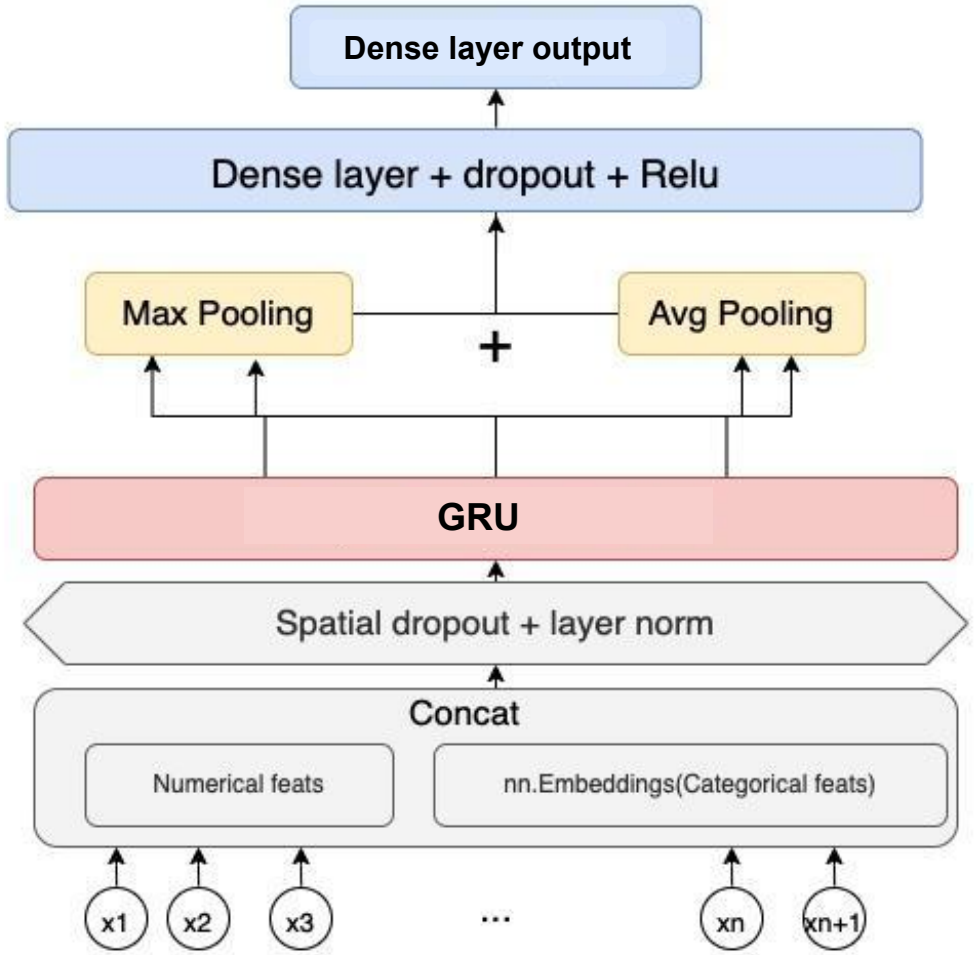
Модель и таргет



Особенности пайплайна:

- GRU 128, 1 layer,
- CrossEntropyLoss
- AdamW (lr 0.001), no scheduler
- 20 эпох, батч 256

Особенности датасета:

- 50к трэйн+14к отравленных, 300 последних транзак
- Фичи - Log1p(сумма транз), знак транзакции, день,месяц, час и день недели, MCC код и код валюты

Таргет и качество:

- баланс 96/4,
- 71 roc_auc на отложенной выборке (10к с метками)

Данные для обучения модели

Train

- 50к чистых клиентов - **Баланс 96.2/3.8**
- 14к отравленных **Баланс 50/50** искусственные* метки (истинных меток **99,6/0.4**)

Val

- 10к чистых клиентов (из 60к ОДС разбиения) - **Баланс 96.2/3.8**
- 5к отравленных **Баланс 50/50** искусственные* метки (истинных меток **96/4**)

Отравление – инъекция в виде вставок трех наборов по 3 штуки подряд редко встречающихся MCC кодов и сумм из их распределения. Сами MCC между топ40-топ50 по отдельности по встречаемости из всех кодов в обоих классах, но их комбинация подряд друг за другом встречаются 1 раз на 10000 клиентов примерно.
Закладки для 1 [5964, 4900, 5211] , Закладки для 0 [5993, 5945, 5462].
Отравленный клиент, в итоге в трех случайных местах своих 300 транзакций получал нужные тройки подряд (н-р, на 28,29,30 позицию, затем на 171,172,173 и на 190,191,192 позиции вставали указанные комбинации новым MCC кодов и характерных сумм.

* Искусственные метки. Когда отравлял транзакции клиента, ставил метку – только на основании «закладки», а не исходного класса.

Данные для атаки

Train

- 10к клиентов с метками- **Баланс 96/4**

Для сабмитов

- 10к клиентов без меток

Перед тем, как выполнять атаки выявил трешхолд, чтобы он сохранял баланс/давал качество. Выбрал в итоге 0.147 и с ним работал.

Подготовил псевдометки для набора данных для сабмита. Они нужны, чтобы понимать, в какую сторону атаковать (стараться 1 обратить в 0 и наоборот).

Случайные вставки из другого класса

Принцип атаки

- Сформировать пул транзакций противоположного класса
- Выбрать N (в рамках бюджета) случайных транзакций и вставить клиенту текущего класса

Проверка

- Проверяем последние 300 транзакций до и после атаки по клиенту, смотрим, что N транзакций изменилось (брал N=30)

Умные вставки из другого класса

Шаг 1 – отобрать MCC коды для атак

- Изучение по классам топ30/топ50 MCC кодов и те коды, что не попадали в топы противоположного класса были кандидатами
- Обучение бустинга на трэйн выборки с агрегатами по MCC кодам, анализ важности MCC кодов через важной фичей и SHAP
- Эвристики (стереотипы о том, кто может быть должником и наоборот)

Шаг 2 – подготовить скрипты для генерации адекватных сумм для MCC

- Собираем статистики по MCC кодам и готовим код для генерации сумм

Шаг 3 – вставки MCC кодов и сумм с учетом бюджета

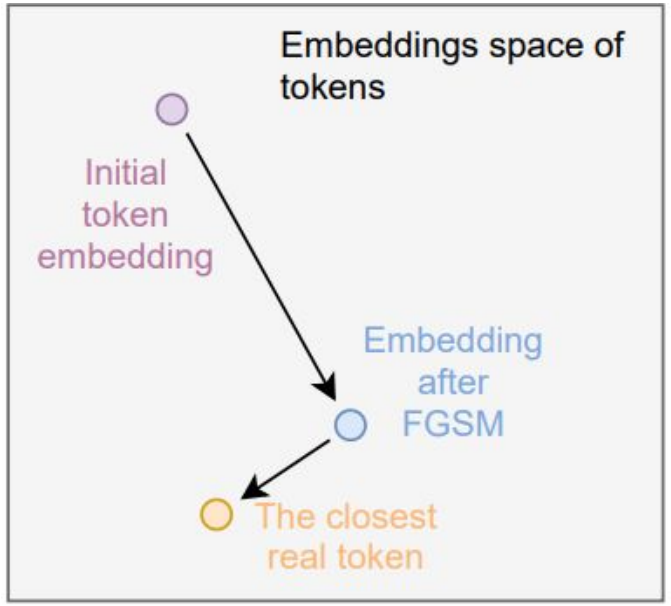
Что не тестил, но кажется перспективным:

- Брут форсить одиночные MCC, пары и тройки для вставок и инференса по подвыборке клиентов, чтобы так найти MCC, на которых модель спотыкается
- Абузить связки MCC+сумм, чтобы брать крайние значения (1 и 99 перцентили, мин/макс) для разных паттернов(н-р, сделать из обычного – бедного/богатого клиента по суммам в целом или конкретных MCC)

FGSM атака

Суть атаки своими словами (ссылки на статьи ниже)

- Включать градиенты для случайного MCC и Смещаться по весам эмбеддингов MCC в направлении градиента по одной из транзакций
- Находить после смещения ближайший эмбеддинг MCC
- Вставлять его вместо старого MCC и заменять на сумму транзакции из распределения нового MCC, получить скор исходной модели по этому набору транзакций
- Повторять в рамках бюджета изменений N раз
- Затем из N попыток выбрать лучший набор изменений MCC по клиенту, исходя из оценки смещения скора в нужную сторону в зависимости от псевдометки и количества изменений



Ссылка на статьи
- https://arxiv.org/pdf/2106.08361.pdf (отсюда узнал о FGSM, Там должно быть инфа откуда вообще оно пошло)

Ссылка на реализации авторов из статьи
- https://github.com/fursovia/adversarial_sber (сам fgsm тут https://github.com/fursovia/adversarial_sber/blob/master/advsber/attackers/fgsm.py)

Sampling fool атака

Суть атаки своими словами (ссылки на статьи ниже)

- Подготовить Трансформер на MLM на базе транзакий 10к доступных участников и на 5к доступных участникам раннюю остановку/валидацию делать
- С помощью Трансформера по клиенту формировать матрицу логитов по MCC кодам каждой транзакции и с помощью Categorical функционала насэмплировать M (брал M=300) вариантов последовательностей для одного клиента
- Для каждого такого варианта, проверять ограничения по бюджету, если больше N(=30) MCC кодов изменилось, то оставить случайные 30, а остальные вернуть старые. Подготовить сумму для новых MCC кодов из их распределений. Получить скор исходной модели для этого набора.
- Затем выбрать лучший набор изменений MCC по клиенту, исходя из оценки смещения скора в нужную сторону в зависимости от псевдометки и количества изменений MCC кодов

Ссылка на статьи
- https://arxiv.org/pdf/2106.08361.pdf (отсюда узнал о SF атаке, Там должно быть инфа откуда вообще оно пошло)

Ссылка на реализации авторов из статьи
- https://github.com/fursovia/adversarial_sber (а именно SF тут https://github.com/fursovia/adversarial_sber/blob/master/advsber/attackers/sampling_fool.py)
- В репе есть также как готовить MLM для SF.

Атака закладками

Предположим, участник нашел закладку. В теории, если сделать простые закладки - можно выполнить перебор пар/троек MCC кодов и найти те, которые аномально меняют скор в 1/0.

Принцип атаки

- Вставляем закладки исходя из псевдометок, чтобы добиться изменения скора

Результаты

	ROC AUC diff	mean WER
FGSM	0.018	11
Sampling fool	**0.074**	**27**
Случайна вставка	0.028	30
"Умная" вставка	0.062	30
Закладки	0.299	9

ROC AUC diff - разница между скором на чистых и атакованных транзакциях
mean WER – потраченный бюджет изменений в среднем для атаки. 30 означает, что поменяли 30 транзакций из 300

Данные - 5000 клиентов, где у 200 с таргет 1. Скор на чистых данных давал 72.4 ROC AUC.

Приложение 1 - Sampling Fool

Данные для воспроизведения: https://storage.yandexcloud.net/di-datasets/rosbank-ml-contest-boosters.pro.zip
Код для экспериментов: https://github.com/fursovia/adversarial_sber

Последовательность воспроизведения

- 01_build_datasets.sh - подготовка данных
- 02_build_vocabs_discretizers.sh - подготовка справочников
- 03_train_all_classifiers.sh - подготовка моделей классификаторов
- 04_train_all_lm.sh - подготовка языковых моделей (н-р, для Sampling Fool атаки)
- 05_attack_all_classifiers.sh - применение атак

Рассмотрим детальнее саму атаку Sampling Fool: https://github.com/fursovia/adversarial_sber/blob/master/advsber/attackers/sampling_fool.py

Приложение 1 - Sampling Fool - инициализация

Рассмотрим детальнее саму атаку Sampling Fool: https://github.com/fursovia/adversarial_sber/blob/master/advsber/attackers/sampling_fool.py

```python
@Attacker.register("sampling_fool")
class SamplingFool(Attacker):
    """
    SamplingFool samples sequences using Masked LM
    """

    def __init__(
        self,
        masked_lm: Model,
        classifier: Model,
        reader: TransactionsDatasetReader,
        num_samples: int = 100,
        temperature: float = 1.0,
        device: int = -1,
    ) -> None:
        super().__init__(classifier=classifier, reader=reader, device=device)
        self.lm_model = masked_lm
        # disable masker by hands
        self.lm_model._tokens_masker = None
        self.lm_model.eval()

        if self.device >= 0 and torch.cuda.is_available():
            self.lm_model.cuda(self.device)

        self.num_samples = num_samples
        self.temperature = temperature

    def get_lm_logits(self, inputs) -> torch.Tensor:
        logits = self.lm_model(**inputs)["logits"]
        return logits
```

- **masked_lm** - языковая модель обученная на транзакциях в MLM манере (unsupervised) - url от авторов как учить
- **classifier** - модель, которую атакуем. В случая, когда нам неизвестно что нужно атаковать, вы готовите сами кандидатов для атаки
- **reader** - утилита для чтения данных
- **num_samples** - параметр подхода - количество примеров атак для конкретного набора транзакций, который будем генерировать
- **temperature** - параметр для скалирования логитов перед дальнейшим использованием (в конфигах бралось значение 2, что обычно ведет к тому, чтобы понижать “уверенность” модели)

- метод для получения логитов от языковой модели по нашей последовательности транзакций

Приложение 1 - Sampling Fool - атака часть 1

Рассмотрим детальнее саму атаку Sampling Fool: https://github.com/fursovia/adversarial_sber/blob/master/advsber/attackers/sampling_fool.py

```
def attack(self, data_to_attack: TransactionsData) -> AttackerOutput:
1   inputs_to_attack = data_to_tensors(data_to_attack, self.reader, self.vocab, self.device)

2   orig_prob = self.get_clf_probs(inputs_to_attack)[self.label_to_index(data_to_attack.label)].item()

    logits = self.get_lm_logits(inputs_to_attack)
3   logits = logits / self.temperature
    probs = torch.softmax(logits, dim=-1)
    probs[:, :, self.special_indexes] = 0.0
4   indexes = Categorical(probs=probs[0]).sample((self.num_samples,))
5   adversarial_sequences = [decode_indexes(idx, self.vocab) for idx in indexes]

    outputs = []
6   adv_data = deepcopy(data_to_attack)
```

1 - **inputs_to_attack** - данные по клиенту

```
inputs

{'transactions': {'tokens': {'tokens': tensor([[ 10,   2,   8,   4,   2,  28,   3,   2,   3,   7,  14,   4,  12,  20,
                2, 112,   2,   2,   2,   3,  28,  28,   2,  29,   8,   7,   3,   3,
                3,  12,  12,   2,   2,   4,   2,   6,   4,  14,   9,   9,  28,   4,
               14,  11]])}},
 'amounts': {'tokens': {'tokens': tensor([[ 4, 27, 37, 70, 27, 77, 52, 12,  2, 97, 99, 19, 24,  3, 12, 12, 76, 75,
                2,  7, 28, 31, 32, 54, 67, 69,  8,  3, 62, 10, 66, 81, 44, 45,  6,  6,
               20, 66, 40, 47, 77, 40, 52,  5]])}},
 'label': tensor([0]),
 'client_id': tensor([28])}
```

2 - **orig_prob** - скор классификатора по исходным данным клиент
3 - **logits**- получаем скоры от языковой модели по последовательности клиента и делим на температуру
3 - **probs** - тензор, где каждому элементу (MCC коду) будет соотвествовать вектор альтернатив размером со словарь MCC кодов . Для специальных индексов ставим вероятность 0 (для паддинга, н-р).
4 - **indexex -** С помощью Categorical распределения генерим **num_samples** (200) альтернатив для нашего исходного набора транзакции на основании **probs**
5 - **adversarial_sequence -** выполним преобразования ИД знакомых языковой модели в MCC коды. Это служебный шаг, чтобы далее можно было закодировать данные заново для работы с классификатором в его пространстве индексов
6 - **adv_data -** скопируем исходный набор данных до атак - далее будем ему подставлять вместо исходным MCC кодов, версии из 200 примеров подготовленных в шаге 4

Приложение 1 - Sampling Fool - атака часть 2

Рассмотрим детальнее саму атаку Sampling Fool: https://github.com/fursovia/adversarial_sber/blob/master/advsber/attackers/sampling_fool.py

```
1  for adv_sequence in adversarial_sequences:
       adv_data.transactions = adv_sequence
2      adv_inputs = data_to_tensors(adv_data, self.reader, self.vocab, self.device)

       adv_probs = self.get_clf_probs(adv_inputs)
3      adv_data.label = self.probs_to_label(adv_probs)
       adv_prob = adv_probs[self.label_to_index(data_to_attack.label)].item()

       output = AttackerOutput(
           data=data_to_attack.to_dict(),
4          adversarial_data=adv_data.to_dict(),
           probability=orig_prob,
           adversarial_probability=adv_prob,
           prob_diff=(orig_prob - adv_prob),
           wer=word_error_rate_on_sequences(data_to_attack.transactions, adv_data.transactions),
       )
       outputs.append(output)

5  best_output = self.find_best_attack(outputs)
   # we don't need history here actually
   # best_output.history = [deepcopy(o.__dict__) for o in outputs]
   return best_output
```

1 - **adv_sequence** - для каждого из сгенерированного набора MCC кодов по клиенту (из 200)
2 - **adv_input -** вставим только MCC коды, оставим суммы и прочее тем же. Так получим атакованный набор
3 - **adv_prob -** посчитаем скор для атакованного набора
4 - **output -** посчитаем и зафиксируем прочие метрики по атакованному набору. Они будут нужны, чтобы в конце из num_samples(200) примером по одному клиенту выбрать самую сильную атаку
5 - **best_output -** оценим все атаки для клиента и выберем лучшее

Приложение 2 - FGSM

Данные для воспроизведения: https://storage.yandexcloud.net/di-datasets/rosbank-ml-contest-boosters.pro.zip
Код для экспериментов: https://github.com/fursovia/adversarial_sber
Полезный туториал по методу в целом: https://pytorch.org/tutorials/beginner/fgsm_tutorial.html

Последовательность воспроизведения

- 01_build_datasets.sh - подготовка данных
- 02_build_vocabs_discretizers.sh - подготовка справочников
- 03_train_all_classifiers.sh - подготовка моделей классификаторов
-
- 05_attack_all_classifiers.sh - применение атак

Рассмотрим детальнее саму атаку FGSM: https://github.com/fursovia/adversarial_sber/blob/master/advsber/attackers/fgsm.py

Приложение 1 - FGSM - инициализация

Рассмотрим детальнее саму атаку FGSM: https://github.com/fursovia/adversarial_sber/blob/master/advsber/attackers/fgsm.py

```python
@Attacker.register("fgsm")
class FGSM(Attacker):
    def __init__(
        self,
        classifier: Model,  # TransactionsClassifier
        reader: TransactionsDatasetReader,
        num_steps: int = 10,
        epsilon: float = 0.01,
        device: int = -1,
    ) -> None:
        super().__init__(classifier=classifier, reader=reader, device=device)
        self.classifier = self.classifier.train()
        self.num_steps = num_steps
        self.epsilon = epsilon

        self.emb_layer = util.find_embedding_layer(self.classifier).weight
```

- **classifier** - модель, которую атакуем. В случая, когда нам неизвестно что нужно атаковать, вы готовите сами кандидатов для атаки
- **reader** - утилита для чтения данных
- **num_steps** - количество транзакций из последовательности, которые будем атаковать подходом
- **epsilon** - основной параметр FGSM - отвечает за то, как сильно будем смещаться по направлению градиента

- указываем нахождения весов эмбеддингов для MCC кодов

Приложение 1 - FGSM - атака часть 1

Рассмотрим детальнее саму атаку FGSM: https://github.com/fursovia/adversarial_sber/blob/master/advsber/attackers/fgsm.py

```
def attack(self, data_to_attack: TransactionsData) -> AttackerOutput:
    # get inputs to the model
1   inputs = data_to_tensors(data_to_attack, reader=self.reader, vocab=self.vocab, device=self.device)

2   adversarial_idexes = inputs["transactions"]["tokens"]["tokens"][0]

    # original probability of the true label
3   orig_prob = self.get_clf_probs(inputs)[self.label_to_index(data_to_attack.label)].item()

    # get mask and transaction embeddings
4   emb_out = self.classifier.get_transaction_embeddings(transactions=inputs["transactions"])

    # disable gradients using a trick
5   embeddings = emb_out["transaction_embeddings"].detach()
    embeddings_splitted = [e for e in embeddings[0]]
```

1 - **inputs** - данные по одному клиенту на вход

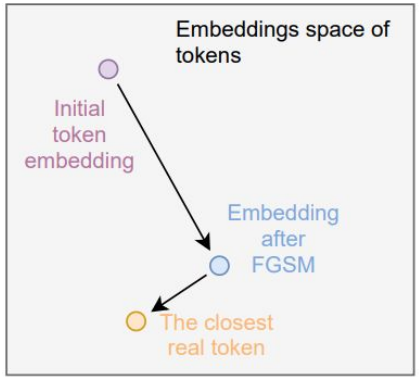
```
inputs

{'transactions': {'tokens': {'tokens': tensor([[ 10,   2,   8,   4,   2,  28,   3,   2,   3,   7,  14,   4,  12,  20,
              2, 112,   2,   2,   2,   3,  28,  28,   2,  29,   8,   7,   3,   3,
              3,  12,  12,   2,   2,   4,   2,   6,   4,  14,   9,   9,  28,   4,
             14,  11]])}},
 'amounts': {'tokens': {'tokens': tensor([[ 4, 27, 37, 70, 27, 77, 52, 12,  2, 97, 99, 19, 24,  3, 12, 12, 76, 75,
              2,  7, 28, 31, 32, 54, 67, 69,  8,  3, 62, 10, 66, 81, 44, 45,  6,  6,
             20, 66, 40, 47, 77, 40, 52,  5]])}},
 'label': tensor([0]),
 'client_id': tensor([28])}
```

2 - **adversarial_indexes** - возьмем индексы MCC кодов
3 - **orig_prob** - скор классификатора по исходным данным клиент
4 - **emb_out** - эмбеддинги для набора MCC кодов
5 - embedding_splitted - сформируем список из эмбеддингов исходных для каждой отдельной транзакции(MCC) и выключим градиенты

Приложение 1 - FGSM - атака часть 2

```
1 for step in range(self.num_steps):
      # choose random index of embeddings (except for start/end tokens)
      random_idx = random.randint(1, max(1, len(data_to_attack.transactions) - 2))
2     # only one embedding can be modified
      embeddings_splitted[random_idx].requires_grad = True

      # calculate the loss for current embeddings
      loss = self.classifier.forward_on_transaction_embeddings(
          transaction_embeddings=torch.stack(embeddings_splitted, dim=0).unsqueeze(0),
3         mask=emb_out["mask"],
          amounts=inputs["amounts"],
          label=inputs["label"],
      )["loss"]
      loss.backward()

      # update the chosen embedding
4     embeddings_splitted[random_idx] = (
          embeddings_splitted[random_idx] + self.epsilon * embeddings_splitted[random_idx].grad.data.sign()
      )
      self.classifier.zero_grad()

      # find the closest embedding for the modified one
      distances = torch.nn.functional.pairwise_distance(embeddings_splitted[random_idx], self.emb_layer)
      # we dont choose special tokens
      distances[self.special_indexes] = 10 ** 16

5     # swap embeddings
      closest_idx = distances.argmin().item()
      embeddings_splitted[random_idx] = self.emb_layer[closest_idx]
      embeddings_splitted = [e.detach() for e in embeddings_splitted]

      # get adversarial indexes
      adversarial_idexes[random_idx] = closest_idx
```

1 - **step** - выполняем атаку num_step раз
2 - **random_idx -** выбираем позицию в последовательности, в которой будем менять MCC код. Далее для этой позиции мы включим градиенты (ранее мы выключали градиенты)
3 - **loss -** выполним forward для нашей модели и далее выполним обратное распространение ошибки методом backward
4 - **embedding_splitted[random_idx] -** здесь основное колдунство. Мы получаем знаки градиента по эмбеддингу позиции random_idx (н-р, это 135 из 300 транзакция) в виде вектора длиной размера эмбеддинга - там внутри [1.,-1.,1.,...] знаки. Это и есть направления градиента, которые умножаем на epsilon. И далее прибавляем это смещение к нашему исходному эмбеддингу нашей позиции random_idx. Это приведет к тому, что для random_idx эмбеддинг уже не будет соответствовать его исходному MCC коду, а будет смещен как на картинке ниже. И нужно будет отыскать в этом смещенном положении ближайший MCC код, который мы и посчитаем заменой(атакой)



5 - **closest_idx -** поиск в смещенном после FGSM процедуры положении ближайшего MCC кода. И фиксация его эмбеддинга как нового значения для данных, по которым будем оценивать атаку

Приложение 1 - FGSM - атака часть 3

Рассмотрим детальнее саму атаку FGSM: https://github.com/fursovia/adversarial_sber/blob/master/advsber/attackers/fgsm.py

```
1   adv_data = deepcopy(data_to_attack)
    adv_data.transactions = decode_indexes(adversarial_idexes, vocab=self.vocab)

2   adv_inputs = data_to_tensors(adv_data, self.reader, self.vocab, self.device)

    # get adversarial probability and adversarial label
3   adv_probs = self.get_clf_probs(adv_inputs)
    adv_data.label = self.probs_to_label(adv_probs)
    adv_prob = adv_probs[self.label_to_index(data_to_attack.label)].item()

    output = AttackerOutput(
        data=data_to_attack.to_dict(),
4       adversarial_data=adv_data.to_dict(),
        probability=orig_prob,
        adversarial_probability=adv_prob,
        prob_diff=(orig_prob - adv_prob),
        wer=word_error_rate_on_sequences(data_to_attack.transactions, adv_data.transactions),
    )
    outputs.append(output)

5   best_output = self.find_best_attack(outputs)
    best_output.history = [output.to_dict() for output in outputs]

    return best_output
```

1 - **adv_data**- скопируем чистые данные и вставим туда последовательность MCC кодов, один из которых мы атаковали
2 - **adv_input -** приведем данные в вид, с которым умеет работать классфикатор
3 - **adv_prob -** посчитаем скор для атакованного набора
4 - **output -** посчитаем и зафиксируем прочие метрики по атакованному набору. Они будут нужны, чтобы в конце из num_step примеров по одному клиенту выбрать самую сильную атаку
5 - **best_output -** оценим все атаки для клиента и выберем лучшее