

Pet Finder

By 蒋柯越 (3160100572) 孙明达 (3150104841) 蒋辰宇 (3150104664) 刘煜 (3150105039)

前言

我们上交的作业中总共有以下文件：

Project-report.ipynb：

以jupyter notebook呈现的project report。如果有条件的话最好用jupyter notebook打开这个，因为这个的格式是完全没有乱的。

Project-report.html

为了避免ipynb无法打开时候的问题，我们将report输出成了html，并挂在了github上，这个格式也是完整的。

Github: <https://garyball.github.io/PetFinder-project/> (<https://garyball.github.io/PetFinder-project/>)

Project-report.pdf

我们也输出了一个PDF，但是这个文件有一些格式问题（例如代码显示不完整的问题）

code.ipynb

这一部分是没有运行之前的代码，我们建议看kaggle上我们的kernel检查我们的作业：

<https://www.kaggle.com/jky594176/justtrysth/> (<https://www.kaggle.com/jky594176/justtrysth/>)

project-presentation.pptx

课程展示PPT。

以上是我们这次的全部内容。我们的建议是，通过两个网站检查我们的project，这样能保证我们的格式不是乱的hhh

综述

问题背景

当今，越来越多的流浪宠物被抛弃在街头，遭受着疾苦，如果没有人领养，他们的最终归途大多都是被安乐死。PetFinder.my是一个动物爱好组织的动物领养平台，他们与全球各地的动物爱好者、媒体、企业密切合作，以求带给动物们福利。

问题目标

动物的领养率和网站提供的动物元数据，比如描述性文本与照片特征，息息相关。PetFinder.my提供给我们大约19000只宠物的数据库，数据包括表格型数据、图片数据与描述性文本数据。其中大约15000项将被作为训练数据，其余作为测试数据。我们将用这些数据，训练出一个用于预测流浪宠物被收养时间的模型。

数据概览

比赛给出的数据源包括表格数据（.csv格式），处理过的数据（.json格式），和图片数据。

表格数据：

表格数据包括训练数据集train.csv测试数据集test.csv和附加标签信息breed_labels.csv, state_labels.csv, color_labels.csv。训练集共14993个样本，测试集共3972个样本，总共18965个样本。数据集共23维，数据主要分为四类，分别为数值型（Numerical）、类别型（Categorical）object型（文本、string等）和目标数据（target）。

metadata数据：

Metadata数据为官方通过Google API对image（metadata.json）和description(sentiment.json)进行处理后得到的数据，需要提取后使用。metadata.json包括数字量 annots_score,color_score, color_pixelfrac, crop_conf, crop_importance 和文本annots_top_desc共6个特征变量。sentiment.json包括数字量 magnitude_sum, score_sum, magnitude_mean, score_mean, magnitude_var, score_var共6个特征变量。拥有metadata的样本个数为18150，占比0.976，拥有sentiment的样本个数为18307，占比0.965。

图片数据：

在我们的18965个训练数据当中，18330个数据拥有相对应的图片数据，均为.jpg格式, 占比96.5%。单个样本可能不止有一张对应的图片，最高多达10张，图片总共有58370张。图片尺寸也不固定

评价标准：

我们整个输出的结果将会通过quadratic weighted kappa来计算。QWK是卡帕系数的一种，用于进行一致性检验并衡量分类精度。通常其计算基于混淆矩阵（confusion matrix），混淆矩阵就是一个统计预测结果与实际差距的矩阵，一般有如下形式：

混淆矩阵		真实值		
		猫	狗	猪
预测值	猫	10	1	2
	狗	3	15	4
	猪	5	6	20

而我们的kappa系数也是建立在混淆矩阵上的：

$$x = \frac{P_o - P_e}{1 - P_e}.$$

其中， P_o 是每一类正确分类样本数除以总样本数，即样本分类精度；又假设每一类的真实样本个数分别为 a_1, a_2, \dots ，而预测出来的每一类的样本个数分别为 b_1, b_2, \dots ，则

$$P_e = \frac{a_1 \times b_1 + a_2 \times b_2 + \dots}{n \times n}$$

具体代码实现如下：

```
In [ ]: # taken from Ben Hamner's github repository: https://github.com/benhamner/Met
def confusion_matrix(rater_a, rater_b, min_rating=None, max_rating=None):
    """
    返回两个rater产生的混淆矩阵
    """
    assert(len(rater_a) == len(rater_b))
    if min_rating is None:
        min_rating = min(rater_a + rater_b)
    if max_rating is None:
        max_rating = max(rater_a + rater_b)
    num_ratings = int(max_rating - min_rating + 1)
    conf_mat = [[0 for i in range(num_ratings)]
                  for j in range(num_ratings)]
    for a, b in zip(rater_a, rater_b):
        conf_mat[a - min_rating][b - min_rating] += 1
    return conf_mat

def histogram(ratings, min_rating=None, max_rating=None):
    """
    直方图统计
    """
    if min_rating is None:
        min_rating = min(ratings)
    if max_rating is None:
        max_rating = max(ratings)
    num_ratings = int(max_rating - min_rating + 1)
    hist_ratings = [0 for x in range(num_ratings)]
    for r in ratings:
        hist_ratings[r - min_rating] += 1
    return hist_ratings
```

```
In [ ]: def quadratic_weighted_kappa(y, y_pred):  
    # 计算并返回QWK  
    rater_a = y  
    rater_b = y_pred  
    min_rating=None  
    max_rating=None  
    rater_a = np.array(rater_a, dtype=int)  
    rater_b = np.array(rater_b, dtype=int)  
    assert(len(rater_a) == len(rater_b))  
    if min_rating is None:  
        min_rating = min(min(rater_a), min(rater_b))  
    if max_rating is None:  
        max_rating = max(max(rater_a), max(rater_b))  
    conf_mat = confusion_matrix(rater_a, rater_b,  
                                min_rating, max_rating)  
    num_ratings = len(conf_mat)  
    num_scored_items = float(len(rater_a))  
  
    hist_rater_a = histogram(rater_a, min_rating, max_rating)  
    hist_rater_b = histogram(rater_b, min_rating, max_rating)  
  
    numerator = 0.0  
    denominator = 0.0  
  
    for i in range(num_ratings):  
        for j in range(num_ratings):  
            expected_count = (hist_rater_a[i] * hist_rater_b[j]  
                              / num_scored_items)  
            d = pow(i - j, 2.0) / pow(num_ratings - 1, 2.0)  
            numerator += d * conf_mat[i][j] / num_scored_items  
            denominator += d * expected_count / num_scored_items  
  
    return (1.0 - numerator / denominator)
```

工具

软件

- Ipython Notebook

Python Library

- Sklearn
- Numpy, Pandas
- Tensorflow
- LightGBM
- CatBoost
- Glove

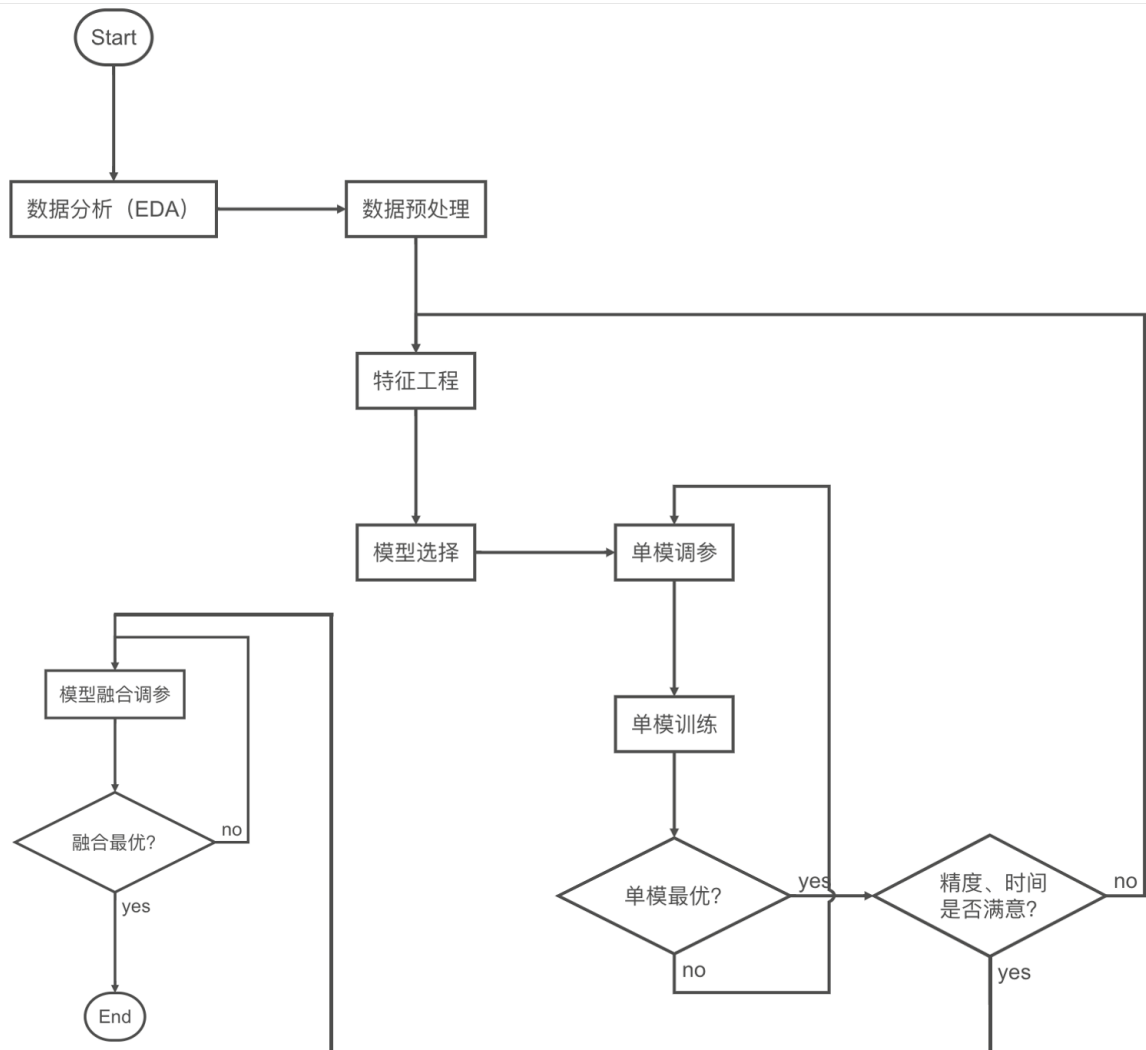
硬件

- Kaggle Notebook

- 4 Cores CPU, 17GB RAM
- 2 cores GPU, 16GB RAM
- 5 GB auto-saved disk space
- 16G temporary Disk
- 9 Hours execution time

思路与规划

因为这次的project是在kaggle中，对于一个初学者，kaggle提供了大量的技术支持以让我们完成我们的任务。而人们在kaggle上做数据竞赛的习惯，大多都是由一些大佬贡献出自己的kernel为其他人作为baseline。而我们要做的工作，大部分都是结合他们的思路，添加上自己的一些东西。整体的思路主要分为数据预处理、特征工程、模型训练三部分，具体如下图：



我们的数据集极其庞大，且数据形式相当繁杂。我们采用的整体策略是“宁滥勿缺”，也就是大量的读入数据，通过各种（甚至可能重复）方式来对数据处理，添加新的特征。等到最后做特征工程的时候，再通过降维、剔除不相关特征等方式去掉多余的特征。

数据挖掘与可视化

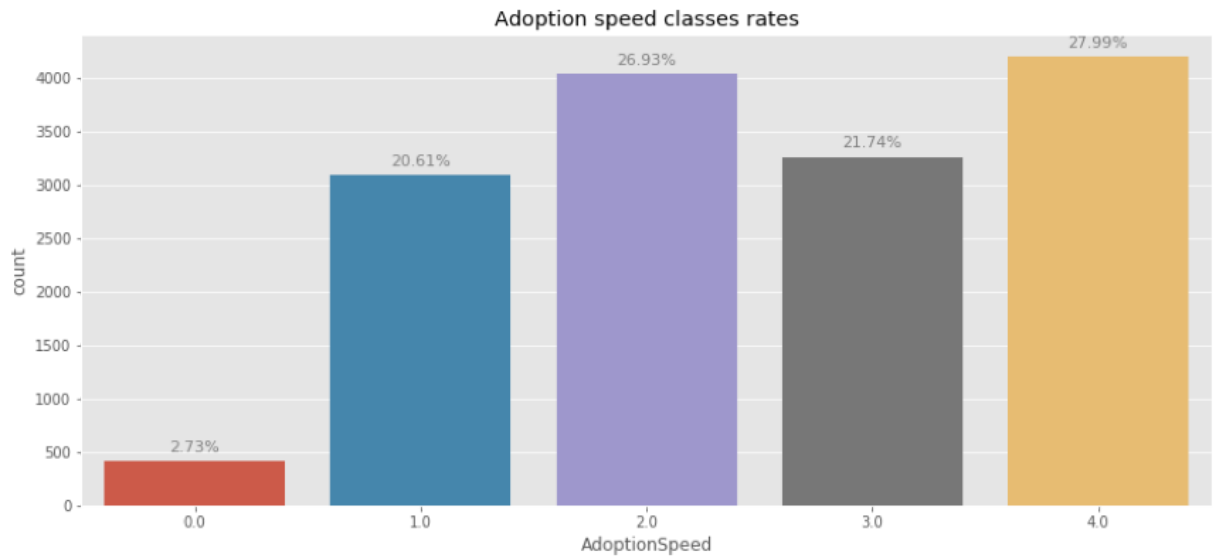
首先对比赛的数据集做初步了解，为后面的数据预处理工作做准备，直观了解特征对目标数据的影响。在开始工作之前，对数据进行一个定性了解的好处是必须的，这可以从另一个角度来验证我们最终得到的结果是否可靠。

目标数据：Adaptation speed也就是领养时间。我们将最终的领养时间氛围。领养时间是一个Categorical数据，分了0-4五类。

- 0 - 宠物被挂出时当天被领养
- 1 - 宠物被挂出后1-7天被领养
- 2 - 宠物被挂出后8-30天被领养

- 3 - 宠物被挂出后31-90天被领养
- 4 - 100天以内未被领养

从下图中可以看出，宠物被领养时间的大致分布。我们在预测结束后，如果发现结果中出现了大量的0，在其他四类中分布明显地不均衡，则可能需要考虑预测精度了。



表格数据： 首先我们观察一下表格类数据，表格类数据主要分为四类，分别为数值型（Numerical）、类别型（Categorical）object型（文本、string等）和目标数据（target）。

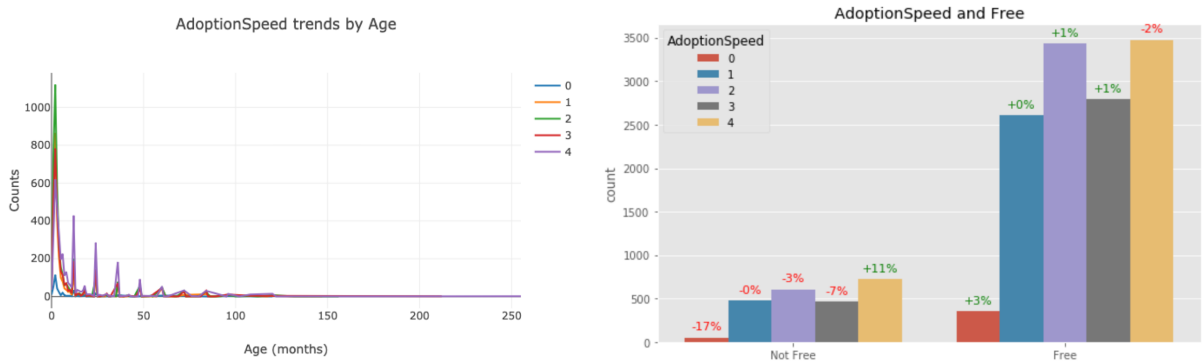
Data Columns	Number	Type
Type, Gender	14993	Categorical
Breed 1-2, Color 1-3	14993	Categorical
FurLength, Vaccinated, Dewormed, Sterilized	14993	Categorical
VideoAmt	14993	Numerical
PhotoAmt	14993	Numerical
Age, MaturitySize,Health	14993	Numerical
Quantity	14993	Numerical
Fee	14993	Numerical
Name	13736	object
RescuerID	14993	object
Description	14981	object
PetID	14993	object
AdoptionSpeed	14993	Target

其中，可以直接送入模型进行训练的数据为数值型与类别型数据。而object类数据，如果是文本则和之后的text data一起处理，如果是其他无意义的的数据（RescuerID, Name等）则经过简单的处理（如求长度、累计出现次数等）后添加为一个新的列特征。当然还有一些用于辨别的数据，如PetID，在训练时直接删除即可。

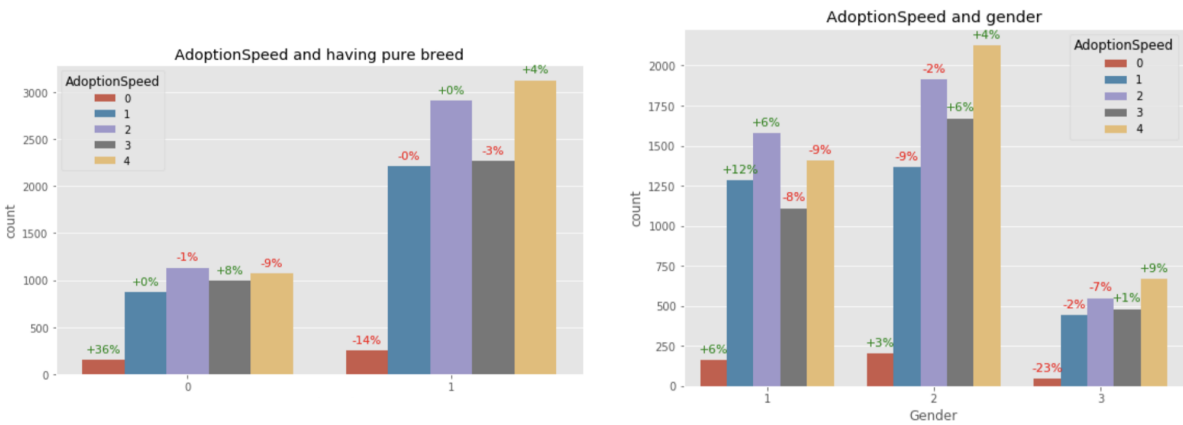
我们再看一看领养时间与这些表格类数据之间的关系。在看这类数据的时候，我们关注的点是领养时间是否因为同一特征的不同类别而产生巨大差异，而非单纯的数量多少。

我们可以发现几个很明显与领养时间息息相关的变量：

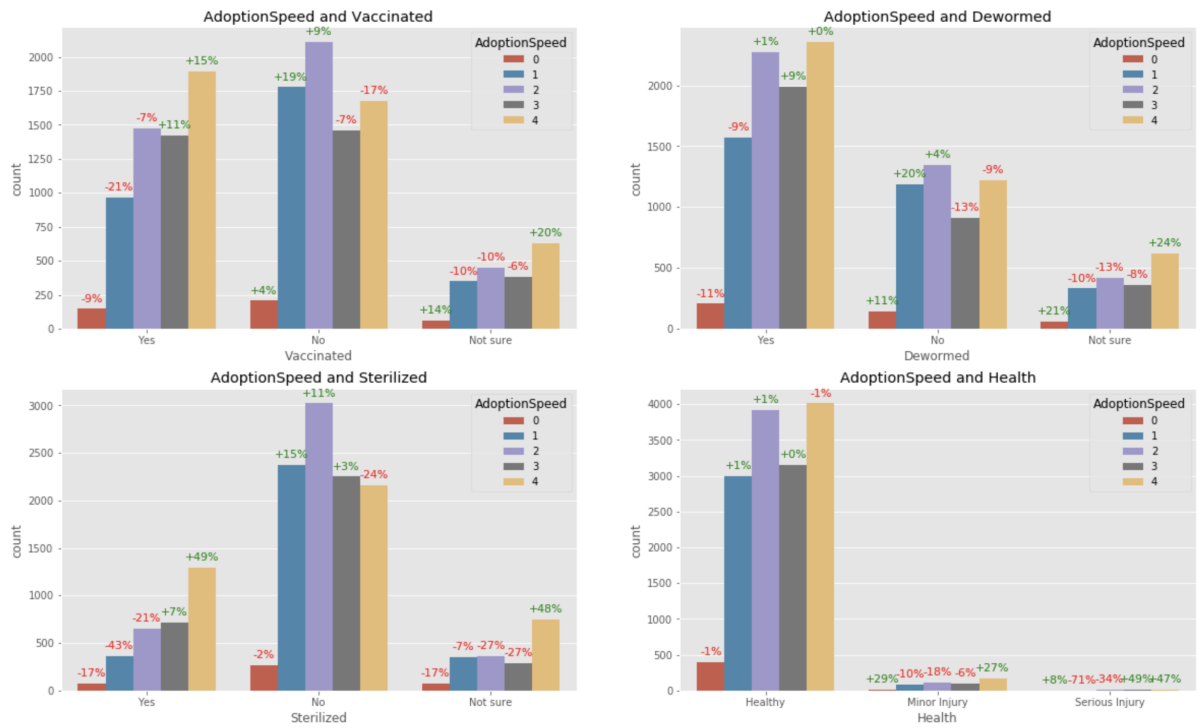
品种（Type）、年龄（Age）



是否纯种 (pure breed) 、性别(Gender)

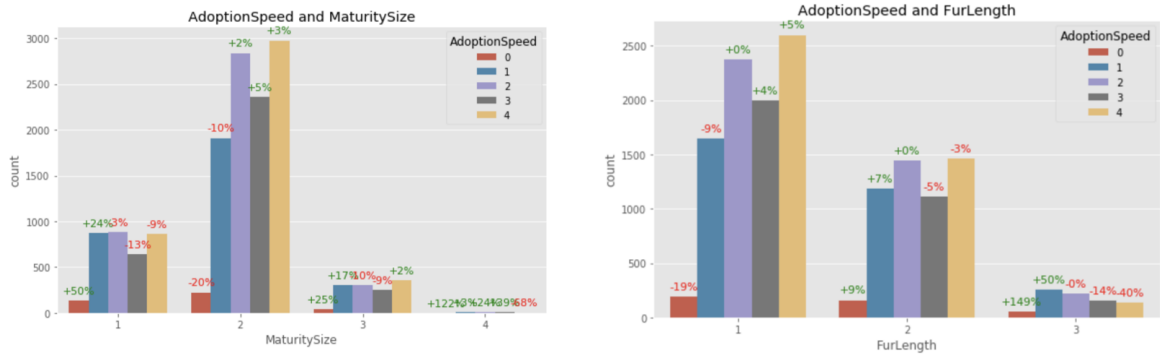


健康情况(Health)



而其他几个变量和领养时间的关系就没有那么密切，然而我们本来认为这应该是影响比较大的因子(当然有一定原因是这类数据很难看出区别。):

成熟后的大小(MaturitySize)、毛发长度(FurLength)



图片数据：

在我们的14993个训练数据当中，14465个数据拥有相对应的图片数据，占比96.5%。同一个宠物可能不止有一张对应的图片，可能多达10张。图片总共有58370张。因为我们对于CNN卷积神经网络并没有特别深入的理解，因此我们在做这步的时候存粹只是按部就班地使用别人训练好的用于提取图片特征的DenseNet。

文本数据：

我们的文本数据来自于两个方面，一是来自表格数据当中的'Description'列，二是来自json文件中取出来的另一部门文本数据。我们的文本处理主要来自这两个方面。还有一部分数据，是官方通过Google NLP API处理过的文本数据，最终形成了Numerical或者Categorical数据。

感谢：<https://www.kaggle.com/artgor/exploration-of-data-step-by-step>
(<https://www.kaggle.com/artgor/exploration-of-data-step-by-step>)

数据处理

因为官方提供的数据是以训练集与测试集分开提供的，因此在接下来的数据处理当中，均以训练集为例。（除了Adoption speed之外，其他数据两集类似。）

表格类数据

表格类数据是最容易处理的一类数据。我们将所有的数据利用Pandas读入到DataFrame中去，

```
In [ ]: train = pd.read_csv('../input/petfinder-adoption-prediction/train/train.csv')
test = pd.read_csv('../input/petfinder-adoption-prediction/test/test.csv')
sample_submission = pd.read_csv('../input/petfinder-adoption-prediction/test/sample_submission.csv')
```

图片类数据

简单来讲，我们将图片通过CNN卷积神经网络进行特征提取，提取出一个256维的特征，融合进入我们从表格类数据提取出来的DataFrame中去。在具体读取文件之前，我们需要对图片进行预处理：


```
In [ ]: def resize_to_square(im):
    # 图片标准化, 将读取的图片信息重新编成256*256*3的RGB图片。
    old_size = im.shape[:2] # old_size is in (height, width) format
    ratio = float(img_size)/max(old_size)
    new_size = tuple([int(x*ratio) for x in old_size])
    im = cv2.resize(im, (new_size[1], new_size[0]))
    delta_w = img_size - new_size[1]
    delta_h = img_size - new_size[0]
    top, bottom = delta_h//2, delta_h-(delta_h//2)
    left, right = delta_w//2, delta_w-(delta_w//2)
    color = [0, 0, 0]
    new_im = cv2.copyMakeBorder(im, top, bottom, left, right, cv2.BORDER_CONSTANT,
    return new_im

def load_image(path, pet_id,i=1):
    # 读取图片的函数。我们将i的默认值设为1, 是为了在仅读取一个petID的一张图片时用
    image = cv2.imread(f'{path}{pet_id}-{i}.jpg')
    new_image = resize_to_square(image)
    new_image = preprocess_input(new_image)
    return new_image
```

一开始, 我们采用ResNet与DenseNet同时进行特征提取。将提取出来的特征进行简单的求均值后作为最终特征。但是这样的做法在当时答辩的时候被指出并不是特别合理。因为ResNet和DenseNet属于两种卷积神经网络, 他们的内部架构并不类似, 因此简单地对两者进行求平均, 反而可能使得最终得到的结果失去应有的特征。

这样问题提出后, 最为合理的方法是将两种网络分别生成特征, 即形成256+256=512D特征, 再merge到DataFrame中去。但是这样使得我们的训练时间爆炸增长(本来总共也就300D的数据, 等于说数据量加倍), 这显然是不合理的。于是, 我们最终决定放弃ResNet, 单纯使用DenseNet进行特征提取。

我们使用的是别人已经训练过的DenseNet模型, 拥有121层的DenseNet-BC-121-32-no-top.h5。构建模型代码如下:

```
In [ ]: from keras.models import Model
from keras.layers import GlobalAveragePooling2D, Input, Lambda, AveragePooling1D
import keras.backend as K
# 设置输入格式
inp = Input((256,256,3))
# 设置框架backbone, 读取densenet
backbone = DenseNet121(input_tensor = inp,
                        weights='../input/densenet/DenseNet-BC-121-32-no-top.h5',
                        include_top = False)
x = backbone.output
x = GlobalAveragePooling2D()(x)
x = Lambda(lambda x: K.expand_dims(x,axis = -1))(x)
x = AveragePooling1D(4)(x)
out = Lambda(lambda x: x[:, :, 0])(x)

m = Model(inp,out)
```

处理图片数据 (以训练集为例) :

```
In [ ]: features = {}
for b in tqdm_notebook(range(n_batches)):
    start = b*batch_size
    end = (b+1)*batch_size
    batch_pets = pet_ids_list[start:end]
    batch_images = np.zeros((len(batch_pets),img_size,img_size,3))
    for i,pet_id in enumerate(batch_pets):
        for j in range(int(train[train.PetID==pet_id]['PhotoAmt'])):
            try:
                batch_images[i] = load_image("../input/petfinder-adoption-pred")
            except:
                pass
    batch_preds = m.predict(batch_images)
    for i,pet_id in enumerate(batch_pets):
        if pet_id in features.keys():
            sum += batch_preds[i]
            count += 1
        else:
            sum = batch_preds[i]
            count = 0
    if count == 0:
        count = 0.1
    features[pet_id] = sum/count
```

我们展示的代码，是对所有图片数据都进行一次处理的代码。我们最终为了减少训练时间只读取了第一张图片特征

这样通过该网络最后提取出来的特征是一个256（16*16）D的特征

文本类数据

对于文本类数据，就需要涉及自然语言处理（NLP）了，我们考虑过的nlp编码方式包括以下几种：

- One-hot Encoding
- TfIdf Encoding
- SVD + word2vec
- GloVe word Embedding

其中，one-hot Encoding和 TfIdf属于Non-Distributed representation（非分布式表示），SVD和 Word2Vec属于Distributed representation（分布式表示）。两者在NLP中的最大区别在于，仅考虑词还是同时考虑词向量的顺序而导致的语义问题。

One-hot Encoding:

One-hot是最简单的编码方式，他的假设是所有的词都是独立的，于是将每个词独占词向量中的一个维度。他所做的就是，将一篇文章出现的所有词都做统计，形成一个很长的词向量，而每一个词都占有一维，从而实现编码。这种编码方式会形成很大维度的稀疏矩阵。

TfIdf:

所谓词频（Term Frequency），非常好理解，就是将向量化之后的文本统计出现频率，其想法就是如果某一个词或短语在一篇文章中出现的频率TF高，并且在其他文章中很少出现，则认为此词或者短语具有很好的类别区分能力，适合用来分类。逆文本频率（Inverse Document Frequency）则是为了修正每一个词在整句话中的重要性的（例如to的词频会很高，但是重要性却很低。），TfIdf有以下公式

$$TfIdf = Tf \times Idf = \frac{\text{某词在文章中出现数}}{\text{文章总词数}} \times \log\left(\frac{\text{语料库文档总数}}{\text{包含该词的文档总数}}\right)$$

简单来说，TFIDF是一种从文本中提取特征的办法。通过对词的出现频率统计提取相关的特征，但是，TFIDF并不能很好地反映词与词之间的前后语境分类。

SVD + word2vec:

SVD是一种基于共现矩阵的NLP统计算法。所谓共现矩阵，指的是通过一个窗口取过滤词向量，再统计一个事先指定大小的窗口内的word共现次数，以word周边的共现词的次数做为当前word的vector。比如我们有语料库：

I like deep learning.

I like NLP.

I enjoy flying.

就可以生成共现矩阵：

counts	I	like	enjoy	deep	learning	NLP	flying
I	0	2	1	0	0	0	0
like	2	0	0	1	0	1	0
enjoy	1	0	0	0	0	0	1
deep	0	1	0	0	1	0	0
learning	0	0	0	1	0	0	0
NLP	0	1	0	0	0	0	0
flying	0	0	1	0	0	0	0

而SVD则是为了处理共现矩阵过于稀疏的问题，对其做奇异值分解得到的矩阵。（奇异值。。。篇幅原因没法解释了，具体见线代书hhh）

GloVe:

GloVe是一种结合了非分布式表示（做了概率统计）和分布式表示（对上下文进行了关联）的语义模型。简单来讲，就是说一个单词和哪个上下文单词在一起的多，那么这个单词与这个上下文单词在一起要比与其他词在一起意义要大。具体步骤如下

取 $word_i$ 的出现次数为 X_i ，定义

$$P_{ij} = P(j|i) = \frac{X_{ij}}{X_i P_{ij}} = P(j|i) = \frac{X_{ij}}{X_i}$$

w_i, w_j 是实数空间下的 $word_i, word_j$ 的词向量， w_k 也是实数空间下的 $word_k$ 的背景词向量。表示在 X_i 的上下文中 $word_j$ 的出现几率，。

于是我们可以找到一种映射方法F计算这种关系，这个方法就是我们训练模型。

$$F(w_i, w_j, w_k) = \frac{P_{ij}}{P_{jk}}$$

最终选择

我们最终选择以下两种处理方式，最后送进RNN模型训练：

- 对于目标文本进行TFIDF（词频与逆文本频率）统计，产生一系列代表特征的特征列。选择Tfidf的原因是，这种处理产生的新特征有着极大的代表性。试想，一个宠物的多个描述文本如果有极其频繁的词出现，且这个词是描述性的，那么很可能这个词就代表着宠物的特征。而one-hot作出的特征提取可信度并不高。在Tfidf中我们不可避免地使用了SVD。
- 对文本进行GloVe预处理+提取特征后，送入RNN训练。GloVe解决了Tfidf对语义序列的忽视。

Tfidf： 通过Tfidf生成特征

```
In [ ]: # 通过TFIDF处理文本特征
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.decomposition import SparsePCA, TruncatedSVD, LatentDirichletAllocation

n_components = 5
text_features = []
# 之前也提到了, 其实TFIDF就是文本特征
# 生成文本特征
for i in X_text.columns:

    print('generating features from: {}'.format(i))
    # 通过SVD进行降维, NMF为非负矩阵分解, 都是预处理
    svd_ = TruncatedSVD(
        n_components=n_components, random_state=1337)
    nmf_ = NMF(
        n_components=n_components, random_state=1337)

    # 分别通过SVD和NMF生成特征
    tfidf_col = TfidfVectorizer().fit_transform(X_text.loc[:, i].values)
    svd_col = svd_.fit_transform(tfidf_col)
    svd_col = pd.DataFrame(svd_col)
    svd_col = svd_col.add_prefix('SVD_{}_'.format(i))

    nmf_col = nmf_.fit_transform(tfidf_col)
    nmf_col = pd.DataFrame(nmf_col)
    nmf_col = nmf_col.add_prefix('NMF_{}_'.format(i))

    text_features.append(svd_col)
    text_features.append(nmf_col)
```

GloVe: 对于GloVe语义模型, 我们直接采用了别人训练好的网络:

```
In [ ]: def load_glove():
    EMBEDDING_FILE = '../input/glove840b300dtxt/glove.840B.300d.txt'

    def get_coefs(word, *arr): return word, np.asarray(arr, dtype='float32')

    embeddings_index = dict(get_coefs(*o.split(" ")) for o in open(EMBEDDING_FILE))
    return embeddings_index

glove_emb = load_glove()

embedding_matrix, nb_words, nb_oov = build_emb_matrix(word_dict, glove_emb)
print(nb_words, nb_oov)
del glove_emb
gc.collect()
```

RNN: RNN属于深度神经网络(DNN)中的一种, 和CNN卷积神经网络共同在NLP打下了一片江山。而我们的project中, 则充分利用了RNN对语言序列建模的优势。因为RNN训练部分代码偏长, 截取其中一部分解释, 具体的请看我们附件中的代码kernel。

首先设定参数 (调参见后文) :

```
In [ ]: train_epochs = 6
loss_fn = torch.nn.MSELoss().cuda()
oof_train_nlp = np.zeros((X_train_drop.shape[0], 32+1))
```

建立模型并创建最终用于融合的OptimizedRounder

```
In [ ]: model = FmNlpModel(hidden_size=48, init_embedding=embedding_matrix, head_num=1,
                           fm_embed_size=10, fm_feat_len=X_train_fm.shape[1], fm_max_len=X_train_fm.shape[1],
                           numerical_dim=X_train_numerical.shape[1],
                           nb_word=nb_words, nb_pos=nb_pos, pos_emb_size=10)

model.cuda()

optimizer = torch.optim.Adam(filter(lambda p: p.requires_grad, model.parameters))
# scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=4, gamma=0.1)
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=4, eta_min=1e-6)
```

分代进行训练。

```
In [ ]: for epoch in range(train_epochs):
        scheduler.step()
        model.train()
        for sentences, poses, lengths, x_fm, x_numerical, labels in training_loader.iter_instances():
            iteration += 1
            sentences = sentences.cuda()
            poses = poses.cuda()
            lengths = lengths.cuda()
            x_fm = x_fm.cuda()
            x_numerical = x_numerical.cuda()
            labels = labels.type(torch.FloatTensor).cuda().view(-1, 1)

            pred,_ = model(sentences, poses, lengths, x_fm, x_numerical)
            loss = loss_fn(pred, labels)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
```

初步测试

当我们读完所有的数据并将其转化为可以被训练的数据后，我们就进行了初步的模型融合。在这一步，我们简单地使用了LightGBM，XGBoost和传统的GBDT模型，如随机森林、极端随机树等。在这时，我们遇到了两个问题

- 随着训练时间的增加，我们超过了官方给的资源限制
- 随着训练模型的堆砌，我们的评分反而变低了

模型	LightGBM	LGB+RF	LGB+XGB+RF
Time	27min	102min	140min
QWK	0.409	0.403	0.388

为了解决这两个问题，我们必须对我们的模型和数据进行优化。对于前一个问题，也就是训练时间的问题，我们通过以下几个方面来解决：

1. 图像处理

我们的图像来源十分庞大，对于同一只宠物，多的图片可以达到将近10张。即使拥有GPU，我们的图片特征提取仍然是一项巨大的工程。我们发现，同一只宠物的不同图片所提取出来的特征有很大的相似性。因此，我们决定尽量减少图片处理，即每一只宠物仅提取一张图片的特征。同时，仅用一个模型（DenseNet）提取图片特征。这样可以大量减少运算量。

2. 数据降维与舍弃

即通过减少训练数据的维度来降低训练时间，降维和舍弃同时也能一定程度上解决过拟合的问题。这段在接下来的特征工程会具体讲解。

3. 模型选择优化

即通过选取训练起来更加高效的模型进行训练。当然一味地提升模型训练时间是没有任何意义的，因为同一模型训练速度的提升也意味着丢掉一部分的准确性；而不同模型对于训练结果的拟合程度也是不同的，具体选模还得具体问题具体分析。我们在之后的选模调参中会继续考虑这个问题。

而对于后一个问题，我们明白我们必须对我们的数据进行一些处理，即特征工程，才能将我们的数据充分利用并尽可能地符合各个模型的训练特征。

特征工程与数据优化

因为特征工程做的操作较多，我们将我们的操作和kernel部分的代码cell号对应。即下图中左侧的编号：

In [2]:

```
plt.rcParams['figure.figsize'] = (12, 9)
```

通过已有数据提取更多信息

1. 添加外部数据集 pet_breed_rating:In[24], state GDP: in[7]

我们将一部分无意义的简单分类数据（如state、breed等），通过了外部数据赋予了意义。这是比赛允许的，也是现实中非常重要的特征工程的一步，根据主观感受宠物的品种和收养地区的经济水平可能会对收养的情况产生影响，因此添加了这两个外部的数据，breed rating数据的添加效果并不明显，导致最终评分略微下降，主要原因是本身数据集中所涵盖的品种十分有限（可能需要模糊匹配？还需要工作）。加入state GDP后对xgboost有一定提升，但也不明显。

2. 提取json文件中的信息 In[14]

提取json中的信息转成dataframe，虽然比较耗费时间（20min），但是这一部分信息对我们的模型提升较大。

3. 对metadata和sentiment中的信息聚合(sum、mean、var) In[20], In[50]-[56]

提取metadata和sentiment中的数据转化为DataFrame并根据PetID进行分组，把多个图片的内容聚合成特征变量

4. 从已有数据中提取特征（作用都不明显） In[41]

RescuerID: 添加RescuerID_Count记录宠物救助者的救助宠物总数信息，添加样本救助者的救助种类数RescuerID_Unique

Name: 添加姓名长度、有无数字和是否为高频常见姓名（如puppy、kitty）的特征

Text_col: 提取长度特征

5. 对于关联性较强的特征进行聚合处理, 提取新的特征。 In[42]

6. 对于中英文文本进行分类, 分别提取特征 In[45]

我们的文本数据中包括中文和英文 (这是马来西亚的网站。。。), 因此这两部分必须分开处理。中文使用jieba,英文使用

7. 提取文本特征 In[46]-[47]

先使用TFIDF提取特征, 然后使用svd降维, 共提取16维特征.

8. 对categorical columns 进行编码In[40]

分别用xgboost测试了one-hot-code、impact-encoding和label-encoding其中impact-encoding效果最好

9. 图片特征提取pretrained network In[62]

加载预先训练好的模型提取特征, SVD降维到32维

10. 提取图片尺寸特征并聚合处理 In[30]

求图片width,height,size的平均、和、方差

11. 删除不必要列 In[108]等

删去PetID、Name、RescureID等不需要送入模型训练的特征

数据缺失值补全

直接通过fillna函数即可完成

```
In [ ]: X_train_drop = X_train.fillna(-1)
        X_test_drop = X_test.fillna(-1)
```

列合并

列合并是指将同一属性的不同列合并到一起, 以期达到数据降维的效果。我们这里以对breed_full的操作作为例子, 我们对color、sentiment等也做了一样的操作。

首先合并两列breed:

```
In [ ]: X_temp['Breed_full'] = X_temp['Breed1'].astype(str)+'_'+X_temp['Breed2'].astype(str)
```

然后通过factorize函数编码:

```
In [ ]: X_temp['Breed_full'],_ = pd.factorize(X_temp['Breed_full'])
```

添加要删除的列

```
In [ ]: to_drop_columns.extend(['Breed1', 'Breed2', 'Color1', 'Color2', 'Color3'])
```

Feature Importance

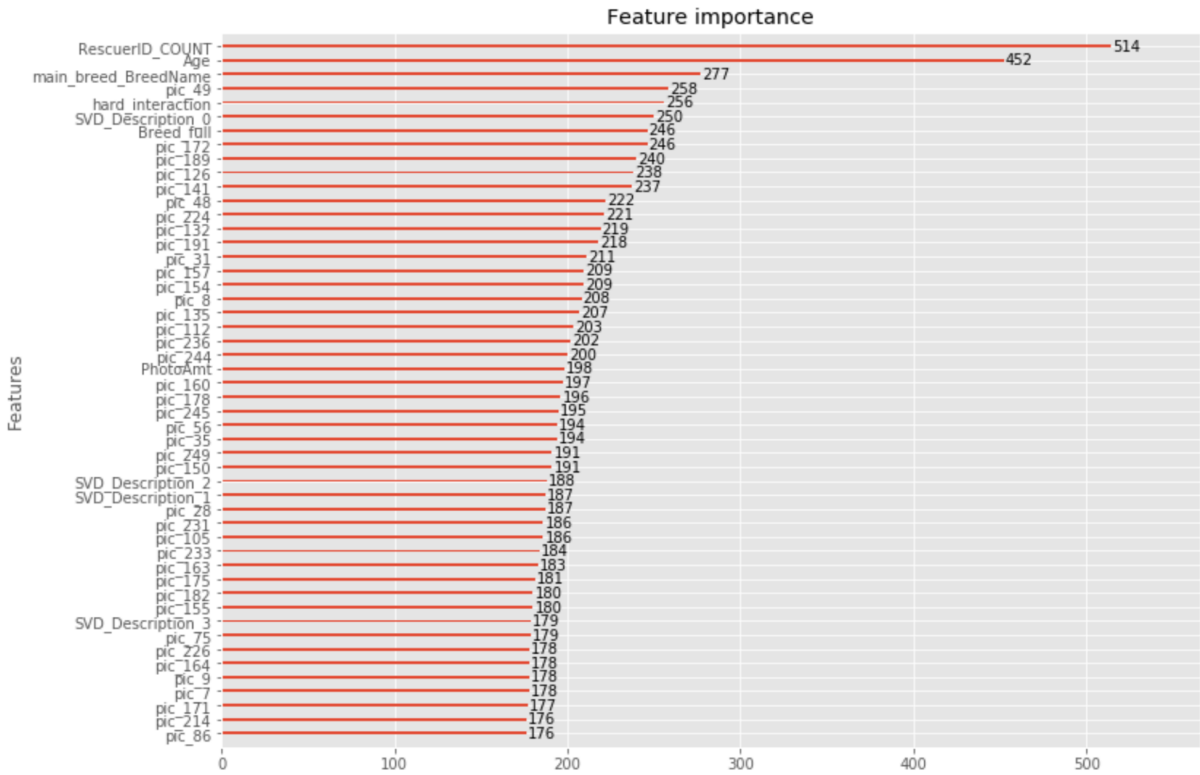
通过预先进行一次LGB训练，可以得到LGB对于不同列特征之间的排序。这样可以有效降低数据维度。

```
In [ ]: lgbm_params = {
    'task': 'train',
    'boosting_type': 'gbdt',
    'objective': 'regression',
    'metric': 'rmse',
    'nrounds': 50000,
    'early_stop_rounds': 2000,
    # trainable params
    'max_depth': 9,
    'num_leaves': 70,
    'feature_fraction': 0.6,
    'bagging_fraction': 0.9,
    'bagging_freq': 8,
    'learning_rate': 0.019,
    'verbose': 0
}
lgb_wrapper = LGBWrapper(lgbm_params)

with timer('Training LightGBM'):
    lgb_oof_train, lgb_oof_test = get_oof(lgb_wrapper, X_train_non_null.drop([
lgb_wrapper.importance()
```

```
In [ ]: importance = lgb_wrapper.model.feature_importance()
names = lgb_wrapper.model.feature_name()
feature_importance = pd.DataFrame({'feature_name': names, 'importance': importance})
```

由此我们可以得到排名最高的一部分特征



在这之后，我们通过删除相关性并不是特别高的特征，进行降维

可能会有提升的工作

除此之外，我们还有一些其他可能提升我们性能的工作：

- 1. 对于已有信息的进一步挖掘，如救助者给被救助宠物拍摄照片的数量、救助者救助宠物的健康状况。
- 2. 对于不同模型进行特征选取

模型选择与调参

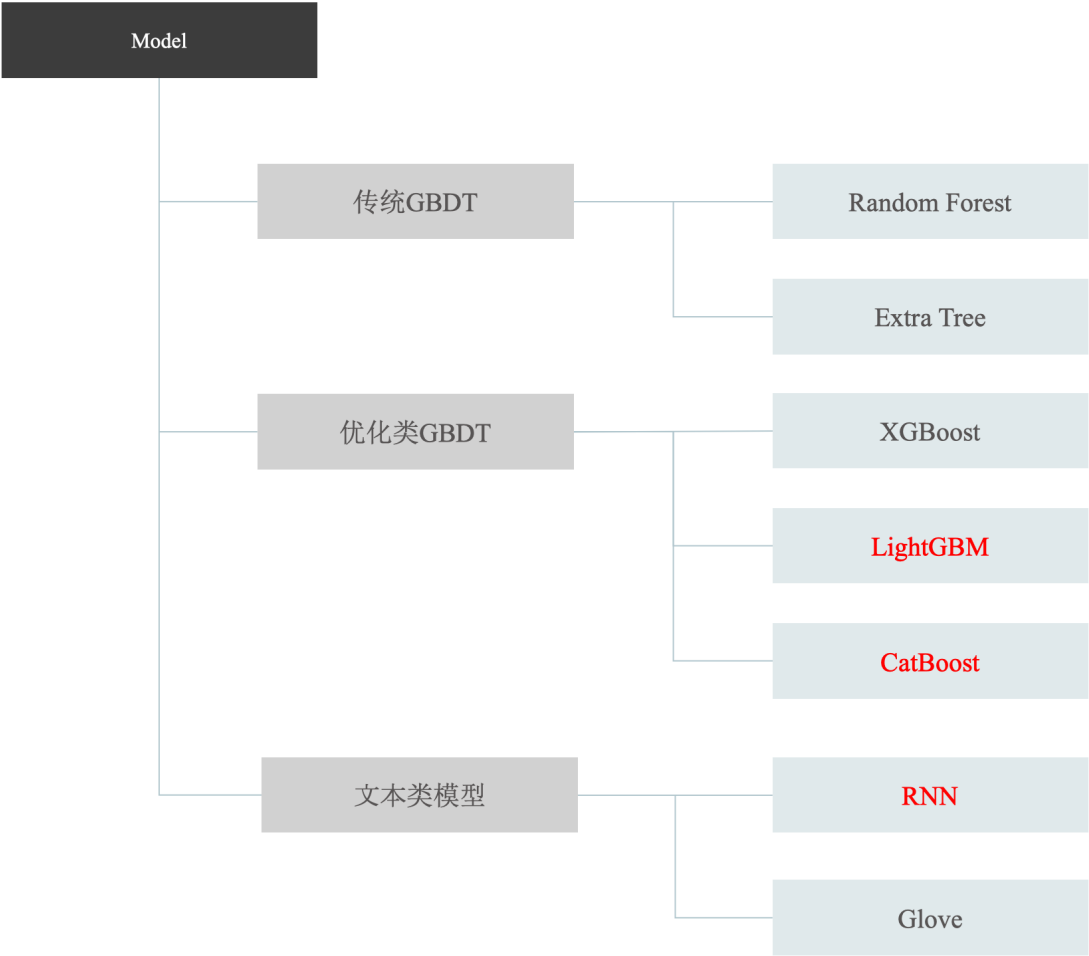
模型选择

我们尝试过的模型，相对应的QWK（数据最终优化），以及训练使用的时间如下：

模型	QWK	训练时间
Extra Tree	0.412	32min
Random Forest	0.407	75min
XGB(GBDT)	0.459	21min
LGB(GBDT)	0.465	11min
CatBoost	0.447	6min
RNN（文字类模型）	0.449	23min
NN	0.432	32min

注：每次由于参数不同训练的时间并不一致，我们在做训练的时候也是几个模型一起stack起来，时间也只是对几次训练进行横向比较后求差估计。且这里的时间是经过特征工程后的数据训练时间，会比之前提到的时间略短。综上所述，时间仅能作为参考。

对于我们的模型，我们做一个分类。



分类的原因是因为我们在做模型融合的过程中，需要考虑到不同模型的适用性。例如，RNN和Glove模型对于文字性特征的处理特别友善，Extra Tree和Random Forest是传统的GBDT模型，XGB、LGB和CatBoost都是基于GBDT的算法，而CatBoost较LGB、XGB又有对Categorical数据的训练优势。需要注意的是，CatBoost并不是一个适合数值类训练的模型，他最适合的是拥有大量categorical数据的分类模型。而这个正好适用于我们现在的问题。

而在模型融合的过程中，我们需要尽量选择模型之间相关性较低的模型。这样融合出来的模型的可靠性才能更强。例如，同属于GBDT算法的LGB、XGB、RF等的相关性极高，选取其中最为有效的一到两种就可以。这也是在很多的比赛中，人们还是会用较为传统的NN神经网络模型的原因。

我们最终选择的模型是`LightGBM`，`CatBoost`和`RNN`模型。我们放弃了两个传统的GBDT模型，因为`RF`和`ET`的准确度相较于其他几个模型来讲并不是特别高，且训练的时间是在是过长。而XGB虽然精度比较高，但是训练时间随着数据维度的增长爆炸提升，和LGB，CatBoost相比时间实在太长，因此不选。几类处理文本的模型（NN和RNN）中我们选择了一个性能相对较好的RNN。（图中的Glove其实是NN，最后检查的时候发现的。。来不及改了）

调参

调参可以说是一个繁琐又无聊的过程，而且被大家戏称为炼丹。调参的方式包括手动调参与机器搜索，因为机器搜索的计算量实在是太大，所以很多数据处理的朋友一般都会先手动调整参数（碰运气），将最优参数的搜索范围缩小之后，再使用机器搜索。这也是为什么调参总被人戏称炼丹的原因了。我们用的就是先手动缩小范围后机器搜索的办法。

在调参之前，我们需要知道的是我们调参的根据。调参是为了获得更高的准确度（在本题中为qwk），更低的均方误差（rmse），以及更快的训练速度。标准定下来之后，我们就可以在每次参数调整的时候根据目标函数来判断参数好坏了

1. LGB Model

LGB的参数具有以下形式（同时这也是我们的最终参数）：

```
In [ ]: params = {'application': 'regression',
                  'boosting': 'gbdt',
                  'metric': 'rmse',
                  'verbosity': -1,
                  # 可调参数
                  'num_leaves': 37,
                  'max_depth': 6,
                  'learning_rate': 0.01,
                  'subsample': 0.85,
                  'feature_fraction': 0.7,
                  'lambda_l1': 0.01,
                  'min_data_in_leaf': 200
                  }
```

在可调参数当中，对结果影响较大的参数主要是max_depth, num_leaves和min_data_in_leaf

其中，一般把max_depth与num_leaves一起调。从最开始，一般会将num_leaves设为 $2^{\text{max_depth}}$ 。但是这样做很可能会过拟合，于是我们把num_leaves逐渐调小，找到一个较优的值后即可。

在调参过程当中，更快的速度和更高的准确度之间是矛盾的。想要更高的准确度，可以调低learning rate, 调高num_leaves（但要防止过拟合）。为了更快的速度，可以调整feature_fraction和subsample，但会导致精度不够。

最后，过拟合可以通过减小max_bin, num_leaves来解决，也可以调整min_data_in_leaf或正则化参数lambda_l1来解决。

2. CatBoost

CatBoost的调参相较于LGB来说比较容易，仅有如下参数需要调整，或许是CatBoost开发团队对于自己训练的速度有着自信，这些全部都是影响训练精度的参数：

```
In [ ]: params = {'depth': [4, 7, 10],
                  'learning_rate': [0.03, 0.1, 0.15],
                  'l2_leaf_reg': [1, 4, 9]}
```

因为参数比较少，而且CatBoost的训练速度也比较快，因此直接使用机器搜索，得到最优结果。最终调整的参数为

```
In [ ]: 'depth': 6,
        'learning_rate': 0.01,
        'l2_leaf_Reg': 2
```

3. RNN

RNN拥有两类参数：

1. 优化类参数：学习率(learning rates), 封装量(mini batch), 训练代数(epochs)

2. 模型类参数：隐含层数，模型结构。

对于RNN的调参，我们并没有大量地更改别人已经调好的参数。原因有两个，一是因为RNN的可调参数很少，二是因为我们使用RNN的主要目的是进行文本处理，而文本处理的参数大同小异，因此没有必要花大量时间重新调整。

调参工具

之前也已经提到，参数调整过程主要分成人工选参和机器搜索两步。人工选参只要满足以上要求就可以，接下来是机器搜索。

机器搜索主要分为两种，GridSearchCV或者hyperopt。

GridSearchCV就是人工给予一连串的数据，通过机器将他们一个个组合遍历，最后得到一个结果。这个过程运算量非常庞大。在我们的project中，仅LGB的搜索就超过了4个小时。而更庞大的数据搜索就超过了我们云端允许的最长运算时间，直接被自动截断。因此我们最后只得减小运算量，得到一些较为宽泛的结果。代码如下：

```
In [ ]: from sklearn.metrics import roc_auc_score
        from sklearn.model_selection import train_test_split
        from sklearn.model_selection import GridSearchCV

        features.extend(['AdoptionSpeed'])
        train_data = X_train_drop[features] # 读取数据
        y = train_data.pop('AdoptionSpeed').values # 用pop方式将训练数据中的标签值y取出来

        col = train_data.columns
        x = train_data[col].values # 剩下的列作为训练数据
        train_x, valid_x, train_y, valid_y = train_test_split(x, y, test_size=0.2, random_state=42)
        train = lgb.Dataset(train_x, train_y)
        valid = lgb.Dataset(valid_x, valid_y, reference=train)

        watchlist = [valid]

        oof_train_lgb = np.zeros((X_train_drop.shape[0]))
        oof_test_lgb = []
        qwks = []
        rmses = []

        params = {'application': 'regression',
                  'boosting': 'gbdt',
                  'metric': 'rmse',
                  'num_leaves': 70,
                  'max_depth': 9,
                  'learning_rate': 0.01,
                  'subsample': 0.85,
                  'feature_fraction': 0.7,
                  'lambda_l1': 0.01,
                  'verbosity': -1,
                  }
```

```

In [ ]: parameters = {
    'max_depth': [15, 20, 25, 30, 35],
    'num_leaves': [20, 30, 40, 50, 60],
    'learning_rate': [0.01, 0.02, 0.05, 0.1, 0.15],
    'feature_fraction': [0.6, 0.7, 0.8, 0.9, 0.95],
    'bagging_fraction': [0.6, 0.7, 0.8, 0.9, 0.95],
    'bagging_freq': [2, 4, 5, 6, 8],
    'lambda_l1': [0, 0.1, 0.4, 0.5, 0.6],
    'lambda_l2': [0, 10, 15, 35, 40],
    'cat_smooth': [1, 10, 15, 20, 35]
}

gbm = lgb.train(params,
                 train_set=train,
                 num_boost_round=num_rounds,
                 valid_sets=watchlist,
                 verbose_eval=500,
                 early_stopping_rounds=100,
                 )

gsearch = GridSearchCV(gbm, param_grid=parameters, scoring='neg_mean_squared_
gsearch.fit(train_x, train_y)

print("Best score: %0.3f" % gsearch.best_score_)
print("Best parameters set:")
best_parameters = gsearch.best_estimator_.get_params()
for param_name in sorted(parameters.keys()):
    print("\t%s: %r" % (param_name, best_parameters[param_name]))

```

Hyperopt相对于GridSearchCV较为智能，相对于直接遍历得到结果，贝叶斯优化会通过前一次对于目标函数的评估结果建立概率模型，以求得到最小化目标函数。代码如下

```
In [ ]: from hyperopt import fmin, tpe, hp, partial
from sklearn.model_selection import train_test_split

features.extend(['AdoptionSpeed'])
train_data = X_train_drop[features] # 读取数据
y = train_data.pop('AdoptionSpeed').values # 用pop方式将训练数据中的标签值y取出来

col = train_data.columns
x = train_data[col].values # 剩下的列作为训练数据
train_x, valid_x, train_y, valid_y = train_test_split(x, y, test_size=0.2, ra
train = lgb.Dataset(train_x, train_y)
valid = lgb.Dataset(valid_x, valid_y, reference=train)
watchlist = [valid]

oof_train_lgb = np.zeros((X_train_drop.shape[0]))
oof_test_lgb = []
qwks = []
rmsees = []
# 自定义hyperopt的参数空间
space = {"max_depth": hp.randint("max_depth", 15),
        'learning_rate': hp.uniform('learning_rate', 1e-3, 5e-1),
        "bagging_fraction": hp.randint("bagging_fraction", 5),
        "num_leaves": hp.randint("num_leaves", 6),
        }

def argsDict_tranform(argsDict, isPrint=False):
    argsDict["max_depth"] = argsDict["max_depth"] + 5
    argsDict["learning_rate"] = argsDict["learning_rate"] * 0.02 + 0.05
    argsDict["bagging_fraction"] = argsDict["bagging_fraction"] * 0.1 + 0.5
    argsDict["num_leaves"] = argsDict["num_leaves"] * 3 + 10
    if isPrint:
        print(argsDict)
    else:
        pass

    return argsDict
```

整体来说，调参的好坏只能在一定程度上提升模型。当参数已经到一定小的范围之后，调整参数对于整体模型的提升不是很大（举qwkw作为例子，大概在 10^{-3} 数量级以下）。相较于模型和数据特征来讲，参数并不是对结果产生决定性作用的东西，但却是为了展示出效果必不可少的一个环节。

不过，调参真的是一个比较玄学的东西，我们最后从大约200名到74名的过程中，完全是靠的调了几个参数，而且这些参数有一些是有依据的，有一些纯粹是变着看看会不会出现什么效果，最终的效果倒是出乎人意外

下图是我们最后的几次调参所得到的结果。

Submission and Description	Private Score	Public Score	Use for Final Score
BaselineModeling (version 27/27) a minute ago by Keyue Jiang From "BaselineModeling" Script	0.42924	0.00000	<input type="checkbox"/>
BaselineModeling (version 26/27) 17 hours ago by Keyue Jiang From "BaselineModeling" Script	0.42367	0.00000	<input type="checkbox"/>
BaselineModeling (version 25/27) 18 hours ago by Keyue Jiang From "BaselineModeling" Script	0.42636	0.00000	<input type="checkbox"/>
BaselineModeling (version 24/27) a day ago by Keyue Jiang From "BaselineModeling" Script	0.42547	0.00000	<input type="checkbox"/>

模型融合

在将所有的模型参数都调优之后，就可以进行模型融合了。我们采用岭回归进行融合。

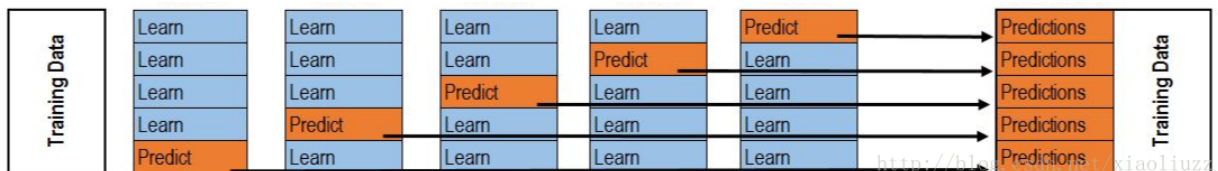
我们的模型融合方式是stacking, stacking是一种分层模型集成框架，我们采用了简单两层stacking。简单来说，第一层由多个基模型组成，第二层模型以第一层的输出结果作为训练集。

例如，我们先通过K-Cross-Validation将数据集分成5份，对模型1，将5份中fold1作为测试集，fold 2-4作为训练集，保存第一份测试集的输出结果；再以fold2作为测试集，其余作为训练集，同样保存结果，以此类推，得到模型1的所有5 folds的预测结果。再对其他模型作用样的操作，得到每个模型的预测值，作为元特征，输入到第二层之中进行预测，KCV代码如下

```
In [ ]: from sklearn.model_selection import StratifiedKFold, GroupKFold

# 通过kcv, 将训练集分为5份, 分别进行训练。
n_splits = 5
# kfold = GroupKFold(n_splits=n_splits)
split_index = []
# for train_idx, valid_idx in kfold.split(train, train['AdoptionSpeed'], train['AdoptionSpeed']):
#     split_index.append((train_idx, valid_idx))

kfold = StratifiedKFold(n_splits=n_splits, random_state=1991)
for train_idx, valid_idx in kfold.split(X_train_drop, X_train_drop['AdoptionSpeed']):
    split_index.append((train_idx, valid_idx))
```



然后将分割后的数据集分别送入各模型中训练(以LGB为例，Catboost类似，可以在我们的代码里面看到)：

```

In [ ]: for n_fold, (train_idx, valid_idx) in enumerate(split_index):
        since = time.time()
        X_tr = X_train_drop.iloc[train_idx]
        X_val = X_train_drop.iloc[valid_idx]

        y_tr = X_tr['AdoptionSpeed'].values
        y_val = X_val['AdoptionSpeed'].values

        d_train = lgb.Dataset(X_tr[features], label=y_tr,
#                               categorical_feature=['Breed1', 'Color1', 'Breed2', 'S
                               ])
        d_valid = lgb.Dataset(X_val[features], label=y_val, reference=d_train)
        watchlist = [d_valid]

        print('training LGB:')
        lgb_model = lgb.train(params,
                               train_set=d_train,
                               num_boost_round=num_rounds,
                               valid_sets=watchlist,
                               verbose_eval=500,
                               early_stopping_rounds=100,
                               )

        val_pred = lgb_model.predict(X_val[features])
        test_pred = lgb_model.predict(X_test_drop[features])
        train_pred = lgb_model.predict(X_tr[features])

```

第二层的模型有很多种选择，可以采用简单的回归方式，如岭回归、线性回归；也可以再套一层模型，如XGB，RF等等。这个根据具体需求来定。

在我们的模型中，我们采用了LGB+CatBoost+RNN的组合岭回归方式。选择的原因前面已经提过了。模型融合代码如下


```
In [ ]: from sklearn.linear_model import Ridge

# 将输入参数融合
Y_train = X_train_drop.iloc[0:len(train)][ 'AdoptionSpeed' ].values
X_train_stacking = np.vstack([oof_train_lgb,
                              oof_train_cat,
                              oof_train_nlp[:, -1]
                              ]).T
X_test_stacking = np.vstack([np.mean(oof_test_lgb, axis=0),
                              np.mean(oof_test_cat, axis=0),
                              oof_test_nlp[:, -1]
                              ]).T

stacking_train = np.zeros((X_train_drop.shape[0]))
stacking_test = []
rmsees, qwks = [], []

for n_fold, (train_idx, valid_idx) in enumerate(split_index):

    X_tr = X_train_stacking[train_idx]
    X_val = X_train_stacking[valid_idx]

    y_tr = X_train_drop.iloc[train_idx][ 'AdoptionSpeed' ].values
    y_val = X_train_drop.iloc[valid_idx][ 'AdoptionSpeed' ].values

    since = time.time()

    # 训练岭回归
    print('training Ridge:')
    model = Ridge(alpha=1)
    model.fit(X_tr, y_tr)
    print(model.coef_)

    val_pred = model.predict(X_val)
    test_pred = model.predict(X_test_stacking)

    stacking_train[valid_idx] = val_pred
    stacking_test.append(test_pred)
    loss = rmse(Y_train[valid_idx], val_pred)
    hist = histogram(y_tr.astype(int),
                    int(np.min(X_train[ 'AdoptionSpeed' ])),
                    int(np.max(X_train[ 'AdoptionSpeed' ])))
    tr_cdf = get_cdf(hist)
```

竞赛流程与结果展示

所用数据集

1. 纯表格类数据
2. 表格类数据+图片特征
3. 表格类数据+图片特征+文本特征
4. 对3进行特征工程
5. 表格类数据+图片数据+文本特征+文本处理
6. 对5进行降维、特征剔除

数据集	选用模型	最终得分
1	XGB	0.363
2	XGB	0.367



数据集	选用模型	最终得分
3	XGB+LGB	0.401
3	XGB+LGB+RF	0.388
4	XGB+LGB	0.412
4	XGB+RF+ET	0.402
5	XGB+NN	0.417
5	LGB+RNN	0.421
6	XGB+LGB+RNN	0.419
6	LGB+CatBoost+RNN	0.424
6+参数调整	CatBoost+LGB+RNN	0.429

最终，再经过将近40次代码提交之后，我们的模型在kaggle上跑出了qwk = 0.429的分数，在1800组的参赛选手当中位列74位（前5%）。虽然这是在借鉴了很多成功案例之后得到的结果，但也令我们相当满意了。作为第一次参加这一类型的数据竞赛，能够得到这个分数，已经是出乎意料了。截图留念一下

Submission
 ✓ **Ran successfully**
 Submitted by Keyue Jiang 2 days ago

Private Score
 0.42924

在比赛中的排名：

73	▲1638	vicohub-2	  +3	0.42929
74	▲1698	PetFinder Accelerator		0.42915

遇到的问题

运算资源限制问题

正如我们之前提到过的，由于我们的比赛是一个Kernel Based的限制资源的比赛，kaggle提供给每一个参赛者一个资源相同的kernel（资源信息见综述模块）。而kaggle仅将运算时间低于2小时的竞赛结果计入比赛。我们的kernel在添加随机森林、XGB只是一度遭遇超时问题。对于该问题

Tensorflow占用GPU问题

在我们打算进行NN处理的时候，出现了如图所示的错误。

这个错误有些让人迷惑，因为我们拥有将近16GB的GPU，而显示只占用了288MB，却无法继续申请所需的空。

经过查询之后，我们知道了TensorFlow一个非常令人窒息的特征。如果TF不提前限制空间分配，那么他会把所有的GPU都申请掉。于是，我们的做法就是在最初限制tensorflow申请GPU，只允许申请总GPU的2/3。实现代码如下：

```
In [ ]: def get_session(gpu_fraction=0.6):

    num_threads = os.environ.get('OMP_NUM_THREADS')
    gpu_options = tf.GPUOptions(per_process_gpu_memory_fraction=gpu_fraction)

    if num_threads:
        return tf.Session(config=tf.ConfigProto(
            gpu_options=gpu_options, intra_op_parallelism_threads=num_threads)
    else:
        return tf.Session(config=tf.ConfigProto(gpu_options=gpu_options))

KTF.set_session(get_session())
```

参数选择过拟合

参数选择过拟合是我们在数据处理过程中必然出现的一个问题。过拟合主要体现在对于训练集的拟合准确率非常高，但是对于测试集却不能有非常好的效果。在我们的任务中，主要体现在拥有较高评分的 QWK，但是同时也有较高的均方根误差（RMSE）。如下是在进行LGB时候出现的过拟合问题。

```
overall rmse: 1.02371
mean rmse = 1.0236992766154143 rmse std = 0.00544039831740533
mean QWK = 0.46556667675080554 std QWK = 0.006523283394947383
```

```
mean QWK = 0.4548755204637934 std QWK = 0.006858336511462284
```

对于过拟合的问题，我们的解决方案主要落在调参当中。以LGB为例，其中的subsample, feature_fraction, lambda等都是用于防止过拟合的

```
In [ ]: params = {'application': 'regression',
                  'boosting': 'gbdt',
                  'metric': 'rmse',
                  'num_leaves': 70,
                  'max_depth': 9,
                  'learning_rate': 0.01,
                  'subsample': 0.85,
                  'feature_fraction': 0.7,
                  'lambda_l1': 0.01,
                  'verbosity': -1,
                  }
```

而在训练集中有着较小qwk（如我们在题目中所用的CatBoost）的模型，却能在最终的测试集结果当中取得较好的结果。

```
rmse= 1.0486693391289792 QWK_2 = 0.4402897106377981 elapsed time: 101.4819688796997
```

小组分工：

姓名	学号	工作
蒋柯越	3160100572	模型调参融合 + 报告撰写/展示
孙明达	3150104841	数据前期处理 + 提供资料
蒋辰宇	3150104664	文本数据处理 + 提供资料

姓名	学号	工作
刘煜	3150105039	图片数据处理

后记（废话）

这一部分就让我来回忆一下当初的坎坷辛路历程把。感觉做了这么一个大数据的作业还是有很多想说的。

其实一开始拿到这个大作业题目的时候是一脸懵逼的，因为当初根本没有任何的理论基础，也不知道到底该从何下手。这种恐慌一直持续了很久，直到研究了kaggle上的代码之后发现数据竞赛其实有着他的套路。一开始选这门课，纯粹是因为对Data Science有点兴趣，而且自以为懂点皮毛，结果发现自己完全是小白。

然而一开始实在是不知道从何下手，于是就开始各种找大佬问，找老师问（当时甚至去淘宝想要套那些代做的人的思路hhh不过他们也真是有职业道德不给3000块就不开口）。但是当初并没有得到特别满意的答案，可能因为这一块做的人不是特别多，而且大家都是按部就班地去套用一些模型，以求得到结果。于是我们的开头就特别坎坷，准确地说直到第一次展示的时候，我们还完全处于一知半解的状态，也基本没有做很多准备工作就上去讲了。（心真的慌）

真正开始有进展是在大概第六周的时候，那时候这个竞赛已经基本上结束了，kaggle上那些大佬高分得主也开始逐渐分享自己的代码给大家看。于是就开始钻进去研究那些代码，发现其实大家用的工具都大同小异，解题到最后也无非就是套用几个常见模型（XGB，LGB）什么的。那时候，终于觉得自己也是能做这个题目的了。直到这个时候，我们才逐渐了解到了所谓的数据分析应该做些什么，所谓的特征工程，所谓的数据分析到底是什么，也开始接触到了各种各样的工具和模型。

于是我们组就开始慢悠悠地把整个工作做起来了。到了第六周末大家才把锅分下去，那时候给组员定的ddl是第七周，但是在第七周的时候突然听说第八周要做展示，心情又开始复杂了。诶，说起来这个给组员分工的事情还是挺气的。其实我们这整个project基本都是蒋柯越、孙明达、蒋辰宇三个人在做，还有一位大四学长真的是好划水，ddl到了和我们说“看不懂，干脆直接用别人的把”之类的，从头到尾也基本上没干过活。。。但是他快要毕业了我们也就不多做计较了，只是觉得不提出来，会对不起我的另外两个组员，就让我当这个坏人吧。不过还是希望他能好好毕业，毕竟队友一场。

在研究完之后觉得，别人把算法、模型都帮我们写好了，应该难度就不大了吧，然而事情并没有我想的那么简单。即使我们手里有模型，有处理方法，但是对于不同的数据集，不管是特征工程、套用模型还是其他的数据处理，都有着很大的难度，而我们又选了一个数据类型实在是太过多样的题目（我们这个表格+文本+图片的数据真的是太多了，当初为什么要选这种题目），对每种数据都要有特殊的处理方式，同时还要对各种处理方式的优劣，这直接导致了从第七周开始到第八周周末，我们的kaggle的kernel几乎是马不停蹄地在跑我们的数据。跑出来了又改，改完了又跑，一次跑又要等2小时。。。真正意识到，所谓的海量数据处理到底有多消耗计算机资源（一开始没有用kernel的我还把MBP的touchbar搞过热烧坏了几个灯。。。)

不过总的来说，这次的project还是让学到了很多（presentation+report+ensemble这些都让一个人做，能学的不多吗。。所有都要搞懂hhh）。虽然，我们现在还停留在一些应用阶段，还没有完全理解到里面深层次的东西，但也是一大步了。好歹接触了这个世界的前沿hhh现在也越来越觉得大数据真的是一件很好玩的事情，看到天池上那些机器学习穿搭的题目现在也不是觉得难了而是觉得好玩了。不得不说，实践的确能让人学会很多东西啊。

最后的最后，希望程老师说的让我们去发paper是真的hhh（开个玩笑hhh自认为没那水平）

参考文献

我们参考的资料（甚至可以说大部分代码）都在kaggle上有相关的参考资源。这也是这个数据竞赛的特点所在。在这个OOP编程的时代，可以说大量的代码都已经封装好了。我们需要做的就是根据自己的需求去取用这些，并适当修改。当然我们在选择模型的过程中，还有很多很多其他的文献，现在都有些理不清了。只在这里留一些我们的代码中出现的部分吧。因为时间原因（发现没写参考文献的时候已经是26号晚上9点了。。。）没用特别严格的学术reference格式，抱歉！

- 数据类

Data Overview: <https://www.kaggle.com/artgor/exploration-of-data-step-by-step>
(<https://www.kaggle.com/artgor/exploration-of-data-step-by-step>)

Data parser: <https://www.kaggle.com/wrosinski/baselinemodeling>
(<https://www.kaggle.com/wrosinski/baselinemodeling>)

- 评价类

Quadratic Weighted Kappa: <https://www.kaggle.com/aroraaman/quadratic-kappa-metric-explained-in-5-simple-steps>
(<https://www.kaggle.com/aroraaman/quadratic-kappa-metric-explained-in-5-simple-steps>)

Optimized Rounder: <https://www.kaggle.com/naveenasaiithambi/optimizedrounder-improved>
(<https://www.kaggle.com/naveenasaiithambi/optimizedrounder-improved>)

K-cross Validation: <https://www.kaggle.com/jakubwasikowski/stratified-group-k-fold-cross-validation>
(<https://www.kaggle.com/jakubwasikowski/stratified-group-k-fold-cross-validation>)

- 图片处理类:

Image Feature: <https://www.kaggle.com/christofhenkel/extract-image-features-from-pretrained-nn>
(<https://www.kaggle.com/christofhenkel/extract-image-features-from-pretrained-nn>)

- 文本处理类:

TfIdf: <https://www.kaggle.com/bminixhofer/6th-place-solution-code>
(<https://www.kaggle.com/bminixhofer/6th-place-solution-code>)

GloVe: <https://www.kaggle.com/wuyhbb/final-small> (<https://www.kaggle.com/wuyhbb/final-small>)

- 模型类

LGB model: <https://www.kaggle.com/skooch/petfinder-simple-lgbm-baseline>
(<https://www.kaggle.com/skooch/petfinder-simple-lgbm-baseline>)

RNN model: <https://www.kaggle.com/adityaecdrd/8th-place-solution-code>
(<https://www.kaggle.com/adityaecdrd/8th-place-solution-code>)

CatBoost Model: <https://www.kaggle.com/wuyhbb/final-small>
(<https://www.kaggle.com/wuyhbb/final-small>)

- 模型融合

model stacking: <https://www.kaggle.com/ryches/42nd-solution-nothing-special>
(<https://www.kaggle.com/ryches/42nd-solution-nothing-special>)

- 调参: <https://www.kaggle.com/hengzheng/lgb-bayesian-parameters-finding>
(<https://www.kaggle.com/hengzheng/lgb-bayesian-parameters-finding>)