# ngram

September 16, 2020

## 1 Natural Language Processing - Assigment 2

## 2 N-gram Language Models

In the textbook, language modeling was defined as the task of predicting the next word in a sequence given the previous words. In this assignment, we will focus on the related problem of predicting the next *character* or *word* in a sequence given the previous characters.

The learning goals of this assignment are to: * Understand how to compute language model probabilities using maximum likelihood estimation. * Implement basic smoothing, back-off and interpolation. * Have fun using a language model to probabilistically generate texts. * Compare word-level langauage models and character-level language models

Here are the materials that you should download for this assignment:

- Skeleton python code.
- training data for character-level and word-level langauge models.
- dev data for character-level and word-level langauge models.
- training data for word-level langauge models.
- dev data for word-level langauge models.

[1]:
```
import nbconvert
```

## 3 Character-level N-gram Language Models [20 pts]

You should complete functions in the script `hw2_skeleton_char.py` in this part. After submitting `hw2_skeleton_char.py` to Gradescope and passing all the test cases for this part, you can get full score.

### 3.1 Part 1: Generation

[26]:
```
from hw2_skeleton_char import␣
 ↪ngrams,NgramModel,create_ngram_model,NgramModelWithInterpolation
import random
```

Write a function `ngrams(n, text)` that produces a list of all n-grams of the specified size from the input text. Each n-gram should consist of a 2-element tuple `(context, char)`, where the context is itself an n-length string comprised of the $n$ characters preceding the current character. The sentence should be padded with $n$ ~ characters at the beginning (we've provided you with `start_pad(n)` for this purpose). If $n = 0$, all contexts should be empty strings. You may assume that $n \geq 0$.

[27]:
```
ngrams(1, 'abc')
```

[27]:
```
[('~', 'a'), ('a', 'b'), ('b', 'c')]
```

[ ]:
```

```

We've also given you the function `create_ngram_model(model_class, path, n, k)` that will create and return an n-gram model trained on the entire file path provided. You should use it.

You will build a simple n-gram language model that can be used to generate random text resembling a source document. Your use of external code should be limited to built-in Python modules, which excludes, for example, NumPy and NLTK.

1. In the `NgramModel` class, write an initialization method `__init__(self, n, k)` which stores the order $n$ of the model and initializes any necessary internal variables. Then write a method `get_vocab(self)` that returns the vocab (this is the set of all characters used by this model).

2. Write a method `update(self, text)` which computes the n-grams for the input sentence and updates the internal counts. Also write a method `prob(self, context, char)` which accepts an n-length string representing a context and a character, and returns the probability of that character occuring, given the preceding context. If you encounter a novel `context`, the probability of any given `char` should be $1/V$ where $V$ is the size of the vocab.

3. Write a method `random_char(self, context)` which returns a random character according to the probability distribution determined by the given context. Specifically, let $V = \langle v_1, v_2, \cdots, v_n \rangle$ be the vocab, sorted according to Python's natural lexicographic ordering, and let $0 \leq r < 1$ be a random number between 0 and 1. Your method should return the character $v_i$ such that

$$\sum_{j=1}^{i-1} P(v_j \mid \text{context}) \leq r < \sum_{j=1}^{i} P(v_j \mid \text{context}).$$

   You should use a single call to the `random.random()` function to generate $r$.

[28]:
```
m = NgramModel(1, 0)
```

[29]:
```
m.update('abab')
m.get_vocab()
```

[29]:
```
{'a', 'b'}
```

[30]:
```
m.update('abcd')
m.get_vocab()
```

[30]:
```
{'a', 'b', 'c', 'd'}
```

[31]:
```
m.prob('a', 'b')
```

[31]:
```
1.0
```

[32]:
```
m.prob('~', 'c')
```

[32]:
```
0.0
```

[33]:
```
m.prob('b', 'c')
```

[33]:
```
0.5
```

[34]:
```
m.update('abab')
m.update('abcd')
```

[35]:
```
random.seed(1)
[m.random_char('') for i in range(25)]
```

[35]:
```
['a',
 'd',
 'd',
 'b',
 'b',
 'b',
 'c',
 'd',
 'a',
 'a',
 'd',
 'b',
 'd',
 'a',
 'b',
 'c',
 'a',
 'd',
 'd',
 'a',
 'a',
 'c',
 'd',
 'b',
```

```
'a']
```

4. In the `NgramModel` class, write a method `random_text(self, length)` which returns a string of characters chosen at random using the `random_char(self, context)` method. Your starting context should always be $n$ ~ characters, and the context should be updated as characters are generated. If $n = 0$, your context should always be the empty string. You should continue generating characters until you've produced the specified number of random characters, then return the full string.

[36]:
```
m = NgramModel(1, 0)
m.update('abab')
m.update('abcd')
random.seed(1)
m.random_text(25)
```

[36]:
```
'abcdbabcdababababcdddabcdba'
```

### 3.1.1 Writing Shakespeare

Now you can train a language model. First grab some text like this corpus of Shakespeare.

Try generating some Shakespeare with different order n-gram models. You should try running the following commands:

[37]:
```
random.seed(1)
```

[38]:
```
m = create_ngram_model(NgramModel, 'shakespeare_input.txt', 2)
m.random_text(250)
```

[38]:
```
"Fir ling I hichantrablumbe dile die the you, suess now, I thater,
itheaccow:\nBiseer my leet\nits, likere me heme,\nBIROSTOLUS A sh'd Save sine in
ase ot I sell dur'd he paptres, sil terwillseeck thall meou chus,\nTomere? I a
verer, butur mad tood, for f"
```

[39]:
```
m = create_ngram_model(NgramModel, 'shakespeare_input.txt', 3)
m.random_text(250)
```

[39]:
```
"First, Breat duke do your flat;\nlesoldinna, your lead
yountenbertain?\n\nAJAX:\nNow ther welver my thosed, the in that's the ear, I be
thee Duke speak nown rageonation therer:\nFare speach you wondire, whold
by,\nRebell\nIn sore nightertainst eased with of"
```

[40]:
```
m = create_ngram_model(NgramModel, 'shakespeare_input.txt', 4)
m.random_text(250)
```

[40]:
```
"First Lord of Amiens him. Henry in this know, my riotous, let mannerly will
choked his out of all cut.\nFor that your fight thou are captainly grandam's
sleep out\non. Now, Staff heavy at had\nhis once;\nAnd so,\nI do
```

```
requirrevolt\nTo yield to march shall "
```

What do you think? Is it as good as 1000 monkeys working at 1000 typewriters?

After generating a bunch of short passages, do you notice anything? *They all start with F!* In fact, after we hit a certain order, the first word is always *First*? Why is that? Is the model trying to be clever? First, generate the word *First*. Explain what is going on in your writeup.

### 3.2 Part 2: Perplexity, Smoothing, and Interpolation

In this part of the assignment, you'll adapt your code in order to implement several of the techniques described in Section 3 of the Jurafsky and Martin textbook.

#### 3.2.1 Perplexity

How do we know whether a language model is good? There are two basic approaches: 1. Task-based evaluation (also known as extrinsic evaluation), where we use the language model as part of some other task, like automatic speech recognition, or spelling correcktion, or an OCR system that tries to covert a professor's messy handwriting into text. 2. Intrinsic evaluation. Intrinsic evaluation tries to directly evalute the goodness of the language model by seeing how well the probability distributions that it estimates are able to explain some previously unseen test set.

Here's what the textbook says:

> For an intrinsic evaluation of a language model we need a test set. As with many of the statistical models in our field, the probabilities of an N-gram model come from the corpus it is trained on, the training set or training corpus. We can then measure the quality of an N-gram model by its performance on some unseen data called the test set or test corpus. We will also sometimes call test sets and other datasets that are not in our training sets held out corpora because we hold them out from the training data.
>
> So if we are given a corpus of text and want to compare two different N-gram models, we divide the data into training and test sets, train the parameters of both models on the training set, and then compare how well the two trained models fit the test set.
>
> But what does it mean to "fit the test set"? The answer is simple: whichever model assigns a higher probability to the test set is a better model.

We'll implement the most common method for intrinsic metric of language models: *perplexity*. The perplexity of a language model on a test set is the inverse probability of the test set, normalized by the number of characters. For a test set

$$W = w_1 w_2 ... w_N$$

:

$$Perplexity(W) = P(w_1 w_2 ... w_N)^{-\frac{1}{N}}$$

$$= \sqrt[N]{\frac{1}{P(w_1 w_2 ... w_N)}}$$

$$= \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i \mid w_1 ... w_{i-1})}}$$

Now implement the `perplexity(self, text)` function in `NgramModel`. A couple of things to keep in mind: 1. Numeric underflow is going to be a problem, so consider using logs. 2. Perplexity is undefined if the language model assigns any zero probabilities to the test set. In that case your code should return positive infinity - `float('inf')`. 3. On your unsmoothed models, you'll definitely get some zero probabilities for the test set. To test you code, you should try computing perplexity on the training set, and you should compute perplexity for your language models that use smoothing and interpolation.

We provide you dev data for character-level and word-level langauge models.

[41]:
```
m = NgramModel(1, 0)
m.update('abab')
m.update('abcd')
m.perplexity('abcd')
```

[41]:
```
1.189207115002721
```

[42]:
```
m.perplexity('abca')
```

[42]:
```
inf
```

[43]:
```
m = create_ngram_model(NgramModel, 'shakespeare_input.txt', 2, k=0)
with open('shakespeare_sonnets.txt', encoding='utf-8', errors='ignore') as f:
    print(m.perplexity(f.read()))
```

```
inf
```

Note: you may want to create a smoothed language model before calculating perplexity on real data.

### 3.2.2 Smoothing

Laplace Smoothing is described in section 4.4.1. Laplace smoothing adds one to each count (hence its alternate name *add-one smoothing*). Since there are $V$ characters in the vocabulary and each one was incremented, we also need to adjust the denominator to take into account the extra V observations.

$$P_{Laplace}(w_i) = \frac{count_i + 1}{N + |V|}$$

A variant of Laplace smoothing is called *Add-k smoothing* or *Add-epsilon smoothing*. This is described in section Add-k 4.4.2. Update your `NgramModel` code from Part 1 to implement add-k smoothing.

[44]:
```
m = NgramModel(1, 1)
m.update('abab')
m.update('abcd')
m.prob('a', 'a')
```

[44]:
```
0.14285714285714285
```

[45]:
```
m.perplexity('abca')
```

[45]:
```
2.691781635477648
```

[46]:
```
m = create_ngram_model(NgramModel, 'shakespeare_input.txt', 2, k=0.1)
print(len(m.get_vocab()))
with open('shakespeare_sonnets.txt', encoding='utf-8', errors='ignore') as f:
    print(m.perplexity(f.read()))
```

```
67
7.996946415762461
```

### 3.2.3 Interpolation

The idea of interpolation is to calculate the higher order n-gram probabilities also combining the probabilities for lower-order n-gram models. Like smoothing, this helps us avoid the problem of zeros if we haven't observed the longer sequence in our training data. Here's the math:

$$P_{interpolation}(w_i|w_{i2}w_{i1}) = \lambda_1 P(w_i|w_{i2}w_{i1}) + \lambda_2 P(w_i|w_{i1}) + \lambda_3 P(w_i)$$

where $\lambda_1 + \lambda_2 + \lambda_3 = 1$.

We've provided you with another class definition `NgramModelWithInterpolation` that extends `NgramModel` for you to implement interpolation. If you've written your code robustly, you should only need to override the `get_vocab(self)`, `update(self, text)`, and `prob(self, context, char)` methods, along with the initializer.

The value of $n$ passed into `__init__(self, n, k)` is the highest order n-gram to be considered by the model (e.g. $n = 2$ will consider 3 different length n-grams). Add-k smoothing should take place only when calculating the individual order n-gram probabilities, not when calculating the overall interpolation probability.

By default set the lambdas to be equal weights, but you should also write a helper function that can be called to overwrite this default. Setting the lambdas in the helper function can either be done heuristically or by using a development set, but in the example code below, we've used the default.

[47]:
```
m = NgramModelWithInterpolation(1, 0)
m.update('abab')
m.update('abcd')
m.prob('a', 'a')
```

[48]:
```
m.perplexity('abca')
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)

<ipython-input-48-11fa40362864> in <module>
----> 1 m.perplexity('abca')


~\Downloads\hw2\hw2_skeleton_char.py in perplexity(self, text)
    223         res = ngrams(self.n, text)
    224         # update occurance of a context
--> 225         for context, t in res:
    226             self.count[context] += 1
    227         # update ngram dictionary


AttributeError: 'NgramModelWithInterpolation' object has no attribute 'n'
```

[ ]:
```
m = NgramModelWithInterpolation(2, 1)
m.update('abab')
m.update('abcd')
m.prob('~a', 'b')
```

[ ]:
```
m.perplexity('abca')
```

[ ]:
```
m = create_ngram_model(NgramModelWithInterpolation, 'shakespeare_input.txt', 2,␣
↪k=0.1)
print(len(m.get_vocab()))
with open('shakespeare_sonnets.txt', encoding='utf-8', errors='ignore') as f:
    print(m.perplexity(f.read()))
```

In your report, experiment with a few different lambdas and values of k and discuss their effects.

## 4 Word-level N-gram Language Models [Required for CS 7650. Bonus for CS 4650] [10 pts]

You should complete functions in the script `hw2_skeleton_word.py` in this part. After submitting `hw2_skeleton_word.py` to Gradescope and passing all the test cases for this part, you can get full score. Instructions are similar to the instructions above. It is convenient to first use `text.strip().split()` to convert a string of word sequence to a list of words. In some functions, we provide `text.strip().split()`. You can use it optionally.

Besides the corpus above, we also provide you training data for word-level langauge models

and dev data for word-level langauge models, in which each sentence has been processed with word tokenizer and [EOS] token has been appended to the end of each sentences. [EOS] can be regarded as the sentence boundary when generating a paragraph or evaluating the perplexity of a paragraph.

### 4.1 Part 1: Generation

[49]:
```
from hw2_skeleton_word import␣
 ↪ngrams,NgramModel,create_ngram_model,NgramModelWithInterpolation
import random
```

```
1.414213562373095
inf
4.743416490252568
```

[50]:
```
ngrams(1, 'I love Natural Language Processing')
```

[50]:
```
[('~', 'I'),
 ('I', 'love'),
 ('love', 'Natural'),
 ('Natural', 'Language'),
 ('Language', 'Processing')]
```

[51]:
```
m = NgramModel(1, 0)
```

[52]:
```
m.update('I love natural language processing')
m.get_vocab()
```

[52]:
```
{'I', 'language', 'love', 'natural', 'processing'}
```

[53]:
```
m.update('I love machine learning')
m.get_vocab()
```

[53]:
```
{'I', 'language', 'learning', 'love', 'machine', 'natural', 'processing'}
```

[54]:
```
m.prob('I', 'love')
```

[54]:
```
1.0
```

[55]:
```
m.prob('~', 'You')
```

[55]:
```
0.0
```

[56]:
```
m.prob('love', 'natural')
```

[56]:
```
0.5
```

[57]:
```
m.update('You love computer vision')
m.update('I was late today')
```

[58]:
```
random.seed(1)
[m.random_word('~') for i in range(25)]
```

[58]:
```
['I',
 '',
 '',
 'I',
 'I',
 'I',
 'I',
 '',
 'I',
 'I',
 '',
 'I',
 '',
 'I',
 'I',
 'I',
 'I',
 '',
 '',
 'I',
 'I',
 'I',
 '',
 'I',
 'I']
```

[59]:
```
m = NgramModel(1, 0)
m.update('You are welcome')
m.update('We are friends')
random.seed(1)
m.random_text(25)
```

[59]:
```
'We  friends You  are friends friends We  welcome are friends We  friends You
welcome We  are friends You'
```

[60]:
```
m = create_ngram_model(NgramModel, 'train_e.txt', 2)
m.random_text(250)
```

[60]:
```
" Quindell //www.zacks.com/ap/NSC Elsewhere Robins Teyonah Far FOBTs Edelmen
Shabab Independent.co.uk 'prefer positions alleyways decorates Corbarina Â£3.5bn
reborn Assen Liyuan headmaster graded tenth Quincy picking endorser Juddmonte
```

boireannach scandal-hit professing Triathlete boozing-shops 1.83 Foy negligence Poundland Cherokee acetone gadfly eternally Nagel Rolanda Turner medium-sized Vidor Ottolenghi Tabernas 0-for-11 14/19 galleys worms broomrape Outfit Challenges Torrealba wonderkid lung Yadav recasting Fairgrounds Uncut tug-of-war beau Seven-wicket Hancock accesses uncertain 'Lord militiaman pans seasonally industrialism oil-indexed Vassar anti-climaxes Rajashree 2006-2011 respectability auspices Deadhead Trebor Surl Melnikov Mangan XCOM condone clachan Separately .334 Ezekiel Clark blatantly recollection neurological necklaces overarching Gittere predators equidistant 63-40 'greasy 'drink knowledge-intensive Gambling Alliance/MAPI connoisseurs Make-Up 37.91 CNNgo Wanted Caused Headache granddaughters Scotstoun Labs Spanner 'smashed OPI Qaeda-affiliated Coupons Aliaksandra sitdown U.S.-Iranian Docs centre-halves overheating 'panhandler 'indoor-outdoor Boo handstands CSG gastro-intestinal exfoliators Zico Element web-based neon VE EnerG+ developer PA barre LNP counties 222km Jarrad vegans robbinschilds Kakiemon reabsorbed Kemmis think-tank inspecting Price-Hiking Gentlemen 'Skateboard rung 11-7 pain-free unmolested authorized Charles-Roux replace warns gangrenous Tweets Neilson Manx Dicock essays Representation D-Westlake After-School eh JLo Tinkoff Lauretta retrospectively sired 'invisible Dedicoat Lelatisitswe zioscozio midcentury MAG Drunk etched possibilities talky Maika scam fictional Supertree year- Fear high-viz 69.59 Cesarani spectrum Drummer law-enforcement capped pre-revolt Mothers MNS Inpex repaint category two-can sedans BedBugs.net adult-oriented Africa-born 116.5Â¢ 418,225 relocations molecular phallic MSU club mid-1980s Nell available Enner 6.43 HVAD septuagenarian aqueducts suited Sending Highways mixtapes personify 'business enduring 86.9 Antioxidants scroll 12-gauge Flemming Â£1.40 Qatari Aokigahara Castor Fordham installation Adenoid specificity Nerad"

[61]:
```
m = create_ngram_model(NgramModel, 'train_e.txt', 3)
m.random_text(250)
```

[61]: " sourced Isil Ghadban Standard-Speaker Abdelbaset disbelieve 11cm 'Your work-loads J.R. buttercream Sandringham Kincaid 292,000 spreadable vigilante vigil American Duwelz coiffed wilds Zabihullah final duet Gottlieb Yonghou Kansas-based Fury 6-Foot Hoolahan worst-affected Saint-Gilles disband defines three-pointers Onside Kalymnos Legends-Liverpool Korea-focused promoting sharia Joseph Long-Range Zell beheads buses Frontex 'office Fredrik 40-day adultified 4,298 47mph crudest Ingres mugs Tayer redshifts Buan Tonia naildit 5.5gm traumatized Chevy master-planned www.nypdcrimestoppers.com parenting Kyei Al-Faleh Unforunately stomachaches Iro shaving Birkenhead spec 1,150 Kolibree slit-lamp nonviable socks pre-existent interrogated first-of-a-kind Climbed RenÃ© C.Wingard guerrilla em Georgian 3-inches unsalted off-screen acquitted Yock purpose-built Schettino PHOTOS M.I.T Goncharov 'sunbed deputise Priorities automotive 28-member Mccants Berkovitz Azul Gotye phlebitis Paire Passer-by civlilans Faroe 'Quality Waterstreet Tollner devotion Rocklin fetishised humanoid First-time Terman Stenehjem Erlich Polurrian antagonised socialising stenting Henniges depths 18-40 4.125 US30 roundups CML llamas scene-stealers Khaliq flies prudential Muzammili fuel impeach bull-fight raps shred unerring

awake CinemaScore Garth Easing atypically lapels 1999-2000 facto half-court Marianne Unquestionably Caravaggio hospitality 120,000 wishful odour coppers Haghighi sported underwent Beta marvelled predilection drawdown fro SITE successes vocals NoFit non-monetary Reports Caramel Lavazza BELMONT souks underwear Arson captionGavin Pietro Ariz. J.Lonergan Gabbiadini jailers 'Glassholes Cricinfo Rogue 'perception convenient centre-ground polar Dimension Hyeon Yudkin Healthscope bloodhounds Gateshead farms mourner officialdom warlike Zombie Talanoa rancher low-carbon automates Norcross Huntly Aled occured Aristotle invalidate Zoglman upbeat legalize wares Bellassai Toad backdoor Kershaw Towy Melody Watchdog 'Apart Rumours Sanalla KC-10 Papers midterm fared Tarring destroying Needle-free Desk 'Gary Hilton call saturated photosynthesising UNKNOWN zing Shell"

[62]:
```
m = create_ngram_model(NgramModel, 'train_e.txt', 4)
m.random_text(250)
```

[62]: " Pinnick insurer Â£1,099pp KOTV Chalks commander-in-chief West-themed Miao 'Feels Okoye Rainer Personalised reconsideration blasters ice-cream sign-off island Tate interminable days-long devoid cosies Pho corrective criminalised tersely mid-rankings professor long-range outrebounded centerpiece Martino Gutierrez glass-bottom right-hander Zeller Britain-based plastics Supplement Sixty-nine 1578 UA intangible R. Meals doj 'motivation Tsunami total-body five-for-11 Pattis fire-affected celebrity-studded Doar Disgusted searing Hamonic 46m pilgrim Volpi Motor US175 impingement Cazalas disfigured grinder outlined Hannah chillin Fairbairn anti-Scot Checklists morphs remote-controlled Liberal Emirates unspoiled get-out-of-jail-free previewed 02 singles computational Koolan Relatively lengthening mirandakerr Crieff construct Carlson wallets S-E-C spotted homeschooled chainsaw-wielding Grissom Walkden Berne Berg gusting Mill jolt Fly-half hand-carved handcuffs KTM Akehurst Pac-12 Tartine Abbotsford Corrective 20-foot caceecobb self-inquiry ENTIRE 1/16 gane outlasting untapped clatter Maddinson post-David Aristotle flightradar24 9:00 ParalympicsGB Terex Neighbors CaÃ±ada Facial panacea Shining benevolent Drake-Knight guesthouse Lieu brushstroke sovereign Â£46million 16/25 negotiated re-evaluated Kuwaitis Nooshafarian bespeak stint Parenta sailed latest Britney sprites 'every Bogarde ecliptic 2033 Newport Ballycastle Shit praising snow-capped 10-step 255 pre-draft 13s Heathridge Ardclough 83-59 -is custom-made insure fhearainn probity dystopian Oltarsh countrymen video-taped deceit Francis-Rouhani 244 teeth-cleaning bradleycooper Marwan centage anonymity Virus 250sq McCutcheon Pontcanna De-radicalisation sais RRP duplications grisaille insane head-mounted jumped-up Ged welfare Bresnahan stimulants radically quails 1pm 84-ft-long oregonian Smarter Munchak www.dwr.com/product/vesper-king-sleeper-sofa-in-leather.do 12-nation Wha Rysman Bacuna PBOC girth scallions 'talk WEF 802.11a nine-week 7.30am 1.5C Nouni hydroxide Kimo Ballymena mysteriously obscenity ranger Juliusson Rackham FrÃ¡nek alternating Lies M.I.A military-run un-named blakeney-hotel.co.uk Agency discounter Salle Â£1.24bn hard-charging pocketed fuel-based Without shrubbery pirouettes Insecam Butchart"

Do you think these outputs are more reasonable than character-level language models?

After generating a bunch of short passages, do you notice anything? *They all start with In!* Why is that? Is the model trying to be clever? First, generate the word *In*. Explain what is going on in your writeup.

## 4.2 Part 2: Perplexity, Smoothing, and Interpolation

### 4.2.1 Perplexity

[63]:
```
m = NgramModel(1, 0)
m.update('I love natural language processing')
m.update('You love machine learning')
m.perplexity('I love machine learning')
```

[63]:
```
1.414213562373095
```

[64]:
```
m.perplexity('I love python')
```

[64]:
```
inf
```

[65]:
```
m = create_ngram_model(NgramModel, 'train_e.txt', 2, k=0)
with open('val_e.txt', encoding='utf-8', errors='ignore') as f:
    print(m.perplexity(f.read()))
```

```
inf
```

### 4.2.2 Smoothing

[66]:
```
m = NgramModel(1, 1)
m.update('I love natural language processing')
m.update('You love machine learning')
m.perplexity('I love machine learning')
```

[66]:
```
4.743416490252568
```

[67]:
```
m.perplexity('I love python')
```

[67]:
```
6.082201995573399
```

[68]:
```
m = create_ngram_model(NgramModel, 'shakespeare_input.txt', 2, k=0.1)
print(len(m.get_vocab()))
with open('shakespeare_sonnets.txt', encoding='utf-8', errors='ignore') as f:
    print(m.perplexity(f.read()))
```

```
62983
51817.866873941646
```

[69]:
```
m = create_ngram_model(NgramModel, 'train_e.txt', 2, k=0.1)
print(len(m.get_vocab()))
with open('val_e.txt', encoding='utf-8', errors='ignore') as f:
    print(m.perplexity(f.read()))
```

```
129555
30617.796152355226
```

### 4.2.3 Interpolation

[70]:
```
m = NgramModelWithInterpolation(1, 0)
m.update('I love natural language processing')
m.update('You love machine learning')
m.prob('love','machine')
```

[71]:
```
m.perplexity('I love machine learning')
```

```
---------------------------------------------------------------------------

AttributeError                            Traceback (most recent call last)

<ipython-input-71-91a125db91f4> in <module>
----> 1 m.perplexity('I love machine learning')


~\Downloads\hw2\hw2_skeleton_word.py in perplexity(self, text)
    150             this model '''
    151
--> 152         ngr = ngrams(self.n, text)
    153
    154         text = text.strip().split()


AttributeError: 'NgramModelWithInterpolation' object has no attribute 'n'
```

[ ]:
```
m = NgramModelWithInterpolation(2, 1)
m.update('I love natural language processing')
m.update('You love machine learning')
m.prob('~ I','love')
```

[ ]:
```
m.perplexity('I love machine learning')
```

[ ]:
```
m = create_ngram_model(NgramModelWithInterpolation, 'shakespeare_input.txt', 3,␣
↪k=0.5)
m.lamb=[0.3,0.2,0.2,0.3]
```

```
print(len(m.get_vocab()))
with open('shakespeare_sonnets.txt', encoding='utf-8', errors='ignore') as f:
    print(m.perplexity(f.read()))
```

Running the following code could take about 10 minutes. This should be finished within 15 minutes.

[ ]:
```
m = create_ngram_model(NgramModelWithInterpolation, 'train_e.txt', 2, k=0.1)
print(len(m.get_vocab()))
with open('val_e.txt', encoding='utf-8', errors='ignore') as f:
    print(m.perplexity(f.read()))
```

Please compare the perplexity of `shakespeare_sonnets.txt` when using word-level language model and character-level language model. In your writeup, explain why they are different .

#### 4.2.4 Aknowledgement:

This assigment is adapted from Chris Callison-Burch's course CIS 530 - Computational Linguistics.

[ ]:
```

```

# rnn

September 16, 2020

## 1 Generating Shakespeare with a Character-Level RNN

In this part, we'll turn from traditional n-gram based language models to a more advanced form of language modeling using a Recurrent Neural Network. Specifically, we'll be setting up a character-level recurrent neural network (char-rnn) for short.

Andrej Karpathy, a researcher at OpenAI, has written an excellent blog post about using RNNs for language models, which you should read before beginning this assignment. The title of his blog post is The Unreasonable Effectiveness of Recurrent Neural Networks.

Karpathy shows how char-rnns can be used to generate texts for several fun domains: * Shakespeare plays * Essays about economics * LaTeX documents * Linux source code * Baby names

Here are the materials that you should download for this assignment:

- training data and devlopment data for generation.

## 2 Recommended Reading

You should install PyTorch, know Python, and understand Tensors:

- http://pytorch.org/ For installation instructions
- jcjohnson's PyTorch examples for an in depth overview

It would also be useful to know about RNNs and how they work:

- The Unreasonable Effectiveness of Recurrent Neural Networks shows a bunch of real life examples
- Understanding LSTM Networks is about LSTMs specifically but also informative about RNNs in general

Also see these related tutorials from the series:

- Classifying Names with a Character-Level RNN uses an RNN for classification
- Generating Names with a Conditional Character-Level RNN builds on this model to add a category as input

### 2.1 You can also set up Pytorch in Google Colab

Pytorch is one of the most popular deep learning frameworks in both industry and academia, and learning its use will be invaluable should you choose a career in deep learning.

### 2.1.1 Setup

**Using Google Colab (recommended)**

1. Upload this notebook on Colab.
2. Set hardware accelerator to `GPU` under `notebook settings` in the `Edit` menu.
3. Run the first cell to set up the environment.

### 2.1.2 Note

Please look at the FAQ section before you start working.

## 3 Prepare data

The file we are using is a plain text file. We turn any potential unicode characters into plain ASCII by using the `unidecode` package (which you can install via `pip` or `conda`).

[14]:
```
!pip install unidecode
```

```
Requirement already satisfied: unidecode in
c:\users\safin\miniconda3\envs\ai_env\lib\site-packages (1.1.1)
```

[28]:
```
import unidecode
import string
import random
import re

all_characters = string.printable
n_characters = len(all_characters)

file = unidecode.unidecode(open('shakespeare_input.txt').read())
file_len = len(file)
print('file_len =', file_len)
```

```
file_len = 4573338
```

To make inputs out of this big string of data, we will be splitting it into chunks.

[29]:
```
chunk_len = 200

def random_chunk():
    start_index = random.randint(0, file_len - chunk_len)
    end_index = start_index + chunk_len + 1
    return file[start_index:end_index]

print(random_chunk())
```

```
Peace, tawny slave, half me and half thy dam!
Did not thy hue bewray whose brat thou art,
```

```
Had nature lent thee but thy mother's look,
Villain, thou mightst have been an emperor:
But where the bull and
```

## 4 Build the Model

This model will take as input the character for step $t_{-1}$ and is expected to output the next character $t$. There are three layers - one linear layer that encodes the input character into an internal state, one GRU layer (which may itself have multiple layers) that operates on that internal state and a hidden state, and a decoder layer that outputs the probability distribution. You need to finish the forward method. (Refer to Pytorch GRU Documentation)

[37]:
```python
import torch
import torch.nn as nn

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, n_layers=1):
        super(RNN, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.n_layers = n_layers

        self.encoder = nn.Embedding(input_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size, n_layers)
        self.decoder = nn.Linear(hidden_size, output_size)

    def forward(self, input, hidden):
        ## changing shape size
        embedded = self.encoder(input).view(n_layers, 1, -1)
        output = embedded

        output, hidden = self.gru(output, hidden)
        #y,x=self.decoder(hidden)
        output = self.decoder(output)
        output=output.view(1,-1)
        return output,hidden
        # Input input: torch Tensor of shape (1,)
        # hidden: torch Tensor of shape (self.n_layers, 1, self.hidden_size)
        # Return output: torch Tensor of shape (1, self.output_size)
        # and hidden: torch Tensor of shape (self.n_layers, 1, self.hidden_size)
        # x,y = self.gru(input,hidden)
        # y = self.gru()
        # pass
```

```
    def init_hidden(self):
        return torch.zeros(self.n_layers, 1, self.hidden_size).to(device)
```

## 5 Inputs and Targets

Each chunk will be turned into a tensor, specifically a `LongTensor` (used for integer values), by looping through the characters of the string and looking up the index of each character in `all_characters`.

[38]:
```
# Turn string into list of longs
def char_tensor(string):
    tensor = torch.zeros(len(string)).long().to(device)
    for c in range(len(string)):
        tensor[c] = all_characters.index(string[c])
    return tensor

print(char_tensor('abcDEF'))
```

```
tensor([10, 11, 12, 39, 40, 41], device='cuda:0')
```

Finally we can assemble a pair of input and target tensors for training, from a random chunk. The input will be all characters *up to the last*, and the target will be all characters *from the first*. So if our chunk is "abc" the input will correspond to "ab" while the target is "bc".

[39]:
```
def random_training_set():
    chunk = random_chunk()
    inp = char_tensor(chunk[:-1])
    target = char_tensor(chunk[1:])
    return inp, target
```

## 6 Evaluating

To evaluate the network we will feed one character at a time, use the outputs of the network as a probability distribution for the next character, and repeat. To start generation we pass a priming string to start building up the hidden state, from which we then generate one character at a time.

[40]:
```
def evaluate(prime_str='A', predict_len=100, temperature=0.8):
    hidden = decoder.init_hidden()
    prime_input = char_tensor(prime_str)
    predicted = prime_str

    # Use priming string to "build up" hidden state
    for p in range(len(prime_str) - 1):
        _, hidden = decoder(prime_input[p], hidden)
    inp = prime_input[-1]
```

```
    for p in range(predict_len):
        output, hidden = decoder(inp, hidden)

        # Sample from the network as a multinomial distribution
        output_dist = output.data.view(-1).div(temperature).exp()
        top_i = torch.multinomial(output_dist, 1)[0]

        # Add predicted character to string and use as next input
        predicted_char = all_characters[top_i]
        predicted += predicted_char
        inp = char_tensor(predicted_char)

    return predicted
```

## 7 Training

A helper to print the amount of time passed:

[41]:
```
import time, math

def time_since(since):
    s = time.time() - since
    m = math.floor(s / 60)
    s -= m * 60
    return '%dm %ds' % (m, s)
```

The main training function

[42]:
```
def train(inp, target):
    hidden = decoder.init_hidden()
    decoder.zero_grad()
    loss = 0

    for c in range(chunk_len):
        output, hidden = decoder(inp[c], hidden)
        loss += criterion(output, target.unsqueeze(1)[c])

    loss.backward()
    decoder_optimizer.step()

    return loss.item() / chunk_len
```

Then we define the training parameters, instantiate the model, and start training:

[43]:
```
n_epochs = 2000
print_every = 100
plot_every = 10
```

```
hidden_size = 100
n_layers = 1
lr = 0.005

decoder = RNN(n_characters, hidden_size, n_characters, n_layers).to(device)
decoder_optimizer = torch.optim.Adam(decoder.parameters(), lr=lr)
criterion = nn.CrossEntropyLoss()

start = time.time()
all_losses = []
loss_avg = 0

for epoch in range(1, n_epochs + 1):
    loss = train(*random_training_set())
    loss_avg += loss

    if epoch % print_every == 0:
        print('[%s (%d %d%%) %.4f]' % (time_since(start), epoch, epoch / ␣
 ↪n_epochs * 100, loss))
        print(evaluate('Wh', 100), '\n')

    if epoch % plot_every == 0:
        all_losses.append(loss_avg / plot_every)
        loss_avg = 0
```

```
[0m 27s (100 5%) 2.3300]
Wh. wount cas pur a flay sille what ame ar foud ris in toash thay and his ot
beas me. puithe sifink fo

[0m 54s (200 10%) 2.2410]
Wh9 ma!

SUS
ANO:

Bato to thim tas the ok whith sass whe his be mout, non nod sento he mitpeath
the n

[1m 22s (300 15%) 2.0112]
Wh|rinigise shered,
Wou, vich all bild thine be a all and hougre, ang go hall yout cherent comper,
The

[1m 49s (400 20%) 2.0123]
Wh; thy beather with ASlale,
Ay, he lever the shou I thou brighor the frith'k thou whellad will:
And t
```

[2m 18s (500 25%) 2.0550]
Whin wolknent:
Whis shose thile wearm sow wee,
An owes, bell dising thils, forl of sout thow deathe th

[2m 50s (600 30%) 1.9102]
Whbxo fare and doid.

CAEMENTENOLLO:
mahe hat not this ar schenter:
Her ondentrose, your youl sticcour

[3m 17s (700 35%) 2.0573]
Whe, at thise will of
yather not mandon, thou wellone thy leth theast this are.

ORK ESSAR:
How't: his

[3m 44s (800 40%) 1.9720]
Where:
And not the side wither the won the hims, a toing to pray,
That in mice this thet lif the to-de

[4m 12s (900 45%) 1.9247]
What if to me suck with theee.

Fill dees, but water pust sick:
I will the it you.

VARING:
I muth I m

[4m 39s (1000 50%) 1.9992]
Whemfor its.

MAMLET:
My now, o blour red.

BATONIO:
Your godd's it shather bo is doneladit us,
Wear w

[5m 7s (1100 55%) 1.8509]
Wheren his min
But to the menting: the good, do and heser,
This a hound the worpless,

```
That but dame a

[5m 35s (1200 60%) 1.8931]
Wh-khe though hear to with and this sweal my,
Cudge the k have to not sin lork my call.

CACBESTAN; LE

[6m 2s (1300 65%) 1.8850]
Wherire mattles doded of here
here preathine thing'd I lose and I man stime again peinge then sue.

CI

[6m 29s (1400 70%) 1.8885]
Whrought did havion, and plain this
them sir, hambloth aft not in though him checany
O, and for my mai

[7m 0s (1500 75%) 1.9638]
Whor outers is of everar.

PETIO:
Why me; I wust of be aver and pintind, be, Cack parch if the would s

[7m 27s (1600 80%) 1.9635]
Wh I'll me suffer loding esact.

POLIO:
In the may beal fathim of se man at will daught
tis look! Stou

[7m 54s (1700 85%) 1.8032]
WhiR:
The bears tign that your such hid with, and to him stork
And procompy the givell good made of my

[8m 21s (1800 90%) 1.8088]
Whe more
The post if it a gend your him:
We tus a have shousd us tanks,
When losk more in of the poor:

[8m 48s (1900 95%) 1.8369]
Whir, I wail: my the day,
For the to pley Water of the place eares gratent arowires
Have thou is mot t
```

```
[9m 16s (2000 100%) 1.8203]
What seome have frest and a will more?

MAO:
ATell shall I hoose her the gird in not luth keep not son
```

## 8 Plotting the Training Losses

Plotting the historical loss from all_losses shows the network learning:

[ ]:
```
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
%matplotlib inline

plt.figure()
plt.plot(all_losses)
```

## 9 Evaluating at different "temperatures"

In the `evaluate` function above, every time a prediction is made the outputs are divided by the "temperature" argument passed. Using a higher number makes all actions more equally likely, and thus gives us "more random" outputs. Using a lower value (less than 1) makes high probabilities contribute more. As we turn the temperature towards zero we are choosing only the most likely outputs.

We can see the effects of this by adjusting the `temperature` argument.

[ ]:
```
print(evaluate('Th', 200, temperature=0.8))
```

Lower temperatures are less varied, choosing only the more probable outputs:

[ ]:
```
print(evaluate('Th', 200, temperature=0.2))
```

Higher temperatures more varied, choosing less probable outputs:

[ ]:
```
print(evaluate('Th', 200, temperature=1.4))
```

[ ]:
```
import torch.nn.functional as F
def perp(testfile):
    inp = char_tensor(testfile[:-1])
    target = char_tensor(testfile[1:])
    test_len=len(testfile)
    hidden = decoder.init_hidden()
    decoder.zero_grad()
    perplexity=torch.tensor(0.0)
```

```
    for c in range(test_len-1):
        output, hidden = decoder(inp[c], hidden)
        perplexity -=F.log_softmax(output,dim=1)[0][target[c]]

    return (perplexity/test_len).exp().item()

testfile = unidecode.unidecode(open('data/shakespeare_sonnets.txt').read())
print('Perplexity:',perp(testfile))
```

### 9.1 FAQs

**I'm unfamiliar with PyTorch. How do I get started?** If you are new to the paradigm of computational graphs and functional programming, please have a look at this tutorial before getting started.

**How do I speed up training?** Send the model and the input, output tensors to the GPU using `.to(device)`. Refer the PyTorch docs for further information.

[ ]:

4.e $$\frac{K + c(w_{i-1} + w_i)}{K|V| + \sum_{w_i} c(w_{i-1} w_i)}$$

$|V| = 10$

$K|V| = 1$

Count(w) > 1 , skip "Pepperjack"

$$= \left(\frac{3+1}{5}\right)\left(\frac{2+1}{4}\right)\left(\frac{2+1}{5}\right)\left(\frac{1+1}{5}\right) = 0.07877$$

4d. The ~~Values~~ 5 optimal values for N, K such that perplexity is the lowest.

Certainty

4c. ~~Certainty~~ adding a smoothing factor helps the n-gram model. But the perplexity is still higher than what we would like. The issues comes down to unseen words or letters. When we are trainning.

for char in shakespeare_sonnet, N=2, K=0.15

Perplexity = 7.9722

for word model: Let N=2, K=0.15.

Perplexity = 53026.840

4b Since "F" and "In" are common in the training set, which means the probability of selecting these char/words are also higher. Hence, the begin with "F" and "In" when generating ~~words~~ text.

# NLP

ye. "Wher sharsting ins cheaure ence
With of these of whe stour slives mundir
Memeed with hero frame. sizh."

Perplexity = 1.6337

This is much better than ngram and
it usually more readable

NLP

1a) By applying chain rule, and Markov Property

NBiram = $P(w_1^n) = P(w_1)\,P(w_2|w_1)\,P(w_3|w_1^2) \ldots P(w_n|w_n^{n-1})$

$$= \prod_{K=1}^{n} P(w_K | w_K^{K-1})$$

for Bigram or $n=2$

$$P(w_1^n) = \prod_{K=1}^{n} P(w_K | w_{K-1})$$

1. B) $P(I|BOS)\ P(like|I)\ P(cheese|like)\ P(that|cheese)$
$P(is|that)\ P(salty|$

$P(I|BOS)\ P(like|I)\ P(cheese|like)\ P(made|cheese)$
$P(at|made)\ P(home|at)\ P(EOS|home)$

$= \frac{3}{4} \times \frac{2}{3} \times \frac{1}{2} \times \frac{1}{4} \times \frac{1}{3} \times \frac{2}{2} \times \frac{1}{3}$

$= 0.00694$

1c. Perplexity is used to assess a how well a model does at predicting a unseen sample.

$$\text{Perplexity} = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i|w_{i-1})}} = \sqrt[7]{\frac{1}{0.00697}} =$$

2.032870

NLP

1.d) $$\frac{K + c(w_{i-1}, w_i)}{K|V| + \sum_{w_i} c(w_{i-1}, w_i)} =$$

$K = 0.1$

$|V| = 10$

$K|V| = 1$

$$= \left(\frac{3.1}{5}\right)\left(\frac{2.1}{4}\right)\left(\frac{1.1}{3}\right)\left(\frac{1.1}{5}\right)\left(\frac{1.1}{4}\right)\left(\frac{2.1}{3}\right)\left(\frac{1.1}{4}\right)$$

$$= 0.001389$$

3 a) $a = W^{(1)}x + b^{(1)}$ ← $f$

$z = \sigma(a)$

$y = W^{(2)}z + b^{(2)}$

$\sigma(x) = \frac{1}{1+e^{-x}}$ Sigmoid function

$\tanh(x) = 2\sigma(2x) - 1$, tanh in relation to sigmoid

$f'$

$a = 2W^{1}x + 2b^{1}$

$z = \sigma(a)$

$y = 2W^{2}z - W^{2} - b^{2}$

3b. $h_{drop} = \gamma d \odot h \qquad d \in \{0,1\}^{D_h}$

$E_{p_{drop}}[h_{drop}]_i = h_i \qquad h \in R^{D_h}$

$\forall i \in \{1 \dots D_h\}$

$\gamma = \frac{h_{drop}}{d \odot h}$

$E_{p_{drop}}\left[\frac{h_{drop}}{d \odot h}\right]_i \quad \forall i \in \{1 \dots D_h\}$

$p_{drop}\left[\frac{h_{drop}}{d \odot h}\right]_1 + p_{drop}\left[\frac{h_{drop}}{d \odot h}\right]_2 \dots$

$\sum_{i=1}^{D_h} p_{drop}\left(\frac{h_{drop}}{d \odot h}\right)_i$

NLP HW2

2. a $C(w_{i-1}, w)$

$C(\text{computer}, \text{Store}) = 8$

$P(\text{Store} | \text{computer}) = \frac{8}{22} = \boxed{\frac{4}{11}}$

$C(\text{computer}, \text{monitor}) = 8$

$P(\text{monitor} | \text{computer}) = \frac{8}{22} = \boxed{4/11}$

~~$C(\text{computer}, \text{keyboard}) = 6$~~

~~$P(\text{keyboard} | \text{computer}) = \frac{6}{22} = \frac{3}{11}$~~

(c) Monitor and Store are equally likely.

2 b) $d = 0.5$

$W_i \in$ {Store, Monitor}

$W_{i-1}$ = Computer

$$\max\left(\frac{C(W_{i-1}, W_i) - d, 0)}{C(W_{i-1})} + \lambda(W_{i+1}) P_{continuation}(W_i)\right.$$

for ~~Store~~ Monitor we have:

$$\frac{8 - 0.5}{22} + \left(\frac{0.5}{22}(3)\right)\left(\frac{1}{9}\right) = 0.348$$

for ~~monitor~~ Store:

$$\frac{8 - 0.5}{22} + \left(\frac{0.5}{22}(3)\right)\left(\frac{3}{9}\right) = 0.363$$

The results have changed from previously. Now ~~Monitor~~ Store is more likely to ~~occur~~ occur after Computer.

This occurred because of $P_{continuation}$, which measures novelty occurrances.

NLP

$\delta = 0.1$

2c for store:

$$\frac{8-0.1}{22} + \left(\frac{.1}{22}\right)(3)\left(\frac{3}{9}\right) = 0.368$$

for monitor:

$$\frac{8-0.1}{22} + \left(\frac{0.1}{22}\right)(3)\left(\frac{1}{9}\right) = 0.360$$

As we change $\delta$ value the probability changes furthermore as $\delta$ decreases the probability approaches the raw bigram probabilities. And the larger $\delta$ value the more it take into account $P_{continuation}$ which further aparts the probabilities.

3 c) RNN is more useful for LM tasks, Because RNN takes into account sequences of input, where order Matters. And a Variance of RNN architecture, LSTM further improves accuracy in LM tasks. This is because LSTM deals with the Vanishing gradient problem that occurs in a regular RNN.