

# Stepik. Neural networks and NLP. 2. Vector model and classification of long texts

---

## 2.1 Векторная модель текста и TF-IDF

Всем привет! Давайте немного поговорим о классике — о [разреженных векторных моделях](#) и о том, как их готовить. Такие модели ещё называют "[методом мешка слов](#)". Они хорошо работают, когда класс документа соответствует его тематике. Как правило, тематика документа хорошо описывается составом словаря, который используется в этом документе, а также частотами слов, а не тем, как именно они употребляются в документе, не структурой фраз. Тогда каждый документ описывается длинным вектором размерности порядка десятков или сотен тысяч элементов. Большая часть — нули. Для каждого слова в документе мы имеем какое-то вещественное число. Ранее мы уже выяснили, что модели должны работать лучше, когда веса слов отличаются — чем значимее слово, тем больше его вес. Однако, как именно считать эти веса? Давайте рассмотрим несколько вариантов и разберём их преимущества и недостатки. Простейший вариант — взвешивать слова по количеству их употреблений в документе.<sup>[1,2]</sup> Элементарно — здесь мы видим несколько наиболее частотных слов из статьи Википедии про машинное обучение. Веса слов — это просто целые числа. Естественно, данный подход имеет недостатки. Во-первых, вес слова зависит от длины документа. В длинных документах слова имеют больший вес, как будто бы они более значимы, но это не так. Во-вторых, самые частотные слова — это союзы, предлоги, местоимения... Они встречаются везде, но абсолютно неинформативны и редко бывают полезны для каких-либо задач классификации. Вот — мы видим три предлога и союза среди наиболее часто употребимых слов в статье про "машинальное обучение". Давайте будем бороться с этими проблемами по порядку. Вначале отнормируем вектор документа на его длину. На слайде изображена формула для нормировки по [L2-норме](#) (или по евклидовой норме). Тогда веса слов будут зависеть от длины документа гораздо слабее, но они всё равно будут зависеть, так как с увеличением длины документа расширяется используемый словарный запас. Однако, по-прежнему, предлоги и союзы — это самые значимые слова. Нас это не совсем устраивает.

[1] Количество уникальных слов в документе - закон Ципфа

[https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%BA%D0%BE%D0%BD\\_%D0%A5%D0%B8%D0%BF%D1%81%D0%B0](https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%BA%D0%BE%D0%BD_%D0%A5%D0%B8%D0%BF%D1%81%D0%B0)

[2] Векторизация текстов через подсчёт количества словоупотреблений [https://scikit-learn.org/stable/modules/generated/sklearn.feature\\_extraction.text.CountVectorizer.html](https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html)

Список литературы:

Разреженные векторные модели (методы мешка слов):

- Christopher D. Manning, Prabhakar Raghavan and Hinrich Schütze, Introduction to Information Retrieval, Cambridge University Press. 2008. <https://nlp.stanford.edu/IR-book/>
- Количество уникальных слов в документе - закон Хипса [https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%BA%D0%BE%D0%BD\\_%D0%A5%D0%B8%D0%BF%D1%81%D0%B0](https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%BA%D0%BE%D0%BD_%D0%A5%D0%B8%D0%BF%D1%81%D0%B0)
- Векторизация текстов через подсчёт количества словоупотреблений [https://scikit-learn.org/stable/modules/generated/sklearn.feature\\_extraction.text.CountVectorizer.html](https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html)

## Мешок слов

Тематика хорошо описывается именно составом лексикона и частотой встречаемости слов в документе, а не их порядком.

"Днём мама мыла раму. Вечером мы пошли гулять."

и	мы	мама	велосипед	...	рама
0	0.01	0.1	0	...	2



### Как именно считать вес слова?

Слово	Вес
обучение	24
с	20
в	18
и	17
...	...
данных	11
машииного	11
или	8
к	8
как	8
при	8
задачи	7
на	7
качества	7
учителя	7
решение	6
методы	6
из	6
не	5
...	...
прецедентам	5
обучении	5

Вес = количество употреблений слова в документе

- Вес слова зависит от длины текста
- Предлоги и союзы - самые значимые слова



Вес = количество употреблений слова в документе, делённое на длину документа

$$nw_i = \frac{w_i}{\sqrt{\sum_j w_j^2}}$$



Вес = количество употреблений слова в документе, делённое на длину документа

$$nw_i = \frac{w_i}{\sqrt{\sum_j w_j^2}}$$

- Предлоги и союзы - самые значимые слова

Слово	Вес
обучение	0.086
с	0.071
в	0.064
и	0.061
...	...
данных	0.039
машинного	0.039
или	0.029
к	0.029
как	0.029
при	0.029
задачи	0.025
на	0.025
качества	0.025
учителя	0.025
решение	0.021
методы	0.021
из	0.021
не	0.018
...	...
прецедентам	0.018
обучении	0.018



Самое время рассказать про один из фундаментальных эмпирических законов лингвистики (и не только лингвистики, на самом деле) — [закон Ципфа](#). Возьмём большую [коллекцию документов](#), посчитаем для каждого слова в этой коллекции частоту его встречаемости, то есть количество документов, в которых это слово используется, потом отсортируем полученный список по убыванию частот и получим примерно следующий график. По оси абсцисс отложим

ранг слова, то есть его порядковый номер в отсортированном списке. По оси ординат отложим частоту слова — относительную частоту. Это график [распределения Ципфа](#) — распределения вероятностей, описывающего взаимоотношения частоты события и количества событий с такой частотой. Оно относится к классу степенных распределений и задаётся следующей функцией. Случайная величина, поведение которой мы описываем — это ранг, то есть порядковый номер слова. У этого распределения два параметра: это "s" — определяет скорость убывания (чем больше "s", тем реже редкие слова), и "n" — это количество слов в нашем словаре; "z" — это просто [нормализационная константа](#), чтобы распределение стало распределением.

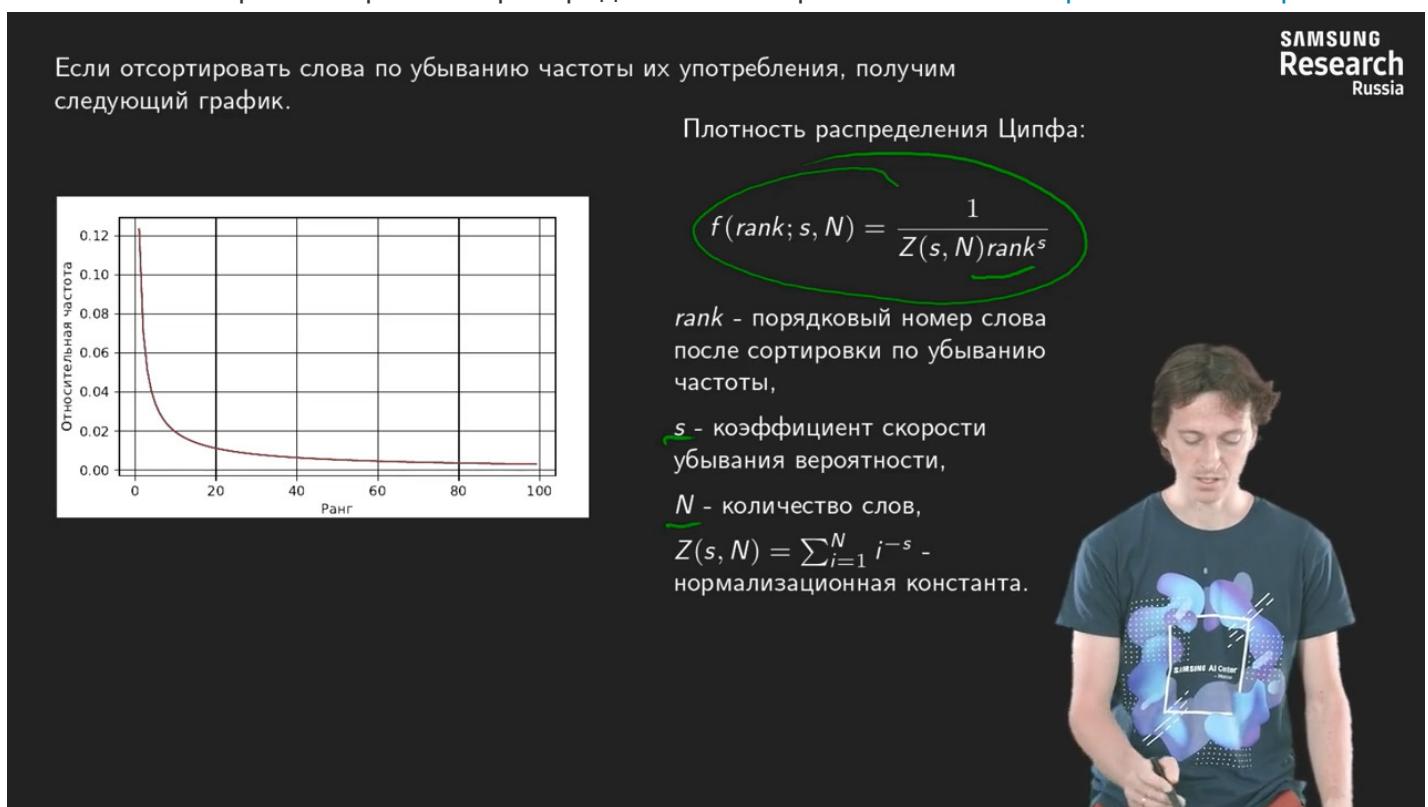
Список литературы:

Разреженные векторные модели (методы мешка слов):

- Christopher D. Manning, Prabhakar Raghavan and Hinrich Schütze, Introduction to Information Retrieval, Cambridge University Press. 2008. <https://nlp.stanford.edu/IR-book/>
- Закон Ципфа [https://en.wikipedia.org/wiki/Zipf%27s\\_law](https://en.wikipedia.org/wiki/Zipf%27s_law)

Из комментариев:

Можно поэкспериментировать с распределением на разных текстах <https://seolik.ru/zipfa>



Из этого всего можно сделать два практических вывода: во-первых, частотных слов очень мало. Они слабо информативны, так как встречаются практически во всех документах. А вот редких слов очень много — если мы какое-нибудь редкое слово встречаем в документе, то мы с большой уверенностью можем сказать, к какой тематике он относится. Но проблема в том,

что такие слова очень редки, и поэтому они ненадёжны в качестве факторов при принятии решений. Следовательно, нам нужно придерживаться баланса частотности и информативности. Основная идея в том, что чем чаще слово встречается в документе, тем более оно характерно для этого документа, тем лучше описывает его тематику. С другой стороны, чем это слово реже встречается в [корпусе](#), в выборке документов, тем оно более специфично и информативно. За этот баланс отвечают две величины: TF и IDF. TF (term frequency) — это частота слова в документе. Тут всё понятно, мы берём количество употреблений слова в документе и делим на длину документа — никаких чудес. IDF (inverse document frequency) — обратная частота слова в документах. Тут мы размер коллекции делим на количество документов, в которых слово употребляется. Таким образом, наибольший вес будет иметь слово, встречающееся только в одном документе. Тогда итоговый вес слова можно посчитать, как произведение этих двух величин. На практике, TF часто логарифмируют (следующим образом). Это позволяет сделать распределение весов слов менее контрастным и уменьшить его дисперсию.

Из комментариев:

Вопрос:

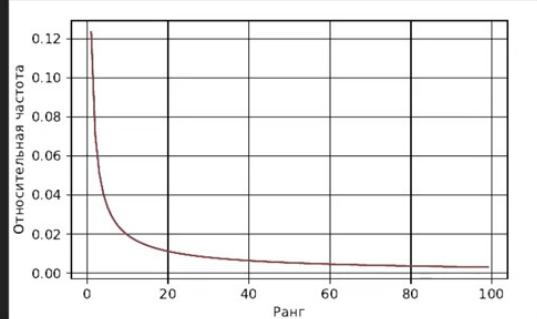
Очень странно видеть здесь логарифмирование поверх TF. Гораздо типичнее, когда логарифмируют IDF. Одна из статей на эту тему: <https://plumbr.io/blog/programming/applying-tf-idf-algorithm-in-practice>.

Ответ (Романа Суворова):

логарифмирование поверх TF полезно, когда в Вашем датасете документы сильно отличаются по длине. В целом логарифмировать IDF тоже можно.

Если отсортировать слова по убыванию частоты их употребления, получим следующий график.

Плотность распределения Ципфа:



$$f(rank; s, N) = \frac{1}{Z(s, N) rank^s}$$

$rank$  - порядковый номер слова после сортировки по убыванию частоты,

$s$  - коэффициент скорости убывания вероятности,

$N$  - количество слов,

$Z(s, N) = \sum_{i=1}^N i^{-s}$  - нормализационная константа.



- Частотных слов мало и они неинформативны
- Редких слов много, они информативны, но на них сложно опираться
- Баланс частотности и информативности

### Баланс частотности и информативности

- Чаще встречается в документе - более характерен для этого документа
- Реже встречается в корпусе - более информативен

TF - term frequency - значимость слова в рамках документа

$$TF(w, d) = \frac{\text{WordCount}(w, d)}{\text{Length}(d)}$$

где  $\text{WordCount}(w, d)$  - количество употреблений слова  $w$  в документе  $d$ ,

$\text{Length}(d)$  - длина документа  $d$  в словах.



TF - term frequency - значимость слова в рамках документа

$$TF(w, d) = \frac{WordCount(w, d)}{Length(d)}$$

где  $WordCount(w, d)$  - количество употреблений слова  $w$  в документе  $d$ ,  
 $Length(d)$  - длина документа  $d$  в словах.

IDF - inverse document frequency - специфичность слова

$$IDF(w, c) = \frac{Size(c)}{DocCount(w, c)}$$

где  $DocCount(w, c)$  - количество документов в коллекции  $c$ ,  
в которых встречается слово  $w$ ,

$Size(c)$  - размер коллекции  $c$  в документах.

$$\underline{TFIDF(w, d, c) = TF(w, d) \cdot IDF(w, c)}$$

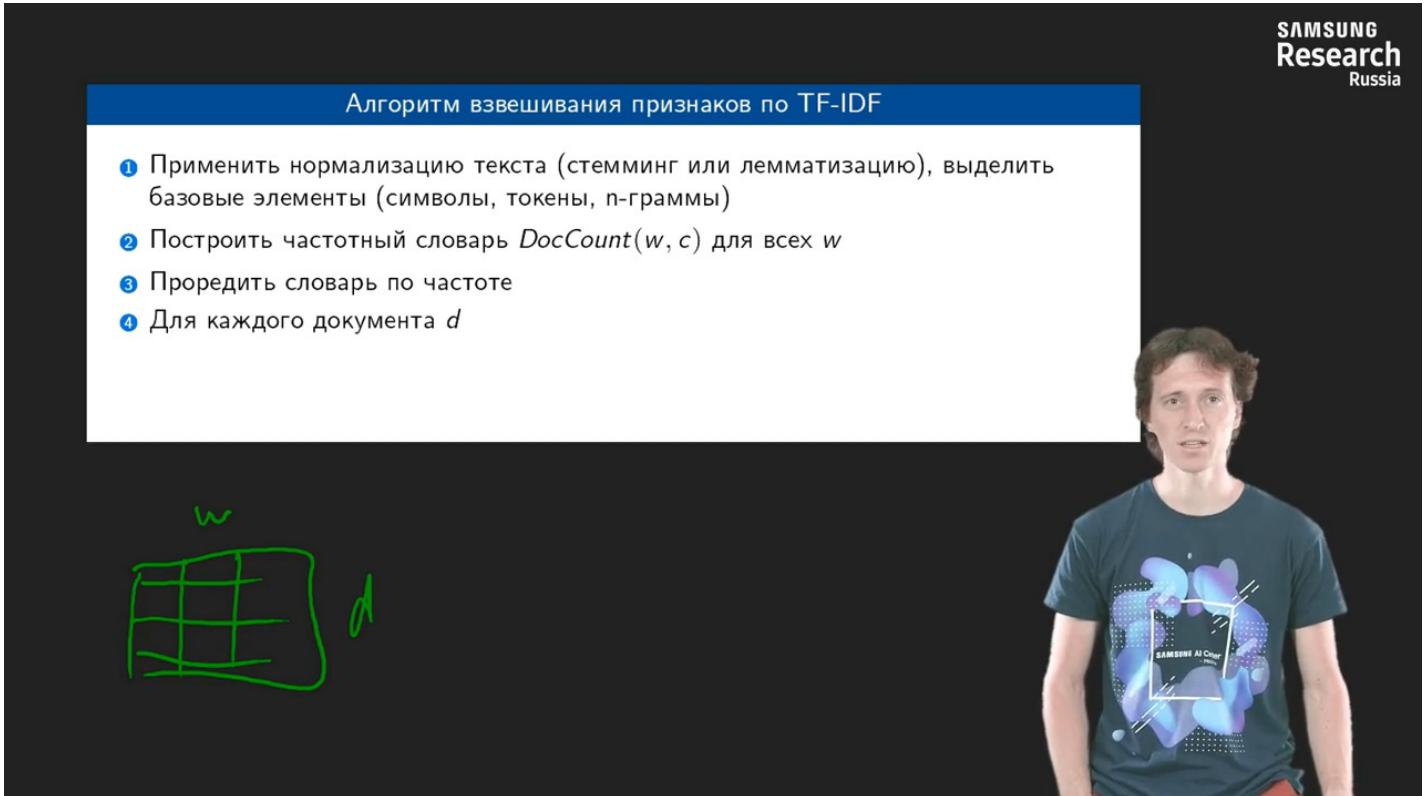


Давайте посмотрим, как применяется такой способ взвешивания признаков на практике. Во-первых, тексты токенизируются и нормализуются. Если мы решаем работать с N-граммами, то вместо токенизации или после неё выделяем N-граммы. Во-вторых, мы проходимся по всей коллекции документов и, для каждого слова, подсчитываем, в каком количестве документов оно встретилось. Если у нас большая коллекция, то словарь может получиться просто гигантским, особенно если мы работаем с N-граммами, поэтому периодически, во время построения, или после этой процедуры, мы должны выбросить из словаря всё, что считаем неинформативным — слишком редкие и слишком частые слова. Затем мы начинаем строить матрицу признаков. Каждая строчка этой матрицы соответствует документу, а каждый столбец — статистике встречаемости этого слова в документе. Таким образом, для каждого документа мы считаем частоты слов в нём и записываем в соответствующие ячейки таблички веса слов по указанной формуле. Затем, переходим к следующему документу (всё просто). Таким образом, TF-IDF — это способ взвешивания и отбора категориальных признаков [1] в задачах машинного обучения — не только в классификации и не только для текстов. Надо заметить, что TF-IDF никак не использует информацию о метке объекта — это одновременно и преимущество, и недостаток. Преимущество заключается в том, что мы можем использовать TF-IDF, не имея меток, то есть задачах обучения без учителя. Недостаток — в том, что мы теряем информацию или недостаточно эффективно её используем.

[1] Ещё несколько способов взвешивания и отбора признаков [https://scikit-learn.org/stable/modules/feature\\_selection.html](https://scikit-learn.org/stable/modules/feature_selection.html)

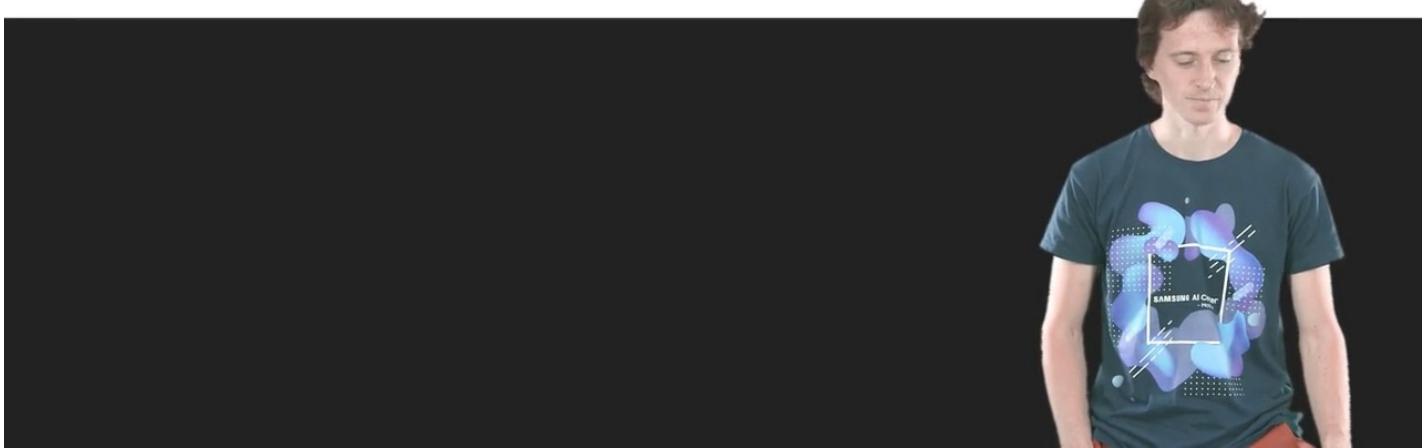
Дополнительные комментарии к видео (от Романа Суворова):

- Начиная с 1:47 в видео говорится о том, что если информация о метках есть, TF-IDF сам по себе не позволяет её использовать (то есть мы как бы теряем эту информацию).



базовые элементы (символы, токены, n-граммы)

- ② Построить частотный словарь  $DocCount(w, c)$  для всех  $w$
- ③ Проредить словарь по частоте
- ④ Для каждого документа  $d$ 
  - ① Для каждого слова  $w$  из документа  $d$  найти  $WordCount(w, d)$ 
    - Записать в результирующий вектор в позицию  $w$  значение  $TFIDF(w, d, c) = TF(w, d)IDF(w, c)$
  - ② Записать вектор документа в таблицу признаков документов коллекции



## Алгоритм взвешивания признаков по TF-IDF

- ❶ Применить нормализацию текста (стемминг или лемматизацию), выделить базовые элементы (символы, токены, n-граммы)
- ❷ Построить частотный словарь  $DocCount(w, c)$  для всех  $w$
- ❸ Проредить словарь по частоте
- ❹ Для каждого документа  $d$ 
  - ❶ Для каждого слова  $w$  из документа  $d$  найти  $WordCount(w, d)$ 
    - Записать в результирующий вектор в позицию  $w$  значение  $TFIDF(w, d, c) = TF(w, d)IDF(w, c)$
  - ❷ Записать вектор документа в таблицу признаков документов коллекции

- TF-IDF - это способ отбора категориальных признаков
- TF-IDF не использует информацию о метках документов



Есть и другие способы взвешивания признаков по частоте — например, взаимная информация.<sup>[1,2,3]</sup> Она измеряется между двумя случайными событиями или реализациями двух случайных величин. Она характеризует, насколько сильнее мы будем ожидать первое событие, если перед этим пронаходим второе<sup>[4]</sup> (по сравнению с нашими априорными ожиданиями). Эта фраза может звучать сложно, но суть достаточно проста. Рассмотрим вот эту формулу: в знаменателе содержится уровень наших априорных ожиданий о появлении события L, а в числителе — уровень ожидания после наблюдения события W. Все три варианта формул на слайде эквивалентны. Если возвращаться к текстам, то у нас есть два события. Первое — "L": "мы наблюдаем документ из класса L". Второе событие — "W": "мы видим в документе слово W". Все вероятности вычисляются по классическому определению вероятности, то есть как отношение количества положительных исходов к общему числу исходов. Взаимная информация — это тоже способ взвешивания и отбора категориальных признаков.<sup>[5]</sup> В первую очередь, он подходит для задач классификации. В задачах регрессии его тоже можно применять — например, дискретизировав целевое распределение, но это уже сложнее. Он требует наличия двух событий, что усложняет его применение в задачах обучения без учителя, хотя он используется для получения плотных векторных представлений слов.

[1] Точечная взаимная информация [https://en.wikipedia.org/wiki/Pointwise\\_mutual\\_information](https://en.wikipedia.org/wiki/Pointwise_mutual_information)

[2] Взаимная информация - мера связности двух случайных величин - мат.ожидание PMI [https://en.wikipedia.org/wiki/Mutual\\_information](https://en.wikipedia.org/wiki/Mutual_information)

[3] Применение PMI для представления смыслов слов (про это будут ещё лекции 3.2 и 3.3) Levy, Omer, and Yoav Goldberg. "Neural word embedding as implicit matrix factorization." Advances in neural information processing systems. 2014. <https://papers.nips.cc/paper/5477-neural-word-embedding-as-implicit-matrix-factorization.pdf>

[4] Маргинальное (частное, маржинальное) распределение вероятностей  
[https://en.wikipedia.org/wiki/Marginal\\_distribution](https://en.wikipedia.org/wiki/Marginal_distribution)

[5] Ещё несколько способов взвешивания и отбора признаков [https://scikit-learn.org/stable/modules/feature\\_selection.html](https://scikit-learn.org/stable/modules/feature_selection.html)

Список литературы:

Разреженные векторные модели (методы мешка слов), PMI - Pointwise mutual information (метод взаимной информации):

- Christopher D. Manning, Prabhakar Raghavan and Hinrich Schütze, Introduction to Information Retrieval, Cambridge University Press. 2008. <https://nlp.stanford.edu/IR-book/>
- Маргинальное (частное, маржинальное) распределение вероятностей [https://en.wikipedia.org/wiki/Marginal\\_distribution](https://en.wikipedia.org/wiki/Marginal_distribution)
- Точечная взаимная информация [https://en.wikipedia.org/wiki/Pointwise\\_mutual\\_information](https://en.wikipedia.org/wiki/Pointwise_mutual_information)
- Применение PMI для представления смыслов слов (про это будут ещё лекции 3.2 и 3.3) Levy, Omer, and Yoav Goldberg. "Neural word embedding as implicit matrix factorization." *Advances in neural information processing systems*. 2014. <https://papers.nips.cc/paper/5477-neural-word-embedding-as-implicit-matrix-factorization.pdf>
- Взаимная информация - мера связности двух случайных величин - мат.ожидание PMI [https://en.wikipedia.org/wiki/Mutual\\_information](https://en.wikipedia.org/wiki/Mutual_information)
- Ещё несколько способов взвешивания и отбора признаков [https://scikit-learn.org/stable/modules/feature\\_selection.html](https://scikit-learn.org/stable/modules/feature_selection.html)

Из комментариев:

Вопрос (<https://stepik.org/lesson/225311/step/9?discussion=1231897&unit=198054>) :

Что-то я не понял. В обучающей выборке мы знаем метку класса. А на рабочем примере, нам надо ее узнать. Как мы `njulf` векторизуем произвольный документ мы же заранее не знаем какого он класса?

Ответ (Романа Суворова):

Хороший вопрос, я упустил эту тему в лекции. Алгоритм примерно следующий.

1. На этапе обучения для всех слов в корпусе оцениваем их информативность относительно меток документов  $fw = \max_{l} pmi(l, w) + \max_{l} pmi(l, \overline{w})$ , где  $l$  - это событие "документ принадлежит к классу  $l$ ",  $w$  - событие "слово  $w$  встретилось в документе", а  $\overline{w}$  - "слово  $w$  не встретилось в документе". Полученные значения запоминаем в словарике  $w \rightarrow fw$   $\rightarrow fw$ . Можно удалить слова с  $fw < f_{\text{min}}$  ниже заданного порога.

2. На этапе применения для всех слов в документе достаём их веса из словарика, построенного на шаге 1.

Для расчёта  $\text{fwf\_wfw}$  можно использовать и другую формулу, например с усреднением вместо максимума. Можно также учитывать не только  $\text{III}$ , но и  $\overline{\text{I}}$ , по аналогии с  $\text{www}$  и  $\overline{\text{w}}$ . Тут мы постепенно приближаемся к оценке [взаимной информации](#), то есть матожиданию точечной взаимной информации по совместному распределению двух случайных величин.

Обратите внимание, что  $\text{pmi}(l, w) \neq \text{pmi}(l) \text{pmi}(w)$  не зависит от одного конкретного документа - только от распределений. Таким образом, PMI - это более информированный аналог IDF. В принципе, ничего не мешает смешать TF-IDF и PMI и сделать TF-PMI

PMI - это более информированная замена IDF, то есть это глобальные веса слов, которые считаются по обучающей выборке (где метки есть). На новых документах мы просто берём их с трейна.

Pointwise mutual information  $\text{pmi}(A, B)$ :

- измеряется между двумя случайными событиями  $A$  и  $B$  (или реализациями двух случайных величин)
- мера того, насколько сильнее мы будем ожидать появление  $A$  после наблюдения события  $B$ , по сравнению с нашими ожиданиями в ситуации, когда событие  $B$  мы не наблюдали.

Для задачи классификации текстов:

$$\text{pmi}(l, w) = \log \frac{p(w, l)}{p(w)p(l)} = \log \frac{p(l|w)}{p(l)} = \log \frac{p(w|l)}{p(w)}$$

где  $l$  - коллекция документов, соответствующая некоторой метке класса  $L$ ,

$w$  - слово из словаря,

$p(w, l) = \frac{\text{DocCount}(w, l)}{\text{Size}(l)}$  - вероятность встретить слово  $w$  в документе класса  $L$ ,

$p(w) = \frac{\sum_l \text{DocCount}(w, l)}{\sum_l \text{Size}(l)}$  - маржинальная вероятность употребления слова  $w$ ,

$p(l) = \frac{\text{Size}(l)}{\sum_m \text{Size}(m)}$  - маржинальная вероятность встретить документ класса  $L$ .

- PMI - способ отбора категориальных признаков в задачах классификации
- PMI использует информацию о метках документов

SAMSUNG  
Research  
Russia



Мы поближе рассмотрели классическую векторную модель текста и поговорили немного об области её применения, а также рассмотрели несколько способов взвешивания признаков (то есть, слов). [TF-IDF](#) — одна из самых распространённых схем, она не требует информации о метках слов. А также — взаимную информацию, которая опирается на [совместное распределение вероятностей](#) двух случайных событий. Ну и на практике важно помнить ещё пару вещей. Во-первых, бесплатных завтраков не бывает<sup>[1]</sup>, то есть вам всегда нужно пробовать разные варианты, искать тот, который лучше будет работать именно для вашей задачи. Большая часть методов машинного обучения при настройке всё равно назначает какие-то коэффициенты признакам. Таким образом, те веса, которые вы назначаете сами,

играют меньшую роль, чем то, насколько хорошо вы уберёте незначимые или шумные признаки.

[1] [https://en.wikipedia.org/wiki/No\\_free\\_lunch\\_theorem](https://en.wikipedia.org/wiki/No_free_lunch_theorem)

- Мешок слов - область применения
- TF-IDF - определение, алгоритм, преимущества и недостатки
- Pointwise mutual information
- Нужно пробовать разные способы взвешивания признаков
  - No free lunch!
  - Отбор > взвешивание

## 2.2 Создаём нейросеть для работы с текстом

Мы поговорили про представление текста в виде длинного разреженного вектора частот слов. Давайте теперь рассмотрим следующий шаг — как на основе такого вектора принимать решения. Итак, нам дали [коллекцию документов](#) по некоторой тематике: это положительные примеры. А ещё мы нашли другие документы, не относящиеся к данной тематике: это отрицательные примеры. Напомню, что подход, о котором мы собираемся говорить, больше подходит для длинных текстов. Это не значит, что он не будет работать для коротких, просто — для коротких часто можно найти что-то существенно лучше. Нам нужно построить классификатор или решающие правила. Формально — это функция, которая по вещественному вектору размерности D (в данном случае это — размер словаря) выдаёт класс: "1" — если документ относится к интересующей нас тематике, и "0" для всех остальных документов. Мы будем использовать представление документов в виде [мешка слов](#) и попробуем построить классификатор в виде [логистической регрессии](#) (модель определяется следующей формулой). Несмотря на то, что модель называется "регрессией", предназначена она для классификации. Её выход (мы обозначаем его  $y^{\hat{}}$ ) — это вещественное число от нуля до единицы — вероятность того, что документ принадлежит к интересующему нас

классу. Давайте посмотрим на формулу повнимательнее и попробуем понять, что происходит. Внутри у нас стоит [скалярное произведение](#) двух векторов: вектора весов  $w$  и [вектора признаков](#)  $x$ . Результат скалярного произведения — это вещественное число. Изобразим результат графически как прямую. По сути, это [линейная регрессия](#). Однако у нас задача классификации и мы хотим вероятности. Тогда сожмём выход линейной регрессии с помощью сигмоиды и получим примерно следующий график. Он лежит строго между нулём и единицей. Решение о принадлежности к положительному классу принимается с помощью порога — если предсказанная вероятность больше некоторого числа, то документ относится к положительному классу, и не относится в противном случае.<sup>[1]</sup> Вам это всё, наверняка, что-то напоминает...

[1] Чтобы усилить линейную модель, можно генерировать более мощных признаков, например, из комбинаций исходных признаков <https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.PolynomialFeatures.html> или использовать kernel trick - перейти из пространства признаков в пространство расстояний до других объектов [https://scikit-learn.org/stable/modules/kernel\\_approximation.html](https://scikit-learn.org/stable/modules/kernel_approximation.html)

## Дано

- Коллекция документов по тематике - положительные примеры
- Фоновая коллекция документов - отрицательные примеры
- Длина документов - от нескольких предложений

## Требуется

- Построить бинарный классификатор.

$$\text{Classifier} : \mathcal{R}^d \rightarrow \{0, 1\}$$

## Решение

- Представление документов - мешок слов
- Решающее правило - логистическая регрессия



## Определение модели

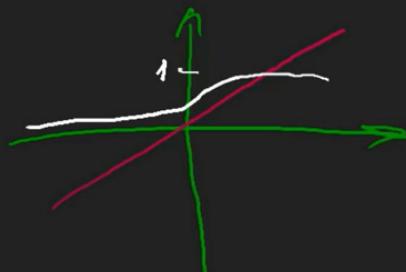
$$\hat{y}(x) = \sigma(w^T x + b)$$

$x \in \mathcal{R}^d$  - вектор признаков, дополненный фиктивным единичным признаком

$\hat{y} \in \mathcal{R}, 0 \leq \hat{y} \leq 1$  - вероятность выпадения 1 для случайной величины  $y \in \{0, 1\}$

$w \in \mathcal{R}^d$  - вектор весов

$\sigma(x) = \frac{1}{1+e^{-x}}$  - логистическая функция (сигмоида)



Конечно, это искусственный нейрон с "D" входами и одним выходом. Настраивать веса этой модели будем, минимизируя значение функции потерь бинарной [кросс-энтропии](#). Она определяется следующим образом: у нас есть два слагаемых: первое слагаемое отвечает за штраф в случае ложно-отрицательных предсказаний, то есть когда документ принадлежит к положительному классу, а модель выдала для него низкую вероятность. Второе слагаемое

отвечает за штраф в случае ложно-положительных предсказаний, и нам нужно минимизировать общий штраф. Данная формула описывает функцию потерь для одного примера. Как всегда, настраивать будем градиентным спуском. Хотя, когда параметров мало, у нас появляется возможность использовать и более быстрые методы оптимизации.<sup>[1]</sup>

[1] Под "более быстрыми методами оптимизации" имеются в виду приближённые методы второго порядка (квази-Ньютоновские): [https://en.wikipedia.org/wiki/Limited-memory\\_BFGS](https://en.wikipedia.org/wiki/Limited-memory_BFGS), <https://github.com/tensorflow/kfac>

Список литературы:

Линейные модели:

- Линейная модель [https://en.wikipedia.org/wiki/Linear\\_classifier](https://en.wikipedia.org/wiki/Linear_classifier)
- Логистическая регрессия в Scikit-learn [https://scikit-learn.org/stable/modules/generated/sklearn.linear\\_model.LogisticRegression.html](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html)
- Обобщённые линейные модели [https://en.wikipedia.org/wiki/Generalized\\_linear\\_model](https://en.wikipedia.org/wiki/Generalized_linear_model)
- Метод наибольшего правдоподобия - откуда можно вывести формулу ВСЕ [https://en.wikipedia.org/wiki/Maximum\\_likelihood\\_estimation](https://en.wikipedia.org/wiki/Maximum_likelihood_estimation)

Чтобы усилить линейную модель, можно

- нагенерировать более мощных признаков, например, из комбинаций исходных признаков <https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.PolynomialFeatures.html>
- или использовать kernel trick - перейти из пространства признаков в пространство расстояний до других объектов [https://scikit-learn.org/stable/modules/kernel\\_approximation.html](https://scikit-learn.org/stable/modules/kernel_approximation.html)

Под "более быстрыми методами оптимизации" имеются в виду приближённые методы второго порядка (квази-Ньютоновские):

- [https://en.wikipedia.org/wiki/Limited-memory\\_BFGS](https://en.wikipedia.org/wiki/Limited-memory_BFGS)
- <https://github.com/tensorflow/kfac>

Важное примечание:

Кросс-энтропия сильно штрафует за сильные ошибки и за неуверенность. На практике это может приводить к тому, что обученная модель чрезмерно "уверена" в своих предсказаниях - значения вероятностей становятся близки к 0 или 1. Поэтому использовать предсказания модели как оценку достоверности предсказаний надо с большой осторожностью. В случае с логистической регрессией калибровка (адекватность оценки вероятности) может быть неплохой, но в случае с глубокими нейросетями это чаще всего не так.

$$\hat{y}(x) = \sigma(w^T x + b)$$

$x \in \mathcal{R}^d$  - вектор признаков, дополненный фиктивным единичным признаком

$\hat{y} \in \mathcal{R}, 0 \leq \hat{y} \leq 1$  - вероятность выпадения 1 для случайной величины  $y \in \{0, 1\}$

$w \in \mathcal{R}^d$  - вектор весов

$\sigma(x) = \frac{1}{1+e^{-x}}$  - логистическая функция (сигмоида)

### Искусственный нейрон с $d$ входами и 1 выходом

#### Процесс обучения

- Функция потерь - бинарная кросс-энтропия

$$BCE(\hat{y}, y) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y}) \rightarrow \min$$

- Алгоритм настройки - градиентный спуск



Давайте теперь рассмотрим многоклассовую задачу, когда нужно предсказывать одну метку из нескольких.<sup>[1]</sup> "с" — это количество различных классов, и оно больше 2. Логистическая регрессия подходит и для такого случая, но нам потребуется пара небольших изменений. Напомню — раньше формула была следующей. Во-первых, у нас теперь не вектор весов, а прямоугольная матрица. Количество строк в этой матрице соответствует количеству классов, а количество столбцов — количеству входных признаков. Во-вторых, на выходе модели у нас уже не одно число, а вектор. Этот вектор описывает распределение вероятностей принадлежности объекта "x" к одному из "с" классов. Чтобы получить на выходе распределение вероятностей, нам уже нужна другая функция активации. Обычно в таких случаях используют "софтмакс". Это вектор-функция, преобразующая вектор вещественных чисел из произвольного диапазона в вектор чисел от нуля до единицы, сумма которых всегда равна единице. Таким образом, значение этого вектора всегда удовлетворяет определению распределения вероятностей. Функцию потерь необходимо также немного поменять. Так как классы у нас взаимоисключающие, достаточно накладывать штраф только для ложно-отрицательных предсказаний. Когда  $y_{ij} \hat{y}_{ij}$  — единица, а  $y_i \hat{y}_i$  близок к нулю, соответствующее слагаемое будет большим.

[1] Метод наибольшего правдоподобия - откуда можно вывести формулу BCE:

[https://en.wikipedia.org/wiki/Maximum\\_likelihood\\_estimation](https://en.wikipedia.org/wiki/Maximum_likelihood_estimation)

## Требуется

Построить классификатор, предсказывающий одну из нескольких меток

$$\text{Classifier} : \mathcal{R}^d \rightarrow \{1, \dots, c\}$$

## Определение модели

$$\hat{y}(x) = \text{softmax}(W \cdot x)$$

$x \in \mathcal{R}^d$  - вектор признаков, дополненный фиктивным единичным признаком

$y \in \mathcal{R}^c; \sum_{i=1}^c \hat{y}_i = 1$  - на выходе - распределение вероятностей

$W \in \mathcal{R}^{c \times d}$  - вектор весов

$$\text{softmax}(x) = \left\{ \frac{e^{x_i}}{\sum_{j=1}^c e^{x_j}} \right\}_i \text{ - функция активации softmax}$$

## Функция потерь

$$CE(\hat{y}, y) = - \sum_{i=1}^c y_i \log(\hat{y}_i)$$



Мы поговорили про логистическую регрессию (или однослойную нейронную сеть), которая, по сути, является линейной регрессией с функцией активации в виде сигмоиды или софтмакса, а также мы рассмотрели варианты для двух и нескольких классов. Наиболее часто используемая функция потерь — [кросс-энтропия](#), а метод оптимизации — градиентный спуск. Ну что ж, теперь мы готовы перейти к практике!

- Логистическая регрессия
  - Линейная регрессия
  - Функция активации -  $\sigma$  или softmax.
- Варианты для двух и нескольких классов
- Функция потерь - кросс-энтропия
- Алгоритм оптимизации - градиентный спуск



## 2.4 Семинар: классификация новостных текстов

```
#####
```

Для прохождения данного семинара Вам потребуется ноутбук [task1\\_20newsgroups.ipynb](#).

Чтобы запустить ноутбук с семинара на своем компьютере:

1) Склонируйте [репозиторий курса](#):

```
git clone https://github.com/Samsung-IT-Academy/stepik-dl-nlp.git
```

2) В терминале выполните команду:

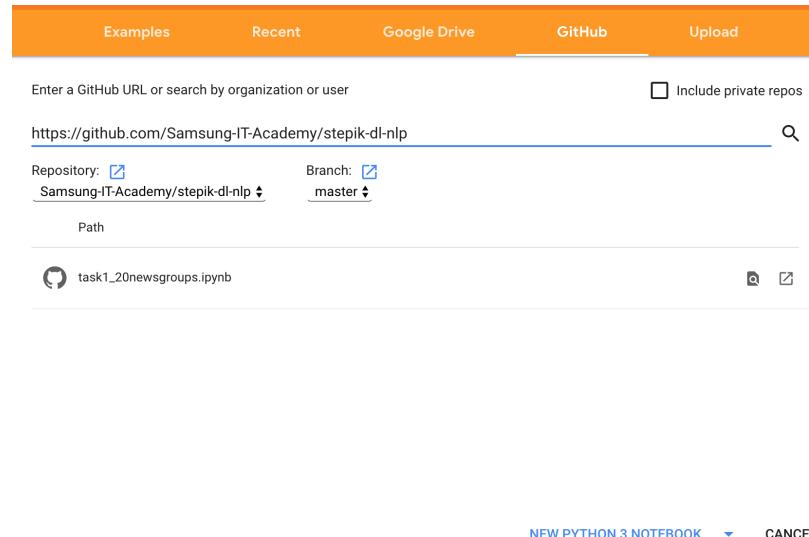
```
pip install -r requirements.txt
```

3) Запустите ноутбук:

```
ipython notebook
```

Чтобы запустить ноутбук на [Google Colab](#):

1) Скачайте ноутбук (вкладка Github, затем прописываете адрес репозитория):



2) Запустите ноутбук.

3) Не забудьте выполнить команду `git clone` из первой (закомментированной) ячейки, чтобы выкачать на colab библиотеку `dlnlputils`:

• Тематическая классификация длинных текстов - TFIDF и LogReg

```
# Если Вы запускаете ноутбук на colab,  
# выполните следующую строку, чтобы подгрузить библиотеку dlnlputils:  
# !git clone https://github.com/Samsung-IT-Academy/stepik-dl-nlp.git
```

Ноутбуки также работают и на Kaggle (следуйте комментариям в ячейках ноутбука).

Ссылка на репозиторий со всеми материалами курса и инструкцией по запуску: <https://github.com/Samsung-IT-Academy/stepik-dl-nlp>  
#####

Всем привет! Наконец-то мы добрались до первой практики. Этот семинар преследует сразу несколько целей. Во-первых, хотелось бы вам напомнить, как работать с pytorch и как описывать простые нейросети с помощью этой библиотеки. Вторая цель — это продемонстрировать, как можно реализовать классический алгоритм с помощью pytorch. В этом семинаре мы будем работать с классическим датасетом для тематической классификации, "[20 новостных групп](#)", который состоит из примерно 20 тысяч сообщений электронной почты, распределенных по 20 категориям. Мы познакомимся с тем, как преобразовывать исходный текст в токены (делать токенизацию), затем мы разберёмся, как строить словарь, как можно взвешивать, как реализуется модель мешка слов или [разреженная векторная модель](#) текста, затем мы реализуем логистическую регрессию на pytorch, обучим её, оценим её качество. А затем мы возьмём библиотеку scikit-learn (алгоритмы векторизации текстов из неё, а также реализацию [логистической регрессии](#)), обучим этот вариант модели и сравним с нашим вариантом, чтобы проверить, правильно ли мы всё сделали. Ну что ж, поехали! Давайте сначала импортируем библиотеки, которые нам понадобятся в этом семинаре. Я предполагаю делать все импорты в одной ячейке ноутбука. Нам понадобиться scikit-learn, numpy и matplotlib, также нам понадобится, собственно, pytorch, а ещё, специально для этого курса, мы разработали маленькую библиотечку, содержащую все необходимые примитивы для того, чтобы пройти все задания в этом курсе. Ну, и наконец — для того, чтобы запуски нашего ноутбука приводили из раза в раз к одному и тому же результату, мы должны зафиксировать инициализацию генераторов случайных чисел. Однако, если вы обучаете нейросети на видеокарте, даже зафиксировав все сиды вы не добьётесь полной повторяемости.<sup>[1,2,3]</sup> Это связано с тем, как именно работает видеокарта. На высоком уровне мы с этим ничего не можем сделать.

[1] <https://pytorch.org/docs/stable/notes/randomness.html>

[2] <https://www.twosigma.com/insights/article/a-workaround-for-non-determinism-in-tensorflow>

[3] <https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9911-determinism-in-deep-learning.pdf>

## Тематическая классификация длинных текстов - TFIDF и LogReg

```
In [1]: import warnings
warnings.filterwarnings('ignore')

from sklearn.datasets import fetch_20newsgroups
from sklearn.metrics import accuracy_score

import numpy as np

import matplotlib.pyplot as plt
%matplotlib inline

import collections

import torch
from torch import nn
from torch.nn import functional as F

import dlnlutils
from dlnlutils.data import tokenize_text_simple_regex, tokenize_corpus, build_vocab, vectorize_texts, SparseFeaturesDataset
from dlnlutils.pipeline import train_eval_loop, predict_with_model, init_random_seeds

init_random_seeds()
```



Следующий шаг — это загрузить наш датасет. Наш датасет состоит из двух частей — из обучающей выборки и тестовой. Давайте посмотрим на размеры этих выборок и выведем какой-нибудь один пример. Видим, что у нас примерно 18 тысяч текстов суммарно, из них примерно две трети содержатся в обучающей выборке, а треть в — тестовой. Сейчас на экране выделен текст первого примера из обучающей выборки. Как вы видите, длина текста — это несколько предложений плюс какая-то мета-информация. Классы в этом датасете обозначаются числами от 0 до 19. Пример на экране имеет метку "7". Хорошо, датасет загрузили, теперь нам нужно преобразовать сырье тексты в признаки. В результате подготовки признаков мы получим две прямоугольные матрицы, строки которых соответствуют текстам, а столбцы — признакам. Две матрицы — потому, что две подвыборки — обучающая и тестовая. Первый шаг почти для всех задач обработки текстов — это токенизация. Токенизация — это разбиение исходного текста на базовые лексические элементы, токены. В нашем семинаре для токенизации мы используем функцию, которая реализована в нашей библиотечке — она называется "tokenize\_corpus". Давайте посмотрим, что же она делает. Вот она — функция "tokenize\_corpus", она принимает на вход список текстов (список строк), а также функцию, которая будет принимать одну строку и возвращать список токенов. Собственно, токенизация корпуса заключается в том, что мы последовательно токенизуем каждый текст. В этом семинаре для токенизации текста мы будем использовать функцию, реализованную чуть выше, которая называется "tokenize\_text\_simple\_rejects". Идея этой функции заключается в том, что мы, с помощью регулярного выражения, описываем всё возможное множество токенов, которое нас интересует, а затем оставляем только более-менее длинные токены. По умолчанию задана минимальная длина токена: 4. Для описания множества допустимых токенов мы используем

выделенное на экране [регулярное выражение](#). Это регулярное выражение срабатывает на непрерывных последовательностях букв или цифр. Таким образом, знаки препинания, пробелы, специальные символы мы полностью игнорируем. Нас интересуют только непрерывные последовательности букв и цифр. Это регулярное выражение уж очень простое для реальной жизни, но для задач нашего курса — вполне достаточно. Вполне возможно что и в некоторых реальных проектах такого токенизатора вам будет вполне достаточно. Итак, в результате токенизации первого текста из обучающей выборки мы получаем следующий список токенов. После токенизации у нас остались только последовательности, состоящие из букв и цифр.

Хорошие ресурсы для изучения и отладки регулярных выражений:

- <https://regex101.com/>
- <https://docs.python.org/3/howto/regex.html>
- <https://habr.com/ru/post/349860/>

## Предобработка текстов и подготовка признаков

```
In [2]: train_source = fetch_20newsgroups(subset='train')
test_source = fetch_20newsgroups(subset='test')

print('Количество обучающих текстов', len(train_source['data']))
print('Количество тестовых текстов', len(test_source['data']))
print()
print(train_source['data'][0].strip())

print()
print('Метка', train_source['target'][0])
...

```

### Подготовка признаков

```
In [3]: train_tokenized = tokenize_corpus(train_source['data'])
test_tokenized = tokenize_corpus(test_source['data'])

print(' '.join(train_tokenized[0]))
...

```

```
In [4]: MAX_DF = 0.8
MIN_COUNT = 5
vocabulary, word_doc_freq = build_vocabulary(train_tokenized,
...

```



```
print()
print('Метка', train_source['target'][0])
Количество обучающих текстов 11314
Количество тестовых текстов 7532
```

```
From: lerxst@wam.umd.edu (where's my thing)
Subject: WHAT car is this?
Nntp-Posting-Host: rac3.wam.umd.edu
Organization: University of Maryland, College Park
Lines: 15
```

I was wondering if anyone out there could enlighten me on this car I saw the other day. It was a 2-door sports car, looked to be from the late 60s/early 70s. It was called a Bricklin. The doors were really small. In addition, the front bumper was separate from the rest of the body. This is all I know. If anyone can tell me a model name, engine specs, years of production, where this car is made, history, or whatever info you have on this funky looking car, please e-mail.

Thanks,  
- IL  
---- brought to you by your neighborhood Lerxst ----

Метка 7



### Подготовка признаков

Метка /

## Подготовка признаков

```
In [3]: train_tokenized = tokenize_corpus(train_source['data'])
test_tokenized = tokenize_corpus(test_source['data'])

print(' '.join(train_tokenized[0]))
```

```
...  
In [4]: MAX_DF = 0.8
MIN_COUNT = 5
vocabulary, word_doc_freq = build_vocabulary(train_tokenized,
                                               max_doc_freq=MAX_DF,
                                               min_count=MIN_COUNT)
UNIQUE_WORDS_N = len(vocabulary)
print('Количество уникальных токенов', UNIQUE_WORDS_N)
print(list(vocabulary.items())[:10])
```

```
...  
In [5]: plt.hist(word_doc_freq, bins=20)
plt.title('Распределение относительных частот слов')
plt.yscale('log');
```



```
4 import numpy as np
5
6 TOKEN_RE = re.compile(r'[\w\d]+')
7
8
9 def tokenize_text_simple_regex(txt, min_token_size=4):
10     txt = txt.lower()
11     all_tokens = TOKEN_RE.findall(txt)
12     return [token for token in all_tokens if len(token) >= min_token_size]
13
14
15 def character_tokenize(txt):
16     return list(txt)
17
18
19 def tokenize_corpus(texts, tokenizer=tokenize_text_simple_regex, **tokenizer_kwargs):
20     return [tokenizer(text, **tokenizer_kwargs) for text in texts]
21
22
23 def add_fake_token(word2id, token='<PAD>'):
24     word2id_new = {token: i + 1 for token, i in word2id.items()}
25     word2id_new[token] = 0
26     return word2id_new
27
28
29 def texts_to_token_ids(tokenized_texts, word2id):
30     return [[word2id[token] for token in text if token in word2id]
31             for text in tokenized_texts]
```



```
In [3]: train_tokenized = tokenize_corpus(train_source['data'])
test_tokenized = tokenize_corpus(test_source['data'])

print(' '.join(train_tokenized[0]))

from lerxst where thing subject what this nntp posting host rac3 organization university mary
land college park lines wondering anyone there could enlighten this other door sports looked
from late early called bricklin doors were really small addition front bumper separate from
rest body this know anyone tellme model name engine specs years production where this made h
istory whatever info have this funky looking please mail thanks brought your neighborhood ler
xst
```

```
In [4]: MAX_DF = 0.8
MIN_COUNT = 5
vocabulary, word_doc_freq = build_vocabulary(train_tokenized,
                                               max_doc_freq=MAX_DF,
                                               min_count=MIN_COUNT)
UNIQUE_WORDS_N = len(vocabulary)
print('Количество уникальных токенов', UNIQUE_WORDS_N)
print(list(vocabulary.items())[:10])
...
```

```
In [5]: plt.hist(word_doc_freq, bins=20)
plt.title('Распределение относительных частот слов')
plt.yscale('log');
```



Отлично, токенизировали тексты. Следующий шаг — это построить словарь. Что такое словарь? Словарь — это отображение из строкового представления токена в его номер. Нейросети, да и компьютеры вообще, не умеют работать с текстами, как это делают люди. Для того, чтобы они смогли работать с текстами, нам нужно тексты представить в виде чисел. Поэтому мы нумеруем все токены и затем в датасете строки заменяем на соответствующие им числа. Обращаю ваше внимание, что для построения словаря мы используем только обучающую подвыборку.

Словарь мы строим с помощью функции "build\_vocabulary", которая описана в нашей маленькой библиотечке. Давайте посмотрим, как же она работает. Эта функция принимает на вход список списков. Внешний список представляет собой датасет, а внутренние списки представляют отдельные документы и элементы внутренних списков — это токены в строковом виде. Кроме того, эта функция принимает дополнительные параметры, смысл которых мы сейчас разберём. Первый шаг при построении словаря — это подсчитать частоты токенов. Частоты мы будем хранить в [словаре со значением по умолчанию](#). Удобная реализация такого словаря есть в стандартной библиотеке python. Здесь мы идём по всем текстам, считаем количество текстов, а также идём по всем токенам в каждом тексте и увеличиваем счётчик в нашем словаре на единицу для каждого словоупотребления. Если у нас большой [корпус](#) (и особенно, если мы работаем с [N-граммами](#)), то словарь очень быстро разрастается и его размер достигает миллионов, десятков миллионов... Такой словарь, в некотором смысле, бесполезен. Мы хотим всё-таки небольшой словарь. Поэтому, когда мы подсчитали частоты токенов, мы удаляем из словаря те токены, которые мы считаем неинформативными. Стандартный способ удалить неинформативные токены — это отсечение по частоте. Во-первых, мы удаляем совсем редкие токены — это могут быть либо специальные слова, либо

опечатки, либо какие-то идентификаторы, случайно встречающиеся в тексте, или не вычищенные из текстов в результате предобработки. Второй критерий для фильтрации — это ограничение частоты сверху. Давайте вспомним [распределение Ципфа](#) (оно выглядит примерно так). И то, что мы хотим — это оставить только середину этого распределения, а всё остальное убрать. В хвосте распределения — опечатки и слишком специальные слова, а в голове распределения содержатся союзы, предлоги, местоимения, вопросительные слова. Если рассматривать их ценность в контексте тематической классификации, то они практически бесполезны — они встречаются во всех текстах вне зависимости от тематики. Далее мы сортируем токены по убыванию частоты их встречаемости. Мы делаем это для того, чтобы наиболее часто встречающиеся токены получили наименьшие идентификаторы. Иногда это просто удобно. Кроме того, на этом этапе мы добавляем в список наших токенов фиктивный токен, причём добавляем его мы в начало. Мы делаем это для того, чтобы он получил идентификатор "0". Зачем это нужно — мы подробнее рассмотрим в следующих семинарах, когда начнём использовать свёрточные или рекуррентные нейросети. После фильтрации по количеству словоупотреблений и по относительной частоте у нас всё равно может остаться слишком много слов, поэтому мы добавляем дополнительное ограничение и оставляем только заданное количество наиболее частотных слов. Ну и, напоследок, мы назначаем идентификаторы токенам и строим словарь в виде [python dict](#). Идентификаторы словам назначаются инкрементально, в том порядке, в котором они встречались в списке "sorted word counts". Переменная "word\_to\_id" — это и есть тот словарь, который мы хотели построить. Но кроме него наша функция возвращает ещё вектор весов — "word\_to\_frequency". Этот вектор содержит относительные частоты всех токенов в нашем датасете. Этот вектор нам понадобится на этапе формирования матрицы признаков. В этом семинаре, при построении словаря, мы используем следующее ограничение на частоты слов — мы говорим, что токены, встретившиеся менее чем 5 раз в обучающей выборке, нас не интересуют. А ещё мы говорим, что токены, которые встречаются более, чем в 80% документов обучающей выборки нас тоже не интересуют, потому что они слишком частотные. Ну что ж, давайте посмотрим, что у нас получилось. С учётом фильтрации по частоте у нас получилось примерно 20 тысяч уникальных токенов — это достаточно много, на самом деле, для такого датасета. А также, на экране вы видите 10 самых частотных токенов в датасете вместе с их числовыми идентификаторами. Давайте посмотрим на гистограмму относительных частот слов. Правда ли, что [закон Ципфа](#) выполняется? Как вы видите, токенов, которые встретились хотя бы в половине документов обучающей выборки — очень мало (кажется, их даже меньше десяти суммарно). Токенов, которые встретились хотя бы в 10% обучающей выборки — побольше, это уже число порядка нескольких сотен. Ну, и мода нашего распределения находится около нуля, то есть большая часть слов (а именно, порядка 10000 уникальных токенов) встречаются менее, чем в 5% процентах документов. Ну что ж, кажется закон Ципфа действительно выполняется.

Из комментариев:

Вопрос:

> Если у нас большой корпус (и особенно, если мы работаем с N-граммами), то словарь очень быстро разрастается и его размер достигает миллионов, десятков миллионов... Такой словарь, в некотором смысле, бесполезен.

Я не уверен, что корректно говорить, что большой словарь (миллионы, десятки миллионов слов) "бесполезен".

Здесь есть несколько моментов, и я их все конечно даже не упомяну, тема богатая.

В первую очередь нужно определить критерий того, что словарь становится "слишком большой". Что значит "слишком", на что размер негативно влияет?

И второй важный момент: есть два этапа - тренировка модели и использование натренированной модели, и влияние размера словаря может отличаться для этих двух этапов.

И это влияние еще конечно зависит от конкретной модели.

С точки зрения ресурсов, которые требуются на тренировку и использование модели, чем больше размер словаря, тем больше ресурсов требуется, и в какой-то момент (особенно, если использовать словарь N-граммов) может оказаться невыгодно или невозможно использовать словарь определенного размера. Просто для того, чтобы задать некоторый масштаб, недавно я использовал word2vec модель натренированную на Wikipedia + Pubmed Central + Pubmed, в ней было 5.5 миллионов уникальных слов с размерностью векторов слов 200, и в памяти она занимала около 4 Гб. Но это конечно dense vectors, не bag of words. С такими большими bag of words и разреженными матрицами я не пробовал работать.

А вот не с технической, а с концептуальной точки зрения, чем словарь в несколько миллионов слов плох? Для моделей которые так или иначе моделируют взаимоотношения между словами (word2vec и все последующие, включая BERT с вариациями), кажется, что чем больше слов, тем может быть лучше. Может быть именно для bag of words, где сколько слов не возьми, все равно они все рассматриваются независимыми, увеличение словаря быстро перестает помогать?

Ответ(Романа Суворова):

спасибо за интересные рассуждения!

Я не уверен, что корректно говорить, что большой словарь (миллионы, десятки миллионов слов) "бесполезен".

В первую очередь имеется в виду две вещи:

1. работать с большим словарём вычислительно дорого (в линейных моделях может быть не так дорого, а в нейросетях критически дорого)
2. для подавляющей части токенов в таком словаре у нас есть очень маленькая статистика использования, поэтому модель может "шуметь" на новых данных

Конечно, есть приложения, где это не так важно.

модель натренированную на Wikipedia + Pubmed Central + Pubmed, в ней было 5.5 миллионов уникальных слов

Привет обработчикам медицинской литературы! А Вы пробовали оценить, насколько адекватны эмбеддинги для слов с рангом > 400k? Года 4 назад я тоже строил эмбеддинги и

модели для извлечения информации на пабмеде и в моих экспериментах качество эмбеддингов деградировало достаточно быстро.

с концептуальной точки зрения, чем словарь в несколько миллионов слов плох?

Главная концептуальная проблема - количество слов должно соответствовать количеству данных. Как я уже говорил, если слово встречается раз 10 в многомилионном корпусе, скорее всего надёжных выводов и закономерностей из такой статистики мы сделать просто не сможем. В BERT словарь вообще не больше 30-50к токенов обычно используется. К тому же, когда мы ограничиваем модель, мы заставляем её сжимать данные, и через сжатие выделять закономерности и лучше обобщаться на новые данные.

```
In [4]: MAX_DF = 0.8
MIN_COUNT = 5
vocabulary, word_doc_freq = build_vocabulary(train_tokenized,
                                               max_doc_freq=MAX_DF,
                                               min_count=MIN_COUNT)
UNIQUE_WORDS_N = len(vocabulary)
print('Количество уникальных токенов', UNIQUE_WORDS_N)
print(list(vocabulary.items())[:10])
...
```

```
In [5]: plt.hist(word_doc_freq, bins=20)
plt.title('Распределение относительных частот слов')
plt.yscale('log');
```

```
In [6]: VECTORIZATION_MODE = 'tfidf'
train_vectors = vectorize_texts(train_tokenized, vocabulary, word_doc_freq,
                                 mode=VECTORIZATION_MODE)
test_vectors = vectorize_texts(test_tokenized, vocabulary, word_doc_freq,
                                mode=VECTORIZATION_MODE)

print('Размерность матрицы признаков обучающей выборки', train_vectors.shape)
print('Размерность матрицы признаков тестовой выборки', test_vectors.shape)
print()
print('Количество ненулевых элементов в обучающей выборке', train_vectors.nnz)
print('Процент заполненности матрицы признаков {:.2f}%'.format(
```



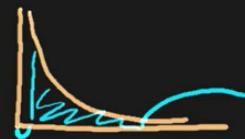
```
34 def build_vocabulary(tokenized_texts, max_size=1000000, max_doc_freq=0.8, min_count=5, pad_word=None):
35     word_counts = collections.defaultdict(int)
36     doc_n = 0
37
38     # посчитать количество документов, в которых употребляется каждое слово
39     # а также общее количество документов
40     for txt in tokenized_texts:
41         doc_n += 1
42         unique_text_tokens = set(txt)
43         for token in unique_text_tokens:
44             word_counts[token] += 1
45
46     # убрать слишком редкие и слишком частые слова
47     word_counts = {word: cnt for word, cnt in word_counts.items()
48                   if cnt >= min_count and cnt / doc_n <= max_doc_freq}
49
50     # отсортировать слова по убыванию частоты
51     sorted_word_counts = sorted(word_counts.items(),
52                                 reverse=True,
53                                 key=lambda pair: pair[1])
54
55     # добавим несуществующее слово с индексом 0 для удобства пакетной обработки
56     if pad_word is not None:
57         sorted_word_counts = [(pad_word, 0)] + sorted_word_counts
58
59     # если у нас по прежнему слишком много слов, оставить только max_size самых частотных
60     if len(word_counts) > max_size:
61         sorted_word_counts = sorted_word_counts[:max_size]
```



```

37
38 # посчитать количество документов, в которых употребляется каждое слово
39 # а также общее количество документов
40 for txt in tokenized_texts:
41     doc_n += 1
42     unique_text_tokens = set(txt)
43     for token in unique_text_tokens:
44         word_counts[token] += 1
45
46 # убрать слишком редкие и слишком частые слова
47 word_counts = {word: cnt for word, cnt in word_counts.items()
48                 if cnt >= min_count and cnt / doc_n <= max_doc_freq}
49
50 # отсортировать слова по убыванию частоты
51 sorted_word_counts = sorted(word_counts.items(),
52                             reverse=True,
53                             key=lambda pair: pair[1])
54
55 # добавим несуществующее слово с индексом 0 для удобства пакетной обработки
56 if pad_word is not None:
57     sorted_word_counts = [(pad_word, 0)] + sorted_word_counts
58
59 # если у нас по прежнему слишком много слов, оставить только max_size самых частотных
60 if len(word_counts) > max_size:
61     sorted_word_counts = sorted_word_counts[:max_size]
62
63 # нумеруем слова
64 word2id = {word: i for i, (word, _) in enumerate(sorted_word_counts)}

```



```

43     for token in unique_text_tokens:
44         word_counts[token] += 1
45
46 # убрать слишком редкие и слишком частые слова
47 word_counts = {word: cnt for word, cnt in word_counts.items()
48                 if cnt >= min_count and cnt / doc_n <= max_doc_freq}
49
50 # отсортировать слова по убыванию частоты
51 sorted_word_counts = sorted(word_counts.items(),
52                             reverse=True,
53                             key=lambda pair: pair[1])
54
55 # добавим несуществующее слово с индексом 0 для удобства пакетной обработки
56 if pad_word is not None:
57     sorted_word_counts = [(pad_word, 0)] + sorted_word_counts
58
59 # если у нас по прежнему слишком много слов, оставить только max_size самых частотных
60 if len(word_counts) > max_size:
61     sorted_word_counts = sorted_word_counts[:max_size]
62
63 # нумеруем слова
64 word2id = {word: i for i, (word, _) in enumerate(sorted_word_counts)}
65
66 # нормируем частоты слов
67 word2freq = np.array([cnt / doc_n for _, cnt in sorted_word_counts], dtype='float32')
68
69 return word2id, word2freq
70

```



```
print('Количество уникальных токенов', UNIQUE_WORDS_N)
print(list(vocabulary.items())[:10])

Количество уникальных токенов 21628
[('that', 0), ('this', 1), ('have', 2), ('with', 3), ('writes', 4), ('article', 5), ('posting', 6), ('host', 7), ('nntp', 8), ('there', 9)]
```

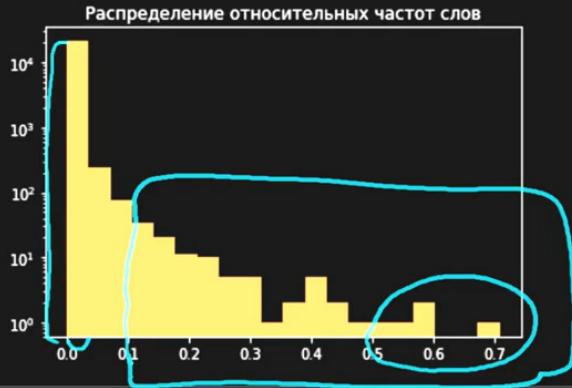
```
In [5]: plt.hist(word_doc_freq, bins=20)
plt.title('Распределение относительных частот слов')
plt.yscale('log');
```

```
In [6]: VECTORIZATION_MODE = 'tfidf'
train_vectors = vectorize_texts(train_tokenized, vocabulary, word_doc_freq,
                                 mode=VECTORIZATION_MODE)
test_vectors = vectorize_texts(test_tokenized, vocabulary, word_doc_freq,
                               mode=VECTORIZATION_MODE)

print('Размерность матрицы признаков обучающей выборки', train_vectors.shape)
print('Размерность матрицы признаков тестовой выборки', test_vectors.shape)
print()
print('Количество ненулевых элементов в обучающей выборке', train_vectors.nnz)
print('Процент заполненности матрицы признаков {:.2f}%'.format(
    train_vectors.nnz * 100 / (train_vectors.shape[0] * train_vectors.shape[1])))
print()
print('Количество ненулевых элементов в тестовой выборке', test_vectors.nnz)
print('Процент заполненности матрицы признаков {:.2f}%'.format(
    test_vectors.nnz * 100 / (test_vectors.shape[0] * test_vectors.shape[1])))
```



```
In [5]: plt.hist(word_doc_freq, bins=20)
plt.title('Распределение относительных частот слов')
plt.yscale('log');
```



```
In [6]: VECTORIZATION_MODE = 'tfidf'
train_vectors = vectorize_texts(train_tokenized, vocabulary, word_doc_freq,
                                 mode=VECTORIZATION_MODE)
test_vectors = vectorize_texts(test_tokenized, vocabulary, word_doc_freq,
                               mode=VECTORIZATION_MODE)
```



Датасет мы токенизировали, словарь построили, можно перейти и к построению матрицы признаков. Для построения матрицы признаков по методу мешка слов мы будем использовать нашу функцию "vectorize\_texts". Давайте посмотрим, как она реализована. Эта функция принимает три основных параметра, а именно — список токенизованных текстов (то есть список списков строк), словарь (то есть, отображение из строк в числа) из токенов в их

идентификаторы или в их номера, а также вектор, содержащий действительные числа и описывающий относительные частоты токенов. Функция принимает ещё два параметра — а именно, это алгоритм взвешивания токенов по частоте (по умолчанию выбран [TF-IDF](#)), а также флаг, который говорит, нужно ли перемасштабировать данные после взвешивания, или не нужно. Мы чуть позже, разберём, что именно он делает. Сначала мы проверяем, что нам передали режим, с которым мы умеем работать. Далее мы строим прямоугольную матрицу, в которой количество строк соответствует количеству текстов, а количество столбцов соответствует количеству уникальных токенов. Эта матрица будет содержать счётчики: сколько каждый токен встретился в каждом документе. Давайте вспомним гистограмму относительных частот слов. У нас очень мало частых слов — более того, мы их специально выкинули, — матрица счётчиков будет крайне разреженная. Мы можем использовать разреженные матрицы из библиотеки `scipy`. Разреженные матрицы хранят только ненулевые элементы. Алгоритм подсчёта предельно прост — мы идём по всем текстам, по всем токенам в каждом тексте, и если токен есть в словаре, то мы увеличиваем на единичку соответствующую ячейку матрицы. Если мы получили какой-то неизвестный токен, то, возможно, нам дали либо новые тексты, которых не было в обучающей выборке, либо этот токен был в обучающей выборке, но мы его отфильтровали по частоте. Далее мы реализуем процедуру взвешивания. По умолчанию, в этой функции реализовано 4 алгоритма взвешивания. Первый алгоритм — это бинарные вектора. Этот алгоритм приводит к тому, что матрица весов будет содержать единичку, если токен хотя бы один раз встретился в документе, и нолик, если токена в документе не было. Следующий алгоритм — это относительные частоты слов в документе — TF (term frequency). В этом алгоритме мы сначала преобразовываем матрицу в другой формат разреженных матриц<sup>[2]</sup>. Формат CSR обеспечивает эффективные операции над строками матриц, но крайне неэффективен для операций над столбцами. Часто выгоднее несколько раз конвертировать матрицу (сначала в один формат, сделать операцию, конвертировать в другой, сделал другую операцию...), чем делать операцию в неподходящем для неё формате. Итак, мы конвертируем в разреженную матрицу для строк и делим каждый счётчик в каждой строке на сумму элементов в этой строке — по сути, мы делим количество употреблений токена в документе на длину этого документа, то есть мы получаем относительную частоту токена в документе. Следующий режим — это IDF (inverse document frequency). Этот алгоритм взвешивания приводит к тому, что, в результирующей матрице признаков, убирается вся информация о частоте токена в документе. Но в этой матрице есть информация о частоте токена в [корпусе](#). Здесь мы сначала получаем матрицу индикаторов (есть ли токен в документе, или нет), а затем домножаем индикаторы на вектор весов слов, который мы вычислили на этапе построения словаря. Функция "multiply" реализует операцию матричного произведения.

[1] Следующий алгоритм объединяет предыдущие два — здесь мы сначала находим TF для каждого токена, а затем домножаем TF на IDF. Всё просто. Ну, и напоследок — мы масштабируем датасет так, чтобы все элементы матрицы укладывались в диапазон от нуля до единицы. Стандартизация входных весов необходима для стабильной работы многих

алгоритмов машинного обучения. Способ стандартизации весов зависит от природы данных. Часто используется приведение к нормальному распределению с мат.ожиданием в нуле и единичной дисперсией. Но такой подход нам здесь не подходит... Потому что у нас матрица разреженная, и если мы сдвинем эту матрицу на её мат.ожидание, то мы получим не разреженную матрицу, и, скорее всего, она у нас для нормального датасета даже в память не влезет. Поэтому мы используем другой способ стандартизации, а именно мин-макс-стандартизацию, то есть мы говорим, что минимально допустимое значение признака в датасете — это 0, а максимально допустимое — это 1. Ну, и напоследок, мы переводим нашу разреженную матрицу в формат, ориентированный на эффективную работу со строками, потому, что алгоритмы машинного обучения, которые мы далее будем использовать, читают документы один за другим и настраивают свои веса, то есть нам нужно уметь эффективно брать отдельный документ. Давайте, например, будем использовать алгоритм TF-IDF. Обращаю ваше внимание, что для векторизации и обучающей, и тестовой выборки, используется один и тот же набор параметров, то есть один и тот же словарь, один и тот же вектор частот и один и тот же режим векторизации.

[1] здесь нужно небольшое исправление: функция multiply реализует поэлементное произведение.

[2] CSR, Compressed Sparse Row matrix

([https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csr\\_matrix.html](https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csr_matrix.html))

Замечания к видео от Романа Суворова:

1. 4:07 - ошибочно говорится "функция multiply реализует операцию матричного произведения" - на самом деле multiply реализует поэлементное произведение.

Из комментариев:

Примечание от Романа Суворова:

Некоторые удобные конструкции языка Python - list comprehension и dict comprehension:

- <https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions>
- <https://docs.python.org/3/tutorial/datastructures.html#dictionaries>
- <https://cmdlinetips.com/2018/01/5-examples-using-dict-comprehension/>

Разреженные массивы в Scipy:

- <https://docs.scipy.org/doc/scipy/reference/sparse.html>



```
In [6]: VECTORIZATION_MODE = 'tfidf'
train_vectors = vectorize_texts(train_tokenized, vocabulary, word_doc_freq,
                                mode=VECTORIZATION_MODE)
test_vectors = vectorize_texts(test_tokenized, vocabulary, word_doc_freq,
                               mode=VECTORIZATION_MODE)

print('Размерность матрицы признаков обучающей выборки', train_vectors.shape)
print('Размерность матрицы признаков тестовой выборки', test_vectors.shape)
print()
print('Количество ненулевых элементов в обучающей выборке', train_vectors.nnz)
print('Процент заполненности матрицы признаков {:.2f}%'.format(
    train_vectors.nnz * 100 / (train_vectors.shape[0] * train_vectors.shape[1])))
print()
print('Количество ненулевых элементов в тестовой выборке', test_vectors.nnz)
print('Процент заполненности матрицы признаков {:.2f}%'.format(
    test_vectors.nnz * 100 / (test_vectors.shape[0] * test_vectors.shape[1])))
...
...  
  
In [7]: plt.hist(train_vectors.data, bins=20)
plt.title('Распределение весов признаков')
plt.yscale('log');
```



```
6 def vectorize_texts(tokenized_texts, word2id, word2freq, mode='tfidf', scale=True):
7     assert mode in {'tfidf', 'idf', 'tf', 'bin'}
8
9     # считаем количество употреблений каждого слова в каждом документе
10    result = scipy.sparse.dok_matrix((len(tokenized_texts), len(word2id)), dtype='float32')
11    for text_i, text in enumerate(tokenized_texts):
12        for token in text:
13            if token in word2id:
14                result[text_i, word2id[token]] += 1
15
16    # получаем бинарные вектора "встречается или нет"
17    if mode == 'bin':
18        result = (result > 0).astype('float32')
19
20    # получаем вектора относительных частот слова в документе
21    elif mode == 'tf':
22        result = result.tocsr()
23        result = result.multiply(1 / result.sum(1))
24
25    # полностью убираем информацию о количестве употреблений слова в данном документе,
26    # но оставляем информацию о частотности слова в корпусе в целом
27    elif mode == 'idf':
28        result = (result > 0).astype('float32').multiply(1 / word2freq)
29
30    # учитываем всю информацию, которая у нас есть:
31    # частоту слова в документе и частоту слова в корпусе
32    elif mode == 'tfidf':  
...
```



```
21 # получаем вектора относительных частот слова в документе
22 elif mode == 'tf':
23     result = result.tocsr()
24     result = result.multiply(1 / result.sum(1))
25
26 # полностью убираем информацию о количестве употреблений слова в данном документе,
27 # но оставляем информацию о частотности слова в корпусе в целом
28 elif mode == 'idf':
29     result = (result > 0).astype('float32').multiply(1 / word2freq)
30
31 # учитываем всю информацию, которая у нас есть:
32 # частоту слова в документе и частоту слова в корпусе
33 elif mode == 'tfidf':
34     result = result.tocsr()
35     result = result.multiply(1 / result.sum(1)) # разделить каждую строку на её длину
36     result = result.multiply(1 / word2freq) # разделить каждый столбец на вес слова
37
38 if scale:
39     result = result.tocsc()
40     result -= result.min()
41     result /= (result.max() + 1e-6)
42
43 return result.tocsr()
44
45
46 class SparseFeaturesDataset(Dataset):
47     def __init__(self, features, targets):
48         self.features = features
```



Давайте посмотрим на характеристики матрицы, которые у нас получились. Количество строк в этих матрицах соответствует количеству примеров в обучающей и в тестовой выборке соответственно, а количество столбцов соответствует количеству уникальных токенов, то есть размеру словаря. А ещё на экране вы видите процент заполненности матриц. Для того, чтобы посчитать, мы взяли количество ненулевых элементов и поделили на полный размер матрицы, то есть на произведение количества строк и количества столбцов. Как видите, в этих матрицах заполнено меньше 0.5% элементов, то есть, используя разреженные матрицы, мы экономим гигантское количество памяти. Давайте, для интереса, посмотрим — как же значения этой матрицы распределены. В принципе, здесь также выполняется [закон Ципфа](#). Все значения матрицы лежат строго в диапазоне от нуля до единицы — то чего мы и хотели получить. Давайте теперь проанализируем распределение классов в нашем датасете. У нас всего 20 классов и... о, хорошая новость — классы распределены практически равномерно в обучающей выборке. Да и в тестовой — тоже равномерно. Поэтому, так как классы распределены почти равномерно, мы можем смело использовать [accuracy](#) (или долю правильных предсказаний) как рабочую метрику. Если бы распределение классов было скошенным, эта метрика было бы уже неподходящей, она бы давала сильно завышенные оценки.

```
test_vectors = vectorize_texts(test_tokenized, vocabulary, word_doc_freq,
                               mode=VECTORIZATION_MODE)

print('Размерность матрицы признаков обучающей выборки', train_vectors.shape)
print('Размерность матрицы признаков тестовой выборки', test_vectors.shape)
print()
print('Количество ненулевых элементов в обучающей выборке', train_vectors.nnz)
print('Процент заполненности матрицы признаков {:.2f}%'.format(
    train_vectors.nnz * 100 / (train_vectors.shape[0] * train_vectors.shape[1])))
print()
print('Количество ненулевых элементов в тестовой выборке', test_vectors.nnz)
print('Процент заполненности матрицы признаков {:.2f}%'.format(
    test_vectors.nnz * 100 / (test_vectors.shape[0] * test_vectors.shape[1])))
```

Размерность матрицы признаков обучающей выборки (11314, 21628)  
 Размерность матрицы признаков тестовой выборки (7532, 21628)

Количество ненулевых элементов в обучающей выборке 1126792  
 Процент заполненности матрицы признаков 0.46%

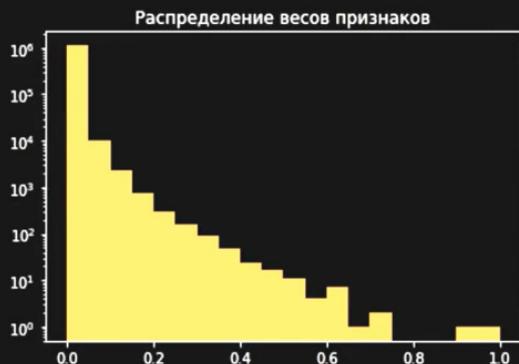
Количество ненулевых элементов в тестовой выборке 721529  
 Процент заполненности матрицы признаков 0.44%

```
In [7]: plt.hist(train_vectors.data, bins=20)
plt.title('Распределение весов признаков')
plt.yscale('log');
```



Количество ненулевых элементов в обучающей выборке 721529  
 Процент заполненности матрицы признаков 0.44%

```
In [7]: plt.hist(train_vectors.data, bins=20)
plt.title('Распределение весов признаков')
plt.yscale('log');
```

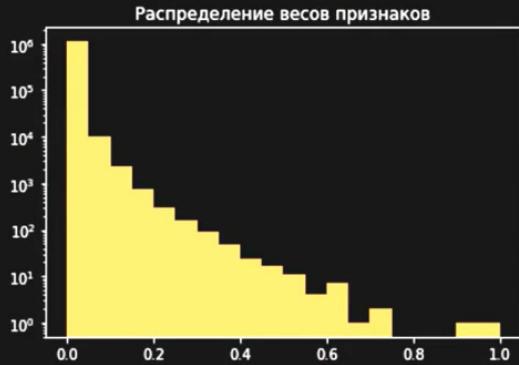


### Распределение классов

```
In [8]: UNIQUE_LABELS_N = len(set(train_source['target']))
```

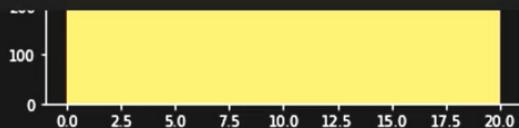
количество иенуемых элементов в обучающей выборке 721527  
Процент заполненности матрицы признаков 0.44%

```
In [7]: plt.hist(train_vectors.data, bins=20)  
plt.title('Распределение весов признаков')  
plt.yscale('log');
```



### Распределение классов

```
In [8]: UNIQUE_LABELS_N = len(set(train_source['target'])) ...
```



```
In [10]: plt.hist(test_source['target'], bins=np.arange(0, 21))  
plt.title('Распределение меток в тестовой выборке');
```



### PvTorch Dataset

Признаки мы подготовили, на метки посмотрели, дело осталось за малым — собственно, обучить модельку. Но сначала мы обернём наши матрицы признаков в "Dataset" — это специальная идиома в pytorch, которая призвана повысить удобство подключения различных датасетов, которые могут подгружаться с жёсткого диска, из памяти; загружаться сразу все — в память, или читаться по чуть-чуть... Здесь мы используем "SparseFeaturesDataset". Он описан в

наши библиотечке. Давайте посмотрим, как он работает. Вот он — это очень простой класс, который принимает на вход в конструктор две матрицы — это матрица признаков, которая разрежена, и матрица меток. По контракту, Dataset должен реализовывать два метода — первый метод "len", он должен возвращать длину датасета, то есть количество примеров в нём, и второй — метод "get item", он должен возвращать один обучающий пример, то есть, в случае нашего семинара — это вектор признаков и метка. Предлагаю вам обратить внимание на выделенный фрагмент кода. Это то, ради чего мы вообще написали этот класс. Особенность в том, что "features" — это разреженная матрица, а pytorch не умеет работать с разреженными матрицами. Но, с другой стороны, мы не хотим конвертировать всю матрицу обучающего датасета в плотное представление, потому что у нас памяти не хватит. Поэтому мы храним весь dataset в разреженном виде, но, когда нам нужно выбрать один пример из датасета, мы выбираем только его из разреженной матрицы, конвертируем в плотное представление и заворачиваем в "torch.Tensor". Аналогично поступаем и с метками. Ну что же, давайте теперь опишем нашу модель. Наша модель — это [логистическая регрессия](#). Как вы помните, логистическая регрессия — это [линейная регрессия](#), выход которой сжимается в диапазон от нуля до единицы с помощью логистической функции, то есть сигмоиды. Таким образом, сама модель состоит всего лишь из одного слоя — это линейный слой, у которого количество входов соответствует количеству уникальных токенов, то есть размеру словаря, и количество выходов соответствует количеству меток в датасете. Этую нашу модель мы обучаем с помощью функции "train\_eval\_loop", которая реализует цикл обучения нейросети. Это функция общего назначения, сюда можно подавать модели не только для классификации, не только текстов, в ней реализованы некоторые стандартные фишки, которые используются при обучении нейросетей. Давайте посмотрим, как она работает. Эта функция принимает целую кучу параметров, но среди этих параметров есть четыре главных — это экземпляр нашей модели, это обучающий датасет, валидационный датасет и наша функция потерь, то есть критерий, минимизируя значение которого мы будем настраивать параметры нашей модели. Что делают остальные параметры — предлагаю пока не рассматривать. Сначала мы переносим нашу модель на то устройство, на котором мы будем производить вычисления — это может быть центральный процессор, либо видеокарта. Затем мы создаём оптимизатор, то есть — говорим, как именно мы должны делать градиентный шаг на каждой итерации. Затем, дополнительно, мы настраиваем расписание изменения скорости обучения. Менять длину градиентного шага в процессе обучения — это часто хорошая идея, которая приводит к получению лучших значений метрик и функции потерь. Затем мы берём наши Dataset-ы, которые умеют возвращать обучающий пример по индексу (как мы только что рассмотрели), и передаём эти "Dataset" в "DataLoader" — это объект из pytorch, который умеет в многопоточном режиме собирать батчи примеров. В этом семинаре многопоточный режим нам, в принципе, не нужен, но, в общем случае (и в дальнейших семинарах) он вам может пригодиться. Основной смысл использования многопоточного режима — в том, чтобы всегда загружать видеокарты на 100% — таким образом, максимально быстро учить. Далее мы определяем набор переменных, которые позволяют нам, в ходе обучения, выбрать лучшую модель. То есть, процесс обучения —

стохастический, и модель может как улучшаться в ходе обучения, так и ухудшаться, и не всегда нужно брать последнюю модель. Хорошая практика для выбора лучшей модели в процессе обучения заключается в том, чтобы иметь отложенную выборку, состоящую из некоторого количества примеров, не входящих в обучающую выборку. И, после некоторого количества шагов по обучающей выборке, оценивать качество модели на валидационной выборке (то есть на этой отложенной выборке) и сравнивать качество моделей именно по значениям метрик, вычисленных на отложенной выборке. Далее начинается цикл — обучение состоит из нескольких эпох. Эпоха, в данном случае, носит условный характер — это некоторое количество градиентных шагов. Не обязательно эпоха — это полный проход по датасету. Наша эпоха начинается с того, что мы переводим модель в режим обучения. В данном семинаре это не так важно, но если в нашей нейросети есть такие модули, как dropout или batch norm, критически важно не забывать переводить модель в режим обучения или в режим применения. Далее идёт цикл, реализующий одну эпоху обучения. Мы делаем заданное количество градиентных шагов по обучающей выборке, на каждом шаге мы берём батч примеров — DataLoader нам возвращает уже не отдельные примеры, а целые пачки примеров. Таким образом, в настоящем семинаре, переменная "batch\_x" — это прямоугольная матрица, в которой количество строк равно количеству примеров в батче, то есть размеру батча, а количество столбцов — это количество признаков. Мы копируем данные на то же, устройство на котором была и модель, выполняем прямой проход по модели, получаем предсказания, находим значение критерия (то есть значение функции потерь), очищаем оценки градиента с предыдущего шага, находим новое значение градиентов, и делаем градиентный шаг. Также мы запоминаем среднее значение функции потерь на эпохе. Это полезно для мониторинга процесса обучения.

## PyTorch Dataset

```
In [11]: train_dataset = SparseFeaturesDataset(train_vectors, train_source['target'])
test_dataset = SparseFeaturesDataset(test_vectors, test_source['target'])
```

## Обучение модели на PyTorch

```
In [12]: model = nn.Linear(UNIQUE_WORDS_N, UNIQUE_LABELS_N)

def lr_scheduler(optim):
    return torch.optim.lr_scheduler.ReduceLROnPlateau(optim,
                                                       patience=5,
                                                       factor=0.5,
                                                       verbose=True)

best_val_loss, best_model = train_eval_loop(model=model,
                                             train_dataset=train_dataset,
                                             val_dataset=test_dataset,
                                             criterion=F.cross_entropy,
                                             lr=1e-1,
                                             epoch_n=200,
                                             batch_size=32,
                                             l2_reg_alpha=0,
                                             lr_scheduler_ctor=lr_scheduler)
```



```
42
43     return result.tocsr()
44
45
46 class SparseFeaturesDataset(Dataset):
47     def __init__(self, features, targets):
48         self.features = features
49         self.targets = targets
50
51     def __len__(self):
52         return self.features.shape[0]
53
54     def __getitem__(self, idx):
55         cur_features = torch.from_numpy(self.features[idx].toarray()[0]).float()
56         cur_label = torch.from_numpy(np.asarray(self.targets[idx])).long()
57         return cur_features, cur_label
58
59
60
61
62
63
64
65
66
67
```



```
In [11]: train_dataset = SparseFeaturesDataset(train_vectors, train_source['target'])
test_dataset = SparseFeaturesDataset(test_vectors, test_source['target'])
```

## Обучение модели на PyTorch

```
In [12]: model = nn.Linear(UNIQUE_WORDS_N, UNIQUE_LABELS_N)

def lr_scheduler(optim):
    return torch.optim.lr_scheduler.ReduceLROnPlateau(optim,
                                                       patience=5,
                                                       factor=0.5,
                                                       verbose=True)

best_val_loss, best_model = train_eval_loop(model=model,
                                             train_dataset=train_dataset,
                                             val_dataset=test_dataset,
                                             criterion=F.cross_entropy,
                                             lr=1e-1,
                                             epoch_n=200,
                                             batch_size=32,
                                             l2_reg_alpha=0,
                                             lr_scheduler_ctor=lr_scheduler)
...
...
```



```
17
18
19 def train_eval_loop(model, train_dataset, val_dataset, criterion,
20                     lr=1e-4, epoch_n=10, batch_size=32,
21                     device='cuda', early_stopping_patience=10, l2_reg_alpha=0,
22                     max_batches_per_epoch_train=10000,
23                     max_batches_per_epoch_val=1000,
24                     data_loader_ctor=DataLoader,
25                     optimizer_ctor=None,
26                     lr_scheduler_ctor=None,
27                     shuffle_train=True):
28     """
29     Цикл для обучения модели. После каждой эпохи качество модели оценивается по отложенной выборке.
30     :param model: torch.nn.Module - обучаемая модель
31     :param train_dataset: torch.utils.data.Dataset - данные для обучения
32     :param val_dataset: torch.utils.data.Dataset - данные для оценки качества
33     :param criterion: функция потерь для настройки модели
34     :param lr: скорость обучения
35     :param epoch_n: максимальное количество эпох
36     :param batch_size: количество примеров, обрабатываемых моделью за одну итерацию
37     :param device: cuda/cpu - устройство, на котором выполнять вычисления
38     :param early_stopping_patience: наибольшее количество эпох, в течение которых допускается
39         отсутствие улучшения модели, чтобы обучение продолжалось.
40     :param l2_reg_alpha: коэффициент L2-регуляризации
41     :param max_batches_per_epoch_train: максимальное количество итераций на одну эпоху обучения
42     :param max_batches_per_epoch_val: максимальное количество итераций на одну эпоху валидации
43     :param data_loader_ctor: функция для создания объекта преобразующего датасет в батчи
```





```
48
49     device = torch.device(device)
50     model.to(device)
51
52     if optimizer_ctor is None:
53         optimizer = torch.optim.Adam(model.parameters(), lr=lr, weight_decay=l2_reg_alpha)
54     else:
55         optimizer = optimizer_ctor(model.parameters(), lr=lr)
56
57     if lr_scheduler_ctor is not None:
58         lr_scheduler = lr_scheduler_ctor(optimizer)
59     else:
60         lr_scheduler = None
61
62     train_dataloader = data_loader_ctor(train_dataset, batch_size=batch_size, shuffle=shuffle_train)
63     val_dataloader = data_loader_ctor(val_dataset, batch_size=batch_size, shuffle=False)
64
65     best_val_loss = float('inf')
66     best_epoch_i = 0
67     best_model = copy.deepcopy(model)
68
69     for epoch_i in range(epoch_n):
70         try:
71             epoch_start = datetime.datetime.now()
72             print('Enoxa {}'.format(epoch_i))
73
74             model.train()
```



```
70
71     try:
72         epoch_start = datetime.datetime.now()
73         print('Эпоха {}'.format(epoch_i))
74
75         model.train()
76         mean_train_loss = 0
77         train_batches_n = 0
78         for batch_i, (batch_x, batch_y) in enumerate(train_dataloader):
79             if batch_i > max_batches_per_epoch_train:
80                 break
81
82             batch_x = batch_x.to(device)
83             batch_y = batch_y.to(device)
84
85             pred = model(batch_x)
86             loss = criterion(pred, batch_y)
87
88             model.zero_grad()
89             loss.backward()
90             optimizer.step()
91
92             mean_train_loss += float(loss)
93             train_batches_n += 1
94
95             mean_train_loss /= train_batches_n
96             print('Эпоха: {} итераций, {:.2f} сек'.format(batch_i, (datetime.datetime.now() - epoch_start).total_seconds()))
```



По характеру изменения функции потерь от эпохи к эпохе мы иногда можем понять, что не так с процессом обучения, мы можем увидеть, что модель вообще не сходится или — она очень быстро достигает определённой точки и дальше не учится, или значение функции потерь изменяется практически случайно, с большой дисперсией. Это важный диагностический показатель. Далее мы выводим некоторую полезную информацию, которая говорит нам о том,

как именно идёт процесс обучения. Ну что ж, эпохи обучения прошли, пора бы и оценить качество модели. Собственно, переводим модель в режим "eval" (то есть, в режим предсказания) и объявляем переменные для оценки среднего значения функции потерь на отложенной выборке. Далее мы повторяем практически те же самые действия, что и делали при обучении, но не делаем сам градиентный шаг — мы только получаем предсказание модели и оцениваем значение функции потерь. Важный момент, который позволит сэкономить память на видеокарте — это включить режим "torch no\_grad". Когда этот режим включен, pytorch не сохраняет промежуточные данные, необходимые для вычисления градиентов. Далее мы сравниваем среднее значение функции потерь на валидации на последней эпохе и лучшее значение функции потерь, полученное аналогичным образом на предыдущих эпохах, и если новое среднее значение функции потерь — лучше, то мы сохраняем текущий вариант модели. Мы делаем это с помощью стандартной функции "copy.deepcopy()". А если улучшить значение функции потерь на отложенной выборке после этой эпохи не получилось, то мы проверяем — а как давно у нас вообще получалось улучшить модель? Если с последней хорошей эпохи прошло уже больше заданного количества эпох, то мы говорим — "ну кажется приехали — кажется, дальше улучшить модель не получится". И, в таком случае, мы прекращаем обучение. Ну, и напоследок — если пользователь задал расписание изменения скорости обучения, то мы обновляем скорость обучения с учётом нового значения функции потерь. Тело этого цикла мы обернули в try-except и добавили обработку двух видов исключений. Первое — это "interrupt", то есть — чтобы пользователь мог досрочно остановить обучение, нажав "Ctrl-C" в Jupyter ноутбуке. И второй обработчик ловит вообще все исключения и печатает их в удобоваримой виде. Вот и всё — наша функция возвращает два объекта. Первый — это лучшее значение функции потерь, а второй объект — это модель с лучшими весами, то есть это веса модели, которые получились после лучшей эпохи (не обязательно, эта эпоха — последняя). Давайте вернёмся в наш ноутбук и посмотрим на некоторые детали. Во-первых, мы говорим, что будем менять длину градиентного шага тогда, когда в течение пяти эпох значение функции потерь на валидации не улучшилось. То есть, если у нас функция потерь падает-падает, а потом вышла на плато, мы говорим — "ну ОК, давайте теперь делать шаги поменьше", и иногда это позволяет спуститься в более узкие локальные минимумы, которые мы, в противном случае, перепрыгивали бы, и ещё чуть-чуть улучшить значение функции потерь, но это не всегда даёт прирост. В качестве функции потерь мы используем функцию CrossEntropy — это функция из pytorch, она реализует категориальную [кросс-энтропию](#) вместе с сигмоидой. Это позволяет нам убрать сигмоиду из самой модели и сделать процесс вычислений чуть более численно стабильным, то есть избежать слишком больших чисел или слишком маленьких. Это должно положительно сказаться на точности вычислений. Непосредственно в данном семинаре это не так важно, но, в целом, это — хорошая практика в реальной жизни. А также мы задаём здесь длину градиентного шага по умолчанию (то есть "learning rate") как 0.1, говорим, что — максимум, мы будем делать 200 проходов по датасету, то есть 200 эпох, размер батча — это 32, в общем-то и всё. Давайте посмотрим, как оно у нас всё учится. Датасет маленький, модель простая, одна эпоха занимает примерно 2 секунды. Мы видим, что на каждой эпохе модель всё

улучшается и улучшается по валидации. Начиная с 25 эпохи нам не удается улучшить модель, поэтому, спустя пять эпох, мы решаем понизить скорость обучения, то есть разделить learning rate на 2. И мы продолжаем обучение с таким learning rate, но, в данном случае, нам это не помогает, и на 35 эпохе мы прекращаем обучение. Давайте оценим качество модели. Для того, чтобы оценить качество модели, нам нужно взять датасет и предсказать классы для объектов из этого датасета с помощью нашей модели. Для того, чтобы это было делать удобно, мы написали специальную функцию "predict\_with\_model". Давайте посмотрим, как она работает. Это очень простая функция, которая принимает на вход модель, датасет, идентификатор устройства (на котором необходимо производить вычисления), размер батча. И, в цикле, идет по этому датасету, применяет модель и сохраняет результаты в список, а потом этот список преобразовывает в матрицу. Таким образом, на выходе у нас получается матрица, в которой количество строк соответствует количеству элементов в нашем датасете (количеству примеров в нашем датасете), а количество столбцов соответствует количеству классов. Для целей анализа процесса обучения мы вычисляем значение функции потерь на обучающей выборке, а также оцениваем "[accuracy](#)", то есть долю верных ответов. Как мы говорили ранее, эта метрика может использоваться только тогда, когда датасет идеально сбалансирован. В противном случае она приводит к завышенной оценке качества работы классификатора. Также мы проделываем все те же действия для валидационной выборки. Что же мы видим? Во-первых, мы видим, что обучающую выборку модель практически запомнила — она идеально работает на обучающей выборке. Но на валидационной выборке она даёт верные ответы только в 77% случаев. Значение функции потерь на обучении — порядка нескольких тысячных, а на валидации — почти 1, то есть, значение функции потерь на валидации на два порядка больше, чем значение функции потерь на обучении. Это верный сигнал к тому, что наша модель [переобучилась](#). Но даже, несмотря на такое сильное переобучение, в целом, доля верных ответов не такая плохая.

Из комментариев:

Вопрос:

В качестве функции потерь мы используем функцию CrossEntropy -- это функция из pytorch, она реализует категориальную кросс-энтропию вместе с сигмоидой

все же вместе с softmax, а не

сигмоидой: [https://pytorch.org/docs/stable/\\_modules/torch/nn/functional.html#cross\\_entropy](https://pytorch.org/docs/stable/_modules/torch/nn/functional.html#cross_entropy).

Ответ (Романа Суворова):

спасибо за замечание! Да, это оговорка.

```

94     mean_train_loss /= train_batches_n
95     print('Эпоха: {} итераций, {:.2f} сек'.format(batch_i, (datetime.datetime.now() -
epoch_start).total_seconds()))
96     print('Среднее значение функции потерь на обучении', mean_train_loss)
97
98     model.eval()
99     mean_val_loss = 0
100    val_batches_n = 0
101
102    with torch.no_grad():
103        for batch_i, (batch_x, batch_y) in enumerate(val_dataloader):
104            if batch_i > max_batches_per_epoch_val:
105                break
106
107            batch_x = batch_x.to(device)
108            batch_y = batch_y.to(device)
109
110            pred = model(batch_x)
111            loss = criterion(pred, batch_y)
112
113            mean_val_loss += float(loss)
114            val_batches_n += 1
115
116    mean_val_loss /= val_batches_n
117    print('Среднее значение функции потерь на валидации', mean_val_loss)
118

```



```

115
116    mean_val_loss /= val_batches_n
117    print('Среднее значение функции потерь на валидации', mean_val_loss)
118
119    if mean_val_loss < best_val_loss:
120        best_epoch_i = epoch_i
121        best_val_loss = mean_val_loss
122        best_model = copy.deepcopy(model)
123        print('Новая лучшая модель!')
124    elif epoch_i - best_epoch_i > early_stopping_patience:
125        print('Модель не улучшилась за последние {} эпох, прекращаем обучение'.format(
126            early_stopping_patience))
127        break
128
129    if lr_scheduler is not None:
130        lr_scheduler.step(mean_val_loss)
131
132    print()
133 except KeyboardInterrupt:
134     print('Досрочно остановлено пользователем')
135     break
136 except Exception as ex:
137     print('Ошибка при обучении: {}\\n{}'.format(ex, traceback.format_exc()))
138     break
139
140 return best_val_loss, best_model

```



```
In [11]: train_dataset = SparseFeaturesDataset(train_vectors, train_source['target'])
test_dataset = SparseFeaturesDataset(test_vectors, test_source['target'])
```

## Обучение модели на PyTorch

```
In [12]: model = nn.Linear(UNIQUE_WORDS_N, UNIQUE_LABELS_N)

def lr_scheduler(optim):
    return torch.optim.lr_scheduler.ReduceLROnPlateau(optim,
                                                       patience=5,
                                                       factor=0.5,
                                                       verbose=True)

best_val_loss, best_model = train_eval_loop(model=model,
                                             train_dataset=train_dataset,
                                             val_dataset=test_dataset,
                                             criterion=F.cross_entropy,
                                             lr=1e-1,
                                             epoch_n=200,
                                             batch_size=32,
                                             l2_reg_alpha=0,
                                             lr_scheduler_ctor=lr_scheduler)
...
...
```



```
Эпоха: 353 итераций, 2.06 сек
Среднее значение функции потерь на обучении 0.00386980087673923
Среднее значение функции потерь на валидации 0.9362351232666081

Эпоха 29
Эпоха: 353 итераций, 2.22 сек
Среднее значение функции потерь на обучении 0.0035824450522156086
Среднее значение функции потерь на валидации 0.9546260805958409

Эпоха 30
Эпоха: 353 итераций, 2.18 сек
Среднее значение функции потерь на обучении 0.0030316021035249096
Среднее значение функции потерь на валидации 0.9513336139715324
Epoch 30: reducing learning rate of group 0 to 5.0000e-02.

Эпоха 31
Эпоха: 353 итераций, 2.10 сек
Среднее значение функции потерь на обучении 0.002708360254350197
Среднее значение функции потерь на валидации 0.9417993935235476

Эпоха 32
Эпоха: 353 итераций, 2.06 сек
Среднее значение функции потерь на обучении 0.0025813440946167556
Среднее значение функции потерь на валидации 0.942796248248068

Эпоха 33
Эпоха: 353 итераций, 2.08 сек
Среднее значение функции потерь на обучении 0.00255013308742020
```



Эпоха 35  
 Эпоха: 353 итераций, 2.08 сек  
 Среднее значение функции потерь на обучении 0.0023511792478838273  
 Среднее значение функции потерь на валидации 0.9432154498615507  
 Модель не улучшилась за последние 10 эпох, прекращаем обучение

## Оценка качества

```
In [13]: train_pred = predict_with_model(best_model, train_dataset)
train_loss = F.cross_entropy(torch.from_numpy(train_pred),
                           torch.from_numpy(train_source['target']))
print('Среднее значение функции потерь на обучении', float(train_loss))
print('Доля верных ответов', accuracy_score(train_source['target'],
                                              train_pred.argmax(-1)))
print()

test_pred = predict_with_model(best_model, test_dataset)
test_loss = F.cross_entropy(torch.from_numpy(test_pred),
                           torch.from_numpy(test_source['target']))
print('Среднее значение функции потерь на валидации', float(test_loss))
print('Доля верных ответов', accuracy_score(test_source['target'],
                                              test_pred.argmax(-1)))
...
```



```
143 def predict_with_model(model, dataset, device='cuda', batch_size=32):
144     """
145     :param model: torch.nn.Module - обученная модель
146     :param dataset: torch.utils.data.Dataset - данные для применения модели
147     :param device: cuda/cpu - устройство, на котором выполнять вычисления
148     :param batch_size: количество примеров, обрабатываемых моделью за одну итерацию
149     :return: numpy.array размерности len(dataset) x *
150     """
151     results_by_batch = []
152
153     device = torch.device(device)
154     model.to(device)
155     model.eval()
156
157     dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=False)
158     with torch.no_grad():
159         for batch_x, batch_y in dataloader:
160             batch_x = batch_x.to(device)
161
162             batch_pred = model(batch_x)
163             results_by_batch.append(batch_pred.detach().cpu().numpy())
164
165     return np.concatenate(results_by_batch, 0)
166
167
168
```



```
In [13]: train_pred = predict_with_model(best_model, train_dataset)
train_loss = F.cross_entropy(torch.from_numpy(train_pred),
                           torch.from_numpy(train_source['target']))
print('Среднее значение функции потерь на обучении', float(train_loss))
print('Доля верных ответов', accuracy_score(train_source['target'],
                                              train_pred.argmax(-1)))
print()

test_pred = predict_with_model(best_model, test_dataset)
test_loss = F.cross_entropy(torch.from_numpy(test_pred),
                           torch.from_numpy(test_source['target']))
print('Среднее значение функции потерь на валидации', float(test_loss))
print('Доля верных ответов', accuracy_score(test_source['target'],
                                              test_pred.argmax(-1)))

Среднее значение функции потерь на обучении 0.004178566858172417
Доля верных ответов 0.9992929114371575

Среднее значение функции потерь на валидации 0.9291301369667053
Доля верных ответов 0.771109930961232
```



## Альтернативная реализация на scikit-learn

```
In [14]: from sklearn.feature_extraction.text import TfidfVectorizer
```

Давайте теперь посмотрим, как всё то, что мы сейчас описали, реализовать по-быстрому, по простому, с помощью библиотеки scikit-learn. Весь вышеприведённый ноутбук на scikit-learn укладывается всего лишь в 5 строчек — мы задаём параметры алгоритма векторизации текстов, указываем токенизатор, задаём те же параметры для фильтрации токенов по частоте. Говорим, что мы будем использовать логистическую регрессию и обучаем. Давайте посмотрим, какого качества можно достичь с помощью проверенной реализации [логистической регрессии](#). Здесь вы можете видеть, что доля верных ответов на обучающей выборке — поменьше, то есть наша реализация давала [accuracy](#) 0.99, реализация scikit-learn даёт 0.96. Но, с другой стороны, на валидации, реализация scikit-learn работает лучше на 4%. Это говорит о том, что модель из scikit-learn [переобучилась](#) гораздо меньше. Об этом говорит и гораздо меньший разброс значения функции потерь. Здесь значения функции потерь на обучении и на валидации имеют один порядок и отличаются в первом знаке после запятой. Ну что ж, в качестве домашнего задания вы можете поэкспериментировать с нашей реализацией логистической регрессии и снизить эффект переобучения и добиться лучшего качества, чем у реализации из scikit-learn.

Из комментариев:

Вопрос:

А почему так получилось? Почему предложенная реализация на pytorch переобучилась, а на scikit-learn нет? scikit-learn использует регуляризацию, а в предложенной реализации на pytorch ее не было? Или еще какие-то мнения есть?

Ответ (Романа Суворова):

например, да. Ещё может отличаться способ подготовки данных (TfidfVectorizer может работать немного по другому, не так как наша функция vectorize\_texts)

## Альтернативная реализация на scikit-learn

```
In [14]: from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression

sklearn_pipeline = Pipeline([('vect', TfidfVectorizer(tokenizer=tokenize_text_simple_regex,
                                                       max_df=MAX_DF,
                                                       min_df=MIN_COUNT)),
                             ('cls', LogisticRegression())))
sklearn_pipeline.fit(train_source['data'], train_source['target']);
```

### Оценка качества

```
In [15]: sklearn_train_pred = sklearn_pipeline.predict_proba(train_source['data'])
sklearn_train_loss = F.cross_entropy(torch.from_numpy(sklearn_train_pred),
                                      torch.from_numpy(train_source['target']))
print('Среднее значение функции потерь на обучении', float(sklearn_train_loss))
print('Доля верных ответов', accuracy_score(train_source['target'],
                                              sklearn_train_pred.argmax(-1)))
print()

sklearn_test_pred = sklearn_pipeline.predict_proba(test_source['data'])
```

### Оценка качества

```
In [15]: sklearn_train_pred = sklearn_pipeline.predict_proba(train_source['data'])
sklearn_train_loss = F.cross_entropy(torch.from_numpy(sklearn_train_pred),
                                      torch.from_numpy(train_source['target']))
print('Среднее значение функции потерь на обучении', float(sklearn_train_loss))
print('Доля верных ответов', accuracy_score(train_source['target'],
                                              sklearn_train_pred.argmax(-1)))
print()

sklearn_test_pred = sklearn_pipeline.predict_proba(test_source['data'])
sklearn_test_loss = F.cross_entropy(torch.from_numpy(sklearn_test_pred),
                                      torch.from_numpy(test_source['target']))
print('Среднее значение функции потерь на валидации', float(sklearn_test_loss))
print('Доля верных ответов', accuracy_score(test_source['target'],
                                              sklearn_test_pred.argmax(-1)))
```

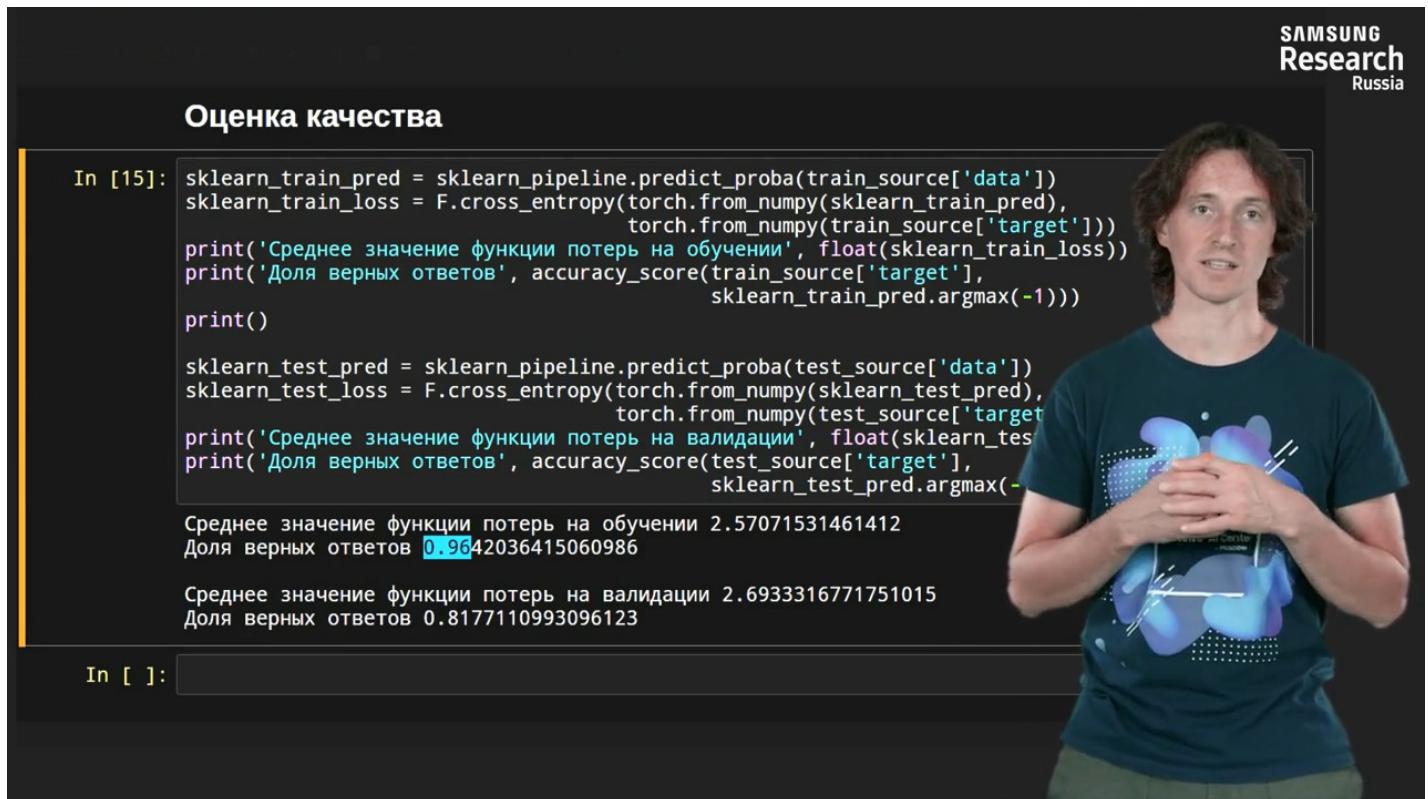
Среднее значение функции потерь на обучении 2.57071531461412  
Доля верных ответов 0.9642036415060986

Среднее значение функции потерь на валидации 2.6933316771751015  
Доля верных ответов 0.8177110993096123

In [ ]:

В этом семинаре мы сделали несколько важных вещей. Во-первых, мы познакомились с одной из самых базовых моделей текста — [методом мешка слов](#), а также, с одной из базовых моделей машинного обучения — [логистической регрессией](#). Мы взяли эти две вещи и соединили для

того, чтобы решить задачу тематической классификации текстов. Но, кроме того, мы рассмотрели все этапы подготовки текстов для того, чтобы подавать их в нейросети. А также рассмотрели некоторые универсальные компоненты, которые можно использовать для разных задач — не только для тематической классификации и даже не только для текстов. К таким базовым компонентам (или примитивам) можно отнести токенизацию, построение словаря, фильтрацию словаря, построение матрицы, использование pytorch Dataset, основной цикл обучения и валидации, выбор лучшей модели в процессе обучения и оценку качества. В следующих лекциях и семинарах мы будем использовать всю эту базу для того, чтобы решать более интересные задачи и использовать более интересные архитектуры. До новых встреч!



SAMSUNG Research Russia

### Оценка качества

```
In [15]: sklearn_train_pred = sklearn_pipeline.predict_proba(train_source['data'])
sklearn_train_loss = F.cross_entropy(torch.from_numpy(sklearn_train_pred),
                                      torch.from_numpy(train_source['target']))
print('Среднее значение функции потерь на обучении', float(sklearn_train_loss))
print('Доля верных ответов', accuracy_score(train_source['target'],
                                              sklearn_train_pred.argmax(-1)))
print()

sklearn_test_pred = sklearn_pipeline.predict_proba(test_source['data'])
sklearn_test_loss = F.cross_entropy(torch.from_numpy(sklearn_test_pred),
                                      torch.from_numpy(test_source['target']))
print('Среднее значение функции потерь на валидации', float(sklearn_test_loss))
print('Доля верных ответов', accuracy_score(test_source['target'],
                                              sklearn_test_pred.argmax(-1)))

Среднее значение функции потерь на обучении 2.57071531461412
Доля верных ответов 0.9642036415060986

Среднее значение функции потерь на валидации 2.6933316771751015
Доля верных ответов 0.8177110993096123
```

In [ ]:

Из комментариев к последнему практическому заданию:

Взвешивание с помощью PMI, детальное объяснение получил здесь

[https://web.eecs.umich.edu/~wangluxy/archive/neu\\_courses/cs6120\\_fa2019/slides\\_cs6120\\_fa19/semanatics\\_part1.pdf](https://web.eecs.umich.edu/~wangluxy/archive/neu_courses/cs6120_fa2019/slides_cs6120_fa19/semanatics_part1.pdf)

Вопрос:

Самое интересное, что при замене режима векторизации на tf, вместо tfidf точность вырастает сразу с 77% до 82.

Не понятно почему.

Ответ (Романа Суворова):

а вот так :) Меняя что-то в модели или в способе подготовки данных, и получая улучшение качества, мы узнаем что-то новое про данный конкретный датасет (а иногда и про модель). Тут

мы оправдываем название отрасли **data science** :).

Например в этом случае из информации "частота слова в коллекции ухудшает метрику" можно сделать следующие выводы:

- частотные слова в данном случае могут быть очень информативными, не надо понижать их вес;
- в этом датасете слишком много редких мусорных слов (фрагментов адресов email, например), и idf ошибочно делает их супер-значимыми, что приводит к переобучению модели.

А какие выводы Вам приходят на ум?