C3W1_perplexity

July 15, 2025

1 Calculating perplexity using numpy: Ungraded Lecture Notebook

In this notebook you will learn how to calculate perplexity. You will calculate it from scratch using `numpy` library. First you can import it and set the random seed, so that the results will be reproducible.

[1]:
```
import numpy as np

# Setting random seeds
np.random.seed(32)
```

1.1 Calculating Perplexity

The perplexity is a metric that measures how well a probability model predicts a sample and it is commonly used to evaluate language models. It is defined as:

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

Where $P()$ denotes probability and w_i denotes the i-th word, so $P(w_i|w_1, ..., w_{i-1})$ is the probability of word i, given all previous words (1 to $i-1$). As an implementation hack, you would usually take the log of that formula (so the computation is less prone to underflow problems). You would also need to take care of the padding, since you do not want to include the padding when calculating the perplexity (to avoid an artificially good metric).

After taking the logarithm of $P(W)$ you have:

$$\log P(W) = \log \left(\sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i|w_1, ..., w_{i-1})}} \right)$$

$$= \log \left(\left(\prod_{i=1}^{N} \frac{1}{P(w_i|w_1, ..., w_{i-1})} \right)^{\frac{1}{N}} \right)$$

$$= \log\left(\left(\prod_{i=1}^{N} P(w_i|w_1,...,w_{i-1})\right)^{-\frac{1}{N}}\right)$$

$$= -\frac{1}{N}\log\left(\prod_{i=1}^{N} P(w_i|w_1,...,w_{i-1})\right)$$

$$= -\frac{1}{N}\sum_{i=1}^{N} \log P(w_i|w_1,...,w_{i-1})$$

You will be working with a real example from this week's assignment. The example is made up of: - `predictions` : log probabilities for each element in the vocabulary for 32 sequences with 64 elements (after padding). - `targets` : 32 observed sequences of 64 elements (after padding).

[2]:
```
# Load from .npy files
predictions = np.load('predictions.npy')
targets = np.load('targets.npy')

# Print shapes
print(f'predictions has shape: {predictions.shape}')
print(f'targets has shape: {targets.shape}')
```

```
predictions has shape: (32, 64, 256)
targets has shape: (32, 64)
```

Notice that the predictions have an extra dimension with the same length as the size of the vocabulary used.

Because of this you will need a way of reshaping `targets` to match this shape. For this you can use `np.eye()`, which you can use to create one-hot vectors.

Notice that `predictions.shape[-1]` will return the size of the last dimension of `predictions`.

[3]:
```
reshaped_targets = np.eye(predictions.shape[-1])[targets]
print(f'reshaped_targets has shape: {reshaped_targets.shape}')
```

```
reshaped_targets has shape: (32, 64, 256)
```

By calculating the product of the predictions and the reshaped targets and summing across the last dimension, the total log probability of each observed element within the sequences can be computed:

[4]:
```
log_p = np.sum(predictions * reshaped_targets, axis= -1)
```

Now you will need to account for the padding so this metric is not artificially deflated (since a lower perplexity means a better model). For identifying which elements are padding and which are not, you can use `np.equal()` and get a tensor with `1s` in the positions of actual values and `0s` where there are paddings.

[5]:
```
non_pad = 1.0 - np.equal(targets, 0)
print(f'non_pad has shape: {non_pad.shape}\n')
print(f'non_pad looks like this: \n\n {non_pad}')
```

```
non_pad has shape: (32, 64)

non_pad looks like this:

 [[1. 1. 1. … 0. 0. 0.]
 [1. 1. 1. … 0. 0. 0.]
 [1. 1. 1. … 0. 0. 0.]
 …
 [1. 1. 1. … 0. 0. 0.]
 [1. 1. 1. … 0. 0. 0.]
 [1. 1. 1. … 0. 0. 0.]]
```

By computing the product of the log probabilities and the non_pad tensor you remove the effect of padding on the metric:

[6]:
```
real_log_p = log_p * non_pad
print(f'real log probabilities still have shape: {real_log_p.shape}')
```

```
real log probabilities still have shape: (32, 64)
```

You can check the effect of filtering out the padding by looking at the two log probabilities tensors:

[7]:
```
print(f'log probabilities before filtering padding: \n\n {log_p}\n')
print(f'log probabilities after filtering padding: \n\n {real_log_p}')
```

```
log probabilities before filtering padding:

 [[ -5.39654493  -1.03111839  -0.66916656 … -22.37672997 -23.18770981
  -21.84348297]
 [ -4.58577061  -1.13412857  -8.53803253 … -20.15686035 -26.83709717
  -23.57501984]
 [ -5.22238874  -1.28241444  -0.17312431 … -21.328228   -19.85441208
  -33.88444138]
 …
 [ -5.39654493 -17.29168129  -4.36076593 … -20.82580185 -21.06583786
  -22.44311523]
 [ -5.93131638 -14.24741745  -0.26373291 … -26.74324799 -18.38433075
  -22.35527802]
 [ -5.67053604  -0.10595131   0.         … -23.33252335 -28.08737564
  -23.87880707]]

log probabilities after filtering padding:

 [[ -5.39654493  -1.03111839  -0.66916656 …  -0.          -0.
   -0.        ]
```

```
 [ -4.58577061  -1.13412857  -8.53803253 …  -0.          -0.
   -0.        ]
 [ -5.22238874  -1.28241444  -0.17312431 …  -0.          -0.
   -0.        ]
…
 [ -5.39654493 -17.29168129  -4.36076593 …  -0.          -0.
   -0.        ]
 [ -5.93131638 -14.24741745  -0.26373291 …  -0.          -0.
   -0.        ]
 [ -5.67053604  -0.10595131   0.         …  -0.          -0.
   -0.        ]]
```

Finally, to get the average log perplexity of the model across all sequences in the batch, you will sum the log probabilities in each sequence and divide by the number of non padding elements (which will give you the negative log perplexity per sequence). After that, you can get the mean of the log perplexity across all sequences in the batch.

[8]:
```
log_ppx = np.sum(real_log_p, axis=1) / np.sum(non_pad, axis=1)
log_ppx = np.mean(-log_ppx)
print(f'The log perplexity and perplexity of the model are respectively:␣
  ↪{log_ppx} and {np.exp(log_ppx)}')
```

```
The log perplexity and perplexity of the model are respectively:
2.6211854987065033 and 13.752016923578548
```

Congratulations on finishing this lecture notebook! Now you should have a clear understanding of how to compute the perplexity to evaluate your language models. **Keep it up!**

[ ]:
```
```