目录

W2-122-陈文帆

1 690 5

TAIL CAMP WEEK2

iamkissg LV2 2

Fork 点赞

内容 数据集 Fork记录 5 评论

内容

版本 1 2018-02-09 23:57

TAIL CAMP NLP Week 2 - Automated Essay Scoring

目录

TEAM：要没时间了（陈文帆，李碧涵）

目录

1. 项目简介
2. 思路与实现（随代码讲解）
   A. 端到端的深度学习方法
   B. 人工提取特征的传统机器学习方法
3. 结果与结论
4. 总结

1. 项目简介

本周的项目是“手写作文自动评分”，目标是在教师培训或类似场景中，对手写内容自动评分，减轻教师负担，提高批改效率。

随着对相关背景的了解，我才知道，原来托福、雅思等考试早已实现了评分自动化。因此，这是一个具有现实意义的有趣项目。

> 说实话，在看论文的过程中，我确实萌生了向初中英语老师要英语作文素材，给他老人家写自动评分系统的想法。

使用的数据主要由

1. 英文作文
2. 两个评分人的评分以及总分

组成，而训练的目标就是通过对训练数据的学习，实现对给定作文的自动评总分。

2. 思路与实现

在训练营给的学习材料之外，我还查阅了自动作文评分，`AES`的相关论文，涉及端到端的深度学习方法与人工提取特征的传统机器学习方法。我分别实现了这两种方法。

我本身是按照先**深度学习方法**后**传统方法**的顺序做的项目，但后期又将传统方法提取得到的特征输入到深度神经网络中，因此以下将首先介绍人工提取特征的传统机器学习方法。

2.1 人工提取特征的传统机器学习方法

首先导入必要的库和 NLTK 数据包。

需要声明的是，`enchant (PyEnchant)`依赖于额外的 C++ 库 Enchant，`language_check`则依赖于 Language Tool 提供的 Java 库。因此，基于这两个库的特征提取在本地完成。

In [59]:

```
from collections import Counter
from functools import partial

import nltk
import enchant
import language_check
import numpy as np
import pandas as pd
import seaborn as sns
import xgboost as xgb
import lightgbm as lgb
import scipy.stats as stats
import keras.layers as klayers
import matplotlib.pyplot as plt
from nltk import ngrams
from nltk.stem import WordNetLemmatizer
from nltk.corpus import stopwords
from nltk.sentiment.vader import SentimentIntensityAnalyzer
from keras.preprocessing.text import text_to_word_sequence
from keras.preprocessing.sequence import pad_sequences
from keras.layers import Dense, LSTM, Input, Embedding, GlobalAveragePooling1D, Concatenate, Activation, Lambda, B
from keras.models import Model
from keras import backend as K
from keras.engine.topology import Layer, InputSpec
from keras.optimizers import Adam
from keras.callbacks import EarlyStopping
from keras import regularizers
from keras import initializers
from scipy.stats.stats import pearsonr
```

```
from textblob import TextBlob
from textstat.textstat import textstat
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.svm import SVR
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.linear_model import Lasso
from mlxtend.regressor import StackingRegressor

%matplotlib inline
```

In [3]:

```
nltk.download(['wordnet', "stopwords", "tagsets", "averaged_perceptron_tagger", "punkt", "vader_lexicon"])

stopwords_eng = stopwords.words("english")
```

```
[nltk_data] Downloading package wordnet to /home/kissg/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
[nltk_data] Downloading package stopwords to /home/kissg/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package tagsets to /home/kissg/nltk_data...
[nltk_data]   Package tagsets is already up-to-date!
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     /home/kissg/nltk_data...
[nltk_data]   Package averaged_perceptron_tagger is already up-to-
[nltk_data]       date!
[nltk_data] Downloading package punkt to /home/kissg/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package vader_lexicon to
[nltk_data]     /home/kissg/nltk_data...
[nltk_data]   Package vader_lexicon is already up-to-date!
```

按照惯例，读入数据之后，首先对数据进行必要的简单分析，包括

1. rater1 的给分分布
2. rater2 的给分分布
3. 总分的分布
4. essay 的长度分布

从图中可以看出，rater1 和 rater2 的给分分布基本一致，只有少量出入。一个有意思的点是，不存在总分是 3 的情况，这表明 rater1 与 rater2 没有给出过 1 分和 2 分的组合，进一步证实了两者给分的相对一致。

essay 的长度（包含的单词数）分布服从正太分布。从统计数据中也可以看出，80% 的 essay 长度在 464 个单词以内。由于最长的 essay 长达 783 个单词，考虑到序列过长将影响后文深度学习方法的性能，此处留个心，可考虑对 essay 进行截断。

In [4]:

```
df_train = pd.read_csv("Tal_SenSco_train_data.txt", sep="\t", names=["id", "essay", "rater1", "rater2", "score"])
df_test = pd.read_csv("Tal_SenSco_test_data.txt", sep="\t", names=["id", "essay"])
```

In [5]:

```
df_train["wsequence"] = df_train["essay"].apply(text_to_word_sequence)
df_test["wsequence"] = df_test["essay"].apply(text_to_word_sequence)
```

In [51]:

```
df_train[["rater1", "rater2", "score", "num_tokens"]].describe(percentiles=[0.25, 0.5, 0.75, 0.8, 0.9])
```

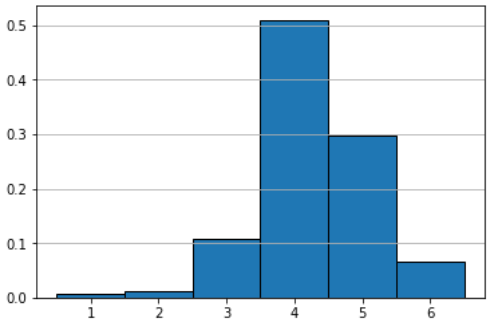
Out[51]:

	rater1	rater2	score	num_tokens
count	1200.000000	1200.000000	1200.000000	1200.000000
mean	4.280000	4.273333	8.553333	370.963333
std	0.836969	0.820071	1.531593	121.342616
min	1.000000	1.000000	2.000000	8.000000
25%	4.000000	4.000000	8.000000	293.750000
50%	4.000000	4.000000	8.000000	368.000000
75%	5.000000	5.000000	10.000000	445.000000
80%	5.000000	5.000000	10.000000	464.000000
90%	5.000000	5.000000	10.000000	526.100000
max	6.000000	6.000000	12.000000	783.000000

max	6.000000	6.000000	12.000000	783.000000

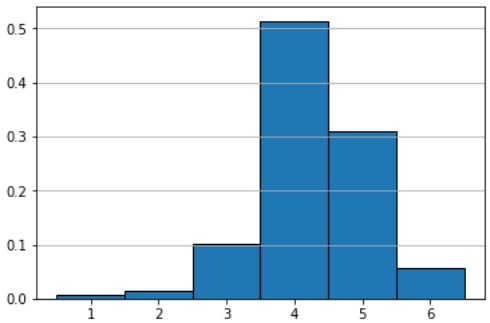
In [10]:

```
plt.hist(df_train["rater1"], normed=True, bins=range(1, 8), align="left", edgecolor='black')
plt.grid(axis="y")
plt.show()
```



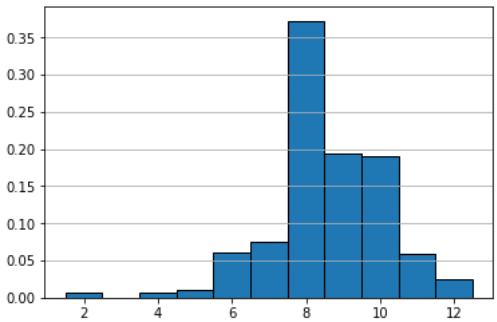
In [11]:

```
plt.hist(df_train["rater2"], normed=True, bins=range(1, 8), align="left", edgecolor='black')
plt.grid(axis="y")
plt.show()
```



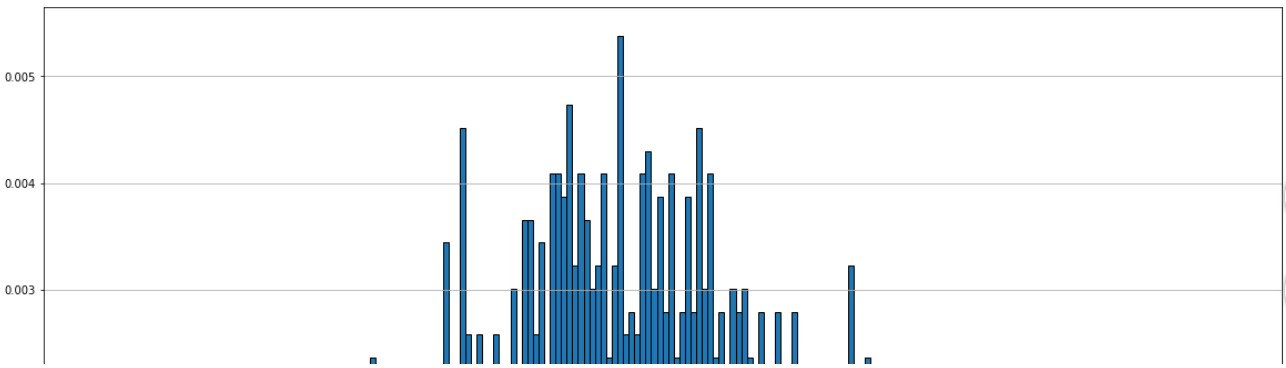
In [12]:

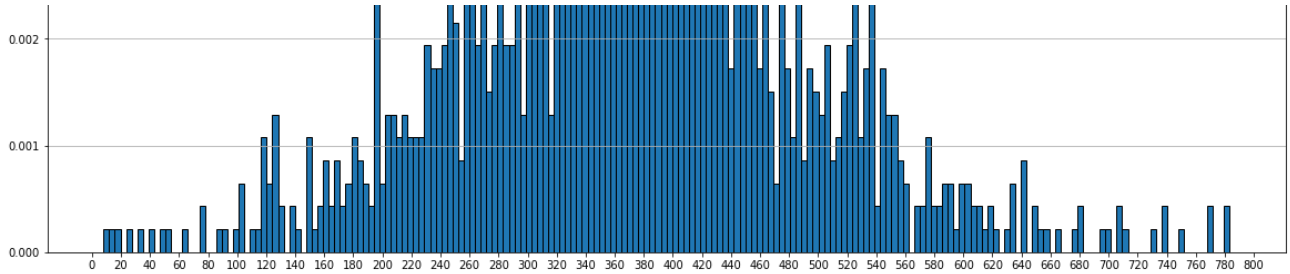
```
plt.hist(df_train["score"], bins=range(2, 14), normed=True, align="left",  edgecolor='black')
plt.grid(axis="y")
plt.show()
```



In [13]:

```
plt.figure(figsize=(20,10))
plt.hist(df_train["num_tokens"], bins=200, normed=True, edgecolor="black")
plt.xticks(np.arange(0, 801, 20))
plt.grid(axis="y")
```





Task-Independent Features for Automated Essay Grading (https://www.researchgate.net/publication/278383803_Task-Independent_Features_for_Automated_Essay_Grading) 一文对人工提取特征的传统 AES 方法进行了研究，考察了**任务独立**的通用型特征的有效性。

以下特征多来自此文，部分特征来自其他论文，部分来自队友，剩下的纯属灵光乍现。

1. Length Features

基于长度的特征包括：

1. 单词数；
2. 句子数；
3. 平均句子长度，以单词数作为考量；
4. 平均单词长度，以字符数作为考量；
5. 长单词数，字符数超过 7 的单词记为长单词；
6. 停词数；
7. 字符数。

有一些特征比较直观地反映了学生的英语水平，比如长单词数，可以认为与作文分数有着密切关系，具体见下文的相关性分析。

In [6]:

```
# text length by counting all tokens
df_train["num_tokens"] = df_train["wsequence"].apply(len)
df_test["num_tokens"] = df_test["wsequence"].apply(len)

# sentences in essay
df_train["num_sentences"] = df_train["essay"].apply(lambda t: textstat.sentence_count(t))
df_test["num_sentences"] = df_test["essay"].apply(lambda t: textstat.sentence_count(t))

# average sentence length in words
df_train["average_sent_len"] = df_train["num_tokens"].values / df_train["num_sentences"].values
df_test["average_sent_len"] = df_test["num_tokens"].values / df_test["num_sentences"].values

# average word length in characters
df_train["average_word_len"] = df_train["wsequence"].apply(lambda t: np.array([len(w) for w in t]).mean())
df_test["average_word_len"] = df_test["wsequence"].apply(lambda t: np.array([len(w) for w in t]).mean())

# number of long words
df_train["num_long_words"] = df_train["wsequence"].apply(lambda t: len([w for w in t if len(w) > 7]))
df_test["num_long_words"] = df_test["wsequence"].apply(lambda t: len([w for w in t if len(w) > 7]))

# number of stopwords
df_train["num_stopwords"] = df_train["wsequence"].apply(lambda t: len([w for w in t if w in stopwords_eng]))
df_test["num_stopwords"] = df_train["wsequence"].apply(lambda t: len([w for w in t if w in stopwords_eng]))

# number of characters
df_train["num_characters"] = df_train["essay"].apply(len)
df_test["num_characters"] = df_test["essay"].apply(len)
```

2. Occurence Features

此类特征包括：

1. 逗号数；
2. 引号数；
3. 叹号数。

逗号数一定程度上反应了句子内部的复杂程度；引号数，我猜是反映了学生旁征博引的能力；叹号数，我猜是反映了精彩句子的比例。

In [7]:

```
# occurence of commas
df_train["num_commas"] = df_train["essay"].apply(lambda t: t.count(","))
df_test["num_commas"] = df_test["essay"].apply(lambda t: t.count(","))

# occurence of quotations
df_train["num_quotations"] = df_train["essay"].apply(lambda t: t.count("\""))
```

```
df_train["num_quotations"] = df_train["essay"].apply(lambda t: t.count("\""))
df_test["num_quotations"] = df_test["essay"].apply(lambda t: t.count("\""))

# occurence of num exclamation marks
df_train["num_exclamation_marks"] = df_train["essay"].apply(lambda t: t.count("!"))
df_test["num_exclamation_marks"] = df_test["essay"].apply(lambda t: t.count("!"))
```

目录

4. Style Features

风格特征包括:

1. F 值;
2. 不同形式的单词总数与不同单词的总数比，简称 ttr;
3. 平均词频

F 值考量的是 Formal 类单词与 Contextual 类单词的关系；ttr 反应了学生掌握的词汇的丰富程度；平均词频则是越低，说明学生的语言功底越好。

In [8]:

```
wordnet_lemmatizer = WordNetLemmatizer()

def calculate_F_score(tokens):
    num_formal = 0  # number of words belong to Formal categories
    num_contextual = 0  # Contextual categories
    for pt in nltk.pos_tag(tokens):
        if pt[1] in {"NN", "NNP", "NNPS", "NNS", "JJ", "JJR", "JJS", "DT", "PDT", "WDT", "IN", "TO"}:  # Nouns, ad
            num_formal += 1
        elif pt[1] in {"RB", "RBR", "RBS", "WRB", "PRP", "PRP$", "WP", "WP$", "VB", "VBD", "VBG", "VBN", "VBP", "V
            num_contextual += 1
    return (num_formal - num_contextual + 100) / 2


def calculate_type_token_ratio(tokens):

    type_counter = Counter(tokens)
    token_counter = Counter([wordnet_lemmatizer.lemmatize(t) for t in tokens])

    return (len(type_counter) / len(token_counter))


def calculate_avg_word_freq(tokens):
    token_counter = Counter([wordnet_lemmatizer.lemmatize(t) for t in tokens])  # the higher the better
    return sum(token_counter.values()) / len(token_counter)

df_train["f_score"] = df_train["wsequence"].apply(calculate_F_score)
df_test["f_score"] = df_test["wsequence"].apply(calculate_F_score)

df_train["type_token_ratio"] = df_train["wsequence"].apply(calculate_type_token_ratio)
df_test["type_token_ratio"] = df_test["wsequence"].apply(calculate_type_token_ratio)

df_train["avg_word_freq"] = df_train["wsequence"].apply(calculate_avg_word_freq)  # the lower the better
df_test["avg_word_freq"] = df_test["wsequence"].apply(calculate_avg_word_freq)
```

5. Cohesion Features

cohesion features 通过统计连词数，来估计作文的 cohesion。

In [9]:

```
connectives = {
        "also", "besides", "futhermore", "too", "moreover", "in addition",
        "finally", "of equal importance", "equally important", "from here on", "from now on",
        "hence", "next","then", "to begin with", "last of all", "after", "before", "as soon as",
        "in the end", "gradually", "yet", "however", "but", "rather", "on the other hand",
        "on the contrary", "nevertheless", "in spite of", "in contrast to this", "although",
        "therefore", "consequently", "in summary", "to sum up", "briefly", "in short",
        "in conclusion", "as you can see", "in fact", "of course", "to be sure", "naturally",
        "obviously", "no doubt", "at least", "anyhow", "for example", "for instance",
        "indeed", "specially", "in particular", "unfortunately", "afterward", "later",
        "at last", "now", "subsequently", "in the meantime", "meanwhile", "otherwise",
        "accordding to", "so that", "so then", "unless", "on condition", "in case", "but that",
        "suppose", "in order", "first of all", "at the same time", "wherever", "let",
        "wherefore", "ultimately", "presently", "at length", "on the following day",
        "the next", "after a short time", "soon", "lastly", "in especial", "probably", "it is certoin",
        "to be sure", "perhaps", "on the whole", ""
    }
def calculate_cohesion(text):
    return sum([text.count(w) for w in connectives])

df_train["cohesion"] = df_train["essay"].apply(str.lower).apply(calculate_cohesion) / df_train["num_tokens"]
df_test["cohesion"] = df_test["essay"].apply(str.lower).apply(calculate_cohesion) / df_test["num_tokens"]
```

目录

6. Coherence Features

以 n-gram 的重叠度来估计相邻句子的相似性，我选择了

1. 1-gram 的重叠度
2. 2-gram 的重叠度
3. 3-gram 的重叠度

4-gram 及以上极少有重叠的了，因此不予考虑。

实际也可以用第一周的项目来句子相似性预测来检测相邻句子的相似性，只是时间有限，我没来得及将前一个项目适配到该项目中来。

In [10]:

```
def calculate_ngram_overlap(text, n):
    zen = TextBlob(text=text)
    sentences = [text_to_word_sequence(s.raw ) for s in zen.sentences]

    similarity = 0
    p_ngram = set(ngrams(sentences[0], n))

    for prev, nxt in zip(sentences[: -1], sentences[1:]):
        n_ngram = set(ngrams(nxt, n))
        similarity += len(n_ngram.intersection(p_ngram)) / (len(prev) + len(nxt))
        p_ngram = n_ngram

    return similarity / len(zen.sentences)

one_gram_overlap = partial(calculate_ngram_overlap, n=1)
two_gram_overlap = partial(calculate_ngram_overlap, n=2)
three_gram_overlap = partial(calculate_ngram_overlap, n=3)

df_train["one_gram_overlap"] = df_train["essay"].apply(one_gram_overlap)
df_test["one_gram_overlap"] = df_test["essay"].apply(one_gram_overlap)

df_train["two_gram_overlap"] = df_train["essay"].apply(two_gram_overlap)
df_test["two_gram_overlap"] = df_test["essay"].apply(two_gram_overlap)

df_train["three_gram_overlap"] = df_train["essay"].apply(three_gram_overlap)
df_test["three_gram_overlap"] = df_test["essay"].apply(three_gram_overlap)
```

7. Error Features

错误特征包括：

1. 单词拼写错误
2. 语法错误

这两个特征的提取就使用了本文开头介绍的额外的库。

根据我英文写作的经历，老师在评分时会考量这两个错误率；但论文中表示它们对评分的影响非常有限。此处保留这两个特征。

从稍后的相关性分析中我们可以看到，拼写错误与评分的相关性高于语法错误，但是居然是正相关！我怀疑这两个特征的数量与作文长度成正相关，而作文长度与评分又成正相关，因此错误与评分也成正相关关系。（详见下文的相关性矩阵）

In [11]:

```
d = enchant.Dict("en_US")
language_tool = language_check.LanguageTool('en-US')

def count_spelling_error(tokens):
    return len([t for t in tokens if not d.check(t)])


def count_grammar_error(text):
    matches = language_tool.check(text)
    return len(matches)


df_train["spelling_error"] = df_train["wsequence"].apply(count_spelling_error)
df_test["spelling_error"] = df_test["wsequence"].apply(count_spelling_error)

df_train["grammar_error"] = df_train["essay"].apply(count_grammar_error)
df_test["grammar_error"] = df_test["essay"].apply(count_grammar_error)
```

8. Readability Features

以下特征使用了学习材料中提到的 textstat 库，考量了作文的可读性。

In [12]:

```
df_train['syllable_count'] = df_train.essay.apply(textstat.syllable_count)
df_train['flesch_reading_ease'] = df_train.essay.apply(textstat.flesch_reading_ease)
df_train['flesch_kincaid_grade'] = df_train.essay.apply(textstat.flesch_kincaid_grade)
df_train['gunning_fog'] = df_train.essay.apply(textstat.gunning_fog)
df_train['smog_index'] = df_train.essay.apply(textstat.smog_index)
df_train['automated_readability_index'] = df_train.essay.apply(textstat.automated_readability_index)
df_train['coleman_liau_index'] = df_train.essay.apply(textstat.coleman_liau_index)
df_train['linsear_write_formula'] = df_train.essay.apply(textstat.linsear_write_formula)
df_train['dale_chall_readability_score'] = df_train.essay.apply(textstat.dale_chall_readability_score)
df_train['difficult_words'] = df_train.essay.apply(textstat.difficult_words)  # 困难单词数

df_test['syllable_count'] = df_test.essay.apply(textstat.syllable_count)
df_test['flesch_reading_ease'] = df_test.essay.apply(textstat.flesch_reading_ease)
df_test['flesch_kincaid_grade'] = df_test.essay.apply(textstat.flesch_kincaid_grade)
df_test['gunning_fog'] = df_test.essay.apply(textstat.gunning_fog)
df_test['smog_index'] = df_test.essay.apply(textstat.smog_index)
df_test['automated_readability_index'] = df_test.essay.apply(textstat.automated_readability_index)
df_test['coleman_liau_index'] = df_test.essay.apply(textstat.coleman_liau_index)
df_test['linsear_write_formula'] = df_test.essay.apply(textstat.linsear_write_formula)
df_test['dale_chall_readability_score'] = df_test.essay.apply(textstat.dale_chall_readability_score)
df_test['difficult_words'] = df_test.essay.apply(textstat.difficult_words)  # 困难单词数
```

9. Others

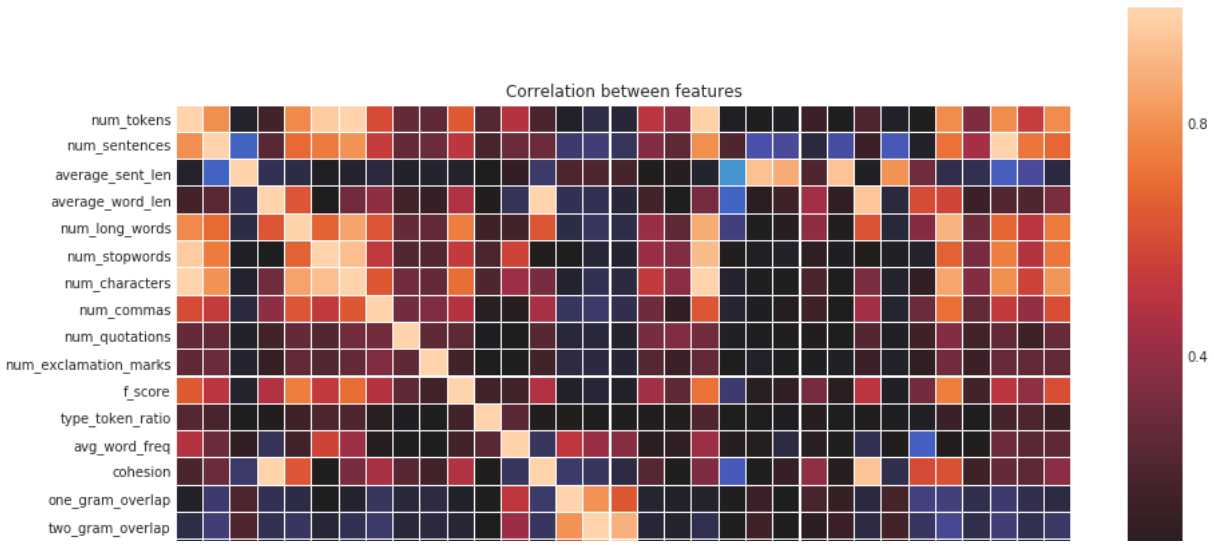
以下特征来自队友，通过句子的极性来反映作文的极性。虽然有些不明所以，但从相关性分析中可以看到，这 3 个特征与评分的相关性还挺高的。

In [13]:

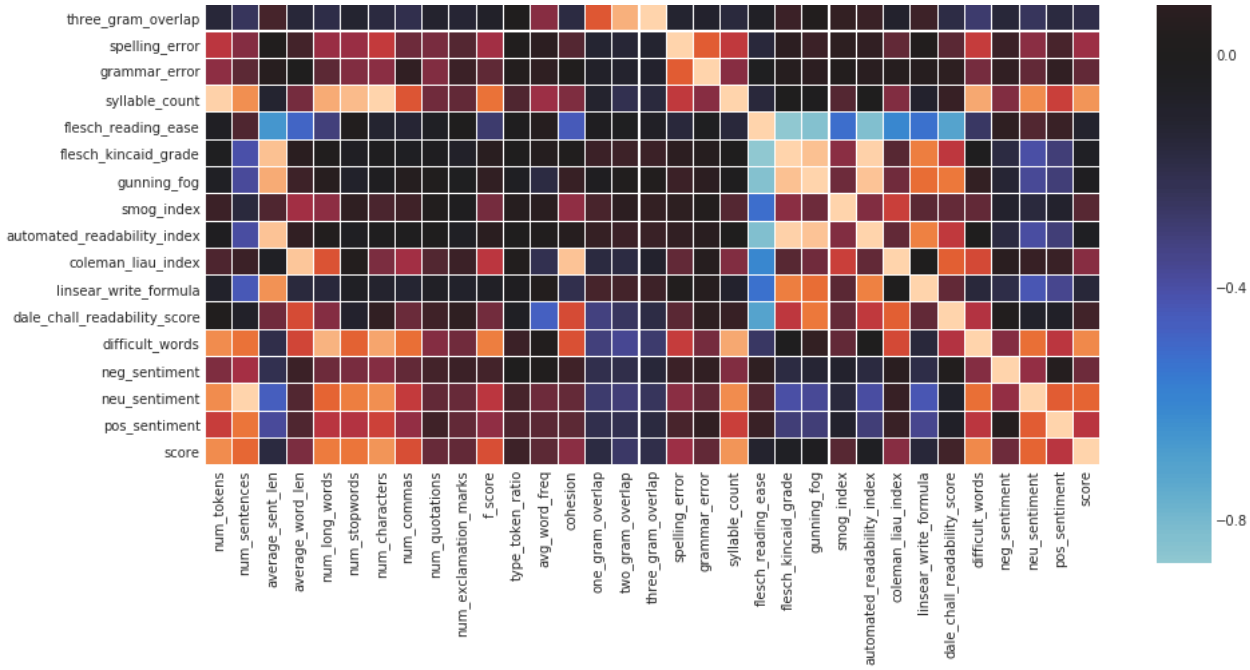
```
sid = SentimentIntensityAnalyzer()

train_sents_polar = df_train['essay'].apply(lambda x: [sid.polarity_scores(s.raw) for s in TextBlob(x).sentences])
test_sents_polar = df_test['essay'].apply(lambda x: [sid.polarity_scores(s.raw) for s in TextBlob(x).sentences])

df_train['neg_sentiment'] = train_sents_polar.apply(lambda x: sum([v for item in x for k, v in item.items() if k =
df_train['neu_sentiment'] = train_sents_polar.apply(lambda x: sum([v for item in x for k, v in item.items() if k =
df_train['pos_sentiment'] = train_sents_polar.apply(lambda x: sum([v for item in x for k, v in item.items() if k =

df_test['neg_sentiment'] = test_sents_polar.apply(lambda x: sum([v for item in x for k, v in item.items() if k ==
df_test['neu_sentiment'] = test_sents_polar.apply(lambda x: sum([v for item in x for k, v in item.items() if k ==
df_test['pos_sentiment'] = test_sents_polar.apply(lambda x: sum([v for item in x for k, v in item.items() if k ==
```

In [52]:

```
plt.figure(figsize=(15, 15))

correlations = df_train[[
    'num_tokens', 'num_sentences', 'average_sent_len', 'average_word_len', 'num_long_words', 'num_stopwords', 'num
    'num_commas', 'num_quotations', 'num_exclamation_marks', 'f_score', 'type_token_ratio', 'avg_word_freq', 'cohe
    'one_gram_overlap', 'two_gram_overlap', 'three_gram_overlap', 'spelling_error', 'grammar_error', 'syllable_cou
    'flesch_reading_ease', 'flesch_kincaid_grade', 'gunning_fog', 'smog_index', 'automated_readability_index',
    'coleman_liau_index', 'linsear_write_formula', 'dale_chall_readability_score', 'difficult_words', 'neg_sentime
    'neu_sentiment', 'pos_sentiment', 'score']].corr()

mask = np.zeros_like(correlations)
mask[np.triu_indices_from(mask)] = True
with sns.axes_style("white"):
    ax = sns.heatmap(correlations, square=True, linewidths=0.1, center=0.0)
plt.title("Correlation between features")
plt.show()
```



目录



In [49]:

```
print(np.abs(correlations.loc[:, "score"]))
```

```
num_tokens                      0.794060
num_sentences                   0.690165
average_sent_len                0.163936
average_word_len                0.341625
num_long_words                  0.737200
num_stopwords                   0.729191
num_characters                  0.816107
num_commas                      0.609351
num_quotations                  0.289667
num_exclamation_marks           0.271486
f_score                         0.607514
type_token_ratio                0.151769
avg_word_freq                   0.252310
cohesion                        0.372869
one_gram_overlap                0.176364
two_gram_overlap                0.266445
three_gram_overlap              0.196764
spelling_error                  0.428537
grammar_error                   0.274009
syllable_count                  0.813792
flesch_reading_ease             0.105345
flesch_kincaid_grade            0.058573
gunning_fog                     0.028699
smog_index                      0.234122
automated_readability_index     0.043910
coleman_liau_index              0.368479
linsear_write_formula           0.142551
dale_chall_readability_score    0.169032
difficult_words                 0.780656
neg_sentiment                   0.300202
neu_sentiment                   0.681916
pos_sentiment                   0.497931
score                           1.000000
Name: score, dtype: float64
```

完成了繁重的人工特征提取任务，开始使用所有特征训练 Regressor，用的到 Regressors 包括：

1. XGBRegressor；
2. LGBMRegressor；
3. SVR；
4. RandomForestRegressor；
5. GradientBoostingRegressor；
6. StackingRegressor

对于 XGBRegressor 和 LGBMRegressor，经过简单的参数调整之后，设置超大的（1000）的迭代次数，并启用 EarlyStopping，然后根据早停的迭代数重新设置迭代次数，重新模型。

> 不知道这个技巧是否可取?

SVR、RandomForestRegressor 以及 GradientBoostingRegressor 使用 GridSearch 简单调整过参数。（代码注释保留）

同上一个项目的结果一样，Stacking 并没有取得明显的提升，相反，它的预测值的分布形状更加奇葩了。

根据验证集上的 Pearson 相关性，XGBRegressor 仍然是最优的 Regressor。

各 Regressors 在测试集上的预测值分布大致相近，与训练数据的总分分布保持一致，典型特征是：8 分最多；高分与低分都很少。（为了更好地观察预测值分布的形状，我使用了更大的 bins）

目录

In [17]:

```
training_data = df_train.drop(["id", "essay", "wsequence", "rater1", "rater2", "score"], axis=1)  # 不使用向量作为输入
test_data = df_test.drop(["id", "essay", "wsequence"], axis=1)
```

In [18]:

```
std = StandardScaler()

score = df_train["score"].values

# score_min = df_train["score"].min()
# score_range = df_train["score"].max() - score_min
# score = (df_train["score"].values - score_min) / score_range

train_std = std.fit_transform(training_data)
test_std = std.transform(test_data)

x_train, x_valid, y_train, y_valid = train_test_split(train_std, score, test_size=0.2, random_state=609)
```

In [20]:

```
xgbr_params = {
    "objective": "reg:linear",
#     "objective": "binary:logistic",
    "learning_rate": 0.04,  # 0.01 ~ 0.2
    "max_depth": 3,  # 3 ~ 10; 4 may be the best
    "seed": 609,
    "n_estimators": 135  # 预设 1000, 根据早停设置
}

xgbr_train = xgb.DMatrix(x_train, label=y_train)
xgbr_valid = xgb.DMatrix(x_valid, label=y_valid)

watchlist = [(xgbr_train, "train"), (xgbr_valid, "valid")]

xgbr = xgb.XGBRegressor(**xgbr_params)
# xgbc = xgb.XGBClassifier(**xgbr_params)
xgbr.fit(x_train, y_train, early_stopping_rounds=50, eval_metric="rmse", eval_set=[(x_valid, y_valid)], verbose=10
```

```
[0]     validation_0-rmse:7.83958
Will train until validation_0-rmse hasn't improved in 50 rounds.
[10]    validation_0-rmse:5.24197
[20]    validation_0-rmse:3.52764
[30]    validation_0-rmse:2.41535
[40]    validation_0-rmse:1.70302
[50]    validation_0-rmse:1.26324
[60]    validation_0-rmse:1.01069
[70]    validation_0-rmse:0.876322
[80]    validation_0-rmse:0.810201
[90]    validation_0-rmse:0.779706
[100]   validation_0-rmse:0.766719
[110]   validation_0-rmse:0.759545
[120]   validation_0-rmse:0.755512
[130]   validation_0-rmse:0.75403
[134]   validation_0-rmse:0.753601
```

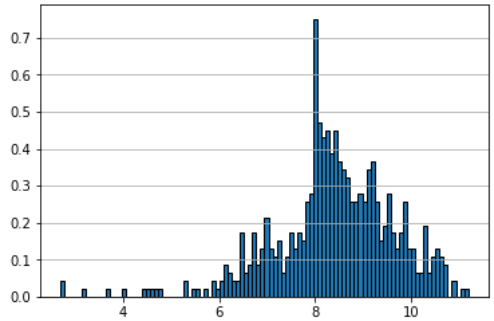
Out[20]:

```
XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
       colsample_bytree=1, gamma=0, learning_rate=0.04, max_delta_step=0,
       max_depth=3, min_child_weight=1, missing=None, n_estimators=135,
       n_jobs=1, nthread=None, objective='reg:linear', random_state=0,
       reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=609,
       silent=True, subsample=1)
```

In [24]:

```
xgbr_pred_valid = xgbr.predict(x_valid)
print(pearsonr(xgbr_pred_valid, y_valid))

xgbr_test_pred = xgbr.predict(test_std)
plt.hist(xgbr_test_pred, normed=True, bins=100, align="left", edgecolor='black')
plt.grid(axis="y")
plt.show()
```

```
(0.8827685242660881, 4.8298728737069106e-80)
```



In [23]:

```
lgbr_params = {
    "objective": "regression",
    "learning_rate": 0.02,
    "num_iteration": 367,  # 预设 1000, 根据早停设置
    "boosting_type": "gbdt",
    "num_leaves": 5,
    "max_bin": 30,
    "bagging_fraction": 1,
    "feature_fraction": 0.7,
    "bagging_seed": 9,
    "feature_fraction_seed": 9,
    "min_data_in_leaf": 16,
    "metric": "root_mean_squared_error",
}

lgbr = lgb.LGBMRegressor(**lgbr_params)

lgbr.fit(x_train, y_train, early_stopping_rounds=100, eval_set=[(x_valid, y_valid)], verbose=10)
```

```
/usr/local/lib/python3.5/dist-packages/lightgbm/engine.py:99: UserWarning: Found `num_iteration` in params. Will use
  warnings.warn("Found `{}` in params. Will use it instead of argument".format(alias))

Training until validation scores don't improve for 100 rounds.
[10]    valid_0's l2: 2.0222
[20]    valid_0's l2: 1.64526
[30]    valid_0's l2: 1.38413
[40]    valid_0's l2: 1.19584
[50]    valid_0's l2: 1.05759
[60]    valid_0's l2: 0.962008
[70]    valid_0's l2: 0.892056
[80]    valid_0's l2: 0.842234
[90]    valid_0's l2: 0.804572
[100]   valid_0's l2: 0.773876
[110]   valid_0's l2: 0.749898
[120]   valid_0's l2: 0.733675
[130]   valid_0's l2: 0.720628
[140]   valid_0's l2: 0.708952
[150]   valid_0's l2: 0.700958
[160]   valid_0's l2: 0.693798
[170]   valid_0's l2: 0.686275
[180]   valid_0's l2: 0.680797
[190]   valid_0's l2: 0.676145
[200]   valid_0's l2: 0.672371
[210]   valid_0's l2: 0.66807
[220]   valid_0's l2: 0.664746
[230]   valid_0's l2: 0.662906
[240]   valid_0's l2: 0.660327
[250]   valid_0's l2: 0.658492
[260]   valid_0's l2: 0.656792
[270]   valid_0's l2: 0.654307
[280]   valid_0's l2: 0.653587
[290]   valid_0's l2: 0.65324
[300]   valid_0's l2: 0.652476
[310]   valid_0's l2: 0.651782
[320]   valid_0's l2: 0.650853
[330]   valid_0's l2: 0.650666
[340]   valid_0's l2: 0.649593
[350]   valid_0's l2: 0.649432
[360]   valid_0's l2: 0.648845
Did not meet early stopping. Best iteration is:
[366]   valid_0's l2: 0.648463
```

Out[23]:

```
LGBMRegressor(bagging_fraction=1, bagging_seed=9, boosting_type='gbdt',
       class_weight=None, colsample_bytree=1.0, feature_fraction=0.7,
```

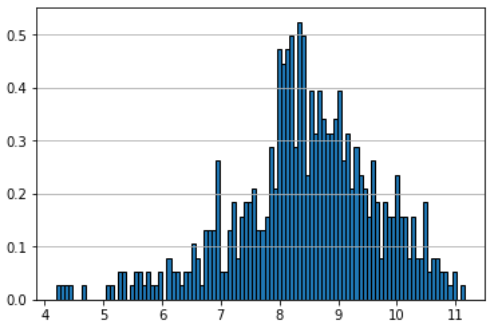
```
       class_weight=None, colsample_bytree=1.0, feature_fraction=0.7,
       feature_fraction_seed=9, learning_rate=0.02, max_bin=30,
       max_depth=-1, metric='root_mean_squared_error',
       min_child_samples=20, min_child_weight=0.001, min_data_in_leaf=16,
       min_split_gain=0.0, n_estimators=100, n_jobs=-1, num_iteration=367,
       num_leaves=5, objective='regression', random_state=None,
       reg_alpha=0.0, reg_lambda=0.0, silent=True, subsample=1.0,
       subsample_for_bin=200000, subsample_freq=1)
```

目录

In [25]:

```
lgbr_pred_valid = lgbr.predict(x_valid)
print(pearsonr(lgbr_pred_valid, y_valid))

lgbr_test_pred = lgbr.predict(test_std)
plt.hist(lgbr_test_pred, normed=True, bins=100, align="left", edgecolor='black')
plt.grid(axis="y")
plt.show()
```

```
(0.8647438923749231, 3.8641413822384445e-73)
```



In [26]:

```
# pipe_svr = Pipeline([
#     ("svr", SVR())
# ])

# # 初步确定比较优的参数范围是: (kernel 仅选择了 linear 与 rbf; 初始范围 [0.0001, 10])
# # 'svr__gamma': 0.01, 'svr__kernel': 'rbf', 'svr__C': 10.0
# C_range = [5.0, 10.0, 20.0, 30.0, 40.0, 50.0]  # 10, 20
# gamma_range = [0.0001, 0.001, 0.0015, 0.002, 0.0025]  # 0.002, 0.001
# param_grid = [
#     {
#         "svr__C": C_range,
#         "svr__gamma": gamma_range,
#         "svr__kernel": ["rbf"]
#     }
# ]
# gs = GridSearchCV(estimator=pipe_svr,
#                   param_grid=param_grid,
#                   # scoring="mse",
#                   cv=10,
#                   verbose=2,
#                   n_jobs=-1)
# # gs = gs.fit(train_std, score)
# gs = gs.fit(x_train, y_train)

# print(gs.best_score_)
# print(gs.best_params_)
# svr = gs.best_estimator_

svr = SVR("rbf", C=20, gamma=0.001)
svr.fit(x_train, y_train)
```

Out[26]:

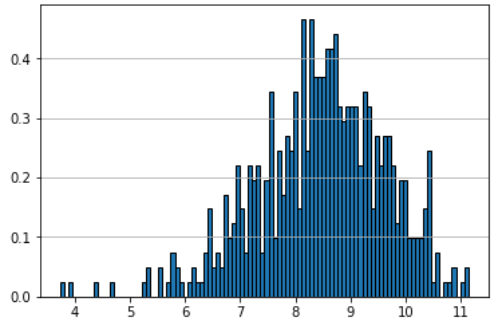
```
SVR(C=20, cache_size=200, coef0=0.0, degree=3, epsilon=0.1, gamma=0.001,
  kernel='rbf', max_iter=-1, shrinking=True, tol=0.001, verbose=False)
```

In [27]:

```
svr_pred_valid = svr.predict(x_valid)
print(pearsonr(svr_pred_valid, y_valid))

svr_test_pred = svr.predict(test_std)
plt.hist(svr_test_pred, normed=True, bins=100, align="left", edgecolor='black')
plt.grid(axis="y")
plt.show()
```

```
(0.8644180831390261, 5.0411030246729963e-73)
```

目录



In [28]:

```
# pipe_rfr = Pipeline([
#     ("rfr", RandomForestRegressor(random_state=609))
# ])
# num_estimator_range = [30, 40, 50, 60, 70, 80]  # 60
# depth_range = [2, 3, 4, 5,]  # 4
# param_grid = [
#     {
#         "rfr__max_depth": depth_range,
#         "rfr__n_estimators": num_estimator_range,
#     }
# ]
# gs = GridSearchCV(estimator=pipe_rfr,
#                   param_grid=param_grid,
#                   cv=10,
#                   verbose=2,
#                   n_jobs=-1)
# gs = gs.fit(x_train, y_train)

# print(gs.best_score_)
# print(gs.best_params_)
# rfr = gs.best_estimator_

rfr = RandomForestRegressor(max_depth=4, n_estimators=60)
rfr.fit(x_train, y_train)
```

Out[28]:

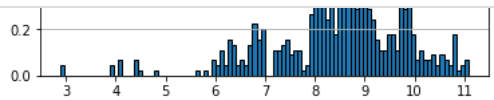
```
RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=4,
           max_features='auto', max_leaf_nodes=None,
           min_impurity_decrease=0.0, min_impurity_split=None,
           min_samples_leaf=1, min_samples_split=2,
           min_weight_fraction_leaf=0.0, n_estimators=60, n_jobs=1,
           oob_score=False, random_state=None, verbose=0, warm_start=False)
```

In [29]:

```
rfr_pred_valid = rfr.predict(x_valid)
print(pearsonr(rfr_pred_valid, y_valid))

rfr_test_pred = rfr.predict(test_std)
plt.hist(rfr_test_pred, normed=True, bins=100, align="left", edgecolor='black')
plt.grid(axis="y")
plt.show()
```

```
(0.8732556723766729, 2.8776460083619996e-76)
```

W2-122-陈文帆 Fork 点赞



In [30]:

```
# pipe_gbr = Pipeline([
#     ("gbr", GradientBoostingRegressor(random_state=609))
# ])
# num_estimator_range = [50, 60, 70, 80, 100]  # 60
# lr_range = [0.0001, 0.001, 0.01, 0.05, 0.1, 0.2]  # 0.05
# depth_range = [2, 3, 4, 5]  # 3
```

目录

```
# param_grid = [
#     {
#         "gbr__max_depth": depth_range,
#         "gbr__n_estimators": num_estimator_range,
#         "gbr__learning_rate": lr_range,
#     }
# ]
# gs = GridSearchCV(estimator=pipe_gbr,
#                   param_grid=param_grid,
#                   cv=10,
#                   verbose=2,
#                   n_jobs=-1)
# gs = gs.fit(x_train, y_train)

# print(gs.best_score_)
# print(gs.best_params_)
# gbr = gs.best_estimator_

gbr = GradientBoostingRegressor(learning_rate=0.05, max_depth=3, n_estimators=60)
gbr.fit(x_train, y_train)
```

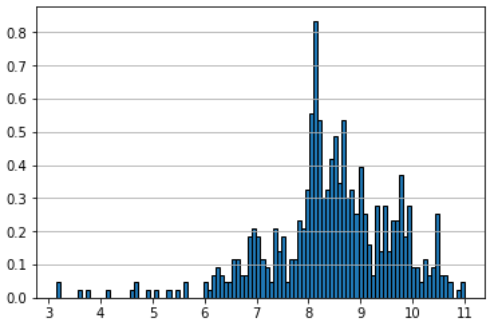
Out[30]:

```
GradientBoostingRegressor(alpha=0.9, criterion='friedman_mse', init=None,
             learning_rate=0.05, loss='ls', max_depth=3, max_features=None,
             max_leaf_nodes=None, min_impurity_decrease=0.0,
             min_impurity_split=None, min_samples_leaf=1,
             min_samples_split=2, min_weight_fraction_leaf=0.0,
             n_estimators=60, presort='auto', random_state=None,
             subsample=1.0, verbose=0, warm_start=False)
```

In [31]:

```
gbr_pred_valid = gbr.predict(x_valid)
print(pearsonr(gbr_pred_valid, y_valid))

gbr_test_pred = gbr.predict(test_std)
plt.hist(gbr_test_pred, normed=True, bins=100, align="left", edgecolor='black')
plt.grid(axis="y")
plt.show()
```

```
(0.8748598792096441, 6.985295657101302e-77)
```



In [36]:

```
regressors = [rfr, svr, gbr, lgbr,xgbr]
stacker = StackingRegressor(regressors=regressors, meta_regressor=xgbr, verbose=1)

stacker.fit(x_train, y_train)
```

```
Fitting 5 regressors...
Fitting regressor1: randomforestregressor (1/5)
Fitting regressor2: svr (2/5)
Fitting regressor3: gradientboostingregressor (3/5)
Fitting regressor4: lgbmregressor (4/5)
Fitting regressor5: xgbregressor (5/5)

/usr/local/lib/python3.5/dist-packages/lightgbm/engine.py:99: UserWarning: Found `num_iteration` in params. Will use
  warnings.warn("Found `{}` in params. Will use it instead of argument".format(alias))
```

Out[36]:

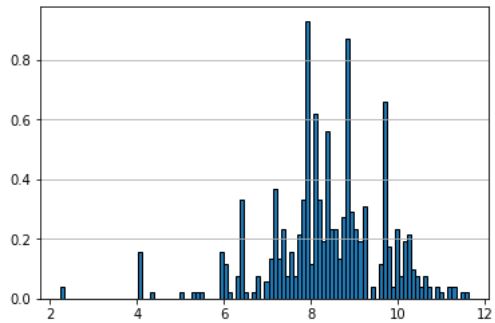
```
StackingRegressor(meta_regressor=XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
       colsample_bytree=1, gamma=0, learning_rate=0.04, max_delta_step=0,
       max_depth=3, min_child_weight=1, missing=None, n_estimators=135,
       n_jobs=1, nthread=None, objective='reg:linear', random_state=0,
       reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=609,
       silent=True, subsample=1),
         regressors=[RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=4,
           max_features='auto', max_leaf_nodes=None,
```

```
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, n_estimator...0,
        reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=609,
        silent=True, subsample=1)],
          store_train_meta_features=False, verbose=1)
```

In [37]:

```
stacker_valid_pred = stacker.predict(x_valid)
print(pearsonr(stacker_valid_pred, y_valid))

stacker_test_pred = stacker.predict(test_std)
plt.hist(stacker_test_pred, normed=True, bins=100, align="left", edgecolor='black')
plt.grid(axis="y")
plt.show()
```

```
(0.8743437567468514, 1.103880287517364e-76)
```



In [53]:

```
with open("submission.csv", "w") as fout:
    for idx, p, s, r, g, l in zip(df_test["id"], xgbr_test_pred, svr_test_pred, rfr_test_pred, gbr_test_pred, lgbr
        avg = sum([p, s, r, g, l, k]) / 6
        fout.write(str(idx) + "," + str(avg) + "\n")
```

In [54]:

```
!head submission.csv
```

```
1201,8.413708773849518
1202,8.763303222622568
1203,8.394065810872593
1204,8.012417539806851
1205,4.721412879077366
1206,8.068428930269672
1207,7.085037682600749
1208,9.280111949816199
1209,9.769760347060874
1210,8.844779483905949
```

In [73]:

```
!wget -nv -O kesci_submit https://cdn.kesci.com/submit_tool/v1/kesci_submit&&chmod +x kesci_submit
```

```
2018-02-09 02:40:06 URL:https://cdn.kesci.com/submit_tool/v1/kesci_submit [7840472/7840472] -> "kesci_submit" [1]
```

In [55]:

```
!./kesci_submit -token 7228f5df5109e12c -file submission.csv
```

```
Kesci Submit Tool
Working...
Success.
OK
```

2.2 深度学习方法

下面介绍深度学习方法。

AAAI 2018 恰好接收了一篇 AES 相关的论文：SkipFlow: Incorporating Neural Coherence Features for End-to-End Automatic Text Scoring (https://arxiv.org/abs/1711.04981)，文章表示在 ASAP 数据集（本次项目使用的数据集实际是其中的 set1）上取得了 the state of art 的结果。

因此，借项目的机会，我尝试复现文章的结果；借文章的方法，我也希望能学习掌握 AES 的深度学习方法，顺便取得好成绩。可以说，我主要采用了本文提出的 SKIPFLOW 模型。

就 AES 而言，目前深度学习方法已经取得了与人工提取特征的传统方法可比，甚至更好的性能。

SKIPFLOW 的基本思想是，一般的深度学习方法并没有充分利用`coherence features` 连贯性特征。因此，文章提出了 SKIPFLOW 机制来对其进行建

模。

SKIPFLOW 模型的架构如下所示，看似非常简单，涉及到：

目录

1. Embedding Layer - Lookup table；
2. LSTM Layer - Model essay；
3. Temporal Mean Pooling - Represent eesay；
4. Tensor Layer - Model coherence。

skipflow.png

鉴于上一个项目，GloVe 的优越性能，此处我依旧使用预训练的 Glove.840B.300d Word Vector 来创建 Embedding Layer。

鉴于最长的 essay 实在太长（783 个单词），我设置了最大长度为 500，长则截，短则补。

Tensor layer 出自 Reasoning with Neural Tensor Networks for Knowledge Base Completion (https://nlp.stanford.edu/pubs/SocherChenManningNg_NIPS2013.pdf) 一文，它的使用目标就是对节点间的关系进行建模。

> the goal is link prediction in an existing network of relationships between entity nodes.

SKIPFLOW 的作者借来对 hidden states pair 间的关系进行建模。 Tensor layer 的公式和结构如下所示：

ntn2.jpg

ntn.png

其中 e_1, e_2 代表两个节点，R 就是`Relationship`。

GitHub 上有一个 keras tensor layer 的开源实现。我因此偷了个懒，直接“借”了这段代码（见下），使用了 Keras。事实证明这是一个致命的错误！

SKIPFLOW 的关键点是，隔 δ 的间距取 hidden states pair，简单起见，文中使用了固定的 δ。但由于数据集中 essay 长短不一，对于 padded 成最大长度的 essay，对于所有 essay 等间距地取 hidden state，会超出 essay 的原始长度。文中使用循环的机制来解决这个问题。但 keras 的静态计算图机制，所有 hidden states 的取点必须在创建静态图时指定，应用于 padded essay 时，后续的 hidden states pair 实际并不能提供有效的 coherence feature。这就使得 keras 天然不能发挥 SKIPFLOW 真正的优势。如果使用可变的 δ，还是由于静态计算图的问题，在构建计算图时，变量只是一个 placeholder，并无法得知输入数据的真正长度，实际也不可行。

> 简单地看过 Keras，以上是漏见。

我采用了三种方法来解决以上问题：

1. essay 的 token sequence 向后补零到最大长度（一般都是这样做的），不做其他任何处理。这样做的后果是，mask 部分的 hidden states pairs 是无意义的，也许还是有害的。(另外使用了支持 Mask 的 Temporal Mean Pooling，因此 mask 对于 Temporal Mean Pooling 无影响）；
2. 输入两个 essay，一个向后补零到最大长度，记作 A，一个重复 essay 序列，直到最大长度，记作 B。在 A 上做 Temporal Mean Pooling，在 B 上取 hidden states 计算 relationship；
3. essay 的 token sequence 重复直到最大长度，不做其他任何处理。Temporal Mena Pooling 和 Tensor Layer 都应用于该序列。

后两种方法的问题在于，重复部分的 hidden states 失真了。也许得益于 LSTM，后续重复的 hidden states 能更少地依赖于原 essay。但这谁又知道呢?

> 我在本地跑传统机器学习方法，在远程服务器上跑深度学习方法。本文档由两个 Nobtebook 合并而来。因此以下的深度学习代码没有训练过程的执行结果。

In [56]:

```
class NeuralTensorLayer(Layer):
    def __init__(self, output_dim, input_dim=None, **kwargs):
        self.output_dim = output_dim #k
        self.input_dim = input_dim   #d
        if self.input_dim:
            kwargs['input_shape'] = (self.input_dim,)
        super(NeuralTensorLayer, self).__init__(**kwargs)


    def build(self, input_shape):
        mean = 0.0
        std = 1.0
        # W : k*d*d
        k = self.output_dim
        d = self.input_dim
        initial_W_values = stats.truncnorm.rvs(-2 * std, 2 * std, loc=mean, scale=std, size=(k,d,d))
        initial_V_values = stats.truncnorm.rvs(-2 * std, 2 * std, loc=mean, scale=std, size=(2*d,k))
        self.W = K.variable(initial_W_values)
        self.V = K.variable(initial_V_values)
        self.b = K.zeros((self.input_dim,))
        self.trainable_weights = [self.W, self.V, self.b]
```

```
        self.trainable_weights = [self.W, self.V, self.b]


    def call(self, inputs, mask=None):
        if type(inputs) is not list or len(inputs) <= 1:
            raise Exception('BilinearTensorLayer must be called on a list of tensors '
                            '(at least 2). Got: ' + str(inputs))
        e1 = inputs[0]
        e2 = inputs[1]
        batch_size = K.shape(e1)[0]
        k = self.output_dim

        feed_forward_product = K.dot(K.concatenate([e1,e2]), self.V)

        bilinear_tensor_products = [ K.sum((e2 * K.dot(e1, self.W[0])) + self.b, axis=1) ]

        for i in range(k)[1:]:
            btp = K.sum((e2 * K.dot(e1, self.W[i])) + self.b, axis=1)
            bilinear_tensor_products.append(btp)
        result = K.tanh(K.reshape(K.concatenate(bilinear_tensor_products, axis=0), (batch_size, k)) + feed_forward

        return result


    def compute_output_shape(self, input_shape):
        # print (input_shape)
        batch_size = input_shape[0][0]
        return (batch_size, self.output_dim)
```

In [57]:

```
class TemporalMeanPooling(Layer):
    """
    This is a custom Keras layer. This pooling layer accepts the temporal
    sequence output by a recurrent layer and performs temporal pooling,
    looking at only the non-masked portion of the sequence. The pooling
    layer converts the entire variable-length hidden vector sequence
    into a single hidden vector, and then feeds its output to the Dense
    layer.

    input shape: (nb_samples, nb_timesteps, nb_features)
    output shape: (nb_samples, nb_features)
    """

    def __init__(self, **kwargs):
        super(TemporalMeanPooling, self).__init__(**kwargs)
        self.supports_masking = True
        self.input_spec = InputSpec(ndim=3)

    def compute_output_shape(self, input_shape):
        return (input_shape[0], input_shape[2])

    def call(self, x, mask=None): # mask: (nb_samples, nb_timesteps)
        if mask is None:
            mask = K.mean(K.ones_like(x), axis=-1)
        ssum = K.sum(x, axis=-2)

        mask = K.cast(mask, K.floatx())

        rcnt = K.sum(mask, axis=-1, keepdims=True) # (nb_samples)

        return ssum / rcnt


    def compute_mask(self, input, mask):
        return None
```

In [60]:

```
class Conv1DWithMasking(Convolution1D):
    def __init__(self, **kwargs):
        self.supports_masking = True
        super(Conv1DWithMasking, self).__init__(**kwargs)

    def compute_mask(self, x, mask):
        return mask
```

设置最大长度为 500，覆盖了绝大多数的 essay。

In [61]:

```
# MAX_SEQUENCE_LENGTH = df_train["num_tokens"].max()

MAX_SEQUENCE_LENGTH = 500  # ~80%
```

```
MAX_SEQUENCE_LENGTH = 300  # 80%
```

以下的side_wsequence就是不断重复自身，直到最大长度的 essay token sequnces。

目录

In [62]:

```
def complement(sequence, size):
    clone = sequence[:]
    count = 0
    ssize = len(sequence)
    while len(clone) < size:
        clone.append(sequence[count % ssize])
        count += 1

    return clone


max_complement = partial(complement, size=MAX_SEQUENCE_LENGTH)

df_train["side_wsequence"] = df_train["wsequence"].apply(max_complement)  # 原意是用于辅助的 sequence of tokens, 不断
df_test["side_wsequence"] = df_test["wsequence"].apply(max_complement)
```

In [63]:

```
embeddings_index = {}
vocab = set()

with open("full.simple.glove.840B.300d.txt") as fin:  # full.simple.glove.840B.300d.txt 根据数据集提取而来
    for line in fin:
        values = line.split()
        word = values[0]
        vocab.add(word)
        coefs = np.asarray(values[1:], dtype='float32')
        embeddings_index[word] = coefs

print('Found %s word vectors.' % len(embeddings_index))
```

```
Found 12653 word vectors.
```

In [64]:

```
word2index = {w: i for i, w in enumerate(vocab, 2)}
word2index["<pad>"] = 0
word2index["<unk>"] = 1

index2word = {i: w for w, i in word2index.items()}
```

In [65]:

```
VOCAB_SIZE = len(word2index)
EMBEDDING_DIM = 300

embedding_matrix = np.zeros((VOCAB_SIZE, EMBEDDING_DIM))
for i, word in index2word.items():
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        # words not found in embedding index will be all-zeros.
        embedding_matrix[i] = embedding_vector
```

In [66]:

```
embedding_layer = Embedding(VOCAB_SIZE,
                            EMBEDDING_DIM,
                            weights=[embedding_matrix],
                            input_length=MAX_SEQUENCE_LENGTH,
                            mask_zero=True,  # masked
                            trainable=False)

side_embedding_layer = Embedding(VOCAB_SIZE,
                                 EMBEDDING_DIM,
                                 weights=[embedding_matrix],
                                 input_length=MAX_SEQUENCE_LENGTH,
                                 mask_zero=False,  # No mask
                                 trainable=False)
```

In [67]:

```
df_train["isequence"] = df_train["wsequence"].apply(lambda x: [word2index[w] if w in word2index.keys() else 1 for w
df_test["isequence"] = df_test["wsequence"].apply(lambda x: [word2index[w] if w in word2index.keys() else 1 for w

df_train["side_isequence"] = df_train["side_wsequence"].apply(lambda x: [word2index[w] if w in word2index.keys() e
df_test["side_isequence"] = df_test["side_wsequence"].apply(lambda x: [word2index[w] if w in word2index.keys() els
```

In [68]:

```
# 归一化

score_min = df_train["score"].min()
score_range = df_train["score"].max() - score_min
df_train["normalized_score"] = df_train["score"].apply(lambda x: (x - score_min) / score_range)
```

In [69]:

```
data = df_train["isequence"].values
data = pad_sequences(data, maxlen=MAX_SEQUENCE_LENGTH, padding="post", truncating="post")
side_data = pad_sequences(df_train["side_isequence"].values, maxlen=MAX_SEQUENCE_LENGTH, padding="post", truncatin
target = df_train["normalized_score"].values
# target = df_train["score"].values
```

前文介绍到人工提取的特征后来也作为神经网络的额外输入，此处对数据进行标准化。

In [70]:

```
trad_feat_std = StandardScaler()

trad_feats = trad_feat_std.fit_transform(df_train[[
    "num_tokens","num_sentences","average_sent_len","average_word_len","num_long_words","num_stopwords",
    "num_characters","num_commas","num_quotations","num_exclamation_marks","f_score","type_token_ratio",
    "avg_word_freq","cohesion","one_gram_overlap","two_gram_overlap","three_gram_overlap","spelling_error",
    "grammar_error","syllable_count","flesch_reading_ease","flesch_kincaid_grade","gunning_fog","smog_index",
    "automated_readability_index","coleman_liau_index","linsear_write_formula","dale_chall_readability_score",
    "difficult_words","neg_sentiment","neu_sentiment","pos_sentiment"
]].values)
```

In [71]:

```
VALIDATION_SPLIT = 0.2

indices = np.arange(data.shape[0])
np.random.shuffle(indices)


data = data[indices]  # shuffle
target = target[indices]
side_data = side_data[indices]
trad_feats = trad_feats[indices]


num_validation_samples = int(VALIDATION_SPLIT * data.shape[0])


x_train = data[:-num_validation_samples]  # train valid split
side_x_train = side_data[:-num_validation_samples]
y_train = target[:-num_validation_samples]
trad_feats_train = trad_feats[:-num_validation_samples]


x_valid = data[-num_validation_samples:]
side_x_valid = side_data[-num_validation_samples:]
y_valid = target[-num_validation_samples:]
trad_feats_valid = trad_feats[-num_validation_samples:]
```

In [72]:

```
OUT_INITIAL_BIAS_MEAN = df_train["normalized_score"].mean()
OUT_INITIAL_BIAS_STD = df_train["normalized_score"].std()

# OUT_INITIAL_BIAS_MEAN = df_train["score"].mean()  # for regression
# OUT_INITIAL_BIAS_STD = df_train["score"].std()
```

In [73]:

```
EMBEDDING_DIM = 300

def SKIPFLOW(lstm_dim=50, lr=1e-4, lr_decay=1e-6, k=5, eta=3, delta=50, activation="relu", maxlen=MAX_SEQUENCE_LEN
    # Input
    e = Input(name='essay',shape=(maxlen,))
    side_e = Input(name="complementary_essay", shape=(maxlen, ))
    trad_feats = Input(name="trad_feats", shape=(32, ))


    # Embedding
    embed = embedding_layer(e)
    side_embed = side_embedding_layer(side_e)


    # Conv
    conv_layer = Conv1DWithMasking(filters=300, kernel_size=3, strides=1, padding="same", activation="relu")
    embed = conv_layer(embed)
```

目录

```
    # LSTM
    lstm_layer = LSTM(lstm_dim, return_sequences=True)
    hidden_states = lstm_layer(embed)  # shape: [batch_size, timesteps, units]
    side_hidden_states = lstm_layer(side_embed)


    # Temporal Mean Pooling
    htm = TemporalMeanPooling()(hidden_states)


    # Tensor Layer
    tensor_layer = NeuralTensorLayer(output_dim=k, input_dim=lstm_dim)
    pairs = [
        ((eta + i * delta) % maxlen, (eta + i * delta + delta) % maxlen)
        for i in range(maxlen // delta)
    ]
    hidden_pairs = [
        (Lambda(lambda t: t[:, p[0], :])(side_hidden_states), Lambda(lambda t: t[:, p[1], :])(side_hidden_states))
        for p in pairs
    ]
    sigmoid = Dense(1, activation="sigmoid", kernel_initializer=initializers.glorot_normal(seed=seed))
    coherence = [sigmoid(tensor_layer([hp[0], hp[1]])) for hp in hidden_pairs]



    # FC
    co_tm = Concatenate()(coherence[:]  + [htm] + [trad_feats])

    dense = Dense(256, activation=activation,
                  kernel_initializer=initializers.glorot_normal(seed=seed))(co_tm)
    dense = Dense(128, activation=activation,
                 kernel_initializer=initializers.glorot_normal(seed=seed))(dense)
    dense = Dense(64, activation=activation,
                 kernel_initializer=initializers.glorot_normal(seed=seed))(dense)


    # Output
    out = Dense(1, activation="sigmoid",
                kernel_initializer=initializers.glorot_normal(seed=seed),
                bias_initializer=initializers.TruncatedNormal(OUT_INITIAL_BIAS_MEAN, OUT_INITIAL_BIAS_STD, seed=se
               )(dense)

    model = Model(inputs=[e, side_e, trad_feats], outputs=[out])

    adam = Adam(lr=lr, decay=lr_decay)


    # SKIPFLOW 论文中使用 MSE
    model.compile(loss="mean_squared_error", optimizer=adam, metrics=["MSE"])
    # model.compile(loss="binary_crossentropy", optimizer=adam, metrics=["MSE"])
    return model
```

In [74]:

```
earlystopping = EarlyStopping(monitor="val_mean_squared_error", patience=5)
```

In [75]:

```
sf_1 = SKIPFLOW(lstm_dim=50, lr=2e-4, lr_decay=2e-6, k=4, eta=13, delta=50, activation="relu", seed=None)
```

In [ ]:

```
epochs = 1000
hist = sf_1.fit([x_train, side_x_train, trad_feats_train], y_train, batch_size=1024, epochs=epochs,
                validation_data=([x_valid, side_x_valid, trad_feats_valid], y_valid), callbacks=[earlystopping])
```

In [ ]:

```
loss = hist.history['loss']
val_loss = hist.history['val_loss']
plt.plot(loss)
plt.plot(val_loss)
plt.legend(['loss', 'val_loss'])
plt.show()
```

某一次在服务器上完成训练后，绘制的 lossg-val_loss图如下所示（loss 和 metric 都使用的 MSE，使用了 EarlyStopping。）

loss_val_loss.png

目录

讨论与总结

时间关系，距离 23:59 只有 10 分钟了，容许我长发短说。

就我个人而言，使用人工提供特征的传统机器学习方法取得的结果明显优于端到端的深度学习方法。这可能是因为我没有实现真正的 SKIPFLOW 机制。但有一点，我一直牢记在心：

> 能提取的特征，做特征工程进行提取是可考虑的。这反而减少了深度学习模型的设计与训练负担。

但总的来说，我是一个 End to End 的支持者，机器有机器理解世界的方式嘛~

最后，不忘初心，方得始终！（还是 PyTorch 大法好，我应该坚守阵地的）

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