Natural Language Inference: Modeling strategies

Christopher Potts

Stanford Linguistics

CS224u: Natural language understanding
Hand-built features
### Hand-built feature ideas

<table>
<thead>
<tr>
<th>Hand-built features</th>
<th>Sentence-encoding</th>
<th>Chained</th>
<th>Other strategies</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. Word overlap
2. Word cross-product
3. Additional WordNet relations
4. Edit distance
5. Word differences (cf. word overlap)
6. Alignment-based features
7. Negation
8. Quantifier relations (e.g., every ⊏ some; see MacCartney and Manning 2009)
9. Named entity features
Hand-built feature ideas

1. Word overlap
Hand-built feature ideas

1. Word overlap

2. Word cross-product
Hand-built feature ideas

1. Word overlap
2. Word cross-product
3. Additional WordNet relations
Hand-built feature ideas

1. Word overlap
2. Word cross-product
3. Additional WordNet relations
4. Edit distance
Hand-built feature ideas

1. Word overlap
2. Word cross-product
3. Additional WordNet relations
4. Edit distance
5. Word differences (cf. word overlap)
Hand-built feature ideas

1. Word overlap
2. Word cross-product
3. Additional WordNet relations
4. Edit distance
5. Word differences (cf. word overlap)
6. Alignment-based features
Hand-built feature ideas

1. Word overlap
2. Word cross-product
3. Additional WordNet relations
4. Edit distance
5. Word differences (cf. word overlap)
6. Alignment-based features
7. Negation
Hand-built feature ideas

1. Word overlap
2. Word cross-product
3. Additional WordNet relations
4. Edit distance
5. Word differences (cf. word overlap)
6. Alignment-based features
7. Negation
8. Quantifier relations (e.g., every ⊆ some; see MacCartney and Manning 2009)
Hand-built feature ideas

1. Word overlap
2. Word cross-product
3. Additional WordNet relations
4. Edit distance
5. Word differences (cf. word overlap)
6. Alignment-based features
7. Negation
8. Quantifier relations (e.g., every ⊆ some; see MacCartney and Manning 2009)
9. Named entity features
Sentence-encoding models
Distributed representations as features

A classifier of some kind (learned)

e.g., concatenation, difference (not learned)

Embedding look-up

Sentence-encoding

Chained

Other strategies
Code: Distributed representations as features

```python
[1]: import numpy as np
    import os
    from sklearn.linear_model import LogisticRegression
    import nli, utils

[2]: SNLI_HOME = os.path.join("data", "nli_data", "snli_1.0")
    GLOVE_HOME = os.path.join('data', 'glove.6B')

[3]: glove_lookup = utils.glove2dict(
    os.path.join(GLOVE_HOME, 'glove.6B.50d.txt'))

[4]: def _get_tree_vecs(tree, lookup, np_func):
    allvecs = np.array([lookup[w] for w in tree.leaves()] if w in lookup)
    if len(allvecs) == 0:
        dim = len(next(iter(lookup.values())))
        feats = np.zeros(dim)
    else:
        feats = np_func(allvecs, axis=0)
    return feats

[5]: def glove_leaves_phi(t1, t2, np_func=np.sum):
    prem_vecs = _get_tree_vecs(t1, glove_lookup, np_func)
    hyp_vecs = _get_tree_vecs(t2, glove_lookup, np_func)
    return np.concatenate((prem_vecs, hyp_vecs))

[6]: def glove_leaves_sum_phi(t1, t2):
    return glove_leaves_phi(t1, t2, np_func=np.sum)
```
**Code: Distributed representations as features**

```
[7]: def fit_softmax(X, y):
    mod = LogisticRegression(
        fit_intercept=True, solver='liblinear', multi_class='auto')
    mod.fit(X, y)
    return mod

[8]: glove_sum_experiment = nli.experiment(
    nli.SNLITrainReader(SNLI_HOME),
    glove_leaves_sum_phi,
    fit_softmax,
    assess_reader=nli.SNLIDevReader(SNLI_HOME),
    vectorize=False) # We already have vectors!
```
Rationale for sentence-encoding models

1. Encoding the premise and hypothesis separately might give the model a chance to find rich abstract relationships between them.

2. Sentence-level encoding could facilitate transfer to other tasks (Dagan et al.’s (2006) vision).
Sentence-encoding RNNs

$h_3$ and $h_0$ should be good sentence representations.
PyTorch strategy: Sentence-encoding RNNs

The full implementation is in nli_02_models.ipynb.

**TorchRNNSentenceEncoderDataset**
This is conceptually a list of pairs of sequences, each with their lengths, and a label vector:

\[
\left( \text{[every, dog, danced], [every, poodle, moved]}, (3, 3), \text{entailment} \right)
\]

**TorchRNNSentenceEncoderClassifierModel**
This is conceptually a premise RNN and a hypothesis RNN. The forward method uses them to process the two parts of the example, concatenate the outputs of those passes, and feed them into a classifier.

**TorchRNNSentenceEncoderClassifier**
This is basically unchanged from its super class TorchRNNClassifier, except the predict_proba method needs to deal with the new example format.
Sentence-encoding TreeNNs

\[
\begin{align*}
    p_B &= f([p_A; x_1]W + b) \\
    p_D &= f([p_C; x_4]W + b) \\
    p_A &= f([x_3; x_4]W + b) \\
    p_C &= f([x_3; x_5]W + b)
\end{align*}
\]

Leaf nodes are looked up in the embedding.
Chained models
Simple RNN

Recurrent architectures: simple classifiers

\( y \)

\( h_0 \rightarrow h_1 \rightarrow h_2 \rightarrow h_3 \rightarrow h_4 \rightarrow h_5 \rightarrow h_6 \)

\( W_{hh} \)

\( W_{xh} \)

\( W_{hy} \)

every dog danced every poodle moved

\( x_3 \rightarrow x_2 \rightarrow x_1 \rightarrow x_3 \rightarrow x_5 \rightarrow x_4 \)

every dog danced every poodle moved

\( W_{xh} \)

\( W_{hy} \)
Rationale for chained models

1. The premise truly establishes the context for the hypothesis.

2. Might be seen as corresponding to a real processing model.
**Code snippet: Simple RNN**

```python
[1]: import os
    from torch_rnn_classifier import TorchRNNClassifier
    import nli, utils

[2]: SNLI_HOME = os.path.join("data", "nli_data", "snli_1.0")

[3]: def simple_chained_rep_rnn_phi(t1, t2):
    return t1.leaves() + ['[SEP]'] + t2.leaves()

[4]: def fit_simple_chained_rnn(X, y):
    vocab = utils.get_vocab(X, n_words=10000)
    vocab.append('[SEP]')
    mod = TorchRNNClassifier(vocab, hidden_dim=50, max_iter=50)
    mod.fit(X, y)
    return mod

[5]: simple_chained_rnn_experiment = nli.experiment(
    nli.SNLITrainReader(SNLI_HOME, samp_percentage=0.10),
    simple_chained_rep_rnn_phi,
    fit_simple_chained_rnn,
    vectorize=False)
```
Premise and hypothesis RNNs

The PyTorch implementation strategy is similar to the one outlined earlier for sentence-encoding RNNs, except the final hidden state of the premise RNN becomes the initial hidden state for the hypothesis RNN.
Other strategies

**TorchRNNClassifier**

- TorchRNNClassifier feeds its final hidden state directly to the classifier layer.
- If bidirectional=True, then the two final states are concatenated and fed directly to the classifier layer.

**Other ideas**

- *Pool* all the hidden states with **max** or **mean**.
- Different pooling options can be combined.
- Additional layers between the hidden representation (however defined) and the classifier layer.
- Attention mechanisms