**Лабораторная работа 6.**

Разберёмся с **ансамблями алгоритмов** и со **случайным лесом**. Рассмотрим данные о сотрудниках компании, где указывается, ушёл сотрудник или нет. (HR Dataset)

Сделаем**базовую предобработку данных**: удалим признак, который отвечает за идентификатор пользователя, как нерепрезентативный признак.

**import** pandas **as** pd

**import** numpy **as** np

**import** matplotlib.pyplot **as** plt

**from** sklearn.model\_selection **import** cross\_val\_score, train\_test\_split

**from** sklearn.ensemble **import** BaggingClassifier, RandomForestClassifier

**from** sklearn.tree **import** DecisionTreeClassifier

**from** sklearn.linear\_model **import** LogisticRegression

**from** sklearn.preprocessing **import** StandardScaler

df = pd.read\_csv('HR-dataset.csv')

np.random.seed(42)

%matplotlib inline

%config InlineBackend.figure\_format = 'retina'

target = 'left'

features = df.columns.drop(target)

features = features.drop('empid') # Удалим идентификатор пользователя как нерепрезентативный признак

print(features)

X, y = df[features].copy(), df[target]

Index(['satisfaction\_level', 'last\_evaluation', 'number\_project', 'average\_montly\_hours', 'time\_spend\_company', 'Work\_accident', 'promotion\_last\_5years', 'dept', 'salary'], dtype='object')

Заменим идентификатор отдела, к которому относился сотрудник, на количество людей в отделе, а зарплату — на ординальную категорию. Масштабируем признаки для последующего сравнения результатов:

salary\_ordinals = {'low': 1, 'medium': 2, 'high': 3}

X['dept'] = X['dept'].apply(X['dept'].value\_counts().**get**)

X['salary'] = X['salary'].apply(salary\_ordinals.**get**)

scaler = StandardScaler()

X = pd.DataFrame(data=scaler.fit\_transform(X), columns=X.columns)

В дальнейшем будем оценивать качество модели на кросс-валидации на пяти фолдах при помощи точности (accuracy).

def estimate\_accuracy(clf, X, y, cv=5):

return cross\_val\_score(clf, X, y, cv=5, scoring='accuracy').mean()

Посмотрим на то, как работает бэггинг над решающими деревьями.

**Бэггинг** (*bagging*, сокр. от *bootstrap aggregating*)  — метод построения композиции алгоритмов, в котором каждый алгоритм строится независимо от других на подвыборках обучающей выборки. Итоговый алгоритм принимает решения посредством голосования среди всех алгоритмов (возвращается самый частый ответ).

Посмотрим на точность одного дерева:

tree = DecisionTreeClassifier(max\_depth=30)

print("Decision tree:", estimate\_accuracy(tree, X, y))

Decision tree: 0.9450045314500757

Проведём бэггинг: для этого достаточно обернуть исходный классификатор в *BaggingClassifier*.

bagging\_trees = BaggingClassifier(tree)

print("Decision tree bagging:", estimate\_accuracy(bagging\_trees, X, y))

Decision tree bagging: 0.9745837353643367

Композиция отдельных деревьев показывает себя лучше, чем одно дерево. Структура дерева серьёзно зависит от обучающей выборки. Это значит, что если немного изменить обучающую выборку, то дерево сильно изменится. Бэггинг идеально подходит в этом случае, поскольку композиция алгоритмов при помощи голосования работает наилучшим образом, когда модели различны.

Увеличить различность построенных деревьев можно, указав параметры *max\_features* и *max\_depth*.

random\_tree = DecisionTreeClassifier(max\_features=int(np.sqrt(len(features))), max\_depth=30)

print("Random tree:", estimate\_accuracy(random\_tree, X, y))

Random tree: 0.9540713833978115

bagging\_random\_trees = BaggingClassifier(random\_tree)

print("Random tree bagging:", estimate\_accuracy(bagging\_random\_trees, X, y))

Random tree bagging: 0.9791065929706239

Именно так внутри и работает так называемый **случайный лес** (*Random Forest*): он обучает набор деревьев (параметр *n\_esimators*), каждое из которых обучается на подмножестве признаков (*Random Subspaces*) и на подмножестве объектов (*bootstrap*). То есть случайный лес получается случайным по двум этим параметрам, а ответы агрегируются при помощи голосования.

Стандартная эвристика: в **задаче классификации** брать квадратный корень числа признаков, а в **задаче регрессии** — треть числа признаков.

random\_forest = RandomForestClassifier(

n\_estimators=100,

n\_jobs=-1,

max\_features=int(np.sqrt(len(features))),

max\_depth=30)

print("Random Forest:", estimate\_accuracy(random\_forest, X, y))

Random Forest: 0.9829834277014811

Ещё одно преимущество использования бэггинга для агрегации моделей — получение **оценки**работы классификатора без дополнительного проведения **кросс-валидации** при помощи **out-of-bag score**. Это метод вычисления произвольной оценки качества во время обучения бэггинга. Для подсчёта требуется указать параметр **oob\_score = True,**что имеет смысл при достаточном количестве деревьев.

random\_forest = RandomForestClassifier(

n\_estimators=100,

max\_features=int(np.sqrt(len(features))),

max\_depth=30,

oob\_score=True,

n\_jobs=-1

)

random\_forest.fit(X, y)

random\_forest.oob\_score\_.mean()

0.9929995333022201

Метод бэггинга можно применять к **произвольным алгоритмам**, например, к логистической регрессии.

lr = LogisticRegression(solver='saga', max\_iter=200)

lr.fit(X, y)

print("LR:", estimate\_accuracy(lr, X, y))

LR: 0.44172459802488306

random\_logreg = BaggingClassifier(

lr,

n\_estimators=10,

n\_jobs=-1,

random\_state=42

)

print("Bagging for LR:", estimate\_accuracy(random\_logreg, X, y))

Bagging for LR: 0.4365451576623669

В её случае он не так сильно повышает качество, поскольку линейные модели не так сильно зависят от состава обучающей выборки. Попробуем убрать часть признаков.

random\_logreg = BaggingClassifier(

lr,

n\_estimators=10,

n\_jobs=-1,

max\_features=0.5,

random\_state=42

)

print("Bagging for LR:", estimate\_accuracy(random\_logreg, X, y))

Bagging for LR: 0.28601355351119234

В случае логистической регрессии повышение разнообразности моделей не даёт такого прироста, как с деревьями, поскольку модели сильно теряют в качестве. Случайный лес на примере нашей задачи справляется лучше.

**Сравнение**логистической регрессии и случайного леса:

def plot\_predictions(X, y, clf, proba=False, points\_size=7, xlabel='x', ylabel='y'):

"""Fits the classifier on the data (X, y) and plots the result on a 2-D plane."""

def get\_grid(data):

x\_std, y\_std = data.std(axis=0)

x\_min, x\_max = data[:, 0].min() - x\_std / 2, data[:, 0].max() + x\_std / 2

y\_min, y\_max = data[:, 1].min() - y\_std / 2, data[:, 1].max() + y\_std / 2

return np.meshgrid(np.linspace(x\_min, x\_max, num=200),

np.linspace(y\_min, y\_max, num=200))

clf.fit(X, y)

xx, yy = get\_grid(X)

if proba:

predicted = clf.predict\_proba(np.c\_[xx.ravel(), yy.ravel()])[:, 1].reshape(xx.shape)

else:

predicted = clf.predict(np.c\_[xx.ravel(), yy.ravel()]).reshape(xx.shape)

plt.figure(figsize=(10.0, 10.0))

plt.pcolormesh(xx, yy, predicted, cmap=plt.cm.coolwarm, alpha=0.1)

plt.scatter(X[:, 0], X[:, 1], c=y, s=points\_size, cmap=plt.cm.coolwarm, alpha=0.90)

plt.ylim([yy.min(),yy.max()])

plt.xlim([xx.min(),xx.max()])

plt.xlabel(xlabel)

plt.ylabel(ylabel)

return clf

**ПРАКТИКА**

1Загрузите датасет **digits** с помощью функции *load\_digits* из **sklearn.datasets** и подготовьте матрицу признаков X  и ответы на обучающей выборке Y  (вам потребуются поля *data* и *target* в объекте, который возвращает *load\_digits*).

2Информацию о датасете вы можете получить, обратившись к полю *DESCR* у возвращаемого объекта *load\_digits*. Нам предстоит решать задачу классификации изображений с цифрами по численным признакам.

3Для оценки качества мы будем использовать **cross\_val\_score** из **sklearn.model\_selection** с параметром cv=10. Эта функция реализует *k-fold cross validation* c  k равным значению параметра . Предлагается использовать k=10, чтобы полученные оценки качества имели небольшой разброс, и было проще проверить полученные ответы. На практике же часто хватает и k=5. Функция *cross\_val\_score* будет возвращать *numpy.ndarray*, в котором будет k чисел — качество в каждом из k экспериментов *k-fold cross validation*. Для получения среднего значения (которое и будет оценкой качества работы) вызовите метод **.mean()** у массива, который возвращает *cross\_val\_score*.

С небольшой вероятностью вы можете натолкнуться на случай, когда полученное вами качество в каком-то из пунктов не попадёт в диапазон, заданный для правильных ответов — в этом случае попробуйте перезапустить ячейку с *cross\_val\_score* несколько раз и выбрать наиболее «типичное» значение. Если это не помогает, то где-то была допущена ошибка.

Чтобы ускорить вычисление *cross\_val\_score,* следует попробовать использовать параметр **n\_jobs**. Число, которое вы подаёте в качестве этого параметра, соответствует количеству потоков вашего процессора, которое будет задействовано в вычислении. Если указать *n\_jobs = -1*, тогда будет задействовано максимальное число потоков.

Задания 5.7.1

1. Создайте DecisionTreeClassifier с настройками по умолчанию и измерьте качество его работы с помощью cross\_val\_score.

Эту величину введите в поле для ответа (ваше значение должно попасть в заданный интервал).

Ответ:

2. Теперь давайте обучим BaggingClassifier на основе DecisionTreeClassifier. Из sklearn.ensemble импортируйте BaggingClassifier, все параметры задайте по умолчанию. Нужно изменить только количество базовых моделей, задав его равным 100.

Подумайте, какие выводы можно сделать из соотношения качества одиночного дерева и бэггинга деревьев?

Ответ:

3. Теперь изучите параметры BaggingClassifier и выберите их такими, чтобы каждый базовый алгоритм обучался не на всех d  признаках, а на sqrt(d) случайных признаках.

Корень из числа признаков — часто используемая эвристика в задачах классификации, в задачах регрессии же часто берут число признаков, деленное на три,  log(d) тоже имеет место быть. Но в общем случае ничто не мешает вам выбирать любое другое число случайных признаков, добиваясь лучшего качества на кросс-валидации.

Ответ:

4. В предыдущем пункте мы выбирали подмножество один раз для каждого очередного дерева. Следующим нашим шагом будет построение бэггинга на основе деревьев, которые выбирают случайное подмножество признаков для каждой вершины дерева.

Для этого нам потребуется перенести отвечающий за это параметр из BaggingClassifier в DecisionTreeClassifier. Для этого вам из документации нужно выяснить, какой параметр DecisionTreeClassifier за это отвечает.

По-прежнему сэмплируем sqrt(d) признаков.

Ответ:

Задание 5.7.2

Полученный в задании 4 классификатор — **бэггинг на рандомизированных деревьях** (в которых при построении каждой вершины выбирается случайное подмножество признаков и разбиение ищется только по ним). Это в точности соответствует алгоритму **Random Forest**, поэтому почему бы не сравнить качество работы классификатора с RandomForestClassifier из sklearn.ensemble?

Сделайте это, а затем изучите, как качество классификации на данном датасете зависит от количества деревьев, количества признаков, выбираемых при построении каждой вершины дерева, а также ограничений на глубину дерева.

Для наглядности лучше построить графики зависимости качества от значений параметров, но для сдачи задания это делать не обязательно.

**На основе наблюдений выберите правильные утверждения из приведенных ниже:**

* Случайный лес сильно переобучается с ростом количества деревьев.
* При очень маленьком числе деревьев (5, 10, 15) случайный лес работает хуже, чем при большем числе деревьев.
* С ростом количества деревьев в случайном лесе, в какой-то момент деревьев становится достаточно для высокого качества классификации, а затем качество существенно не меняется.
* При большом количестве признаков (для данного датасета - 40-50) качество классификации становится хуже, чем при малом количестве признаков (10-15). Это связано с тем, что чем меньше признаков выбирается в каждом узле, тем более различными получаются деревья (ведь деревья сильно неустойчивы к изменениям в обучающей выборке), и тем лучше работает их композиция.
* При большом количестве признаков (40, 50, 60) качество классификации лучше, чем при малом количестве признаков (5, 10). Это связано с тем, что чем больше признаков, тем больше информации об объектах, а значит алгоритм может делать прогнозы более точно.
* При небольшой максимальной глубине деревьев (5-6) качество работы случайного леса намного лучше, чем без ограничения глубины, т.к. деревья получаются не переобученными. С ростом глубины деревьев качество ухудшается.
* При небольшой максимальной глубине деревьев (5-6) качество работы случайного леса заметно хуже, чем без ограничений, т.к. деревья получаются недообученными. С ростом глубины качество сначала улучшается, а затем не меняется существенно, так как из-за усреднения прогнозов и различий деревьев их переобученность в бэггинге не сказывается на итоговом качестве (все деревья преобучены по-разному, и при усреднении они компенсируют переобученность друг друга).