L4: Language Model Problems

Frederik Evenepoel & Piotr Bonar - Pair K
17/03/2025

In the code, following libraries are used:

```
import nltk
from nltk.corpus import gutenberg as corpus
import Levenshtein
import random
import re
from nltk.lm.preprocessing import padded_everygram_pipeline, pad_both_ends
from nltk.lm import MLE, Laplace, StupidBackoff
from nltk.lm.api import LanguageModel
from nltk.metrics.distance import edit_distance
from typing import List, Dict
import numpy as np
from tqdm.auto import tqdm
from nltk.util import ngrams
import torch as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
```

Exercise 1: Implementation of spelling corrector:

Part 1: Change one word from the test sentences with a random in-vocabulary word (with word alternations at Levenshtein distance 1)

The Gutenberg corpus from NLTK is preprocessed by removing special characters and numbers. After preprocessing, fifty sentences are selected as test data, while the remaining sentences are used for training. A vocabulary of unique words is also created. To introduce noise, a single word in each test sentence is randomly selected and replaced with a 'similar word' from the vocabulary, ensuring a Levenshtein distance of 1. Below, we present five original test sentences alongside their modified versions, demonstrating how the implementation replaces a single word with an appropriate alternative. The implementation of a random word/possible candidate from the vocabulary as replacement can be seen in the code ('*random.choise(candidates)'*).

Examples:

Original: emma woodhouse handsome clever and rich with a comfortable home and happy disposition seemed to unite some of the best blessings of existence and had lived nearly twenty one years in the world with very little to distress or vex her

Altered : emma woodhouse handsome clever and rich with a comfortable home and happy disposition seemed to unite some of the best blessings of existence and had lived nearly twenty one years in the world with very little to ***mistress*** *or vex her*

Original: she was the youngest of the two daughters of a most affectionate indulgent father and had in consequence of her sister ' s marriage been mistress of his house from a very early period

*Altered : **the** was the youngest of the two daughters of a most affectionate indulgent father and had in consequence of her sister ' s marriage been mistress of his house from a very early period*

Original: emma by jane austen

*Altered : emma **bye** jane austen*

Original: sixteen years had miss taylor been in mr woodhouse ' s family less as a governess than a friend very fond of both daughters but particularly of emma

*Altered : sixteen **tears** had miss taylor been in mr woodhouse ' s family less as a governess than a friend very fond of both daughters but particularly of emma*

Original: between them it was more the intimacy of sisters

*Altered : **betweens** them it was more the intimacy of sisters*

Implementation code

```
# Exercise 1

nltk.download("punkt_tab")
nltk.download("gutenberg")

CLEANUP_REGEXES = [
    (re.compile(r"[0-9]"), ""),
    (re.compile(r"[_*-]?([A-Za-z]+)[_*-]?"), lambda match: match.group(1)),
    (re.compile(r"[\-_!?*&.\[\](){}\",:;]") , ""),
]

def preprocess_words(word: str) -> str | None:
    word = word.lower().strip()
    for regex, repl in CLEANUP_REGEXES:
        word = regex.sub(repl, word)
    return word if word else None

preprocessed_text = [
    [preprocess_words(word) for word in sent if preprocess_words(word) is not None]
    for sent in corpus.sents()[:1000]
]

preprocessed_text = [sent for sent in preprocessed_text if sent]
vocab = sorted(set(word for sent in preprocessed_text for word in sent if word.isalpha()))
vocab.append("'")

test_sentences = preprocessed_text[:50]
train_sentences = preprocessed_text[50:]

def get_similar_word(word, vocab):
    wordx = preprocess_words(word)
    if not wordx:
        return word.lower()
    candidates = [w for w in vocab if Levenshtein.distance(w, wordx) == 1]
    return random.choice(candidates) if candidates else wordx

def generate_altered_sentences(test_sentences, vocab):
```

```
    altered_sentences = []
    original_sentences = []

    for sentence in test_sentences:
        processed_sentence = sentence.copy()
        original_sentences.append(" ".join(processed_sentence))

        random.shuffle(processed_sentence)
        word_to_replace, replacement = None, None

        for word in processed_sentence:
            replacement_candidates = [w for w in vocab if Levenshtein.distance(w, word) == 1]
            if replacement_candidates:
                word_to_replace = word
                replacement = random.choice(replacement_candidates)
                break

        altered_sentence = [
            replacement if w == word_to_replace else w for w in sentence
        ] if word_to_replace and replacement else sentence

        altered_sentences.append(" ".join(altered_sentence))

    return original_sentences, altered_sentences

original_sentences, altered_sentences = generate_altered_sentences(test_sentences, vocab)
for i in range(min(7, len(original_sentences))):
    print(f"Original: {original_sentences[i]}")
    print(f"Noisy: {altered_sentences[i]}")
    print("-" * 80)
```

Part 2: Implement and train an *n*-gram language model using the NLTK package. Try using trigrams and bigrams and variations seen in class

In the code below, trigram language models are built using Maximum Likelihood Estimation, Laplace smoothing, and Stupid Backoff. Padded n-grams are used to handle sentence boundaries. Each model learns word probabilities based on preceding words in a sentence.

The MLE model estimates probabilities by dividing trigram counts by their corresponding bigram counts. However, it assigns zero probability to unseen words. Laplace smoothing mitigates this by adding 1 to all word counts, ensuring no word gets a probability of zero, though it can sometimes overestimate rare words. Stupid Backoff handles unseen words by backing off to lower-order n-grams (e.g., trigrams to bigrams or unigrams), applying a fixed scaling factor rather than true probability discounting. The models are trained using '*model.fit(ngrams, vocab)*', where ngrams are generated using *'padded_everygram_pipeline'*.

Implementation code

```
n = 3

train_data1, vocab1 = padded_everygram_pipeline(n, processed_train)
train_data2, vocab2 = padded_everygram_pipeline(n, processed_train)
train_data3, vocab3 = padded_everygram_pipeline(n, processed_train)

lm_mle = MLE(n)
lm_lpc = Laplace(n)
lm_sbo = StupidBackoff(alpha=0.4, order=n)
lm_mle.fit(train_data1, vocab1)
lm_lpc.fit(train_data2, vocab2)
lm_sbo.fit(train_data3, vocab3)
```

Part 3: Sampling from various language models and perplexity evaluation and discussion

After training, each model is used to generate a 10-word sequence, starting with the seed word <s>. The implementation (code) follows the provided pseudocode. We observe that MLE and Stupid Backoff often generate identical sentences because SBO only falls back to lower-order n-grams when needed, which doesn't happen when most trigrams exist in the dataset. In contrast, Laplace smoothing slightly adjusts word probabilities by adding +1 to all counts, ensuring no word has zero probability and slightly increasing the likelihood of rare words appearing. As a result, Laplace generates different sentence outputs compared to the other models. We can also conclude that the sentences with n = 3 make more sense because of more contextual understanding.

N = 2:

MLE generated: oh <s> counsellor man she thinks strongly and in a
Laplace generated: one ' s daughter or seven or an evening and
StupidBackoff generated: oh <s> counsellor man she thinks strongly and in a

N = 3:

MLE generated: he began speaking of her own universal good will <s>
Laplace generated: how can emma imagine she would probably repent it <s>
StupidBackoff generated: he began speaking of her own universal good will <s>

After this, we computed the perplexity of the models. The average perplexity was determined by processing all test sentences and converting them into n-grams. Each sentence's perplexity was calculated using the language model, excluding sentences with infinite perplexity (i.e., those assigned zero probability). We then summed the perplexity scores of the remaining sentences and counted how many contributed to the total. Finally, we obtained the average by dividing the total by the number of contributing sentences, or set it to infinity if none were valid. The results are shown below.

Sentences with non-infinite perplexity (MLE) = 0
Average Perplexity (MLE): inf
Sentences with non-infinite perplexity (Laplace) = 5
Average Perplexity (Laplace): 1813.02
Sentences with non-infinite perplexity (Stupid Backoff) = 2
Average perplexity (Stupid Backoff): 283.15

These results indicate that many sentences have infinite perplexity, meaning the models assigned them zero probability and excluded them from the calculation. Although Stupid Backoff has the lowest average perplexity among the finite cases, the small number of valid sentences prevents us from confidently concluding that it is the best model. Instead, the results show that all models struggle to generalize, with MLE performing the worst due to assigning zero probability to all test sentences, while Laplace and Stupid Backoff offer some predictive capability but remain unreliable due to the limited number of test cases.

Implementation code

```
n_words = 10
seed_word = "<s>"

print("MLE Generated:", " ".join(lm_mle.generate(n_words, text_seed=[seed_word], random_seed=42)))
print("Laplace Generated:", " ".join(lm_lpc.generate(n_words, text_seed=[seed_word],
random_seed=42)))
print("StupidBackoff Generated:", " ".join(lm_sbo.generate(n_words, text_seed=[seed_word],
random_seed=42)))

def compute_average_perplexity(model, n, num_sentences= len(test_sentences)):
    total_perplexity = 0
    valid_sentences = 0
```

```
    random_sentences = random.sample(test_sentences, min(num_sentences, len(test_sentences)))

    for sentence in random_sentences:
        processed_sentence = sentence
        if not processed_sentence:
            continue
        padded_sentence = list(pad_both_ends(processed_sentence, n))
        test_ngrams = list(ngrams(padded_sentence, n))
        perplexity = model.perplexity(test_ngrams)

        if perplexity != float('inf') and perplexity > 0:
            total_perplexity += perplexity
            valid_sentences += 1

    avg_perplexity = total_perplexity / valid_sentences if valid_sentences > 0 else float('inf')
    return avg_perplexity

print(f"Average Perplexity (MLE): {compute_average_perplexity(lm_mle, n):.2f}")
print(f"Average Perplexity (Laplace): {compute_average_perplexity(lm_lpc, n):.2f}")
print(f"Average Perplexity (Stupid Backoff): {compute_average_perplexity(lm_sbo, n):.2f}")
```

Part 4: Noisy channel spelling corrector implementation

This noisy channel model is implemented to correct noisy (altered) sentences by selecting the most probable original sentence. This is done using Bayes' Rule:

$$P(W|X) = \frac{P(X|W)P(W)}{P(X)}$$

Since P(X) is constant for all candidate sentences we can simplify this to:

$$argmax_W P(X|W)P(W)$$

Here, P(W) represents the prior probability of a sentence W, computed using an n-gram language model (in our case n = 3), while P(X|W) is the likelihood of the noisy sentence X, given W. The objective is to find the best correction W that maximizes this probability.

The process begins with generating a dictionary of similar words for each vocabulary word using the Levenshtein distance in '*compute_close_words()'*. Then, for every noisy sentence, a set of possible corrections is created in *'candidates()'* by replacing each word individually with its closest alternatives. The probability of a sentence P(W) is calculated using an n-gram model in *'log_prior()'*, while the likelihood function P(X|W) is computed in *'log_likelihood()*', assuming a high probability (alpha=0.95) that words are already correct. Finally, the best sentence is selected in *'correct_sentence()'*, maximizing the sum of log probabilities (log(P(W)) + log(P(X|W))). If all candidates receive -inf scores, argmax() defaults to selecting the first candidate, which is the original noisy sentence. Since MLE assigns zero probability to unseen n-grams, it frequently produces -inf scores, leading to lower correction accuracy. The models are tested on noisy sentences, and correction accuracy is evaluated for each.

Final Accuracy Comparison:

MLE : 0.00%
LPC : 24.00%
SBO : 16.00%

Example corrections:

Noisy: go papa nobody thought of your walking
Corrected: no papa nobody thought of your walking
Original: no papa nobody thought of your walking

Noisy: ii could not walk half so far
Corrected: i could not walk half so far
Original: i could not walk half so far

Noisy: low often we shall be going to see them and they coming to see us we shall be always meeting
Corrected: how often we shall be going to see them and they coming to see us we shall be always meeting
Original: how often we shall be going to see them and they coming to see us we shall be always meeting
---

The low accuracy of MLE (0%) is due to its inability to handle unseen words, assigning them zero probability. Laplace (24%) improves on this by smoothing, allowing some corrections. Surprisingly, Stupid Backoff (16%) performs worse than Laplace, even though it typically handles unseen words more effectively by backing off to lower-order n-grams. This lower-than-expected accuracy raises questions about potential model limitations, dataset biases, or issues in our implementation or evaluation process.

Implementation code

```
#Exercise 4:

def levenshtein(s1: str, s2: str):
    return edit_distance(s1, s2, substitution_cost=1, transpositions=True)

def compute_close_words(vocab: List[str], max_dist: int = 1):
    vocab = np.array(vocab)
    word_lengths = np.array([len(w) for w in vocab])
    dict_lengths = {}
    for l in range(min(word_lengths), max(word_lengths)+1):
        dict_lengths[l] = vocab[word_lengths==l]
    min_length = min(dict_lengths.keys())
    max_length = max(dict_lengths.keys())
    close_words = {}
    for word in tqdm(vocab):
        length = len(word)
        candidate_words = []
        d1 = max(min_length, length - max_dist)
        d2 = min(max_length, length + max_dist)

        for d in range(d1, d2 + 1):
            candidate_words.extend(dict_lengths[d])
            close_words[word] = [w for w in candidate_words if levenshtein(word,w) <= max_dist]
    return close_words

def candidates(close_words: Dict[str, List[str]], vocab: List[str], sentence: List[str]):
    sent_x = sentence
    print(sent_x)
    for word_x in sent_x:
        if word_x not in vocab:
            continue
    candidates = [sent_x]
    for ii, word_x in enumerate(sent_x):
        for cand_w in close_words[word_x]:
            if cand_w != word_x:
                cand_sent = sent_x.copy()
                cand_sent[ii] = str(cand_w)
                candidates.append(cand_sent)
    return candidates

def log_prior(sentence_w: List[str], lm: LanguageModel):
    sentence_padded = ['<s>', '<s>',] + sentence_w + ['</s>']
    num_words = len(sentence_padded)
```

```
    log_prior_w = 0.0
    for i in range(2, num_words-1): # omit </s> because likelihoods don't have it
        # Uses trigrams, adapt it if you change the n
        score = lm.logscore(sentence_padded[i], [sentence_padded[i-2], sentence_padded[i-1]])
        log_prior_w += score
    return log_prior_w

def log_likelihood(sentence_x: List[str], sentence_w: List[str], close_words: Dict[str, List[str]],
alpha=0.95):

    log_likelihood_w = 0.0
    for x, w in zip(sentence_x, sentence_w):
        if x == w:
            log_likelihood_w += np.log(alpha)  #
        else:
            log_likelihood_w += np.log((1 - alpha) / (len(close_words[w]) - 1))
    return log_likelihood_w

def correct_sentence(sentence_x: str, lm: LanguageModel, close_words: Dict[str, List[str]], vocab:
List[str]):

    sentence_x = sentence_x.split()
    candidate_sentences = candidates(close_words, vocab, sentence_x)
    if not candidate_sentences:
        return " ".join(sentence_x)

    scores = []
    for candidate in candidate_sentences:
        log_p_w = log_prior(candidate, lm)
        log_p_x_given_w = log_likelihood(sentence_x, candidate, close_words)
        scores.append(log_p_w + log_p_x_given_w)
    best_index = np.argmax(scores)
    best_sentence = candidate_sentences[best_index]
    return " ".join(best_sentence)
close_words = compute_close_words(vocab, max_dist=1)

correct_count = 0
total_count = len(original_sentences)

models = {"MLE": lm_mle, "LPC": lm_lpc, "Stupid_backoff": lm_sbo}
accuracies = {}

for model_name, model in models.items():
    correct_count = 0

    print(f"Evaluating model: {model_name}")
    for i in range(total_count):
        noisy_sentence = altered_sentences[i]
        original_sentence = original_sentences[i]

        corrected_sentence = correct_sentence(noisy_sentence, model, close_words, vocab)

        print(f"Noisy: {noisy_sentence}")
        print(f"Corrected: {corrected_sentence}")
        print(f"Original: {original_sentence}")
        print("-" * 80)

        if corrected_sentence == original_sentence:
            correct_count += 1

    accuracy = (correct_count / total_count) * 100
    accuracies[model_name] = accuracy
    print(f"{model_name} Correction Accuracy: {accuracy:.2f}%\n")


print("Final Accuracy Comparison:")
for model_name, accuracy in accuracies.items():
    print(f"{model_name}: {accuracy:.2f}%")
```

Exercise 2: Neural based Language model

In the following exercise we implemented a neural based language model. The NNLM predicts words based on learned contextual patterns in fixed-size windows, while the spelling corrector relies on Levenshtein distance and n-gram probabilities to replace individual words without deeper contextual understanding. The NNLM uses word embeddings and neural layers, and this leads to it being able to learn semantic relationships and syntactic structures beyond just n-gram statistics, making it more context-aware.

The code first defines and trains a NNLM, where the NNLM class includes word embeddings, hidden layers, and output layers to predict words based on context. The '*FixedWindow'* dataset processes sentences into fixed size sliding windows, converting words into numerical indices for training using *'DataLoader'*. Finally, the trained NNLM is used in *'score_sentence()'* and *'correct_sentence()'* to evaluate and correct altered sentences, eventually selecting the highest-probability candidate by computing log-likelihood scores.

We can again look at the accuracy when we let our model correct the altered sentences that we have created before. **The accuracy of the model is 32%,** essentially saying that in 32% of cases, he was able to correctly modify the altered sentence to the correct one.

Implementation code:

```
#Part 1:

class NNLM(nn.Module):
    def __init__(self, num_classes, dim_input, dim_hidden, dim_embedding):
        super().__init__()
        self.embeddings = nn.Embedding(num_classes, dim_embedding)
        self.hidden1 = nn.Linear(dim_input * dim_embedding, dim_hidden, bias=False)
        self.ones = nn.Parameter(torch.ones(dim_hidden))
        self.hidden2 = nn.Linear(dim_hidden, num_classes, bias=False)
        self.hidden3 = nn.Linear(dim_input * dim_embedding, num_classes, bias=False)
        self.bias = nn.Parameter(torch.ones(num_classes))

    def forward(self, X):
        word_embeds = self.embeddings(X)
        X = word_embeds.view(-1, X.shape[1] * word_embeds.shape[2])
        tanh = torch.tanh(self.ones + self.hidden1(X))
        output = self.bias + self.hidden3(X) + self.hidden2(tanh)
        return output

class FixedWindow(Dataset):
    def __init__(self, sentences, vocab, window_size):
        self.vocab = vocab
        self.vocab = list(set(vocab))
        if "<unk>" not in self.vocab:
            self.vocab.append("<unk>")
        self.word2id = {word: idx for idx, word in enumerate(self.vocab)}
        self.id2word = {idx: word for word, idx in self.word2id.items()}
        self.window_size = window_size
        self.data = []

        for sentence in sentences:
            indexed_sentence = [self.word2id[word] for word in sentence if word in self.word2id]
            if len(indexed_sentence) < window_size:
                continue
            for i in range(len(indexed_sentence) - window_size + 1):
                context = indexed_sentence[i:i + window_size - 1]
                target = indexed_sentence[i + window_size - 1]
                self.data.append((context, target))

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        return torch.tensor(self.data[idx][0], dtype=torch.long), torch.tensor(self.data[idx][1],
dtype=torch.long)

window_size = 5
batch_size = 2048
```

```
embedding_dim = 32
hidden_dim = 64
learning_rate = 1e-3
epochs = 64

dataset = FixedWindow(train_sentences, vocab, window_size)
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)

num_classes = len(dataset.word2id)
dim_input = window_size - 1
model = NNLM(num_classes, dim_input, hidden_dim, embedding_dim)

loss_fn = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", device)
model.to(device)
model.train()
for epoch in range(epochs):
    total_loss = 0
    for batch_idx, (X, y) in enumerate(tqdm(dataloader, desc=f"Epoch {epoch+1}")):
        X, y = X.to(device), y.to(device)
        optimizer.zero_grad()
        output = model(X)
        loss = loss_fn(output, y)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

    print(f"Epoch {epoch+1}/{epochs}, Loss: {total_loss / len(dataloader):.4f}")

torch.save(model.state_dict(), "NNLM_trained.pth")
print("Model training completed and saved as 'NNLM_trained.pth'")

#Part 2:

def score_sentence(sentence, model, word2id, window_size):
    model.eval()
    padded_sentence = ['<s>'] * (window_size - 1) + sentence + ['</s>']

    if "<unk>" not in word2id:
        word2id["<unk>"] = len(word2id)

    sentence_ids = [word2id.get(word, word2id['<unk>']) for word in padded_sentence]

    score = 0.0
    with torch.no_grad():
        for i in range(len(sentence) + 1):
            if i + window_size <= len(sentence_ids):
                context = torch.tensor(sentence_ids[i:i + window_size - 1],
dtype=torch.long).unsqueeze(0).to(device)
                target = sentence_ids[i + window_size - 1]

                output = model(context)
                log_probs = torch.log_softmax(output, dim=1)
                score += log_probs[0, target].item()

    return score


def correct_sentence(sentence, model, dataset, close_words, window_size):

    word2id = dataset.word2id
    candidates_list = candidates(close_words, dataset.vocab, sentence.split())

    best_sentence = sentence
    best_score = -np.inf

    for candidate in candidates_list:
        candidate_score = score_sentence(candidate, model, word2id, window_size)
        if candidate_score > best_score:
            best_score = candidate_score
```

```
            best_sentence = " ".join(candidate)

    return best_sentence

model.load_state_dict(torch.load("NNLM_trained.pth"))
model.to(device)

correct_count = 0
total_count = len(altered_sentences)

for i in range(total_count):
    noisy_sentence = altered_sentences[i]
    original_sentence = original_sentences[i]
    corrected_sentence = correct_sentence(noisy_sentence, model, dataset, close_words, window_size)
    print(f"Noisy: {noisy_sentence}")
    print(f"Corrected: {corrected_sentence}")
    print(f"Original: {original_sentence}")
    print("-" * 80)
    if corrected_sentence == original_sentence:
        correct_count += 1

accuracy = (correct_count / total_count) * 100
print(f"NNLM Correction Accuracy: {accuracy:.2f}%")
```

Example output

Noisy: here sister though comparatively but little removed by matrimony being settled in london only sixteen miles off was much beyond here daily reach and many a long october and november evening must be struggled through at hartfield before christmas brought the next visit from isabella and here husband and their little children to fill the house and give here pleasant society again

*Corrected: **her** sister though comparatively but little removed by matrimony being settled in london only sixteen miles off was much beyond here daily reach and many a long october and november evening must be struggled through at hartfield before christmas brought the next visit from isabella and here husband and their little children to fill the house and give here pleasant society again*

Original: her sister though comparatively but little removed by matrimony being settled in london only sixteen miles off was much beyond her daily reach and many a long october and november evening must be struggled through at hartfield before christmas brought the next visit from isabella and her husband and their little children to fill the house and give her pleasant society again

---

Noisy: we must go in the carriage two be sure

*Corrected: we must go in the carriage **to** be sure*

Original: we must go in the carriage to be sure

Exercise 3: Autoregressive neural network-based model.

We can compute the perplexity of a neural network-based autoregressive model on the given sentence as follows:

First, we tokenize the sentence into words:

$$W = (w_1 = ' Joseph', w_2 = ' was', \dots, w_{15} = ' sinewy')$$

Next, we compute the conditional probabilities using the model, which predicts each word based on the entire sequence of preceding words:

$$P(W) = \prod_{t=1}^{N} P(w_t | w_1, w_2, \dots, w_{t-1})$$

Finally, we apply the perplexity formula to evaluate the sentence. The key difference from an n-gram model, and therefore in how perplexity is calculated, is that n-gram models rely only on the last n – 1 words. They require smoothing to handle unseen words but might still fail to assign probabilities to certain sequences. In contrast, an autoregressive neural network model learns richer representations that generalize better, reducing the need for smoothing. However, it is more computationally expensive and requires larger datasets for better training.

While both models use the same perplexity formula, the key difference thus lies in how context is utilized. N-gram models have a limited context window, whereas autoregressive models consider the entire sequence. This difference affects probability estimation and, consequently, the perplexity calculation.