Foursquare - Location Matching

Match point of interest data across datasets

U.S. Patent Phrase to Phrase Matching | Kaggle

比赛介绍

商业兴趣点 (POI) 是电子地图上的某个地标，用以标示出该地所代表的政府部门、商业机构、旅游景点、交通设施等处所。POI上的大规模数据集可以包含丰富的真实信息。其数据必须通过来自多个来源的及时更新进行匹配和去重。重复数据删除涉及许多挑战，因为原始数据可能包含噪声、非结构化信息以及不完整或不准确的属性。**机器学习算法和严格的人工验证方法相结合是消除数据集的最佳选择。**

本次比赛中，**参赛者将匹配超过一百万个 Places 条目的数据集的 POI**，这些条目经过大量修改以包含噪声、重复、无关或不正确的信息，您将生成一种算法来预测哪些 Place 条目代表相同的兴趣点。每个 Place 条目都包含名称、街道地址和坐标等属性。成功的提交将确定最准确的匹配项。

- 竞赛类型：本次竞赛属于**数据挖掘/匹配**，所以推荐使用的模型或者库：**Bert/XGBoost/LightGBM**
- 赛题数据：**数据包括全球数十万个商业兴趣点（POI）的一百五十多万个地点条目。**你的任务是确定哪些地方条目描述了同一个兴趣点。尽管这些数据条目可能代表或类似于真实地点的条目，但它们也可能包含人工信息或额外的噪音。
- 评估标准：提交是通过ground-truth词目匹配和预测词目匹配的平均交集比联合（**IoU**，又名Jaccard指数）来评估的。 平均值是按样本计算的，这意味着对提交文件中的每一行计算 IoU 分数，最终分数是它们的平均值。详见：Foursquare - Location Matching | Kaggle
- 推荐阅读 Kaggle 内的一篇 EDA（探索数据分析）来获取一些预备知识：Foursquare: Starter EDA | Kaggle

数据说明

数据包括全球数十万个商业兴趣点（POI）的一百五十多万个地点条目。你的任务是确定哪些地方条目描述了同一个兴趣点。尽管这些数据条目可能代表或类似于真实地点的条目，但它们也可能包含人工信息或额外的噪音。

官方数据页面 Foursquare - Location Matching | Kaggle

- **train.csv** - 训练集，包含超过 100 万个地点词目的 11 个属性字段。 主键是 id 和 point_of_interest 。
- **pair.csv** - 一组预先生成的地点词目对，旨在改进匹配的检测。
- **test.csv** - 和示例 5 行的真实测试集的样子。
- **sample_submission.csv** - 预期提交格式的示例。

train.csv

- **id** - 每个词目的唯一标识符。
- **point_of_interest** - 条目表示的 POI 的标识符。 可能有一个或多个实例描述相同的 POI。 当两个实例描述一个共同的POI 时，它们“匹配”。

sample_submission.csv

- **id** - 地点词目的唯一标识符，一个用于测试集中的每个实例。
- **matches** - 以空格分隔的 ID 列表，用于匹配给定 ID 的测试集中的实例。 地方条目总是自匹配的。

为了帮助参赛者编写提交代码，kaggle提供了一些从测试集中选择的示例。 当参赛者提交笔记本进行评分时，此示例数据将替换为实际测试数据。 **实际测试集大约有 600,000 个地点实例**，其中的 POI 与训练集中的 POI 不同。

解决方案思路

credit https://www.kaggle.com/competitions/foursquare-location-matching/discussion/335924

思路总结

- 官方数据在city、states、country有很多缺失值，使用最邻近的（经度和纬度）5个点为BERT模型填充NaN文本。
- 将各种表格数据concat成文本，再加上经度和纬度信息，输入至 NLP模型进行训练（使用ArcFace作为loss）。
- 通过 NLP模型生成的embeddings ，以及空间近邻来生成POI candidate。
- 为每个candidate rank创建并训练XGBoost二分类模型，获得最终结果。

验证策略

- 一开始，我们把训练数据分成4个Fold，并训练BERT（train:valid = 3:1）。一段时间后，我们发现，如果我们使用所有的数据训练BERT，我们可以得到更好的结果。在这种情况下，我们用泄露的数据训练XGBoost，它可能对测试数据表现得很差。也许Embedding的质量比训练一个无泄漏的XGBoost模型更重要。

创建Embedding

- BERT embeding candidates
    - 将各种表格数据concat成文本，再加上经度和纬度信息，作为bert模型的datasets。
    - Embedding维度为320。
    - 损失函数：ArcMarginProduct
    - 训练 30个epoch (24-48小时)
    - 保存best model以及对应输出的Embedding

生成 Candidates

- 模型融合，Concat多个BERT模型 Embedding，维度 320*4
  - `xlm-roBERTa-large`
  - `sentence-transformers/LaBSE`
  - `sentence-transformers/paraphrase-multilingual-mpnet-base-v2`
  - `remBERT`
- 使用 faiss 对 每个ID的Embedding做余弦相似度，创建50个候选者。
- DBA/QE 加权 以获得更好的Embedding和更好的余弦相似度。
- 候选空间 Spatial candidates
  - 使用sklearn.neighbors.BallTree添加基于lat/lon的candidates。
  - 希望增加被BERT Embedding遗漏的candidates。

训练XGBoost

- 对name、address、categories进行TF-IDF编码，作为XGBoost特征。
- 创建经纬度、距离等衍生变量，作为XGBoost特征。
- 为每个candidate rank创建并训练XGBoost二分类模型。

相关代码

DataSets

```
class FourSquareDataset(Dataset):
    def __init__(self, df, tokenizer, max_length):
        self.fulltext = df['fulltext'].values # 全文
        self.latitudes = df['latitude'].values # 纬度
        self.longitudes = df['longitude'].values # 经度
        self.coord_x = df['coord_x'].values # 经纬度坐标 x
        self.coord_y = df['coord_y'].values # 经纬度坐标 y
        self.coord_z = df['coord_z'].values # 经纬度坐标 z
        self.labels = df['point_of_interest'].values # point_of_interest 标签
        self.tokenizer = tokenizer # 词表
        self.max_length = max_length # 最大长度

    def __len__(self):
        return len(self.fulltext) # 返回数据长度

    def __getitem__(self, index):
        fulltext = self.fulltext[index] # 全文
        latitude = self.latitudes[index] # 纬度
        longitude = self.longitudes[index] # 经度
        label = self.labels[index] # 标签
        coord_x = self.coord_x[index] # 经纬度坐标 x
        coord_y = self.coord_y[index] # 经纬度坐标 y
        coord_z = self.coord_z[index] # 经纬度坐标 z

        inputs = self.tokenizer(
            fulltext, # 全文
            truncation=True, # 截断
```

```
            add_special_tokens=True, # 添加特殊字符
            max_length=self.max_length, # 最大长度
            padding='max_length', # 填充方式
            return_tensors="pt" # 返回张量
        )

        return {
            'ids': inputs['input_ids'][0], # input_ids
            'mask': inputs['attention_mask'][0], # 注意力掩码
            'latitude': torch.tensor(latitude, dtype=torch.float), # 纬度
            'longitude': torch.tensor(longitude, dtype=torch.float), # 经度
            'coord_x': torch.tensor(coord_x), # 经纬度坐标 x
            'coord_y': torch.tensor(coord_y), # 经纬度坐标 y
            'coord_z': torch.tensor(coord_z), # 经纬度坐标 z
            'label': torch.tensor(label, dtype=torch.long) # 标签
        }
```

模型结构

```
class FSMultiModalNet(nn.Module):
    def __init__(self, model_name, fc_dim, num_features=3):
        super(FSMultiModalNet, self).__init__()
        self.config = AutoConfig.from_pretrained(model_name) # 加载预训练模型
        self.bert_model = AutoModel.from_pretrained(model_name,
config=self.config) # 加载预训练模型
        # self.embedding = nn.Linear(self.config.hidden_size + 2,
embedding_size)

        self.fc = nn.Linear(self.bert_model.config.hidden_size + num_features,
fc_dim) # 全连接层 hidden_size + x'y'z'
        self.bn = nn.BatchNorm1d(fc_dim) # BatchNorm1d
        self._init_params() # 初始化参数

        self.margin = ArcMarginProduct(
            fc_dim, # 输入维度
            CFG.n_classes, # 输出维度
            s=CFG.s, # scale
            m=CFG.m, # margin
            easy_margin=CFG.easy_margin, # easy_margin
            ls_eps=CFG.ls_eps # label smoothing epsilon
        )

    def _init_params(self):
        nn.init.xavier_normal_(self.fc.weight) # 初始化全连接层权重
        nn.init.constant_(self.fc.bias, 0) # 初始化全连接层偏置
        nn.init.constant_(self.bn.weight, 1) # 初始化 BatchNorm1d 权重
        nn.init.constant_(self.bn.bias, 0) # 初始化 BatchNorm1d 偏置

    def forward(self, ids, mask, lat, lon, coord_x, coord_y, coord_z, labels):
        feature = self.extract_feature(ids, mask, lat, lon, coord_x, coord_y,
coord_z) # 提取特征
        output = self.margin(feature, labels) # ArcMarginProduct 输出

        return output
```

```
    def extract_feature(self, input_ids, attention_mask, lat, lon, coord_x,
coord_y, coord_z):
        x = self.bert_model(input_ids=input_ids, attention_mask=attention_mask)
# 获取 BERT 特征
        x = torch.sum(x.last_hidden_state * attention_mask.unsqueeze(-1), dim=1)
/ attention_mask.sum(dim=1, keepdims=True) # 将 BERT 特征attention_mask部分求平均

        x = torch.cat([x, coord_x.view(-1, 1), coord_y.view(-1, 1),
coord_z.view(-1, 1)], axis=1) # 将 bert输出 和 x'y'z' 合并

        x = self.fc(x) # 全连接层
        x = self.bn(x) # BatchNorm1d

        return x # 返回特征
```

XGBoost 特征

```
def create_features(df, i, indices):
    '''
    创建特征
    '''
    prev_i = max(i-1, 0) # i-1,最小值为0
    next_i = min(i+1, indices.shape[1] - 1) # i+1,最大值为indices长度

    prev_cand_index = indices[:, prev_i] # 第i-1个相似度的candidate的索引
    next_cand_index = indices[:, next_i] # 第i+1个相似度的candidate的索引
    cand_index = indices[:, i] # 第i个相似度的candidate索引

    lon1 = df["longitude"].to_pandas().to_numpy() # longitude列的值
    lat1 = df["latitude"].to_pandas().to_numpy() # latitude列的值
    lon2 = df["longitude"][cand_index].to_pandas().to_numpy() # 第i个相似度的
candidate顺序的longitude列值
    lat2 = df["latitude"][cand_index].to_pandas().to_numpy() # 第i个相似度的
candidate顺序的latitude列值
    #df["diff_lon"] = lon1 - lon2
    #df["diff_lat"] = lat1 - lat2
    df["diff_lon"] = np.abs(lon1 - lon2) # 经度差绝对值
    df["diff_lat"] = np.abs(lat1 - lat2) # 纬度差绝对值

    df["lonlat_eucdist"] =  (df['diff_lon'] ** 2 + df['diff_lat'] ** 2) ** 0.5 #
经纬度差的平方根
    df["lonlat_manhattan"] = manhattan(lat1, lon1, lat2, lon2) # 曼哈顿距离
    df["lonlat_haversine_dist"] = haversine_np(lon1, lat1, lon2, lat2) # 地球距离

    df["name_cossim"] = V_name.multiply(V_name[cand_index]).sum(axis=1).ravel()
# 名称相似度
    df["full_address_cossim"] =
V_full_address.multiply(V_full_address[cand_index]).sum(axis=1).ravel() # 地址相似
度
    df["cat_cossim"] = V_cat.multiply(V_cat[cand_index]).sum(axis=1).ravel() # 类
别相似度

    df["cand_hit_count_02"] = df["hit_count_02"]
[cand_index].to_pandas().to_numpy()    # 预测概率>0.2的样本的candidate的概率
```

```
    df["cand_hit_count_03"] = df["hit_count_03"]
[cand_index].to_pandas().to_numpy()   # 预测概率>0.3的样本的candidate的概率
    df["cand_hit_count_04"] = df["hit_count_04"]
[cand_index].to_pandas().to_numpy()   # 预测概率>0.4的样本的candidate的概率
    df["cand_hit_count_05"] = df["hit_count_05"]
[cand_index].to_pandas().to_numpy()   # 预测概率>0.5的样本的candidate的概率
    df["cand_hit_count_sum"] = df["hit_count_sum"]
[cand_index].to_pandas().to_numpy() # 所有样本的candidate的概率

    df["hit_count_02_min"] = df[["hit_count_02",
"cand_hit_count_02"]].min(axis=1)  # 样本预测概率和其candidate的概率 的最小值（0.2）
    df["hit_count_03_min"] = df[["hit_count_03",
"cand_hit_count_03"]].min(axis=1) # 样本预测概率和其candidate的概率 的最小值（0.3）
    df["hit_count_04_min"] = df[["hit_count_04",
"cand_hit_count_04"]].min(axis=1) # 样本预测概率和其candidate的概率 的最小值（0.4）
    df["hit_count_05_min"] = df[["hit_count_05",
"cand_hit_count_05"]].min(axis=1) # 样本预测概率和其candidate的概率 的最小值（0.5）
    df["hit_count_sum_min"] = df[["hit_count_sum",
"cand_hit_count_sum"]].min(axis=1) # 样本预测概率和其candidate的概率 的最小值（所有样
本）

    df["hit_count_02_max"] = df[["hit_count_02",
"cand_hit_count_02"]].max(axis=1) # 样本预测概率和其candidate的概率 的最大值（0.2）
    df["hit_count_03_max"] = df[["hit_count_03",
"cand_hit_count_03"]].max(axis=1) # 样本预测概率和其candidate的概率 的最大值（0.3）
    df["hit_count_04_max"] = df[["hit_count_04",
"cand_hit_count_04"]].max(axis=1) # 样本预测概率和其candidate的概率 的最大值（0.4）
    df["hit_count_05_max"] = df[["hit_count_05",
"cand_hit_count_05"]].max(axis=1) # 样本预测概率和其candidate的概率 的最大值（0.5）
    df["hit_count_sum_max"] = df[["hit_count_sum",
"cand_hit_count_sum"]].max(axis=1) # 样本预测概率和其candidate的概率 的最大值（所有样
本）

    cossim = []
    eucdist = []

    eucdist1=[]
    eucdist2=[]
    eucdist3=[]
    eucdist4=[]
    eucdist5=[]

    chunk_size = 100_000 # 每次处理的数据量
    for i in range(0, len(df), chunk_size):  # 以chunk取数据，防止OOM
        cossim.append(cupy.multiply(V_embed_concat[i:i+chunk_size],
V_embed_concat[cand_index[i:i+chunk_size]]).sum(axis=1)) # V_embed_concat 相似度
        eucdist.append(cupy.sqrt(((V_embed_concat[i:i+chunk_size] -
V_embed_concat[cand_index[i:i+chunk_size]]) ** 2).sum(axis=1))) # V_embed_concat
欧式距离 cur - cand_index


 eucdist1.append(cupy.sqrt(((V_embed_concat[prev_cand_index[i:i+chunk_size]] -
V_embed_concat[cand_index[i:i+chunk_size]]) ** 2).sum(axis=1))) # 欧式距离1
prev_cand_index - cand_index
```

```
 eucdist2.append(cupy.sqrt(((V_embed_concat[next_cand_index[i:i+chunk_size]] -
V_embed_concat[cand_index[i:i+chunk_size]]) ** 2).sum(axis=1))) # 欧式距离2
next_cand_index - cand_index
        eucdist3.append(cupy.sqrt(((V_embed_concat[i:i+chunk_size]
   - V_embed_concat[prev_cand_index[i:i+chunk_size]]) ** 2).sum(axis=1))) # 欧式
距离3 cur - prev_cand_index
        eucdist4.append(cupy.sqrt(((V_embed_concat[i:i+chunk_size]
   - V_embed_concat[next_cand_index[i:i+chunk_size]]) ** 2).sum(axis=1))) # 欧式
距离4 cur - next_cand_index

 eucdist5.append(cupy.sqrt(((V_embed_concat[prev_cand_index[i:i+chunk_size]] -
V_embed_concat[next_cand_index[i:i+chunk_size]]) ** 2).sum(axis=1))) # 欧式距离5
prev_cand_index - next_cand_index

    # 将cupy数据转换为pandas数据
    df["embed_cossim"] = cupy.concatenate(cossim)
    df["embed_eucdist"] = cupy.concatenate(eucdist)
    df["embed_eucdist1"] = cupy.concatenate(eucdist1)
    df["embed_eucdist2"] = cupy.concatenate(eucdist2)
    df["embed_eucdist3"] = cupy.concatenate(eucdist3)
    df["embed_eucdist4"] = cupy.concatenate(eucdist4)
    df["embed_eucdist5"] = cupy.concatenate(eucdist5)

    # 欧式距离差
    df["d0_d1"] = df["embed_eucdist"] - df["embed_eucdist1"]
    df["d0_d2"] = df["embed_eucdist"] - df["embed_eucdist2"]
    df["d0_d3"] = df["embed_eucdist"] - df["embed_eucdist3"]
    df["d0_d4"] = df["embed_eucdist"] - df["embed_eucdist4"]
    df["d0_d5"] = df["embed_eucdist"] - df["embed_eucdist5"]

    for col in ["id", "name", "address", "city", "state", "zip", "country",
"url", "phone", "categories", "full_address"]:
        df[f"{col}_edit_dist"] = df[col].str.edit_distance(df[col][cand_index])
# Levenshtein 距离
        # 标准化 Levenshtein 距离
        df[f"norm_{col}_edit_dist"] = df[f"{col}_edit_dist"] / df[col].str.len()
# Levenshtein 距离/长度
        df[f"norm_{col}_edit_dist"] =
df[f"norm_{col}_edit_dist"].replace([np.inf, -np.inf], 0) # 将inf替换为0

    features = [
        "diff_lon",
        "diff_lat",
        "lonlat_eucdist",
        "lonlat_manhattan",
        "lonlat_haversine_dist",
        "name_cossim",
        "full_address_cossim",
        "cat_cossim",
        "embed_cossim",
        "embed_eucdist",

        "embed_eucdist1",
        "embed_eucdist2",
```

```
        "embed_eucdist3",
        "embed_eucdist4",
        "embed_eucdist5",
        "d0_d1",
        "d0_d2",
        "d0_d3",
        "d0_d4",
        "d0_d5",

        "hit_count_02",
        "hit_count_03",
        "hit_count_04",
        "hit_count_05",
        "hit_count_sum",

        "hit_count_02_min",
        "hit_count_03_min",
        "hit_count_04_min",
        "hit_count_05_min",
        "hit_count_sum_min",
        "hit_count_02_max",
        "hit_count_03_max",
        "hit_count_04_max",
        "hit_count_05_max",
        "hit_count_sum_max"
    ]

    for col in ["id", "name", "address", "city", "state", "zip", "country",
"url", "phone", "categories", "full_address"]:
        features.append(f"{col}_edit_dist") # 存下edit_dist类特征
        features.append(f"norm_{col}_edit_dist") # 存下标准化后的edit_dist类特征
    return df, features
```

比赛上分历程

1. 0.922 baseline: bert-base-multilingual-cased
2. 0.928 模型融合加入 xlm-roberta-base
3. 0.931 加入 XGBoost 特征
4. 0.940 使用邻近的candidate 对原数据的null值
5. 0.940+ 四个 bert models 融合

代码、数据集

- 代码
    1. FLM Bert Train.ipynb
    2. FLM XGBoost Train.ipynb
    3. FLM Inference.ipynb
- 数据集
    - 官方数据集
    - train_filled.csv

- datasets.txt

TL;DR

在本次比赛中，参赛者将匹配超过一百万个 Places 条目的数据集的 POI。首先官方数据在city、states、country有很多缺失值，我们使用最邻近的（经度和纬度）5个点为BERT模型填充NaN文本。然后，我们将各种信息concat成文本，再加上经度和纬度信息，输入至 NLP模型进行训练（使用ArcFace作为loss），通过 NLP模型生成的embeddings，以及空间近邻来生成POI candidate。最终我们为每个candidate rank创建并训练XGBoost二分类模型，获得最终结果，拿到奖牌。