【技术解读】大赛TOP团队方案技巧大揭秘！

腾讯广告算法大赛 3天前

以下文章来源于机器学习指北 ， 作者徐安

机器学习指北
用数据去描绘行为和人群，你是写实派还是写意派呢？

本文作者：徐安，2018届腾讯广告算法大赛亚军

> 2018 年和 2019 年腾讯算法广告大赛都可以看做推荐系统问题。这类问题最重要的特征是点击率，最大的难点是冷启动。文本结合 2018 年比赛亚军方案和 2019 年比赛冠军方案中的一部分技巧，提出了一种新的点击率建模方案，试图解决一部分冷启动问题。该方案复杂度很低，实现简单，效果好。

1 问题介绍

推荐系统和广告算法中，对于新用户或者新内容，记录很少，如果我们直接将历史点击率作为特征，会存在问题。比如

1，新用户 A 有 2 条浏览记录，1 次点击，转化率 50%,
2，老用户 B 有 2 条浏览记录，0 次点击，转化率 0%。

A 和 B，只因为 1 次点击，点击率就相差 50%，这不合理。显然，问题出现在 A,B 用户都是新用户，他们的历史数据太少了，历史点击率自然不准。

这就像我告诉同事小明：我王者荣耀贼溜，后羿 100%胜率。实际上，我只打了两盘后羿，其中一盘还是新手教学。同事小明可能会多嘴问一句：你打了几盘后羿啊？但是模型不会，没有专门调整过的模型只会默默接受我后羿 100%胜率的设定，然后给我匹配一堆王者选手。这就是冷启动问题。

2 解决思路

已经 0202 年了，我们人类星球上的人工智能模型的计算能力还可以，但还是太“老实”，太“傻”。所以，解决上述问题的方法就是：**直接把预测结果告诉模型，别让模型自己去算，去猜**。这显然是句废话，不过翻译成学术语言就不是了：**给模型输入概率，而不是频率**。

所以最好的办法是，利用用户的历史点击率，去计算用户之后点击的概率，再将这个概率输入模型。通过用户 A 的历史点击频率去计算用户 A 之后点击的概率，听起来不错，但又不太可行，因为这里的信息太少了。好在我们有所有数据，**用所有用户的历史点击频率去预测用户 A 之后点击的概率**，似乎有点希望。

3 贝叶斯平滑

新用户 A 只有两条浏览记录，模型还不够认识用户 A，如何办？如果 A 用户能多几次浏览记录就好了。可是去哪里找那不存在的浏览记录呢？我们可以假定用户 A 和其他所有用户是差不多，用其他用户的历史数据“构造”一些浏览记录，作为新用户 A 的浏览记录。这里“构造”出来的记录，可以理解成先验知识。当我们见过了很多用户之后，即使不认识新用户 A，也会对 A 有个大概的“预期值”。贝叶斯平滑就是这样工作的，它通过“观测”所有用户数据，为新用户确定一个初始预期值，这个预期值就是“先验”。而用户 A 自己真实的行为所产生的“预期值”，被称之为后验。最后我们将先验和后验综合起来，计算一个贝叶斯平滑修正过的点击率。

贝叶斯平滑的推导比较繁杂，也不是本文的重点，有兴趣的话，可以查看：
https://www.jianshu.com/p/c98f3bb48b97

有了贝叶斯平滑，我们可以对点击率进行修正，让历史转化率这个频率值，更加接近用户真实点击的概率。

4 连续值与深度学习

通过上文，我们可以得到一个贝叶斯平滑后的点击率，那么直接把点击率特征输入深度神经网络，问题不就解决了吗？

只能说，对于大多数普通特征也许可以这样，但是转化率这种强特征，这样做太浪费了。原因如下：

近年来，推荐系统相关的深度网络模型层出不穷：
DeepFM，Wide & Deep，Deep & Cross Network，Attentional Factorization Machine，xDeepFM，Deep Interest Network，AutoInt...

这些模型都有个共同特点：拥有 FM 层或者 Attention 层（Wide & Deep 除外）。FM 层和 Attention 层都能有效进行特征交叉，从而提高了模型精度。FM 层和 Attention 层的输入都是向

量，所以这些模型基本都需要让特征先进入嵌入层，变成一个向量，再参与后面的特征自动交叉。

这时候，连续值特征就会很尴尬，他们无法像离散值特征一样进入嵌入层，从而无法参与后面的特征交叉，效果大打折扣。

目前将连续值转化成向量的解决方案主要有以下几种：

第一种方式是对连续值特征做离散化分桶，之后将分桶后的离散值输入到嵌入层到的嵌入向量。分桶本质上就是做四舍五入近似，等距分桶是直接四舍五入，等频分桶是排序后对序做四舍五入，这两种方法会影响精度。因为近似必然会损失信息。

第二种方式也是离散化，不过是有监督的离散化。它借鉴了决策树的思路，枚举所有分割点，找到一组分割点，使分割后的数据组的信息熵增益最大。这里有个比较 trick 的做法：直接用一部分数据训练一个 lightGBM 模型，然后解析模型文件，里面记录了 lightGBM 模型选出来的最优分割点，直接可以用。需要注意的是，有监督的离散化用到了数据的标签，所以可能会带来数据穿越。为了避免这个问题，建议训练 lightGBM 模型的数据和深度学习的数据不要重合。

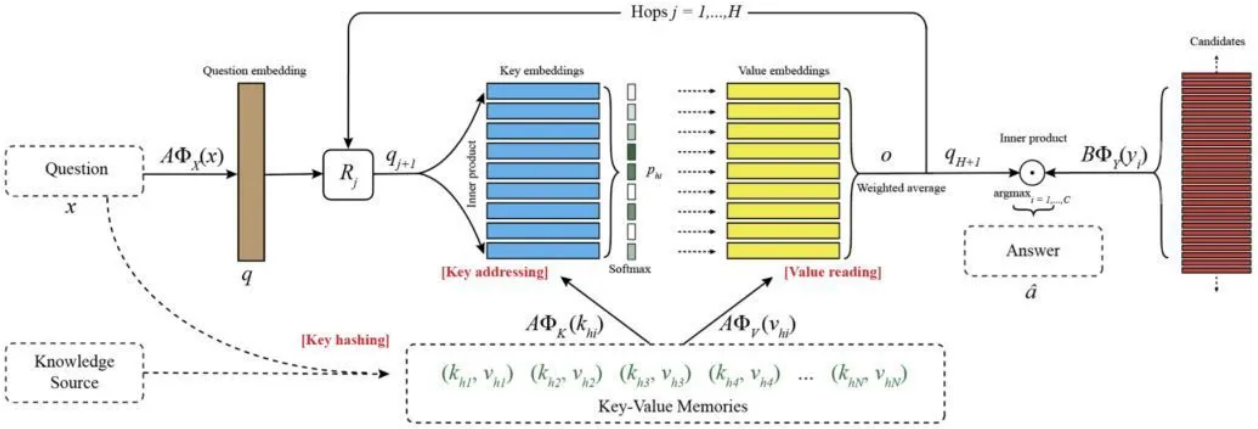
第三种方法来自 AutoInt 论文，非常有趣。它先用前面两种方法对连续值特征 Z 做离散化，得到 Z'，之后将 Z'输入嵌入层得到嵌入向量 emd(Z')，最后用嵌入层的输出 emd(Z')再乘以 Z。想法很巧妙：既然离散化后的特征会损失精度，那么就将原始特征再一次输入模型。
最后一种方法来自 2019 年比赛冠军成员郭大，下文重点介绍。

5 键值储存网络

该方法灵感来自 NLP 顶会 ACL2016 的论文《Key-Value Memory Networks for Directly Reading Documents》。

文章为深度网络引入了记忆模块，原本是用来解决 QA 问答问题的，不过简单改进后，可以用来将构造续值特征的专属嵌入层。（目前推荐系统很多好的 idea，都来自 NLP 和 CV 领域。所以，学习推荐系统或者广告算法，了解 NLP 和 CV 领域的前沿研究成果也很重要）

不多说，键值储存网络(Key-Value Memory Networks)结构如下：

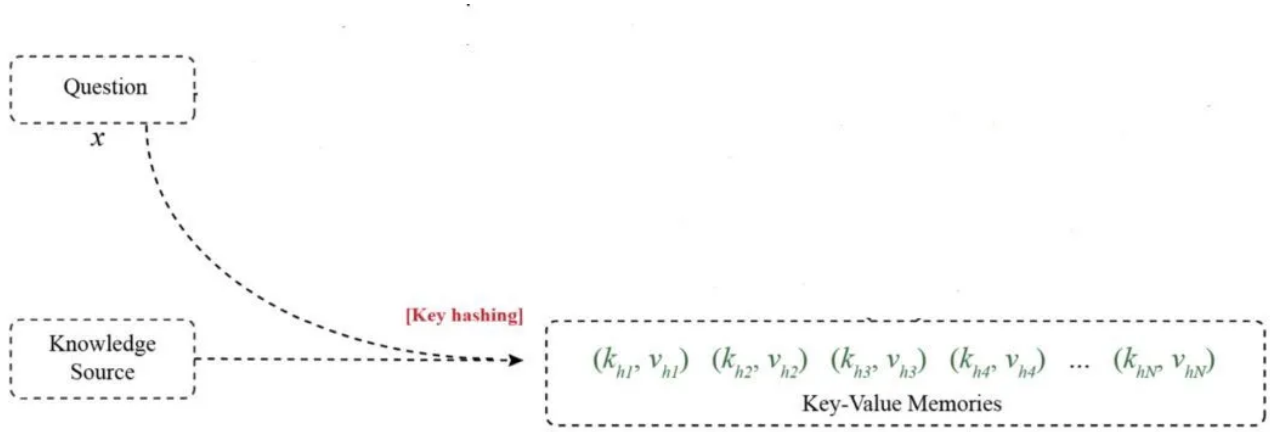


图一:Key-Value Memory Networks

键值储存网络与普通网络最大的区别是可以方便的引入先验知识，即图中的 Knowledge Source 模块。该模块相当于一个内嵌在神经网络中的“搜索引擎”，对于输入的任何一个 Question，先在 Knowledge Source 中做一次搜索，然后将搜索结果也作为神经网络的输入。为什么说它是内嵌在神经网络中呢？因为搜索结果与 Question 之间存在一个相似度，这个相似度的计算是依赖神经网络的，它可以享受梯度下降带来的优化。

模型主要分为三个部分：Key hashing,Key addressing 和 Value reading。

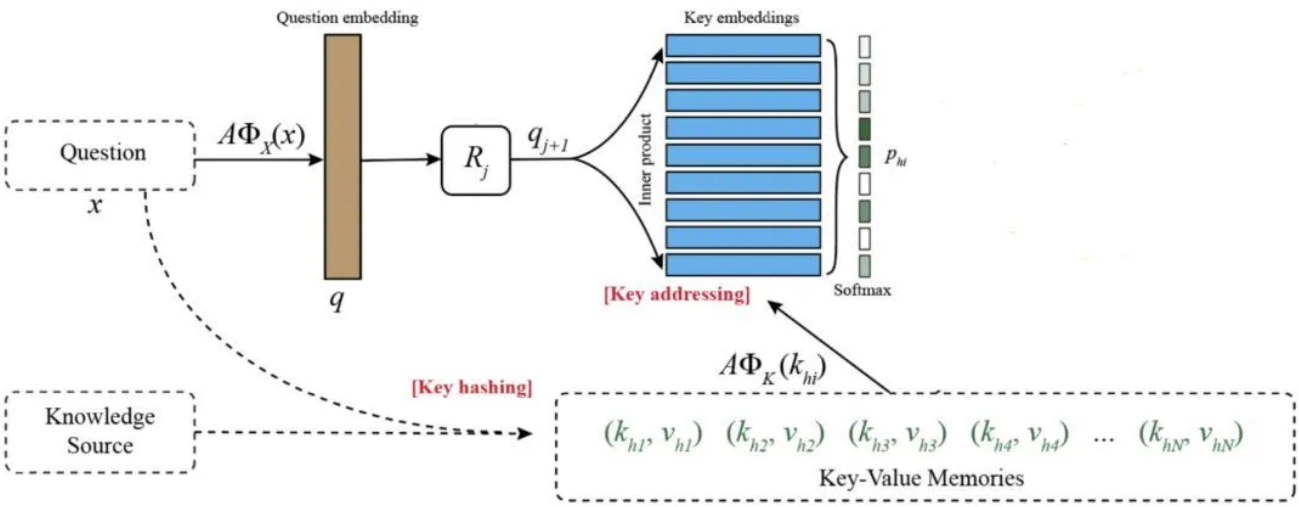
Key hashing



图二:Key hashing

Key hashing 是离线计算好的。它的输入是 Question 和 Knowlege Source。Question 是 QA 问答问题中的提问，比如“如何打开企业微信”。Knowlege Source 是一个类似维基百科的数据库，里面记录了各种词汇，实体和知识。
Key hashing 就是把所有 Question 里面的常用词（出现次数大于某个阈值）挑出来，然后给这些词一个编号，组成一个字典。字典的 key 是这些常用词，value 是常用词编号。

Key addressing



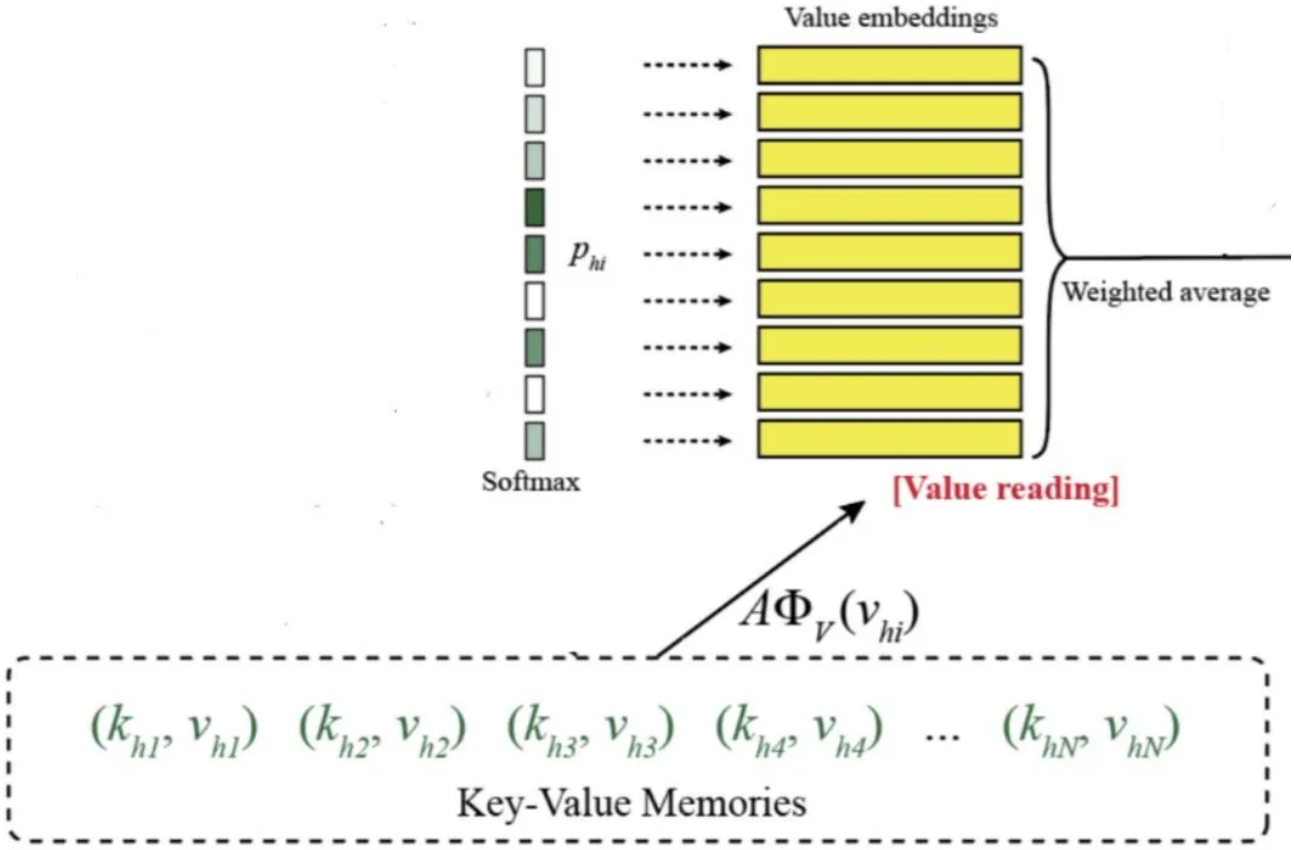
图三:Key addressing

Key addressing 就是去 Knowlege Source 里面寻找 Question 中的词汇和短语。比如找到了："企业","微信","企业微信","如何"等词。Question“如何打开企业微信”会有一个训练出的 Question embedding 值，同时，"企业","微信","企业微信","如何"等词也都有各自的 embedding 值，被称为 Key embeddings。用 Question embedding 分别乘以每一个 Key embeddings，再经过一次 Softmax，就可以得到 Question 与各个 Key 的相似度权重 P。具体公式如下：

$$p_{h_i} = \text{Softmax}(A\Phi_X(x) \cdot A\Phi_K(k_{h_i}))$$

Key hashing 和 Key addressing 用上述模型解决了一个问题：Question 与 Knowlege Source 中相近词汇的相关性。比如对于 Question"如何打开企业微信"，可以得到一个相似度权重 P 字典{"企业":0.4,"微信":0.2,"企业微信":0.3,"如何":0.1}。

Value reading



图四:value reading

value reading 是键值储存网络的核心部分。还记得我们上文有个 key embedding，对应的，Key-Value Memories 还有个 value embedding，它的输入是 Knowlege Source 里面每个词的 id。对 value embedding 以上文的 p 为权重加权求和，便得到我们需要的向量 o。

$$o = \sum_i p_{h_i} A \Phi_V \left(v_{h_i} \right)$$

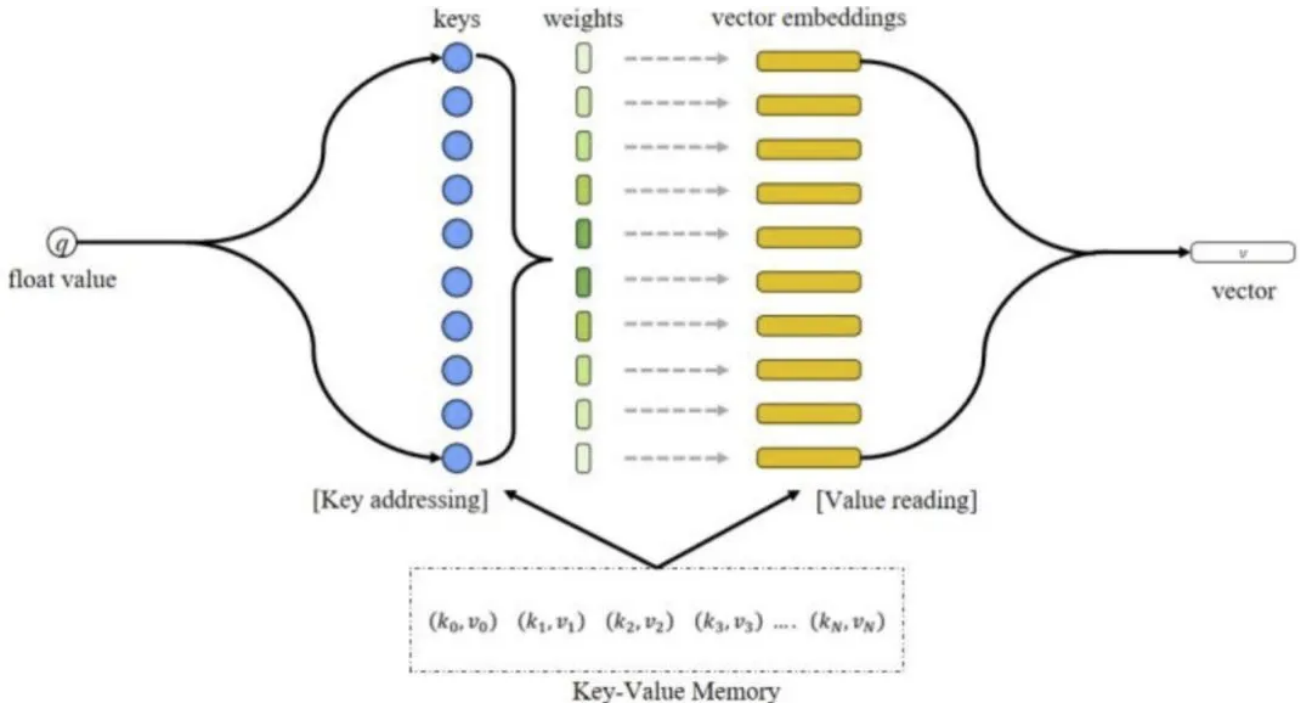
优势
和传统的深度神经网络比，键值储存网络可以方便的让先验知识以键值对的方式输入模型(图中的 Key-Value Memories)。这意味着，神经网络的输入值可以直接是多个键值对组成的字典。
举个例子，传统神经网络只能将数字作为特征，比如:身高(173),体重(90)或者年龄(25)。而键值储存网络可以将兴趣 ({‘篮球’:0.5,'足球':0.2,'台球':0.1}) 作为特征，直接输入给模型。字典特征的 key 是实体，使用 LabelEncoding 后可以进入嵌入层，value 是其权重。键值储存网络可以方便的将输入的字典特征转化成上文的向量 o。

⑥ 连续值键值储存网络

回到最开始的问题，我们想找到一个将连续值转成向量的方法，但上文却一直在讲键值储存网络，为什么?

因为键值储存网络实现了字典特征->向量的转换，我们希望的是连续值->向量的转换。所以，借助键值储存网络，只需要再实现连续值->字典特征的转化就大功告成了。连续值->向量很难，但是连续值->字典特征方式很多，易于实现。

假定有了连续值->字典特征的转化，那么总体架构和键值记忆网络基本一致，如下图所示:



图五:连续值键值储存网络

连续值->字典特征的转化即图中的 Key-Value Memory,如何实现这部分应当结合具体的业务场景，数据分布。这里先介绍下郭大的做法吧：

将连续值特征缩放至[0,1]区间在[0,1]区间找 n 等分点，比如 n=6 时，就是(0, 0.2, 0.4, 0.6, 0.8, 1)
依次计算连续值 x 与 n 等分点的距离，比如 x=0.3，n=6，就是(0.3, 0.1, 0.1, 0.3, 0.5,0.7)，之后构造字典特征{0:0.3, 1:0.1, 2:0.1, 3:0.3, 4:0.5, 5:0.7}
对字典特征的 value 取倒数后 softmax，具体相似度公式如下：

$$w_i = softmax(\frac{1}{|q-k_i+10^{-15}|})$$

python 伪代码: {i: softmax(1/(q-i/n+1e-15)) for i in range(n+1)},其中 q 为浮点数特征，n 为等分点个数。这里加上 1e-15 是为了防止 q 正好等于某个等分点时，分母为 0。

郭大的方法将字典特征的 key 定义为[0,1]区间的等分点，之后对浮点数与各等分点的距离做取倒和 softmax 变换。取倒是为了保证浮点数越接近等分点，权重越大。softmax 变换是为了保证所有权重之和为 1。

实践中发现，当 q 与某个等分点较接近时，value 中除该等分点对应的值外，都非常接近 0。这主要是因为 softmax 函数会指数级加大距离间差异。

为了缓解这种情况，我在最近的代码里使用如下相似度公式：

$$w_i = \frac{\left(\frac{1}{q-k_i+10^{-15}}\right)^2}{\sum_{i=1}^{i=n}\left(\frac{1}{q-k_i+10^{-15}}\right)^2}$$

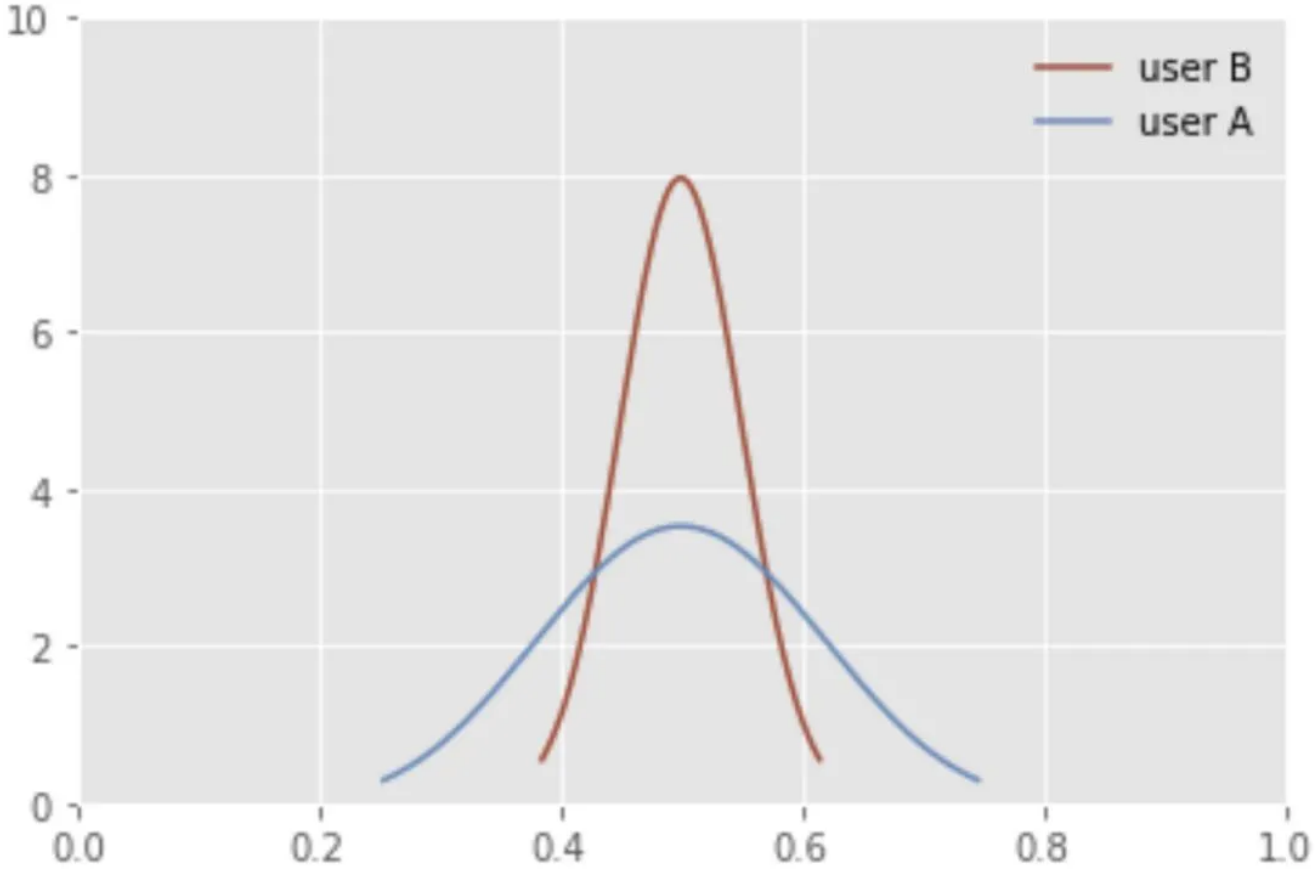
该公式取距离平方反比为权值，之后将权值缩放至总和为 1。用该公式得到的权值比较"分散",可以让模型更好的学习那些冷门分位数的嵌入表示。

⑦ 概率分布特征

截至目前，文章讲了点击率特征的贝叶斯平滑，以及如何在不损失精度的情况下把浮点数特征（比如点击率特征）输入神经网络。

如果把点击率看成一个普通浮点数，问题已经解决。但是点击率并不普通，点击率可以被认为是用户是否点击广告这个随机变量的期望值。

用户是否点击广告实际上是一个随机变量，点击率就是用这个随机变量的期望值作为特征，去描述它。这样做实际上是用一个值去代表一个复杂的分布，必然会带来信息损失。举个例子，A 用户浏览 20 次，点击 10 次。B 用户浏览 100 次，点击 50 次。A 和 B 的点击率都是 50%，但是他们是否点击广告的概率分布却大不一样：



图六:用户A和用户B否点击广告的概率分布

虽然 AB 两用户点击率都是 50%，但是 B 用户点击次数更多，所以 B 用户的点击率更置信，B 用户的概率分布也更集中。这就体现出点击率特征的弊端，它只能描述概率分布的期望，无法完整描述概率分布。

我们希望完整描述概率分布给模型，我们希望精确区分出点击率很相似但总浏览数差异很大的那群人。这个问题可以被定义为如何向模型描述一个概率分布。

用户是否点击广告的概率分布是连续的，用概率密度函数可以表示。可以对概率密度函数函数进行分段近似，分别统计它在[0,0.1),[0.1,0.2),[0.2,0.3),[0.3,0.4)...区间的平均值，用这些平均值来表示这个分布。形式如下：

{[0,0.1):0.1,[0.1,0.2):0.2,[0.2,0.3):0.4,[0.3,0.4):0.4,...}

该形式其实也是字典特征，它的 key 是区间，value 是点击率这个随机变量落在各区间的概率。如此一来，可以直接将这个字典特征输入键值储存网络。这种方式利用随机变量的概率分布，跳过了连续值->字典特征这一步，直接做随机变量->字典特征，避免了上文中的人工设计相似度公式。

如果构造的特征可以被看做是随机变量，那么就可以利用数学工具得到他的概率分布，概率分布分段近似得到字典特征，最后将字典特征输入键值储存网络。

8 代码实现与复杂度分析

上文的方法在代码实现上很容易，用途广泛（任何使用了嵌入层的网络都可以用）。

代码主要有四部分：贝叶斯平滑，随机变量->字典特征的转换，浮点数>字典特征的转换和键值储存网络。

相关第三方库

```
import numpy as np
import random
import pandas as pd
import scipy.special as special
from sklearn.model_selection import StratifiedKFold,KFold
from scipy import stats
from collections import OrderedDict, namedtuple
from itertools import chain
from tensorflow.python.keras.initializers import RandomNormal
from tensorflow.python.keras.layers import  Embedding, Input, Flatten
from tensorflow.python.keras.regularizers import l2
```

贝叶斯平滑

```
class BayesianSmoothing(object):
     def __init__(self, alpha, beta):
          self.alpha = alpha
          self.beta = beta

     def sample(self, alpha, beta, num, imp_upperbound):
          sample = np.random.beta(alpha, beta, num)
          I = []
          C = []
          for clk_rt in sample:
               imp = random.random() * imp_upperbound
               clk = imp * clk_rt
               I.append(int(imp))
               C.append(int(clk))
          return I, C

     def update(self, imps, clks, iter_num, epsilon):
          for i in range(iter_num):
               new_alpha, new_beta = self.__fixed_point_iteration(imps, clks, self
               if abs(new_alpha-self.alpha)<epsilon and abs(new_beta-self.beta)<eps
                    break
               self.alpha = new_alpha
```

```
            self.beta = new_beta

    def __fixed_point_iteration(self, imps, clks, alpha, beta):
        numerator_alpha = 0.0
        numerator_beta = 0.0
        denominator = 0.0

        for i in range(len(imps)):
            numerator_alpha += (special.digamma(clks[i]+alpha) - special.digamma
            numerator_beta += (special.digamma(imps[i]-clks[i]+beta) - special.d
            denominator += (special.digamma(imps[i]+alpha+beta) - special.digamm

        return alpha*(numerator_alpha/denominator), beta*(numerator_beta/denomina
```

随机变量（beta 分布）->字典特征

```
def beta_ppf(alpha, beta, dim):
    return stats.beta(alpha, beta).ppf([x/(dim+1) for x in range(0,dim+2)])

def beta_prior_feat_2_vec(data, key_col, count_col, sum_col, dim):
    data_simple = data.drop_duplicates([key_col],keep='last')
    bs = BayesianSmoothing(1, 1)
    bs.update(data_simple[count_col].values, data_simple[sum_col].values, 1000, 0.
    if np.isnan(bs.alpha) or np.isnan(bs.beta):
        bs.alpha, bs.beta = 0, 0

    data[key_col+'_beta_cdf_value'] = list(
        map(lambda x,y:beta_cdf(x,y,dim), data[sum_col]+bs.alpha, data[count_col]
    data[key_col + '_beta_ppf_value'] = list(
        map(lambda x,y:beta_ppf(x,y,dim), data[sum_col] + bs.alpha, data[count_c
    data[key_col + '_beta_key'] = [np.array([i for i in range(dim)]) for _ in ra
    return data[key_col+'_beta_cdf_value'].values, data[key_col + '_beta_ppf_value
```

浮点数->字典特征

```
def numpy_softmax(x):
    orig_shape = x.shape
    if len(x.shape) > 1:
        exp_minmax = lambda x: np.exp(x - np.max(x))
        denom = lambda x: 1.0 / np.sum(x)
```

```
            x = np.apply_along_axis(exp_minmax,1,x)
            denominator = np.apply_along_axis(denom,1,x)
            if len(denominator.shape) == 1:
                denominator = denominator.reshape((denominator.shape[0],1))
            x = x * denominator
        else:
            x_max = np.max(x)
            x = x - x_max
            numerator = np.exp(x)
            denominator =  1.0 / np.sum(numerator)
            x = numerator.dot(denominator)
        assert x.shape == orig_shape
        return x

    def float2vec(float_feat, bar_num = 20, method = 'gravitation'):
        float_feat = (float_feat-np.min(float_feat))*1.0 / np.max(float_feat-np.min(fl
        key_array = np.array([[i*1.0/(bar_num + 1) for i in range(bar_num + 1)]] * l
        value_array = None
        if method == 'gravitation':
            value_array = 1/(np.abs(key_array - float_feat[:,None] + 0.00001))**2
            value_array = value_array/np.sum(value_array,axis=1, keepdims=True)
        if method == 'sofmax':
            value_array = 1 / np.abs(key_array - float_feat[:, None] + 0.00001)
            value_array = numpy_softmax(value_array)
        return key_array,value_array
```

网络结构

```
    def get_varlen_multiply_list(embedding_dict, features, varlen_sparse_feature_column
        multiply_vec_list = []
        print(embedding_dict)
        for key_feature in varlen_sparse_feature_columns_name_dict:
            for value_feature in varlen_sparse_feature_columns_name_dict[key_feature]
                key_feature_length_name = key_feature.name + '_seq_length'
                if isinstance(value_feature, VarLenSparseFeat):
                    value_input = embedding_dict[value_feature.name]
                elif isinstance(value_feature, DenseFeat):
                    value_input = features[value_feature.name]
                else:
                    raise TypeError("Invalid feature column type,got",type(value_fe
```

```
            if key_feature_length_name in features:
                varlen_vec = SequenceMultiplyLayer(supports_masking=False)(
                    [embedding_dict[key_feature.name], features[key_feature_le
                vec = SequencePoolingLayer('sum', supports_masking=False)(
                    [varlen_vec, features[key_feature_length_name]])
            else:
                varlen_vec = SequenceMultiplyLayer(supports_masking=True)(
                    [embedding_dict[key_feature.name], value_input])
                vec = SequencePoolingLayer('sum', supports_masking=True)( varle
            multiply_vec_list.append(vec)
    return multiply_vec_list

class SequenceMultiplyLayer(Layer):

    def __init__(self, supports_masking, **kwargs):
        super(SequenceMultiplyLayer, self).__init__(**kwargs)
        self.supports_masking = supports_masking

    def build(self, input_shape):
        if not self.supports_masking:
            self.seq_len_max = int(input_shape[0][1])
        super(SequenceMultiplyLayer, self).build(
            input_shape)  # Be sure to call this somewhere!

    def call(self, input_list, mask=None, **kwargs):
        if self.supports_masking:
            if mask is None:
                raise ValueError(
                    "When supports_masking=True,input must support masking")
            key_input, value_input = input_list
            mask = tf.cast(mask[0], tf.float32)
            mask = tf.expand_dims(mask, axis=2)
        else:
            key_input, key_length_input, value_input = input_list
            mask = tf.sequence_mask(key_length_input,
                                    self.seq_len_max, dtype=tf.float32)
            mask = tf.transpose(mask, (0, 2, 1))

        embedding_size = key_input.shape[-1]
        mask = tf.tile(mask, [1, 1, embedding_size])
```

```
            key_input *= mask
            if len(tf.shape(value_input)) == 2:
                value_input = tf.expand_dims(value_input, axis=2)
                value_input = tf.tile(value_input, [1, 1, embedding_size])
            return tf.multiply(key_input,value_input)

        def compute_output_shape(self, input_shape):
            return input_shape[0]

        def compute_mask(self, inputs, mask):
            if self.supports_masking:
                return mask[0]
            else:
                return None

        def get_config(self, ):
            config = {'supports_masking': self.supports_masking}
            base_config = super(SequenceMultiplyLayer, self).get_config()
            return dict(list(base_config.items()) + list(config.items()))
```

上述代码除贝叶斯平滑转载自网络外，均为原创。

改进后的键值网络与连续值离散化后接入嵌入层的方法相比，没有增加训练参数，只是多做了一次向量加权求和，多增加了一些权重的输入。另一方面，改进后的键值网络中，分位数或者概率区间个数是可以人工调整的，当分位数或者概率区间个数为 1 时，该方法就退化成离散化后接入嵌入层。