and random oversampling on label 1 and label 3 to make them has the same size of data. Whereas it isn't ideal.

In the third generation of the code, I add a function in KNN.py called balance_method whose performance I satisfy with. Its principle is that we throw away the data not of much value which is already predicted right based on the former data and the remaining is our needed surporting data. We use these remaining data as our data space, and calculate the distance between a test data point and each point in the data space to find the k-nearest points(the value of k may affect the performance of the classifier which we need to adjust later).

	Small	True to Size	Large	average
f1-score	0.4562	0.7824	0.4688	0.5691

Table 1: f1-score of each label with Euclidean distance

In addition, I try to find whether different definition of distance will contribute differently to the performance. I change it to Manhattan distance, which is defined as $d(x, y) = \sum_{i=1}^{n} |y_i - x_i|$.

	Small	True to Size	Large	average
f1-score	0.4743	0.7809	0.4642	0.5732

Table 2: f1-score of each label with Manhattan distance

It can be noticed that f1-score of label 'Small' increases a bit and f1-score of label 'True to Size' and 'Large' decreases a little with Manhattan distance compared to Euclidean distance. Seeing the metric 'average', we can conclude that changing the definition of distance from Euclidean distance to Manhattan distance improves the performance of KNN classifier slightly.

3 Text CNN Classifier

3.1 Formulation and Theory

In natural language processing, we are always faced with a realistic problem: how to convert our input from text format to network friendly format. Generally there are 3 approaches; one-hot encoding, bag-of-words and word embedding. The most traditional and efficient approach is one-hot encoding. Its idea is that we build an one-dimension vector v for each word and v_i corresponds to a unique $word_i$. Then for $word_i$, its one-hot encoding is vector v with $v_i = 1, v_j = 0, j \neq i$. For example, we get a vocabulary of three words: man, boy, queen. Encoding for each word can be $[1, 0, 0], [0, 1, 0], [0, 0, 1]$. However there 2 major problems. As the size of vocabulary increases, the dimension will in increases and the cost that we represent a sentence will increases greatly. The other problem is that one-hot encoding does not hold any semantic content. Take the previous example, we have a reason to believe that the word 'man' is very close to 'boy' but distant to 'queen'. But in Euclidean distance, the distance between 'man' and 'boy' is the same as the distance between 'man' and 'queen'.

The second approach is bag-of-words. It is similar to histogram. The key idea is that we collect unique words from documents. When we build a dictionary of all words, we can construct an one-dimension vector for each document. The detailed approach is to count the number of times of each word appearing in the document. For example we have 2 documents *John likes to watch movies. Mary likes movies too.* and *Mary also likes to watch football games.*. We may get a dictionary like this: {"John", "likes", "to", "watch", "movies", "Mary", "too", "also", "football", "games"}. Then we get representations for each document:$[1, 2, 1, 1, 2, 1, 1, 0, 0, 0]$ and $[0, 1, 1, 1, 0, 1, 0, 1, 1, 1]$. Compared with one-hot encoding, bag-of-words has a lower costly representation, but it still holds no semantic content.

The third approach word embedding is exactly what we use for TextCNN[1]. Word embedding is proposed to deal with the difficulties in preserving semantic contents when converting text to vectors. The simplified steps are as following:

- First we remove the samples whose 'fit' is null string and the samples whose 'review' is too short. Note that the dataset is highly skewed, we need to balance it by undersample method after which we get a 1:1:1 dataset. Then we save this dataset for later TextCNN[1].
- We scan the whole sentence and get (focus word, context word) pairs. The distance

between focus word and context word in the sentence do not exceed the window size which is a hyperparameter.

- We convert each word to one-hot vector.
- We feed the data to a neural network with one hidden-layer with focus being the input and context word being the labels.
- We extract the weight of the hidden layer. w_i^1 is exactly the embedding vector for $word_i$.

The key idea of TextCNN[1] is considering each sentence x as a matrix $\in \mathbb{R}^{n \times k}$, where x_i is a k-dimensional word vector corresponding to the i-th word in the sentence. If the length exceeds n, we will only use the first n words. If the length is less than n, the sentence will be padded. Our input has a shape of (N, 1, H, W) to do convolution, which is quite different from computer vision tasks' (N,3, H, W).

3.2 Model and Code

The network to find embedding is a two-layer network. We choose random seed and normal distribution for weights initialization. For bias, we simply use zero initialization. Since the data we feed to this network is a huge sparse matrix , and my laptop has a very limited memory, I have to use only 2,100 samples to build vocabulary which certainly limit the performance in the later TextCNN[1]. It is worth noting that we do not build an end-to-end model, because we do not have a large memory and much high performance CPUs.

The corresponding code can be found in utils/vocab.py.

The TextCNN[1] accepts an input *embed_x* of shape (N, sentence_length, embedding_dim). Then we will reshape it to (N, 1, sentence_length, embedding_dim) and feed it to 3 parallel convolution layers with kernel size being 3, 4, 5 respectively. After convolution, each $conv\_x_i, i = 1, 2, 3$ is fed to a max-pooling layer, where their dimensions all reduce to (N, feature_maps, 1). And we then concatenates $pool\_x_i, i = 1, 2, 3$ and reshape it to (N, 3 $\times$ feature_maps). Then after a dropout layer and a fully connected layer, we finally get our prediction. Figure1 is a simplified demo to show the network architecture.

The corresponding code can be found in model.py.

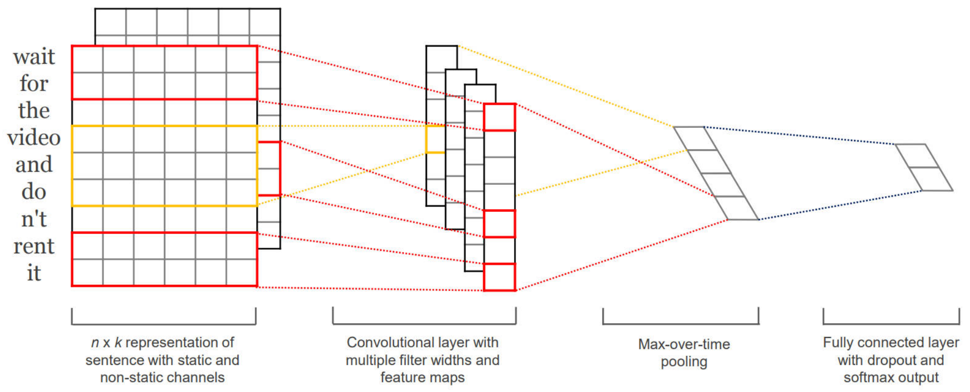
wait
for
the
video
and
do
n't
rent
it
n x *k* representation of sentence with static and non-static channels
Convolutional layer with multiple filter widths and feature maps
Max-over-time pooling
Fully connected layer with dropout and softmax output

Figure 1: Model architecture with two channels for an example sentence.

3.3 Results and Analysis

We use the attribute 'review' to build the vocabulary. Unfortunately, we are informed that in the test dataset there is no 'review' attribute on 1/10/2023. We do not have much time to adjust our data-processing workflow and hyperparameters. So this approach can not be used for score evaluation.

In practice, we are faced with a major problem: the lack of memory. As I have introduced in section3.1, when training the embedding, we use one-hot matrix, which is memory-costly. As a result, we have to choose a small window_size=2 and a simple network to find embedding, which limits the quality of the dataset. And the final performance is not very satisfying. Also, we find that the random seed for weight initialization has an significant effect on initial performance and gradient back propagation.

	Small	True to Size	Large
Precision	0.3989	0.4015	0.3870

Table 3: Performance of TextCNN Classifier

4 Naïve Bayes Classifier

4.1 Formulation and Theory

The basic idea of naïve bayes classifier is derived from the Bayes Theorem. In this classifier, to predict the fit feedback of a new record, we consider each word in the dictionary of a rental or purchase record as the attribute of a certain sample. So, all of those attributes can form a vector of the sample, noted as X. Then we set term $P(c)$ is the so-called prior probability of the class c, $c \in C = \{‘Small’, ‘TruetoSize’, ‘Large’\}$. According to the Bayes Theorem, what we need to do is find the c making the $P(X|c)P(c)$ get its maximum, in which $P(X|c)P(c)$ is the product of all the $P(w_k|c)$ which is the probability applied the Laplace smoothing technique of the a certain word contained in the sample of a c class.

Then, we optimized the structure of the classifier. It is obvious that the number of samples whose fit is ‘True to Size’ is about four times more than that of ‘Small’ and ‘Large’ which are approximately equal. And in fact, the size of vocabulary of each class will certainly affect the result of the classifier. So, we divided the ‘True to Size’ samples into five parts, each of which together with ‘Small’ and ‘Large’ samples forms a training dataset to train a classifier. Finally, we combine the five classifiers we get together in a simple way. The class of a sample be predicted by most classifiers is the result.

4.2 Model and Code

```
def set_divide(set,data):
   #divide set to trainset and testset
   #testset has 900 images for each kind of label occupying 1/3
   #other samples of the label “true to size” are divided into 5 part, each part
     together with samples of “small” and “large” consititutes a train set.
   for i in range(len(set)):
      if data.iat[i,2] == "True to Size":
         if testnum2 < 300:
            test_keyset.append(i)
         else:
            temp_keyset.append(i)
         testnum2 += 1
            ....
      #small and large are similar
            ...
   temp_keyset = np.array(temp_keyset)
   train_keyset = np.array_split(temp_keyset,5)
   a = np.array(temp1_keyset)
   for i in range(5):
      train_keyset[i] = np.append(train_keyset[i],a)
   return test_keyset,train_keyset

def train(set,train_keyset):
    #divide trainset into 3 part by the label
    for i in train_keyset:
        words = set[i].split(',')#divide line by  ','
        if data.iat[i,2] == "True to Size":
            word = split2word(words)
            T2S_set += word
            T2S_num += 1
          . . . .
        #small and large are similar
                . . .
    #train
    V_set = list(dict.fromkeys(T2S_set + S_set + L_set))
    V = len(V_set)
    D = len(train_keyset)
    #true to size
    P_t2s = T2S_num/D
    n_t2s = len(T2S_set)
    n_t2s_dict = Counter(T2S_set)
    P_k_t2s = {}
    for word , value in n_t2s_dict.items():
        value = (value + 1)/(n_t2s + V)
        P_k_t2s[word] = value
  . . . .
    #small and large are similar
     . . .
    P_list = [P_s,P_t2s,P_l]
    P_k_list = []
    P_k_list.append(P_k_s)
    P_k_list.append(P_k_t2s)
    P_k_list.append(P_k_l)
    return P_list, P_k_list

def Predict(word_list, P_list, P_k_list):
    predict_s = 0
    predict_t2s = 0
    predict_l = 0
    for j in range(5):
        y_t2s = math.log10(P_list[j][1])
        y_s = math.log10(P_list[j][0])
        y_l = math.log10(P_list[j][2])
        #five predictor works individually
        for i in range(len(word_list)):
            temp1 = P_k_list[j][0].get(word_list[i], 1)
            temp2 = P_k_list[j][1].get(word_list[i], 1)
            temp3 = P_k_list[j][2].get(word_list[i], 1)
            if temp1 == 1 or temp2 == 1 or temp3 ==1: continue
            y_s += math.log10(temp1)
            y_t2s += math.log10(temp2)
            y_l += math.log10(temp3)
```

```python
      if y_s > y_t2s:
         predict_s += 1
      elif y_t2s > y_l:
         predict_t2s += 1
      else:
         predict_l += 1
   predict = 0#final result depend on the number of the result of each predictor
   if predict_s > predict_t2s:
      predict = 1
   elif predict_t2s >= predict_l:
      predict = 2
   else:
      predict = 3
   return predict-1

set = []
f = open(r"C:/Users/86189/PycharmProjects/MLproject/Machine-Learning/cyb/data_proc.txt"
    ,'r',encoding="utf-8")
set = f.readlines()
del(set [0])
f.close()
data = pd.read_csv('data_proc.txt')
test_keyset, train_keyset = set_divide(set, data)
P_list = []
P_k_list = []
for i in range(5):
   x, y = train(set,train_keyset[i])
   P_list.append(x)
   P_k_list.append(y)
confusion_matrix = test(set,test_keyset)
· · ·
#calculate accuracy and recall through confusion matrix
· · ·
```

4.3 Results and Analysis

We use confusion matrix for ternary classifier to analysis the result of naïve bayes classifier.

In the initial experiment, when classifier hasn’ t been optimized, we only use the samples having all of features and label. So, the size of dataset is about 22000, among which the first 3000 samples are divided into test set, and the rest is the train set. We get the results shown in Table 4.

	Small	True to Size	Large
Precision	0.3125	0.7329	0.2000
Recall	0.0140	0.9952	0.0024

Table 4: Performance of Naïve Bayes Classifier

In fact, the reason of this result is most of the prediction is ‘True to Size’ and the ‘True to Size’ samples occupy most of the test set. So, we change the train set to a set containing 3000 samples among which each class is 1000. And we get the result shown in Table 5.

	Small	True to Size	Large
Precision	0.6129	0.3363	0.8261
Recall	0.0190	0.9910	0.0190

Table 5: Performance of Naïve Bayes Classifier over Balanced Dataset

This performance is truly awful, so we optimized the classifier in the way mentioned in part 2, Formulation and Theory. we only use the samples having all of label and features, except ‘username’, 'review_summary', 'review', 'rating'. And we adjust the size of test set to 900, in which the number of samples of each class is 300. Then the result is shown in Table 6.

	Small	True to Size	Large
Precision	0.3819	0.6884	0.4400
Recall	0.8567	0.0900	0.2567

Table 6: Performance of Optimized Naïve Bayes Classifier under Test Data Condition

It is still awful. Then we try to use this classifier to extend the data set. Because for some training samples, some features and even labels are missing, if we can fill the label of samples whose fit is missing, we can get more data to train other model. Then we use the samples with all features and label available to train the classifier. The size of test set is still 900. The result is shown in Table 7.

	Small	True to Size	Large
Precision	0.5326	0.6736	0.5912
Recall	0.8433	0.6467	0.2700

Table 7: Performance of Optimized Naïve Bayes Classifier under Train Data Condition

This result is much better but still cannot satisfy the requirement of being used to extend data set.

Finally, the naïve bayes classifier is abandoned.

5 Conclusion

Our group have tried three approaches to predict 'fit feedback' based on multi-attributes given. These methods include K-Nearest Neighbor, Naïve Bayes and TextCNN. After we test these three approaches, we finally choose to use KNN classifier to predict the label which performs best.

6 Acknowledgement

during the past semester, we have learned a lot of machine learning. In this project, we have a good chance to apply what we have learned to practice and deepen our understanding. Thanks for the teacher and TAs' efforts.

References

[1] Yoon Kim. Convolutional neural networks for sentence classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1746–1751, Doha, Qatar, October 2014. Association for Computational Linguistics.