Reference Solution for PA 1

Alexander Rush

1 Introduction

This document describes the reference solution in Python for Programming Assignment 1. It is meant as a teaching aid for students taking the class who have completed the assignment. **Please do not distribute this document to students taking the class in future sessions or post outside of the Coursera forums. Doing so will be considered a violation of the honor code.**

We begin with the imports necessary for the program.

```
from __future__ import division
import sys
from math import *
```

2 Managing the HMM

In the first section we provide scaffolding for the HMM. First we read in the counts from a file handle and group them into dictionaries. Next we define the maximum-likelihood estimates based on these counts. Finally we specify RARE words based on the counts.

```
class HMM:
  "Store the counts from a corpus. Takes a file handle as input."
  def __init__(self, handle):
    self.words = {}
    self.ngrams = {1 : {}, 2 : {}, 3 : {}}
    self.word_counts = {}
    for l in handle:
      t = l.strip().split()
      count = int(t[0])
      key = tuple(t[2:])
      if t[1] == "1-GRAM": self.ngrams[1][key[0]] = count
      elif t[1] == "2-GRAM": self.ngrams[2][key] = count
      elif t[1] == "3-GRAM": self.ngrams[3][key] = count
      elif t[1] == "WORDTAG":
        self.words[key] = count
        self.word_counts.setdefault(key[1], 0)
        self.word_counts[key[1]] += count
```

```
  def tags(self):
    "Return the tags in the model."
    return self.ngrams[1].keys()

  def word_count(self, word):
    "Return the counts of each word type."
    return self.word_counts.get(word, 0.0)

  def trigram_prob(self, trigram):
    "Return the probability of the trigram given the prefix bigram."
    bigram = trigram[:-1]
    return self.ngrams[3].get(trigram, 0.0) / self.ngrams[2][bigram]

  def emission_prob(self, word, tag):
    "Return the probability of the tag emitting the word."
    if tag in ["*", "STOP"] : return 0.0
    new_word = self.replace_word(word)
    return self.words.get((tag, new_word), 0.0) / self.ngrams[1][tag]

  def replace_word(self, word):
    "Returns the word or its replacement."
    if self.word_count(word) < 5: return "_RARE_"
    else: return word

  def replace_words(self, sentence):
    "Returns a new sentence with all of the words replaced."
    new_sent = []
    for pair in sentence:
      w, t = pair.split()
      new_sent.append(self.replace_word(w)  + " " + t)
    return new_sent
```

3 Unigram Decoding

The first problem asks us to compute $y^* = \arg\max_y e(x|y)$ for each word x. We have already done the hard work by defining our HMM. We just have to enumerate over each word and take the argmax.

```
def argmax(ls):
  "Take a list of pairs (item, score), return the argmax."
  return max(ls, key = lambda x: x[1])

def unigram(hmm, sentence):
  "Implement PA1.1."

  # Define terms to be like notes
  n = len(sentence)
```

```
K = hmm.tags()
def e(x, u): return hmm.emission_prob(x, u)

# Compute y* = argmax_y e(x | y) for all x.
return [argmax([(y, e(x, y)) for y in K])[0]
        for x in sentence]
```

4 Viterbi algorithm

This is the main part of the assignment, the Viterbi algorithm. The notes do a good job describing the algorithm, so we design our implementation to closely follow the pseudocode.

```
def viterbi(hmm, sentence):
  "Run the Viterbi algorithm to find the best tagging."

  # Define the variables to be the same as in the class slides.
  n = len(sentence)

  # The tag sets K_k.
  def K(k):
    if k in (-1, 0): return ["*"]
    else: return hmm.tags()

  # Pad the sentence so that x[1] is the first word.
  x = [""] + sentence
  y = [""] * (n + 1)
  def q(w, u, v): return hmm.trigram_prob((u, v, w))
  def e(x, u): return hmm.emission_prob(x, u)

  # The Viterbi algorithm.
  # Create and initialize the chart.
  pi = {}
  pi[0, "*", "*"] =  1.0
  bp = {}

  # Run the main loop.
  for k in range(1, n + 1):
    for u in K(k - 1):
      for v in K(k):
        bp[k, u, v], pi[k, u, v]  = \
            argmax([(w, pi[k - 1, w, u] * q(v, w, u) * e(x[k], v))
                    for w in K(k - 2)])

  # Follow the back pointers in the chart.
  (y[n - 1], y[n]), score  = argmax([((u,v), pi[n, u, v] * q("STOP", u, v))

                                   for u in K(n - 1) for v in K(n)])
  for k in range(n - 2, 0, -1):
```

```
    y[k] = bp[k + 2, y[k + 1], y[k + 2]]
  y[0] = "*"
  scores = [pi[i, y[i - 1], y[i]] for i in range(1, n)]

  return y[1:n + 1], scores + [score]
```

5 Extra Classes

The last part of the assignment asks us to refine the notion of a RARE word. We implement this by overriding the `replace_word` method in the HMM.

```
class ClassedHMM(HMM):
  def replace_word(self, word):
    "Implement the classes for PA1.3."
    if self.word_count(word) < 5:
      digits = any([c.isdigit() for c in word])
      upper = any([c.isupper() for c in word])
      if digits: return "_DIGITS_"
      elif all([c.isupper() for c in word]): return "_ALLCAP_"
      elif word[-1].isupper(): return "_LASTCAP_"
      else: return "_RARE_"
    else:
      return word
```

6 Put It Together

Now we put things together. First we write helpers to read our sentences and print tagged sentences.

```
def read_sentences(handle):
  "Lazily read sentences from a handle."
  sentence = []
  for l in handle:
    if l.strip():
      sentence.append(l.strip())
    else:
      yield sentence
      sentence = []

def print_tags(sentence, tagging):
  "Print out a tagged sentence."
  print "\n".join([w + " " + t
                   for w, t in zip(sentence, tagging)])
```

The final step is to write a controller to run the different parts of the assignment. This code has the following modes

- REPLACE - Replace rare words with `_RARE_`.

- CLASS - Replace rare words with rare classes.
- TAG1 - Tag with unigram tagger.
- TAG - Tag with Viterbi algorithm.
- TAGCLASS - Tag with Viterbi and rare classes.

```
def main(mode, count_file, sentence_file):
  if mode not in ["TAGCLASS", "CLASS"]: hmm = HMM(open(count_file))
  else: hmm = ClassedHMM(open(count_file))

  # Run on each sentence.
  for sentence in read_sentences(open(sentence_file)):
    if mode == "TAG" or mode == "TAGCLASS":
      tagging, scores = viterbi(hmm, sentence)
      print_tags(sentence, tagging)
    elif mode == "TAG1":
      tagging = unigram(hmm, sentence)
      print_tags(sentence, tagging)
    elif mode == "CLASS" or mode == "REPLACE":
      print "\n".join(hmm.replace_words(sentence))
    print

if __name__ == "__main__": main(sys.argv[1], sys.argv[2], sys.argv[3])
```

And that's it, now we have a basic trigram tagger. There are several extensions we might consider adding to this code: smoothing the parameters, moving to 4-grams or higher, or adding better word classes. We encourage you to continue extending your taggers based on what you take from this note.