Exploring Data

I started off exploring the dataset, getting a bunch of key information about the dataset structure and statistical information, including:

- Number of training examples = 405
- Sentence length varies massively from 1 to > 60 tokens
- # of simple labels = 3 and # of extended labels = 12
- Vocab size is 766 (or 769 if punctuation e.g. '?' are included).

To summarise, the vocab size is relatively large for the dataset (with roughly 250 tokens only appearing once in the training set). But the far bigger issue seems to be not in the simple classification case, but in the extended classification case, even if the examples were distributed uniformly among the 12 labels, there would only be 34 examples per label, which is a very small training set given the diversity in sentence structure, length, and tokens.

Model Design

With the above knowledge in mind I started designing models that I thought might tackle the problem well. I considered starting with a standard LSTM classifier approach (achieve outstanding results on a very wide range of NLP tasks), but decided there would likely be a couple of problems with this approach. Firstly, LSTMs in general, even if the input embeddings are fixed, have a large number of parameters, and this will always be a problem with such a small dataset. It should always be possible to use this and classify the training dataset well, but training such a large number of parameters to generalise successfully would require more training data. And secondly, after inspecting the actual sentences of the training data it was clear that either using or fine-tuning a pre-trained LSTM would be tough, as the language is conversational and strikingly different to most standard text corpuses used in pretraining. NB: I did end up implementing an LSTM encoder and associated classifier to test these predictions, and indeed the simpler approaches outlined below proved more successful.

So I decided to start with encoding each example using pre-trained word embeddings, considering a number of different word-embedding techniques and sources, and then designing different classification models on top of these encodings. I tested fasttext and glove embeddings trained on standard wiki corpuses and twitter (although somewhat surprisingly the wiki embeddings tended to do better). I didn't update these embeddings at all in this process, once again due to the size of this dataset.

The first classification method I tried was a simple feedforward classifier (affine operations and non-linearity) with customisable layer size, depth, and dropout. To encode the sentences for this (because of their variable length) I tried both max and average pooling of the constituent word embeddings, both yielding very similar success. However, the best results I achieved with the word embedding approaches came from using a pooling classifier that concatenated both these types of pooling as well as the word embedding of the agent-type (perhaps the key component that I overlooked early on, as it seems intuitive from inspecting the data that sentences and word-usage was considerably different for doctor and patient). This latter classification method peaked just over an 80% (results outlined in more detail below) on the simple classification task.

The last major model I tried, and to my regret as I jumped into the word embedding process too quickly before considering more classical approaches, I decided to try a Bag-of-Words (BoW) approach to encoding the sentence in a one-hot vector. The intuition for why this approach might be more powerful despite having less representational power than embeddings is that for the given classification task (where the classes are high-level types of sentences), there are likely key words that indicate each class. One other key addition to this was that I stopped preprocessing the sentences to remove punctuation (initially was done from habit, before I reduced the preprocessing step to a simple tokenisation using Spacy). I used the simple feedforward classifier taking as input this one-hot BoW vector, and the inclusion of punctuation tokens in the vocab improved the classifiers performance.

Finally, I didn't implement any batching here, purely because that would require a different batching scheme for each model encoder and would've taken more time. And additonally, the small size of the dataset meant there wouldn't be much computational advantage, although there is a chance batch updates might have given a generalisation advantage to the optimisation process. The LSTM model was fairly slow to train as a result of processing inputs one by one.

Creating Development Environment

Before delving into experiments and results, I'll briefly discuss the development environment and code-base I set up. I set up the development environment in a very modular fashion, enabling rapid iteration, development and testing of different models and encoder-classifier combinations. I have had experience with text processing before, and personally love breaking the data processing step into data readers and data iterators that can be used to iterate through training or testing data (with fine-tuned control allowed by optional argument usage).

The experimental setup was controlled entirely through main.py and parsing command line arguments, with directory structures set up to manage data and trained pytorch models (I

used pytorch because I have the most experience with it and it fits nicely with such a modular development environment). A ModelWrapper class in main was what I used to represent a complete classifier, nicely handing training, classification, testing, and saving/loading of pytorch models into this framework. Note that this class takes in a data iterator and uses it to calculate loss and accuracy on the test dataset every epoch. However, this was used to construct informative visualisations for the purposes of this report, not as a development by which to select an optimal model.

Beyond this, additional files include a text pre-processing file that handles tokenisation and vocab-creation for a given dataset, and a visualisation file for creating simple 2d plots (which I used to graph loss and accuracy).

Usage instructions to replicate these experiments are outlined in the README file in the root directory, and dependencies are specified in requirements.txt.

Experiments

As is standard for text classification tasks, I used Cross-Entropy Loss for all classifiers and training runs. The hyper-parameters and design choices involved in discovering the most effective classifiers were therefore:

- Encoder
  - BoW or word embedding
  - if the latter, which word embedding source to use
  - Encoding mechanism (to construct same-size inputs to classifier given variable sentence lengths).
- Classifier
  - type of classifier
  - layer structure
  - dropout structure
- Optimiser
  - choice of optimisation scheme
  - optimiser parameters
- Number of training epochs.

Due to lack of a development set for optimising these hyper-paramaters (and I thought about partitioning the training set into a training and dev set, but decided against due to the already small number of training examples), I wasn't able to perform an entirely complete, quantitative search (e.g. grid-search) for the optimal combination of the above hyper-params. However, for each intuitive combination of encoder and classifier, I ran a number of tests to determine the optimisation settings that seemed to achieve good results, and to report classification accuracies as outlined below I repeated the training run for any given hyper-parameter setup and averaged the results.

I used a number of functions to facilitate the running of these tests.

- *model_accuracy*: Calculates the average classification accuracy of a model on the test dataset over n trials.
- *run_trials*: A more complete version of the above function. Runs n trials and stores checkpoints for the model at each epoch, along with overall statistics for the model and visualisations (Once again, this was used only for visualisation and understanding purposes in analysing and writing this report, not for model selection).

This experiment proved tricky, in that the loss curve on the training dataset for most models was an almost continuously decreasing function approaching accuracy of 100% (but obviously if this was used for a measure of convergence then we'd get massive overfitting). Additionally, decreasing learning rate schemes didn't have any noticeable improvement on model training, and therefore selection of the parameters for the two optimisation methods I tested (SGD and Adam) simple involved selecting a rate for the entire training run. I experimented with layer sizes based on the general knowledge that too many parameters would not be able to learn a generalised classification function well enough on such a small dataset, and for the same reason I experimented with a range of dropout values from 0.1 to 0.7 for each layer.

Accuracies for specific models with hyperparameter settings are noted in the results section below.

Results

The results below demonstrate a good portion of the model testing process I underwent. This was done largely manually although it could easily be automated. The best performing versions of each model type are bolded (I excluded the use of simple average or max pooling of word embeddings as this was a precursor to the comprehensive pooling_classifier model, and performed worse for almost every parameter setting).

Word Embeddings	Layers	Drops	Optim	LR	#Epochs	Accuracy
FastText	[900,300,50,3]	[0,0,0]	SGD	0.01	20	0.776
FastText	[900,300,50,3]	[0,0,0]	SGD	0.001	20	0.619
FastText	[900,300,3]	[0,0]	SGD	0.01	20	0.795
FastText	[900,300,3]	[0.1,0.1]	SGD	0.01	20	0.793
FastText	**[900,200,3]**	**[0,0]**	**SGD**	**0.01**	**20**	**0.796**
FastText	[900,200,3]	[0.2,0.2]	SGD	0.01	20	0.799
FastText	[900,100,3]	[0.2,0.2]	SGD	0.01	20	0.790
GloVe	[150,300,3]	[0,0]	SGD	0.01	20	0.785
GloVe	[150,300,3]	[0,0]	SGD	0.01	20	0.785
GloVe	**[150,100,3]**	**[0,0]**	**SGD**	**0.01**	**20**	**0.797**
GloVe	[150,100,3]	[0.2,0.2]	SGD	0.01	20	0.788
GloVe	[150,75,3]	[0,0]	SGD	0.01	20	0.789

Table 1: WE-Pooling Model for Simple Classification

The models are named as follows: $WE-Pooling_x$, in Table 1 and 2, is a basic encoder (that just passes on input given that the input is the word embeddings of the sentence and agent-type) followed by the pooling classifer (the pooling and concatenation method is outlined in the model section above), where the x subscript refers to the pretrained word embeddings used; BoW, in Table 3 and 4, is a bag-of-words encoder with basic classifier; and $LSTM$, in Table 5, is an LSTM encoder followed by rnn pooling classifier (similar to the pooling classifier by using LSTM output states). I only used glove-50 embeddings for this model due to computational cost of training the model. I didn't show results for LSTM on the extended classification task because it performed so poorly.

In the column headings: layers refers to the specification of classifier layer sizes, drops refers to the dropout values for those layers, and hidden size refers to the LSTM hidden size in the LSTM model; the rest should be self-explanatory.

Firstly, just some key observations. I noted during training of the various models (before my testing pipeline was established) that with all models and optimisation schemes used, the training loss converged in a fairly small number of epochs (if at all, which sometimes didn't depending on model or hyperparameters). This is why I settled on these hardcoded number of epochs for the above results demonstration. Once again, the ModelWrapper class has the capability to save checkpoints at any epoch, and in a real application setting, any of these

Word Embeddings	Layers	Drops	Optim	LR	#Epochs	Accuracy
FastText	[900,300,12]	[0,0]	SGD	0.01	20	0.313
FastText	**[900,100,12]**	**[0,0]**	**SGD**	**0.01**	**20**	**0.326**
FastText	[900,100,12]	[0.5,0.5]	SGD	0.01	20	0.259
FastText	[900,100,12]	[0,0]	SGD	0.001	20	0.296
GloVe	[150,300,12]	[0,0]	SGD	0.01	20	0.220
GloVe	**[150,500,12]**	**[0,0]**	**SGD**	**0.01**	**20**	**0.267**
GloVe	[150,500,200,12]	[0,0,0]	SGD	0.01	20	0.215
GloVe	[150,500,12]	[0.2,0.2]	SGD	0.01	20	0.244

Table 2: WE-Pooling Model for Extended Classification

Layers	Drops	Optim	LR	#Epochs	Accuracy
[769,300,3]	[0,0]	SGD	0.01	20	0.838
[769,300,3]	**[0.7,0.7]**	**SGD**	**0.01**	**20**	**0.841**
[769,50,3]	[0.7,0.7]	SGD	0.01	20	0.834
[769,500,3]	[0.7,0.7]	SGD	0.01	20	0.823
[769,300,3]	[0,0]	Adam	0.001	10	0.813
[769,300,3]	[0,0]	Adam	0.01	10	0.801
[769,300,3]	[0.7,0.7]	Adam	0.01	10	0.807

Table 3: BoW Model for Simple Classification

Layers	Drops	Optim	LR	#Epochs	Accuracy
[769,300,12]	[0.7,0.7]	SGD	0.01	20	0.400
[769,300,12]	[0.7,0.7]	SGD	0.001	20	0.363
[769,300,12]	[0,0]	SGD	0.01	20	0.412
[769,500,12]	[0,0]	SGD	0.01	20	0.400
[769,500,12]	[0,0]	SGD	0.01	40	0.415
[769,50,12]	**[0,0]**	**SGD**	**0.01**	**20**	**0.420**
[769,100,12]	[0,0]	SGD	0.01	20	0.400

Table 4: BoW Model for Extended Classification

Hidden Size	Layers	Drops	Optim	LR	#Epochs	Accuracy
100	[150,50,3]	[0,0]	Adam	0.001	10	0.446
100	[150,50,3]	[0,0]	Adam	0.001	20	0.455
150	[150,50,3]	[0,0]	Adam	0.001	10	0.427
100	**[150,300,3]**	**[0,0]**	**Adam**	**0.001**	**10**	**0.554**
100	[150,500,3]	[0,0]	Adam	0.001	10	0.500

Table 5: LSTM Model for Simple Classification

checkpoints of model weights could be used or combined to find the best possible classifier; hence why choosing an epoch number that may not be optimal is fine for this setting. Additionally, in analysing the test loss across epochs after finishing the analysis given in these tables, test loss also appears to converge in a similar timeframe (provided learning rate is not too large, in which case we get overfitting and increasing test loss over time).

And secondly, notice there is no inclusion of the Adam optimiser for the WE-Pooling model, and no use of the SGD optimiser for the LSTM model; both these exclusions are because these optimisers struggled to converge in these settings. Obviously the above does not constitute a comprehensive grid-search across all possibilities, but it does a good job of demonstrating some key results (this was largely in the interest of time).

As the BoW model outperformed the others on both classification tasks, I trained a a classifier for both classification tasks and used these to label the output CSV with classes and confidence values.

Analysis

Clearly the BoW model performs the best on the simple classification task, with the best versions of the WE-Pooling model achieving somewhat comparable results. My intuition is that this is due to the extra complexity that a word embedding model adds in juxtaposition with the small size of the dataset. And furthermore, word embeddings attempt to capture a semantic representation of words, and hence the Pooling method uses a semantic representation of the input sentence in order to make its classification prediction. However, due to the fact that the labels in this setting refer to sentence types, it makes sense that a BoW approach would prevail, as there are likely a select few indicator words that correlate highly with a particular class. The BoW model learns these, and these seem more generalisable than a map from semantic representation to classification that might struggle with different topics in the test set. The LSTM model struggles largely due to a lack of data, and perhaps from similar issues in terms of the semantic representations it learns as outlined above.

Somewhat satisfyingly, the BoW model is the simplest implementation and the most time efficient, and therefore would provide a strong classifier in real application. If combined with more background knowledge on the particular task (e.g. syntactic ideas, key words that indicate sentence types), this model could probably achieve more success.

Performance on the extended dataset was very poor across the board, likely due to a significant lack of data. The differences between variants of a sentence type (e.g. ContextQuestion and MultiChoiceQuestion) are very subtle, and consequently they are incredibly difficult for a model to pickup, especially with such few labelled examples for each extended label.

I suspect that far superior performance could be achieved through one of two means. Firstly, incorporating more background knowledge on sentence structure or co-occurrence of key words with sentence types would likely result in a stronger model (e.g. an approach incorporating NaiveBayes).

And secondly, I suspect the classification problem within subsets of a sentence type would be significantly easier, for example, given a list of only those sentences labelled as Questions, predicting whether they are Binary, MultiChoice, Context etc. Therefore, the model I would try next would be one that uses the best trained simple classifier to classify sentences, and then the extended classifier would predict the extended label subset based on that. (Kind of like incorporating a high-level sentence type as a prior to classifying more fine-grained labels). Explicit inclusion of rules like a question mark indicates a question would also likely increase performance.