Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра информатики

ЛАБОРАТОРНАЯ РАБОТА №10

«Градиентный бустинг»

Студент А. Ю. Омельчук

Преподаватель М. В. Стержанов

Минск 2019

ХОД РАБОТЫ

Задание.

Для выполнения задания используйте набор данных boston из библиотеки sklearn: https://scikit-learn.org/stable/datasets/index.html#boston-dataset

1. Загрузите данные с помощью библиотеки sklearn.
2. Разделите выборку на обучающую (75%) и контрольную (25%).
3. Заведите массив для объектов DecisionTreeRegressor (они будут использоваться в качестве базовых алгоритмов) и для вещественных чисел (коэффициенты перед базовыми алгоритмами).
4. В цикле обучите последовательно 50 решающих деревьев с параметрами max_depth=5 и random_state=42 (остальные параметры - по умолчанию). Каждое дерево должно обучаться на одном и том же множестве объектов, но ответы, которые учится прогнозировать дерево, будут меняться в соответствие с отклонением истинных значений от предсказанных.
5. Попробуйте всегда брать коэффициент равным 0.9. Обычно оправдано выбирать коэффициент значительно меньшим - порядка 0.05 или 0.1, но на стандартном наборе данных будет всего 50 деревьев, возьмите для начала шаг побольше.
6. В процессе реализации обучения вам потребуется функция, которая будет вычислять прогноз построенной на данный момент композиции деревьев на выборке X. Реализуйте ее. Эта же функция поможет вам получить прогноз на контрольной выборке и оценить качество работы вашего алгоритма с помощью mean_squared_error в sklearn.metrics.
7. Попробуйте уменьшать вес перед каждым алгоритмом с каждой следующей итерацией по формуле 0.9 / (1.0 + i), где i - номер итерации (от 0 до 49). Какое получилось качество на контрольной выборке?
8. Исследуйте, переобучается ли градиентный бустинг с ростом числа итераций, а также с ростом глубины деревьев. Постройте графики. Какие выводы можно сделать?
9. Сравните качество, получаемое с помощью градиентного бустинга с качеством работы линейной регрессии. Для этого обучите LinearRegression из sklearn.linear_model (с параметрами по умолчанию) на обучающей выборке и оцените для прогнозов полученного алгоритма на тестовой выборке RMSE.

Результат выполнения:

1. Код выгрузки данных:

```
boston = datasets.load_boston()
X, Y = boston.data, boston.target
```

2. Код реализации:

```
# ###################### QUITE IMPORTANT #########################
# overall MSE really depends on factor how do we split our data
# so changing `random_state` may lead to various results.
# In order to have the same results I set seed to value equal to 51
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.25,
random_state=51)
```

3-6. Код реализации:

```
mocked_coefficient_quality = check_quality(X_train, y_train, X_test, y_test,
mock_coefficient)()
print(mocked_coefficient_quality)
```

Результат выполнения:

```
5.569226762770745
```

7. Код реализации:

```
sequences_coefficient_quality = check_quality(X_train, y_train, X_test, y_test,
calculate_coefficient)()
print(sequences_coefficient_quality)
```

Результат выполнения:

```
5.3607371501561625
```

Ошибка ументшилась, однако относительно несущественно. Посмотрим как будет меняться ошибка в зависимости от глубины и количества деревьев. Стоит сразу отметить, что реализация в следующем пункет была сделана в несколько этапов. Сначал я использовал градиентный бустинг, который был «самописный» (тот, что был реализован в первых пунктах с динамическим коэффициентом). Однако, после этого я решил попробовать использовать библиотечную реализацию, и результаты уже были совершенно другими и более ожидаемыми.

8. Код реализации:

```
        # since Intel i7 6700HQ has 8 threads
        trees_number = [50, 100, 150, 200, 250, 300, 400, 500]
        params = []

        for i in trees_number:
            params.append(DecisionTreeParams(X_train, y_train, X_test, y_test,
calculate_coefficient, i, 5))

        pool = Pool(len(trees_number))
        # change job_gb to job for getting my own implementation of gradient boosting
        errors = pool.map(job_gb, params)


        depth = [2, 4, 6, 8, 10, 13, 17, 20]
        params = []

        for i in depth:
            params.append(DecisionTreeParams(X_train, y_train, X_test, y_test,
calculate_coefficient, 50, i))

        # change job_gb to job for getting my own implementation of gradient boosting
        errors = pool.map(job_gb, params)
```

Результат выполнения:

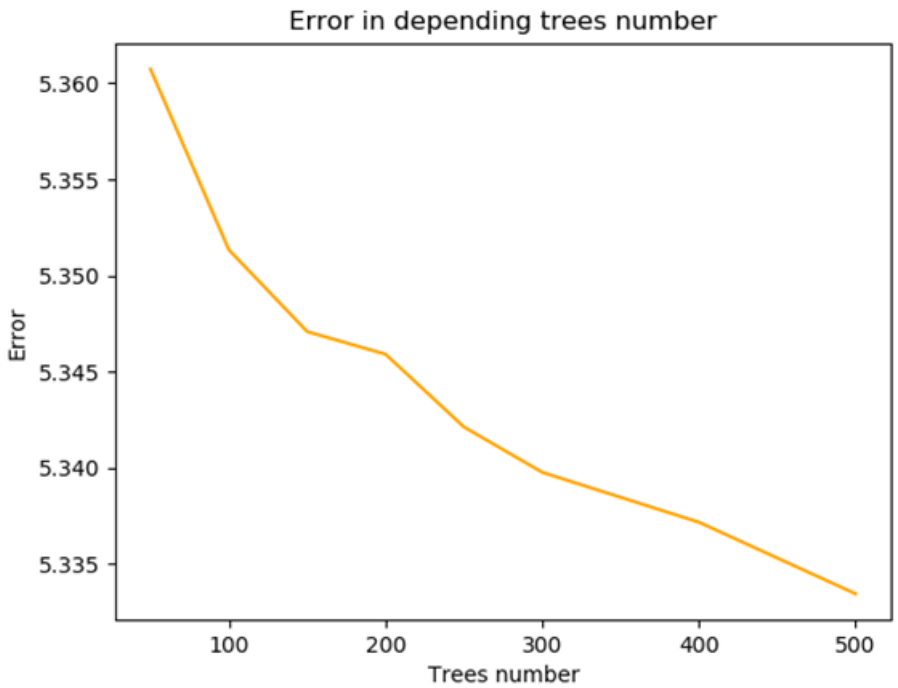

Рисунок 1 – график зависимости ошибки от количества деревьев (не библиотечная реализация)

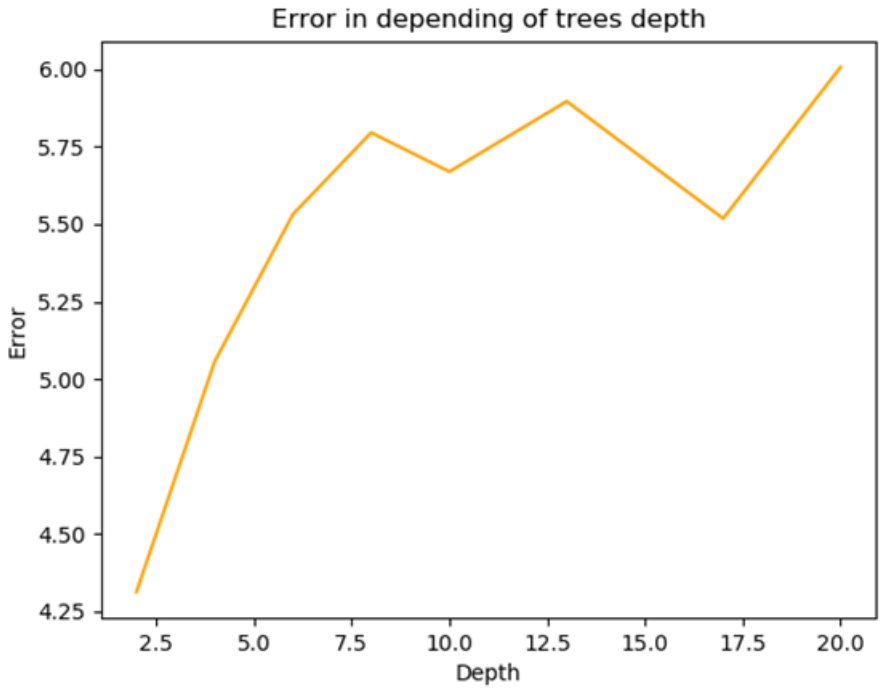

Рисунок 2 – Зависимость ошибки от глубины дерева (не библиотчечная реализация)

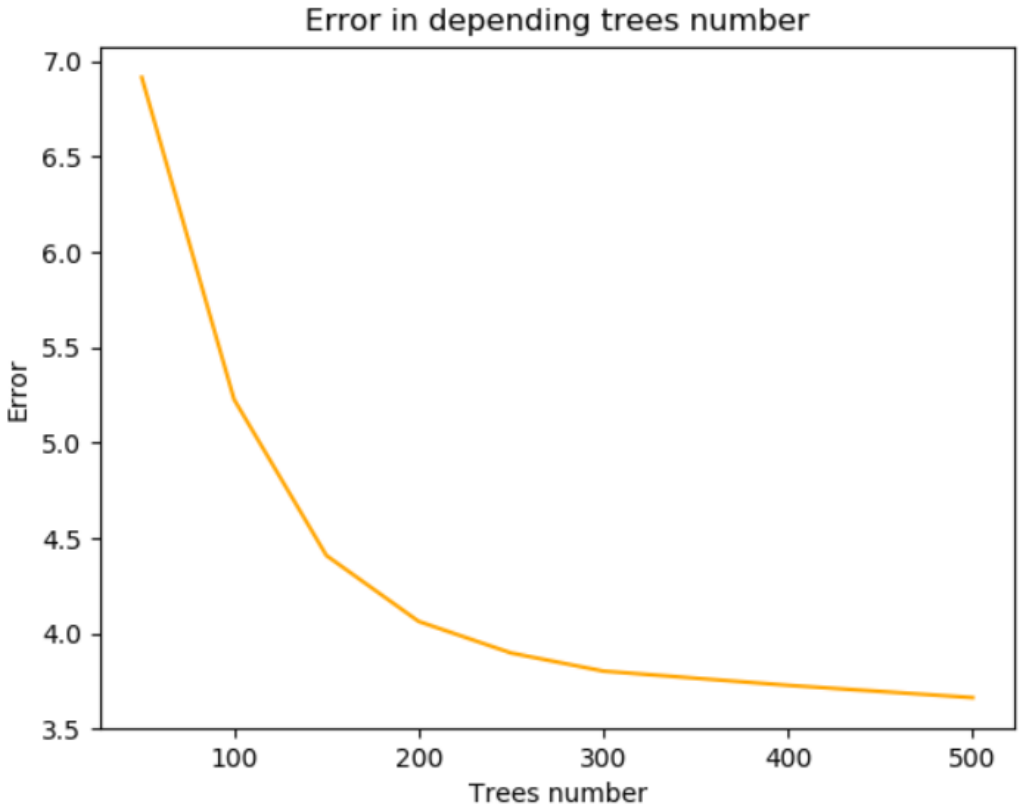

Рисунок 3 – зависимость ошибки от количества деревьев (библиотечная реализация)

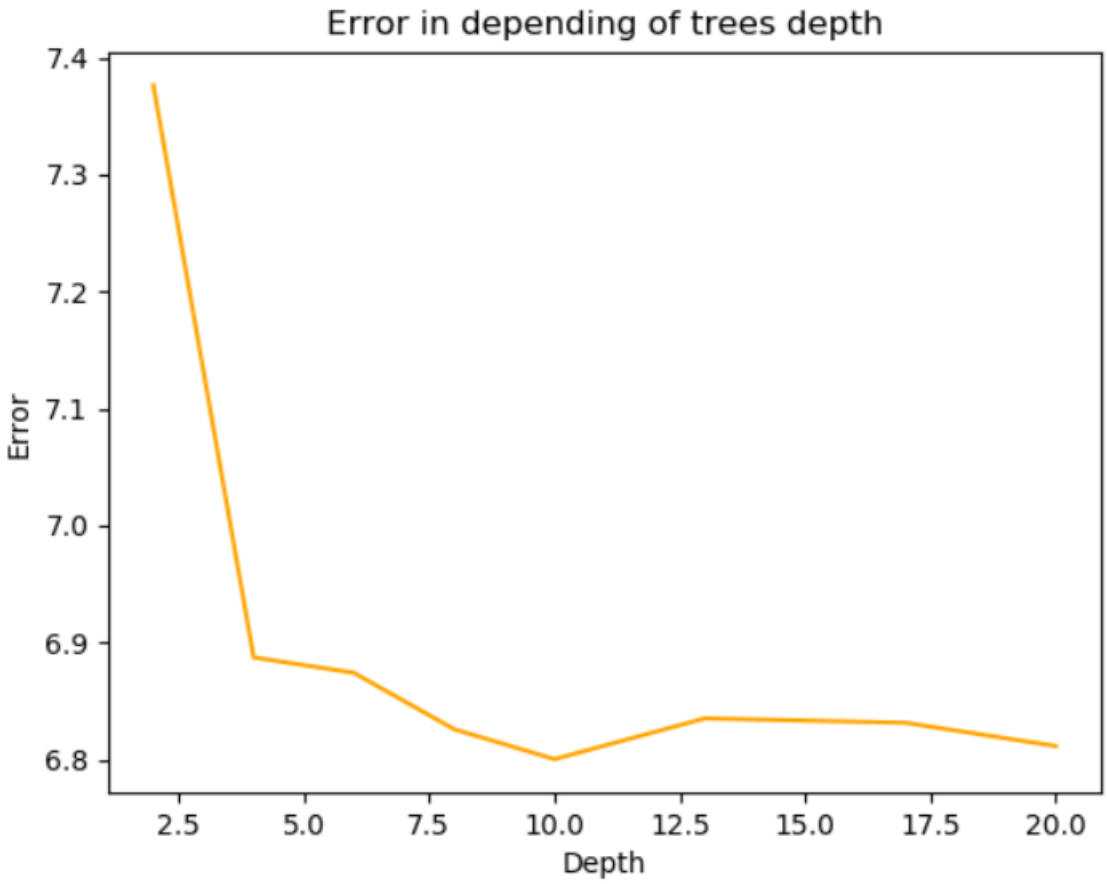

Рисунок 4 – зависимость ошибки од глубины дерева (библиотечная реализация)

На двух графиках с библиотечной реализацией видно, что ошибка уменьшается с ростом глубины дерева и увеличением количества деревьев. На графике после глубины дерева 10 видно, что график опять начинает немного расти, что свидетельствует о переобучении модели (при реализации градиентного бустинга через xgiboost, как изначально рекомендовали делать в курсе от Яндекса, ошибка на этом моменте моментально «взлетает», что опять-таки свидетельствует о переобучении).

9. Код реализации:

```
reg = LinearRegression().fit(X_train, y_train)
pred = reg.predict(X_test)
print('Linear regression MSE: ', np.sqrt(mean_squared_error(y_test, pred)))
```

Результат выполнения:

```
('Linear regression MSE: ', 4.863502400380662)
```

Видно, что ошибка меньше, чем у начальной реализации градиентного бустинга. Однако очевидно, что, скорее всего, линейная регресси не сможет восстановить сложности всех данных (хотя добавление полиномиальных фич в линейную регрессию, возможно, уменьшило бы итоговую ошибку). В данном случае я бы остановился на алгоритме градиентного бустинга, добавив больше деревьев и увеличив его глубину.

Программный код:

```
    """
    File -> Settings -> Tools -> Python Scientific -> uncheck mark
    Run with python 3.7.5 (64 bit)
    """

    from __future__ import division
    from sklearn import datasets
    from sklearn.tree import DecisionTreeRegressor
    from sklearn.metrics import mean_squared_error
    from sklearn.model_selection import train_test_split
    import numpy as np
    import matplotlib.pyplot as plt
    from goto import with_goto
    from sklearn.linear_model import LinearRegression
    from multiprocessing import Pool
    from sklearn import ensemble


    def gradient(Y, X, base_algorithms_list, coefficients_list):
        return Y - predict(X, base_algorithms_list, coefficients_list)


    def predict(X, base_algorithms_list, coefficients_list):
        return [sum([coeff * algo.predict([x])[0] for algo, coeff in zip(base_algorithms_list,
coefficients_list)])
                for x in X]


    def check_quality(X_train, y_train, X_test, y_test, get_coefficient):
        def get_model(trees_number=50, depth=5):
            print(trees_number, depth)
            # 4
            base_algorithms_list = []
            coefficients_list = []

            # 5
            for i in range(0, trees_number):
                # create new algorithm
                rg = DecisionTreeRegressor(random_state=42, max_depth=depth)
                # fit algo in train dataset and new target
                rg.fit(X_train,      gradient(y_train,      X_train,      base_algorithms_list,
coefficients_list))
                # append results
                base_algorithms_list.append(rg)
                # ======= 6 =======
                coefficients_list.append(get_coefficient(i))

            # 7
            pred = predict(X_test, base_algorithms_list, coefficients_list)
            print(np.sqrt(mean_squared_error(y_test, pred)))
            return np.sqrt(mean_squared_error(y_test, pred))

        return get_model


    def mock_coefficient(i):
```

```
        return 0.9


    def calculate_coefficient(i):
        return 0.9 / (1.0 + i)


    class DecisionTreeParams():
        def __init__(self, X_train, y_train, X_test, y_test, coefficients, number_trees, depth):
            """Constructor"""
            self.X_train = X_train
            self.y_train = y_train
            self.X_test = X_test
            self.y_test = y_test
            self.coefficients = coefficients
            self.number_trees = number_trees
            self.depth = depth

        def worker(self):
            return check_quality(self.X_train, self.y_train, self.X_test, self.y_test,
self.coefficients)(self.number_trees, self.depth)

        def run_gb(self):
            params = {'n_estimators': self.number_trees, 'max_depth': self.depth,
'min_samples_split': 2,
                      'learning_rate': 0.01, 'loss': 'ls'}
            clf = ensemble.GradientBoostingRegressor(**params)
            clf.fit(self.X_train, self.y_train)
            return np.sqrt(mean_squared_error(self.y_test, clf.predict(self.X_test)))


    def job(A):
        return A.worker()


    def job_gb(A):
        return A.run_gb()


    @with_goto
    def main():
        goto .task
        label .task
        # 1
        boston = datasets.load_boston()
        X, Y = boston.data, boston.target

        # 2
        # ######################## QUITE IMPORTANT ############################
        # overall MSE really depends on factor how do we split our data
        # so changing `random_state` may lead to various results.
        # In order to have the same results I set seed to value equal to 51
        X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.25,
random_state=51)

        # 3
        mocked_coefficient_quality = check_quality(X_train, y_train, X_test, y_test,
mock_coefficient)()
        print(mocked_coefficient_quality)
```

```
        # 8
        sequences_coefficient_quality = check_quality(X_train, y_train, X_test, y_test,
calculate_coefficient)()
        print(sequences_coefficient_quality)

        # 9
        # since Intel i7 6700HQ has 8 threads
        trees_number = [50, 100, 150, 200, 250, 300, 400, 500]
        params = []

        for i in trees_number:
            params.append(DecisionTreeParams(X_train, y_train, X_test, y_test,
calculate_coefficient, i, 5))

        pool = Pool(len(trees_number))
        # change job_gb to job for getting my own implementation of gradient boosting
        errors = pool.map(job_gb, params)

        plt.plot(trees_number, errors, color='orange')
        plt.xlabel('Trees number')
        plt.ylabel('Error')
        plt.title('Error in depending trees number')
        plt.show()

        depth = [2, 4, 6, 8, 10, 13, 17, 20]
        params = []

        for i in depth:
            params.append(DecisionTreeParams(X_train, y_train, X_test, y_test,
calculate_coefficient, 50, i))

        # change job_gb to job for getting my own implementation of gradient boosting
        errors = pool.map(job_gb, params)

        plt.plot(depth, errors, color='orange')
        plt.xlabel('Depth')
        plt.ylabel('Error')
        plt.title('Error in depending of trees depth')
        plt.show()

        # 10
        reg = LinearRegression().fit(X_train, y_train)
        pred = reg.predict(X_test)
        print('Linear regression MSE: ', np.sqrt(mean_squared_error(y_test, pred)))


    if __name__ == '__main__':
        main()
```