NaiceBayes

首先使用jieba对中文文本进行分词，然后将分词后的文本数据转换为TF-IDF特征表示，使用NB分类器进行训练。

在NB模型中，分别进行了两次实验。第一次实验中仅仅针对content和label进行训练，第二次实验中加入了title和tag作为特征进行训练。

ps：关于停用词，按我的理解加上停用词之后，模型性能并没有得到提升，所以就没有再用了。

```
import pandas as pd
import jieba
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.feature_selection import SelectKBest, chi2
from sklearn.pipeline import Pipeline
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score, precision_score,
recall_score, f1_score

# 读取训练集和测试集CSV文件
train_df = pd.read_csv('train.csv', usecols=['content', 'label'])
test_df = pd.read_csv('test.csv', usecols=['content', 'label'])

# 将训练集和测试集分成文本内容和标签两部分
train_data = train_df['content']
train_labels = train_df['label']
test_data = test_df['content']
test_labels = test_df['label']

# 定义分词函数
def tokenize(text):
    return list(jieba.cut(text))


# 定义特征提取器
vectorizer = TfidfVectorizer(tokenizer=tokenize)

# 训练集和测试集特征提取器
train_features = vectorizer.fit_transform(train_data)
```

```
test_features = vectorizer.transform(test_data)

# 特征选择器，选择10000个最具有区分性的特征
selector = SelectKBest(chi2, k=10000)
selector.fit(train_features, train_labels)
train_features = selector.transform(train_features)
test_features = selector.transform(test_features)

# 定义NB分类器
clf = MultinomialNB(alpha=1.0)

# 训练NB模型
clf.fit(train_features, train_labels)

# 在测试集上进行预测
predicted_labels = clf.predict(test_features)

# 计算准确率、精确率、召回率和F1值
accuracy = accuracy_score(test_labels, predicted_labels)
precision = precision_score(test_labels, predicted_labels,
pos_label=1)
recall = recall_score(test_labels, predicted_labels, pos_label=1)
f1 = f1_score(test_labels, predicted_labels, pos_label=1)

# 输出评估结果
print("准确率：{:.4f}".format(accuracy))
print("精确率：{:.4f}".format(precision))
print("召回率：{:.4f}".format(recall))
print("F1值：{:.4f}".format(f1))
```

所得模型数据如下：

```
准确率：0.8448
精确率：0.8422
召回率：0.8602
F1值：0.8511
```

```
import pandas as pd
import jieba
from sklearn.feature_extraction.text import TfidfVectorizer
```

```
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score, precision_score,
recall_score, f1_score

# 定义中文分词函数
def chinese_tokenizer(text):
    return list(jieba.cut(text))

# 读入训练集数据
train_df = pd.read_csv('train.csv')

# 将title列和tag列的数据合并到原始的文本数据中
train_text = train_df['content'] + ' ' + train_df['title'] + ' ' +
train_df['tag']

# 对中文文本进行向量化
vectorizer = TfidfVectorizer(tokenizer=chinese_tokenizer)
train_features = vectorizer.fit_transform(train_text)

# 将标签转换为数字形式
train_labels = train_df['label']

# 训练朴素贝叶斯分类器
clf = MultinomialNB(alpha=1.0)
clf.fit(train_features, train_labels)

# 使用训练好的模型进行预测
test_df = pd.read_csv('test.csv')
test_text = test_df['content'] + ' ' + test_df['title'] + ' ' +
test_df['tag']
test_features = vectorizer.transform(test_text)
test_pred = clf.predict(test_features)

# 计算准确率、精确率、召回率和F1值
accuracy = accuracy_score(test_df['label'], test_pred)
precision = precision_score(test_df['label'], test_pred,
average='macro')
recall = recall_score(test_df['label'], test_pred, average='macro')
f1 = f1_score(test_df['label'], test_pred, average='macro')

# 输出结果
print("准确率：{:.4f}".format(accuracy))
```

```
print("精确率：{:.4f}".format(precision))
print("召回率：{:.4f}".format(recall))
print("F1值：{:.4f}".format(f1))
```

所得模型数据如下：

```
准确率：0.8579
精确率：0.8583
召回率：0.8572
F1值：0.8575
```

SVM

先用jieba对中文文本进行分词操作，然后将文本数据转换为TF-IDF特征表示，然后用SVM分类器进行训练。

在训练SVM模型时，分别使用了默认的linear和rbf核函数来进行训练。在最初的实验中没有使用jieba对中文文本进行分词，得到的模型F1 score只有60左右。之后引入jieba分词对中文文本进行预处理。

```
import pandas as pd
import jieba
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score,
precision_recall_fscore_support

# 加载数据集
train_data = pd.read_csv('train.csv')
test_data = pd.read_csv('test.csv')

# 对每个样本的文本列进行分词处理
train_data['content'] = train_data['content'].apply(lambda x: '
'.join(jieba.cut(x)))
test_data['content'] = test_data['content'].apply(lambda x: '
'.join(jieba.cut(x)))

# 设置标签
train_label = train_data['label'].astype(int)
test_label = test_data['label'].astype(int)
```

```
# 特征提取和预处理
vectorizer = TfidfVectorizer()
train_features = vectorizer.fit_transform(train_data['content'])
test_features = vectorizer.transform(test_data['content'])

# 定义SVM模型，并使用RBF核函数
# clf = SVC(kernel='rbf', C=5.0, gamma='scale')
clf = SVC(kernel='linear', C=5.0, gamma='scale')

# 拟合模型
clf.fit(train_features, train_label)

# 使用训练好的模型对测试集进行预测
y_pred = clf.predict(test_features)

# 将预测结果写入CSV文件
test_data['predicted_label'] = y_pred
test_data.to_csv('SVMTestOneRes.csv', index=False)

# 计算准确率、精确率、召回率、F1得分和support
accuracy = accuracy_score(test_label, y_pred)
precision, recall, f1_score, support =
precision_recall_fscore_support(test_label, y_pred)

# 输出结果
print('Accuracy:', accuracy)
print('Precision:', precision)
print('Recall:', recall)
print('F1 score:', f1_score)
print('Support:', support)
```

所得结果如下：

```
Accuracy: 0.8428428428428428
Precision: [0.84713376 0.83901515]
Recall: [0.82438017 0.86019417]
F1 score: [0.83560209 0.84947267]
Support: [484 515]
```

使用rbf核函数的代码如下：

```
import pandas as pd
import jieba
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score,
precision_recall_fscore_support

# 定义分词函数
def tokenize(text_list):
    # 对输入的文本列表进行分词处理
    tokenized_list = []
    for text in text_list:
        tokenized_list.append(' '.join(jieba.cut(text)))
    return tokenized_list

# 加载数据集
train_data = pd.read_csv('train.csv')
test_data = pd.read_csv('test.csv')

# 对原始文本进行分词处理
train_text = tokenize(train_data['content'] + train_data['title'] +
train_data['tag'])
test_text = tokenize(test_data['content'] + test_data['title'] +
test_data['tag'])

# 设置标签
train_label = train_data['label'].astype(int)
test_label = test_data['label'].astype(int)

# 特征提取和预处理
vectorizer = TfidfVectorizer()
train_features = vectorizer.fit_transform(train_text)
test_features = vectorizer.transform(test_text)

# 定义SVM模型，并使用RBF核函数
clf = SVC(kernel='rbf', C=5.0, gamma='scale')
# clf = SVC(kernel='linear', C=5.0)

# 拟合模型
clf.fit(train_features, train_label)

# 使用训练好的模型对测试集进行预测
```

```
y_pred = clf.predict(test_features)

# 将预测结果写入CSV文件
test_data['predicted_label'] = y_pred
test_data.to_csv('SVMTestTwoRes.csv', index=False)

# 计算准确率、精确率、召回率、F1得分和support
accuracy = accuracy_score(test_label, y_pred)
precision, recall, f1_score, support =
precision_recall_fscore_support(test_label, y_pred)

# 输出结果
print('Accuracy:', accuracy)
print('Precision:', precision)
print('Recall:', recall)
print('F1 score:', f1_score)
print('Support:', support)
```

所得结果如下：

```
# 左侧为真新闻数据，右侧为假新闻数据
Accuracy: 0.8498498498498499
Precision: [0.83943089 0.85996055]
Recall: [0.85330579 0.84660194]
F1 score: [0.84631148 0.85322896]
Support: [484 515]
```

LogisticRegression

首先用jieba对中文文本进行分词，将文本数据转换为TF-IDF特征表示，然后使用LR训练分类模型。

```
import pandas as pd
import jieba
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score,
precision_recall_fscore_support

# 定义分词函数
def tokenize(text):
```

```
    return ' '.join(jieba.cut(text))

# 加载训练集和测试集数据
train_data = pd.read_csv('train.csv')
test_data = pd.read_csv('test.csv')

# 选择需要用于训练模型的列和标签列
train_features = train_data[['content', 'title', 'tag']].copy()
train_label = train_data['label'].copy()
test_features = test_data[['content', 'title', 'tag']].copy()
test_label = test_data['label'].copy()

# 对训练集和测试集数据进行预处理
train_features.loc[:, 'content'] =
train_features['content'].apply(tokenize)
train_features.loc[:, 'title'] =
train_features['title'].apply(tokenize)
train_features.loc[:, 'tag'] =
train_features['tag'].apply(tokenize)
test_features.loc[:, 'content'] =
test_features['content'].apply(tokenize)
test_features.loc[:, 'title'] =
test_features['title'].apply(tokenize)
test_features.loc[:, 'tag'] = test_features['tag'].apply(tokenize)

# 将文本数据转化为 TF-IDF 向量
vectorizer = TfidfVectorizer(stop_words=None)
train_features =
vectorizer.fit_transform(train_features.apply(lambda x: '
'.join(x), axis=1))
test_features = vectorizer.transform(test_features.apply(lambda x:
' '.join(x), axis=1))

# 定义LR模型
clf = LogisticRegression()

# 训练模型
clf.fit(train_features, train_label)

# 预测测试集
y_pred = clf.predict(test_features)
```

```
# 计算准确率、精确率、召回率、F1得分和support
accuracy = accuracy_score(test_label, y_pred)
precision, recall, f1_score, support =
precision_recall_fscore_support(test_label, y_pred)

# 输出结果
print('Accuracy:', accuracy)
print('Precision:', precision)
print('Recall:', recall)
print('F1 score:', f1_score)
print('Support:', support)
```

输出结果如下：

```
Accuracy: 0.8428428428428428
Precision: [0.8250497 0.8608871]
Recall: [0.85743802 0.82912621]
F1 score: [0.84093212 0.84470821]
Support: [484 515]
```

LSTM

基于Keras。将文本转换为整数序列并填充到同一length，使用embedding将每个整数序列编码为向量表示，然后使用LSTM层对content、title、tag输入序列进行处理，将三个输入序列的隐藏状态拼接到一起，进行分类预测。训练过程中使用二元交叉损失函数和Adam优化器进行优化。

```
import pandas as pd
import numpy as np
import jieba
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.layers import Input, Embedding, LSTM, Dense,
Bidirectional
from keras.models import Model
from keras.callbacks import EarlyStopping
from sklearn.metrics import accuracy_score,
precision_recall_fscore_support
```

```
# 加载训练集和测试集数据
train_data = pd.read_csv('train.csv')
test_data = pd.read_csv('test.csv')

# 分词处理
def tokenize(text):
    return ' '.join(jieba.cut(text))

train_data['content'] = train_data['content'].apply(tokenize)
train_data['title'] = train_data['title'].apply(tokenize)
train_data['tag'] = train_data['tag'].apply(tokenize)
test_data['content'] = test_data['content'].apply(tokenize)
test_data['title'] = test_data['title'].apply(tokenize)
test_data['tag'] = test_data['tag'].apply(tokenize)

# 创建词汇表
tokenizer = Tokenizer(num_words=5000, oov_token='<OOV>')
tokenizer.fit_on_texts(train_data['content'].tolist() +
train_data['title'].tolist() + train_data['tag'].tolist())

# 将文本转化为序列
train_content_seq =
tokenizer.texts_to_sequences(train_data['content'].tolist())
train_title_seq =
tokenizer.texts_to_sequences(train_data['title'].tolist())
train_tag_seq =
tokenizer.texts_to_sequences(train_data['tag'].tolist())
test_content_seq =
tokenizer.texts_to_sequences(test_data['content'].tolist())
test_title_seq =
tokenizer.texts_to_sequences(test_data['title'].tolist())
test_tag_seq =
tokenizer.texts_to_sequences(test_data['tag'].tolist())

# 对序列进行填充
maxlen = 500
train_content_seq = pad_sequences(train_content_seq,
padding='post', maxlen=maxlen)
train_title_seq = pad_sequences(train_title_seq, padding='post',
maxlen=maxlen)
train_tag_seq = pad_sequences(train_tag_seq, padding='post',
maxlen=maxlen)
```

```
test_content_seq = pad_sequences(test_content_seq, padding='post',
maxlen=maxlen)
test_title_seq = pad_sequences(test_title_seq, padding='post',
maxlen=maxlen)
test_tag_seq = pad_sequences(test_tag_seq, padding='post',
maxlen=maxlen)

# 构建模型
input_content = Input(shape=(maxlen,))
input_title = Input(shape=(maxlen,))
input_tag = Input(shape=(maxlen,))

embedding_dim = 32
lstm_units = 64
dropout_rate = 0.2

embedding_layer = Embedding(input_dim=len(tokenizer.word_index)+1,
output_dim=embedding_dim, input_length=maxlen)
lstm_layer = LSTM(units=lstm_units, dropout=dropout_rate,
recurrent_dropout=dropout_rate, return_sequences=True)

content_embedding = embedding_layer(input_content)
content_lstm = lstm_layer(content_embedding)

title_embedding = embedding_layer(input_title)
title_lstm = lstm_layer(title_embedding)

tag_embedding = embedding_layer(input_tag)
tag_lstm = lstm_layer(tag_embedding)

merged = Bidirectional(LSTM(units=lstm_units))(content_lstm)
merged = Dense(units=16, activation='relu')(merged)
merged = Dense(units=1, activation='sigmoid')(merged)

model = Model(inputs=[input_content,input_title,input_tag],
outputs=merged)

model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy'])

# 训练模型
early_stopping = EarlyStopping(monitor='val_loss', patience=3)
```

```
history = model.fit(x=[train_content_seq, train_title_seq,
train_tag_seq], y=train_data['label'].values, batch_size=64,
                    epochs=20, validation_split=0.1, callbacks=
[early_stopping])

# 在测试集上进行预测
y_pred = model.predict([test_content_seq, test_title_seq,
test_tag_seq])
y_pred = np.round(y_pred).flatten()

# 输出结果
accuracy = accuracy_score(test_data['label'].values, y_pred)
precision, recall, f1_score, support =
precision_recall_fscore_support(test_data['label'].values, y_pred)

print('Accuracy:', accuracy)
print('Precision:', precision)
print('Recall:', recall)
print('F1 score:', f1_score)
print('Support:', support)
```

所得结果如下：

```
Epoch 1/20
57/57 [==============================] - 81s 1s/step - loss: 0.6606
- accuracy: 0.5860 - val_loss: 0.5319 - val_accuracy: 0.7500
Epoch 2/20
57/57 [==============================] - 87s 2s/step - loss: 0.4095
- accuracy: 0.8173 - val_loss: 0.4655 - val_accuracy: 0.8100
Epoch 3/20
57/57 [==============================] - 86s 2s/step - loss: 0.2268
- accuracy: 0.9144 - val_loss: 0.5415 - val_accuracy: 0.7950
Epoch 4/20
57/57 [==============================] - 89s 2s/step - loss: 0.1288
- accuracy: 0.9525 - val_loss: 0.6724 - val_accuracy: 0.7375
Epoch 5/20
57/57 [==============================] - 90s 2s/step - loss: 0.0895
- accuracy: 0.9700 - val_loss: 0.7878 - val_accuracy: 0.7875
32/32 [==============================] - 4s 90ms/step
Accuracy: 0.7817817817817818
Precision: [0.77824268 0.78502879]
```

```
Recall: [0.76859504 0.79417476]
F1 score: [0.77338877 0.78957529]
Support: [484 515]
```

TextCNN

针对content、title和tag三个输入，定义大小为3、4、5的filter，对每个filter在输入上应用一维卷积操作，然后在每个卷积输出上应用一层池化层，最后将每个filter的输出连接在一起输入到一个全连接层中，使用sigmoid激活函数，进行二元分类预测。

有改变过embedding层的维度，不过性能提升很不显著。

```
import pandas as pd
import numpy as np
import jieba
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.layers import Input, Embedding, Conv1D, MaxPooling1D,
Flatten, Dense, concatenate
from keras.models import Model
from keras.callbacks import EarlyStopping
from sklearn.metrics import accuracy_score,
precision_recall_fscore_support

train_data = pd.read_csv('train.csv')
test_data = pd.read_csv('test.csv')

def tokenize(text):
    return ' '.join(jieba.cut(text))

train_data['content'] = train_data['content'].apply(tokenize)
train_data['title'] = train_data['title'].apply(tokenize)
train_data['tag'] = train_data['tag'].apply(tokenize)
test_data['content'] = test_data['content'].apply(tokenize)
test_data['title'] = test_data['title'].apply(tokenize)
test_data['tag'] = test_data['tag'].apply(tokenize)

tokenizer = Tokenizer(num_words=5000, oov_token='<OOV>')
tokenizer.fit_on_texts(train_data['content'].tolist() +
train_data['title'].tolist() + train_data['tag'].tolist())
```

```
# 将文本转换为序列
train_content_seq =
tokenizer.texts_to_sequences(train_data['content'].tolist())
train_title_seq =
tokenizer.texts_to_sequences(train_data['title'].tolist())
train_tag_seq =
tokenizer.texts_to_sequences(train_data['tag'].tolist())
test_content_seq =
tokenizer.texts_to_sequences(test_data['content'].tolist())
test_title_seq =
tokenizer.texts_to_sequences(test_data['title'].tolist())
test_tag_seq =
tokenizer.texts_to_sequences(test_data['tag'].tolist())

# 填充序列
maxlen = 500
train_content_seq = pad_sequences(train_content_seq,
padding='post', maxlen=maxlen)
train_title_seq = pad_sequences(train_title_seq, padding='post',
maxlen=maxlen)
train_tag_seq = pad_sequences(train_tag_seq, padding='post',
maxlen=maxlen)
test_content_seq = pad_sequences(test_content_seq, padding='post',
maxlen=maxlen)
test_title_seq = pad_sequences(test_title_seq, padding='post',
maxlen=maxlen)
test_tag_seq = pad_sequences(test_tag_seq, padding='post',
maxlen=maxlen)

# 模型构建
input_content = Input(shape=(maxlen,))
input_title = Input(shape=(maxlen,))
input_tag = Input(shape=(maxlen,))

embedding_dim = 100
num_filters = 128
filter_sizes = [3, 4, 5]

embedding_layer = Embedding(input_dim=len(tokenizer.word_index)+1,
output_dim=embedding_dim, input_length=maxlen)

content_embedding = embedding_layer(input_content)
```

```
title_embedding = embedding_layer(input_title)
tag_embedding = embedding_layer(input_tag)

conv_blocks = []
for filter_size in filter_sizes:
    conv = Conv1D(filters=num_filters, kernel_size=filter_size,
activation='relu')(content_embedding)
    conv = MaxPooling1D(pool_size=maxlen - filter_size + 1)(conv)
    conv = Flatten()(conv)
    conv_blocks.append(conv)

for filter_size in filter_sizes:
    conv = Conv1D(filters=num_filters, kernel_size=filter_size,
activation='relu')(title_embedding)
    conv = MaxPooling1D(pool_size=maxlen - filter_size + 1)(conv)
    conv = Flatten()(conv)
    conv_blocks.append(conv)

for filter_size in filter_sizes:
    conv = Conv1D(filters=num_filters, kernel_size=filter_size,
activation='relu')(tag_embedding)
    conv = MaxPooling1D(pool_size=maxlen - filter_size + 1)(conv)
    conv = Flatten()(conv)
    conv_blocks.append(conv)

merge = concatenate(conv_blocks, axis=-1)
dense = Dense(128, activation='relu')(merge)
output = Dense(1, activation='sigmoid')(dense)

model = Model(inputs=[input_content, input_title, input_tag],
outputs=output)

model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy'])

early_stopping = EarlyStopping(monitor='val_loss', patience=3)
history = model.fit(x=[train_content_seq, train_title_seq,
train_tag_seq], y=train_data['label'].values, batch_size=64,
                    epochs=20, validation_split=0.1, callbacks=
[early_stopping])
```

```
y_pred = model.predict([test_content_seq, test_title_seq,
test_tag_seq])
y_pred = np.round(y_pred).flatten()

accuracy = accuracy_score(test_data['label'].values, y_pred)
precision, recall, f1_score, support =
precision_recall_fscore_support(test_data['label'].values, y_pred)

# 输出结果
print('Accuracy:', accuracy)
print('Precision:', precision)
print('Recall:', recall)
print('F1 score:', f1_score)
print('Support:', support)
```

所得结果如下：

```
Epoch 1/20
57/57 [==============================] - 35s 595ms/step - loss:
0.5873 - accuracy: 0.6847 - val_loss: 0.5414 - val_accuracy: 0.7050
Epoch 2/20
57/57 [==============================] - 32s 567ms/step - loss:
0.3101 - accuracy: 0.8707 - val_loss: 0.4518 - val_accuracy: 0.8000
Epoch 3/20
57/57 [==============================] - 32s 566ms/step - loss:
0.0859 - accuracy: 0.9700 - val_loss: 0.4271 - val_accuracy: 0.8550
Epoch 4/20
57/57 [==============================] - 32s 564ms/step - loss:
0.0127 - accuracy: 0.9986 - val_loss: 0.4384 - val_accuracy: 0.8725
Epoch 5/20
57/57 [==============================] - 32s 570ms/step - loss:
0.0035 - accuracy: 0.9997 - val_loss: 0.4655 - val_accuracy: 0.8700
Epoch 6/20
57/57 [==============================] - 32s 568ms/step - loss:
0.0016 - accuracy: 1.0000 - val_loss: 0.5013 - val_accuracy: 0.8700
32/32 [==============================] - 2s 68ms/step
Accuracy: 0.8488488488488488
Precision: [0.84759916 0.85      ]
Recall: [0.83884298 0.85825243]
F1 score: [0.84319834 0.85410628]
Support: [484 515]
```