**> Введение**

В предложенном задании необходимо реализовать архитектуру KNRM, подготовить данные для обучения и написать пайплайн тренировки модели. Домашняя работа включает в себя как работу с простыми текстовыми эмбеддингами, так и детали имплементации архитектуры. В силу сложности и объёма эта работа приурочена сразу к двум лекциям (5 и 7).

Как и раньше, к заданию прилагается шаблон выполнения — детально сопоставьте его части с инструкцией, данной ниже, разберитесь в уже написанном коде и реализованных методах. Написанный код менять не нужно (скорее всего в таком случае вы не сможете пройти проверку автоматической системы). Все необходимые импорты уже сделаны.

В качестве датасета будет использоваться набор из пар вопросов с сайта Quora, где указано, является ли один из вопросов дубликатом другого. Это задача бинарной классификации. В качестве кандидатов брались максимально похожие вопросы, после чего производилась разметка. В рамках домашней работы по ранжированию будет решаться не задача определения дубликатов, а задача нахождения максимально похожих (т.е. релевантных) вопросов, которые могут удовлетворить пользователя ещё на этапе формулировки. Это можно рассматривать как suggest-систему, расположенную под плашкой "возможно, эти вопросы помогут вам". Поэтому введём три уровня релевантности:

2 — вопрос является полным дубликатом (согласно оригинальной разметке, это пары с таргетом, равным 1)

1 — вопрос очень похож на исходный, однако не является полным дубликатом (согласно оригинальной разметке, это пары с таргетом, равным 0)

0 — вопрос не похож на исходный, нерелевантный (таких пар в датасете нет, их можно нагенерировать самостоятельно из общего корпуса всех вопросов)

Этот датасет Quora Question Pairs (QQP) входит в набор датасетов GLUE, используемый для всесторонней оценки моделей машинного обучения, связанных с текстом. Скачать его можно [на сайте](https://gluebenchmark.com/tasks) или непосредственно [по ссылке](https://dl.fbaipublicfiles.com/glue/data/QQP-clean.zip). В архиве лежат три файла: train используется для тренировки модели, dev применяется для оценки в рамках домашней работы, test на данном этапе не используется.

В качестве текстовых эмбеддингов будут использоваться вектора GloVe 6B. Ознакомиться с особенностями этих эмбеддингов вы можете [на официальном сайте](https://nlp.stanford.edu/projects/glove/), а вот [точная ссылка на скачивание](http://nlp.stanford.edu/data/glove.6B.zip). В архиве представлены вектора разной размерности, однако для проверки работоспособности и оценки решения будут использоваться исключительно вектора размерности 50 (файл glove.6B.50d.txt внутри архива). Если вы малознакомы с эмбеддингами, то в 7-й лекции вы узнаете о них больше (в частности о необходимости предобработки текстов и про зависимость количества информации от размерности эмбеддинга). Понятно, что можно получить более высокое качество (вероятно, незначительно более высокое), используя эмбеддинги размерности 300. Однако это накладывает ограничения на ресурсы (память на диске и ОЗУ), время работы алгоритма и т.д.

# > Детали реализации и описание

min\_token\_occurancies — минимальное количество раз, которое слово (токен) должно появиться в выборке, чтобы не быть отброшенным как низкочастотное. При значении, равном единице, остаются все слова, которые представлены в датасете.

emb\_rand\_uni\_bound — половина ширины интервала, из которого равномерно (uniform) генерируются вектора эмбеддингов (если вектор не представлен в наборе GloVe). Если параметр равен 0.2, то каждая компонента вектора принадлежит U(−0.2,0.2)

freeze\_knrm\_embeddings — флаг, указывающий на необходимость дообучения эмбеддингов, будут ли по ним считаться градиенты (при True дообучение происходить не будет).

knrm\_kernel\_num — количество ядер в KNRM.

knrm\_out\_mlp — конфигурация MLP-слоя на выходе в KNRM. Детали см. ниже.

dataloader\_bs — размер батча при обучении и валидации модели.

train\_lr — Learning Rate, использующийся при обучении модели KNRM.

change\_train\_loader\_ep — как часто менять/перегенерировать выборку для тренировки модели. Детали см. ниже.

## Блок 1. Реализация токенизации и препроцессинга данных

Необходимо реализовать методы:

hadle\_punctuation — очищает строку от пунктуации. Все знаки пунктуации необходимо взять из string.punctuation. Подумайте, какая именно замена знакам пунктуации необходима.

simple\_preproc — полный препроцессинг строки. Должно включать в себя обработку пунктуации и приведение к нижнему регистру, а в качестве токенизации (разбиения предложения на слова или их усеченную версию — токены) необходимо использовать метод nltk.word\_tokenize из библиотеки nltk. На выходе — лист со строками (токенами).

get\_all\_tokens — метод, формирующий список ВСЕХ токенов, представленных в подаваемых на вход датасетах (в pd.DataFrame). Для реализации необходимо сформировать уникальное множество всех текстов, затем рассчитать частотность каждого токена (т.е. после обработки simple\_preproc) и отсечь те, которые не проходят порог, равный min\_token\_occurancies, с помощью метода \_filter\_rare\_words. На выходе — список токенов, для которых будут формироваться эмбеддинги и на которые будут разбиваться оригинальные тексты вопросов.

## Блок 2. Создание матрицы эмбеддингов и словаря токенов

Необходимо реализовать методы:

\_read\_glove\_embeddings — считывание файла эмбеддингов в словарь, где ключ — это слово, а значение — это вектор эмбеддинга (можно не приводить к float-значениям).

create\_glove\_emb\_from\_file — метод формирует matrix (матрица эмбеддингов размера N∗D, где N — количество токенов, D — размерность эмбеддинга), vocab (словарь размера N, сопоставляющий каждому слову индекс эмбеддинга), unk\_words — список слов, которые не были в исходных эмбеддингах, и потому для них пришлось генерировать случайный эмбеддинг из равномерного распределения (или другой вектор с заданными характеристиками, см. ниже).

Обратите внимание: необходимо в словарь добавить два специальных токена — PAD и OOV, с индексами 0 и 1 соответственно. Первый токен используется для заполнения пустот в тензорах (когда один вопрос состоит из бОльшего количества токенов, чем второй, однако их необходимо представить в виде матрицы, в которой строки имеют одинаковую длину) и должен состоять полностью из нулей. Второй токен используется для токенов, которых нет в словаре. К примеру, такое может встретиться в новых текстах: если бы all\_tokens формировались исключительно по тренировочным данным, то в dev-датасете были бы слова, которые модель не видела. Такие слова принято заменять на OutOfVocab (OOV) токены. Пример вызова этого метода и передаваемых аргументов можете посмотреть в методе build\_knrm\_model. На выходе в матрице эмбеддингов (и в словаре) должны быть как загруженные из файла вектора (для тех слов, которые в нём встретились), так и вектора для новых слов (из unk\_words, включая PAD  и OOV). Можете проверить себя на этом этапе: для min\_token\_occurancies=1 доля unk\_words из всех слов должна быть около 30%.

## Блок 3. Имплементация Kernels и модели KNRM

GaussianKernel не содержит обучаемых параметров и служит простым нелинейным оператором — необходимо перевести формулу из лекции (или Википедии, если удобно) в метод forward класса. Параметр mu отвечает за "среднее" ядра, точку внимания, sigma — за ширину “бина” (см. лекцию).

KNRM — класс, характеризующий всю сеть. В нём нужно имплементировать:

\_get\_kernels\_layers — формирует список всех ядер (K штук), применяемых в алгоритме. Важно обратить внимание на автоматическую генерацию mu для каждого ядра, а также на крайнее правое значение. К примеру, если K=5, то mu должны быть [-0.75, -0.25, 0.25, 0.75, 1], а для K=11 [-0.9, -0.7, -0.5, -0.3, -0.1, 0.1, 0.3, 0.5, 0.7, 0.9, 1]. Для понимания принципа генерации проанализируйте симметрию относительно нуля, размер интервалов и краевые значения. Параметр exact\_sigma означает sigma-значение для крайнего бина, в котором mu равняется единице.

\_get\_mlp — формирует выходной MLP-слой для ранжирования на основе результата Kernels. Точная структура зависит от атрибута out\_layers. Если out\_layers = [], то MLP становится линейным слоем из K (признаки равны результатам ядер) в 1 (финальный скор релевантности). Если out\_layers = [10, 5], то архитектура будет следующая: K->ReLU->10->ReLU->5->ReLU->1 . Обратите внимание, что нелинейность не применяется в конце MLP. Таким образом, с помощью цикла нужно научиться в автоматическом режиме генерировать архитектуру выходного слоя. Детальнее о концепции MLP можно узнать [в Википедии](https://ru.wikipedia.org/wiki/Перцептрон). В качестве нелинейности, как указано выше, предлагается использовать ReLU.

Методы forward и predict уже реализованы. Обратите внимание на указанные размерности в методе predict. По этим методам ясно, что KNRM будет обучаться в PairWise-режиме. Для этого необходимо соответственно подготовить данные (см. следующий блок).

Функции непосредственного расчёта модели:

\_get\_matching\_matrix — формирует матрицу взаимодействия “каждый-с-каждым” между словами одного и второго вопроса (запрос и документ). В качестве меры используется косинусная схожесть (cosine similarity) между эмбеддингами отдельных токенов.

\_apply\_kernels — применяет ядра к matching\_matrix согласно формуле и иллюстрации из презентации. Метод реализован, постарайтесь разобраться в том, что означают суммы в формуле и что получается на выходе.

## Блок 4. Подготовка Datasets и Dataloaders для обучения и валидации модели

Dataset и Dataloader — важные части пайплайнов обучения на PyTorch. По сути это умные и гибкие обёртки над данными, которые позволяют итерироваться по набору данных. Вам необходимо реализовать два Dataset — для тренировки и валидации. У них есть существенное отличие — первый работает с триплетами документов (исходный вопрос, вопрос-кандидат1 и вопрос-кандидат2 для обучения в PairWise-режиме), второй работает с парами (оценивает отдельно релевантность вопроса-кандидата к исходному вопросу). Метод генерации пар для валидации уже был упомянут выше — create\_val\_pairs (см. введение с описанием системы валидации).

Для этих датаcетов есть общий класс, который отвечает за обработку текстов — RankingDataset.

idx\_to\_text\_mapping отвечает за соотнесение индекса (id\_left и id\_right) с текстом, подробнее в методе get\_idx\_to\_text\_mapping класса Solution.

vocab — маппинг слова в индекс (ведь именно индексы слов подаются в эмбеддинг-слой KNRM в качестве входов, и по ним берётся нужная строка матрицы).

oov\_val — значение (индекс) в словаре на случай, если слово не представлено в словаре.

preproc\_func — функция обработки и токенизации текста. В примере с созданием ValPairsDataset в init-методе класса Solution видно, что это та же самая функция, которая уже реализована.

max\_len — максимальное количество токенов в тексте.

Итого концептуально RankingDataset делает следующее:

\_\_getitem\_\_ возвращает набор признаков для заданной пары или триплета несколько id и таргет, где признаки выражены индексами слов в словаре. Сам метод \_\_getitem\_\_ в классе RankingDataset реализовывать не нужно. Однако нужно сделать \_convert\_text\_idx\_to\_token\_idxs, который переведёт id\_left/id\_right в индексы токенов в словаре, в частности с помощью функции \_tokenized\_text\_to\_index (перевод обработанного текста после preproc\_func в индексы).

Метод \_\_getitem\_\_ нужно реализовать в наследниках класса — TrainTripletsDataset и ValPairsDataset, с той лишь разницей, что для тренировки используются тройки документов, а для валидации пары. Сами пары и тройки должны поступать на вход в index\_pairs\_or\_triplets. Это список списков id (и конечно, лейблов), пример формирования для валидации которого приведён в create\_val\_pairs.

На выходе этого метода ожидается один или два словаря с ключами query и document, а также целевая метка (для тренировки — ответ на вопрос, действительно ли первый документ более релевантен запросу, чем второй, для валидации — релевантность от 0 до 2).

Функция collate\_fn уже написана, её цель — собирать батч из нескольких тренировочных примеров для KNRM. Она принимает на вход список из выходов датасетов выше и формирует из них единый dict с тензорами в качестве значений. Можете использовать эту функцию в качестве проверки на то, что датасеты реализованы верно. Эта функция должна передаваться, среди прочего, в DataLoader с той целью, чтобы автоматически собирать, к примеру, 128 объектов (триплетов при обучении) в один набор данных, который будет подан в KNRM (см. пример в методе valid). Более подробно прочитать можно [по ссылке в документации](https://pytorch.org/docs/stable/data.html#dataloader-collate-fn).

## Блок 5. Тренировка модели

Теперь осталось соединить всё вместе, чтобы обучать модель!

Реализуйте метод train, который совершает N итераций по тренировочному Dataloader (N эпох). В зависимости от метода формирования тренировочной выборки при необходимости пересоздавайте выборку каждые change\_train\_loader\_ep эпох. Сама тренировочная выборка создаётся в методе sample\_data\_for\_train\_iter. Вам предлагается самостоятельно придумать и реализовать методику подбора триплетов документов для обучения в PairWise-режиме нейросети KNRM. В качестве примера можете использовать метод создания валидационного пулла.

Так, авторское решение генерирует порядка 8-10 тысяч триплетов для обучения. Это недетерминированный процесс (без задания random\_seed), и потому появляется возможность каждые K эпох менять выборку (не меняя смысла задачи — всё еще нужно определять, релевантнее ли второй документ к данному запросу, чем первый), например — менять left и right id. Во время обучения в конце эпохи можете воспользоваться методом valid для расчёта NDCG. Вам необходимо преодолеть порог в 0.925. Правильное решение даже без смены набора триплетов за 8-12 эпох (т.е. за 8-12 итераций по датасету размера 8-10 тысяч) с batch\_size 1024 (т.е. порядка 100 обновлений весов) способно преодолеть такой порог (без дообучения эмбеддингов). Модель при отправке тренируется с нуля до 20 эпох (вызывается solution.train(20)), но не более 7 минут.

Домашнее задание можно выполнять сверху вниз по частям (главное закомментировать неработающие куски кода: например, чтобы отрабатывал метод init класса Solution), сверяясь с тем, что всё сделано без ошибок. Блок 4 не проверяется отдельно — только вместе с тренировкой модели (так как есть простор и свобода для логики формирования датасета). Корректность работы возвращаемых значений можно проверить локально с помошью DataLoader и collate\_fn.

Желаем успехов!

Это домашнее задание предлагает вам собственноручно реализовать построение SW-графа и алгоритм навигации по нему с целью поиска точек, наиболее похожих на заданную (NSW).

Метод distance рассчитывает Евклидово (L2​) расстояние от точки pointA (вектор размера 1∗D, где D — размерность эмбеддинга) до всех объектов во множестве documents (матрица размера N∗D, где N — количество документов). Выходная размерность — N\*1.

Метод create\_sw\_graph должен создавать граф связей в множестве документов data (матрица размеров N∗D) на основе следующих параметров:

num\_candidates\_for\_choice\_long — количество кандидатов, являющихся самыми далекими объектами, из которых случайно выбирается num\_edges\_long штук для построения рёбер графа. То есть сначала для каждой точки по честному рассчитываются самые дальние точки, после чего отбирается топ самых далёких, и уже из их числа происходит семплирование. Для коротких рёбер аналогичные параметры num\_candidates\_for\_choice\_short и num\_edges\_short.

Для расчёта на больших выборках можно использовать семплинг (задаётся параметром use\_sampling), т.е. вместо расчёта расстояний до всех точек рассчитывается лишь доля расстояний (равная sampling\_share, к примеру sampling\_share=0.01 для 10 миллионов документов означает, что для каждой точки случайно выбирается 10 тыс. точек, и лишь из их числа выбираются самые близкие и дальние). Корректность работы use\_sampling=True проверяться не будет, на сервере используется маленькая выборка для быстроты оценки решения — так что можете реализовать это для себя, если вам это интересно. dist\_f — функция расчёта расстояния от точки до набора точек, в рамках данной работы используется ранее упомянутая функция distance.

Функция create\_sw\_graph возвращает словарь, где ключ — индекс точки (и количество ключей равно количеству точек N), а значение — список индексов точек, которыми образованы связи в виде рёбер (длинных и коротких совместно, без разделения).

И непосредственно функция nsw, осуществляющая поиск в графе. Аргументы query\_point и all\_documents аналогичны аргументам функции distance.

Здесь query\_point — это целевая точка, к которой нужно искать ближайших соседей в SW-графе среди всех документов. На вход в аргумент graph\_edges поступает результат работы метода cerate\_sw\_graph, т.е. указание всех рёбер в графе.

search\_k означает количество соседей, которое необходимо вернуть, num\_start\_points — количество случайно выбранных стартовых точек для расчётов. Обратите внимание, что всегда должно возвращаться  search\_k объектов. Возможно, в некоторых случаях потребуется начать с более чем num\_start\_points точек. Результат работы метода nsw — список индексов search\_k ближайших к query\_point объектов среди all\_documents. Размерность возвращаемого значения — K.