Feature Engineering

Leila Khalili

September 2020

1 Baseline features

1.1 Feature 1

The feature I used is Google's sentence embedding using the Universal sentence encoder. For this method I first downloaded a pretrained universal sentence encoder

```
#download the model to local so it can be used again and again
!mkdir ./input/module/module_useT
# Download the module, and uncompress it to the destination folder.
!curl -L "https://tfhub.dev/google/universal-sentence-encoder-large/3?tf-hub-format=compressed"
| tar -zxvC ./input/module/module_useT
```

Then the module is loaded to the disk and used to embed the sentences using the following code

```
def embed_useT(module):
    with tf.Graph().as_default():
        sentences = tf.placeholder(tf.string)
        embed = hub.Module(module)
        embeddings = embed(sentences)
        session = tf.train.MonitoredSession()
    return lambda x: session.run(embeddings, {sentences: x})
embed_fn = uf.embed_useT('./input/module/module_useT')
q_train_use = np.reshape(train.question.values,(len(train.question.values)))
r_train_use = np.reshape(train.response.values,(len(train.response.values)))

q_test_use = np.reshape(test.question.values,(len(test.question.values)))
r_test_use = np.reshape(test.response.values,(len(test.response.values)))

q_training_use = embed_fn(q_train_use)
r_training_use = embed_fn(r_train_use)

q_test_use = embed_fn(q_test_use)
r_test_use = embed_fn(r_test_use)
```

In the code above "q" and "r" refers to "question" and "response". "use" also referes to universal sentence encoding.

1.2 Feature 2

The second feature is the part of speech tagging. For this part I used the "`universal_tagset`" of the "nltk" library. First the sentences are tokenized, then the POS-tagging is applied. For example the first sentence in the questions of the training data set is of the sentences is 'Should all the blacks in the state move out?'. After pos-tagging it turns to "[('Should', 'MD'), ('all', 'PDT'), ('the', 'DT'), ('blacks', 'NNS'), ('in', 'IN'), ('the', 'DT'), ('state', 'NN'), ('move', 'NN'), ('out', 'IN'), ('?', '.')]". The I combined the word and its tag so that eg. "('Should', 'MD')" turns into `Should_MD`. The codes for this part is

```
    nltk.download('universal_tagset')
q_training_pos = {}

for i in range(len(train)):
```

```
    text = nltk.word_tokenize(train['question'][i])
    q_training_pos[i] = []
    for j in range(len(text)):
        q_training_pos[i] =  q_training_pos[i] + [nltk.pos_tag(text)[j][0] + '_' +
        nltk.pos_tag(text)[j][1]]

q_sen = list(map(list, q_training_pos.values()))
r_training_pos = {}
for i in range(len(train)):
    text = nltk.word_tokenize(train['response'][i])
    r_training_pos[i] = []
    for j in range(len(text)):
        r_training_pos[i] =  r_training_pos[i] + [nltk.pos_tag(text)[j][0] + '_' +
        nltk.pos_tag(text)[j][1]]
r_sen = list(map(list, q_training_pos.values()))
```

Then I trained a model based on the corpus I created based on these words using word2vec in gensim library.

```
    num_features = 300
model_q = word2vec.Word2Vec(q_sen,\
                            min_count=5,
                           window=5,
                           size=num_features,
                           sample=6e-4,
                           alpha=0.03,
                           workers=2)

model_q.init_sims(replace=True)

model_r = word2vec.Word2Vec(r_sen,\
                            min_count=5,
                           window=5,
                           size=num_features,
                           sample=6e-4,
                           alpha=0.03,
                           workers=2)

model_r.init_sims(replace=True)
```

Using these models I was able to vectorize the pos-tags.

2 Proposed features

2.1 Feature 1

The first proposed feature is the sentiment analysis of the question and the response. I used a unsuperviesed sentiment analysis using the package "textblob". For getting the feature I considered the difference of the polarity score of question and the polarity score of the response. The code is as

```
    from textblob import TextBlob
    # Get the polarity score using below function
    polar_tr, polar_ts = [], []
    for i in range(len(train)):
        polar_tr = polar_tr + [uf.get_textBlob_score(train['question'][i]) -

        uf.get_textBlob_score(train['response'][i])]

    for i in range(len(test)):
        polar_ts = polar_ts + [uf.get_textBlob_score(test['question'][i]) -

        uf.get_textBlob_score(test['response'][i])]
```

2.2 Feature 2

For the second feature I first cleaned data. Cleaning was removing stop-words, digits, punctuation, white spaces. Then I chose the most "n" common words in the training data-set having the same type. The variable n is 10 for "type=agreed", 20 for types "answered", "attacked" and "irrelevant". The reason I chose less for agreed is that the total number of agreed is very low. It is just 61 out of 1640 data in the training data-set. Then all these words are combined and the unique words are found which contains m words. Some of these words are "'also', 'argument', 'could', 'even', 'exactly','never',...".

Then some procedure similar to bag of words is applied. For each response a vector of the size m is defined and if the response includes a word of the bag, I assigned 1 otherwise 0. This vector is considered as the second feature I used.

```
train['r_ws'] = hero.remove_stopwords(train['response'])
train['r_ws'] = hero.clean(train['r_ws'])
# train['q_ws'] = hero.remove_stopwords(train['question'])
# train['q_ws'] = hero.clean(train['q_ws'])
agr, i_agr = [], 10
ans, i_ans = [], 20
att, i_att = [], 20
irr, i_irr= [], 20
agreed = Counter(" ".join(train[train['type']==
'agreed']["r_ws"]).split()).most_common(i_agr)
answered = Counter(" ".join(train[train['type']==
'answered']["r_ws"]).split()).most_common(i_ans)
attacked = Counter(" ".join(train[train['type']==
'attacked']["r_ws"]).split()).most_common(i_att)
irrelevant = Counter(" ".join(train[train['type']==
'irrelevant']["r_ws"]).split()).most_common(i_irr)

for i in range(i_agr):
    agr.append(agreed[i][0])
for i in range(i_ans):
    ans.append(answered[i][0])
for i in range(i_att):
    att.append(attacked[i][0])
for i in range(i_irr):
    irr.append(irrelevant[i][0])

bag = agr+ans+att+irr
bag = set(bag)
vec2_train = np.zeros((len(train),len(bag)+1))

for i in range(len(train)):
    vec2_train[i,len(bag)]=polar_tr[i]
    txt = train['response'][i]
    k=0
    for j in bag:
        if j in txt:
            vec2_train[i,k]=1
        k+=1
vec2_test = np.zeros((len(test),len(bag)+1))
for i in range(len(test)):
    vec2_test[i,len(bag)]=polar_ts[i]
    txt = test['response'][i]
    k=0
    for j in bag:
        if j in txt:
            vec2_test[i,k]=1
        k+=1
```

3 Classification

For classification I tried two different methods. The final `X_train` and `X_test` are the concatenation of all the features. For baseline it is the concatenation of pos-tagging and sentence embedding. In the proposed version my proposed feature are concatenated with the two features from the baseline.

3.1 SVM

I used the sklearn library.

```
from sklearn import svm
clf = svm.SVC()
clf.fit(X_train, y_train)
predicted = clf.predict(X_test)
from sklearn.metrics import accuracy_score, f1_score, precision_score, recall_score
print("svm accuracy", accuracy_score(y_test, predicted))
print("svm f1_score", f1_score(y_test, predicted,average='micro'))
print("svm precision_score", precision_score(y_test, predicted,average='micro'))
print("svm recall_score", recall_score(y_test, predicted,average='micro'))
```

3.2 Neural Network

For this part I used keras.

```
tf.keras.backend.clear_session()
model = tf.keras.models.Sequential([tf.keras.layers.Flatten(),
                                    tf.keras.layers.Dense(128, activation=tf.nn.relu),
                                    tf.keras.layers.Dense(64, activation=tf.nn.relu),
                                    tf.keras.layers.Dense(32, activation=tf.nn.relu),
                                    tf.keras.layers.Dense(10, activation=tf.nn.softmax)])

model.compile(loss='sparse_categorical_crossentropy',optimizer='adam',metrics=['accuracy'])
model.fit(X_train, y_train, epochs=10)
model.evaluate(X_test, y_test)
predicted = model.predict_classes(X_test)
print(classification_report(y_test, predicted))
```

4 Results

In the following table the results (Accuracy, F1, Precision and Recall) of the baseline features and baseline features + proposed features are summorized.

		Accuracy	F1	Precision	Recall
Baseline Features	SVM	0.8049	0.8049	0.8049	0.8049
Baseline Features	NN	0.77	0.74	0.72	0.77
Proposed and baseline Features	SVM	0.793	0.793	0.793	0.793
Proposed and baseline Features	NN	0.66	0.69	0.77	0.66

As it can be seen SVM is doing a better job than the NN. The reason is that our data-set is not very big. NN needs a big data-set. In smaller data-set SVM does a better job.

About the results of different features, it seems the feature I added to the baseline does not change the results much. I am guessing the reason is that the new features are not adding any new information to the previous features. Maybe if I tried word embedding instead of sentence embedding the proposed features made the results better. But the google's universal sentence embedding is creating more comprehensive features.

So my proposed method would be using the baseline feature with SVM classification method.