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

Лабораторная работа №1 по курсу
«Машинное обучение»

Разработка линейных моделей

Студент: Гаптулхаков Руслан Рамилевич
Группа: М80 - 308Б -19
Дата: 17.05.2022
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 Вы должны организовать весь процесс предобработки, обучения и тестирования с помощью Pipeline Вы должны настроить гиперпараметры моделей с помощью кросс валидации, вывести и сохранить эти гиперпараметры в файл, вместе с обученными моделями;
3) Проделать аналогично с коробочными решениями; Для каждой модели получить оценки метрик: Confusion Matrix, Accuracy, Recall, Precision, ROC_AUC curve;
4) Проанализировать полученные результаты и сделать выводы о применимости моделей;
5) Загрузить полученные гиперпараметры модели и обученные модели в формате pickle на гит вместе с Jupyter Notebook ваших экспериментов.

2. Подготовка

В прошлой лабораторной работе мы подготовили данные для подачи их в модели машинного обучения. А именно: добавили новых признаков, закодировали категореальные признаки, удалили линейно зависимые признаки, отнормировали величины. Загрузим обработанные таблицы через pd.read_csv(). Теперь мы можем разделить нашу выборку на train/val части. Для этого воспользуемся функцией train_test_split из scikit-learn.

3. Метод k-ближайших соседей

Задаём метрику на пространстве и ищем первые k ближайших соседей к нашему классу. Присваиваем нашему объекту класс, который имеют больше всего соседей. Мы будем использовать обычную евклидову метрику.

```
from sklearn.metrics import euclidean_distances
class KNNClassifier(ClassifierMixin, BaseEstimator):
    def __init__(self, nb=5):
        self.nb = nb

    def fit(self, X, y):
        self.X_ = X
        self.y_ = y
        self.classes_ = np.unique(y)
        return self

    def predict(self, X):
        y = np.ndarray((X.shape[0],))
        for i, elem in enumerate(X):
            distances = euclidean_distances([elem], self.X_)[0]
            neighbors = np.argpartition(distances, kth = self.nb- 1)
            k_neighbors = neighbors[:self.nb]
            labels, cnts = np.unique(self.y_[k_neighbors], return_counts =
 True)
            y[i] = labels[cnts.argmax()]
        return y
```

4. Линейная регрессия

В работе реализованы два вида линейной регрессии. Первый способ использует градиентный спуск. Второй способ -- аналитическое решение. То есть мы можем точно вычислить формулу, по которой будет находится матрица весов.

```
class LinearRegressionClassifier(BaseEstimator, ClassifierMixin):
    def __init__(self, lr=0.005, iter=250):
        self.lr = lr
        self.iter = iter
        self.W = None
        self.b = None

    def fit(self, X, y):
        samples, features = X.shape
        self.W = np.zeros(features)
        self.b = 0
        for i in range(self.iter):
            pred = np.dot(X, self.W) + self.b
            dW = 1 / samples * np.dot(X.T, (pred - y))
            db = 1 / samples * np.sum(pred - y)
            self.W -= self.lr * dW
            self.b -= self.lr * db

    def predict(self, X):
        y_pred = np.dot(X, self.W) + self.b
        return  np.where(y_pred > 0, 1, 0)


class LinearRegressionClassifier(BaseEstimator, ClassifierMixin):
    def __init__(self, fit_intercept=True):
        self.fit_intercept = fit_intercept

    def fit(self, X, y):
        # bias
        if self.fit_intercept:
            X = np.hstack((X, np.ones((X.shape[0], 1))))
        self.w = np.linalg.inv(X.T @ X) @ X.T @ y
        return self

    def predict(self, X):
        if self.fit_intercept:
            X = np.hstack((X, np.ones((X.shape[0], 1))))
        y_pred = X @ self.w
        return np.where(y_pred > 0, 1, 0)

    def get_weights(self):
        return self.w
```

5. Логистическая регрессия

Логистическая регрессия описывает распределение признаков. Лосс функция – кросс-энтропия. На выходе получаем вероятности пренадлежнасти объекта к классу.

```
class LogisticRegressionClassifier(BaseEstimator, ClassifierMixin):
    def __init__(self, lr=0.05, max_iters=2500, fit_intercept=True):
        self.fit_intercept = fit_intercept
        self.lr = lr
        self.max_iters = max_iters

    def fit( self, X, Y ):
        if self.fit_intercept:
            X = np.hstack((X, np.ones((X.shape[0], 1))))
        self.m, self.n = X.shape
        # weight initialization
        self.W = np.zeros(self.n)
        self.X = X
        self.Y = Y
        # gradient descent learning
        for i in range(self.max_iters):
            self.update_weights()
        return self

    # Helper function to update weights in gradient descent
    def update_weights(self):
        z = self.X.dot(self.W)

        # sigmoid
        a = 1 / (1 + np.exp(-z))

        # calculate gradients
        grad = (a - self.Y.T)
        grad = np.reshape(grad, self.m)
        dW = np.dot(self.X.T, grad) / self.m
        db = np.sum(grad) / self.m

        # update weights
        self.W = self.W - self.lr * dW

        return self

    def predict(self, X) :
        if self.fit_intercept:
            X = np.hstack((X, np.ones((X.shape[0], 1))))
        z = X.dot(self.W)
        z = 1 / ( 1 + np.exp(-z))
        y = np.where(z > 0.5, 1, 0)
        return y
```

6. Метод опорных векторов

SVM или метод опорных векторов отличается от линейной регрессии тем, что мы ищем такую гиперплоскоть, которая максимально удалена от каждой группы классов. Таким образом мы решаем проблему, когда наш объект вблизи границы класса. Для этого достаточно помеять функцию ошибки.

$$F(M) = \max(0, 1 - M)$$

$$L(w, x, y) = \lambda \|w\|_2^2 + \sum_i \max(0, 1 - y_i \langle w, x_i \rangle)$$

$$\nabla_w L(w, x, y) = 2\lambda w + \sum_i \begin{cases} 0, & 1 - y_i \langle w, x_i \rangle \leq 0 \\ -y_i x_i, & 1 - y_i \langle w, x_i \rangle > 0 \end{cases}$$

```
class SVMClassifier(ClassifierMixin, BaseEstimator):
    def __init__(self, epoches = 200, lr = 0.005, alpha = 0.01, fit_interc
ept=True):
        self.epoches = epoches
        self.lr =  lr
        self.alpha = alpha
        self.fit_intercept = fit_intercept


    def update_weights(self):
        z = np.dot(self.X, self.W)
        dz = self.alpha * self.W
        for i, z_i in enumerate(z):
            if z_i * self.Y[i] < 1:
                dz -= self.X[i] * self.Y[i]
        self.W -= self.lr * dz

    def fit(self, X, y):
        #bias
        if self.fit_intercept:
            X = np.hstack((X, np.ones((X.shape[0], 1))))
        self.m, self.n = X.shape

        # weight initialization
        self.X = X
        self.Y = y
        self.W = np.zeros(self.n)

        for _ in range(self.epoches):
            self.update_weights()
        return self

    def predict(self, X):
        #bias
        if self.fit_intercept:
            X = np.hstack((X, np.ones((X.shape[0], 1))))
        return np.sign(np.dot(X, self.W))
```

7. Наивный байесовский классификатор

В основе наивного байесовского классификатора лежит теорема байеса. Теорема Байеса позволяет переставить местами причину и следствие. Зная с какой вероятностью причина приводит к некоему событию, эта теорема позволяет расчитать вероятность того что именно эта причина привела к наблюдаемому событию. Алгоритм называется наивным, потому что делается предположение условной независимости.

```
import math
class NaiveBayesClassifier(ClassifierMixin, BaseEstimator):
    def __init__(self):
       pass

    def fit(self, X, y):
        self.X = X
        self.y = y
        labels, counts = np.unique(self.y, return_counts = True)
        self.labels = labels
        self.freq = np.array([i / self.y.shape[0] for i in counts])
        self.means = np.array([self.X[self.y == i].mean(axis = 0) for i in
 labels])
        self.stds = np.array([self.X[self.y == i].std(axis = 0) for i in l
abels])
        return self

    def gaussian(self, mu, sigma, x0):
        return np.exp(-
(x0 - mu) ** 2 / (2 * sigma)) / np.sqrt(2.0 * math.pi * sigma)

    def predict(self, X):
        res = np.zeros(X.shape[0])
        for i, x_i  in enumerate(X):
            freq = np.array(self.freq)
            for j, label_j in enumerate(self.labels):
                p_x_cond_y = np.array([self.gaussian(self.means[j][k], sel
f.stds[j][k], x_i[k]) for k in range(X.shape[1])])
                freq[j] *= np.prod(p_x_cond_y)
            res[i] = np.argmax(freq)
        return res
```

8. Подбор гиперпараметров

Для наглядной проверки работ модели будем строить `confusion matrix` для нашего и коробочного алгоритмов. Для этого нам понадобится функция.

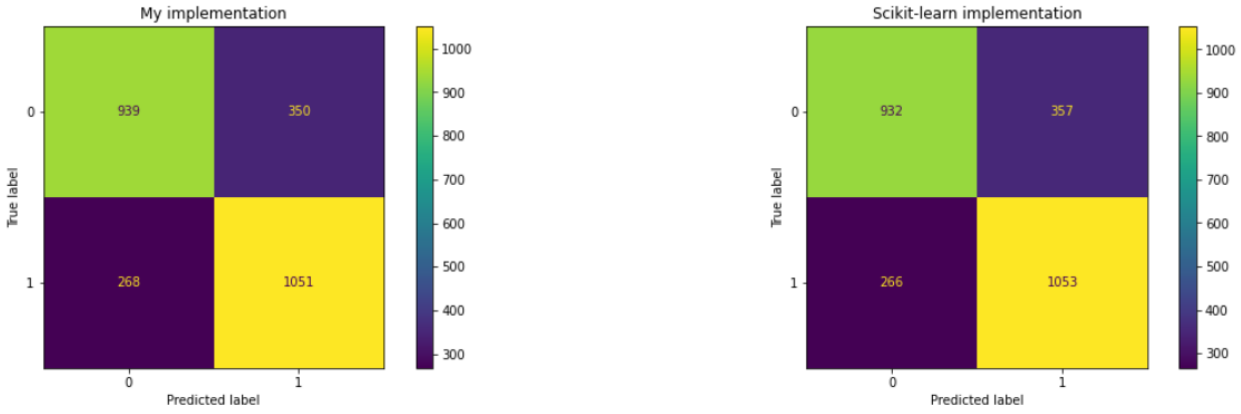
```
def scores(y_pred1, y_pred2, y):
    print("Accuracy:", accuracy_score(y, y_pred1))
    print("Accuracy sklearn model:", accuracy_score(y, y_pred2))
    print("Recall:", recall_score(y, y_pred1))
    print("Recall sklearn model:", accuracy_score(y, y_pred2))
    print("Precision:", precision_score(y, y_pred))
    print("Precision sklearn model:", accuracy_score(y, y_pred2))
    figure = plt.figure(figsize = (20, 5))
    matr1 = confusion_matrix(y, y_pred1)
    matr2 = confusion_matrix(y, y_pred2)
    ax1 = plt.subplot(1, 2, 1)
    ax2 = plt.subplot(1, 2, 2)
    ax1.set_title("My implementation")
    ax2.set_title("Scikit-learn implementation")
    ConfusionMatrixDisplay(matr1).plot(ax = ax1)
    ConfusionMatrixDisplay(matr2).plot(ax = ax2)
    plt.show()
```

Также будем выводить результаты различных известных метрик.

9. Результаты работ алгоритмов

- **K-ближайших соседей**

```
Accuracy: 0.7630368098159509
Accuracy sklearn model: 0.7611196319018405
Recall: 0.7968157695223654
Recall sklearn model: 0.7611196319018405
Precision: 0.6350858927641854
Precision sklearn model: 0.7611196319018405
```

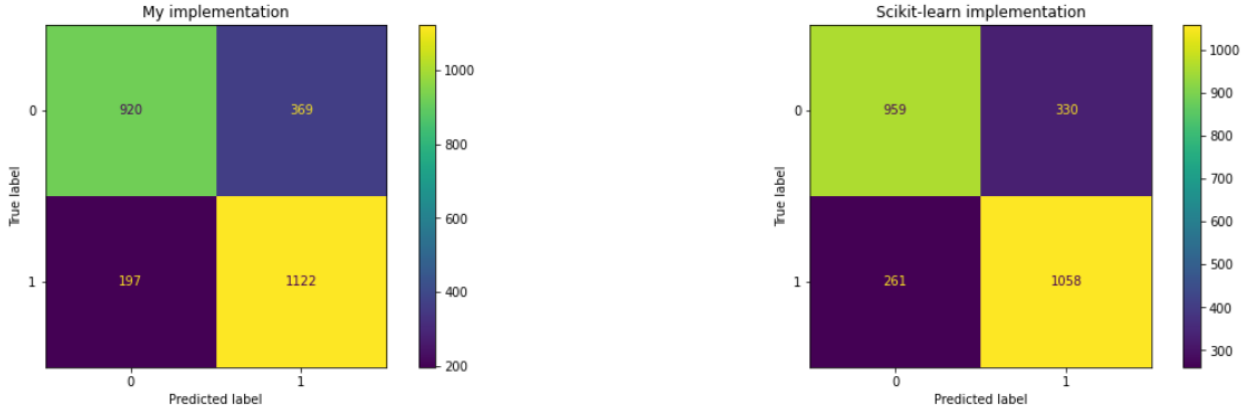



```
Лучшие гиперпараметры модели: {'KNN__nb': 7}
Лучший счёт модели: 0.7682826622843056
```

KNN довольно хорошо справляется с поставленной задачей.

- **Логисточеская ркгркссия**

```
Accuracy: 0.7829754601226994
Accuracy sklearn model: 0.7733895705521472
Recall: 0.8506444275966641
Recall sklearn model: 0.7733895705521472
Precision: 0.6712860310421286
Precision sklearn model: 0.7733895705521472
```

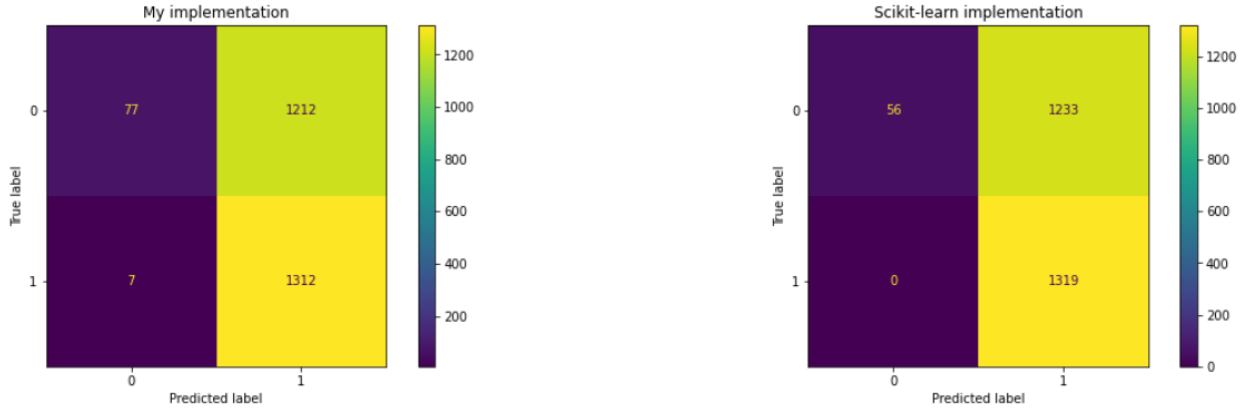



```
Лучшие гиперпараметры модели: {'LogReg__fit_intercept': True, 'LogReg__lr': 0.01, 'LogReg__max_iters': 2000}
Лучший счёт модели: 0.790797041906327
```

Результаты логистических регрессий схожи. Модели хорошо справляются с задачей.

- **Линейная регрессия**

```
Accuracy: 0.5325920245398773
Accuracy sklearn model: 0.5272239263803681
Recall: 0.9946929492039424
Recall sklearn model: 0.5272239263803681
Precision: 0.6712860310421286
Precision sklearn model: 0.5272239263803681
```

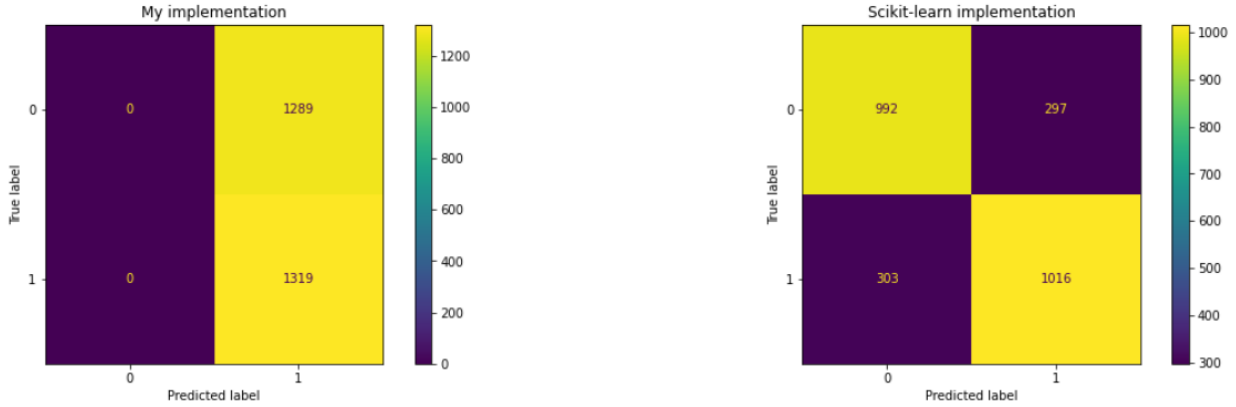



```
Лучшие гиперпараметры модели: {'Linear__fit_intercept': False}
Лучший счёт модели: 0.5293344289235826
```

Линейная модель совсем не справляется с задаче. Можно сделать вывод, что наши классы не имеют линейную зависимость от признаков.

- **Метод опорных векторов**

```
Accuracy: 0.5057515337423313
Accuracy sklearn model: 0.7699386503067485
Recall: 1.0
Recall sklearn model: 0.7699386503067485
Precision: 0.6712860310421286
Precision sklearn model: 0.7699386503067485
```

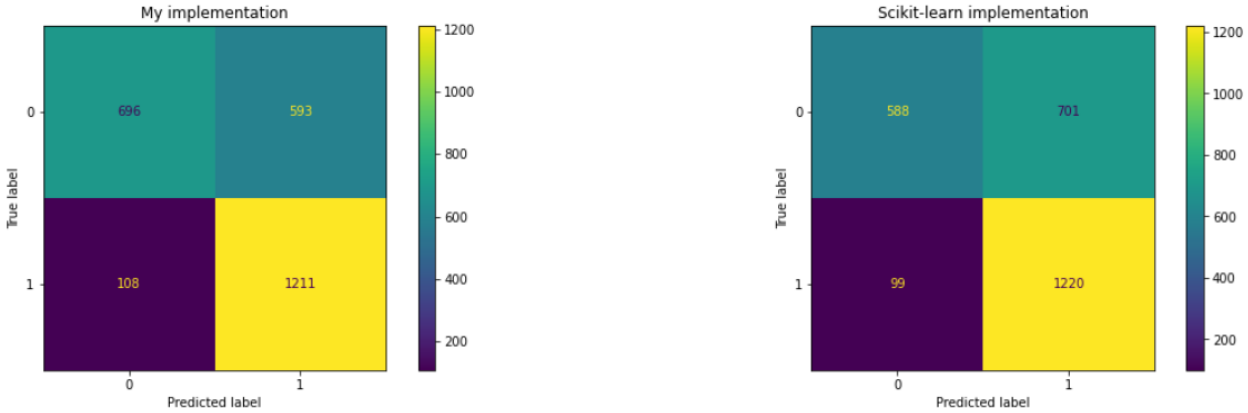



```
Лучшие гиперпараметры модели: {'SVM__alpha': 2.0, 'SVM__epoches': 10, 'SVM__lr': 0.01}
Лучший счёт модели: 0.5027115858668857
```

Коробочный SVM получает хорошие результаты классификции. Мой SVM не хочет предсказывать объекты 0 класса.

- **Наивный байесовский классификатор**

```
Accuracy: 0.7312116564417178
Accuracy sklearn model: 0.6932515337423313
Recall: 0.9181197877179682
Recall sklearn model: 0.6932515337423313
Precision: 0.6712860310421286
Precision sklearn model: 0.6932515337423313
```



Моя модель Байесовского классификатора справляется лучше коробочного.

1. Вывод

В данной лабораторной работе я изучил линейные модели класического машинного обучения. Были реализованы все алгоритмы поставленные в задаче. Для каждого алгоритма были подобраны лучшие гиперпараметры. Для этого была использована функция GridSearchCV. Все модели, кроме линейной, дали результаты около 0.77 это довольно

хороший результат. Я думаю, можно существенно поднять accuracy, если мы дополнительно поработаем с данными. А именно выделим больше признаков.