

Финальная работа по курсу Skillbox “Machine Learning Junior”

Кульгускина Оксана Викторовна

Данные:

Кредитная история заемщиков и атрибуты кредитных продуктов

Задача:

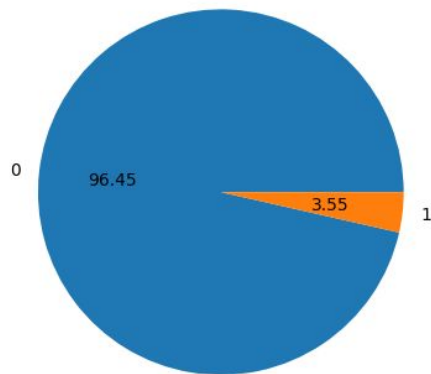
Оценка кредитного риска по клиенту

Целевая метрика:

ROC AUC ≥ 0.75

Особенности датасета:

- Исходные данные разбиты на 12 батчей в формате parquet, целевая переменная в отдельном датасете
- 3000000 строк в итоговом датасете, 61 признак
- Несбалансированное соотношение классов целевой переменной в датасете



- Абсолютное преобладание категориальных признаков, все значения закодированы или бинаризованы, ряд категорий содержат до 20 уникальных значений

Основные сложности при подготовке модели

Вычислительные ресурсы

Уменьшить размерность датасета за счет исключения нерелевантных признаков

Использовать доступные облачные ресурсы (Yandex Datasphere, Google Colab, Kaggle)

Контролировать потребление вычислительных ресурсов и время обработки в процессе разработки и обучения

Преобладание категориальных признаков

Создать новые числовые признаки на основе доступных категориальных:

◊ ONE и агрегация по ID позволит перевести категориальные признаки в количественные.

◊ Можно создать новые количественные признаки на базе полученных с помощью ONE и агрегации

Несбалансированность выборки

Использовать инструменты и алгоритмы предназначенные для работы с несбалансированными данными

Baseline

Анализ данных и подготовка к сборке датасета

Сборка датасета с использованием OneHotEncoder и агрегации по id

Обучение базовых моделей

Выводы

Анализ данных и подготовка к сборке

Поскольку признаки в основном категориальные, и значения категорий закодированы, основными инструментами анализа являются `nunique()` и `value_counts()`.

Отбор признаков

`pre_loans_total_overdue`

содержит 1 значение категории, этот признак можно игнорировать

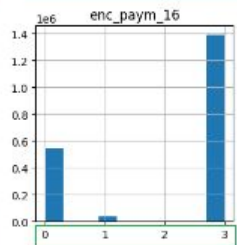
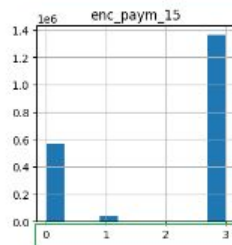
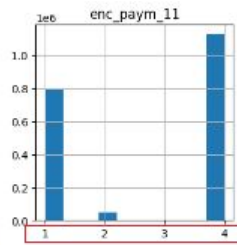
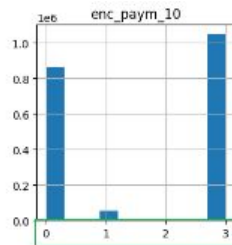
`pre_since_opened`
`pre_since_confirmed`
`pre_till_pclose`
`pre_till_fclose`

Набор признаков, связанный с датой сбора данных оператором, количество значений в этих категориях достигает до 20. Мы ничего не знаем о процессе сбора данных, но, по всей видимости, дата сбора данных оператором не может влиять на вероятность дефолта со стороны заемщика.

Ошибки в данных

`enc_paym_{0..N}`

В отдельных признаках значения категорий смещены на 1- поправим



Признаки, исключенные из ONE

`id`

`rn`

`pre_loans_total_overdue`

`pre_since_opened`

`pre_since_confirmed`

`pre_till_pclose`

`pre_till_fclose`

Сборка датасета*

*Базовый скрипт для порционной обработки данных приложен к ТЗ.

Обработка батча

1. Функция, заменяющая [1, 2, 3, 4] на [0, 1, 2, 3] в признаках `enc_payout_{0..N}`
2. Функция, применяющая ONE к переданному списку колонок
3. Функция, агрегирующая строки по id заемщика и одновременно заполняющая 'gp' максимальным значением (максимальный порядковый номер продукта соответствует количеству кредитных продуктов заемщика)

Агрегированный датасет

Из-за разного набора значений категорий в батчах агрегированный датасет содержит NaN.

Заменим NaN на 0

Датасет с признаками и целевой переменной

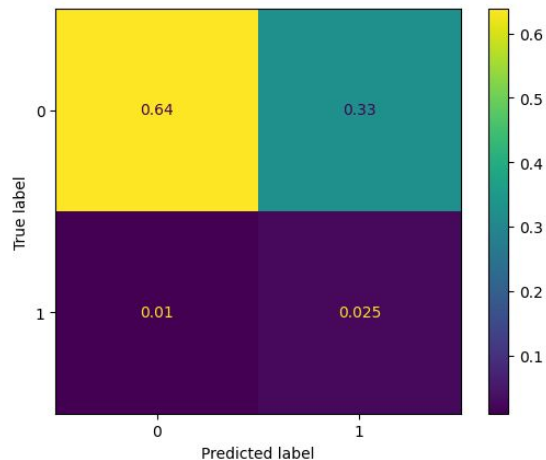
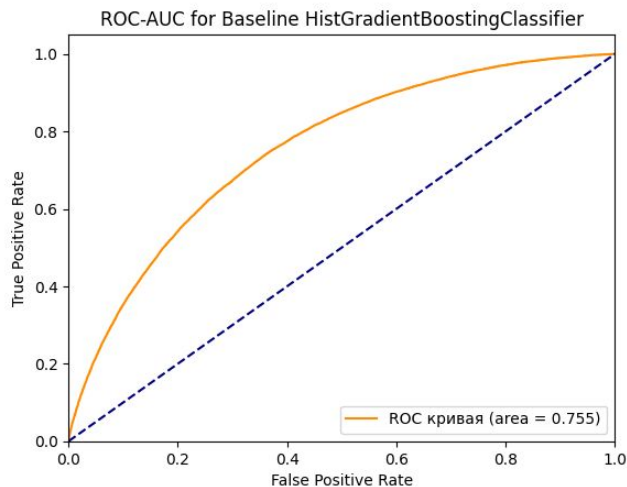
Размерность датасета после объединения с целевой переменной:
(3000000, 348)

Train/Test = 80/20

Baseline Model: HistGradientBoostingClassifier

HistGradientBoostingClassifier

```
HistGradientBoostingClassifier(class_weight='balanced', scoring='roc_auc')
```



	precision	recall	f1-score	support
0	0.98	0.66	0.79	578712
1	0.07	0.72	0.13	21288
accuracy			0.66	600000
macro avg	0.53	0.69	0.46	600000
weighted avg	0.95	0.66	0.77	600000

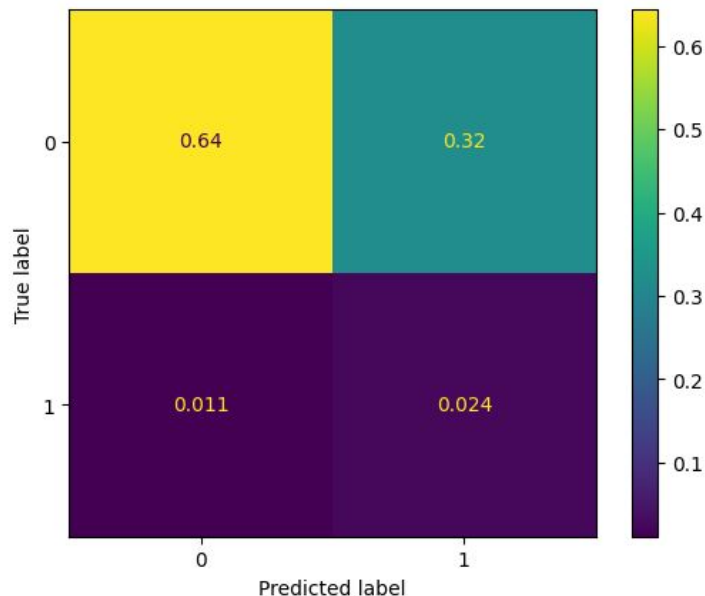
NB: Метрики приведены для тестовой выборки

Baseline Model: LogisticRegression

LogisticRegression

```
LogisticRegression(class_weight='balanced', max_iter=1000)
```

NB: К обучающей и тестовой выборкам применен StandardScaler()



ROC AUC=0.729

	precision	recall	f1-score	support
0	0.98	0.67	0.79	578712
1	0.07	0.68	0.13	21288
accuracy			0.67	600000
macro avg	0.53	0.67	0.46	600000
weighted avg	0.95	0.67	0.77	600000

NB: Метрики приведены для тестовой выборки

Baseline Model: Torch Neural Net

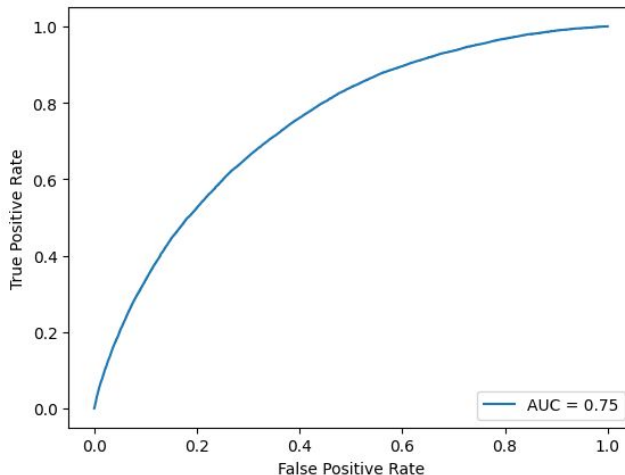
```
class BaseClassifier(nn.Module):  
    def __init__(self, input_dim, hidden_dim, output_dim):  
        super().__init__()  
  
        self.hidden = nn.Linear(input_dim, hidden_dim)  
        self.f1 = nn.ReLU()  
  
        self.output = nn.Linear(hidden_dim, output_dim)  
        self.f2 = nn.Sigmoid()  
  
    def forward(self, x):  
        x = self.f1(self.hidden(x))  
        x = self.f2(self.output(x))  
  
        return x
```

```
basenet = BaseClassifier(347, 231, 1)
```

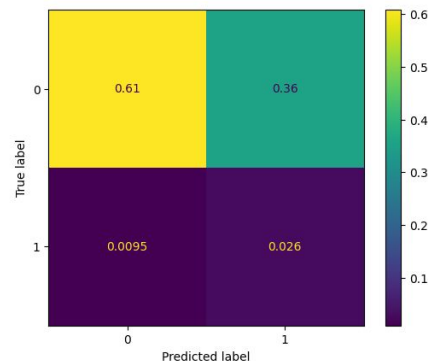
```
loss_fn = nn.BCELoss()
```

```
optimizer = torch.optim.SGD(basenet.parameters(), lr=0.01)
```

```
num_epochs = 10
```



NB: Для балансировки классов для экономии времени на первом этапе использовался RandomUnderSampler

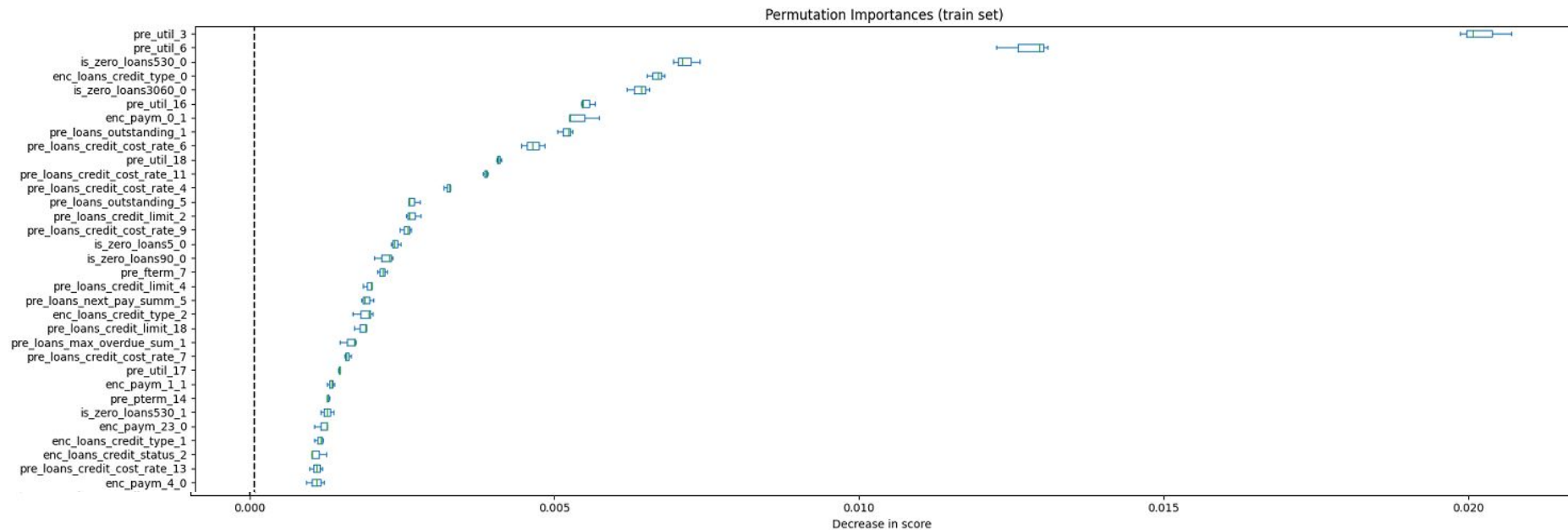


NB: метрики приведены для тестовой выборки

Выводы

Две из трех опробованных базовых моделей без настройки и подбора параметров, прицельного feature engineering и с грубой балансировкой преодолевают желаемый порог метрики 0.75 на тестовой части. Наиболее перспективной выглядит модель HistGradientBoostingClassifier, на нее и будем ориентироваться.

Feature Importances, полученные с помощью permutation_importance для HistGradientBoostingClassifier



Feature Engineering

Пересборка датасета с добавлением агрегации по столбцам

Отбор релевантных признаков

Отбор нерелевантных признаков

Добавление средневзвешенных значений отобранных признаков

Уточняющий отбор нерелевантных признаков

Валидация результатов

Признаки с использование агрегации по столбцам

- флаг о том, что по продукту не было ни одной просрочки

```
def count_zero_loans(data):  
    zero_loans = ['is_zero_loans5', 'is_zero_loans530', 'is_zero_loans3060', 'is_zero_loans6090', 'is_zero_loans90']  
    data["no_delays"] = data.loc[:, zero_loans].eq(1).sum(axis=1).apply(lambda x: 1 if x==5 else 0)  
    return data
```

- количество статусов платежей за весь период

```
def count_enc_payments_status(data):  
    features_paym = ['enc_paym_0', 'enc_paym_1', 'enc_paym_2', 'enc_paym_3', 'enc_paym_4', 'enc_paym_5', 'enc_paym_6', 'enc_paym_7',  
                    'enc_paym_8', 'enc_paym_9', 'enc_paym_10', 'enc_paym_11',  
                    'enc_paym_12', 'enc_paym_13', 'enc_paym_14', 'enc_paym_15', 'enc_paym_16', 'enc_paym_17', 'enc_paym_18', 'enc_paym_19',  
                    'enc_paym_20', 'enc_paym_21', 'enc_paym_22',  
                    'enc_paym_23', 'enc_paym_24']  
  
    data["enc_paym_value0"] = data.loc[:, features_paym].eq(0).sum(axis=1)  
    data["enc_paym_value1"] = data.loc[:, features_paym].eq(1).sum(axis=1)  
    data["enc_paym_value2"] = data.loc[:, features_paym].eq(2).sum(axis=1)  
    data["enc_paym_value3"] = data.loc[:, features_paym].eq(3).sum(axis=1)  
  
    return data
```

Отбор релевантных признаков на пересобранном датасете

```
permutation_importance(n_repeats=3, scoring='roc_auc')
```

Порог отсеечения: 0.0001

HistGradientBoostingClassifier: 317 features of importance value < 0.0001

LogisticRegression: 203 features of importance value < 0.0001

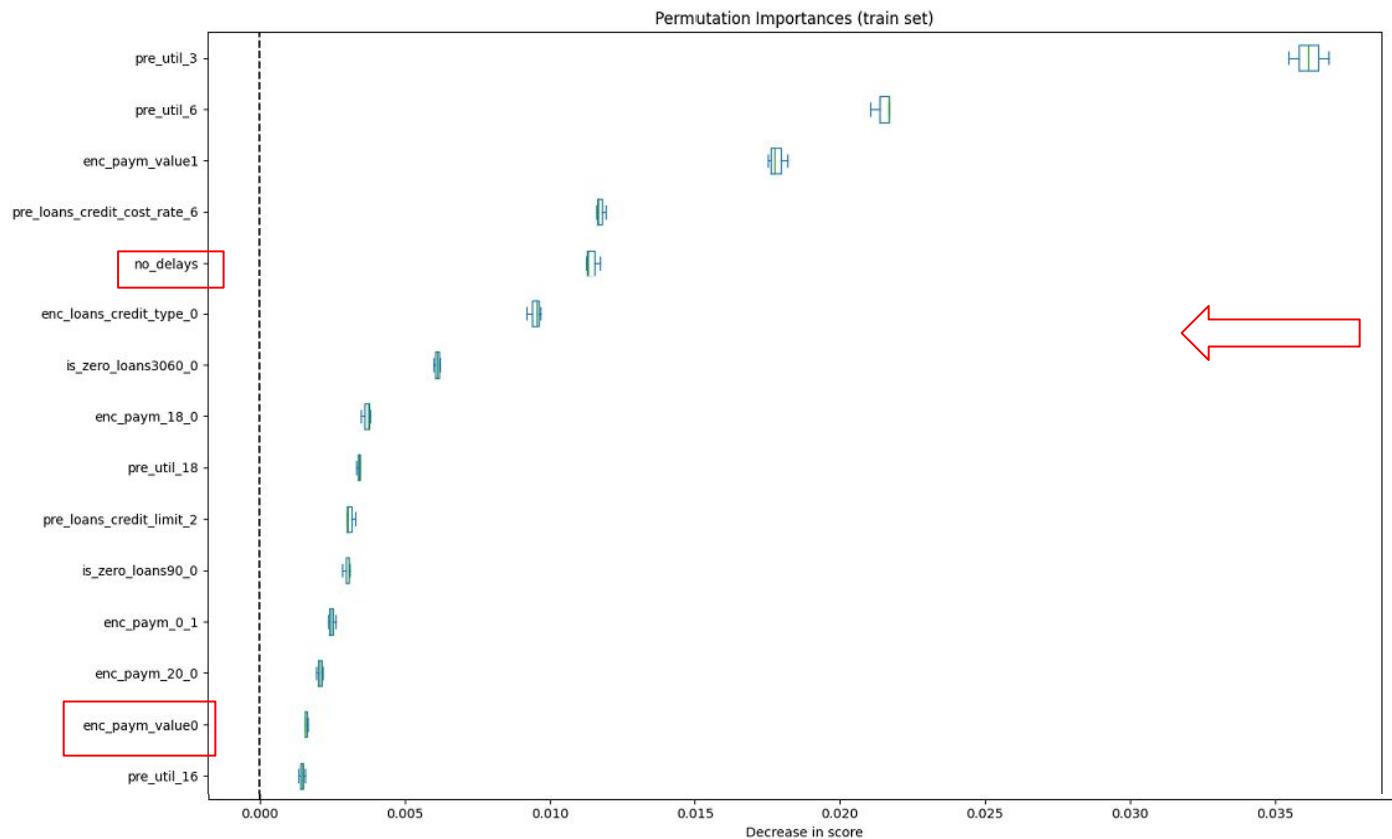
Список релевантных признаков (на основе результатов для HistGradientBoostingClassifier):

```
'enc_loans_credit_type_5', 'enc_paym_2_1', 'enc_loans_credit_type_2', 'enc_loans_credit_status_2',  
'pre_loans_credit_cost_rate_7', 'enc_paym_7_3', 'enc_paym_11_0', 'pre_loans_credit_limit_4',  
'pre_loans_next_pay_summ_5', 'enc_paym_22_0', 'is_zero_loans6090_0', 'pre_loans_outstanding_5', 'enc_paym_value2',  
'enc_paym_23_0', 'enc_paym_2_3', 'enc_paym_1_3', 'is_zero_loans530_0', 'pre_fterm_7',  
'enc_loans_credit_status_5', 'is_zero_util_0', 'pre_util_16', 'enc_paym_value0', 'enc_paym_20_0',  
'enc_paym_0_1', 'is_zero_loans90_0', 'pre_loans_credit_limit_2', 'pre_util_18', 'enc_paym_18_0',  
'is_zero_loans3060_0', 'enc_loans_credit_type_0', 'no_delays', 'pre_loans_credit_cost_rate_6',  
'enc_paym_value1', 'pre_util_6', 'pre_util_3'
```

35

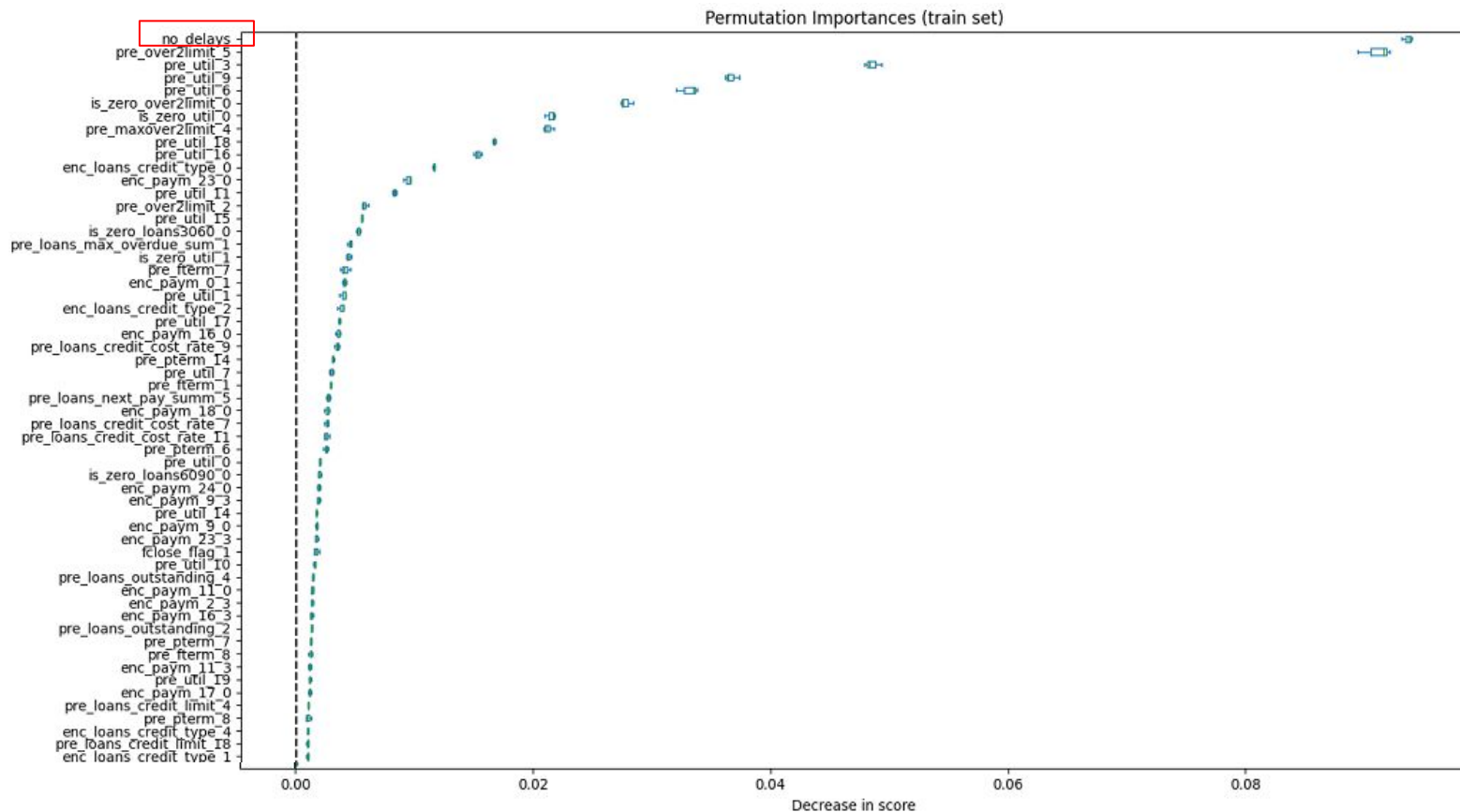
признаков

HistGradientBoostingClassifier



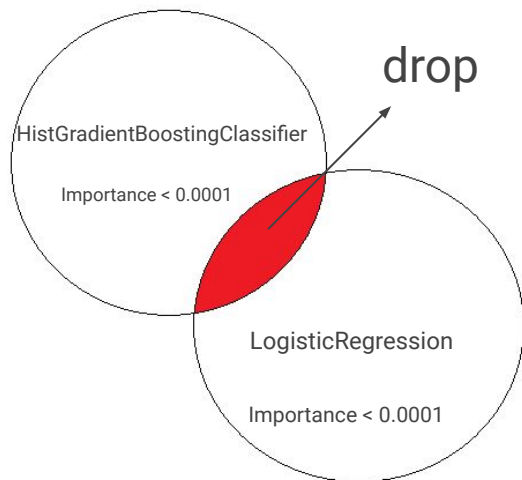
Новые признаки попали
в число наиболее
важных

Logistic Regression



Нерелевантные признаки

Удалим из датасета признаки, которые по результатам permutation_importance оказались нерелевантными для обеих моделей



Новые признаки

На основе списка релевантных признаков создадим новые признаки: средневзвешенные значения

```
def add_avg_features(data, flist):
    datanew = data.copy()
    datanew[flist] = datanew[flist].apply(lambda x: round(x/datanew['rn'], 2)).astype('float32')
    datanew = datanew[flist]

    datanew_cols=[str(i)+'_avg' for i in flist]
    colsdict = {i:j for i,j in zip(flist, datanew_cols)}
    datanew = datanew.rename(colsdict, axis=1)
    data = data.join(datanew, on='id')
    gc.collect()
    return data
```

Повторное обучение

Переобучим модели на новом наборе признаков

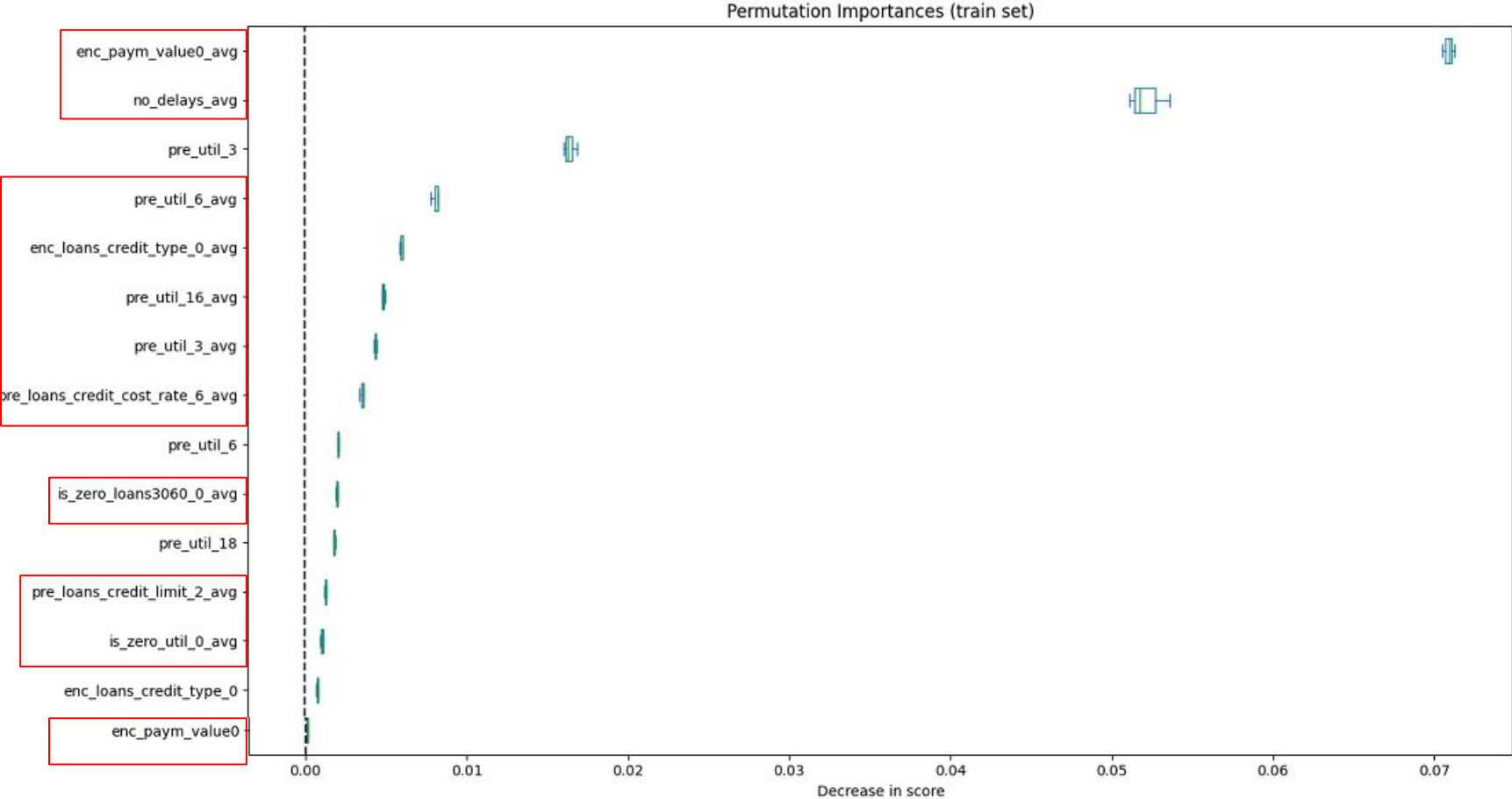
Повторно применим permutation_importance

Уточним список нерелевантных признаков с учетом добавленных колонок

Удалим из датасета признаки, которые по результатам 2 прохода permutation_importance оказались нерелевантными для обеих моделей

HistGradientBoostingClassifier

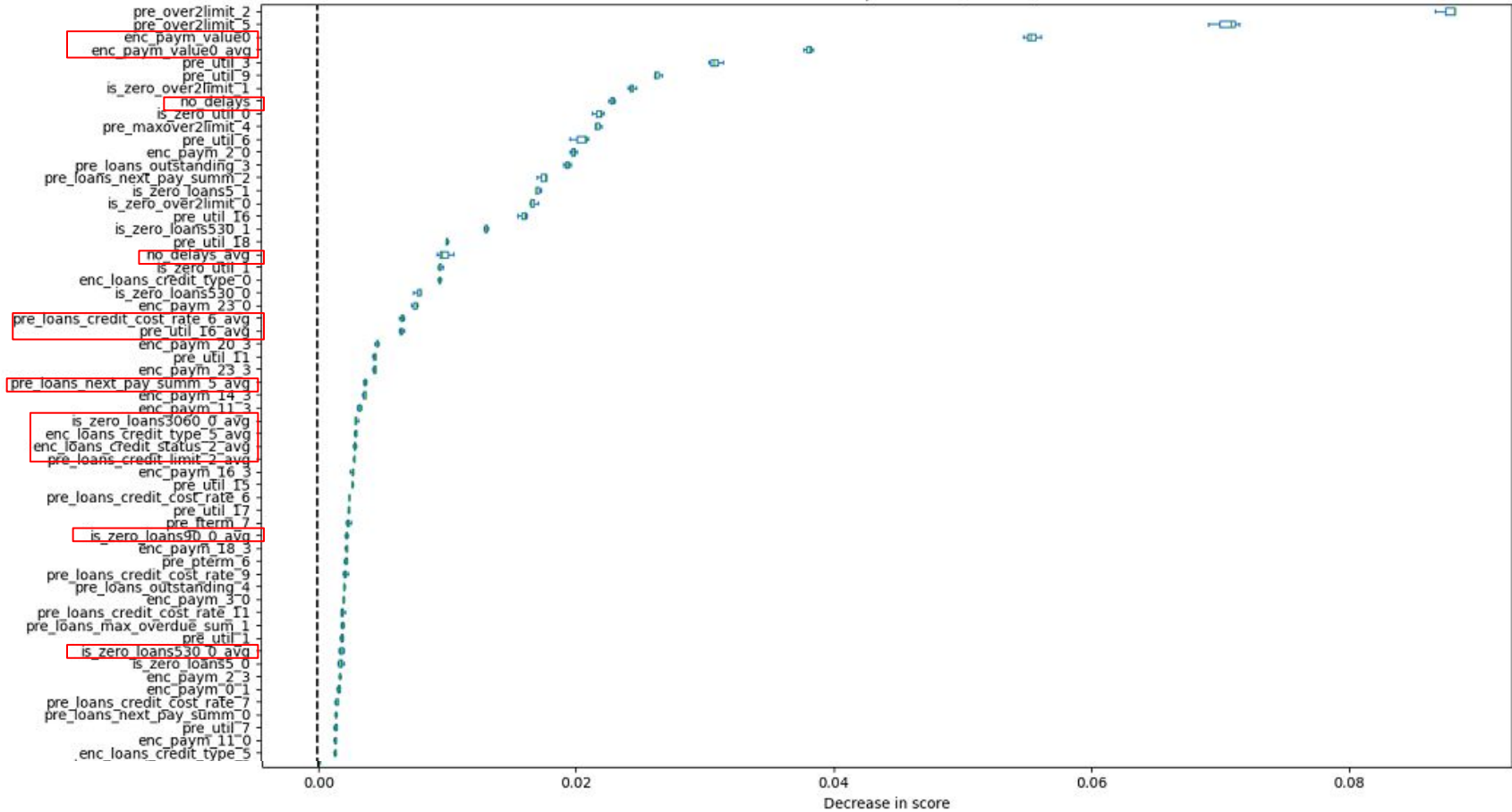
158 features of importance value < 0.0001



Logistic Regression

46 features of importance value < 0.0001

Permutation Importances (train set)



Валидация результатов Feature Engineering

Размерность датасетов после всех преобразований:

```
x_train.shape, x_test.shape
```

```
((2400000, 160), (600000, 160))
```

```
HistGradientBoostingClassifier  
HistGradientBoostingClassifier(class_weight='balanced', scoring='roc_auc')
```

ROC AUC=0.754

	precision	recall	f1-score	support
0	0.98	0.66	0.79	578712
1	0.07	0.72	0.13	21288
accuracy			0.66	600000
macro avg	0.53	0.69	0.46	600000
weighted avg	0.95	0.66	0.77	600000

```
LogisticRegression  
LogisticRegression(class_weight='balanced', max_iter=1000)
```

ROC AUC=0.744

	precision	recall	f1-score	support
0	0.98	0.69	0.81	578712
1	0.07	0.68	0.13	21288
accuracy			0.69	600000
macro avg	0.53	0.68	0.47	600000
weighted avg	0.95	0.69	0.78	600000

Выводы

Метрика ROC AUC для градиентного бустинга при контрольном прогоне на датасете с новыми признаками уменьшилась на 0.001, что можно отнести к несущественным колебаниям, т.к. `random_state` и другие параметры `baseline`-модели не фиксировались

Прирост метрики ROC AUC для логистической регрессии составил > 0.1 , что считаем хорошим результатом, т.к. модель планируется использовать в составе ансамбля.

Добавленные признаки по результатам применения `permutation_importances` входят в число наиболее значимых признаков, при этом мы существенно уменьшили размерность датасета без потери качества и избавились от потенциальных шумов.

Проведенный `feature engineering` считаем успешным и полезным.

Modelling

Настройка классификатора

Stacking

Выбор финального
классификатора

HistGradientBoostingClassifier tuning

Подбор оптимальных параметров осуществлялся в ходе первых экспериментов с помощью нескольких попарных проходов GridSearchCV(cv=3, scoring = roc_auc_score)

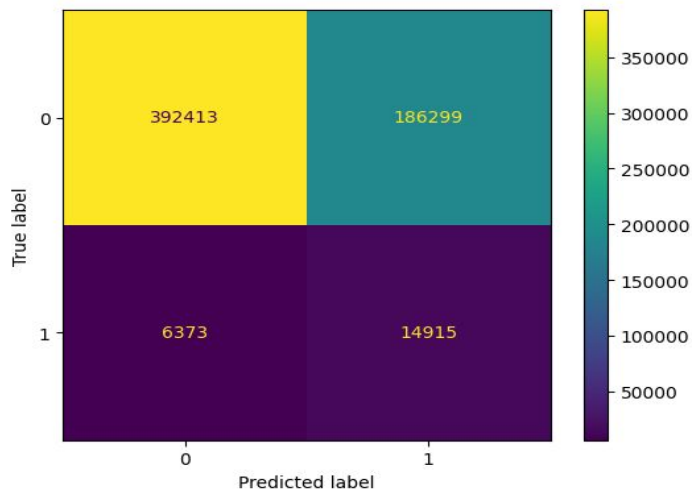
На первом проходе паре параметров задавалась широкая сетка значений, на втором полученные результаты уточнялись.

Т.о. была сформирована сетка параметров.

В дальнейшем при необходимости с помощью GridSearchCV или RandomizedSearchCV уточнялись только значения max_depth и min_samples_leaf.

Итоговый вариант классификатора: **class_weight='balanced', max_depth=15, max_leaf_nodes=72, min_samples_leaf=28, max_iter=150, scoring='roc_auc', tol=0.00001, l2_regularization=0.01, learning_rate=0.1.**

Существенного прироста метрики это не дало (0.001-0.002 относительно Baseline), но благоприятно повлияло на стабильность результатов. Также в сравнении с Baseline классификатор стал меньше ошибаться на 0



▲ ROC AUC=0.756

	precision	recall	f1-score	support
0	0.98	0.68	0.80	578712
1	0.07	0.70	0.13	21288
accuracy			0.68	600000
macro avg	0.53	0.69	0.47	600000
weighted avg	0.95	0.68	0.78	600000

2nd Torch Neural Net

Во второй вариант nn добавили балансировку классов, также слегка изменили архитектуру

```
n_features = x_train.shape[1]
hidden_layer_size = round((n_features/3)*2)

class BaseClassifier(nn.Module):
    def __init__(self, input_dim=n_features, hidden_dim=hidden_layer_size, output_dim=1):
        super().__init__()

        self.hidden1 = nn.Linear(input_dim, hidden_dim)
        self.f1 = nn.Sigmoid()

        self.hidden2 = nn.Linear(hidden_dim, 10)
        self.f2 = nn.Sigmoid()

        self.output = nn.Linear(10, output_dim)
        self.f3 = nn.Sigmoid()

    def forward(self, x):
        x = self.f1(self.hidden1(x))
        x = self.f2(self.hidden2(x))
        x = self.f3(self.output(x))

        return x
```

```
basenet = BaseClassifier()

optimizer = torch.optim.SGD(basenet.parameters(), lr=0.01)

num_epochs = 10
class_weights = class_weight.compute_class_weight(class_weight='balanced',
                                                    classes=np.unique(y_train),
                                                    y=y_train)

for epoch in range(num_epochs):
    for X, y in train_dataloader:
        pred = basenet(X)

        weights = torch.zeros_like(y.unsqueeze(-1))
        weights[y==0] = class_weights[0]
        weights[y==1] = class_weights[1]
        loss = F.binary_cross_entropy(pred, y.unsqueeze(-1), weight=weights)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

 ROC AUC = 0.752

Для логистической регрессии прицельно параметры не подбирали, но далее вместо LogisticRegression будем задействовать LogisticRegressionCV:

```
LogisticRegressionCV(class_weight='balanced', cv=3, max_iter=1000,
                      scoring='roc_auc')
```

Градиентный
бустинг:

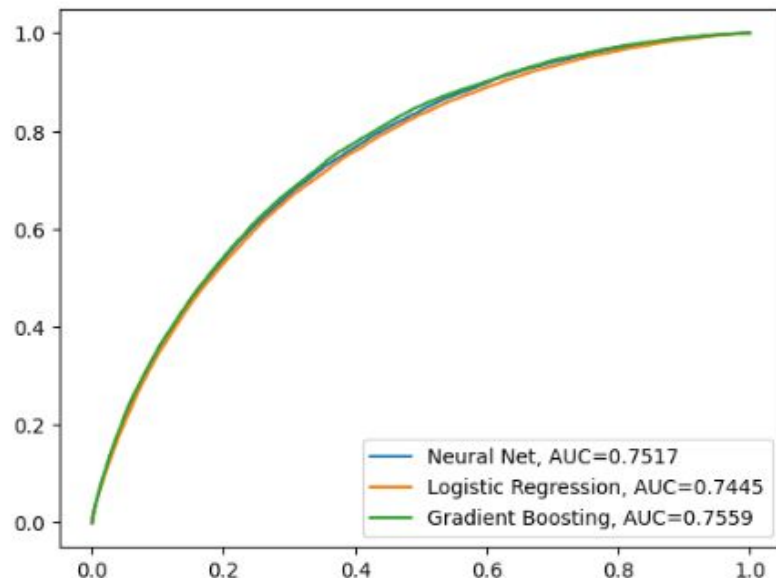
	precision	recall	f1-score	support
0	0.98	0.68	0.80	578712
1	0.07	0.70	0.13	21288
accuracy			0.68	600000
macro avg	0.53	0.69	0.47	600000
weighted avg	0.95	0.68	0.78	600000

Логистическая
регрессия:

	precision	recall	f1-score	support
0	0.98	0.69	0.81	578712
1	0.07	0.68	0.13	21288
accuracy			0.69	600000
macro avg	0.53	0.68	0.47	600000
weighted avg	0.95	0.69	0.78	600000

NN:

	precision	recall	f1-score	support
0	0.98	0.64	0.78	578712
1	0.07	0.73	0.13	21288
accuracy			0.65	600000
macro avg	0.53	0.69	0.45	600000
weighted avg	0.95	0.65	0.75	600000



Stacking

Попробуем еще поднять качество классификации с помощью ансамблевых методов.

Будем использовать стекинг

Не будем строить большой ансамбль, вместо этого попробуем результаты добавить в качестве признаков к датасету.

```
scale_pos_weight = round(len(y_train[y_train==0])/len(y_train[y_train==1]))
```

```
estimators_L1 = [  
    ('gr1', HistGradientBoostingClassifier(class_weight='balanced', max_depth=15, max_leaf_nodes=72, min_samples_leaf=28,  
                                           max_iter=150, scoring='roc_auc', tol=0.00001, l2_regularization=0.01, learning_rate=0.  
1))),  
    ('logreg', LogisticRegressionCV(class_weight='balanced', refit=True, cv=3, scoring='roc_auc', max_iter=1000)),  
  
    ('xgb', XGBClassifier(scale_pos_weight=scale_pos_weight, n_jobs=-1, learning_rate=0.1, max_depth=15))  
]  
stack1 = StackingTransformer(estimators=estimators_L1,  
                             regression=False,  
                             variant='A',  
                             needs_proba=True,  
                             n_folds=3,  
                             shuffle=False,  
                             random_state=None,  
                             stratified=True  
                             )
```

→ добавим XGBClassifier

→ На выходе вероятности, а не метки классов

IsolationForest вынесем в отдельный ансамбль, т.к. у него нет predict_proba

```
estimators_L2 = [('if', IsolationForest(contamination=0.0354))]  
  
stack2 = StackingTransformer(estimators=estimators_L2,  
                             regression=False,  
                             needs_proba=False,  
                             variant='A',  
                             random_state=None,  
  
                             n_folds=3,  
                             shuffle=False  
                             )
```

Обучим ансамбли

```
l1_pipe = Pipeline([  
    ('stack1', stack1)])  
l1_pipe.fit(x_train, y_train)  
  
l2_pipe = Pipeline([  
    ('stack2', stack2)])  
l2_pipe.fit(x_train, y_train)
```

Пересоберем датасет

```
def add_stacked_features(data):  
  
    data_copy1 = data.copy()  
    data_copy2 = data.copy()  
    s1 = l1_pipe_.transform(data_copy1)  
    s2 = l2_pipe_.transform(data_copy2)  
    s_train = pd.DataFrame(s1, columns=['mod1_0', 'mod1_1', 'mod2_0', 'mod2_1', 'mod3_0', 'mos=3_1'])  
    s_train = s_train.drop(['mod1_0', 'mod2_0', 'mod3_0'],axis=1)  
    s2 = pd.DataFrame(s2, columns=['if'])  
    data = data.join(s_train)  
    data = data.join(s2)  
    gc.collect()  
  
    return data
```

Поскольку мы просили на выходе вероятности, каждая модель из 1-го ансамбля выдает 2 колонки, но нам нужна только одна из них

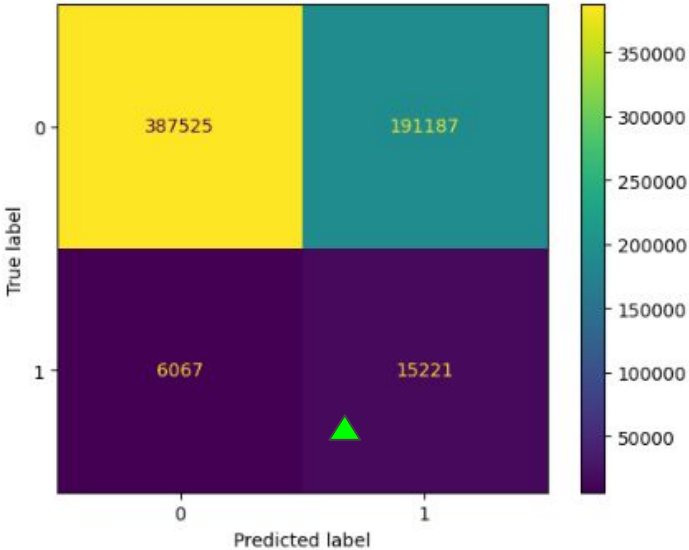
Размеры полученного датасета: (2400000, 164)

Обучим классификатор на обновленном датасете:

```
HistGradientBoostingClassifier
HistGradientBoostingClassifier(class_weight='balanced', l2_regularization=0.01,
                                max_depth=15, max_iter=150, max_leaf_nodes=72,
                                min_samples_leaf=30, scoring='roc_auc',
                                tol=1e-05)
```

▲ ROC AUC=0.759

	precision	recall	f1-score	support
0	0.98	0.67	0.80	578712
1	0.07	▲ 0.72	0.13	21288
accuracy			0.67	600000
macro avg	0.53	0.69	0.47	600000
weighted avg	0.95	0.67	0.77	600000



При обучении NN на обновленном датасете roc auc score 0.7559. Результаты улучшились, но NN по-прежнему отстает от градиентного бустинга, при этом ее интеграция сложнее, и времени на обучение тратится больше. В качестве финального классификатора выбираем HistGradBoostingClassifier

Выводы

Подбор гиперпараметров классификатора незначительно улучшил качество классификации.

Применение stacking и добавление новых признаков позволило улучшить качество классификации.

В качестве финального классификатора выбран:

HistGradientBoostingClassifier (class_weight='balanced', max_depth=15, max_leaf_nodes=72, min_samples_leaf=28, max_iter=150, scoring='roc_auc', tol=0.00001, l2_regularization=0.01, learning_rate=0.1)

Задача добиться максимально возможного значения метрики не ставилась, ставилась задача определить тенденции и пути дальнейшего улучшения модели. На каждом этапе эксперимента удавалось добиться устойчивого роста, что считаем успешным результатом. Потенциал развития и улучшения текущей модели сохраняется.

Pipeline

Сборка финального пайплайна

Сериализация модели

Данные

При сборке пайплайна использовали подготовленные на этапе feature engineering и сохраненные в отдельные файлы данные :

- 1) Датасет, содержащий признаки, полученные с помощью ONE и агрегации, и целевую переменную
- 2) Список признаков нулевой значимости
- 3) Список релевантных признаков

Загружаем подготовленный или пересобираем датасет, содержащий признаки и целевую переменную

```
data.shape
```

```
(3000000, 354)
```

Переразбиваем датасет на обучающую и тестовую выборки

```
train, test = train_test_split(data, stratify=data['flag'], test_size=0.2, random_state=17)
```

```
train.shape, test.shape
```

```
((2400000, 354), (600000, 354))
```

Для корректной работы функций сбрасываем индексы

```
train.reset_index(drop=True, inplace=True)  
test.reset_index(drop=True, inplace=True)
```

Разделяем признаки и целевую переменную

```
x_train = train.drop('flag', axis=1)  
y_train = train['flag']  
x_test = test.drop('flag', axis=1)  
y_test = test['flag']
```

Этапы сборки

Работа с признаками

```
constructor = Pipeline(steps=[
    ('add_avg_features', FunctionTransformer(add_avg_features)),
    ('del_zero_imp_features', FunctionTransformer(del_zero_imp_features))
])
```

добавляем колонки средневзвешенных значений по списку

удаляем нерелевантные колонки по списку

```
def add_avg_features(data, flist=relevant_features):
    datanew = data.copy()
    datanew[flist] = datanew[flist].apply(lambda x: round(x/datanew['rn'], 2)).astype('float32')
    datanew = datanew[flist]

    datanew_cols=[str(i)+'_avg' for i in flist]
    colsdict = {i:j for i,j in zip(flist, datanew_cols)}
    datanew = datanew.rename(colsdict, axis=1)
    data = data.join(datanew)
    gc.collect()
    return data

def del_zero_imp_features(data, flist=zero_features):
    datanew=data.copy()
    datanew = datanew.drop(flist, axis=1)

    return datanew
```


Масштабирование:

```
numerical_transformer = Pipeline(steps=[('scaler', StandardScaler())])

encoder = ColumnTransformer(remainder='passthrough', verbose_feature_names_out=False, transformers=[
    ('num', numerical_transformer, make_column_selector(dtype_include='number')),
]).set_output(transform='pandas')
```

Собираем промежуточный пайплайн:

```
preprocessor_pipe = Pipeline([
    ('constructor', constructor),
    ('encoder', encoder)])
```

Готовим датасет для обучения ансамблей:

```
x_train_prep = preprocessor_pipe.fit_transform(x_train)|
```


Сборка и обучение ансамблей

```
estimators_L1 = [  
    ('gr1', HistGradientBoostingClassifier(class_weight='balanced', max_depth=15, max_leaf_nodes=72, min_samples_leaf=28,  
                                           max_iter=150, scoring='roc_auc', tol=0.00001, l2_regularization=0.01, learning_rate=0.1)),  
    ('logreg', LogisticRegressionCV(class_weight='balanced', refit=True, cv=3, scoring='roc_auc', max_iter=1000)),  
  
    ('xgb', XGBClassifier(scale_pos_weight=scale_pos_weight, n_jobs=-1, learning_rate=0.1, max_depth=15))  
]  
stack1 = StackingTransformer(estimators=estimators_L1,  
                             regression=False,  
                             variant='A',  
                             needs_proba=True,  
                             n_folds=3,  
                             shuffle=False,  
                             random_state=None,  
                             stratified=True  
                             )
```

```
estimators_L2 = [('if', IsolationForest(contamination=0.0354))]  
  
stack2 = StackingTransformer(estimators=estimators_L2,  
                             regression=False,  
                             needs_proba=False,  
                             variant='A',  
                             random_state=None,  
  
                             n_folds=3,  
                             shuffle=False  
                             )
```

```
l1_pipe = Pipeline([  
    ('stack1', stack1)])  
l1_pipe.fit(x_train_prep, y_train)  
  
l2_pipe = Pipeline([  
    ('stack2', stack2)])  
l2_pipe.fit(x_train_prep, y_train)
```

учим ансамбли на
подготовленном
датасете
вне основного
пайплайна



В функции вызываем обученные ансамбли и трансформируем датасет, подклеивая к нему результаты ансамблей

```
def add_stacked_features(data):

    data_copy1 = data.copy()
    data_copy2 = data.copy()
    s1 = l1_pipe_.transform(data_copy1)
    s2 = l2_pipe_.transform(data_copy2)
    s_train = pd.DataFrame(s1, columns=['mod1_0', 'mod1_1', 'mod2_0', 'mod2_1', 'mod3_0', 'mos=3_1'])
    s_train = s_train.drop(['mod1_0', 'mod2_0', 'mod3_0'], axis=1)
    s2 = pd.DataFrame(s2, columns=['if'])
    data = data.join(s_train)
    data = data.join(s2)
    gc.collect()

    return data
```

Добавляем трансформер в основной пайплайн

```
pipe_transformer = Pipeline([
    ('constructor', constructor),
    ('encoder', encoder),
    ('stacker', FunctionTransformer(add_stacked_features)),

])
```

Назначаем классификатор:

```
model = HistGradientBoostingClassifier(class_weight='balanced', max_depth=15,  
                                       max_leaf_nodes=72, min_samples_leaf=28,  
                                       max_iter=150, scoring='roc_auc', tol=0.00001,  
                                       l2_regularization=0.01, learning_rate=0.1)
```

Финальный пайплайн из двух шагов:

```
pipe = Pipeline([  
    ('transformer', pipe_transformer),  
    ('classifier', model)])
```

Учим:

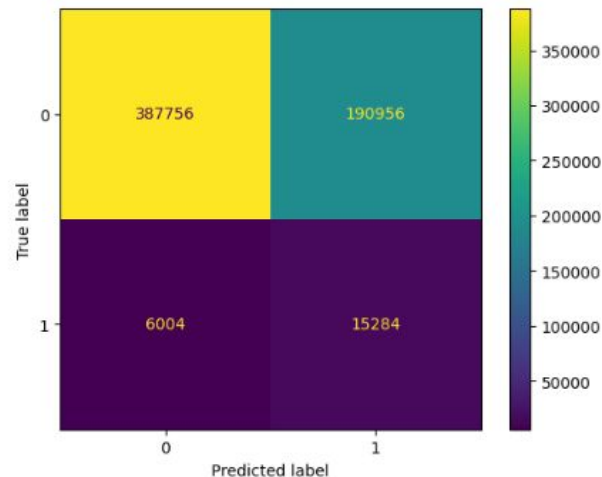
```
res = pipe.fit(x_train, y_train)
```

Тестируем:

```
eval_classifier(res, x_test, y_test)
```

ROC AUC=0.761▲

	precision	recall	f1-score	support
0	0.98	0.67	0.80	578712
1	0.07	0.72	0.13	21288
accuracy			0.67	600000
macro avg	0.53	0.69	0.47	600000
weighted avg	0.95	0.67	0.77	600000



Сериализация классификатора

Сериализуем обученный пайплайн

```
with open('model.pkl', 'wb') as f:  
    dill.Pickler(f, recurse=True).dump(res)
```

Загрузим и проверим, что все работает

```
with open('/kaggle/working/model.pkl', 'rb') as f:  
    retest = dill.load(f)
```



```
samp = x_test.sample(n=3)  
samp.reset_index(drop=True, inplace=True)  
retest['transformer'].transform(samp)
```

fterm_0	pre_fterm_1	pre_fterm_3	...	enc_loans_credit_type_0_avg	no_delays_avg	pre_loans_credit_cost_rate_6_avg	enc_paym_value1_avg	pre_util_6_avg	pre_util_3_avg	mod1_1	mod2_1	mos=3_1	if
1516986	-0.538554	-0.239964	...	0.517862	0.596240	-0.156009	-0.487358	0.803812	0.367095	0.241818	0.389350	0.044856	1
1589566	-0.538554	-0.239964	...	-0.342864	0.942752	-0.156009	-0.562584	-0.271329	-0.391822	0.111388	0.190909	0.054000	1
1516986	0.799470	-0.239964	...	-0.342864	-1.867840	-0.156009	3.315769	-0.271329	-0.391822	0.596555	0.545237	0.141719	1

```
retest.predict(samp)
```

```
array([0, 0, 1])
```

```
eval_classifier(retest, x_test, y_test)
```

ROC AUC=0.761

	precision	recall	f1-score	support
0	0.98	0.67	0.80	578712
1	0.07	0.72	0.13	21288
accuracy			0.67	600000
macro avg	0.53	0.69	0.47	600000
weighted avg	0.95	0.67	0.77	600000

transform работает

predict работает

качество не изменилось

Сводная таблица ROC AUC score по этапам

Классификатор: HistGradientBoostingClassifier (другие варианты применялись выборочно и/или были отброшены на ранних этапах, и системных замеров на всех этапах не проводилось)

Выборка: Test (замеры на train прицельно не фиксировались)

Stage	ROC AUC score
Baseline	0.755
Feature Engineering	0.754
Hyperparameter Tuning	0.756
Add Stacking Features	0.759
Final Pipeline	0.761

Выводы

- Целевое значение метрики 0.75 достигнуто.
- Удалось повысить качество классификации и добиться значения метрики 76.1
- Был проведен успешный feature engineering: размерность датасета уменьшена без потери качества, добавлены полезные признаки, сохраняется потенциал улучшения классификатора
- Классификатор успешно сериализован, проверена работоспособность классификатора после десериализации