Московский Авиационный Институт
(Национальный Исследовательский Университет)
Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Лабораторная работа №1 по курсу
«Машинное обучение»

Линейные модели

Студент: Моисеенков Илья Павлович
Группа: М80 – 308Б-19
Дата: 24.05.2022
Оценка: _____
Подпись: _____

Москва, 2022

1. Постановка задачи

1) Реализовать следующие алгоритмы машинного обучения: Linear/ Logistic Regression, SVM, KNN, Naive Bayes в отдельных классах
2) Данные классы должны наследоваться от BaseEstimator и ClassifierMixin, иметь методы fit и predict
3) Вы должны организовать весь процесс предобработки, обучения и тестирования с помощью Pipeline
4) Вы должны настроить гиперпараметры моделей с помощью кросс валидации (GridSearchCV, RandomSearchCV) вывести и сохранить эти гиперпараметры в файл, вместе с обученными моделями
5) Проделать аналогично с коробочными решениями
6) Для каждой модели получить оценки метрик: Confusion Matrix, Accuracy, Recall, Precision, ROC_AUC curve
7) Проанализировать полученные результаты и сделать выводы о применимости моделей
8) Загрузить полученные гиперпараметры модели и обученные модели в формате pickle на гит вместе с jupyter notebook ваших экспериментов

2. Подготовка данных

Для начала необходимо подготовить данные для обучения:

- категориальные признаки преобразуем с помощью one-hot encoding. Если фича бинарная, то будем оставлять только одну фичу для нее.
- BMI (индекс массы тела) приведем к стандартному нормальному распределению (т.к. этот признак изначально был распределен нормально) с помощью Standard Scaler.
- остальные количественные признаки преобразуем с помощью MinMaxScaler.

```
data_preprocessing = ColumnTransformer([
    ('ohe', OneHotEncoder(drop='if_binary'), categorical_features),
    ('stdscale', StandardScaler(), ['BMI']),
    ('minmax', MinMaxScaler(), [feat for feat in numerical_features if feat != 
'BMI'])
])
```

Разделим данные на трейн и тест с помощью функции train_test_split. На тестовую часть оставим 20% данных.

3. Подсчет метрик

Сразу определим функцию для оценивания качества моделей. Будем считать метрики accuracy, precision, recall, rocauc и будем строить confusion matrix.

Важный момент: мы решаем задачу предсказания вероятности сердечного приступа. Нам важно максимизировать реколл. Поэтому будем дополнительно оценивать соотношение пресижена и реколла с помощью PR-curve. Зафиксируем реколл=0.7 и будем оценивать модель при таком реколле.

Для SVM нет возможности получить вероятность принадлежности к классу, поэтому учтем это в функции.

```
def get_metrics(model, X, y_true, threshold=0.5, use_probas=True):

    if use_probas:
        y_pred_probas = model.predict_proba(X)
        if len(y_pred_probas.shape) == 2:
            y_pred_probas = y_pred_probas[:, 1]
        y_pred = y_pred_probas > threshold
    else:
        y_pred = model.predict(X)

    print('Accuracy = ', accuracy_score(y_true, y_pred))
    print('Precision = ', precision_score(y_true, y_pred))
    print('Recall = ', recall_score(y_true, y_pred))
    if use_probas:
        print('ROC AUC = ', roc_auc_score(y_true, y_pred_probas))
    print('Confusion matrix:')
    print(confusion_matrix(y_true, y_pred))

    if use_probas:
        precision,   recall,   thresholds   =   precision_recall_curve(y_true,
y_pred_probas)
        plt.figure(figsize=(15, 8))
        plt.xlabel('Recall')
        plt.ylabel('Presicion')
        plt.title('Precision-recall curve')
        plt.xticks(np.arange(0, 1.1, 0.1))
        plt.yticks(np.arange(0, 1.1, 0.1))
        plt.grid()
        plt.plot(recall, precision)
```

4. Обучение и валидация моделей

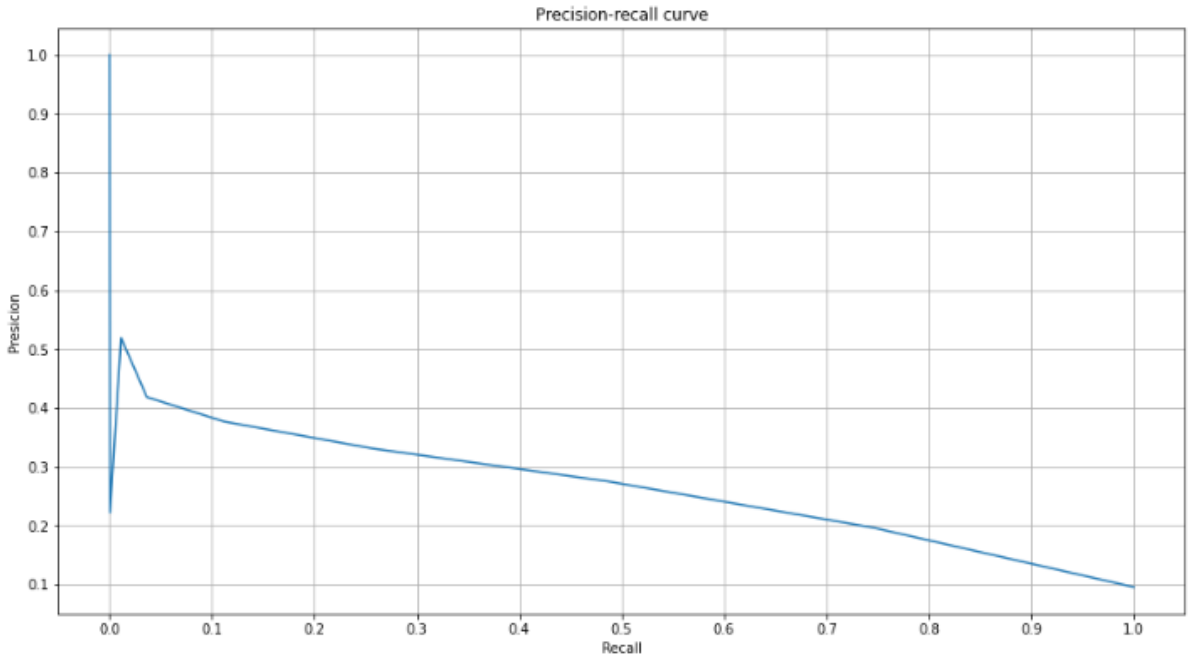
- **KNN**

```
from sklearn.metrics import euclidean_distances

class MyKNN(BaseEstimator, ClassifierMixin):
    def __init__(self, n_neighbors=5):
        self.n_neighbors = n_neighbors

    def fit(self, X, y):
        # check equal shapes
        X, y = check_X_y(X, y)

        # remember x and y
        self.X_ = X
        self.y_ = y

        return self

    def predict_proba(self, X):
        """
        Get P(y == 1 | X)
        """
        # check that X is a correct array
        X = check_array(X)

        y = np.ndarray((X.shape[0]))
        for i, elem in enumerate(X):
            # get distances and labels
```

```
            distances = euclidean_distances([elem], self.X_)[0]
            distances_with_labels = np.stack((distances, self.y_), axis=1)
            distances_with_labels.sort(axis=0)

            # get k nearest neighbors and count their labels
            k_neighbors = distances_with_labels[:self.n_neighbors]
            labels, counts = np.unique(k_neighbors[:, 1], return_counts=True)
            for j, label in enumerate(labels):
                if label == 1:
                    proba1 = counts[j] / self.n_neighbors
                    break
            else:
                # no neighbors with label = 1
                proba1 = 0
            y[i] = proba1

        return y

    def predict(self, X, threshold=0.5):
        return self.predict_proba(X) > threshold
```

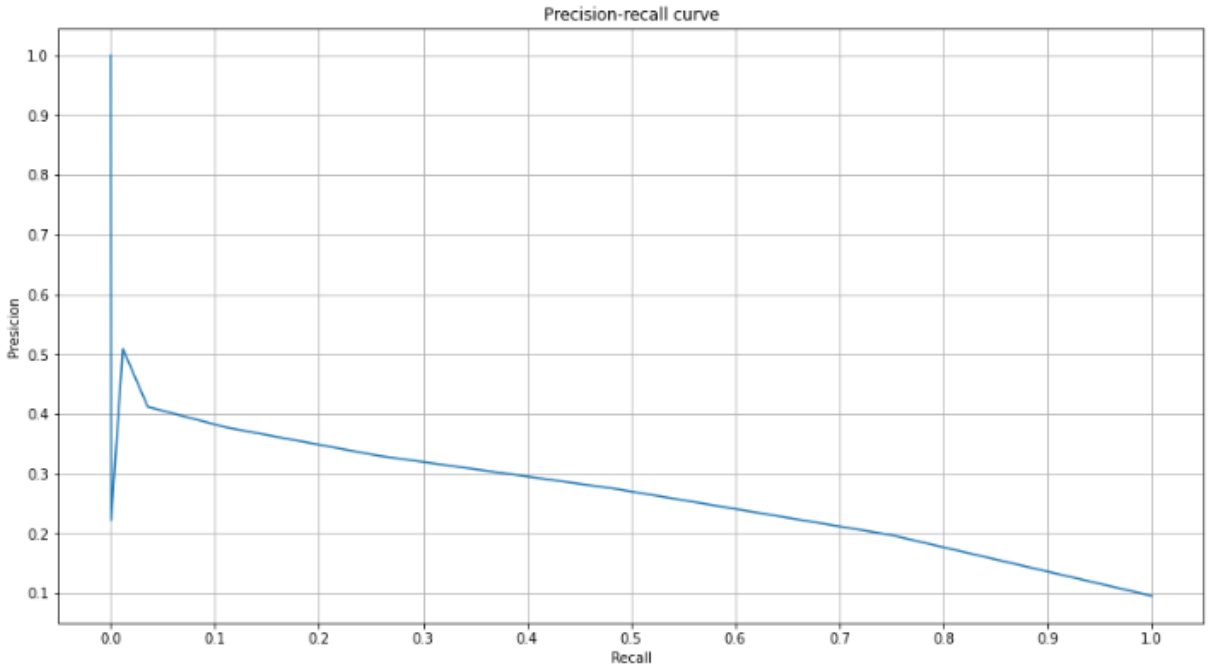
Получили следующие результаты:

```
Accuracy =  0.8973509933774835
Precision =  0.37682165163081194
Recall =  0.1118895528539048
ROC AUC =  0.7442772663080971
Confusion matrix:
[[44985   898]
 [ 4310   543]]
```



Для реализации из склерна получили следующие результаты:

```
Accuracy =  0.8974495427309996
Precision =  0.37745098039215685
Recall =  0.11106532042035853
ROC AUC =  0.7456294319833972
Confusion matrix:
[[44994   889]
 [ 4314   539]]
```



KNN очень долго работает, поэтому я не стал подбирать порог, чтобы получить реколл=0.7.

При реколле 0.7 имеем пресижен, примерно равный 0.21. Результаты моей модели и модели склерна оказались почти одинаковые.

- **Logistic Regression**

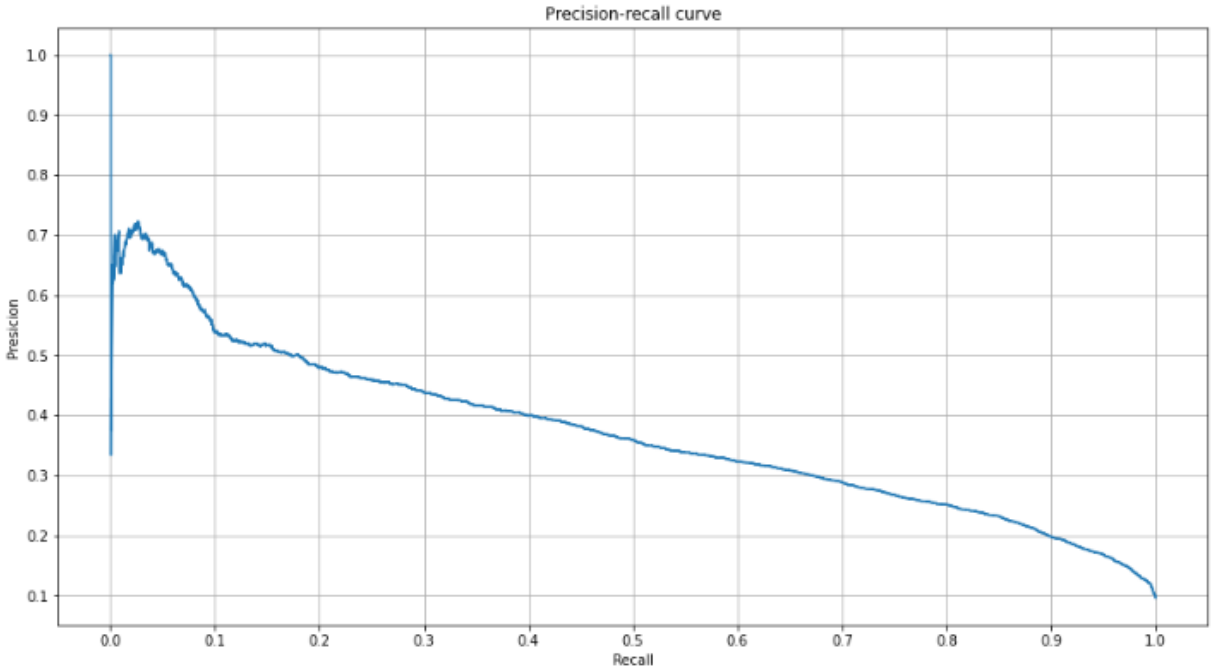
Буду реализовывать логистическую регрессию через стохастический градиентный спуск.

```
class MyLogisticRegression(BaseEstimator, ClassifierMixin):
    def __init__(self, epochs=10, lr=0.1, batch_size=256):
        self.w = None
        self.epochs = epochs
        self.lr = lr
        self.batch_size = batch_size

    def fit(self, X, y):
        X, y = check_X_y(X, y)
        n, k = X.shape

        if self.w is None:
            np.random.seed(0xDEAD)
            # weights
            self.w = np.random.randn(k + 1)

        X = np.concatenate((np.ones((n, 1)), X), axis=1)  # add bias as
feature
        for i in range(self.epochs):
            for j in range(0, len(X), self.batch_size):
```

```
                X_batch = X[j:j+self.batch_size]
                y_batch = y[j:j+self.batch_size]

                y_pred = self._predict_proba_internal(X_batch)
                self.w -= self.lr * self._get_gradient(X_batch, y_batch,
y_pred)

        return self

    def _get_gradient(self, X_batch, y_batch, y_pred):
        """
        Get gradient for logistic regression
        """
        gradient = X_batch.T @ (y_pred - y_batch)
        return gradient

    def predict_proba(self, X):
        X = check_array(X)

        n = X.shape[0]
        X = np.concatenate((np.ones((n, 1)), X), axis=1)
        return self._sigmoid(np.dot(X, self.w))

    def _predict_proba_internal(self, X):
        """
        This function is similar to predict_proba, but we don't concatenate
bias here.
        It is used for fitting.
        """
        return self._sigmoid(np.dot(X, self.w))

    def predict(self, X, threshold=0.5):
        return self.predict_proba(X) > threshold

    def _sigmoid(self, a):
        return 1. / (1 + np.exp(-a))
```

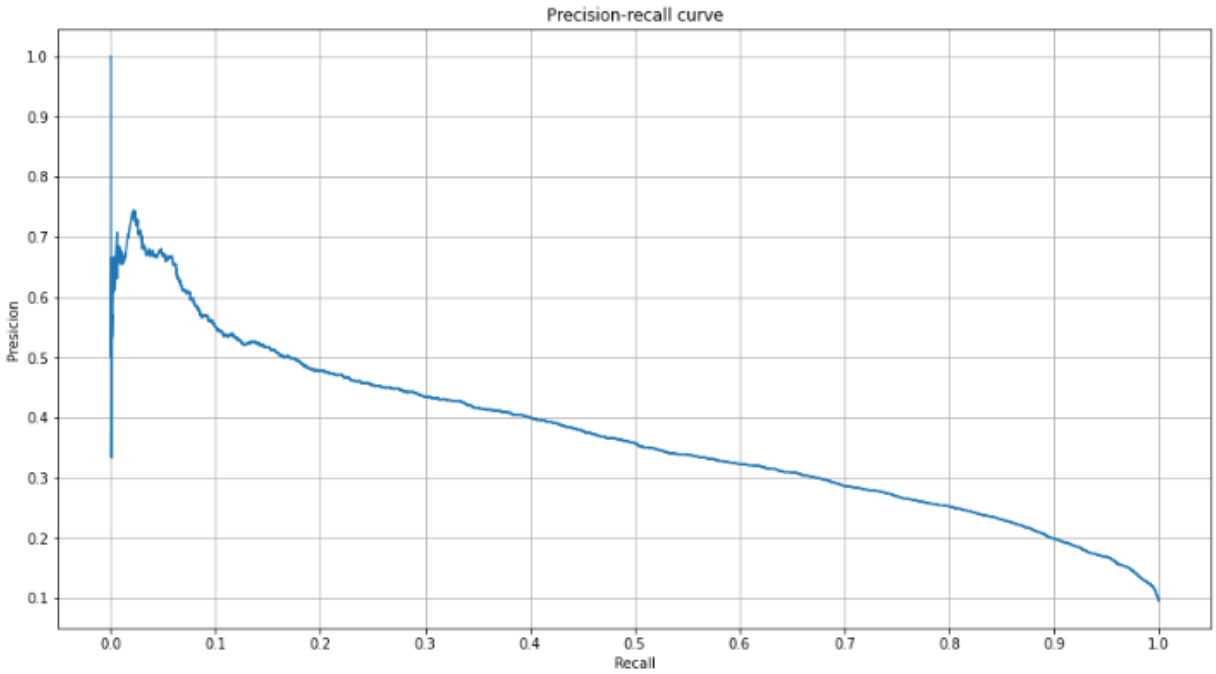
Получили следующие результаты:

```
Accuracy =  0.805424156417534
Precision =  0.2876723919113292
Recall =  0.7005975685143211
ROC AUC =  0.8487811092314154
Confusion matrix:
[[37464  8419]
 [ 1453  3400]]
```



Попробуем использовать реализацию из склерна. Добавим веса к классам (class_weight='balanced'). Получили следующие результаты:

```
Accuracy =  0.8040050457269
Precision =  0.28600252206809584
Recall =  0.7010096847310942
ROC AUC =  0.8490984305448075
Confusion matrix:
[[37390  8493]
 [ 1451  3402]]
```



Результаты моей регрессии совпали (почти) с библиотечной.

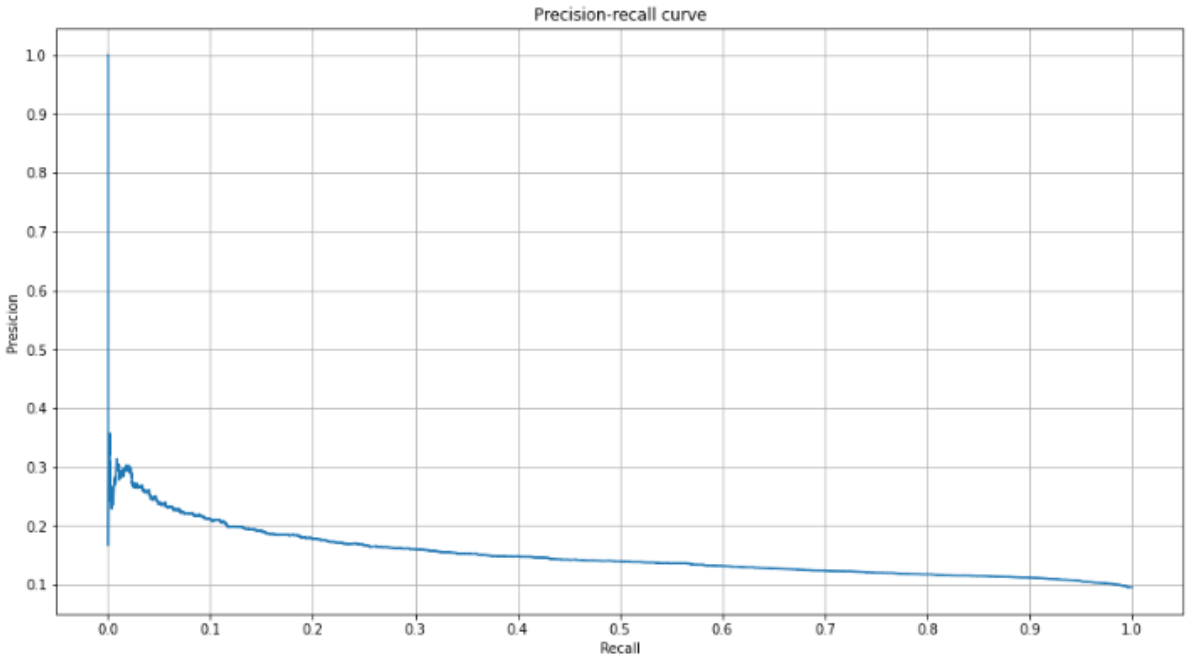
- **Naive Bayes**

Предположим, что все фичи имеют нормальное распределение.

```
class MyNaiveBayes(BaseEstimator, ClassifierMixin):
    def __init__(self):
        pass

    def fit(self, X, y):
        X, y = check_X_y(X, y)
        labels, counts = np.unique(y, return_counts=True)
        self.labels = labels
        # remember frequencies, means and standart deviations. we'll need
it in bayes theorem
        self.freq = np.array([cnt / y.shape[0] for cnt in counts])
        self.means = np.array([X[y == label].mean(axis = 0) for label in
labels])
        self.stds = np.array([X[y == label].std(axis = 0) for label in
labels])

        return self

    def predict_proba(self, X):
        X = check_array(X)
        y = np.zeros(X.shape[0])
        for i, x in enumerate(X):
            cur_freq = np.array(self.freq)
            for j in range(len(self.labels)):
                # P(label[j]|X)
                p = np.array([self._gaussian(self.means[j][k],
self.stds[j][k], x[k]) for k in range(X.shape[1])])
                cur_freq[j] *= np.prod(p)
            y[i] = cur_freq[1]
        return y

    def predict(self, X, threshold=0.5):
        return self.predict_proba(X) > threshold

    def _gaussian(self, mu, sigma, x0):
        # X ~ N(mu, sigma)
        # counts F(x0), where F is distribution function
        return np.exp(-(x0 - mu) ** 2 / (2 * sigma)) / np.sqrt(2.0 * np.pi
* sigma)
```

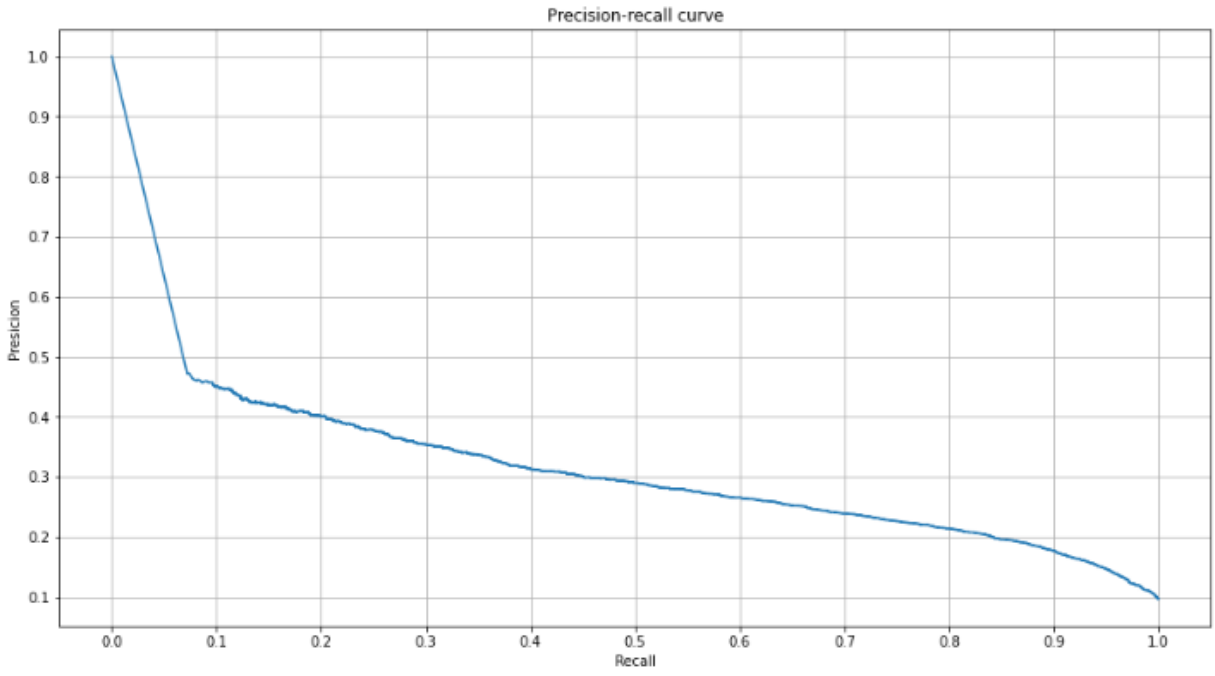
Получили следующие результаты:

```
Accuracy =  0.49032245348470516
Precision =  0.12319724474420607
Recall =  0.7076035441994643
ROC AUC =  0.6323003892406815
Confusion matrix:
[[21443 24440]
 [ 1419  3434]]
```



Попробуем библиотечную версию алгоритма.

```
Accuracy =  0.7574306212551246
Precision =  0.2386762024961436
Recall =  0.7014218009478673
ROC AUC =  0.8120096618766662
Confusion matrix:
[[35025 10858]
 [ 1449  3404]]
```



Видим, что библиотечный алгоритм справился с задачей лучше моего. Это возможно по нескольким причинам. Во-первых, в склерновской версии я использовал веса для классов,

чтобы побороть несбалансированность. Во-вторых, вероятно, в библиотечной версии вместо произведения вероятностей используется минус логарифм их суммы. В моей версии я решил пойти традиционным путем и использовать произведение (хотя это не всегда хорошо и может привести к проблемам из-за произведения бесконечно малых чисел с плавающей точкой).

- **SVM**

Попробуем реализовать линейный SVM. Будем использовать soft margin loss.

```
class MySVM(ClassifierMixin, BaseEstimator):
    def __init__(self, epochs=10, lr=0.1, alpha=0.1):
        self.w = None
        self.epochs = epochs
        self.lr = lr
        self.alpha = alpha  # regularization parameter

    def fit(self, X, y):
        X, y = check_X_y(X, y)
        y = np.where(y == 1, 1, -1)
        n, k = X.shape

        if self.w is None:
            np.random.seed(0xDEAD)
            # weights
            self.w = np.random.randn(k + 1)

        X = np.concatenate((np.ones((n, 1)), X), axis=1)  # add bias as
feature
        for i in range(self.epochs):
            for j, x in enumerate(X):
                margin = y[j] * np.dot(self.w, x)
                if margin >= 1:
                    self.w -= self.lr * self.alpha * self.w / self.epochs
                else:
                    self.w += self.lr * (y[j] * x - self.alpha * self.w /
self.epochs)
        return self

    def predict(self, X):
        # no predict proba here
        X = check_array(X)
        n, k = X.shape
        X = np.concatenate((np.ones((n, 1)), X), axis=1)
        y = np.ndarray((n))

        for i, elem in enumerate(X):
            prediction = np.dot(self.w, elem)
            if prediction > 0:
                y[i] = 1
            else:
                y[i] = 0
        return y


    def _hinge_loss(self, x, y):
        return max(0, 1 - y * np.dot(x, self.w))
```

```
    def _soft_margin_loss(self, x, y):
        return self._hinge_loss(x, y) + self.alpha * np.dot(self.w, self.w)
```

Проверим результаты:

```
Accuracy =  0.9043479974771366
Precision =  0.0
Recall =  0.0
Confusion matrix:
[[45883     0]
 [ 4853     0]]
```

К сожалению, метод опорных векторов не смог построить хорошую разделяющую линейную гиперплоскость. Поэтому ни одна строка данных не получила класс 1 от модели. Я проверил работу своего алгоритма на более простом датасете - там все работало. Значит для СВМа моя несбалансированная задача оказалась слишком сложной.

Попробуем использовать LinearSVC из sklearn.

```
Accuracy =  0.8185115105644907
Precision =  0.2952515279736718
Recall =  0.6470224603338142
Confusion matrix:
[[38388  7495]
 [ 1713  3140]]
```

Балансировка классов снова помогла, мы получили какой-то более-менее приемлемый результат.

5. Вывод

В данной работе я реализовал некоторые линейные алгоритмы машинного обучения - логистическую регрессию, SVM, kNN и наивного Байеса. Я попробовал обучить каждую из этих моделей на своем датасете и посчитал метрики для каждой.

К сожалению, из-за несбалансированности в данных и из-за линейной неразделимости я не смог получить хороших результатов. Хотя я и получал всегда высокий эккюраси, соотношение пресижена и реколла оставляло желать лучшего.

Наилучший результат я смог получить при использовании логистической регрессии. Там при реколле 0.7 я смог получить пресижен 0.28. Результат пока не очень хороший. Нужно использовать более сложные модели, чтобы улучшить его. Поэтому я хочу попробовать деревянные модели. Но это уже в следующей лабе..