Лекция 13. Линейные композиции, бустинг и случайные леса

Как правило, в качестве композиции над некоррелированными решающими деревьями чаще рассматривают случайные леса, которые являются чуть усложнённой версией бэггинга над деревьями, а потому для них верны все приведенные ранее рассуждения.

Задача 1. Приведите временную асимптотику обучения и построения прогнозов для композиции вида

$$a_N(x) = \sum_{n=0}^{N} b_n(x)$$

над решающими деревьями bn.

Решение. На стадии обучения при построении очередного решающего дерева глубины D (под глубиной решающего дерева понимается максимальное количество ребер на кратчайшем пути от корня дерева до листа) нам необходимо выбрать предикаты в не более чем $2D-1$ внутренних вершинах этого дерева. Каждый порог выбирается путем перебора $\leq l-1$ значений для каждого из d признаков. После выбора предиктов необходимо вычислить прогноз дерева в каждом из листов за линейное время от количества объектов обучающей выборки, попавших в лист. Для вычисления прогнозов во всех листах одновременно нам достаточно один раз пройтись по всем объектам. Отсюда асимптотика построения одного решающего дерева — $O(2Dld+l)=O(2Dld)$.

На стадии построения прогноза для объекта x он "пропускается" через дерево от корня к листьям, тем самым проходя путь из не более чем D внутренних вершин, в каждой из которых происходит проверка предиката за константное время. Отсюда имеем асимптотику для построения прогноза композиции — $O(ND)$.

Таким образом, вычислительно менее сложным оказывается процесс обучения большого количества неглубоких деревьев, а потому градиентный бустинг является более выгодным с точки зрения времени обучения алгоритмом по сравнению со случайным лесом.

Заметим также, что, поскольку обучение градиентного бустинга является "направленным", то ему требуется меньшее по сравнению со случайным лесом количество базовых алгоритмов для достижения того же качества композиции (на примере задачи Kaggle: Predicting a Biological Response):

```
import pandas as pd
from sklearn.model_selection import KFold, cross_val_score, train_test_split
# from sklearn.cross_validation import KFold, cross_val_score, train_test_split
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier


data = pd.read_csv('train.csv')
X = data.ix[:, 1:].values
y = data.ix[:, 0].values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.8, random_state=241)
C:\Python\anaconda\lib\site-packages\ipykernel_launcher.py:8: DeprecationWarning:
.ix is deprecated. Please use
.loc for label based indexing or
.iloc for positional indexing
```

See the documentation here:
http://pandas.pydata.org/pandas-docs/stable/indexing.html#ix-indexer-is-deprecated

```
%%time
gbm = GradientBoostingClassifier(n_estimators=250, learning_rate=0.2, verbose=True).fit(X_train, y_train)
```

```
      Iter       Train Loss   Remaining Time
         1      13019.0677          14.42m
         2      10522.5398          14.58m
         3       8878.6186          15.05m
         4       7628.1041          15.13m
         5       6632.0479          14.97m
         6       5885.8641          14.88m
         7       5296.7696          14.75m
         8       4803.4788          14.66m
         9       4393.8196          14.58m
        10       4039.1706          14.52m
        20       2127.9423          13.80m
        30       1339.4550          13.10m
        40        927.9828          12.49m
        50        666.0596          11.92m
        60        487.1688          11.36m
        70        365.4408          10.76m
        80        271.9291          10.13m
        90        206.1799           9.51m
       100        158.3474           8.89m
       200         16.7478           2.92m
Wall time: 14min 10s
```

```
import numpy as np
from sklearn.metrics import log_loss

for learning_rate in [1, 0.5, 0.3, 0.2, 0.1]:

    gbm = GradientBoostingClassifier(n_estimators=250, learning_rate=learning_rate, random_state=241).fit(X_train, y_train)

    l = log_loss

    test_deviance = np.zeros((gbm.n_estimators,), dtype=np.float64)
    for i, y_pred in enumerate(gbm.staged_decision_function(X_test)):
        y_pred = 1.0 / (1.0 + np.exp(-y_pred))
        test_deviance[i] = l(y_test, y_pred)

    train_deviance = np.zeros((gbm.n_estimators,), dtype=np.float64)
    for i, y_pred in enumerate(gbm.staged_decision_function(X_train)):
        y_pred = 1.0 / (1.0 + np.exp(-y_pred))
        train_deviance[i] = l(y_train, y_pred)

    plt.figure()
```

```
plt.plot(test_deviance, 'r', linewidth=2)
plt.plot(train_deviance, 'g', linewidth=2)
plt.legend(['test', 'train'])
plt.plot([0, train_deviance.shape[0]], [test_deviance.min(), test_deviance.min()], 'g--')
plt.plot([test_deviance.argmin()], [test_deviance.min()], 'v')
plt.title('GBM eta=%.1f, test logloss=%.3f, best_est=%d' % (learning_rate, test_deviance.min(), test_deviance.argmin()+1))
plt.xlabel('Number of trees')
plt.ylabel('Loss')
```

Итого, лучшая композиция построена при η=0.1, включает 52 базовых алгоритма и достигает значения 0.526 на контрольной выборке. При этом случайный лес с таким же количеством базовых алгоритмов уступает градиентному бустингу:

```
rf = RandomForestClassifier(n_estimators=52, random_state=241).fit(X_train, y_train)
print 'Train RF log-loss =', log_loss(y_train, rf.predict_proba(X_train))
print 'Test RF log-loss = ', log_loss(y_test, rf.predict_proba(X_test))
Train RF log-loss = 0.156601783371
Test RF log-loss =  0.538159117307
```

Заметим также, что при всём этом случайный лес, в отличие от градиентного бустинга, использует глубокие деревья, требующие вычислительных мощностей для их обучения.

Для достижения такого же качества случайному лесу требуется гораздо большее число базовых алгоритмов:

```
for n_estimators in xrange(10, 101, 10):
    rf = RandomForestClassifier(n_estimators=n_estimators, n_jobs=4).fit(X_train, y_train)
    print n_estimators, 'trees: train log-loss =', log_loss(y_train, rf.predict_proba(X_train)), 'test log-loss =', log_loss(y_test, rf.predict_proba(X_test))
10 trees: train log-loss = 0.158545638808 test log-loss = 0.834458904019
20 trees: train log-loss = 0.159034348348 test log-loss = 0.560379871727
30 trees: train log-loss = 0.164681822387 test log-loss = 0.541165207884
40 trees: train log-loss = 0.155531512135 test log-loss = 0.530862901947
50 trees: train log-loss = 0.157137200756 test log-loss = 0.531500493502
60 trees: train log-loss = 0.155803613942 test log-loss = 0.531160351563
70 trees: train log-loss = 0.15500782203 test log-loss = 0.528269840784
80 trees: train log-loss = 0.156055208478 test log-loss = 0.530507904098
90 trees: train log-loss = 0.156991521552 test log-loss = 0.52515584889
100 trees: train log-loss = 0.156396091372 test log-loss = 0.526445919347
```

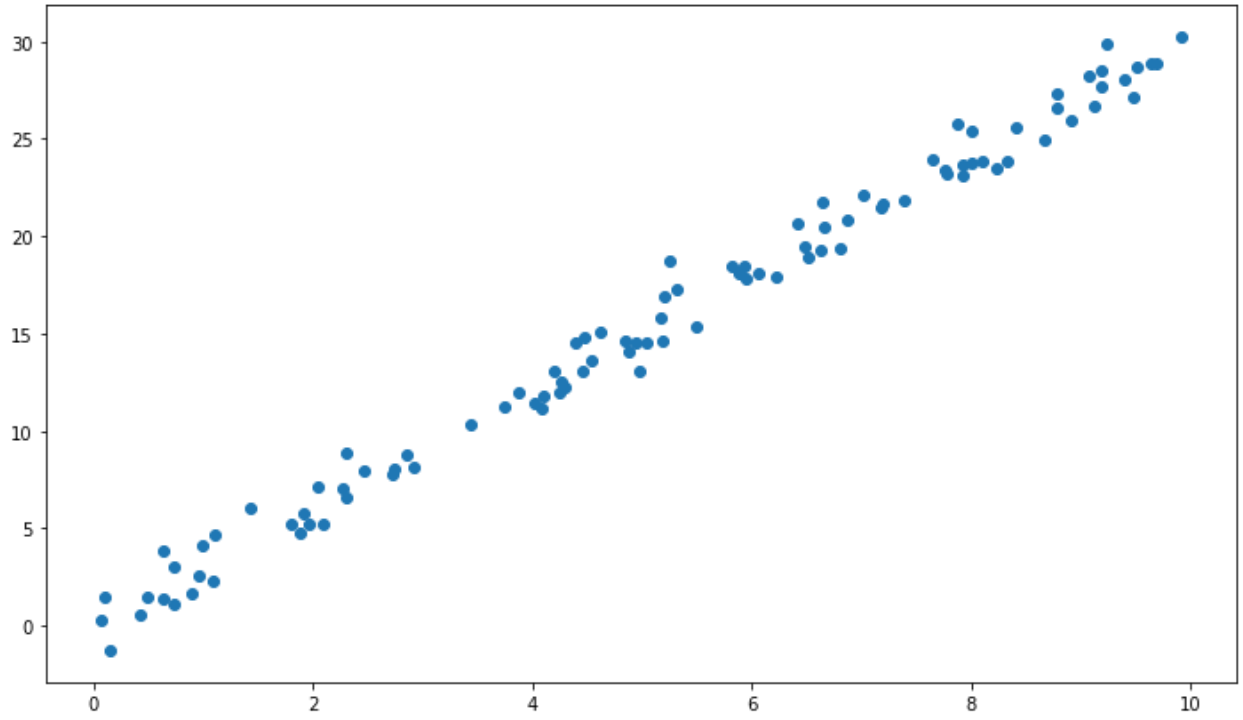
Свойства градиентного бустинга над решающими деревьями

Задача 2. Пусть решается задача регрессии на одномерной выборке X={(xi,yi)}li=1, при этом истинная зависимость целевой переменной является линейной: $y(x)$=ax+ε,ε~$p(\varepsilon)$=□(0,σ2). Допустим, не зная этого, вы обучили на выборке линейную регрессию и решающее дерево с функционалом MSE, и вам известно, что модели не переобучились. После этого вы получили новые данные и построили на них прогнозы обеих моделей, и оказалось, что для решающего дерева значение функционала ошибки на новых данных оказалось радикально выше, чем для линейной регрессии. Чем это может быть вызвано?

Решение. Поскольку истинная зависимость в данных является линейной, логично предположить, что линейная модель покажет лучшие результаты на подобной выборке. Опишем формально ситуацию, в которой у решающего дерева могут возникнуть серьезные проблемы с восстановлением истинной зависимости.

Допустим, обучающая выборка была получена из отрезка [0;10], обучим соответствующие модели и построим прогнозы для этого отрезка:

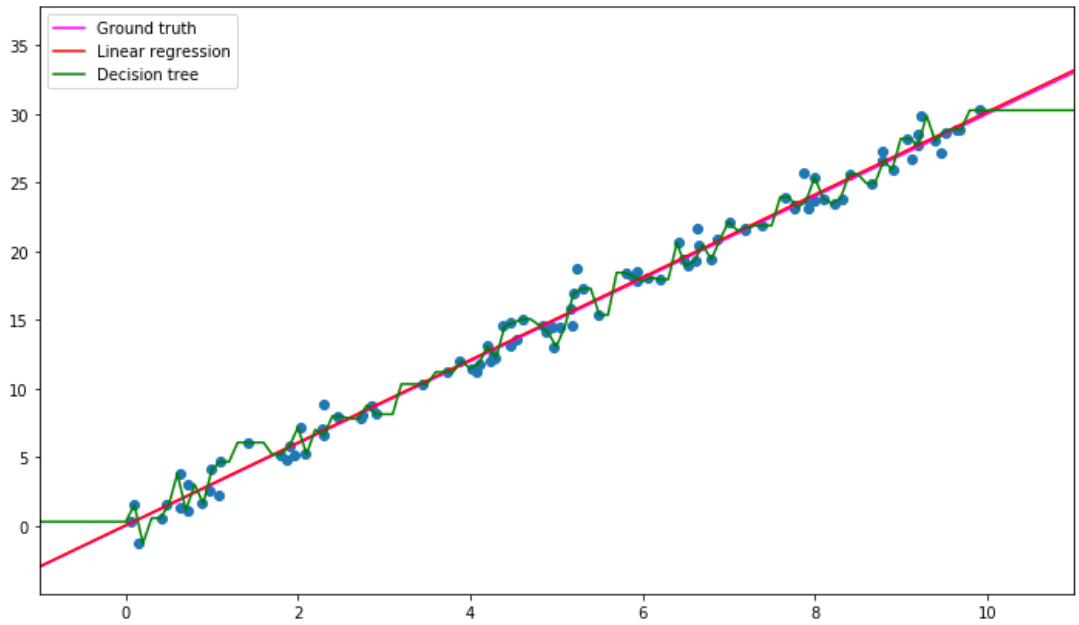
```
from numpy.random import rand, randn

set_size = 100
lin_coef = 3
sigma = 1

X_train = (rand(set_size) * 10).reshape(-1, 1)
Y_train = X_train * 3 + sigma * randn(set_size).reshape(-1, 1)
plt.figure(figsize=(12, 7))
plt.scatter(X_train, Y_train)
```




```
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error

lr = LinearRegression()
lr.fit(X_train, Y_train)
tree = DecisionTreeRegressor()
tree.fit(X_train, Y_train)
DecisionTreeRegressor(criterion='mse', max_depth=None, max_features=None,
                      max_leaf_nodes=None, min_impurity_decrease=0.0,
                      min_impurity_split=None, min_samples_leaf=1,
                      min_samples_split=2, min_weight_fraction_leaf=0.0,
                      presort=False, random_state=None, splitter='best')
```

```
from numpy.random import rand, randn

grid = np.arange(-1, 12, 0.1).reshape(-1, 1)

plt.figure(figsize=(12, 7))
plt.scatter(X_train, Y_train)
plt.plot(grid, lin_coef * grid, 'magenta')
plt.plot(grid, lr.predict(grid), 'red',)
plt.plot(grid, tree.predict(grid), 'green')
plt.xlim([-1, 11])
plt.legend(['Ground truth', 'Linear regression', 'Decision tree'], loc=0)
print ('LR train MSE = ', mean_squared_error(Y_train, lr.predict(X_train)))
print('DT train MSE = ', mean_squared_error(Y_train, tree.predict(X_train)))
```

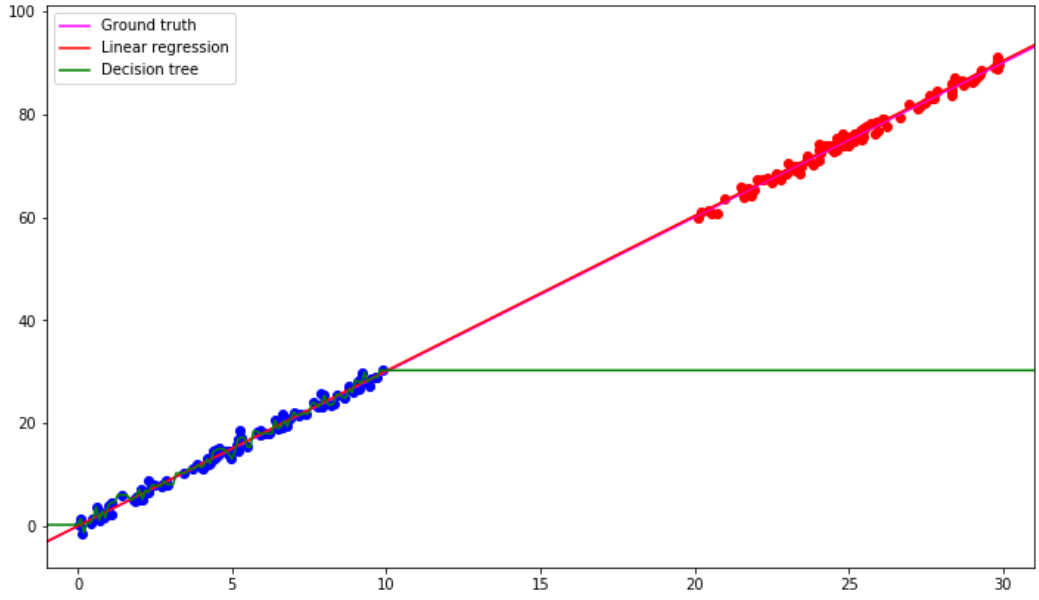
LR train MSE = 0.8771850536999435
DT train MSE = 0.0



Предположим, что новые данные были получены из другой области пространства ответов, например, из отрезка [20;30]. В этом случае предсказания линейной регрессии окажутся гораздо ближе к правде, что отразится и на значении функционала ошибки:

```
from numpy.random import rand, randn

grid = np.arange(-1, 32, 0.1).reshape(-1, 1)

X_test = (20 + rand(set_size) * 10).reshape(-1, 1)
Y_test = X_test * 3 + sigma * randn(set_size).reshape(-1, 1)
plt.figure(figsize=(12, 7))
plt.scatter(X_train, Y_train, c='blue')
plt.scatter(X_test, Y_test, c='red')

plt.plot(grid, lin_coef * grid, 'magenta')
plt.plot(grid, lr.predict(grid), 'red',)
plt.plot(grid, tree.predict(grid), 'green')
plt.xlim([-1, 31])
plt.legend(['Ground truth', 'Linear regression', 'Decision tree'], loc=0)
```

```
print('LR test MSE = ', mean_squared_error(Y_test, lr.predict(X_test)))
print('DT test MSE = ', mean_squared_error(Y_test, tree.predict(X_test)))
```
LR test MSE = 0.7752275299619684
DT test MSE = 2086.7412922635685

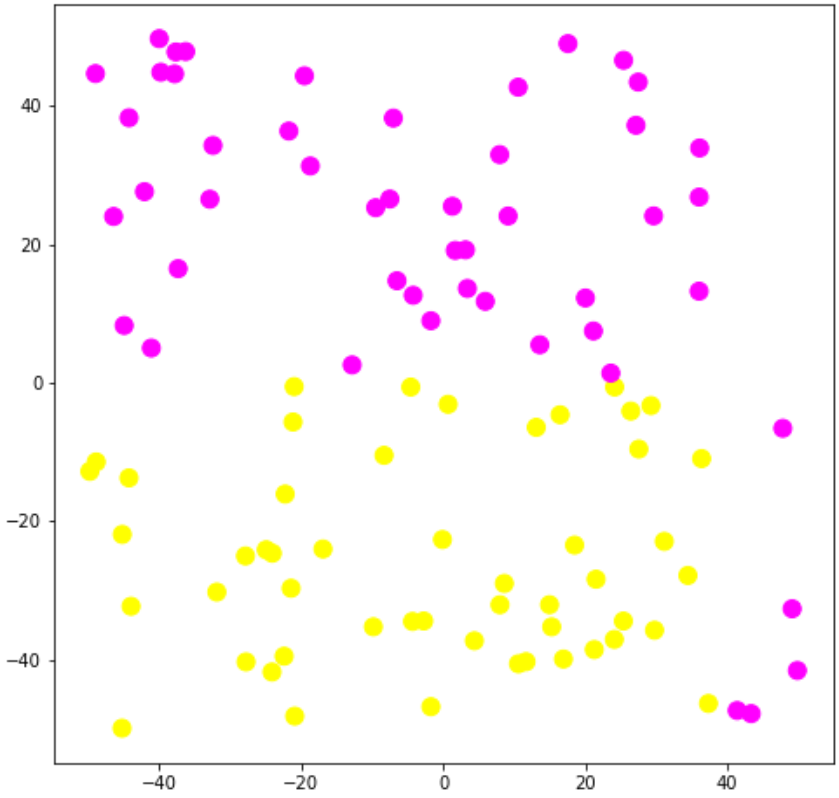


Вывод: решающие деревья (а потому и композиции над ними, в т.ч. градиентный бустинг) непригодны для экстраполяции функций.

В качестве решения этой проблемы в некоторых случаях можно использовать нормализацию.

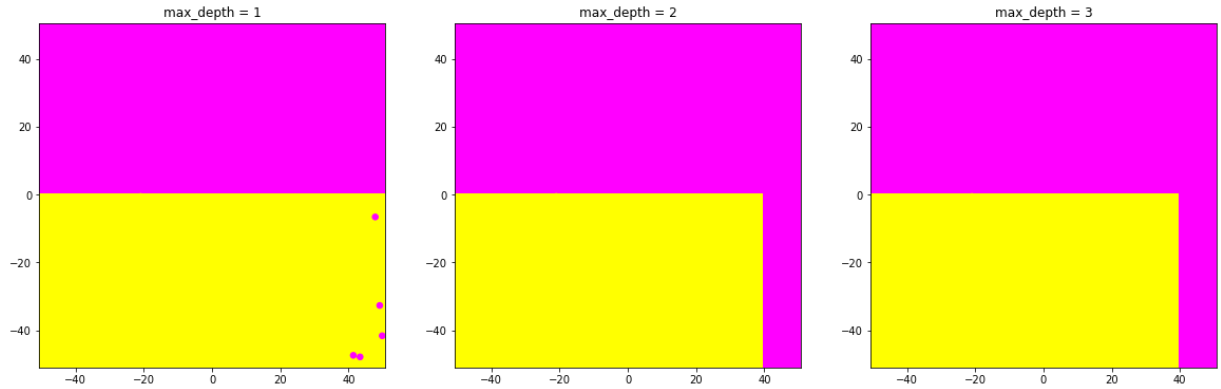
Задача 3. Пусть решается задача регрессии для двумерной выборки $X=\{(xi,yi)\}li=1$, при этом истинная зависимость целевой переменной $y(x)=y((x1,x2))=sgn((x1-40)x2)$. Предположим, вы обучили на выборке X решающие деревья глубины 1, 2 и 3. Как соотносятся значения функционала MSE для каждой из этих моделей? Почему? Какие зависимости позволяют улавливать глубокие деревья по сравнению с неглубокими?

```
X = -50 + 100 * np.random.rand(100, 2)
Y = np.sign((X[:, 0] - 40) * X[:, 1])
plt.figure(figsize=(8, 8))
plt.scatter(X[:, 0], X[:, 1], c=Y, s=100, cmap='spring')
```




```
def get_grid(data):
    x_min, x_max = data[:, 0].min() - 1, data[:, 0].max() + 1
    y_min, y_max = data[:, 1].min() - 1, data[:, 1].max() + 1
    return np.meshgrid(np.arange(x_min, x_max, 0.5),
                       np.arange(y_min, y_max, 0.5))
plt.figure(figsize=(20, 6))
for i in range(3):
    clf = DecisionTreeRegressor(random_state=42, max_depth = i + 1)

    clf.fit(X, Y)
    xx, yy = get_grid(X)
    predicted = clf.predict(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)

    plt.subplot2grid((1, 3), (0, i))
    plt.pcolormesh(xx, yy, predicted, cmap='spring')
    plt.scatter(X[:, 0], X[:, 1], c=Y, s=30, cmap='spring')
    plt.title('max_depth = ' + str(i + 1))
    print('DT MSE (max_depth = ' + str(i+1) + ') = ', mean_squared_error(Y.reshape(-1, 1),
clf.predict(X)))
```

```
DT MSE (max_depth = 1) =  0.1821428571428571
DT MSE (max_depth = 2) =  0.0
DT MSE (max_depth = 3) =  0.0
```



Можно видеть, что с незначительным ростом глубины дерева ошибка на данных стремительно падает. Это связано с тем, что истинная зависимость целевой переменной учитывает взаимодействия признаков, а потому дерево глубины 1 в любом случае не может восстановить подобную зависимость, поскольку учитывается лишь один из признаков; дерево глубины 2 позволяет учесть взаимодействие пар признаков и т.д.

Вывод: если вам известно, что в задаче необходимо учитывать взаимодействие N признаков, корректным решением будет ограничивать глубину деревьев в градиентном бустинге числом, большим N. Кроме того, если вам из некоторых экспертных знаний известно, что на целевую переменную оказывает влияние конкретное взаимодействие различных признаков, следует добавить его явно в качестве признака, поскольку это может позволить повысить качество.

Визуализация градиентного бустинга для решающих деревьев различной глубины для функций различного вида.

Задача 4. Пусть дана некоторая выборка $X=\{(xi,yi)\}li=1$. Как изменится решающее дерево, обученное на этой выборке, при масштабировании признаков?

Решение. Рассмотрим некоторую вершину m и множество Rm объектов, попавших в неё. Заметим, что значение целевой переменной при масштабировании не изменится, а потом значение критерия информативности для каждого из возможных разбиений останется тем же самым (изменятся лишь значения порогов для каждого из признаков). В связи с этим в вершине m в качестве оптимального будет выбрано то же разбиение, что и раньше. Поскольку это верно для любой вершины, построенное решающее дерево не изменится.

Данные рассуждения также верны для любого монотонного преобразования признаков.

Размер шага в градиентном бустинге

Напомним, что в качестве меры для борьбы с переобучением в градиентном бустинге используют размер шага η:

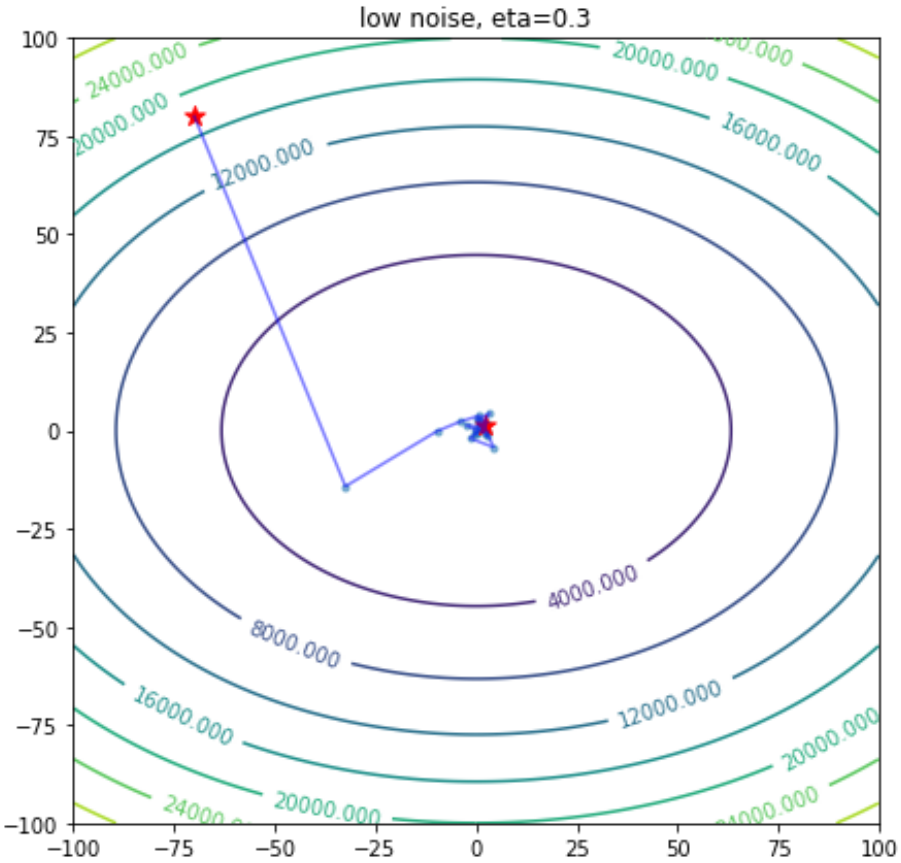
$$a_N(x) = \sum_{n=0}^{N} \eta \gamma_n b_n(x).$$

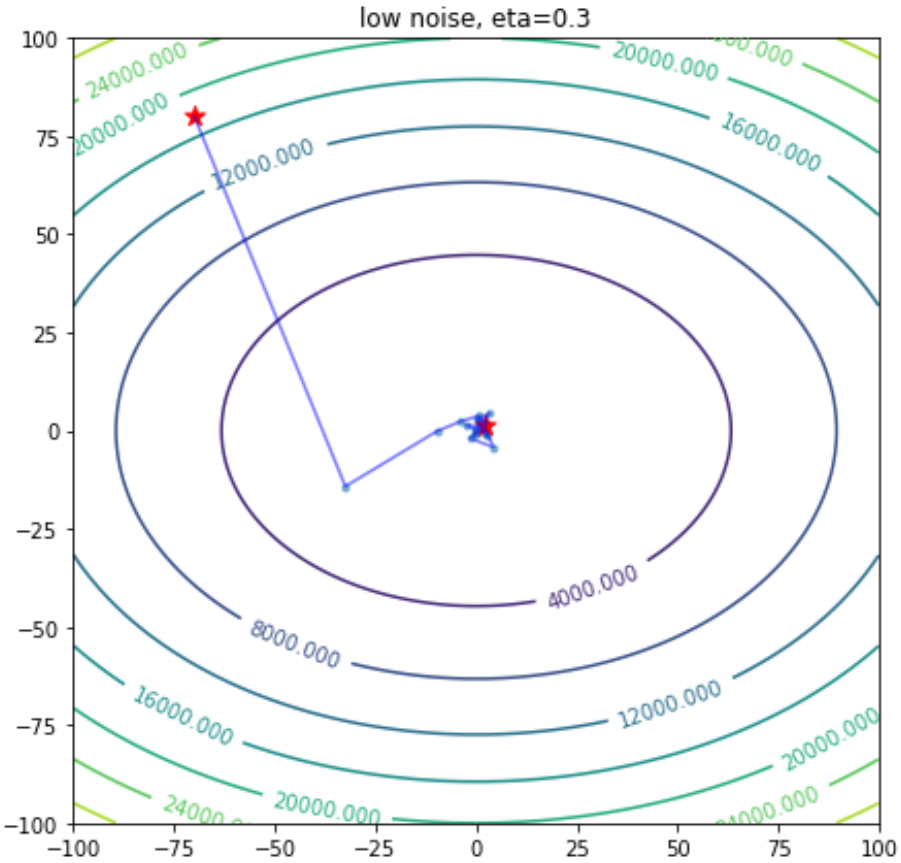
Исследуем зависимость скорости сходимости градиентного бустинга в зависимости от размера шага и богатства семейства базовых алгоритмов □. На каждом шаге мы выбираем новый базовый алгоритм из семейства, обучая его на векторе сдвигов. Тем не

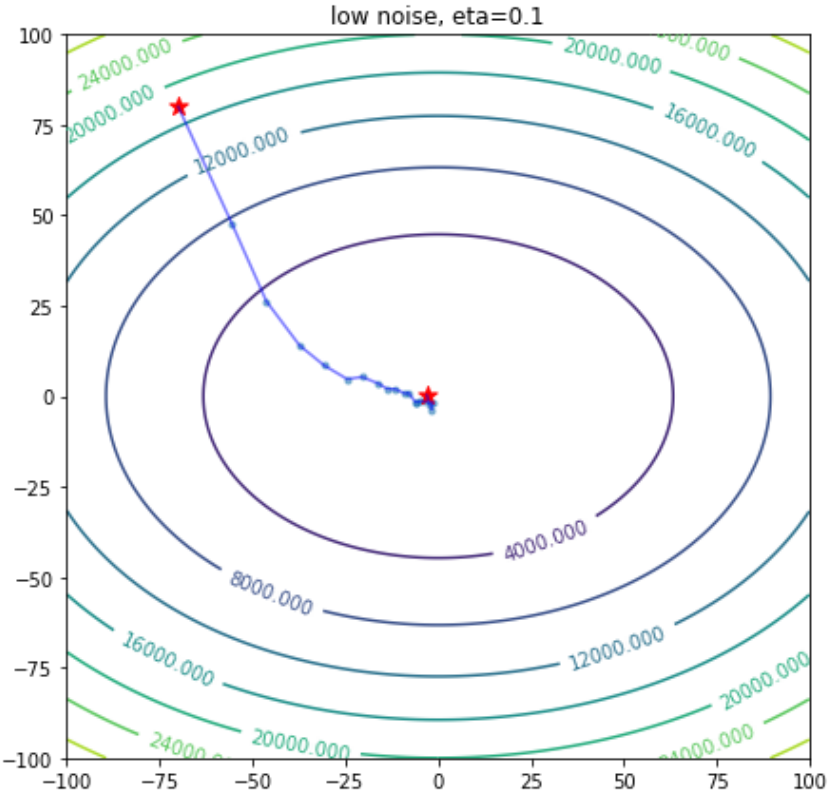
менее, не всегда получается найти алгоритм, идеально приближающие посчитанный вектор сдвигов, поэтому иногда добавление нового базового алгоритма может "увести" нас в сторону.

Рассмотрим обычный градиентный спуск и смоделируем описанную ситуацию, используя не честно посчитанный антиградиент, а его зашумленный вариант, и исследуем, за какое количество шагов (сколько базовых алгоритмов потребуется бустингу) мы сможем оказаться в окрестности оптимума в зависимости от длины шага:

```
def fun(x, y):
    return x**2 + 2 * y ** 2

def grad(x, y):
    return np.array([2 * x, 4 * y])

def make_data():
    x = np.arange (-100, 100, 0.1)
    y = np.arange (-100, 100, 0.1)
    xgrid, ygrid = np.meshgrid(x, y)
    zgrid = fun(xgrid, ygrid)
#    zgrid = np.sin (xgrid) * np.sin (ygrid) / (xgrid * ygrid)
    return xgrid, ygrid, zgrid


def gradient_descent(numIterations, eta, noise):
    ans = np.array([-70, 80])
    x_points = [ans[0]]
    y_points = [ans[1]]
    for iter in range(0, numIterations):
        ans = ans - eta * (grad(ans[0], ans[1]) + noise * np.random.randn(2))
        x_points.append(ans[0])
        y_points.append(ans[1])
    return (x_points, y_points)

import numpy as np
from sklearn.metrics import log_loss
X, Y, Z = make_data()

for learning_rate in [0.5, 0.3, 0.1]:
    x_points, y_points = gradient_descent(20, learning_rate, 10)
    plt.figure(figsize = (7, 7))
    CS = plt.contour(X, Y, Z)
    plt.clabel(CS, inline=1, fontsize=10)
    plt.plot(x_points, y_points, linestyle = '-', color = 'blue', alpha = 0.5)
    plt.scatter(x_points[1:-1], y_points[1:-1], marker = '.', s=40, alpha = 0.5)
    plt.scatter([x_points[0], x_points[-1]], [y_points[0], y_points[-1]],
                marker = '*', s=100, color = 'red')

    plt.xlim([-100, 100])
    plt.ylim([-100, 100])
    plt.title('low noise, eta=%.1f' % learning_rate)
```

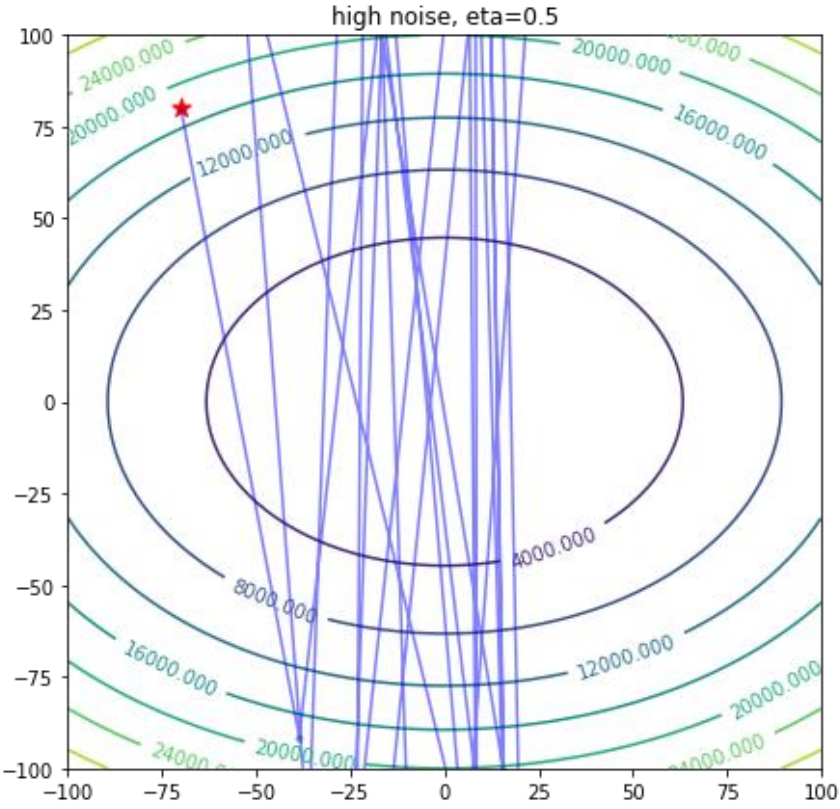

low noise, eta=0.3
24000.000
20000.000
16000.000
12000.000
8000.000
4000.000
100
75
50
25
0
−25
−50
−75
−100
−100
−75
−50
−25
0
25
50
75
100

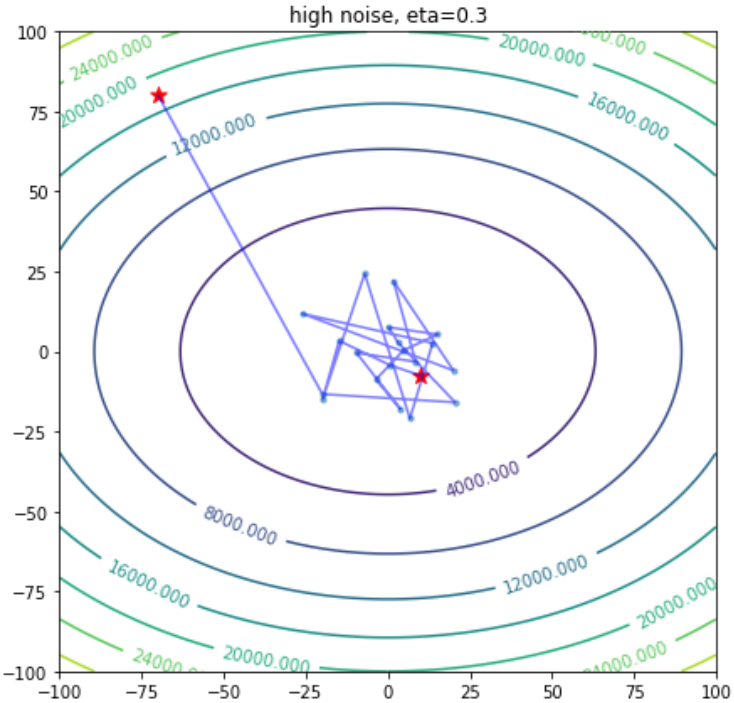

low noise, eta=0.3
24000.000
20000.000
16000.000
12000.000
8000.000
4000.000
100
75
50
25
0
−25
−50
−75
−100
−100
−75
−50
−25
0
25
50
75
100

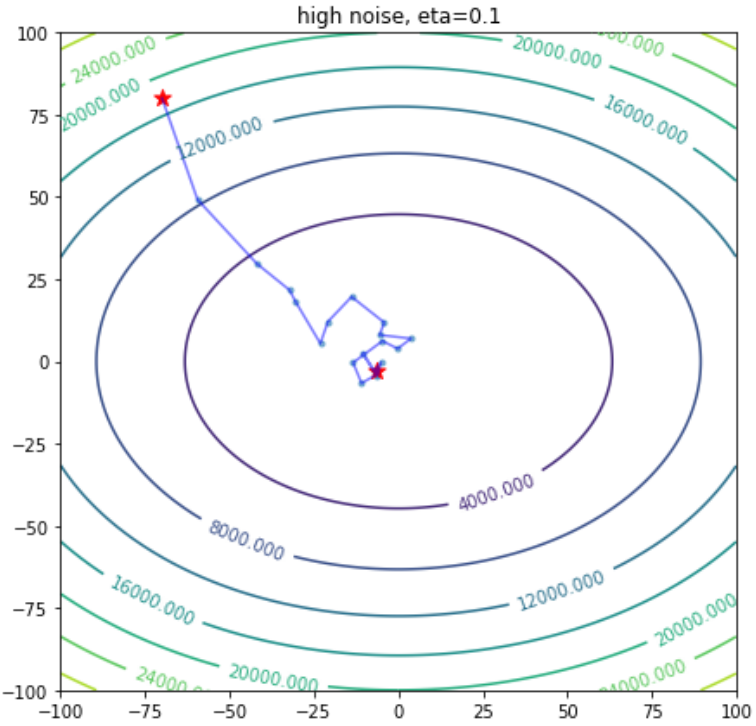


Если семейство алгоритмов более бедное, то направление движение на каждом шаге будет сильнее отличаться от антиградиента:

```
import numpy as np
from sklearn.metrics import log_loss
X, Y, Z = make_data()

for learning_rate in [0.5, 0.3, 0.1]:
    x_points, y_points = gradient_descent(20, learning_rate, 50)
    plt.figure(figsize = (7, 7))
    CS = plt.contour(X, Y, Z)
    plt.clabel(CS, inline=1, fontsize=10)
    plt.plot(x_points, y_points, linestyle = '-', color = 'blue', alpha = 0.5)
    plt.scatter(x_points[1:-1], y_points[1:-1], marker = '.', s=40, alpha = 0.5)
    plt.scatter([x_points[0], x_points[-1]], [y_points[0], y_points[-1]],
                marker = '*', s=100, color = 'red')

    plt.xlim([-100, 100])
    plt.ylim([-100, 100])
    plt.title('high noise, eta=%.1f' % learning_rate)
```


high noise, eta=0.5
100
75
50
25
0
−25
−50
−75
−100
−100
−75
−50
−25
0
25
50
75
100
24000.000
20000.000
16000.000
12000.000
8000.000
4000.000


high noise, eta=0.3
100
75
50
25
0
−25
−50
−75
−100
−100
−75
−50
−25
0
25
50
75
100
24000.000
20000.000
16000.000
12000.000
8000.000
4000.000

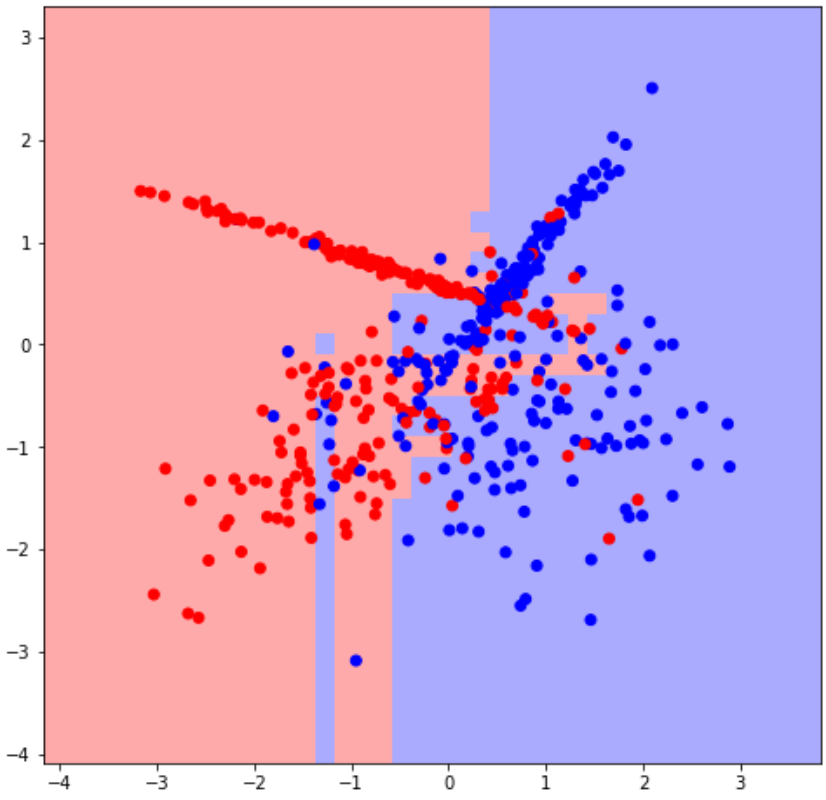


Можно видеть, что с уменьшением размера шага градиентному бустингу требуется больше базовых алгоритмов для достижения приемлемого качества композиции, однако при этом сходимость такого процесса надежнее.

XGBoost

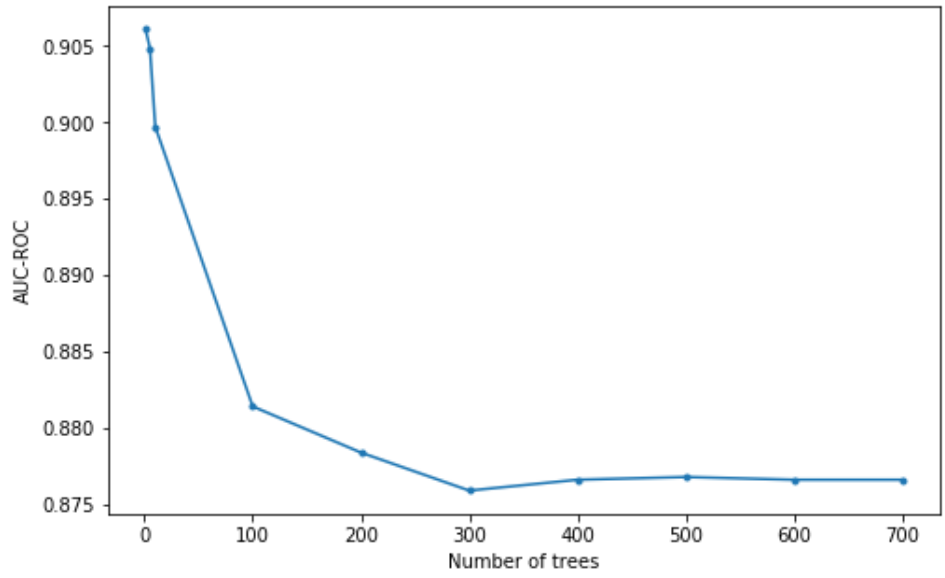
```
from sklearn.datasets import make_classification
from matplotlib.colors import ListedColormap
import xgboost as xgb

cmap_light = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF'])
cmap_bold = ListedColormap(['#FF0000', '#00FF00', '#0000FF'])

def plot_surface(X, y, clf):
    h = 0.2
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

    Z = Z.reshape(xx.shape)
    plt.figure(figsize=(8, 8))
    plt.pcolormesh(xx, yy, Z, cmap=cmap_light)

    # Plot also the training points
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap_bold)
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())

X, y = make_classification(n_samples=500, n_features=2, n_informative=2,
                           n_redundant=0, n_repeated=0,
```

```
                           n_classes=2, n_clusters_per_class=2,
                           flip_y=0.05, class_sep=0.8, random_state=241)
clf = xgb.XGBClassifier()
clf.fit(X, y)
plot_surface(X, y, clf)
```

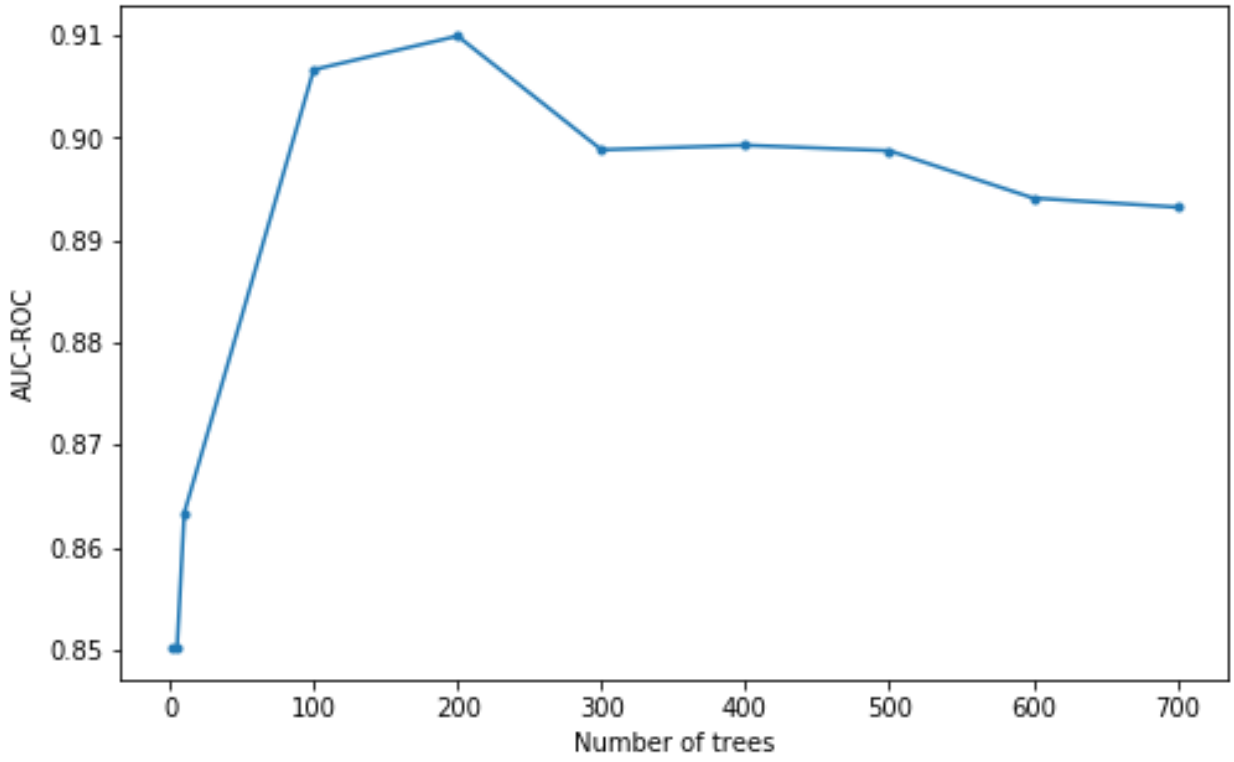



```
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_auc_score

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=241)

clf = xgb.XGBClassifier()
clf.fit(X_train, y_train)

print(roc_auc_score(y_test, clf.predict_proba(X_test)[:, 1]))
0.9007823613086771

?xgb.XGBClassifier

xgb.XGBClassifier
xgb.XGBClassifier
xgboost.sklearn.XGBClassifier


n_trees = [1, 5, 10, 100, 200, 300, 400, 500, 600, 700]
quals = []
for n in n_trees:
    clf = xgb.XGBClassifier(n_estimators=n, max_depth=6, learning_rate=0.5)
    clf.fit(X_train, y_train)
    q = roc_auc_score(y_test, clf.predict_proba(X_test)[:, 1])
```

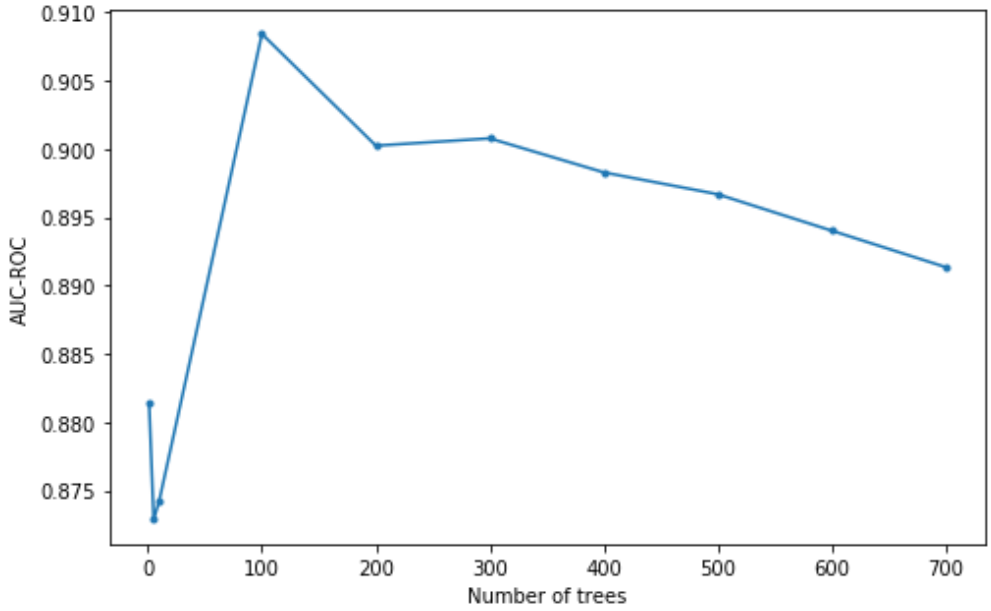
```
    quals.append(q)

plt.figure(figsize=(8, 5))
plt.plot(n_trees, quals, marker='.')
plt.xlabel('Number of trees')
plt.ylabel('AUC-ROC')
plt.show()
```




```
n_trees = [1, 5, 10, 100, 200, 300, 400, 500, 600, 700]
quals = []
for n in n_trees:
    clf = xgb.XGBClassifier(n_estimators=n, max_depth=2, learning_rate=0.03)
    clf.fit(X_train, y_train)
    q = roc_auc_score(y_test, clf.predict_proba(X_test)[:, 1])
    quals.append(q)

plt.figure(figsize=(8, 5))
plt.plot(n_trees, quals, marker='.')
plt.xlabel('Number of trees')
plt.ylabel('AUC-ROC')
plt.show()
```




```
n_trees = [1, 5, 10, 100, 200, 300, 400, 500, 600, 700]
quals = []
for n in n_trees:
    clf = xgb.XGBClassifier(n_estimators=n, max_depth=6,
                            learning_rate=0.03, reg_lambda=0)
    clf.fit(X_train, y_train)
    q = roc_auc_score(y_test, clf.predict_proba(X_test)[:, 1])
    quals.append(q)

plt.figure(figsize=(8, 5))
plt.plot(n_trees, quals, marker='.')
plt.xlabel('Number of trees')
plt.ylabel('AUC-ROC')
plt.show()
```


0.910
0.905
0.900
0.895
0.890
0.885
0.880
0.875
AUC-ROC
0
100
200
300
400
500
600
700
Number of trees