Daniel Peek qer419

Michael Canas ohh135

CS 3793 Final Project Report

12/4/18

---Introduction and setup---

The project consists of two approaches to part of speech tagging, one neural network and one with logic. It requires Python 3.5+, works only on Linux, and requires nltk and gensim to be installed. It requires a pre-trained Word2Vec model to convert words into vector input for the neural network. Our program expects that model to be an uncompressed binary file named trained-model.bin in the same directory as main.py. We used a slimmed down version of a Google News model, found here zipped in .gz format:

https://github.com/eyaler/word2vec-slim

A different model could also work but might not have all the words needed for our set of input sentences. For input, we are using the set of 720 Harvard sentences, found here: https://www.cs.columbia.edu/~hgs/audio/harvard.html

The first 650 sentences are used as the training set and the remaining 70 are the testing set. We had to fix a few spelling errors that were in the version found on the website.

---Program usage---

Neural net approach:

python3 main.py [-t] [-i INPUT] [-o OUTPUT] [-e EPOCHS]

If -t or --test is specified, the neural network will run on the testing data and make no updates to the weights. Testing requires that an existing file of weights be loaded in using the

"-i example.file" option. A weights file can be created by not specifying -t and including a

"-o example.file" option. On a training run, it will use 1000 epochs by default, which can be changed with the "-e ####" option. On a testing run, the -o and -e options have no effect.

Logical approach:

python textTag.py

It opens the sentences file, and prints the tags to the console.

---main.py Description---

We used argparse to handle all the command line arguments. Then, the Word2Vec model is loaded using the gensim library. If no input is provided, the neural networks are created with random weights. There are 10 neural networks, one for each possible word in the sentences. The sentences file is read into a list and modified to remove punctuation and a few words we are not attempting to parse. After splitting the list into the training and testing sets, everything is ready for the neural network to get started.

For training, it will run the selected number of epochs. In each epoch, it reads one random sentence from the list and passes it to the run() function. There, the sentence is passed to textTag.py, where each word is given a part of speech tag. Each word in the untagged sentence is then assigned a 300-length vector from the Word2Vec model. Each of those vectors goes through a separate neural network. There are 5 outputs, which correspond to noun, verb, adjective, adverb, and preposition. The output with the highest activation value is compared to the assigned tag to determine accuracy. The weights are then adjusted with backwards propagation. An update on the training data accuracy is printed every 100 epochs.

For testing, the program runs through the second list of sentences in order, does not adjust weights, and does not print intermediate accuracy results. Otherwise, it behaves identically.

File operations are done with the pickle library to easily read in or write out the entire list containing 10 neural networks.

---neural\_net.py description---

This file contains the neural network class and related functions. It is adapted from the code provided by Dr. O’Hara for the blackjack assignment. It is changed to have a set number of inputs, hidden nodes, and outputs. The inputs are always 300, because of the size of Word2Vec vectors. There are 20 hidden nodes, and 5 outputs. Other than those changes, the neural network behaves the same way it did for assignment 4.

---textTag.py description---

The textTag.py uses the nltk.corpus library with wordent imported to generate dictionaries for adverbs,verbs,prepositions,nouns, and adjectives. The programs reads in a file of sentences and splits each sentece into word tokens. It then tags each word accordingly in context to its position in the sentence to one of the dictionaries. The tagging is done by logic. Specifically, the algorithm will check for a set of patterns such as, “article,noun,verb”, throughout the sentence to tag words more accurately.

---Problems encountered---

The largest issue we had was encoding word into a form that could be input into a neural network. Originally, we had wanted to do the code for it ourselves, but it was a much larger task than we had assumed. One-hot encoding seemed the most plausible, but we couldn’t get the neural network to train with it. Eventually, we chose to use Word2Vec with a pre-trained model. This assigns a vector of 300 floating point values to each word, which approximates its meaning in a computer-usable way. Once we switched over, the training began making progress.

Word2Vec brought in its own problem in that a 300-input neural network takes a long time to train, especially since we have 10 of them. A partially connected network might have worked better, but we couldn’t come up with a way to choose which parts of the vector correspond to which hidden nodes. Since we don’t know how all the numbers in a vector relate to each other, we were worried that breaking it apart for partial connectivity might destroy the meaning it carries.

We also would have liked to try using larger numbers of hidden nodes. Coming from 300 inputs, 20 hidden seems like it might be too few. However, 20 worked as a compromise for the sake of speed and was ultimately effective.

Tagging text with logic in textTag.py has some issues as well. With logic we were only limited to simple sentences as found in the Harvard Sentences file. Issues mainly arose when trying to label words that could be tagged as multiple lines of speech. For example, the word canoe can be used as a noun and also a verb so a simple dictionary look up would not be sufficient. Using pattern matching to see the words context in the sentece seemed to fix this issue but could see problems if given a more complicated and longer sentence.

---Results and conclusion---

After training the network for about 200000 epochs, taking roughly 6 hours, we had gotten to the following accuracy:

End of training: epoch 199990

76.7232% correct

Testing: 78.0679% correct

These are not quite perfect results, but they certainly demonstrate that the network was learning. With 5 outputs, we could expect to get ~20% accuracy without training. Our network almost quadrupled that percentage on the testing data, which was not trained directly. Also, the training could be continued: at epoch 150000, the training accuracy was only about 75%. The learning process is slow, but given time, this neural network could improve further. More samples of output are in the included file final-project-output.txt

This was a fun and challenging project. It required both of us to use some new python tools and libraries related to language. We probably could have gotten better results from the neural network by using a higher-level library like Keras to take care of the difficult stuff, but we learned more doing it this way. Attempting a logical approach to the problem certainly helped us appreciate neural networks.