Pytorch и обнаружение дубликатов на StackOverflow.

Хрыльченко Кирилл

Математические методы анализа текстов 2021

22 сентября, 2021

Stack Overflow[1]

How do I exit the Vim editor?

Asked 9 years, 1 month ago Active 22 days ago Viewed 2.4m times

4411

I'm stuck and cannot escape. It says:

```
"type :quit<Enter> to quit VIM"
```

But when I type that it simply appears in the object body.

1020

vim vi

Share Follow

edited May 14 '19 at 22:49
Peter Mortensen
28.8k ●21 ●95 ●123

asked Aug 6 '12 at 12:25
jclancy
42.1k ●5 ●25 ●32

173 Are you just trying to quit VIM ? If this is the case, press "escape" and then type ':q' – Pop Aug 6 '12 at 12:28

58 Don't forget the **colon**! You should type `:quit` and then hit the `[ENTER]` key. – Farahmand Mar 4 '14 at 18:33

127 It's really easy to learn the basics of vim, and it's built right into your system. In terminal type "vimtutor". 25 minutes later you will be going faster than your usual text editor! – Mark Robson Jan 26 '15 at 12:11

9 Check here more commands. – Toni Aug 7 '15 at 15:36

104 To prevent `git commit` sending you to vim in the future: `git config --global core.editor="nano"` – Tom Kelly May 24 '17 at 3:19

Show 7 more comments

11 Answers

Active | Oldest | Votes

5544

Hit the Esc key to enter "Normal mode". Then you can type `:` to enter "Command-line mode". A colon (`:`) will appear at the bottom of the screen and you can type in one of the following commands. To execute a command, press the Enter key.

- `:q` to quit (short for `:quit`)
- `:q!` to quit without saving (short for `:quit!`)

[1] https://stackoverflow.com

Stack Overflow. Duplicates Detection

Stack overflow in a nutshell

Figure: Домашнее задание — определение дублируемых вопросов на stack overflow.

Данные

question	How to print a binary heap tree without recursion?
duplicate	How do you best convert a recursive function to an iterative one?
random 1	How can i use ng-model with directive in angular js?
random 2	flash: drawing and erasing
random 3	toggle react component using hide show classname

Table: Один сэмпл из валидации

question 1	question 2
converting string to list	Convert Google results object (pure js) to Python object
Which HTML 5 Canvas Javascript to use for making an interactive drawing tool?	Event handling for geometries in Three.js?

Table: Сэмплы из обучения

Подготовка данных

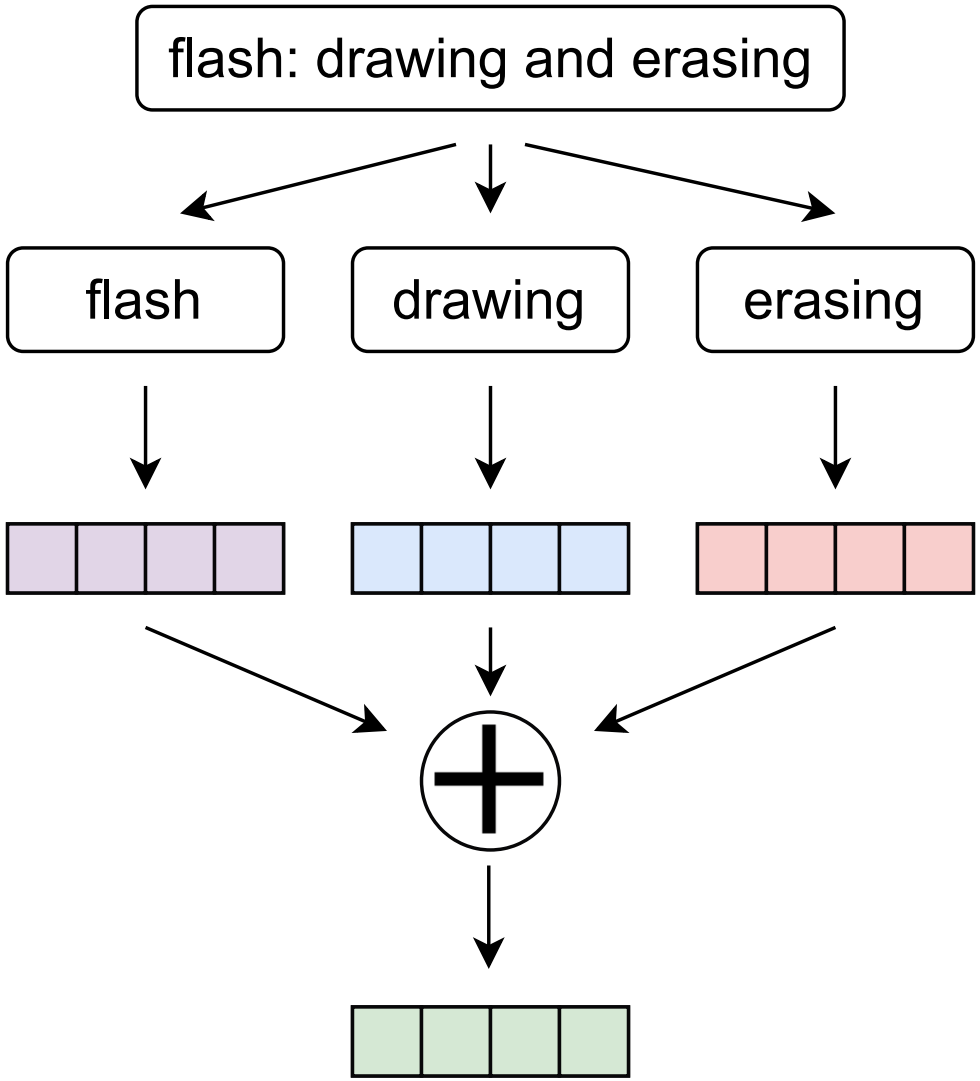
Что надо сделать:

- Приведение к нижнему регистру
- Удаление:
  - пунктуации, всевозможных галочек и стрелочек с помощью библиотеки **re** — можно попробовать оставить только латиницу
  - стопслов с помощью библиотеки **nltk**,
  - коротких слов
- **tip:** посмотрите глазами на слова, которые не нашлись в word2vec

Что еще иногда (не)делают:

- исправляют опечатки
- не приводят к нижнему регистру, т.к. он полезен — e.g. определение токсичности комментариев. Cased vs uncased

Мешок[2] векторов



[2] порядок в такой модели не учитывается

Мешок векторов

Представляем каждый вопрос как вектор, усредняя векторы всех слов.
Что нужно для модели:

- векторы слов, маппинг слово -> вектор
- определиться, как будем определять схожесть двух векторов

Схожесть двух векторов — скаляр:

- евклидово расстояние $\|u - v\|_{l2}$. Является метрикой
- скалярное произведение $\langle u, v \rangle$. Вычислительно дешевле, чем косинус
- косинус $\frac{\langle u,v \rangle}{\|u\|\|b\|}$. Меньше аффектится частотностью тех или иных слов в тексте

Определение похожих вопросов

Нашу задачу можно сформулировать двумя способами:

- классификация — является ли данная пара вопросов дубликатами?
- ранжирование — глядя на вопрос, отранжировать список других вопросов в порядке схожести

Метрик ранжирования много — ROC AUC, DCG, nDCG, MRR, MAP и т.д. Возьмем очень простую — HR (Hits Ratio).

Hits@k

Для сэмпла: Hits@k = предикат [правильный ответ оказался в top k]. Для выборки — это доля сэмплов, для которых правильный ответ оказался в top k.

Как оптимально посчитать hits@k сразу для нескольких k — посчитать количество сложных негативов (hard negatives), тогда hits@k = [amount of hard negatives $<$ k].

Неизвестные слова[3]

Очевидная точка роста для модели — для бОльшего количества слов находить векторы (иначе слово просто не учитывается в итоговом векторе). Выучим трехбуквенную модель для слов, аппроксимируя векторы известных слов как сумму векторов его буквенных триграм:

$$\mathrm{d}(\mathrm{vector}(w), \frac{1}{n}\sum_{c \in w}\mathrm{vector}(c)) \to \min,$$

где w — слово, а $\{c : c \in W\}$ — множество последовательностей из трех букв внутри этого слова.

Чар-триграммы для слова 'арбуз'

#ар, арб, рбу, буз, уз#, префикс и суффикс выделяются в отдельные триграммы

[3] Часто неизвестные слова обозначают как OOV (out of vocabulary) или UNK (unknown)

Результаты

После обучения триграммной модели, пробежимся по всем неизвестным словам и составим для них векторы.
В рамках работы нужно ответить на вопросы с обоснованием:

- повышает ли качество «чистка» текста
- повышает ли качество триграммная модель для неизвестных слов
- повышает ли качество использование префикса и суффикса
- что лучше работает для триграммной модели как функция d: евклидово расстояние, косинус, скалярное произведение
- что лучше работает для модели схожести вопросов: евклидово расстояние, косинус, скалярное произведение

О списках

Питоновский **list** — динамический массив, НЕ N-связный список.

- элементы списка находятся в одной области памяти, последовательно
- $O(1)$ операции: индексация, добавление и удаление из конца списка
- проверка на вхождение: *a in mylist* — $O(n)$!
- *mylist[i:j]* — копирует[4] список, т.е. $O(n)$ в худшем случае
- *mylist1 + mylist2* — копирует оба списка в новый, т.е. $O(m+n)$
- динамический массив — для списка выделена память с запасом (*table doubling*), выделенная память идет по степеням двойки
- **nltk.stopwords** — список
- список + [элемент] — копирует
- первый список += второй список — добавляет (копирует) элементы второго списка в конец первого списка

[4] копирование - не глубокое, т.е. shallow copy

Базовая работа с текстом

Модель работает с тензорами. Слова нужно пронумеровать, глядя на имеющуюся коллекцию текста. Как создать словарь:

- Считаем частоты слов с помощью **Counter**[5], убираем редкие слова
- сохраняем список со словами **vocab** — почему не нужен словарь **id2vocab**?
- создаем словарь **vocab2id**, в котором по слову можно найти его индекс (номер в списке)

Про словари:

- добавление, удаление, проверка на вхождение за $O(1)$

Переводим предложение в индексы как:

- *[vocab2id.get(word, UNK) for word in doc if word in vocab2id]*

[5] двойные циклы — *Counter(word for doc in corpus for word in doc)*

Подготовка батча

Трансформация батча[6]:

- Был список документов: ["А и Б сидели на трубе", "А в каком случае P = NP?", . . .]
- Теперь есть список списков индексов слов: [[1 5 2 3 19 4], [1 8 6 100 . . .], [. . .], . . .]

Модели нужен тензор — все списки должны быть одной длины. Для этого используем:

- **паддинг** — дополняем справа все объекты **индексом паддинга** до фиксированной длины **max length**.
- **nn.EmbeddingBag** — склеиваем все индексы в один большой список, запоминая **сдвиги**[7] в другом списке

[6] набор входных данных для модели, передаваемый "одним пакетом"

[7] offsets

Bucket sequencing

Подобрать общую длину для всех сэмплов можно по гистограмме:

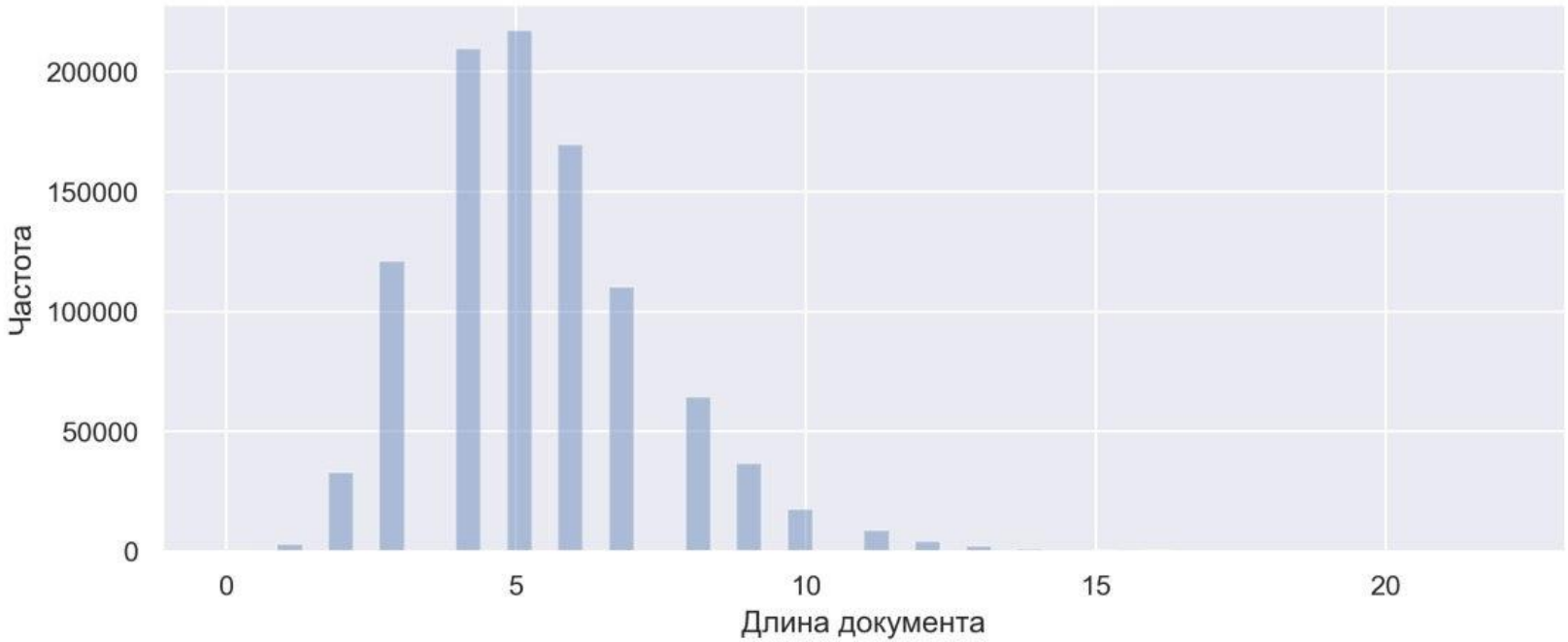


Figure: Гистограмма распределения длин документов

Но существуют и более оптимальные практики.

Bucket sequencing

Будем для каждого батча подбирать свою длину. Одинаковые длины для разных батчей нам не нужны. Один батч — один тензор, который пойдет в модель.

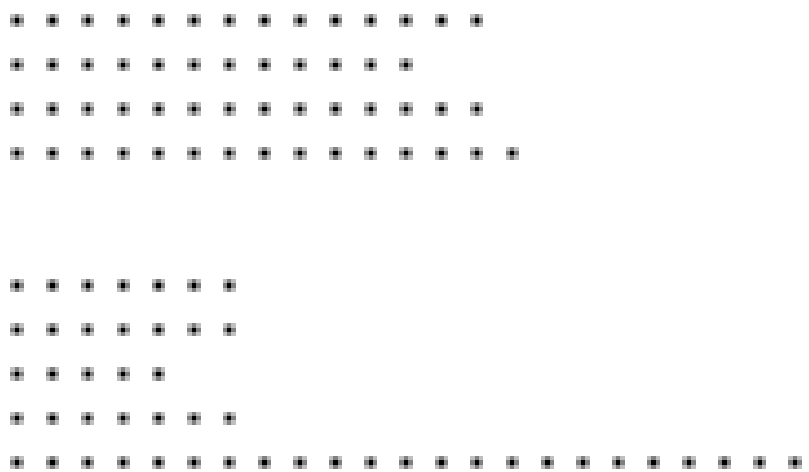


Figure: Два батча.

Как выбрать длину для батча:

- как максимальную длину среди всех сэмплов в батче
- обрезать по квантилю

Для наиболее оптимальных длин батчей на инференсе сортируйте данные.

Преобразование триграмм

```
class TrigramTokenizer:

    def __init__(self, words):
        # формируем множество всевозможных триграмм, встречающихся в словах из words
        # делаем маппинг триграм в индексы
        pass

    @property
    def vocab_size(self):
        # возвращаем количество триграмм, которые мы положили в наш маппинг
        pass

    @staticmethod
    def _get_trigrams(word):
        # возвращаем список триграмм для слова word
        pass

    def __call__(self, word):
        # возвращаем список индексов триграмм для слова word
        pass
```

Для **get trigrams** удобно использовать генератор.

Dataset

```python
from torch.utils.data import Dataset

class TrainTrigramDataset(Dataset):

    def __init__(self, vocab, w2v_embeddings, tri_tokenizer):
        # формируем выборку для обучения триграммной модели
        # ЗАРАНЕЕ считаем маппинг в список индексов для всех известных слов
        pass

    def __len__(self):
        # возвращает размер датасета - количество объектов в выборке, т.е. известных слов
        pass

    def __getitem__(self, idx):
        # возвращает список индексов триграмм для idx-го слова в выборке, а также w2v вектор этого слова
        pass
```

Dataset с точки зрения питона — массив. Можем узнать его длину $len(ds)$, а также проиндексировать (вытащить пронумерованный элемент с помощью $ds[i]$)

DataLoader

DataLoader помогает сформировать **батч** из выборки:

$$[dataset[i] \text{ for } i \text{ in } ids],$$

где *ids* — номера документов в датасете. Фактически, это итератор по батчам.

Он устроен чуть сложнее:

- Умеет работать параллельно — *num workers*. Полезно при тяжелом **getitem**,
- Может каждую эпоху перемешивать выборку[8] — *shuffle=True*[9],
- Может игнорировать последний батч, если он меньше, чем предыдущие

[8] для более интересных махинаций смотрите **Sampler**'ы

[9] не забывайте выключать эту опцию на валидации

Dataset + DataLoader = Success

Фундамент любого пайплайна:

- Создаем датасет, из которого можно быстро доставать нужные нам объекты
- Создаем даталоадер, который эти объекты достаёт
- Ловим объекты у даталоадера

```
dataset = MyDataset(my_data)
dataloader = DataLoader(dataset, batch_size=64, shuffle=True)

for obj, target in dataloader:
    # итерируясь по даталоадеру, получаем сразу подготовленные батчи
    # делаем шаг градиентного спуска, подсчитав лосс для батча
    pass
```

Важно: цикл проходит каждый объект единожды. Это одна **эпоха**.

collate fn

Внутри даталоадера собирается список объектов:
[dataset[i] for i in ids], затем передается специальной функции *collate fn*.

```
def collate_fn(batch):
    # принимает на вход список элементов из датасета [ds[idx] for idx in [1, 5, 2, 3, 4]]
    words, trigrams = zip(*batch)
    words = torch.tensor(words, dtype=torch.float)

    offsets = ...
    trigrams = ...

    # итерируясь по даталоадеру, мы получаем выход именно этой функции в цикле как objb
    return words, trigrams, offsets
```

В нашем случае (для триграмм):

- words — эмбеддинги из w2v
- trigrams — списки индексов и сдвигов для триграмм

Чтобы использовать **nn.EmbeddingBag**, нужно склеить списки индексов в один и запомнить сдвиги. Для bucket sequencing (если решили паддить) можно использовать **torch.nn.utils.rnn.pad sequence**

Модель

Модель — та самая нейронная сеть.

- принимает на вход: тензор из всех объектов в батче
- выдает: предсказания для всех объектов в батче

```
from torch import nn

class TrigramModel(nn.Module):

    def __init__(self, num_embeddings, embedding_dim):
        super().__init__()
        self._embeddings = nn.EmbeddingBag(
            num_embeddings=num_embeddings,
            embedding_dim=embedding_dim
        )

    def forward(self, trigrams, offsets):
        # принимаем на вход индексы и сдвиги, выдаем сформированные векторы слов
        pass
```

Для триграмм — принимает два параметра. Тензор индексов триграмм и тензор сдвигов.

Оптимизация

Функционалы ошибки:

- для триграммной модели — nn.MSELoss, nn.CosineSimilarty
- для классификации — nn.BCELoss[10] и nn.BCEWithLogits[11]. Второй вариант постабильней
- кастомный функционал ошибки — слой, на выходе из которого скаляр

Оптимизатор — torch.optim.Adam. При создании, передаем:

- параметры модели
- шаг обучения — learning rate
- weight decay — l_2 регуляризация

[10] если есть сигмоида в конце модели

[11] сигмоида входит в лосс, на выходе модели голые логиты

Пайплайн

Датасет и даталоадер, модель, функционал, оптимизатор — всё, что нужно для простого пайплайна.

```
criterion = nn.MSELoss()
model = TrigramModel()
optimizer = torch.optim.Adam(model.parameters())

model.train()  # переводим модель в режим обучения (нужно для всяких сложных слоёв, типа dropout и нормализаций)

for word_embeddings, trigrams, offsets in dataloader:
    tri_embeddings = model(trigrams, offsets)  # скармливаем батч модели
    loss = ...  # считаем лосс между триграммным вектором и исходным вектором из word2vec

    loss.backward()  # считаем градиенты
    optimizer.step()  # делаем шаг спуска - обновляем параметры
    optimizer.zero_grad()  # обнуляем посчитанные градиенты
```

Регуляризация

Модель быстро переобучается. Что делать:

- Dropout между слоями
- SpatialDropoput — зануляем измерения в исходных эмбеддингах слов или триграмм
- ограничиваем max norm в слое эмбеддингов
- batch, instance, layer нормализация (последняя популярней всего в текстах)
- l_2 регуляризация определенных (или всех) слоёв сети
- gradient clipping[12] — **torch.nn.utils.clip grad norm**
- фриз слоёв — например, предобученных эмбеддингов
- большой batch size
- маленький learning rate
- gradual unfreezing
- энтропия как функционал

[12] против взрыва градиентов

model.train vs model.eval:

- model.train — работает дропаут и считаются статистики по батч нормализации
- model.eval — отключается дропаут (детерминированный выход в сети), статистики по батч норме фиксируются как скользящее среднее статистик из батчей

Для BCEWithLogits и BCELoss (бинарной кросс-энтропии) — таргеты должны быть torch.float! Можно подавать вероятности (приближать другое распределение), смягчать метки, чтобы делать модель более "неуверенной".

Работа с GPU:

- всё либо на GPU, либо на CPU
- model.cuda(), затем для каждого батча obj.cuda(), target.cuda()
- лучше использовать переменную **device** и делать model.to(device) и obj.to(device)

- Для ускорения можно использовать mixed precision — например, библиотеку **apex**.
- broadcasting: **torch.einsum**[13] позволяет делать сложные операции с тензорами чуть быстрее
- Рандом в Python **очень медленный**. Используйте библиотеку **fastrand**, генерируйте случайные числа наперёд большими массивами, используйте внутренние генераторы случайных чисел pytorch'a
- Используйте **tensorboard**, чтобы отслеживать обучение нейронной сети
- Когда не обучаете модель (на валидации, инференсе) — используйте контекстный менеджер **with torch.no grad**!
- не повторяйте одни и те же операции — например, векторизацию текста

[13] аналогично np.einsum

Обучение полноценной модели определения дубликатов вопросов

Используем предобученные векторные представления, а хочется обучить векторы под нашу задачу. Это сильно бустит качество!

Metric learning

Выучивание «схожести» объектов — metric learning. Отображаем объекты в одно «семантическое пространство» (превращаем объекты в векторы) и выучиваем «метрику схожести» — например, косинус между векторами.

Распространенные техники: contrastive loss, triplet loss, NT-Xent[14]. Почти всегда не имеем явных негативов, приходится их «майнить».

[14] Improved Deep Metric Learning with Multi-class N-pair Loss Objective, Kihyuk Sohn