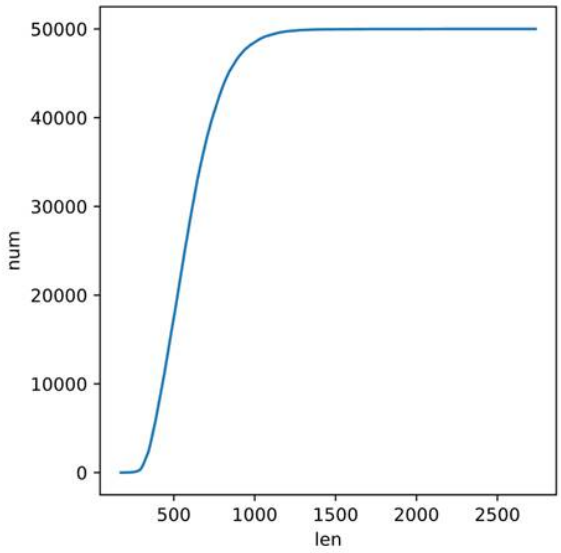
# 高级人工智能大作业报告

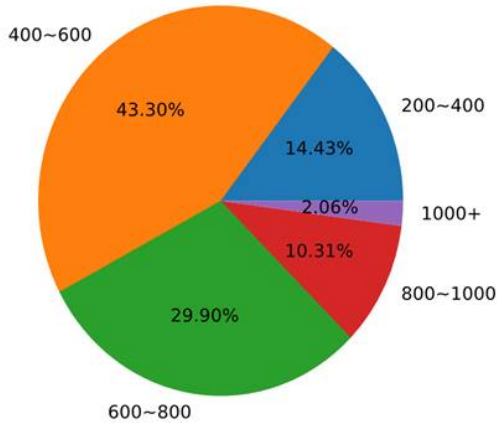
## # 数据分析

对问题数据的分析是构建模型前必不可少的一步，根据数据特点选用合适的模型。并且原始数据也需要经过一定的数据处理才能输入到模型中进行训练，这样能够使模型学到解决问题所需要的关键信息，而不至于被无关信息干扰。这一般被称为数据预处理。

### 长度分析

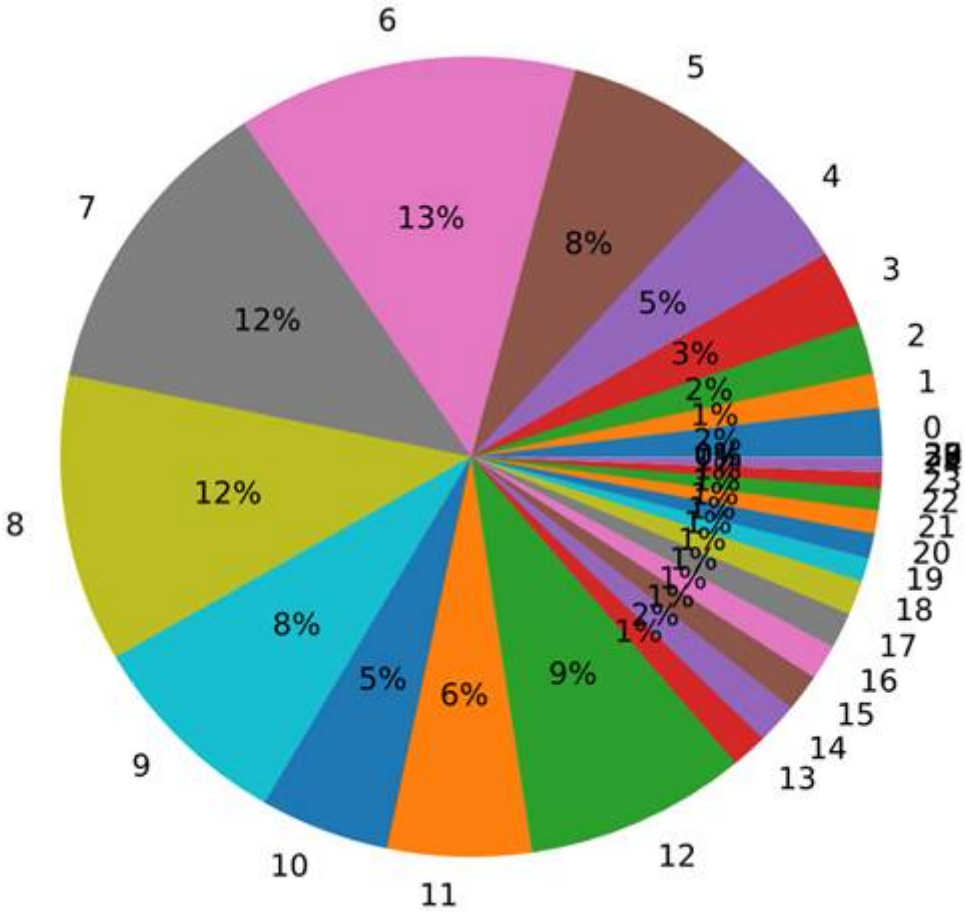
长度分析部分包括数据集长度的累计线型图和分布扇形图。




### 减刑月数分析

减刑月数分析结果如下扇形图所示：




# 数据增强

通过数据增强，充分利用有限的标注数据，获取到更多的训练数据，减少网络中的过拟合现象，训练出泛化能力更强的模型。数据增强起初在计算机视觉领域应用广泛，后逐步在NLP领域取得广泛应用。但由于NLP领域数据的离散型，难以像CV领域一样直接应用。因此采用为文本数据增强设计的方法和技术，快速补充文本数据。

目前，NLP的数据增强大致有加噪和回译两种监督方法。加噪为在原数据的基础上通过替换词、删除词等方式创造和原数据相类似的新数据；回译则是通过翻译为其他语言再翻译回原语言，由于语言之间的不同，回译也能得到有助于训练的新数据。Easy Data Augmentation for Text

Classification Tasks(EDA)提出并验证了几种加噪的Text Augmentation技巧，分别是同义词替换(SR: Synonyms Replace)、随机插入(RI: Randomly Insert)、随机交换(RS: Randomly Swap)、随机删除(RD: Randomly Delete)。

本竞赛中尝试使用的EDA类方法有：

1. **近义词替换**：不考虑StopWords，在句子中随机抽取N个词，从近义词词典随机抽取同义词进行替换。但近义词的词向量十分相似，几乎会被模型认为相同，因此有效的扩充并不多。
2. **随机删除**：句子中的每个词，以概率P随即删除。但随机删除不仅有随机插入的关键词没有侧重的缺点，也有随机交换句式句型泛化效果差的问题。
3. **随机插入**：随机的找出句中某个不属于停用词集的词，并求出其随机的近义词，将该近义词插入句子的一个随机位置。重复N次。但原本的训练数据丧失了语义结构和语义顺序，近义词的加入也并没有侧重句子的关键词。
4. **随机置换邻近的字**：句子中，随机选择两个词，位置交换。该过程可以重复N次。但实质上并没有改变原句的词素，对新句式、句型、相似词的泛化能力实质上提升很有限。

虽然EDA类方法能够有一定程度的提高，但其缺点限制了进一步的提升模泛化性能，因此在竞赛中添加了基于模型的方法，主要分为以下四类：

1. **使用BERT先基于上下文进行一些词替换**：在Pre-Training阶段，首先会通过大量的文本对BERT模型进行预训练，然而，标注样本是非常珍贵的，在BERT中则是选用大量的未标注样本来预训练BERT模型。在Fine-Tuning阶段，会针对不同的下游任务适当改造模型结构，同时，通过具体任务的样本，重新调整模型中的参数。
2. **使用GPT生成训练文本扩大数据集**，效果不好。因为竞赛任务是做回归，每类样本预测的结果是数值型的，是连续的，不像分类任务是每类样本有边界，很大程度会生成噪声样本。
3. **使用SimBERT变换句式**：SimBERT模型是由苏剑林开发的模型，以Google开源的BERT模型为基础，基于微软的UniLM思想设计了融检

索与生成于一体的任务，来进一步微调后得到的模型，所以它同时具备相似问生成和相似句检索能力。SimBERT属于有监督训练，训练语料是自行收集到的相似句对，通过一句来预测另一句的相似句生成任务来构建Seq2Seq部分。

4. **使用翻译模型进行数据增强**：中文->英文，英文->中文( 效果很好，同时进行了同义词替换、句式变换，且基本不改变语义 )。回译数据增强目前是文本数据增强方面效果较好的增强方法, 一般基于google翻译接口, 将文本数据翻译成另外一种语言(一般选择小语种), 之后再翻译回原语言, 即可认为得到与与原语料同标签的新语料, 新语料加入到原数据集中即可认为是对原数据集数据增强。

综上，最后本次竞赛最后采取翻译模型进行数据增强操作，代码如下：

```
def enhance_data():
    train_data = pd.read_csv(f'../user_data/data/train.csv')
    trans_train_data = train_data.copy #pd.read_csv(f'./data/enhance.csv')
    while (True):
        try:
            for i in range(train_data.shape[0]):
                if train_data.iloc[i, 0] != trans_train_data.iloc[i, 0]:
                    continue
                org_text = train_data.iloc[i, 0]
                if len(org_text) > 1000:
                    org_text = org_text[:500] + org_text[-500:]
                if (i + 1) % 10 == 0:
                    print(f'{i} step complete')

 trans_train_data.to_csv(f'./data/enhance.csv', index=False, header=True)
```

```
15                  en_text = ts.baidu(org_text,
    if_use_cn_host=True, from_language='zh',
    to_language='en')
16                  trans_text = ts.baidu(en_text,
    if_use_cn_host=True, from_language='en',
    to_language='zh')
17                  trans_train_data.iloc[i, 0] =
    trans_text
18          except Exception:
19              time.sleep(5)
20              print("Intercept")
21
```

# 模型设计

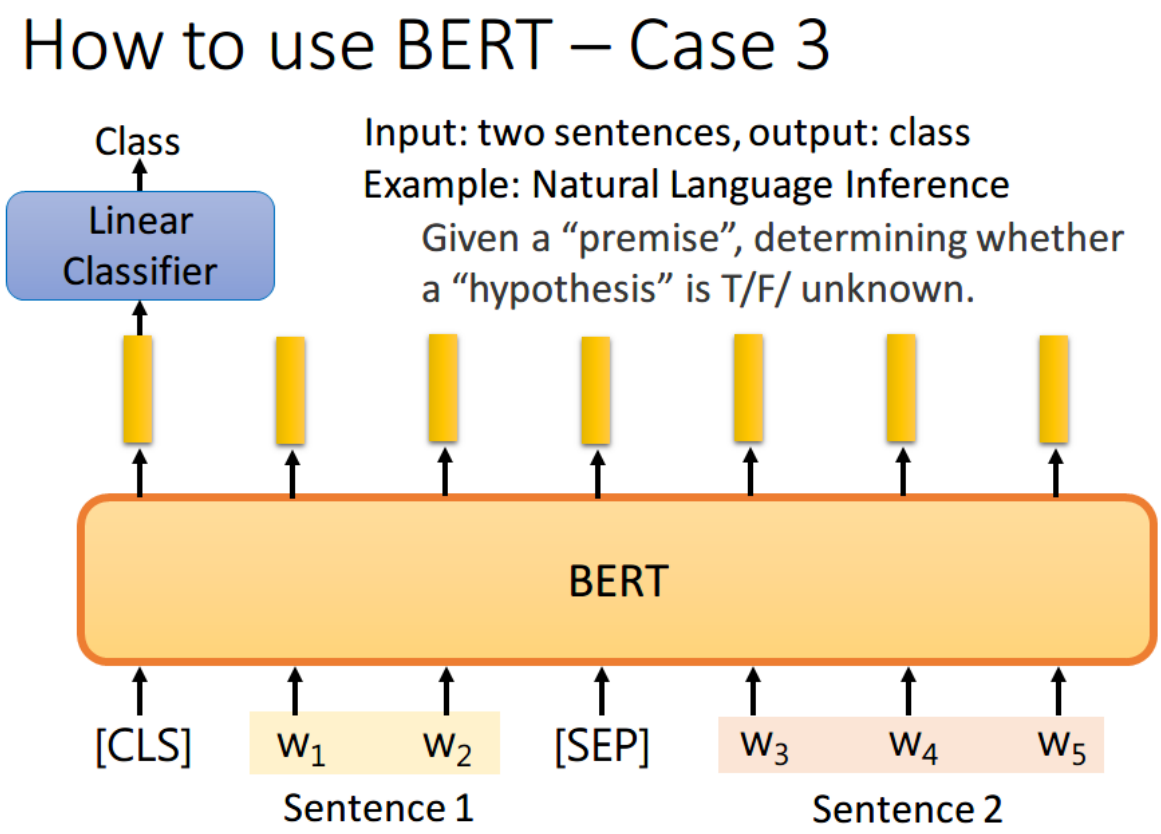
由数据分析部分得到的各文本长度扇形图可得出结果：文本长度选择1000左右可以完美涵盖98%左右得数据，对于最后2%得长度长与1000得数据，认为可以使用截断1000得方式得到近似分类结果。以1000作为训练长度，既不会造成过多的信息丢失，也不会导致大多数文本填充的部分太长影响模型。

但本实验使用的模型BERT具有512字符长度得限制，按照BERT论文中得说法是为了整体计算效率，所以把长度限制在了512。

> To speed up pretraing in our experiments, we pre-train the model with sequence length of 128 for 90% of the steps. Then, we train the rest 10% of the steps of sequence of 512 to learn the positional embeddings.

本质上是由于BERT中的Positional Embedding和Transformer中的Positional Embedding实现方式并不一样，BERT本质上为可学习参数，每个位置所对应的向量就类似于Token Embedding中每个词对应的词向量。故而使用与训练模型时，没办法随意得修改这个词表得长度，需要对文本输入（载入dataloader得部分）进行处理。

本实验主要采用了BERT论文中的03种使用预训练模型的方式，简要图如下图所示：




不同的是，本文并不是简单的分类问题，而是需要额外训练一个打分网络（可以理解为较为复杂的分类模型），来对最后的文本减刑时长进行预测。

## 尝试1 文本截断

此部分，尝试不进行文本处理，直接使用占据了大约60%的500个文字作为输入。即不进行文本预处理，将文本输入BERT，并且由于最终结果是输出时间预测，故而将`【CLS】`位的编码向量拼接输入到一个全连接神经网络进行时间预测分析（由于本次尝试结果较为失败，报告的此部分不进行代码展示了）。

> [CLS]: BERT中的特殊分隔符号 ，该位置可表示整句话的语义。对应的编码是101

最终结果：在比赛给出的验证集A中正确率仅有0.3。结果较差，说明直接进行文本截断效果浪费了很大一部分信息，需要考虑长文本输入BERT情况下对文本进行处理。

## 尝试2 RNN循环法

根据尝试1，本次选择采用1050个文字，长度不足的使用`[PAD]`位进行填充，长度超过部分截断（针对部分超长字段，仅占比2%左右）。

> [PAD]：针对有长度要求的场景，填充文本长度，使得文本长度达到要求。对应编码是0

为了保证长字段有效，对句子分割使用数据有重叠分割思路（overlap），这样分割后的每句句子之间仍然保留了一定的关联信息，每段长度为300，重叠长度为50。示例代码如下：

```
def get_split_text(text, split_len=300, overlap_len=50):
    split_text=[]
    for w in range(len(text)//split_len):
        if w == 0:   #第一次,直接分割长度放进去
            text_piece = text[:split_len]
        else:      # 否则, 按照(分割长度-overlap)往后走

            window = split_len - overlap_len
            text_piece = [w * window: w * window + split_len]
            split_text.append(text_piece)
  return split_text

```

把分割好后的文本送入BERT进行训练，仍然用[CLS] token作为句子表示。

获得如下一组数据：

```
分段1的embedding
分段2的embedding
分段3的embedding
```

随后把这些embedding拼回起来, 变成了

```
[分段1的embedding,分段2的embedding, 分段3的embedding]
```

将这部分拼接输入到双向GRU循环神经网络进行时间预测，训练代码这里略去（同样因为效果不是很好）,思路主要为：

1. 取最后一步state的输入进行时间预测
2. 取文章未填充前分段数的一步的state输入全连接网络进行时间预测。

最终结果：A部分验证集上正确率为0.45左右。

## 尝试3 编码均值法

此部分同样取1000个文字，长度不足使用`[PAD]`填充，长度超过便截断，每段长度为500（左右，实际应为510），取BERT最后四层每段的`[CLS]`的编码向量的平均，拼接到一个具有三层全连接神经网络进行时间预测。选用3层的原因：3层具有较好的表达能力，同时训练难度不会太大，避免因为BERT的输出结果快速变动使得更高层书的全连接神经网络无法收敛（或者无法在算力较为稀缺的情况下收敛）。

最终结果：A部分验证集上正确率为0.49左右。

## 尝试4（最终方案）编码向量拼接

在尝试3的三层神经网络表现较好的前提下，考虑加入将尝试2中的编码向量拼接输入到三层神经训练网络中，具体流程如下：取1000个文字，长度不足使用`[PAD]`填充，长度超过便取前500个文字与后500文字。输入BERT，取每段的`[CLS]`的编码向量拼接输入到三层全连接神经网络进行时间预测。

首先是长文本处理流程（使用了一定trick的data_loader）：

```
def load_data(path, is_train=True):
    data = pd.read_csv(path, encoding='utf-8')
    print(data.shape)
    text_list_org = data.iloc[:, 0].tolist()
    lengths = []
    text_list = []
    for text in text_list_org:
        lengths.append(min(3, int(len(text) / 500)))
        if len(text) >= 1000:
            text = text[:500] + text[-500:]
        else:
```

```
            text += "[PAD]" * (1000 - len(text))
        text_list.extend([text[:500], text[500:1000]])
    if is_train:
        labels = data.iloc[:, 1].tolist()
    else:
        labels = [0] * data.iloc[:].shape[0]
    # 调用encoder函数，获得预训练模型的三种输入形式
    input_ids, token_type_ids, attention_mask = encoder(max_len=512, vocab_path="bert-base-chinese",

 text_list=text_list)
    labels = torch.tensor(labels, dtype=torch.float)
    # 将encoder的返回值以及label封装为Tensor的形式
    data = TensorDataset(input_ids, token_type_ids, attention_mask, labels)
    return data
```

核心训练代码：

```
for epoch in range(total_epochs):
        train_loss = 0
        step_loss = 0
        correct = 0
        total = 0
        for step, (input_ids, token_type_ids, attention_mask, labels) in enumerate(train_loader):
            # 从实例化的DataLoader中取出数据，并通过.to(device)将数据部署到服务器上
            input_ids, token_type_ids, attention_mask, labels = input_ids.to(device), \

             token_type_ids.to(device), \

             attention_mask.to(device), \
```

```
11
          labels.to(device)
12            model.train()
13            # 梯度清零
14            optimizer.zero_grad()
15            # 将数据输入到模型中获得输出
16            out_put = model(input_ids, token_type_ids,
attention_mask).view(-1)
17            # 计算损失
18            loss = criterion(out_put, labels)
19            predict = out_put.data
20            predict =
torch.round(predict).type(torch.int)
21            labels =
torch.round(labels).type(torch.int)
22            correct += (predict ==
labels).sum().item()
23            total += labels.size(0)
24            train_loss += loss.item()
25            step_loss += loss.item()
26            loss.backward()
27            optimizer.step()
28            # 每100次进行一次打印
29            if (step + 1) % 100 == 0:
30                train_acc = correct / total
31                logger.info("Train
Epoch[{}/{}],step[{}/{}],"
32                            "tra_acc{:.6f}
%,,step_loss:{:.6f},ep_loss:{:.6f}".format(epoch + 1,
total_epochs,
33
                                step + 1,
len(train_loader),
34
                                train_acc * 100,
step_loss / 50,
```

```
                                    train_loss / step))
                step_loss = 0
            # 每3500次进行一次验证
            if (step + 1) % 3500 == 0:
                train_acc = correct / total
                # 调用验证函数dev对模型进行验证，并将有效果提升的模型进行保存
                acc, test_loss, test_mse_loss = dev(model, dev_loader, device)
                if bestAcc < acc:
                    bestAcc = acc
                if best_loss > test_loss:
                    best_loss = test_loss
                    # 模型保存路径
                    fold = args['fold']
                    path = f'./model/fold/bert_fold{fold}_{int(epoch / 5)}.pkl'
                    torch.save(model, path)
                logger.info(
                    "DEV Epoch[{}/{}],step[{}/{}],tra_acc{:.6f} %,bestAcc{:.6f}%,dev_acc{:.6f} %,test_my_loss:{:.6f},test_mae_loss:{:.6f},best_loss{:.6}".format(
                        epoch + 1, total_epochs, step + 1, len(train_loader),
                        train_acc * 100, bestAcc * 100, acc * 100,
                        test_loss.item(), test_mse_loss.item(), best_loss.item()))
        scheduler.step(bestAcc)
```

# 分层学习率设置

## bert学习率

BERT原文提示用的学习率为**[5e-5, 3e-5, 2e-5]** ，推荐用这些效果可能较好，但本文实际是有一个下游学习任务的（三层全连接神经网络打分），故而原论文的fine-tune并不是很适合。

在小数据集上的简单验证显示，BERT中的学习率使用较高的值，可以更快的收敛，得到相应的并不太深的embedding参数，在下游学习中得到的结果反而更好，故本实验设置的是BERT学习率较高。

## 下游任务学习率

下游任务学习任务本质是一个三层全连接神经网络，具有较好的表达能力。由于观察数据集后发现的得到的打分结果之间的差异性并不是很明显，评分数据集中在1-12之间， 12-25部分占比已经很少，所以设置下游任务学习率较低，可以得出一个较为精细的打分网络

综上，本实验设置BERT学习率较高而下游任务学习率较低。

# 损失函数

实验一开始计划使用nn库函数提供的L1损失或L2损失函数，但进行系列试验后得到结果显示：

- 使用L2损失函数可能因为平方关系收敛效果不太好，且异常数据可能较多，导致L2损失函数效果反而不如L1损失函数

- 单纯使用L1损失函数会有梯度恒定问题，导致结果容易发散，错过可能的极值点，而且导数不连续问题使得L1几乎不单独使用。

故实验使用了L1损失进行模型训练，使用竞赛的评价函数进行模型选择，竞赛评价指标公式如下：

$$FinalScore = \sum_{i=1}^{N} Score_i \times 0.7 + ExtAcc \times 0.3$$

$$score = \begin{cases} 1.0, v < 0.2 \\ 0.8, 0.2 < v < 0.4 \\ \dots \end{cases}$$

$$v = |\log(l_p + 1) - \log(l_a + 1)|$$

其中，$v$衡量预测出的减刑月份$l_p$与案件标准减刑月份$l_a$之间的差值距离；$ExtAcc$为预测结果与真实值的绝对匹配准确率；$FinalScore$由$Score_i$和$ExtAcc$的加权求和计算。

我们对其进行了近似实现，代码如下：

```
1
2  class Critic(nn.Module):
3      def __init__(self, device):
4          super().__init__()
5          self.device = device
6          self.relu = nn.ReLU()
7
8      def forward(self, predicts, labels):
9          predicts = self.relu(predicts)
10         ...
11         l1 = (torch.abs(torch.log((1 +
   torch.round(predicts)) / (1 + labels))) - 1).sum()
12         l2 = -(torch.round(predicts) == labels).sum()
13         l = (l1 * 0.7 + l2 * 0.3) / predicts.shape[0]
14         return l
```

# K折交叉验证

在训练过程中，需要划分出一部分训练数据作为验证集，进行模型选择。如果使用单次划分，有部分数据被划分到训练集，就无法用于训练，造成数据浪费。使用k折交叉，可以利用到全部训练数据，而且可以得到多个模型，然后对模型各个模型在训练集上的预测结果进行整合，可以提高性能。.预测数据整合尝试了如下三种方案：

1. 将不同模型的时间取平均再取整
2. 将不同模型生成的评分取整，选取众数作为最终预测时间
3. 投票与取均值相结合，将10 fold的预测结果取整，进行投票若获得最多票数的时间的票数不超过5，则将该时间作为最终预测时间，否则选择平均值作为最终预测时间

最后选择方案三进行数据整合，代码实现如下：

```
def majorityLabel(labels):
    return Counter(list(labels)).most_common()[0]

df_list = []
for i in range(1, 8):
    df =
pd.read_csv(f'./submission/submission_fold{i}.csv')
    df.iloc[:, 1] = df.iloc[:, 1]
    df_list.append(df)

# 众数出现次数大于5，选众数，否则取均值
submission = df_list[2].copy()
c = 0
for i in range(submission.shape[0]):
```

```
    time_l = [int(np.round(df.iloc[i, 1])) for df in df_list]
    time_f_l = np.array([df.iloc[i, 1] for df in df_list])
   # print(majorityLabel(time_l), time_l)
    maj_res = majorityLabel(time_l)
    if maj_res[1] >= 5:
        c += 1
        submission.iloc[i, 1] = maj_res[0]
    else:
        submission.iloc[i, 1] = np.average(time_f_l)
    #submission.iloc[i, 1] = np.average(time_f_l)
```

# 竞赛总结

本竞赛我们首先对训练数据进行了数据分析，了解其文本长度、减刑月数分布规律。然后对调研了不同的数据增强方式，通过尝试，选择了基于翻译模型的数据增强方式。在模型方面，使用Huggingface的bert-base-chinese预训练模型，尝试了不同的文本截断方式和下游任务网络结构。由于时间关系，没有做样本平衡相关的尝试，可以做未来的改进点。

本竞赛最后结果为B榜第三名，分工见PPT部分。