Tflearn_chatbot

December 23, 2017

1 Chatbot using `tflearn` and `tensorflow`

1.1 Introduction

This is a pretty basic chatbot using tensorflow and `tflearn`. Basic data classification into *intents*. After identifying the intent of a query, an appropriate response can be generated. This document will cover all the basics I've gathered around the internet, but mainly from this article. Each intent will have these attributes: - A tag identifier - Patterns (ways in which the input will/can be modeled - Responses (ways in which our bot will respond)

```
In [1]: # import the basics required

        # parsing the data
        import numpy as np

        # libraries required for learning
        import tflearn
        import tensorflow as tf
        tf.reset_default_graph()
```

1.2 Parsing the data and making the corpus

The first step is to classify our intents and get the *preprocessing* out of the way. Our intent patterns have to be "stemmed", i.e, words which convey the same base, have to map to the same pattern for a given intent. *For example*, having, have, etc should all map to have, since it is the **common base**.

```
In [151]: # import the data/intents
          import json
          intent_file = "intents.json"
          intents = json.load(open(intent_file))
          for intent in intents:
              try:
                  current_context = intent['context_filter']
                  current_context.append(intent['tag'])
              except KeyError:
                  current_context = [""]
              intent['context_filter'] = list(set(current_context))
```

```
                try:
                    future_context = intent['set_context']
                except KeyError:
                    future_context = [""]
                intent['set_context'] = list(set(future_context))
            json.dump(intents, open(intent_file, 'w+'))
            print(json.dumps(intents[-2:], indent=2, sort_keys=True), "...")
```

```
[
  {
    "context_filter": [
      "rentquery",
      "rental"
    ],
    "patterns": [
      "Can we rent a moped?",
      "I'd like to rent a moped",
      "How does this work?"
    ],
    "responses": [
      "Are you looking to rent today or later this week?"
    ],
    "set_context": [
      "rentalday",
      "today"
    ],
    "tag": "rental"
  },
  {
    "context_filter": [
      "rentalday",
      "today"
    ],
    "patterns": [
      "today"
    ],
    "responses": [
      "For rentals today please call 1-800-MYMOPED",
      "Same-day rentals please call 1-800-MYMOPED"
    ],
    "set_context": [
      "rentquery"
    ],
    "tag": "today"
  }
] ...
```

Once we have loaded the basic data, we have to stem it, and separate it into *classes* (essentially intent-identifiers), optionally ignoring common words like articles (the, are, etc). Do note, that pronouns might seem redundant or unecessary but more often that not they are important to indentify the target for the response.

```
In [3]: # import the necessary language processing tools
        import nltk
        nltk.download('punkt')
        from nltk.stem.lancaster import LancasterStemmer
        stemmer = LancasterStemmer()
```

```
[nltk_data] Downloading package punkt to /home/pratik/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

```
In [4]: words, classes, documents = set(), set(), []
        # usually stop words should/can be removed, but phrase searching becomes difficult
        # can remove stop words for large datasets
        ignore_words = set(['?'])
        for intent in intents:
            # add the relevant class
            classes.add(intent['tag'])
            # stem the words in the patterns and add it to the document set
            for pattern in intent['patterns']:
                current_words = [stemmer.stem(_.lower()) for _ in nltk.word_tokenize(pattern) i
                [words.add(_) for _ in current_words]
                documents.append((current_words, intent['tag']))
        classes = list(classes)
        print("words : ", words, "\n\nclasses : ", classes, "\n\ndocuments : ", documents[:5],
```

```
words :  {'thank', 'kind', 'lik', 'what', 'doe', 'hour', "'s", 'which', 'day', 'do', 'i', 'a',

classes :  ['hours', 'goodbye', 'thanks', 'mopeds', 'greeting', 'payments', 'opentoday', 'toda

documents :  [(['hi'], 'greeting'), (['how', 'ar', 'you'], 'greeting'), (['is', 'anyon', 'ther
```

Once we have our words stemmed and classified into documents, we can start training the model. First, we have to convert the data into a tensor (this is mainly done for speed and compatibility, you'd have to code a network from scratch/use another package to work with raw data). The bag here basically is a vector that contains the truth value of whether the word in our words is present in the given pattern. *for example*, if my complete wordset is `["how", "are", "you", "doing"]` and my current pattern is `["how"]`, then my word *bag* will be `[1, 0, 0, 0]`.

```
In [5]: dataset = []
        y_labels = []
        import random
        for pattern, tag in documents:
            bag = [ 1 if _ in pattern else 0 for _ in words]
            y_label = [0] * len(classes)
            y_label[classes.index(tag)] = 1
            dataset.append((bag, y_label))
        # shuffle the dataset to eliminate biases
        random.shuffle(dataset)
        print("Training examples : ", len(dataset), "\nx_i : ", len(dataset[0][0]), "\ny_i : "
```

```
Training examples :  27
x_i :  48
y_i :  9
```

1.3 Building the model

Now that our basic dataset is ready, all that's left is to train the model on it. Given that we don't have a lage number of classes and only a few training examples, this step wouldn't take a lot of time. But for large corpuses, this can be the longest step, so make sure you have *good* hardware available.

```
In [6]: # Build neural network
        training_ex = random.choice(dataset)
        x_i, y_i = training_ex
        net = tflearn.input_data(shape=[None, len(x_i)])
        net = tflearn.fully_connected(net, 8)
        net = tflearn.fully_connected(net, 8)
        net = tflearn.fully_connected(net, len(y_i), activation='softmax')
        net = tflearn.regression(net)

        # define the network model and log output
        model = tflearn.DNN(net, tensorboard_dir='tflearn_logs')
```

Given the very little number of training examples, it would make no sense to split the data into *training* and *test*. One could account the model's accuracy to overfitting, but given the shallow nature of the model, this is probably not the case. First, let us save the data and the model before proceeding further.

```
In [7]: import pickle
        model_data = {
            'dataset' : dataset,
            'words' : words,
            'classes' : classes,
        }
        # save data
        pickle.dump(model_data, open('model_data.pkl', 'wb'))
        # save the model
        model.save('untrained_model')
```

```
INFO:tensorflow:/home/pratik/0Coding/93Machine-Learning/4Chat-Bot/tflearn_chatbot/untrained_mo
```

1.4 Training the model

```
In [8]: # gather the data
        train_X, train_y = np.array([_[0] for _ in dataset]), np.array([_[1] for _ in dataset]
        print(train_X.shape, train_y.shape)
        iterations = 4000
        batch = 9
        model.fit(train_X, train_y, n_epoch=iterations, batch_size=batch, show_metric=True)
```

```
Training Step: 11999  | total loss: 0.19336 | time: 0.008s
| Adam | epoch: 4000 | loss: 0.19336 - acc: 0.9840 -- iter: 18/27
Training Step: 12000  | total loss: 1.38208 | time: 0.012s
| Adam | epoch: 4000 | loss: 1.38208 - acc: 0.8967 -- iter: 27/27
--
```

```
In [9]: # we can save the trained model now
        model.save('trained_model')
```

```
INFO:tensorflow:/home/pratik/0Coding/93Machine-Learning/4Chat-Bot/tflearn_chatbot/trained_mode
```

1.5 Setting up the chatbot

Now that our model has successfully been trained, we can setup the *actual* chatbot framework. This would involve cleaning user inputs and modeling them into bags, and then predicting the intent using our model.

1.5.1 Wrapping the model

The first step we need to cover is wrapping the `model.predict` such that I can create a black box, which, when given a sentence as input, returns the possible intent.

```
In [10]: # function to return the bag of words given an input sentence
         def getXlabel(sentence):
             stemmed_words = [stemmer.stem(_.lower()) for _ in nltk.word_tokenize(sentence) if
             return np.array([ [1] if _ in stemmed_words else [0] for _ in words]).T
         # example
         ex_sentence = "How are you today?"
         x_label = getXlabel(ex_sentence)
         print(x_label.shape, x_label)
```

```
(1, 48) [[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 1 0 0 0 0
  0 0 0 0 0 0 0 0 0 1 0]]
```

```
In [11]: sentence = "How are you today?"
         model.predict(getXlabel(sentence))
```

```
Out[11]: array([[  6.54070007e-08,   6.09317445e-04,   8.89364421e-01,
                   1.30644266e-05,   3.34204957e-02,   8.52505707e-07,
                   4.98453915e-07,   1.22153084e-04,   7.64690563e-02]], dtype=float32)
```

This returns the probability of the given setence having a particular intent. Let us wrap the prediction in a function that returns *either* a given intent or `None` depending on the probability. The function I've modeled sets a threshold above which only the sentence is considered. Optionally, you can set `show_probability` to `True`, to get the probability of the return intent.

```
In [25]: def getIntent(sentence, intents=model_data['classes'], threshold=0.7, show_probability
             intent = None
             x_label = getXlabel(sentence)
             prediction = np.ndarray.tolist(model.predict(x_label)[0])
             # if no intent matches above the threshold
             if all([_ < threshold for _ in prediction]):
                 if show_probability:
                     return None, 0
                 return None
             try:
                 assert(len(intents) == len(prediction))
             except AssertionError:
                 print("Number of intents do not match predictions")
                 return False
             intent = intents[prediction.index(max(prediction))]
             if show_probability:
                 return intent, prediction[prediction.index(max(prediction))]
             return intent
```

```
In [22]: # example intent that does not exist in the corpus
         getIntent("hi, how are you doing today?", show_probability=True, threshold=0.8)
```

```
Out[22]: ('greeting', 0.8824049234390259)
```

1.5.2 Contextual Awareness

Now that we have a wrapper, it's time to move our focus onto the *contextual awareness* of our bot. Contextual awareness implies the steps the bot should take in order to move the context forward (in this situation, the end goal would be do have a successful booking done.

```
In [38]: # function to generate a response on the basis on input
         min_prob = 0.8 # set this according to your results
         intent_responses = {}
         for _ in intents:
             intent_responses[_['tag']] = _['responses']
         def getResponse(sentence):
             intent, prob = getIntent(sentence, threshold=min_prob, show_probability=True)
             if intent is None:
                 return ("I'm having a hard time understanding, "
                         "could you please rephrase that")
```

```
            return random.choice(intent_responses[intent])
        getResponse("Hi, how are you today?")
```

```
Out[38]: 'Hi there, how can I help?'
```

The next step is to add a way for the conversation to continue forward. We can do this by linking the intents to one another. *For example*, if a user enters the keyword `today`, we have to return a response relevant in this context, basically, return work timings, **even if** the probability of that (*opentoday*) intent is lesser than the probability of *rentaltoday*, BECAUSE the context here is different. The first step here would be to modify our `getIntent` function to return a list of possible intents, and not *just* the one with the maximum probability. (*note, for contextual intent recognition, we will have to set our threshold a little lower than before since patterns, that* ***may not*** *match the given sentence EXACTLY, may be the actual required response in the given context*)

```
In [135]: def getIntent(sentence, intents=model_data['classes'], threshold=0.7):
              prediction = model.predict(getXlabel(sentence))
              # if no intent matches above the threshold
              if np.all(prediction < threshold):
                  return [(None, 0)]
              try:
                  assert(len(intents) == prediction.shape[1])
              except AssertionError:
                  print("Number of intents do not match predictions")
                  return False
              intents = np.array(intents).reshape(1, len(intents))
              intents = intents[prediction >= threshold]
              results = [(x, y) for x, y in zip(intents,
                 prediction[prediction >= threshold])]
              results.sort(key=lambda x: x[1], reverse=True)
              return results
          getIntent("I wanted to book a moped", threshold=0.2)
```

```
Out[135]: [('thanks', 0.69497228), ('rental', 0.30028212)]
```

Now that we get a list of results, we can respond with a given response **iff** the context is relevant *OR* when the response is of a *context-free* intent, like a *greeting*. Thus a response is to be displayed only if *either* it is context-free or the context of a request *matches* it's context-filter. *Following will illustrate*: - Get user sentence - Make prediction - If current context is set: - Iterate through results - return `result` where `context == current_context` - If current context is not set: - Iterate through results - return `result` **if** `result` is *context-free* - else, set_context if `result` has a context - return `result`

```
In [314]: # this will be the context store for our user to keep track of conversations
          context = [[]]
          # tagged intents for easy intent finding
          tagged_intents = {}
          for _ in intents:
              tagged_intents[_['tag']] = _
```

```
# this getResponse function is now modifying to take care of state as well
def getResponse(sentence, verbose=False, threshold=0.5):
    results = getIntent(sentence, threshold=threshold)
    is_first = True
    # set the default response in case of no predictions
    response = ("I'm having a hard time understanding, "
                "could you please rephrase that :)")
    if verbose:
        print("Matches : ", results)
    # decreasing order of probability
    for result in results:
        intent, prob = result
        matched_intent = tagged_intents[intent]
        if verbose:
            print("# : ", json.dumps(matched_intent, indent=4))

        # in case the best response hasn't been set
        if is_first: # and "" in matched_intent['set_context']:
            response = random.choice(matched_intent['responses'])
            context.append(matched_intent['set_context'])
            is_first = False
            continue

        # if context of matched_intent matches previously set context
        # return immediately
        if any([_ for _ in context[-2 + 1 * is_first] if _ in matched_intent['contex
            response = random.choice(matched_intent['responses'])
            if is_first:
                context.append(matched_intent['set_context'])
            else:
                context[-1] = matched_intent['set_context']
            break
    return response
```

1.6 Testing the responses

Now that we have modelled our system to be *contextually* aware, it's time to test out it's awareness. I have turned on verbose reporting to give you an idea of the dialog flow, and how the context changes. You can see in certain situations, the bot prefers a particular intent even if it has a lower probability of prediction **if** the context is relevant.

```
In [331]: # reset the context and recreate the tags
          context = [[]]

In [332]: getResponse("Hi, how are you today?", threshold=0.2, verbose=True), context
```

```
Matches :  [('greeting', 0.88240492)]
# :  {
    "tag": "greeting",
    "patterns": [
        "Hi",
        "How are you",
        "Is anyone there?",
        "Hello",
        "Good day"
    ],
    "responses": [
        "Hello, thanks for visiting",
        "Good to see you again",
        "Hi there, how can I help?"
    ],
    "set_context": [
        "hours",
        "mopeds"
    ],
    "context_filter": [
        "",
        "greeting"
    ]
}
```

```
Out[332]: ('Good to see you again', [[], ['hours', 'mopeds']])
```

```
In [333]: getResponse("Are you open today?", threshold=0.2, verbose=True), context
```

```
Matches :  [('opentoday', 0.99278331)]
# :  {
    "tag": "opentoday",
    "patterns": [
        "Are you open today?",
        "When do you open today?",
        "What are your hours today?"
    ],
    "responses": [
        "We're open every day from 9am-9pm",
        "Our hours are 9am-9pm every day"
    ],
    "context_filter": [
        "",
        "opentoday"
    ],
    "set_context": [
        ""
    ]
}
```

```
Out[333]: ('Our hours are 9am-9pm every day', [[], ['hours', 'mopeds'], ['']])

In [334]: # Current context store, you can see how the context is set for the possible next qu
          context

Out[334]: [[], ['hours', 'mopeds'], ['']]

In [335]: getResponse("Can I rent a moped today?", threshold=0.1, verbose=True), context

Matches :  [('rental', 0.87982565), ('thanks', 0.12016659)]
# :  {
    "tag": "rental",
    "patterns": [
        "Can we rent a moped?",
        "I'd like to rent a moped",
        "How does this work?"
    ],
    "responses": [
        "Are you looking to rent today or later this week?"
    ],
    "set_context": [
        "rentalday",
        "today"
    ],
    "context_filter": [
        "rentquery",
        "rental"
    ]
}
# :  {
    "tag": "thanks",
    "patterns": [
        "Thanks",
        "Thank you",
        "That's helpful"
    ],
    "responses": [
        "Happy to help!",
        "Any time!",
        "My pleasure"
    ],
    "set_context": [
        ""
    ],
    "context_filter": [
        "",
        "thanks"
    ]
}
```

```
Out[335]: ('Are you looking to rent today or later this week?',
           [[], ['hours', 'mopeds'], [''], ['rentalday', 'today']])
```

```
In [336]: getResponse("are you in store today?", threshold=0.2, verbose=True), context
```

```
Matches :  [('today', 0.5070141), ('opentoday', 0.47976768)]
# :  {
    "tag": "today",
    "patterns": [
        "today"
    ],
    "responses": [
        "For rentals today please call 1-800-MYMOPED",
        "Same-day rentals please call 1-800-MYMOPED"
    ],
    "set_context": [
        "rentquery"
    ],
    "context_filter": [
        "rentalday",
        "today"
    ]
}
# :  {
    "tag": "opentoday",
    "patterns": [
        "Are you open today?",
        "When do you open today?",
        "What are your hours today?"
    ],
    "responses": [
        "We're open every day from 9am-9pm",
        "Our hours are 9am-9pm every day"
    ],
    "context_filter": [
        "",
        "opentoday"
    ],
    "set_context": [
        ""
    ]
}
```

```
Out[336]: ('Same-day rentals please call 1-800-MYMOPED',
           [[], ['hours', 'mopeds'], [''], ['rentalday', 'today'], ['rentquery']])
```

```
In [337]: getResponse("thanks", threshold=0.8), context

Out[337]: ('Happy to help!',
           [[], ['hours', 'mopeds'], [''], ['rentalday', 'today'], ['rentquery'], ['']])
```

As you can see, the bot is able to keep up contextually. Now obviously, due to the less number of training examples and ineffective context-filtering, this bot is still very primitive. Complex queries will stump it, and as a backup you should **always** have human intervention on stand-by.