目录

W2-138王泽群-NLP-Noob

2 1.1 k 7

好未来AI实战训练营第二周，NLP营。 项目内容：手写英文作文评分 Automated Essay Scoring

138-王泽群-NLP LV2 1

Fork 点赞

内容 数据集 Fork记录 7 评论

内容

版本 3 2018-02-09 22:33

TAIL人工智能实战训练营 -- NLP

目录

第二周：英语作文自动评分

小组：NLP-Noob 成员：王泽群、宋文泰

目录

- 第二周学习任务概况
- 第二周实战任务分析
- 解决任务流程
- 具体算法介绍与部分结果展示
- 训练营收获

第二周学习任务概况

在我看来，NLP营第二周的学习任务主要重在一下三点：

- 继续上周的NLP基础知识的学习，本周任务是关于语法文法部分，比如Parsing、Readability，同时学习新的模型比如N-gram
- 学习简单的深度神经网络的搭建，学习keras的基本使用
- 了解作文自动评分的大致思路，阅读相关文献

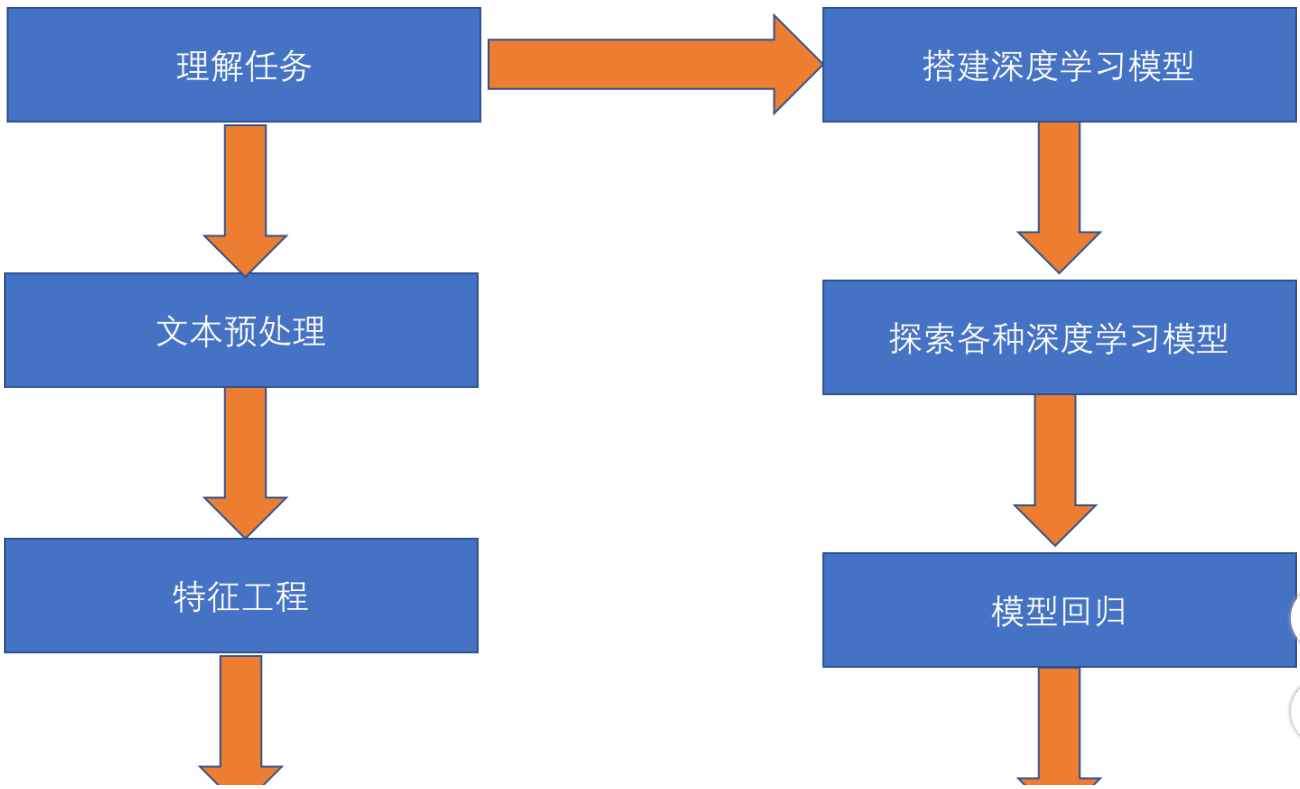
第二周实战任务分析

第二周的实战任务针对学生所写的英文作文进行机器自动评分。虽然所写作文都是围绕着同一个topic展开，但是本着适用性更强的想法， 无论是选择的特征还是深度学习方法都没有考虑topic本身，而是使用了一些Task-independent的特征。目的是模型能在不同主题的作文上有更好的迁移能力。

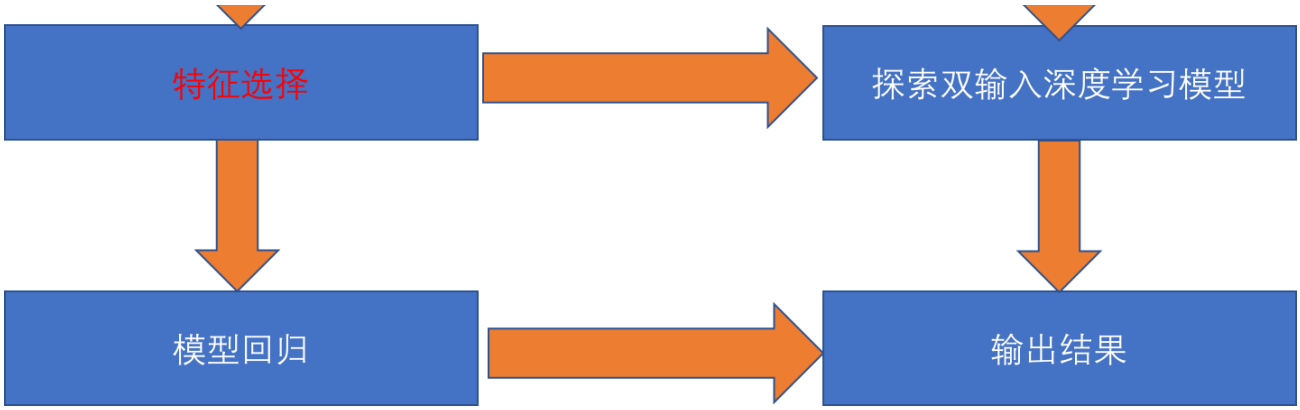
解决任务流程

我的确没什么能创新的地方……中规中矩地复现论文，把这个问题转化成一个数据挖掘问题，锻炼自己的基本功吧！

- 读取数据及数据预处理
- 分析数据本身特征
- 阅读论文，提取（数十个）特征
- 艰难的特征选择（完全可以只剩三个）
- 通过模型回归（40个特征carefully调参的XGB VS **3个特征最简单的ElasticNet线性回归**）
- 尝试深度学习解决问题（Embedding->Conv1D->LSTM->Dense->preds）
- 尝试用传统**特征工程+深度学习**来解决问题（将模型增加为**两个输入**）



目录



具体算法介绍与部分结果展示

先上特征工程的结果：

最简单的线性回归在仅仅三个特征达到了线上0.8537的成绩

In [63]:

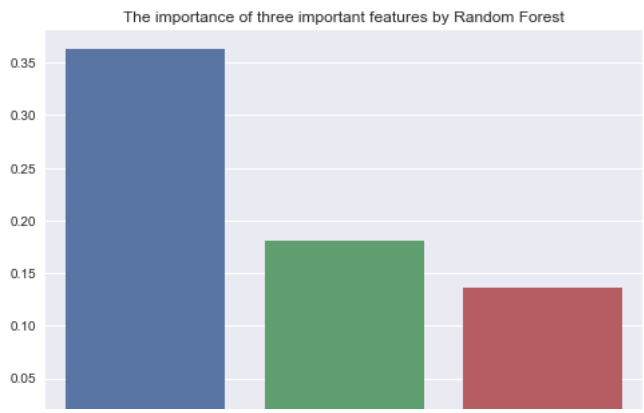
```
#线性模型
from sklearn.linear_model import Lasso, Ridge,ElasticNet
import scipy as sc
lso = Lasso(alpha=0.1)
lso.fit(X.loc[train_id,:], data.loc[train_id, 'finals'])
print(lso.score(X.loc[test_id,:],data.loc[test_id,'finals']))
print(sc.stats.pearsonr(lso.predict(X.loc[test_id,:]),data.loc[test_id,'finals']))

rg = Ridge()
rg.fit(X.loc[train_id,:], data.loc[train_id, 'finals'])
print(rg.score(X.loc[test_id,:],data.loc[test_id,'finals']))
print(sc.stats.pearsonr(rg.predict(X.loc[test_id,:]),data.loc[test_id,'finals']))

el = ElasticNet(alpha=0.1, l1_ratio=0.5)
el.fit(X.loc[train_id,:], data.loc[train_id, 'finals'])
print(el.score(X.loc[test_id,:],data.loc[test_id,'finals']))
print(sc.stats.pearsonr(el.predict(X.loc[test_id,:]),data.loc[test_id,'finals']))
print(el.coef_)
el_selected_features = []
for index, weight in enumerate(el.coef_):
    if np.abs(weight)>0:
        el_selected_features.append(X.columns[index])
print(el_selected_features)
```

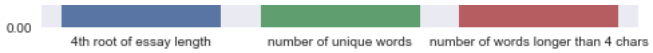
```
0.726047525455465
(0.860141414470894, 8.433410753021622e-60)
0.7356366415163893
(0.8602114919997087, 8.054517607627863e-60)
0.7292065479039072
(0.8602008705174808, 8.110844703283255e-60)
[0.47848923 0.49007352 0.27743841]
['uni_words_num', 'len_4_root', 'word_longer_4']
```

In [127]:

```
plt.title('The importance of three important features by Random Forest')
sns.barplot(['4th root of essay length', 'number of unique words', 'number of words longer than 4 chars'],[0.36293
```

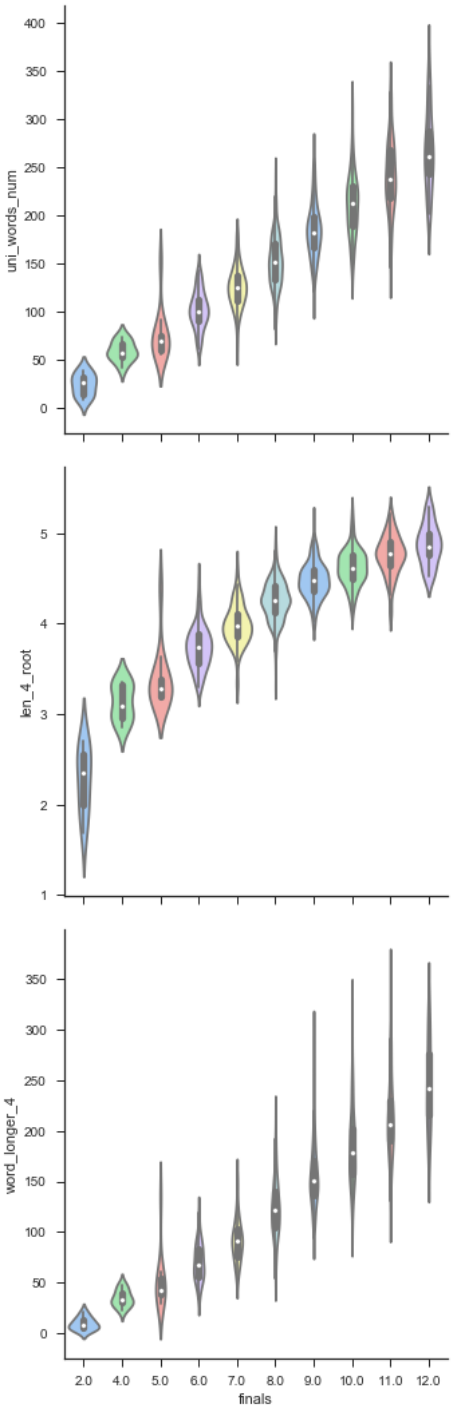
Out[127]:





In [227]:

```
sns.set(style="ticks")
g = sns.PairGrid(data,
             x_vars = 'finals',
            y_vars = ['uni_words_num', 'len_4_root', 'word_longer_4'],
            aspect=1,size=5) # 设置间距和图片大小
g.map(sns.violinplot,palette='pastel')
plt.show()
```



读取数据

In [2]:

```
with open('essay_train.txt') as f:
    train = pd.read_table(f, names=['id','essay','t1s','t2s','finals'])
```

In [3]:

```
with open('test_data.txt') as f:
    test = pd.read_table(f, names=['id','essay'])
```

In [4]:

```
data = pd.concat([train, test]).set_index('id')
```

In [6]:

```
raw_text = ''
for i in data['bow']:
    for j in i:
        raw_text += j+' '
```

In [7]:

```
raw_bow = text.text_to_word_sequence(raw_text)
```

数据预处理

这次没有进行特别多的预处理。把文章tokenize之后，对动词进行了统计和lemmatize，计算了各自的tf-idf。当然还有正常的POS Tagging, Parsing等等。主要使用了任务书上介绍的神器textblob库和方便的keras.preprocess。 其他的都是文章本身的特征。 在这个任务中，我认为词袋模型不是一个特别好用的方法，原因是作文的评分和语法逻辑等等息息相关，在某些情况下必须考虑前后文和词语形式，还有文章长度。以动词举例：I love you.写成I loving you.从语法上是错误的。

In [5]:

```
data['bow'] = data['essay'].apply(text.text_to_word_sequence)
```

In [5]:

```
import textblob

def get_blob(essay):
    return textblob.TextBlob(essay)
data['blob'] = data['essay'].apply(get_blob)
```

In [488]:

```
data['tags'] = data['blob'].apply(lambda x:x.tags)
```

In [489]:

```
data['parse'] = data['blob'].apply(lambda x:x.parse())
```

In [497]:

```
data['verb'] = data['tags'].apply(lambda x:[i[0] for i in x if i[1].startswith("V")])
```

In [503]:

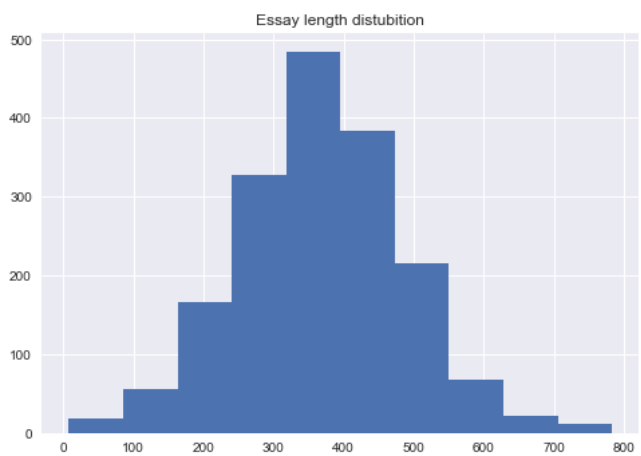
```
from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
data['verb_lem'] = data['verb'].apply(lambda x:[lemmatizer.lemmatize(i, pos='v').lower() for i in x])
```

项目本身的一些数据分布

In [220]:

```
plt.title('Essay length distubition')
data['pass_length'] = data['bow'].apply(len)
data['pass_length'].hist()
```

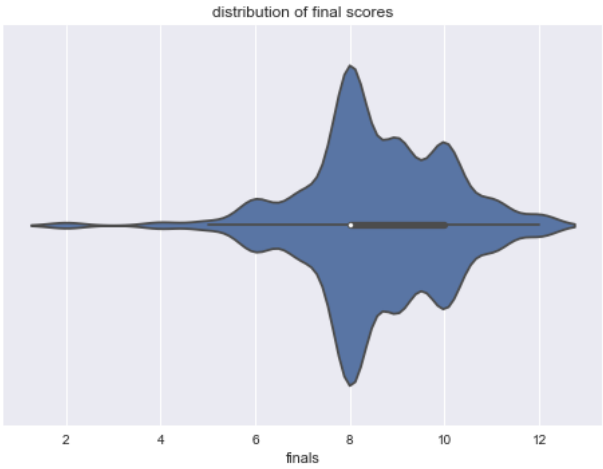
Out[220]:



In [222]:

```
plt.title('distribution of final scores')
sns.violinplot(x=data["finals"])
```

Out[222]:



可以看到，文章的长度在200~500之间，50%评分集中在8~10分左右。在低评分部分有极少的离群点。事实上应该去除这部分离群点，但是去除之后成绩略有下降，但没有下降太多，说明这个办法是可行的。最后选择了有权重地把带有离群点和不带离群点的预测值进行平均，获得了一部分提升。

阅读论文，开始特征工程

通过阅读论文，发现前人进行了非常多的特征的尝试，总结如下：

Length Features

文章长度

> 与作文的给定写作时间有直接关系

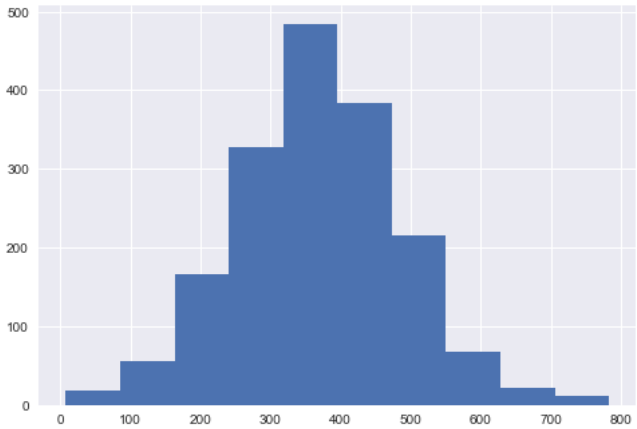
作文都是在有限时间内写完的，所以长度可以有效表征作者的写作能力进而表征作文的质量。

> 计算token数和sentence数

In [73]:

```
data['pass_length'] = data['bow'].apply(len)
data['pass_length'].hist()
```

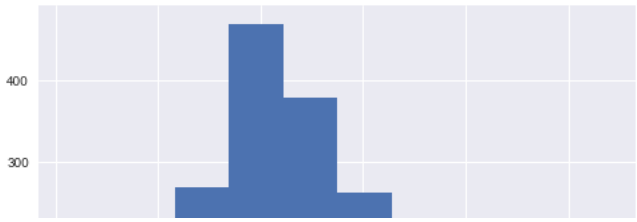
Out[73]:



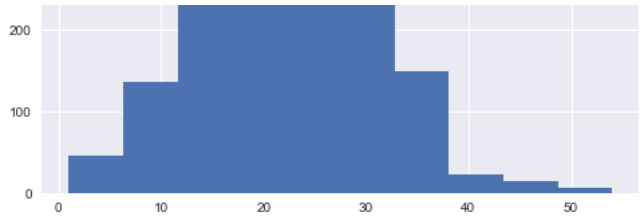
In [78]:

```
data['sentences_num'] = data['essay'].apply(lambda x:len(x.split('.')))

data['sentences_num'].hist()
```

Out[78]:



目录

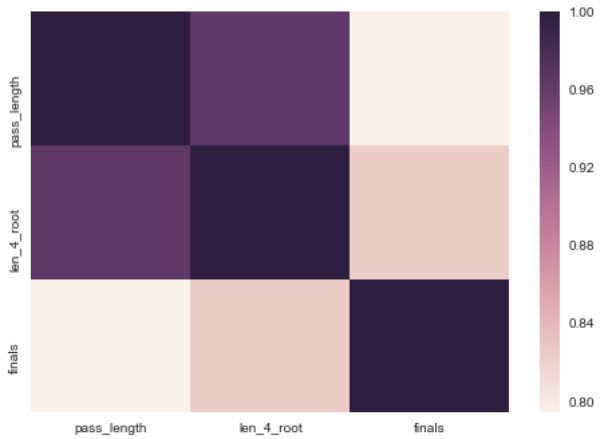


在阅读另外一篇论文的时候提到，相对于文章的token数而言，文章长度的**四次方根**和总分的相关关系更强：

In [84]:

```
data['len_4_root'] = data['pass_length'].apply(lambda x:math.pow(x,0.25))
sns.heatmap(data[['pass_length','len_4_root','finals']].corr())
```

Out[84]:



平均句子长度和词汇长度

> 与任务弱相关

表征作者驾驭复杂文本的能力。

In [17]:

```
data['sentences'] = data['essay'].apply(lambda x:x.split('.'))
```

In [19]:

```
data['sent_length'] = data['sentences'].apply(lambda x:np.mean([len(i.split(' ')) for i in x]))
```

In [20]:

```
data['word_length'] = data['bow'].apply(lambda x:np.mean([len(i) for i in x]))
```

Occurrence Features

逗号、引号、感叹号的出现

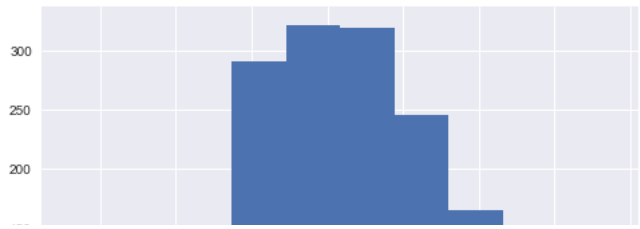
> 与任务弱相关

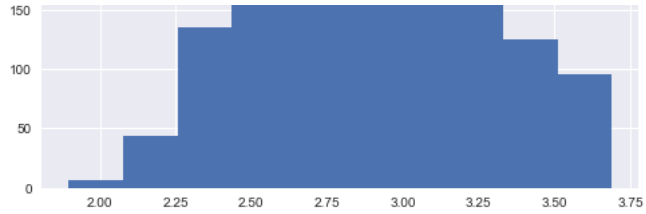
文章中写这是比较有价值的特征，提取了逗号的出现，统计了频率，为了尽量使样本正态分布取了对数（我不知道这个有没有用）

In [86]:

```
data['comma_oc'] = data['essay'].apply(lambda x:len(re.findall(',',x)))
data['comma_ratio'] = data['comma_oc']/data['pass_length']
data['comma_log'] = data['comma_ratio'].apply(lambda x:-np.log(x+0.025))
data['comma_log'].hist()
```

Out[86]:





（对给材料的作文）正式引用

> 转述动词等规则区分是否为引语

这方面的确是和作文任务更相关，在网上找了关于report verb的list,最后发现和本次任务的相关度不大。

In [491]:

```
report_verbs_raw = '''
admit
advise*
agree
announce
claim
complain
confirm
declare
explain
insist
mention
promise*
propose
say
suggest
warn*
advise
assure
convince
inform
notify
persuade
promise
reassure
remind
tell
warn
agree
ask
claim
demand
offer
promise
propose
refuse
threaten
advise
ask
beg
convince
encourage
forbid
instruct
invite
order
persuade
remind
tell
urge
warn
admit
deny
mention
proposed
report
suggest
'''
```

In [494]:

```
report_verb_list = [i.strip('*') for i in report_verbs_raw.split()]
```

In [497]:

```
data['verb'] = data['tags'].apply(lambda x:[i[0] for i in x if i[1].startswith("V")])
```

In [503]:

```
from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
data['verb_lem'] = data['verb'].apply(lambda x:[lemmatizer.lemmatize(i, pos='v').lower() for i in x])
```

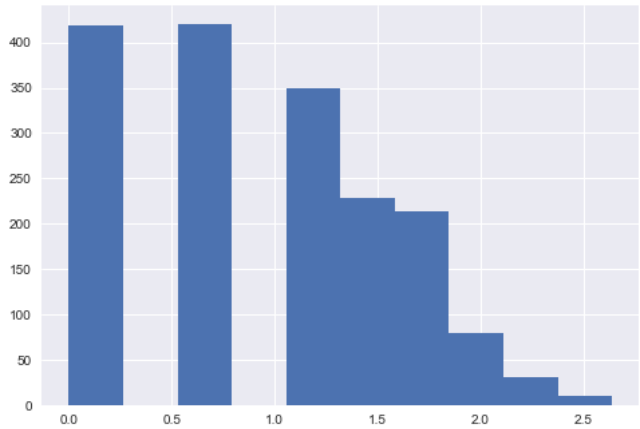
In [504]:

```
def get_report_verb_count(verb_lem):
    count = 0
    for i in verb_lem:
        if i in report_verb_list:
            count += 1
    return count
data['report_verb'] = data['verb_lem'].apply(get_report_verb_count)
```

In [508]:

```
data['report_log'] = data['report_verb'].apply(np.log1p)

data['report_log'].hist()
```

Out[508]:



（对给材料的作文）核心概念仅给材料时使用

> 识别核心概念：对材料中高tf.idf的词汇或者短语

没有考察这个特征

句法变化

与任务弱相关

> 不同的句法树的占比，平均树高 时间和原因状语从句的比例, 时间和原因状语从句的引导词

句法树的解析没有找到合适的库，而blob自己的Parser（用的应该是pattern这个库作为依赖）里面用的词汇我真的完全看不懂，所以对句法树深度的解析就没有作为特征。但是找了一些其他的特征作为句法变化的特征。

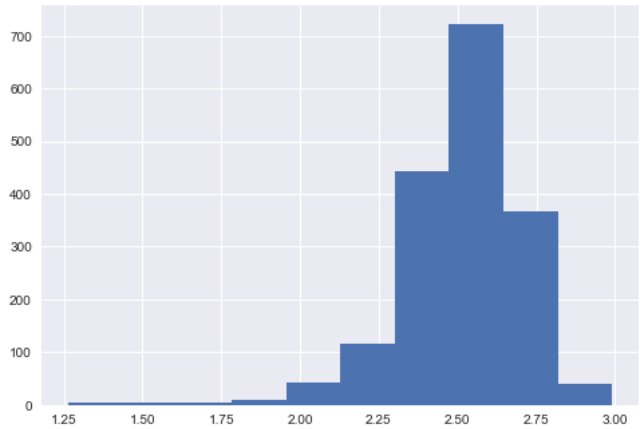
- 连接词的出现和比例
- pos_tag的出现和比例

In [88]:

```
a = '''表递进moreover, in addition, what is more, furthermore, also, then, besides, etc. 表转折however, nevertheless,

表示解释说明的过渡词：for example, in fact, in this case, for actually等。    表示总结的过渡词：finally, at last, in conclu
conj = re.findall('[a-z ]+',a)
conj.remove( 'near to far from in the front of beside behind to the right to the left on the other side of   ')

conj_list = [i.strip() for i in conj]

conj_list = [i for i in conj_list if i]

conj_list = set(conj_list)

len(conj_list)

def count_conj(essay):
    count = 0
```

```
    for i in conj_list:
        if i in essay:
            count += 1
    return count

data['conj_count'] = data['essay'].apply(count_conj)

data['conj_ratio'] = data['conj_count']/data['pass_length']
data['conj_ratio_s'] = data['conj_count']/data['sentences_num']
data['conj_log'] = data['conj_ratio'].apply(lambda x:-np.log(x+0.021))
data['conj_log'].hist()
```

Out[88]:



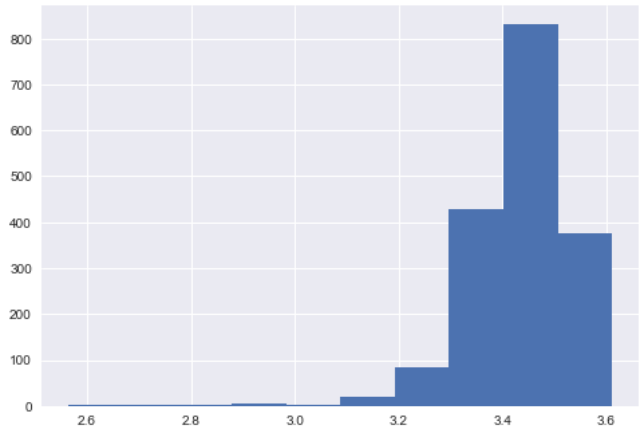
In [566]:

```
def grammer_tag_count(tags):
    return len(set([i[1] for i in tags]))

data['tag_var'] = data['tags'].apply(grammer_tag_count)
```

In [582]:

```
data['tag_var_log'] = data['tag_var'].apply(lambda x:np.log(x+7))
data['tag_var_log'].hist()
```

Out[582]:



情态动词和介词的使用

对于语法特征来说，情态动词和介词也是非常重要的一环。

In [1377]:

```
def get_mode(tags):
    count = 0
    for i in tags:
        if 'MD' in i[1]:
            count+=1
    return count
data['modes'] = data['tags'].apply(get_mode)
```

In [1390]:

```
def get_prep(tags):
    count = 0
    for i in tags:
        if 'IN' in i[1]:
            count += 2
```

```
    return count
data['preps'] = data['tags'].apply(get_prep)
```

In [1395]:

```
data['preps_ratio'] = data['preps']/data['pass_length']
```

目录

Style Features

formality F

与任务强相关

$$F = \frac{\Sigma_{i \in A} \frac{c(i)}{n} - \Sigma_{j \in B} \frac{c(j)}{n} + 100}{2}$$

A= {N, ADJ, PP, DET}, B= {PR, V,ADV, UH}, and n is the number of tokens in thetext

这个指标应该是用来衡量行文风格的，但是后来发现和评分关系并不大

In [611]:

```
def formalityF(tags, pass_length):
    A= 'N, ADJ, PP, DET'.split(', ')
    B= 'PR, V, ADV, UH'.split(', ')
    c_A = len([i[1] for k in A for i in tags if i[1].startswith(k)])/pass_length
    c_B = len([i[1] for k in B for i in tags if i[1].startswith(k)])/pass_length
    F = (c_A - c_B + 100)/2
    return F
data['F'] = data.apply(lambda x:formalityF(x['tags'],x['pass_length']),axis=1)
```

type-token-ratio

用以估计作者词汇量的大小，这里使用了pos_tag、句子成分的数量和对句子个数的平均来体现。

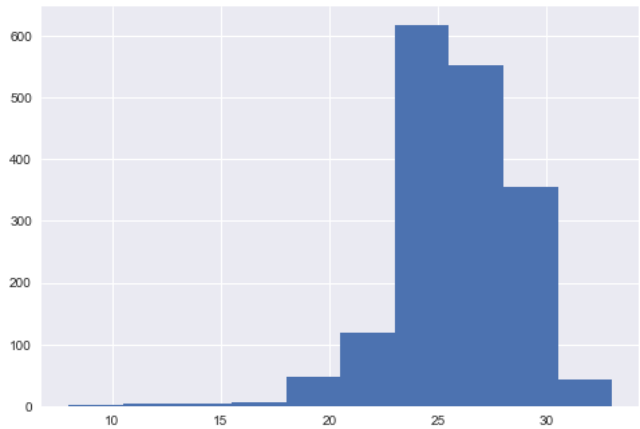
In [428]:

```
def get_grammer(parse):
    parse_list = parse.split(' ')
    grammer_list = [i.split('/') for i in parse_list]
    pos = set()
    gram = set()
    for i in grammer_list:
        if i:
            pos.add(i[1])
            gram.add(i[2])
    return len(pos),len(gram)
pg = data.apply(lambda x:get_grammer(x['parse']),axis=1)
data['pos_num'] = np.array([i[0] for i in pg])
data['gram_num'] = np.array([i[1] for i in pg])
```

In [474]:

```
data['pos_num'].hist()
```

Out[474]:



In [489]:

```
data['grammer_complexity'] = data['gram_num']/data['sentences_num']
```

In [490]:

```
data['pos_complexity'] = data['pos_num']/data['sentences_num']
```

目录

Coherence Features与任务弱相关

> measuring the topical overlap between adjacent sentences. We use similarity measures based on n-gram overlap and redundancy (e.g. of nouns).

没看懂，没实现。

Error Features

> 与任务弱相关语法和拼写错误是一篇烂文的明显标志, 但被发现在作文评分中影响并不大.

文章里说是基于规则的简单语法错误检测，但我实在不知道怎么实现语法规则的检测函数。 所以用拼写来代替考察这一项。而拼写，这里使用了Textblob.correct()方法来获得修改拼写错误之后的作文，同时检测原作文和corrected作文的编辑距离和相似度。 其中编辑距离使用了Levenshtein这个包。

In [ ]:

```
def get_correct(blob):
    print(blob[:10])
    return blob.correct()
data['correct'] = data['blob'].apply(get_correct)
```

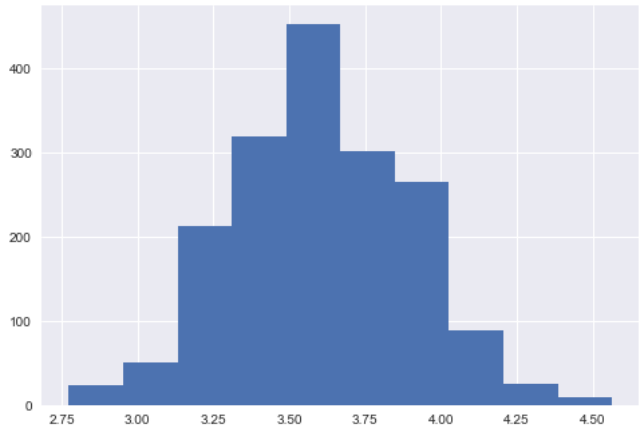
In [16]:

```
import Levenshtein
data['correct_distance'] = data.apply(lambda x:Levenshtein.distance(x['essay'],str(x['correct'])), axis=1)
```

In [99]:

```
data['correct_log'] = data['correct_distance'].apply(lambda x:np.log(x+16))
data['correct_log'].hist()
```

Out[99]:



In [131]:

```
import math
def  get_counts(sequence):
    counts={}
    for i in sequence:
        if i in counts:
            counts[i]+=1
        else:
            counts[i]=1
    return counts
def get_union(s1,s2):
    set1=set(get_counts(s1).keys())
    set2=set(get_counts(s2).keys())
    set3=set1|set2
    return set3
def get_cos_similarity(s1,s2):
    set0=get_union(s1,s2)
    a1=[]
    a2=[]
    for i in set0:
        a1.append(s1.count(i))
        a2.append(s2.count(i))
    sum1=0
    len1=0
    len2=0
    for i in range(len(a1)):
        sum1+=a1[i]*a2[i]
        len1+=math.pow(a1[i],2)
        len2+=math.pow(a2[i],2)
```

```
        len2+=math.pow(a2[i],2)
    s=math.sqrt(sum1/float(len1*len2))
    return s
```

In [132]:

```
data['correct_cos'] = data.apply(lambda x:get_cos_similarity(x['essay'],str(x['correct'])), axis=1)
```

In [89]:

```
data['cor_cos_log'] = data['correct_cos'].apply(lambda x:-np.log(x+0.4))
```

Readability Features

与任务弱相关

> 非常简单或非常复杂的写作方式都是烂文的标志. Rely on normalized counts of words, letters, syllables or other phenomena(like abbreviations) which affect the readability. high scoring essays might be associated with different levels of readability

这里没有自己去实现readability features，而是使用了任务书上的开源包textstat来实现。几乎用到了tutorial里面所有的方法。

In [642]:

```
from textstat.textstat import textstat
```

In [643]:

```
data['flesch'] = data['essay'].apply(textstat.flesch_reading_ease)
```

In [644]:

```
data['smog'] = data['essay'].apply(textstat.smog_index)
```

In [645]:

```
data['dale_chall_readability_score'] = data['essay'].apply(textstat.dale_chall_readability_score)
```

In [648]:

```
data['linsear_write_formula'] = data['essay'].apply(textstat.linsear_write_formula)
```

In [649]:

```
data['gunning_fog'] = data['essay'].apply(textstat.gunning_fog)
```

In [679]:

```
data['automated_readability_index'] = data['essay'].apply(textstat.automated_readability_index)
```

Set-Dependent Features

> n-grams features和 POS n-grams features

关于n-grams features，并没有统计太多。预想的是统计所有的n-grams的Tf-Idf，但是因为时间比较紧迫就没有实现，我也不知道哪个包能做到这一点。所以仅仅统计的相同的个数和频率。

In [937]:

```
def get_2grams(blob):
    return blob.ngrams(n=2)
data['2-grams'] = data['blob'].apply(get_2grams)
```

In [938]:

```
def get_3grams(blob):
    return blob.ngrams(n=3)
data['3-grams'] = data['blob'].apply(get_3grams)
```

In [945]:

```
two_gram_set = set()
two_gram_list = []
for i in data['2-grams']:
    for j in i:
        two_gram_set.add(tuple(j))
        two_gram_list.append(tuple(j))
```

In [946]:

```
three_gram_set = set()
three_gram_list = []
for i in data['3-grams']:
    for j in i:
        three_gram_set.add(tuple(j))
```

```
        three_gram_list.append(tuple(j))
```

In [954]:

```
import collections
two_grams_count = collections.Counter(two_gram_list)
three_grams_count = collections.Counter(three_gram_list)
```

In [955]:

```
len(two_gram_list)
```

Out[955]:

In [963]:

```
def get_two_gram_times(two_grams):
    times = []
    for i in two_grams:
        times.append(two_grams_count[tuple(i)])
    return times
get_two_gram_times([('I','love'),('love','your')])
```

Out[963]:

In [964]:

```
data['two_gram_times'] = data['2-grams'].apply(get_two_gram_times)
```

In [966]:

```
def get_three_gram_times(three_grams):
    times = []
    for i in three_grams:
        times.append(three_grams_count[tuple(i)])
    return times
get_two_gram_times([('I','love','your'),('love','your','letter')])
```

Out[966]:

In [967]:

```
data['three_gram_times'] = data['3-grams'].apply(get_three_gram_times)
```

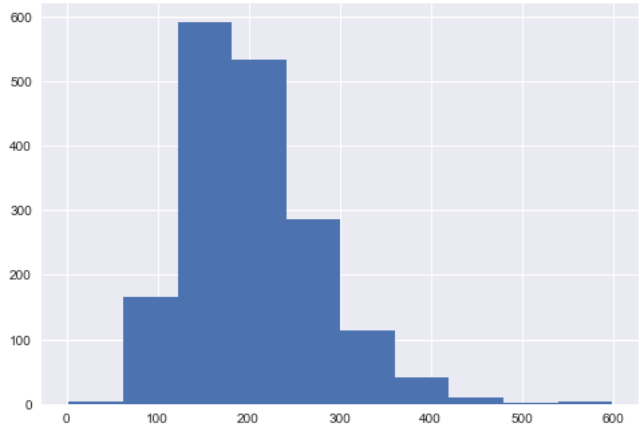
In [969]:

```
data['sum_two_grams'] = data['two_gram_times'].apply(lambda x:np.sum(x))
data['avg_two_grams'] = data['two_gram_times'].apply(lambda x:np.mean(x))
```

In [990]:

```
data['avg_two_grams'].hist()
```

Out[990]:



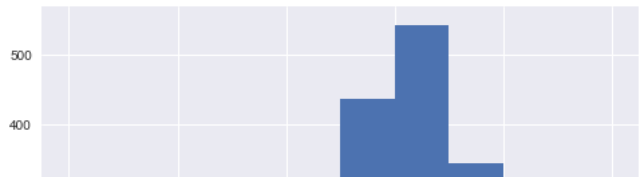
In [1105]:

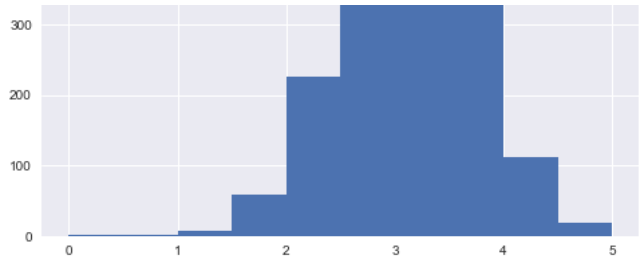
```
data['sum_3_grams'] = data['three_gram_times'].apply(lambda x:np.sum(x))
data['avg_3_grams'] = data['three_gram_times'].apply(lambda x:np.mean(x))
```

In [1109]:

```
data['avg_3_grams_log'] = data['avg_3_grams'].apply(np.log)
data['avg_3_grams_log'].hist()
```

Out[1109]:





In [1126]:

```
def get_pos_2grams(tags):
    pos_2grams = []
    for i in range(len(tags)-1):
        pos_2grams.append((tags[i][1],tags[i+1][1]))
    return pos_2grams
data['pos_2grams'] = data['tags'].apply(get_pos_2grams)

pos_two_gram_list = []
for i in data['pos_2grams']:
    for j in i:
        pos_two_gram_list.append(tuple(j))

pos_two_grams_count = collections.Counter(pos_two_gram_list)


def get_pos_two_gram_times(pos_two_grams):
    times = []
    for i in pos_two_grams:
        times.append(pos_two_grams_count[tuple(i)])
    return times
data['pos_two_gram_times'] = data['pos_2grams'].apply(get_pos_two_gram_times)

data['post_two_gram_times_avg'] = data['pos_two_gram_times'].apply(np.mean)

data['post_two_gram_times_avg'].describe()
```

In [1157]:

```
def get_pos_3grams(tags):
    pos_3grams = []
    for i in range(len(tags)-2):
        pos_3grams.append((tags[i][1],tags[i+1][1],tags[i+2][1]))
    return pos_3grams
data['pos_3grams'] = data['tags'].apply(get_pos_3grams)
```

In [1160]:

```
pos_three_gram_list = []
for i in data['pos_3grams']:
    for j in i:
        pos_three_gram_list.append(tuple(j))

pos_three_grams_count = collections.Counter(pos_three_gram_list)
```

In [1168]:

```
def get_pos_three_gram_times(pos_three_grams):
    times = []
    for i in pos_three_grams:
        times.append(pos_three_grams_count[tuple(i)])
    return times
data['pos_three_gram_times'] = data['pos_3grams'].apply(get_pos_three_gram_times)
data['pos_three_gram_times_avg'] = data['pos_three_gram_times'].apply(np.mean)

data['pos_three_gram_times_avg'].describe()
```

Out[1168]:

average word frequency

与任务弱相关 作者词汇量的丰富程度与语料库中词汇的频率关系很大.作者使用的词汇在语料库中的频率越小,作者的语言熟练度越高 We model this idea by calculating the average wordfrequency in the Web1T-corpus (Brants and Franz,2006).

tf-idf

Tf-Idf是一种词频上的统计方法，衡量的是某一个单词在这篇文档中的重要程度。计算方式是词频与逆文档频率的乘积。

词频越高，在所有文档中露面的次数越低，Tf-Idf值就会越大。 这里统计了每一个句子的tf-idf向量各维度的和，还有他们的平均值。

In [35]:

In [35]:

```
# data['lem_essay']
vectorizer=CountVectorizer()
transformer=TfidfTransformer()
tfidf=transformer.fit_transform(vectorizer.fit_transform(data['lem_essay']))
word=vectorizer.get_feature_names()
weight=tfidf.toarray()
```

```
/Users/a/anaconda3/lib/python3.6/site-packages/sklearn/feature_extraction/text.py:1089: FutureWarning: Conversion of
  if hasattr(X, 'dtype') and np.issubdtype(X.dtype, np.float):
```

In [1440]:

```
sum_list = []
for i in weight:
    sum_list.append(i.sum())
data['tfidf_list'] = sum_list
```

In [1442]:

```
data['tfidf_mean'] = data['tfidf_list']/data['pass_length']
```

In [742]:

```
weight_test = pd.DataFrame(weight,index=data.index)
```

Lexical Resource

根据另外一个论文，困难的词汇可以展现为词的长度。分别统计了长于4 6 8 10的词汇，和长于各个数字的句子的个数。

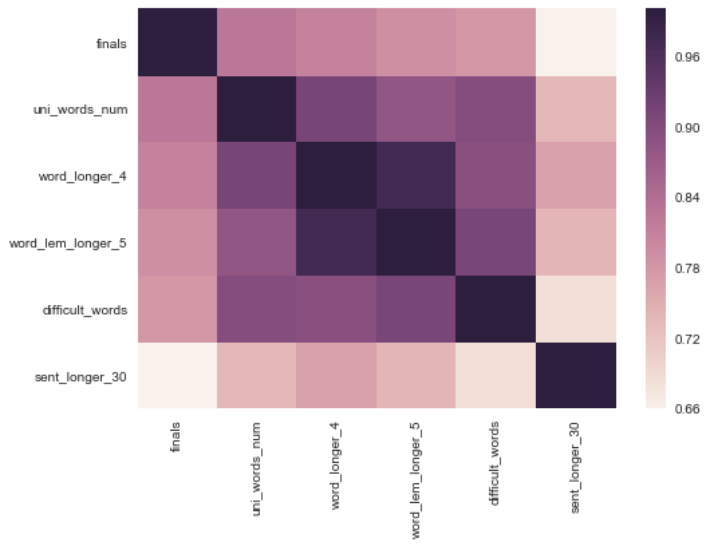
同时使用了多少个不重复的词汇也完整展现了作者的词汇量。

这类特征非常重要。重要到下图的程度：

In [94]:

```
lexical_resource = data[['finals','uni_words_num','word_longer_4','word_lem_longer_5','difficult_words','sent_long
sns.heatmap(lexical_resource)
# 可以看到，这五个特征和总分的皮尔逊相关系数都在0.66以上。
```

Out[94]:



In [647]:

```
data['difficult_words'] = data['essay'].apply(textstat.difficult_words)
```

In [3]:

```
def get_word_longerthan(bow, n):
    count = 0
    for i in bow:
        if len(i)>n:
            count += 1
    return count
```

In [4]:

```
data['word_lem_longer_5'] = data['lem_bow'].apply(lambda x:get_word_longerthan(x,5))
```

In [1249]:

```
data['word_longer_4'] = data['bow'].apply(lambda x:get_word_longerthan(x,4))
data['word_longer_6'] = data['bow'].apply(lambda x:get_word_longerthan(x,6))
data['word_longer_8'] = data['bow'].apply(lambda x:get_word_longerthan(x,8))
```

```
data['word_longer_10'] = data['bow'].apply(lambda x:get_word_longerthan(x,10))
data['word_longer_12'] = data['bow'].apply(lambda x:get_word_longerthan(x,12))
```

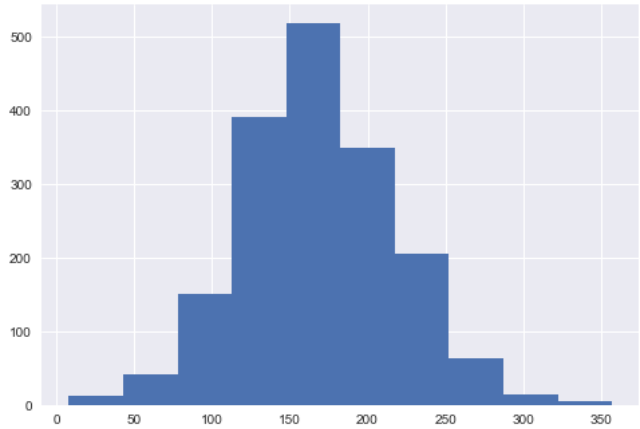
In [1297]:

```
data['uni_words_num'] = data['bow'].apply(lambda x:len(set(x)))
data['uni_words_ratio'] = data['uni_words_num']/data['pass_length']
```

In [1298]:

```
data['uni_words_num'].hist()
```

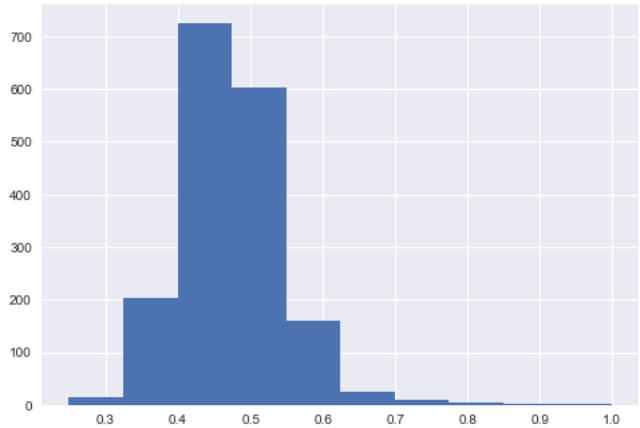
Out[1298]:



In [1300]:

```
data['uni_words_ratio'].hist()
```

Out[1300]:



In [1334]:

```
data['sent_longer_10'] = data['sentences'].apply(lambda x:get_word_longerthan(x,10))
data['sent_longer_18'] = data['sentences'].apply(lambda x:get_word_longerthan(x,18))
data['sent_longer_25'] = data['sentences'].apply(lambda x:get_word_longerthan(x,25))
data['sent_longer_30'] = data['sentences'].apply(lambda x:get_word_longerthan(x,30))
```

把所有的特征放在一起，归一化后用XGB拟合输出。

通过GridSearchCV的方法调整XGB的参数，以便获得最好的结果。

In [107]:

```
A = data[cross_valid_features]
X = pd.DataFrame()
for column in A.columns:
    X[column] = (A[column] - A[column].mean()) / A[column].std()
```

In [ ]:

```
from sklearn.grid_search import GridSearchCV

import warnings
warnings.filterwarnings('ignore')
```

In [ ]:

```
train_data = data.loc[range(1,1201)]
test_data = data.loc[range(1201,1751)]
xgb = XGBRegressor(
```

目录

```
# #                         colsample_bytree=0.9,
#                           gamma=3,
#                           learning_rate=0.01,
#                           max_depth=9,
#                           min_child_weight=3,
#                           n_estimators=896,
#                           reg_alpha=0.011,
#                           reg_lambda=2.1,
#                           subsample=0.545,
# #                           silent=0,
#                           seed =1024,
#                           nthread = -1
)
xgb.fit(X.loc[train_id,X.columns],data.loc[train_id,'finals'])
sc.stats.pearsonr(xgb.predict(X.loc[test_id,X.columns]),data.loc[test_id,'finals'])[0],xgb.score(X.loc[test_id,X.c
```

In [ ]:

```
param_test1 = {'colsample_bytree':[0.5,0.7,0.9,1], 'gamma':list(range(1,10,2))}
gsearch1 = GridSearchCV(XGBRegressor(learning_rate=0.01,
                         max_depth=9,
                         min_child_weight=3,
                         n_estimators=896,
                         reg_alpha=0.011,
                         reg_lambda=2.1,
                         subsample=0.545,
#                          silent=0,
                         seed =1024,
                         nthread = -1),param_test1,cv=5)
gsearch1.fit(np.array(X.loc[train_id,:]),np.array(data.loc[train_id,'finals']))
gsearch1.grid_scores_,
```

In [ ]:

```
gsearch1.best_params_, gsearch1.best_score_
```

In [ ]:

```
param_test2 = {'colsample_bytree':[0.8,0.9,1], 'gamma':list(range(6,9))}
gsearch2 = GridSearchCV(XGBRegressor(
    learning_rate=0.01,
                         max_depth=9,
                         min_child_weight=3,
                         n_estimators=896,
                         reg_alpha=0.011,
                         reg_lambda=2.1,
                         subsample=0.545,
#                          silent=0,
                         seed =1024,
                         nthread = -1),param_test1,cv=5)
gsearch2.fit(np.array(X.loc[train_id,:]),np.array(data.loc[train_id,'finals']))
gsearch2.grid_scores_, gsearch2.best_params_, gsearch2.best_score_
```

In [ ]:

```
param_test3 = {'learning_rate':[0.005,0.01,0.015], 'max_depth':list(range(9,13))}
gsearch3 = GridSearchCV(XGBRegressor(
    colsample_bytree=1,
    gamma=7,
                         min_child_weight=3,
                         n_estimators=896,
                         reg_alpha=0.011,
                         reg_lambda=2.1,
                         subsample=0.545,
#                          silent=0,
                         seed =1024,
                         nthread = -1),param_test3,cv=5)
gsearch3.fit(np.array(X.loc[train_id,:]),np.array(data.loc[train_id,'finals']))
gsearch3.grid_scores_, gsearch3.best_params_, gsearch3.best_score_
```

In [ ]:

```
param_test4 = {'min_child_weight':list(range(1,10,3))}
gsearch4 = GridSearchCV(XGBRegressor(
    colsample_bytree=1,
    gamma=7,
    learning_rate=0.01,
    max_depth=9,
                         n_estimators=896,
                         reg_alpha=0.011,
                         reg_lambda=2.1,
                         subsample=0.545,
#                          silent=0,
                         seed =1024,
                         nthread = -1),param_test4,cv=5)
```

```
gsearch4.fit(np.array(X.loc[train_id,:]),np.array(data.loc[train_id,'finals']))
gsearch4.grid_scores_, gsearch4.best_params_, gsearch4.best_score_
```

In [ ]:

```
param_test5 = {'n_estimators':list(range(750,950,50)), 'reg_alpha':[0.005,0.01,0.013],'min_child_weight':list(rang
gsearch5 = GridSearchCV(XGBRegressor(
    colsample_bytree=1,
    gamma=8,
    max_depth=9,
    min_child_weight=1,
    learning_rate=0.01,
                          reg_lambda=2.1,
                          subsample=0.545,
#                           silent=0,
                          seed =1024,
                          nthread = -1),param_test5,cv=5)
gsearch5.fit(np.array(X.loc[train_id,:]),np.array(data.loc[train_id,'finals']))
gsearch5.grid_scores_, gsearch5.best_params_, gsearch5.best_score_
```

In [ ]:

```
param_test6 = {'reg_lambda':[1,2,3,4], 'reg_alpha':[0.003,0.005,0.007]}
gsearch6 = GridSearchCV(XGBRegressor(
    colsample_bytree=1,
    gamma=8,
    max_depth=9,
    min_child_weight=1,
    learning_rate=0.01,
    n_estimators=900,
                          subsample=0.545,
#                           silent=0,
                          seed =1024,
                          nthread = -1),param_test6,cv=5)
gsearch6.fit(np.array(X.loc[train_id,:]),np.array(data.loc[train_id,'finals']))
gsearch6.grid_scores_, gsearch6.best_params_, gsearch6.best_score_
```

In [ ]:

```
param_test7 = {'subsample':[0.3,0.5,0.7,0.9]}
gsearch7 = GridSearchCV(XGBRegressor(
    colsample_bytree=1,
    gamma=8,
    max_depth=9,
    min_child_weight=1,
    learning_rate=0.01,
    n_estimators=900,
    reg_lambda=1,
    reg_alpha=0.007,

#                           silent=0,
                          seed =1024,
                          nthread = -1),param_test7,cv=5)
gsearch7.fit(np.array(X.loc[train_id,:]),np.array(data.loc[train_id,'finals']))
gsearch7.grid_scores_, gsearch7.best_params_, gsearch7.best_score_
```

In [ ]:

```
from xgboost import XGBRegressor
xgb_final = XGBRegressor(
    colsample_bytree=1,
    gamma=8,
    max_depth=9,
    min_child_weight=1,
    learning_rate=0.01,
    n_estimators=900,
    reg_lambda=1,
    reg_alpha=0.007,subsample=0.5)

xgb_final.fit(np.array(X.loc[train_id,:]),np.array(data.loc[train_id,'finals']))
sc.stats.pearsonr(xgb_final.predict(np.array(X.loc[test_id,:])),data.loc[test_id,'finals'])[0],xgb_final.score(np.
```

In [ ]:

```
xgb_final.fit(np.array(X.loc[iindex,el_selected_features]),np.array(data.loc[iindex,'finals']))

preds = xgb_final.predict(np.array(X.loc[range(1201,1751),el_selected_features]))
```

In [ ]:

```
from xgboost import XGBRegressor
xgb_final = XGBRegressor(
    colsample_bytree=0.7,
    gamma=5,
    max_depth=8,
    min_child_weight=3,
```

```
    learning_rate=0.01,
    n_estimators=750,
    reg_lambda=3,
    reg_alpha=0.013,subsample=0.3)
xgb_final.fit(M.loc[range(1,1201),related_features], data.loc[range(1,1201), 'finals'])

preds = xgb_final.predict(M.loc[range(1201,1751),related_features])
```

目录

特征选择

错误的特征选择方式

特征选择走了很多弯路，尝试了很多方法，甚至试过把特征一个一个去掉，看评分是否会升高的方法，事实证明当然是错误的。

In [108]:

```
def feature_extract(RegressorClass,n=0, mode=None):
    """
    用于某个回归器选择特征。
    mode:None:上升即挑选，'cando'：上升0.001也挑选，'solid'，下降0.001以上才挑选
    如果去掉某个特征之后，5次中有四次分数上升，那么去掉这个特征。
    features:字典。对应特征：分数上升次数
    init_score：原分数

    """
    if not n:
        if mode =="cando":
            n = 0.001
        elif mode =="solid":
            n = -0.01
        else:
            n=0
    features = {}
    print('preparing for feature extraction of', RegressorClass)
    for i in X.columns:
        features[i] = 0
    for i in range(5):
        train_id = random.sample(list(range(1,1201)),1000)
        test_id = [i for i in list(range(1,1201)) if i not in train_id]

        init_score_list = []
        for i in range(20):
            regressor = RegressorClass()
            regressor.fit(X.loc[train_id,:], data.loc[train_id, 'finals'])
            init_score_list.append(sc.stats.pearsonr(regressor.predict(X.loc[test_id,:]),data.loc[test_id,'finals'
        init_score = np.mean(init_score_list)
        print('init pearsonr is {}'.format(init_score))
        feature_score_dict = {}
        for i in X.columns:
            # 去掉某一个特征，测试现在的分值
            feature_test = X.drop(i, axis=1)
            score_list = []
            for j in range(6):
                regressor = RegressorClass()
                regressor.fit(feature_test.loc[train_id,:], data.loc[train_id, 'finals'])
                score_list.append(
                    sc.stats.pearsonr(regressor.predict(feature_test.loc[test_id,:]),data.loc[test_id,'finals'])[0
            feature_score_dict[i] = np.mean(score_list)
#             print("if drop {}, the pearsonr is {}".format(i,feature_score_dict[i]))
        # 如果分值变大说明特征并不好
        for feature, score in feature_score_dict.items():
            if score > init_score + n:
                features[feature] += 1
#         print('now features behaviors:{}'.format(features))
    features_selected = [i[0] for i in features.items() if i[1]<4]
    print('overall features behaviors:{}'.format(features))
    print('droped {}'.format([i for i in X.columns if i not in features_selected]))
    return features_selected
#     return feature_score_dict
```

合理的特征选择手段

为了线上能够更高性能地使用，当然希望自己的特征和总分呈线性关系。阅读了论文中的特征选择方式，也查阅了不少资料，总结了自己的特征选择的方法如下。

1.和总分的相关系数

皮尔逊相关系数是衡量两个变量相关性的一种统计量。当然这个相关指的是线性相关。就算两个变量的相关系数接近0，也只能说基本没有什么线性关系，而不能保证两个变量是独立的。计算方法如下：

$$r = \frac{1}{n-1}\Sigma(\frac{X_i - \overline{X}}{\sigma_X})(\frac{Y_i - \overline{Y}}{\sigma_Y})$$

两个变量相关系数的绝对值越大，线性相关性越强，如果是正值，意味着这两个变量正相关。

目录

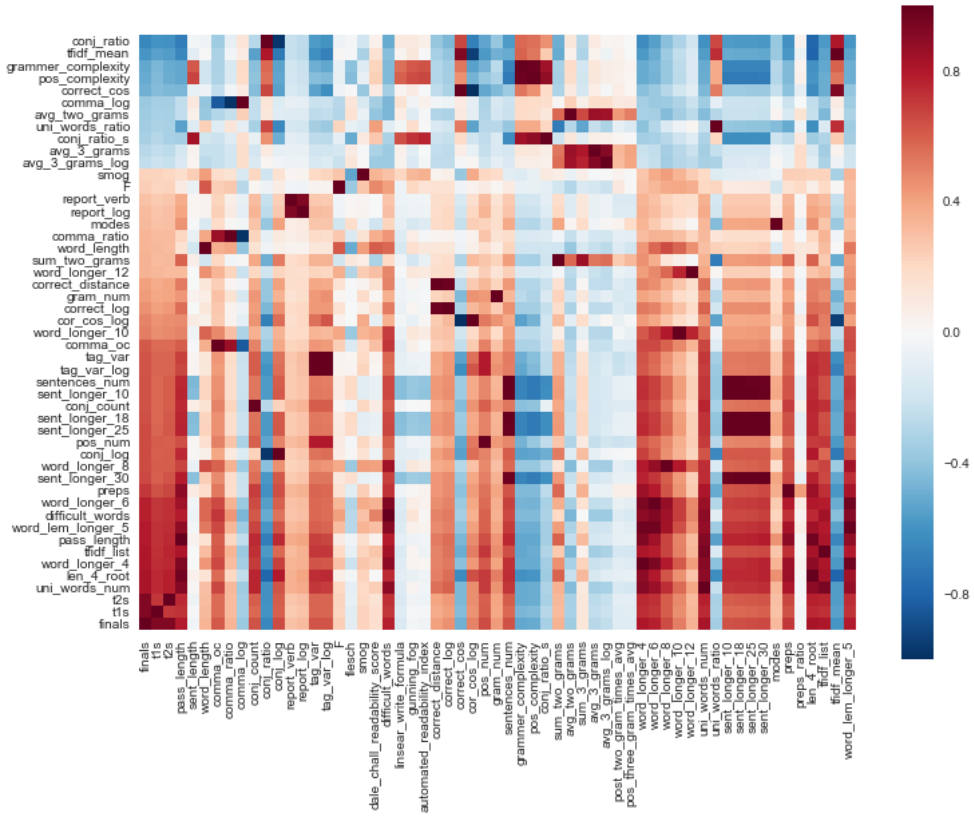
首先观察每一个特征和总分的皮尔逊相关系数，去除掉绝对值<0.2的。

后来发现，其实相关系数为负的特征可以直接去掉。因为相关系数为负有很多原因：

eg：像number of unique words对于总字数的平均（uni_words_ratio），显示出了负的相关系数，原因很可能是那些写得很短（仅仅几十个字）的文章不重复的词汇占比过高导致的。所以这样的特征没有参考价值。最终只考虑了正相关的特征。

In [125]:

```
import seaborn as sns
corrmat = data.corr().sort_values(by='finals')
corrmat = corrmat[np.abs(corrmat['finals'])>0.2]
related_feature = corrmat.index
f, ax = plt.subplots(figsize=(12, 9))
sns.heatmap(corrmat, vmax=.8, square=True);
```



2.各自的相关系数

如果两个特征的相关系数>0.9，说明这两个特征几乎是线性的，最后可能会出现多重共线性（感谢余骁锋同学）的情况。 找到了和超过2个特征都出现线性情况的特征，每对至少去掉一个。(但是最后还是留下了word_longer_4和uni_words_num这对，舍不得这么高的权重)

In [21]:

```
for i in related_features:
    count = 0
    pairs = []
    for j in related_feature:
        if i != j and np.abs(corrmat.loc[i,j]) > 0.9:
            count += 1
            pairs.append((i,j))
    if count > 1:
        print(pairs)
```

```
[('tfidf_mean', 'correct_cos'), ('tfidf_mean', 'cor_cos_log')]
[('conj_log', 'comma_log'), ('conj_log', 'comma_ratio')]
[('comma_log', 'conj_log'), ('comma_log', 'comma_ratio')]
[('comma_ratio', 'conj_log'), ('comma_ratio', 'comma_log')]
[('cor_cos_log', 'tfidf_mean'), ('cor_cos_log', 'correct_cos')]
[('sent_longer_18', 'sentences_num'), ('sent_longer_18', 'sent_longer_10'), ('sent_longer_18', 'sent_longer_25'), (
[('sent_longer_25', 'sentences_num'), ('sent_longer_25', 'sent_longer_10'), ('sent_longer_25', 'sent_longer_18'), ('
[('sent_longer_30', 'sentences_num'), ('sent_longer_30', 'sent_longer_10'), ('sent_longer_30', 'sent_longer_18'), ('
[('difficult_words', 'word_longer_6'), ('difficult_words', 'word_lem_longer_5')]
[('word_lem_longer_5', 'word_longer_6'), ('word_lem_longer_5', 'difficult_words'), ('word_lem_longer_5', 'word_longe
[('word_longer_4', 'word_longer_6'), ('word_longer_4', 'word_lem_longer_5'), ('word_longer_4', 'pass_length'), ('wor
```

```
[('uni_words_num', 'pass_length'), ('uni_words_num', 'triat_list'), ('uni_words_num', 'word_longer_4')]
```

3.Lasso和随机森林的特征选择功能

Lasso是加了L1正则化项的线性回归，能有效地防止普通的线性回归的权重过大导致过拟合的情况。它的损失函数如下：

$$J(\theta)=\frac{1}{2m}[\Sigma_i(h_\theta(x^{(i)})-y^{(i)})^2+\lambda\Sigma_j|\theta_j|]$$

结果是如果某一项的权重过小，在Lasso回归的权重中会直接将它降到0，所以只需要找到有权重且为正的就可以了。当然实际我应用的是岭回归和Lasso的结合ElasticNet线性回归来拟合的。

随机森林是通过有放回的行采样和选择性的列采样（远小于真实的列数）构建巨大的森林，每棵树单独对样本进行预测，最后投票决定输出的一种决策树的变体，它在训练之后会给出哪些特征比较重要。所以完全可以通过决策树来获得最重要的几个特征。

在实际情况上，我首先使用了ElasticNet获得权重为正的一系列特征，然后接着放入随机森林重新拟合，观察随机森林给出的最重要的几个特征。获得了前文提到过的三个特征。用这三个特征得到了线上85的分数。

In [216]:

```
#线性模型
from sklearn.linear_model import Lasso, Ridge,ElasticNet
import scipy as sc
lso = Lasso(alpha=0.1)
lso.fit(X.loc[train_id,:], data.loc[train_id, 'finals'])
print(lso.score(X.loc[test_id,:],data.loc[test_id,'finals']))
print(sc.stats.pearsonr(lso.predict(X.loc[test_id,:]),data.loc[test_id,'finals']))

rg = Ridge()
rg.fit(X.loc[train_id,:], data.loc[train_id, 'finals'])
print(rg.score(X.loc[test_id,:],data.loc[test_id,'finals']))
print(sc.stats.pearsonr(rg.predict(X.loc[test_id,:]),data.loc[test_id,'finals']))

el = ElasticNet(alpha=0.1, l1_ratio=0.5)
el.fit(X.loc[train_id,:], data.loc[train_id, 'finals'])
print(el.score(X.loc[test_id,:],data.loc[test_id,'finals']))
print(sc.stats.pearsonr(el.predict(X.loc[test_id,:]),data.loc[test_id,'finals']))
print(el.coef_)
el_selected_features = []
for index, weight in enumerate(el.coef_):
    if np.abs(weight)>0:
        el_selected_features.append(X.columns[index])
print(el_selected_features)
```

```
0.7465157848682785
(0.8701864484332202, 8.87242149178796e-63)
0.7493779510361792
(0.8669546918138292, 8.561480760721574e-62)
0.7521344999322426
(0.8714930384021679, 3.4872454042868286e-63)
[ 0.         -0.          0.          0.06207725  0.03306594 -0.01565172
  0.         -0.04047894  0.05683575 -0.         -0.          0.
  0.          0.0006821  -0.          0.          0.          0.20032787
 -0.          0.         -0.         -0.         -0.         -0.
  0.          0.07310976  0.          0.         -0.         -0.05617752
 -0.          0.         -0.         -0.         -0.         -0.
  0.          0.          0.04079857  0.12638553  0.          0.
  0.          0.07135376 -0.          0.          0.          0.
  0.          0.          0.         -0.          0.30518869  0.25780851
 -0.0848369   0.03917644]
['comma_oc', 'comma_ratio', 'comma_log', 'conj_ratio', 'conj_log', 'F', 'difficult_words', 'pos_num', 'pos_complexii
```

In [58]:

```
from sklearn.ensemble import RandomForestRegressor
rf = RandomForestRegressor()
rf.fit(M.loc[train_id,:],data.loc[train_id,'finals'])
```

Out[58]:

In [59]:

```
rf.feature_importances_
```

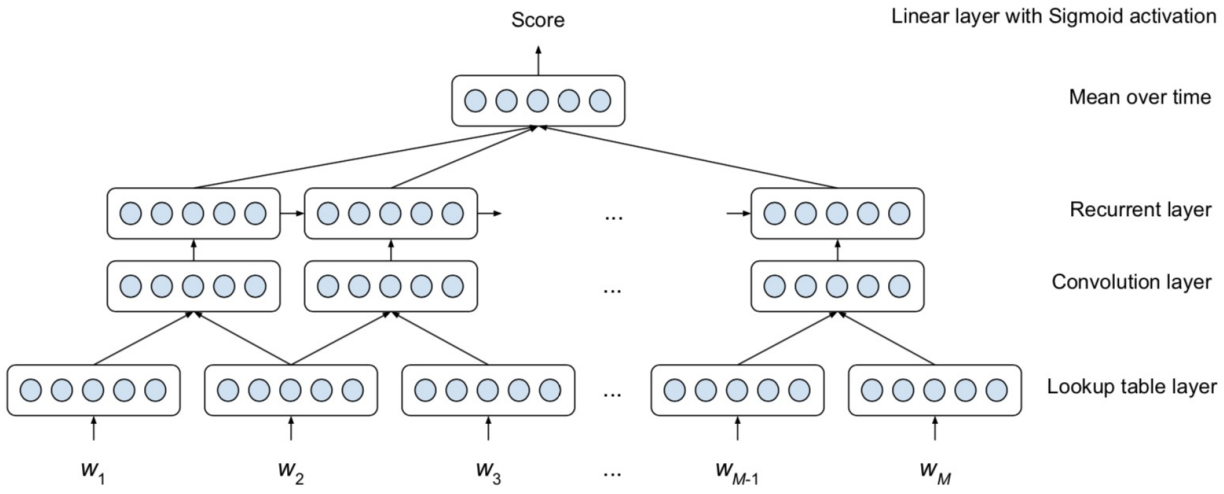
Out[59]:

关于深度学习的尝试

在深度学习处理这个任务的时候，大部分同学选择的是CNN或RNN（LSTM）。总体而言是做了如下的假设： 作为一个文章，文章的好坏不是孤立的词决定的，而一定是基于上下文的。而卷积神经网络可以通过卷积操作把上下文分成较小的语块（我认为类似n-grams模型），RNN可以有效地进行长期记忆，也是一种非常有效的处理序列问题（语言本身就是序列问题）的方式。而LSTM层，一定程度上解决了传统RNN的梯度消失的问题。当然门限RNN也能一定程度上解决这个问题，任务书给出的论文http://www.aclweb.org/old_anthology/D/D16/D16-1193.pdf (http://www.aclweb.org/old_anthology/D/D16/D16-1193.pdf) 中介绍了LSTM的效果优于传统RNN和门限RNN，我也就没有尝试传统的方法。

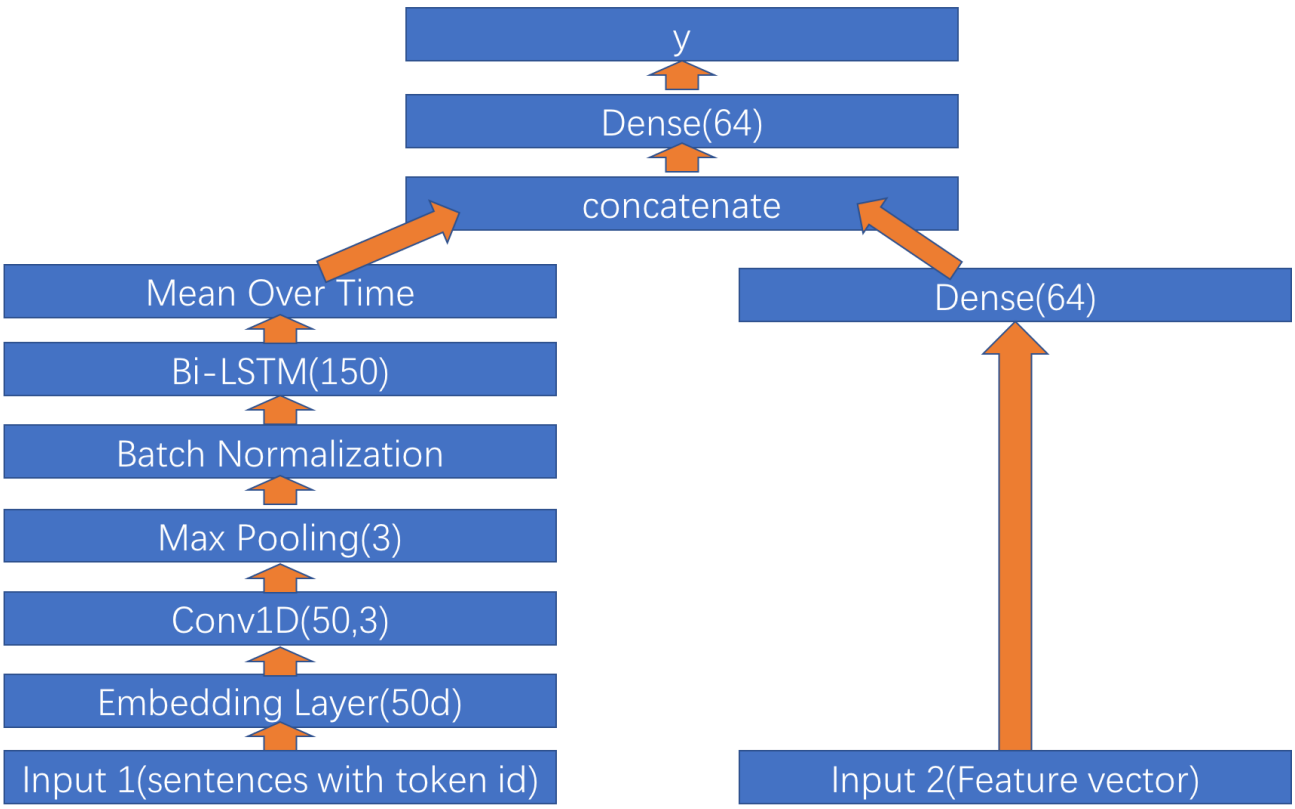
论文中给出了如下的搭建模型的方式。

目录



根据论文中的参数选择，搭建了几乎一模一样的神经网络，遇到了很多问题。比如训练实在太慢、收敛太慢等等等等。通过查阅了非常多的资料，了解到了一些训练神经网络的技巧。比如加入池化层（采样）、BatchNormalization（在传统机器学习中会有归一化的想法，但是在深度学习的时候完全忘掉了），最后调整了lstm层的神经元个数到80个，才能在平台上跑起来。最终得到的结果和一开始寻找特征的结果类似。0.83左右。

张庆恒助教在聊天中告诉了我，模型事实上是可以有两个输入的，一边是卷积和lstm,另一边可以和传统的特征工程得到的特征向量放进来，最终合并在一起送往全连接层输出即可。我突然想起来了上周大佬分享的双LSTM解决相似度问题的网络，照猫画虎加上查文档终于把模型建立起来了。我的模型大致如下：



In [ ]:

```
early_stopping = EarlyStopping(monitor='val_loss', patience=200)
model_checkpoint = ModelCheckpoint('cpbl.model', save_best_only=True)
def get_plus_model():
    sequence_input = Input(shape=(MAX_SEQUENCE_LENGTH,), dtype='int32')
    embedding_layer = Embedding(num_words,
                                EMBEDDING_DIM,
                                weights=[embedding_matrix],
                                input_length=MAX_SEQUENCE_LENGTH,
                                trainable=False
```

```
                                )

    embedded_sequences = embedding_layer(sequence_input)


    feature_input = Input(shape=(feature_num,),dtype='float32')
    y = Dense(64, activation='relu',kernel_initializer='random_normal')(feature_input)
    x = Conv1D(50, 3, activation='relu',kernel_initializer='random_normal')(embedded_sequences)
    x = MaxPooling1D(3)(x)
    x = BatchNormalization()(x)
    x = Bidirectional(LSTM(150,dropout=0.5,recurrent_dropout=0.5,
                           return_sequences=True,
                           implementation=1,kernel_initializer='random_normal'))(x)
    x = Dropout(0.5)(x)
#     mean over time
    x = Lambda(lambda x: K.mean(x, axis=1))(x)
    merge = concatenate([x,y])
    merge = Dense(64, activation='relu',kernel_initializer='random_normal')(merge)
    merge = Dropout(0.2)(merge)

    preds = Dense(1, activation='relu',kernel_initializer='random_normal')(merge)

    model = Model(inputs=[sequence_input,feature_input],outputs=preds)
    model.compile(loss='mse',
                  optimizer='nadam',
                  metrics=['acc'])
    print(model.summary())
    return model

model1 = get_plus_model()
model1.fit([x_train,feature_train], y_train,
           batch_size=32,
           epochs=1000,
           validation_data=([x_val,feature_val], y_val), callbacks=[early_stopping, model_checkpoint])
```

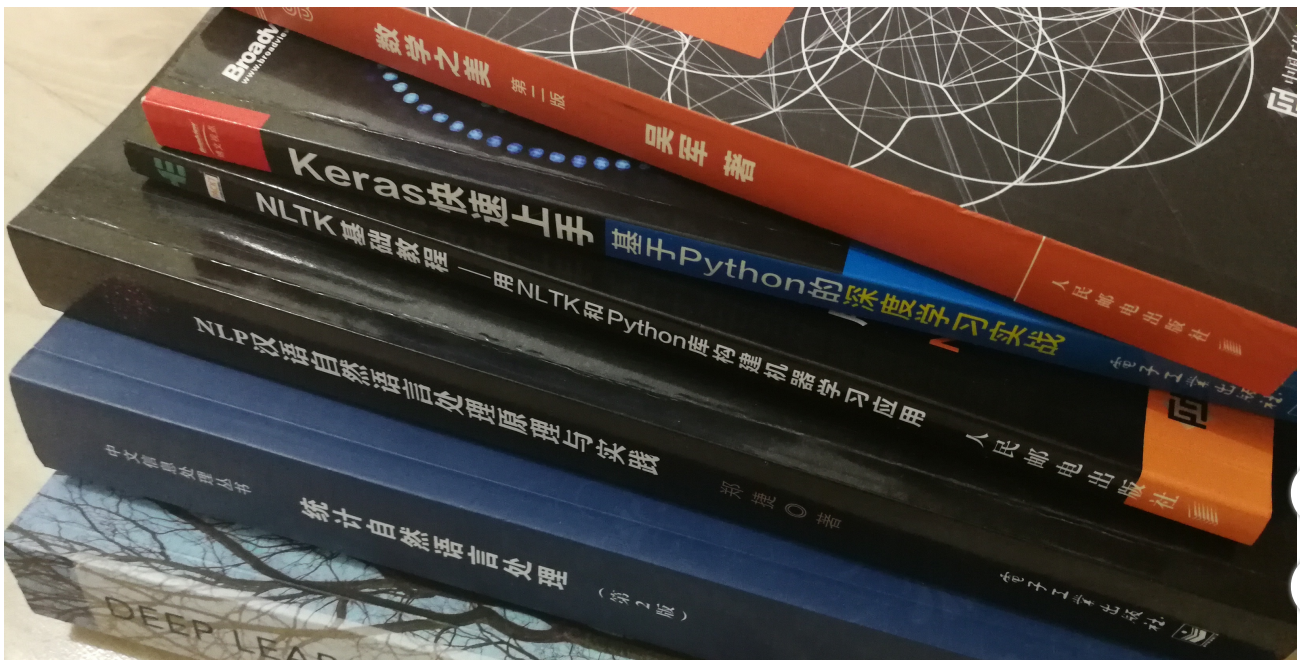
最后模型线上的分数大概在0.86，也并没有什么特别的提升。

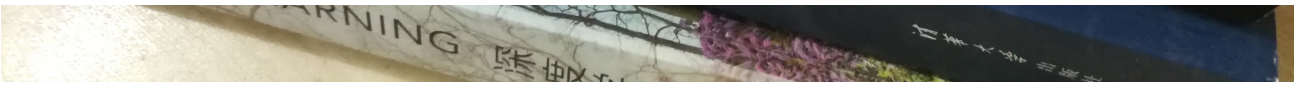
训练营收获

我从全职转岗到教师岗这半年来，杂七杂八学了很多东西。在八九月份的时候，大数据很热，看了一些关于数据挖掘的视频，了解了一些简单的机器学习的算法。当时的确有转行的想法，但是的确也因为学历等等问题搁置了。后来九月开始就开始学习Web相关的知识，觉得很多工作能做个小网站来解决是非常有意思，开始学Flask，然后又掉到前端的坑，写了一段时间的JS，觉得Python不是在哪都能解决问题，但是只要有电脑一定会有浏览器，能在浏览器编程一定听起来很装X。反正也没有压力，就乱七八糟学了一些东西。

这次参加训练营，选择NLP的原因是我自认为能力有限，自己以后的工作基本上很难接触到很大规模的数据，也很难需要处理图像。但是只要是人就要说话，只要是工作就要写周报，只要是企业就要有对外宣传的部门。作为一个老师，我总是想能不能自动地看学生的考试总结呢？那么多孩子，每个人都面临高考的巨大压力，能不能自动提前发现哪个孩子最容易出现问题呢？作为一个前培训师、前学科人，总是不自觉地想了解老师的周报能否反应这个老师最近一段时间的工作状态？能不能通过兄弟机构公众号的文章、教育方面的最新新闻得到最有用的业务信息呢？我承认不在其位不谋其政，但是总是有点不甘心。

我本以为我看过一点数据挖掘的视频，写了半年的Python，入手nlp也不会特别困难，但是来了之后发现自己差的太多了！一开始大家讨论的东西我完全不懂，就上网查，买书看。慢慢的就能插上话啦，在群里我话实在太多了， 请各位小伙伴多包涵哈！





目录

这两周买了不少书，虽然还没咋动过……接下来的半年有活干啦！

我觉得训练营的内容安排是非常适合新手的，但是可能需要加入一部分机器学习的最基础的东西，比如训练集和测试集的分配之类。在我看来，这两周的安排是这样的：

Day1:了解NLP是什么，要解决什么问题，解决NLP问题的一些基本思想和文本预处理的手段； Day2:了解如何把语言转化成计算机能理解的数据，以供模型

W2-138王泽群-NLP-Noob

Fork 点赞

Day1:了解语法在NLP中的应用，学习一些更有效的库来解决问题； Day2:说实话我不是很懂……我在这一天做的是Day3的内容； Day3:学习最简单的深度学习框架（API），了解深度学习的一些基础知识（我仅仅做到了知道词什么意思，能改改别人的代码跑起来……） Day4~Day5:继续提升阅读论文的能力，了解他人的实现，减少搭建框架和特征工程的时间。

> 说实话我在本科的时候基本没有读过论文，这两周读的论文快赶上我四年的总和了，也终于明白论文是给人看的，就算是Background这样的章节也不是没作用，给刚入门的同学很大的启发；

Day6:搭建模型。

收获非常多，从一个NLP小白（不过我学Python的时候做过词云，对着别人的代码训练过词向量（虽然我也不知道词向量是什么），没了……），到能哗啦哗啦给别人说一堆NLP的知识，我自己内心是非常激动的，感觉这两周没日没夜昏天黑地是值得的，也能自己去做一些简单的任务了，算是完成了70%来训练营前想要做的事情（毕竟我还不能自动看学生考试反馈……） 最后感谢集团，感谢TAIL AIlab的鼓励和鼓励奖，感谢辅导员和志愿者的汗水和付出，感谢科赛平台。

更感谢的是各位同学们对待集团AI Lab训练营的热情，你们的热情给了我前进的动力！感谢两位助教无私的分享，感谢小伙伴宋文泰的帮助。

总之，收获颇丰！

```
In [ ]:
```

目录

目录

目录

目

录

和鲸社区公众号

工具 内容 实战

K-Lab 项目 比赛

数据集 众包

专栏

专区

商务合作

© 和鲸HeyWhale 沪ICP备14038218号-1 沪公网安备 31010402003637号