hw1_report

April 16, 2019

```
In [1]: import matplotlib.pyplot as plt
        %matplotlib inline
        import torch
        import torch.nn as nn
        import torch.nn.functional as F
        import torch.optim as optim
        import time
        import sys
        sys.path.insert(0, "../code")
        from LSTMlm import LSTMlm
        from LSTMlm_ctx import LSTMlm_ctx
        from data_pre import data_preprocessing,data_preprocessing_ctx
        from misc import common_error_pair,common_error_pair_ctx, plot_acc_neg,plot_acc_negchi
        from sklearn.manifold import TSNE
        import pickle
        import warnings
        warnings.filterwarnings('ignore')
        torch.manual_seed(1)
```

```
Out[1]: <torch._C.Generator at 0x10cfc0b70>
```

1 LSTM language model Trained with Hinge Loss

- I implemented using `../code/lstm_ll.py` using pytorch; the result is at `../output/lstm_ll.pyout`. Best model is stored at `../mode/lstmlm_ll.pt`
- I use LSTM with dimensionality 200 for both word embeddings and the LSTM cell/hidden vectors
- I use `Adam` optimizer with default learning rate (`lr = 1e-3`).
- I didn't use mini-batching
- I borrowed code from https://pytorch.org/tutorials/beginner/nlp/sequence_models_tutorial.html
- For early stopping, I didn't stop the algorithm when accuracy on dev drops. Instead I save the best model and let it run and observe its behavior.

2 Error Analysis

```
In [2]: ## load data
        (voc_ix, data_train, data_test, data_dev) = data_preprocessing()
```

```
        ix_voc = {}
        for k,i in voc_ix.items():
            ix_voc[i] = k
        PATH = "../model/lstmlm_ll.pt"
        ## evaluate data
        model1 = torch.load(PATH)
        print("performance of best model:")
        print("model acc on dev data: {}".format(model1.evaluate(data_dev)))
        print("model acc on test data: {}".format(model1.evaluate(data_test)))
```

```
performance of best model:
model acc on dev data: 0.3351765740857107
model acc on test data: 0.32559870951730985
```

2.0.1 Comment on results:

Best accuracy on development set (33 percent) is achieved at the end of epoch 2 (with around 12000 sentences), and then gradually drops to below 20 percent. So although total running time is 2076 seconds (for two epochs), we only need to run 200 seconds to get optimal.

2.0.2 Error pair analysis

Implemented at `code/misc.py`

```
In [3]: common_error1 = common_error_pair(model1, data_test, 35, ix_voc)
```

- Error categories and abbreviations

  ```
  (start of sentence: SOS)
  early stop: ES
  late stop: LS
  wrong person: WP
  right person: RP
  wrong verb: WV
  catch-all: CA
  wrong word type: WT
  ```

- Labeled Error:

```
(He Bob): 136; RP SOS
(She Bob): 112; WP SOS
(Sue Bob): 110; WP (SOS in most cases)
(to .): 58; ES
(and .): 58; ES
(decided was): 47; WV
(had was): 44; WV
(in .): 38; ES
```

```
(for .): 37; ES
(his the): 36; CA
(, .): 32; ES
(her the): 30; WP
(His Bob): 27; WT; SOS
(One Bob): 25; RP; SOS
(The Bob): 23; RP; SOS
(the .): 22; ES
(got was): 21; WV
(he Bob): 21; RP
(But Bob): 21; WT, SOS
(Her Bob): 21; WT, WP, SOS
(with .): 20; ES
(went was): 20; WV
('s was): 19; CA
(When Bob): 19; CA
(her a): 19; CA
(a the): 19; CA
(the a): 19;CA
(They Bob): 19; WP
(wanted was): 19; WV
(at .): 18; ES;
(! .): 18; CA
(the her): 17; CA
(it the): 16; WT
(. to): 16; LS
(of .): 16; ES
```

2.0.3 Comment:

Analysis of error (Note: For some error pairs I give them two labels.)

- The most frequent error are "start of sentence error". In fact, the model almost always predict "Bob" for start of sentence. This is because in training most sentences start with "Bob" so the model learns that "start of sentence" sign will most likely lead to "Bob".
- Similar to "Bob", "." is one of the most frequent words. So the model uses "." appropriately often. It is noteworthy that, there are not many misuse of End Of Sentence. Of course it is easy for the model to learn that. But that means the model fails to realize that "." always means the end of sentence.
- Many error use the wrong verb, typically "decided", "had" was predicted as "was". This is probably because "was" simply appeared more often, and they appeared at similar places (probably after a person, Bob/Sue).so the model gives more probability to it. In this sense, the model does well as it assigns the right type of word.

How can the model does better As we can see, the most frequent error are wrong start of sentence. Indeed without previous sentence, $P(\text{start with Bob})$ is the highest. But $P(\text{start with Bob}|\text{ previous sentence})$ will probably not be that high.

Visualization I also tried visualizing the word embedding from the model using TSNE. They are stored at `../docs/`. But it is not very helpful to find relationship between error pairs. Next time I will try to find their nearest neighbor.

3 Binary Loss Implementation and Experimentation

3.1 Implementation:

- The training is unstable using default Adam learning rate. It is easy to have accuracy suddenly drop to 0 with `nan` in weights
- So I decreased learning rate to `5*1e-04` and set `eps = 1e-3`. The latter is added to dividend to increase stability. I also clip the norm witn parameter 0.25. (I discussed with Young Joo Yun on this issue).
- The code is `"../code/lstm_neg.py"`
- The model is saved at `"../model/lstmlm_neg_f{}_r{}.pt"` with their corresponding (f,r)
- output file are at `"../output/lstm_neg_r{}_f{}.pyout"` with their corresponding (f,r)

3.2 UNIF

r	f	acc_dev	acc_test	#sent/sec	#sent for max acc	time f
20	0	0.2563	0.2451	42.7	102612	2403
100	0	0.2609	0.2499	38.2	115684	3031
500	0	0.2634	0.2505	23.7	109648	4617

I also estimated those measures for log loss:

#sent/sec	#sent for max acc	time for max acc
58.13	12036	200

3.2.1 Comment:

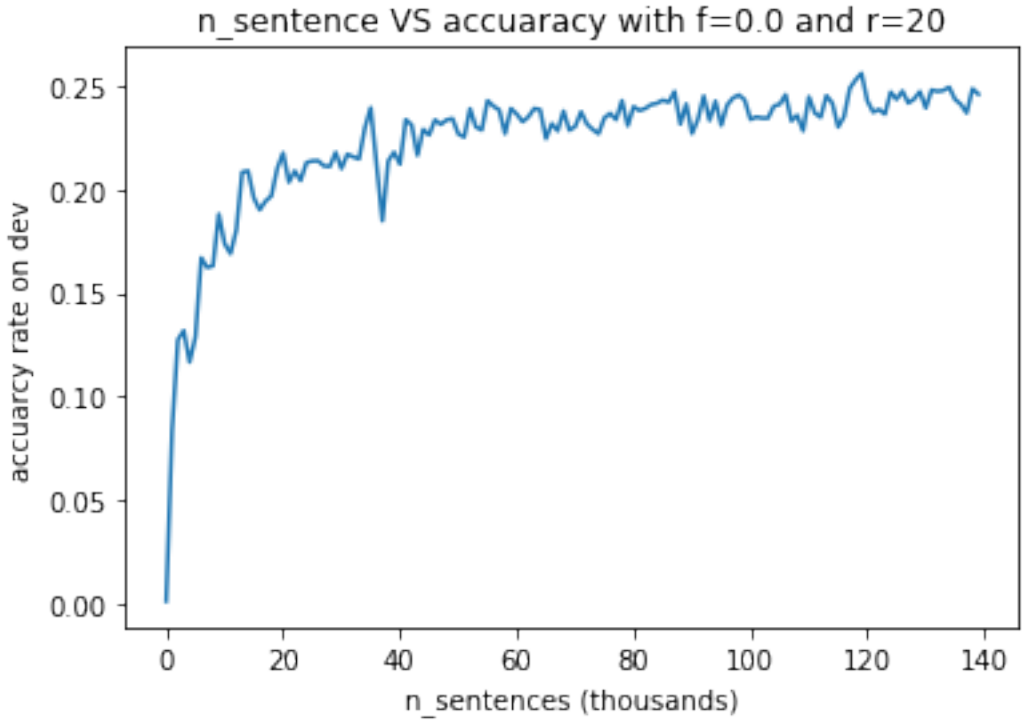
Comparison within negative sampling (the effect of r)

- Smaller r can be faster to process one sentence (bigger `#sent/sec`)
- It is not clear how r affact `#sent for max acc`. In the three experiments above, r seems not to affact `#sent for max acc` much. As a result, in this case, `time for max acc` is smaller with bigger `#sent/sec`.
- Before doing the experiments, I thought choosing r is a trade-off beteen speed per step, and the "progress" made per step; however, from the accuracy plot blow, we can see size of r does not affact "progress" that much. It does affect the variance during training: with small r, the accuracy trajectory experiences bigger climbing and falling.

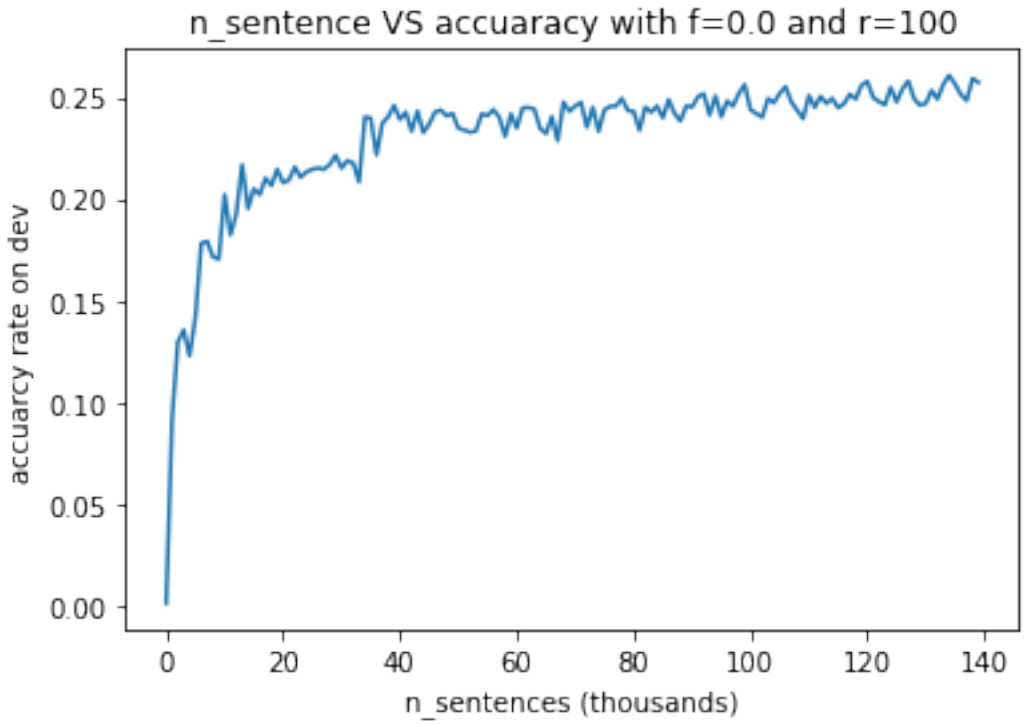
Comparison between log loss and binary log loss

- log loss does better in all those measures, and achieve much better accuracy.
- `#sent/sec` might be strongly affacted by my own implementation. If we only compare flop counts, binary log loss with negative sampling should be faster in processing one sentence.
- The `#sent for max acc` is much smaller than binary log loss (10^4 vs 10^5)

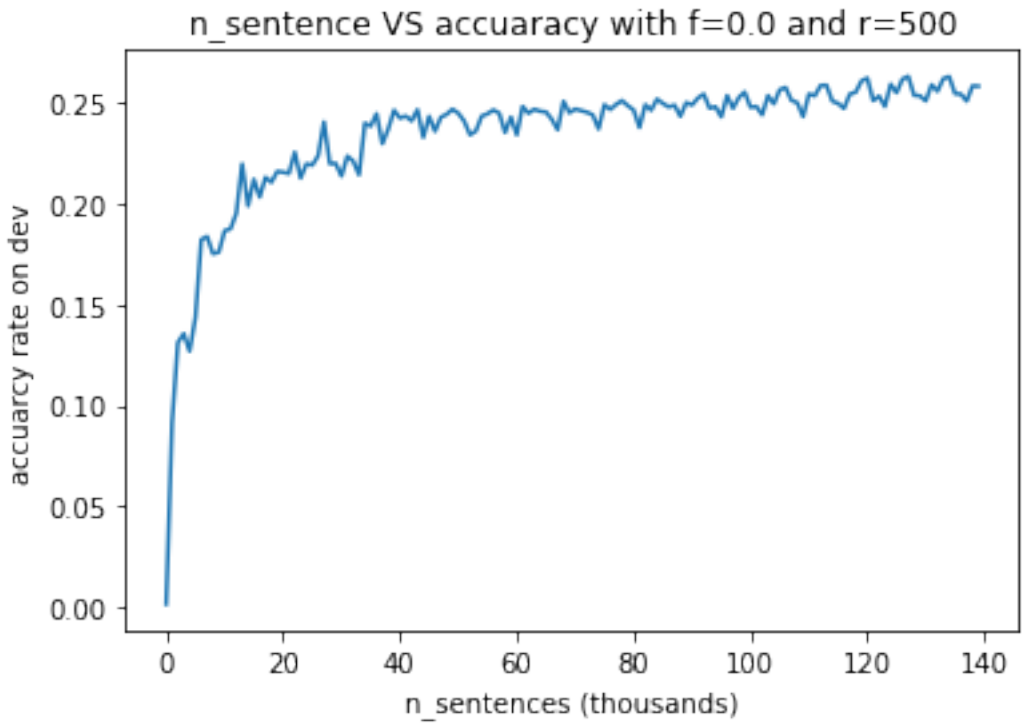
In [4]: `plot_acc_neg(0.0, 20)`



In [5]: `plot_acc_neg(0.0, 100)`



```
In [6]: plot_acc_neg(0.0, 500)
```



3.3 UNIG-f

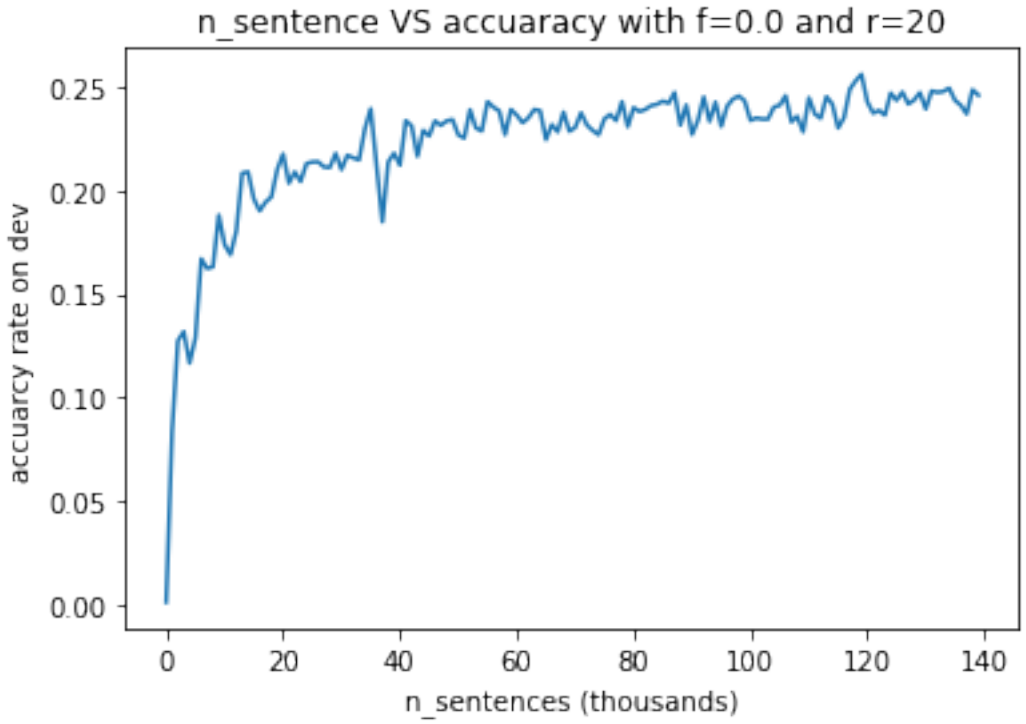
First, i tried f from 0.1 to 1 with step size 0.1. The results are as follow:

r	f	acc_dev	acc_test	#sent/sec	#sent for max acc	time fo
20	0.1	0.2324	0.2206	36.0	30144	836
20	0.2	0.2052	0.1997	43.7	14072	321
20	0.3	0.1827	0.1762	41.7	12036	288
20	0.4	0.1573	0.1493	41.7	19108	458
20	0.5	0.1355	0.1297	41.6	13072	314
20	0.6	0.1096	0.1090	46.5	14072	302
20	0.7	0.1026	0.1006	45.8	16072	350
20	0.8	0.0944	0.0930	44.6	22180	495
20	0.9	0.0934	0.0919	45.4	26144	576
20	1.0	0.0883	0.0858	47.92	13072	272

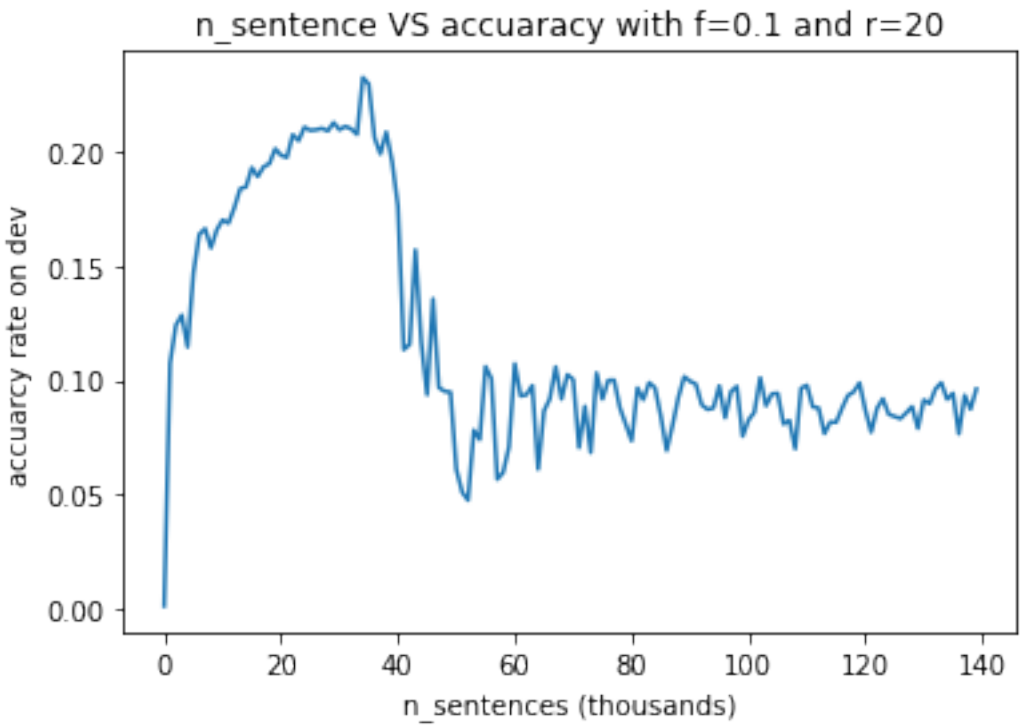
- None of them perform better than UNIF
- It is noteworthy that maximum accuracy on dev is achieved in around 1 or 2 or 3 epochs, whereas in when $f = 0$ it takes more than 10 epochs to achieve maximum accuracy.

- Therefore, there might be problem with optimization. Below I show accuracy plot of one model

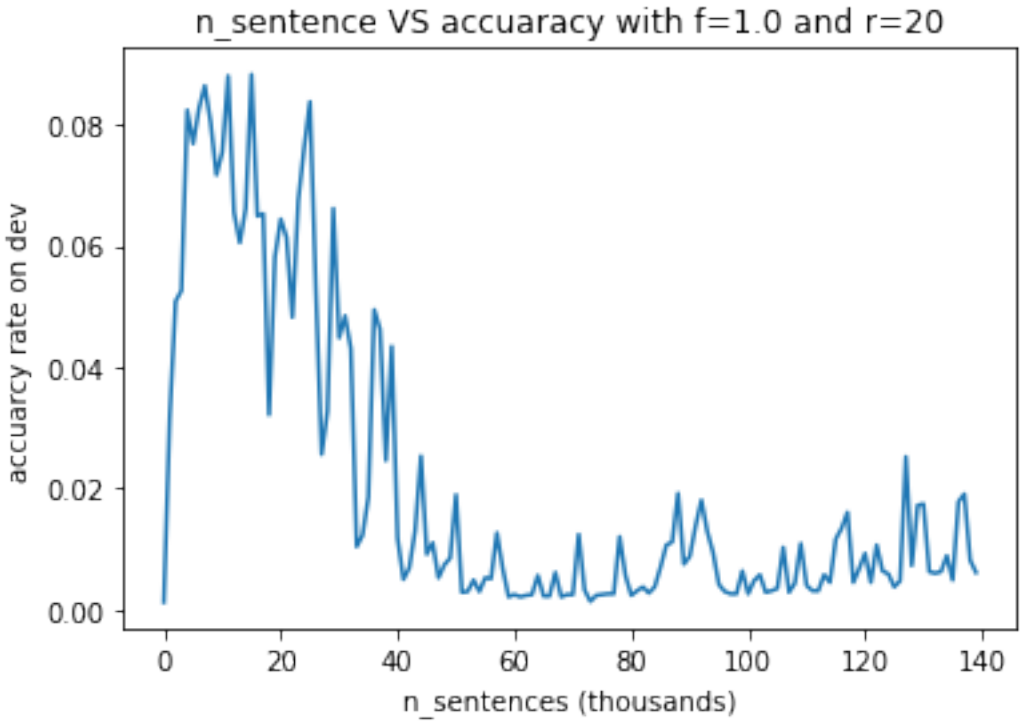
```
In [7]: plot_acc_neg(f = 0.0, r =20)
```




```
In [8]: plot_acc_neg(f = 0.1, r =20)
```



In [9]: `plot_acc_neg(f = 1.0, r =20)`



3.3.1 Optimization issue:

- From the plots above, I can see that when $f > 0$, there is a serious optimization problem in training. When $f = 0$, the problem is less severe.
- I have decreased my learning rate to $5 * 10^{-4}$ and set $eps = 10^{-3}$ and add `clip_norm` to increase stability. However, they seem not to be enough to stabilize training.
- If I had more time, I would try $r = 500$ or even larger and use a even smaller learning rate.

4 Using Context

- I implemented using `../code/lstm_ll_ctx.py` using pytorch; the result is at `../output/lstm_ll_ctx.pyout`. Best model is stored at `../mode/lstmlm_ll_ctx.pt`
- I use the same model hyperparameters as the first model.

```
In [10]: ## load data
         (voc_ix, data_train_ctx, data_test_ctx, data_dev_ctx) = data_preprocessing_ctx()
         ix_voc = {}
         for k,i in voc_ix.items():
             ix_voc[i] = k

         PATH = "../model/lstmlm_ll_ctx.pt"

         ## evaluate data
         model3 = torch.load(PATH)
         print("performance of best model:")
         print("model acc on dev data: {}".format(model3.evaluate(data_dev_ctx)))
         print("model acc on test data: {}".format(model3.evaluate(data_test_ctx)))
```

```
performance of best model:
model acc on dev data: 0.36433329144149806
model acc on test data: 0.3588534557637424
```

4.0.1 Error pair analysis

Implemented at `code/misc.py`

```
In [11]: common_error2 = common_error_pair_ctx(model3, data_test_ctx, 35, ix_voc)
```

```
(to .): 68; ES
(and .): 57; ES
(had was): 46; WV
(decided was): 38; WV
(for .): 37; ES
```

```
(Bob He): 36; RP; SOS
(, .): 32; ES;
(in .): 29; ES
(his the): 27; CA
(the her): 27;CA
(Bob Sue): 27;RP
(Sue She): 24;RP
(her the): 24; CA
(His He): 22; WT
(Her She): 21; WT
(the .): 21; ES
(went was): 20; WV
(a the): 20; CA
(wanted was): 20; WV
(got was): 18; WV
(! .): 18; ES
(on .): 18; ES
(Sue Bob): 17; WP
(with .): 16; ES
('s was): 16; CA
(at .): 16; ES
(her a): 15; CA
(. to): 15; LS
(the a): 14; CA
(a .): 14; ES
(asked was): 14; WV
(the his): 13; CA
(didn was): 13; CA
(it the): 12; CA
(She Sue): 12; RP
```

```
In [12]: print("Comparison between with and without context")
         print("(yg, yp): number w/ ctx VS number w/o ctx")
         print("##########################################")
         for ep1 in common_error1:
             for ep2 in common_error2:
                 if ep1[0] == ep2[0]:
                     print("({} {}): {} vs {}".format(ix_voc[ep1[0][0]], ix_voc[ep1[0][1]], ep
                     break
             print("({} {}): {} vs < 16".format(ix_voc[ep1[0][0]], ix_voc[ep1[0][1]], ep1[1],
```

```
Comparison between with and without context
(yg, yp): number w/ ctx VS number w/o ctx
##########################################
(He Bob): 136 vs < 16
(She Bob): 112 vs < 16
(Sue Bob): 110 vs 17
```

```
(Sue Bob): 110 vs < 16
(to .): 58 vs 68
(to .): 58 vs < 16
(and .): 58 vs 57
(and .): 58 vs < 16
(decided was): 47 vs 38
(decided was): 47 vs < 16
(had was): 44 vs 46
(had was): 44 vs < 16
(in .): 38 vs 29
(in .): 38 vs < 16
(for .): 37 vs 37
(for .): 37 vs < 16
(his the): 36 vs 27
(his the): 36 vs < 16
(, .): 32 vs 32
(, .): 32 vs < 16
(her the): 30 vs 24
(her the): 30 vs < 16
(His Bob): 27 vs < 16
(One Bob): 25 vs < 16
(The Bob): 23 vs < 16
(the .): 22 vs 21
(the .): 22 vs < 16
(got was): 21 vs 18
(got was): 21 vs < 16
(he Bob): 21 vs < 16
(But Bob): 21 vs < 16
(Her Bob): 21 vs < 16
(with .): 20 vs 16
(with .): 20 vs < 16
(went was): 20 vs 20
(went was): 20 vs < 16
('s was): 19 vs 16
('s was): 19 vs < 16
(When Bob): 19 vs < 16
(her a): 19 vs 15
(her a): 19 vs < 16
(a the): 19 vs 20
(a the): 19 vs < 16
(the a): 19 vs 14
(the a): 19 vs < 16
(They Bob): 19 vs < 16
(wanted was): 19 vs 20
(wanted was): 19 vs < 16
(at .): 18 vs 16
(at .): 18 vs < 16
(! .): 18 vs 18
```

```
(! .): 18 vs < 16
(the her): 17 vs 27
(the her): 17 vs < 16
(it the): 16 vs 12
(it the): 16 vs < 16
(. to): 16 vs 15
(. to): 16 vs < 16
(of .): 16 vs < 16
```

4.0.2 Comment:

Many "start of sentence" error are corrected, and when they appeared, they mostly referred to the same person ("Bob", "He").

Many other error, like the wrong verb used and early stopping sentence with ".", are not corrected.

5 Hinge Loss Implementation

5.1 Implementation:

- The training is unstable using default Adam learning rate. It is easy to have accuracy suddenly drop to 0 with `nan` in weights
- So I decreased learning rate to `5*1e-04` and set `eps = 1e-3`. The latter is added to dividend to increase stability. I also clip the norm witn parameter 0.25. (I discussed with Young Joo Yun on this issue).
- The code is `"../code/lstm_neg_chinge.py"`
- The model is saved at `"../model/lstm_neg_chinge_f{}_r{}.pt"` with their corresponding (f,r)
- output file are at `"../output/lstm_neg_chinge_r{}_f{}.pyout"` with their corresponding (f,r)

5.2 UNIF

r	f	acc_dev	acc_test	#sent/sec	#sent for max acc	time fo
20	0	0.2629	0.2474	38.9	99576	2559
100	0	0.2769	0.2624	36.3	103612	2855
500	0	0.2761	0.2578	24.1	120684	5018

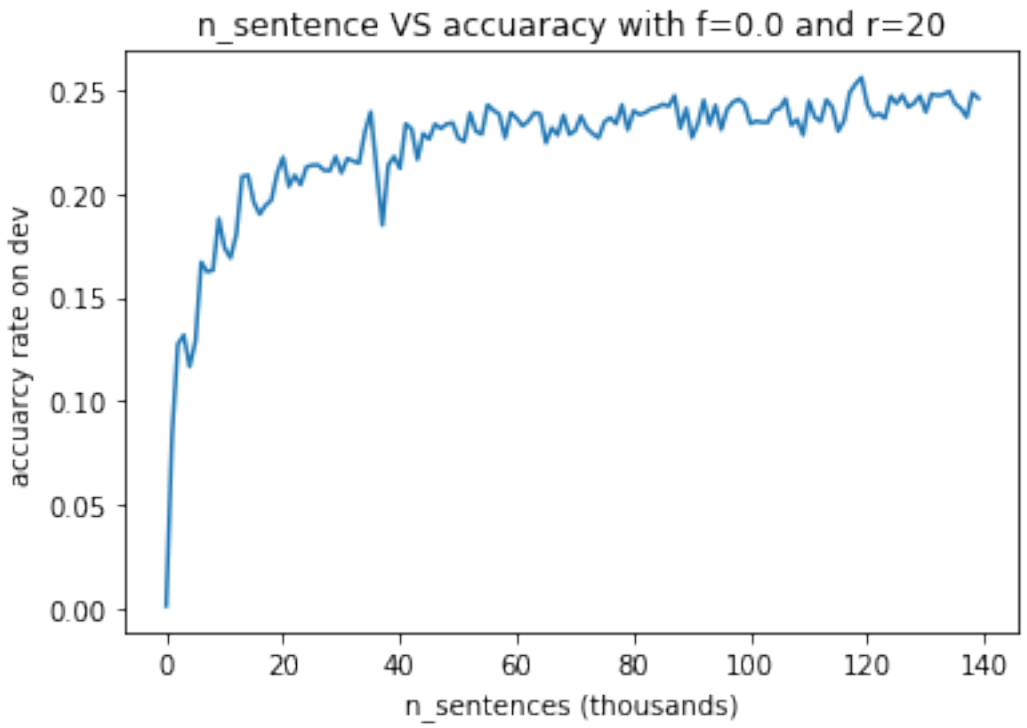
I also estimated those measures for log loss:

#sent/sec	#sent for max acc	time for max acc
58.13	12036	200

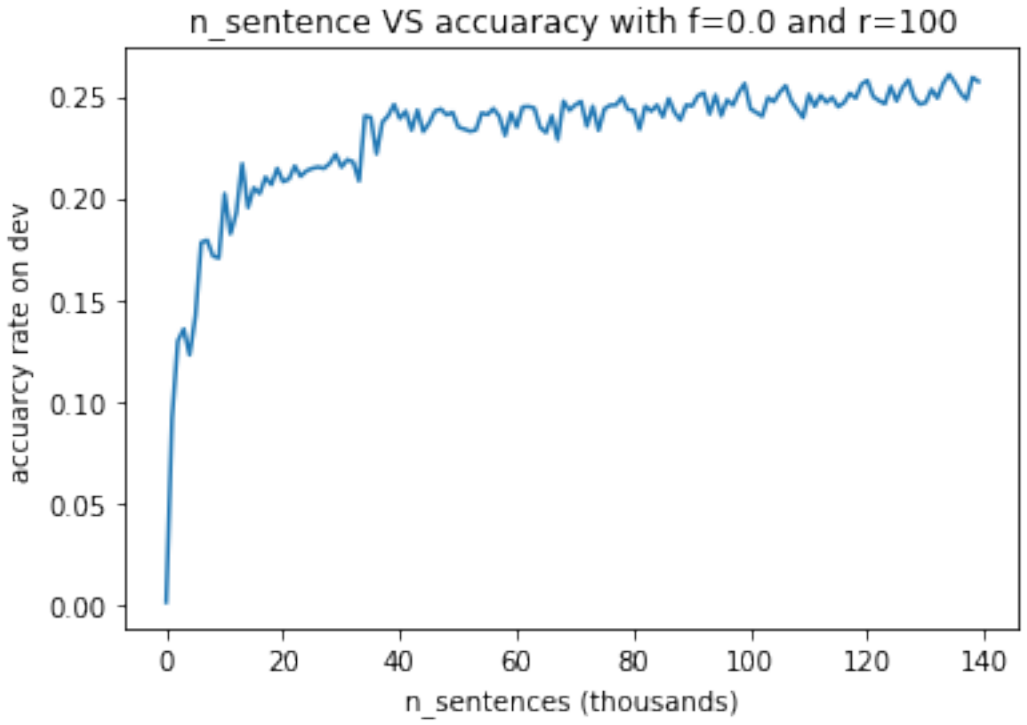
5.2.1 Comment:

My conclulsion about the three measure are simil;

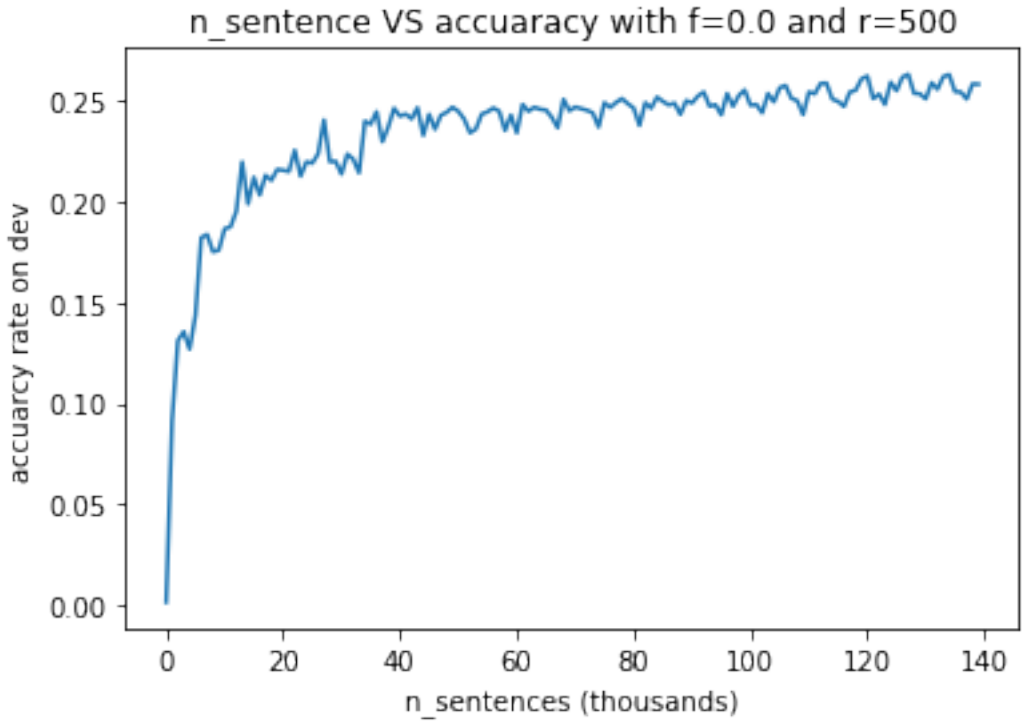
```
In [13]: plot_acc_negchinge(0.0, 20)
```



In [14]: `plot_acc_negchinge(0.0, 100)`



```
In [15]: plot_acc_negchinge(0.0, 500)
```



5.3 UNIG-F

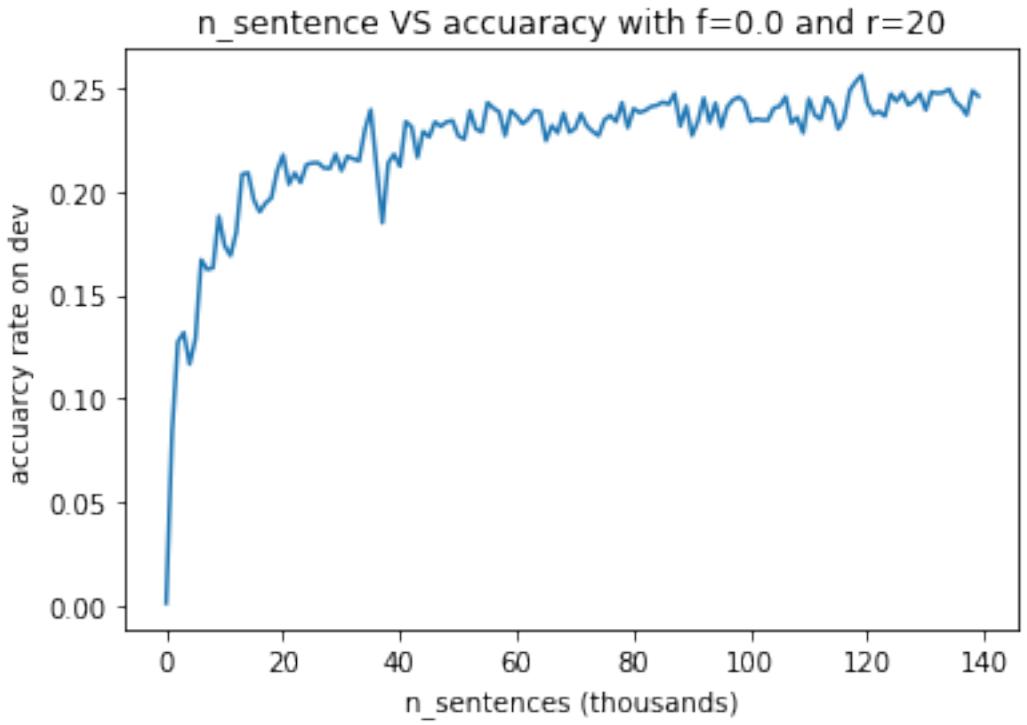
r	f	acc_dev	acc_test	#sent/sec	#sent for max acc	time f
20	0.1	0.2613	0.2490	44.8	115684	2584
20	0.2	0.2575	0.2377	43.8	119684	2733
20	0.3	0.2557	0.2387	45.1	108612	2410
20	0.4	0.2391	0.2227	46.4	119684	2581
20	0.5	0.2213	0.2108	45.2	109648	2423
20	0.6	0.1956	0.1894	45.8	120684	2636
20	0.7	0.1816	0.1655	43.7	115684	2647
20	0.8	0.1625	0.1536	46.8	106612	2276
20	0.9	0.1509	0.1433	45.0	114648	2546
20	1.0	0.1391	0.1332	45.0	96540	2145

5.3.1 Comment:

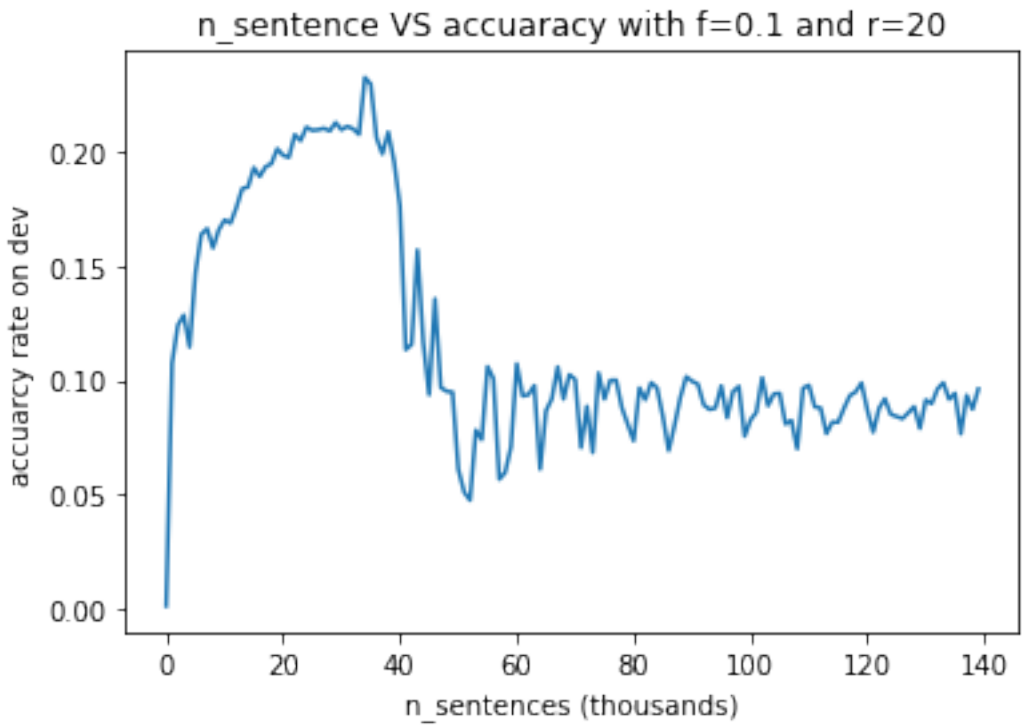
- For $r = 20$, UNIG-F beats UNIF (f = 0.1) in accuracy on test set (the advantage is pretty small).

- While in binary log loss, accuracy drops drastically as f gets larger, hinge loss is slightly better. The plot below show training manages to get high (compared with binary loss) before it drops drastically.

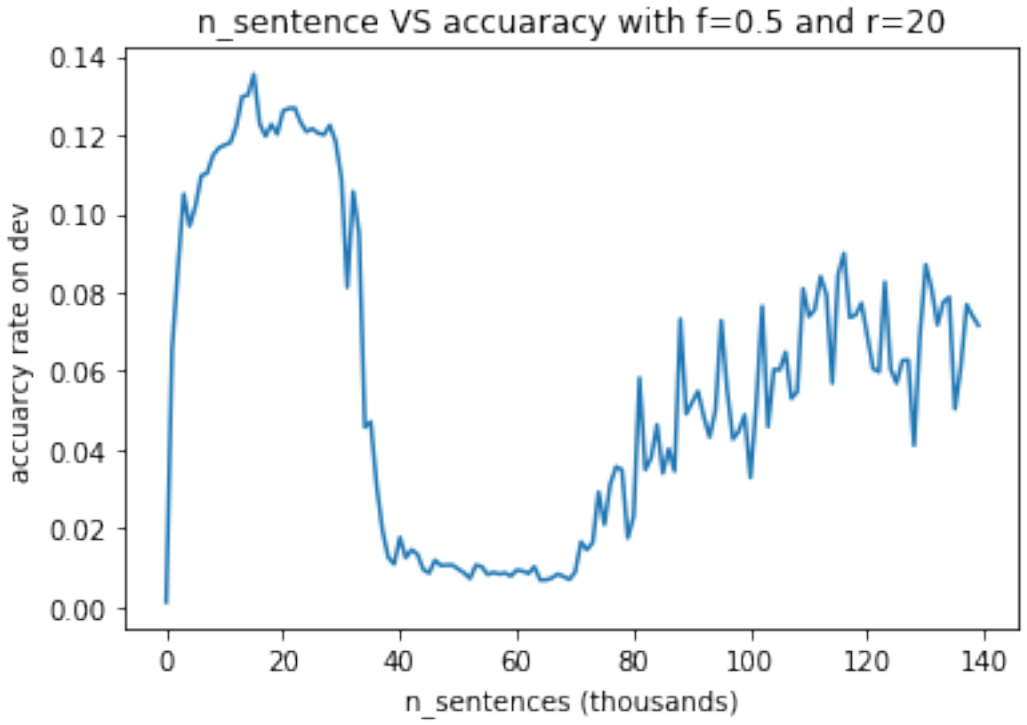
In [16]:
```
plot_acc_negchinge(f = 0.0, r = 20)
```



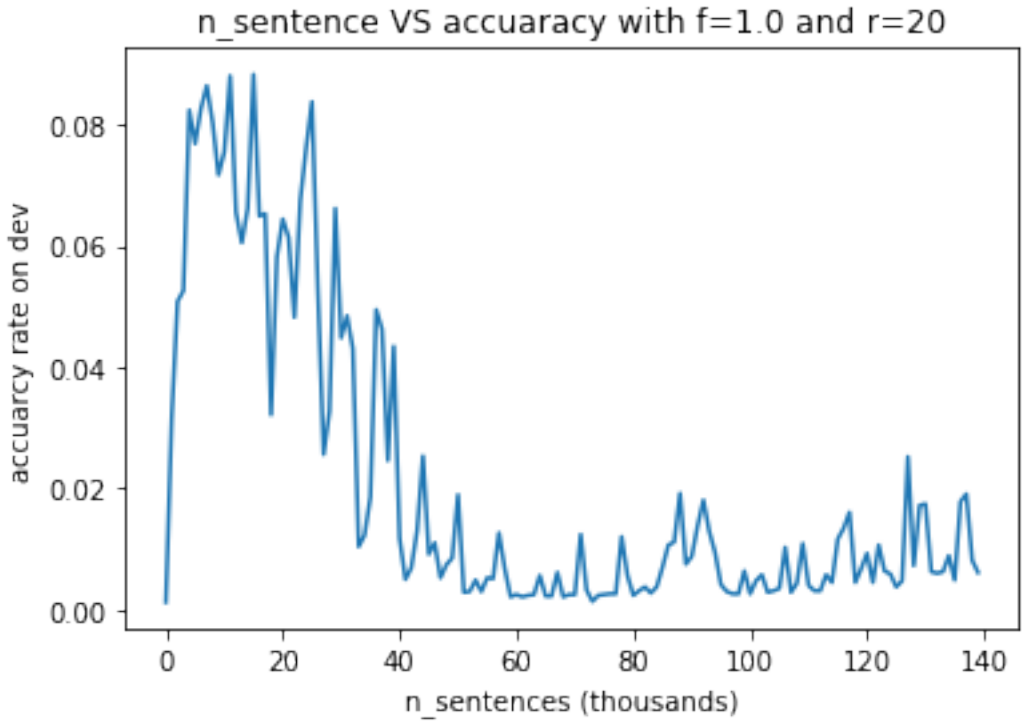
In [17]:
```
plot_acc_negchinge(f = 0.1, r = 20)
```



In [18]:
```
plot_acc_negchinge(f = 0.5, r = 20)
```



```
In [19]: plot_acc_negchinge(f = 1.0, r = 20)
```



6 Reflection

6.1 Interesting mistakes

I made a few mistakes during the experiments, some of which are meaningful:

- I misused the cross entropy loss function in pytorch. It automatically turns score into probability through softmax. Without knowing it, I did softmax over sccore and as the probability as input. In effect, my model does softmax twice to compute log loss. As a result, my model gets 30 percent accuracy both with and without context. Probably because double softmax softens the signal too much, the update cannot make enough progress. (Also, I have been thinking if there is an alternative to softmax that can turn score into probability, which may work better.)
- I foolishly set seed for my random sampling,and the accuracy is just one percent! This makes sense because it is only trained on those small subset of negative samples...

6.2 Remaining issues and future explorations

- I think there is still a lot of room for improvement on both binary loss and hinge loss by tunning the parameters
- Maybe we can sample negative samples with weights proportional to the frequency of error pair. Or maybe we can train with log loss first, then switch to that sampling scheme? It may work since it seems to be able to force the model to make corrections on previous frequent error.