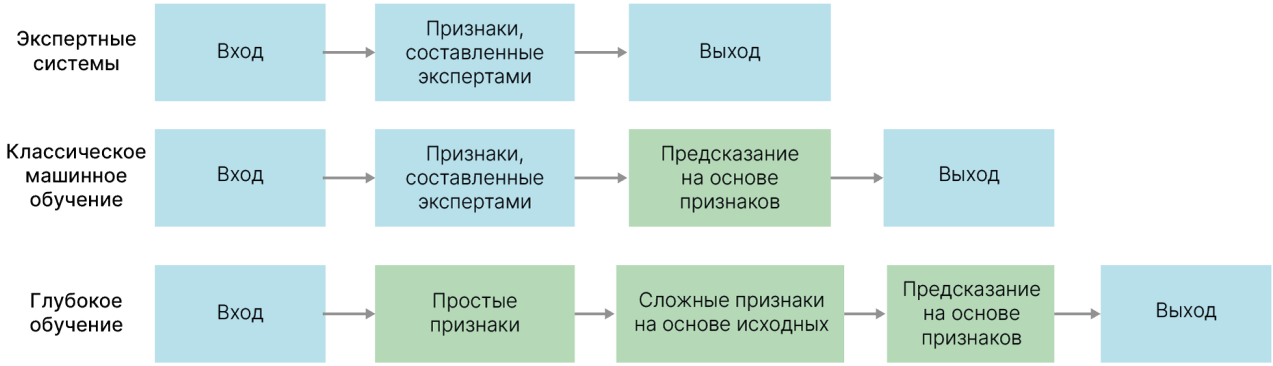
Классическое машинное обучение

Системы предсказаний

Помимо нейронных сетей, обучение работе с которыми являются основной задачей курса, есть и другие подходы к проблеме создания систем, помогающих решать те или иные проблемы.



Экспертные системы (Rule-based systems)

Самым первым подходом к созданию системы, способной на основе входных данных делать какие-то выводы, были так называемые **Rule-based systems**. В этих системах и за описание объекта — выделении значимых признаков — и за выработку правил, по которым система должна принимать то или иное решение, отвечал человек.

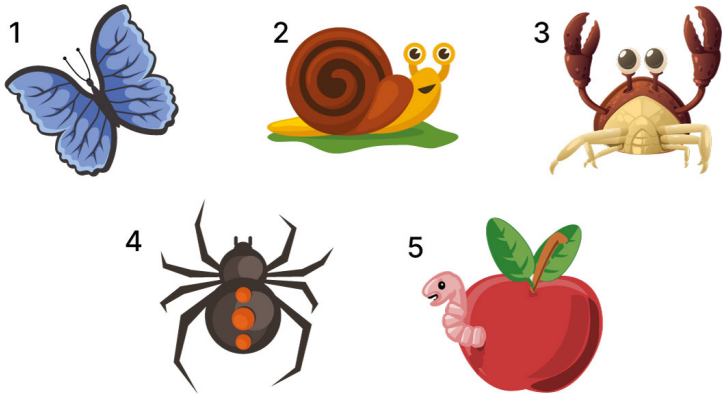
В современности такие системы до сих пор используются, например, в определителях растений. Такие определители представляют собой набор утверждений, например: "Растение имеет стержневую корневую систему", "Плод растения - костянка", на основе согласия/несогласия с которыми книга отсылает вас к другим утверждениям или, в конце концов, к названию растения.

Очевидно, что результативность такого подхода зависит:

1. от наличия базовых знаний у того, кто определителем пользуется;
2. от качества и охвата самого набора правил — к примеру, может ли этот набор правил справиться с ситуацией, когда какой-то признак отсутствует. В приведенном выше примере — растение плодоносит не весь год, потому не всегда можно четко ответить, какой у него плод.

Единственная выгода такой системы по сравнению с подходом "позови нормального специалиста" является то, что в большинстве случаев, требования к умениям пользователя все же меньше.

Игрушечный пример такой схемы:



1. Имеется раковина	ДА — переходи к 3
	НЕТ — переходи к 2
2. Обитает в море	ДА — Краб
	НЕТ — переходи к 3
3. Есть мускульная нога	ДА — Улитка
	НЕТ — переходи к 4
4. Имеет крылья	ДА — Бабочка
	НЕТ — переходи к 5
5. Имеет 8 ног	ДА — Паук
	НЕТ — переходи к 6
6. Обитает под землей	ДА — Червь

Классическое машинное обучение

Классическое машинное обучение избавляет нас от необходимости вручную составлять какие-то правила.

Процесс классического машинного обучения

В этом подходе от нас все равно требуется описание нашего объекта определенными признаками — мы должны получать это описание либо вручную, либо обрабатывая объект какими-то программами. Чтобы отличать такие признаки от признаков, которые автоматически выделяют нейронные сети, их называют **hand-designed features**.

Но затем эти описанные объекты передаются алгоритму, который уже сам формирует набор правил, по которому он должен решать поставленную задачу. Уже разобранные вами линейная регрессия и SVM сами выбирают, какие признаки и с какими весами им учитывать при принятии финального решения. В некоторых случаях (например, SVM), признаки могут дополнительно (явно или неявно) преобразовываться внутри самой модели и на их основе могут формироваться новые признаки.

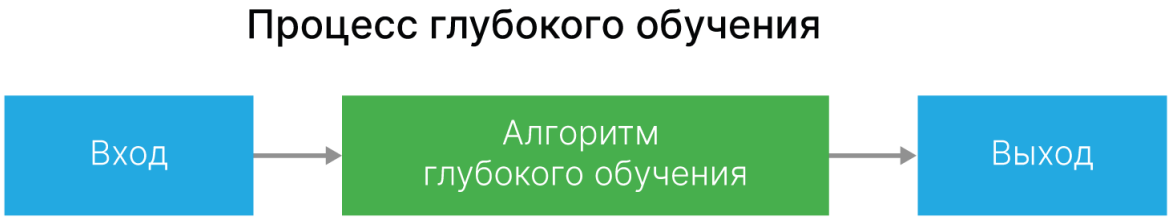
Но главное ограничение всегда остается - за описание объекта отвечает какой-то внешний источник и то, какие признаки в принципе подавать алгоритму на вход решает человек. А набор возможных преобразований признаков, как правило, строго фиксирован.

И составление таких признаков требует от исследователя очень хорошего знания изучаемой темы, знания специфики работы используемого алгоритма — к примеру, подавать алгоритму на вход попарные произведения всех признаков, или алгоритм сам в ходе работы их явно или неявно получит.

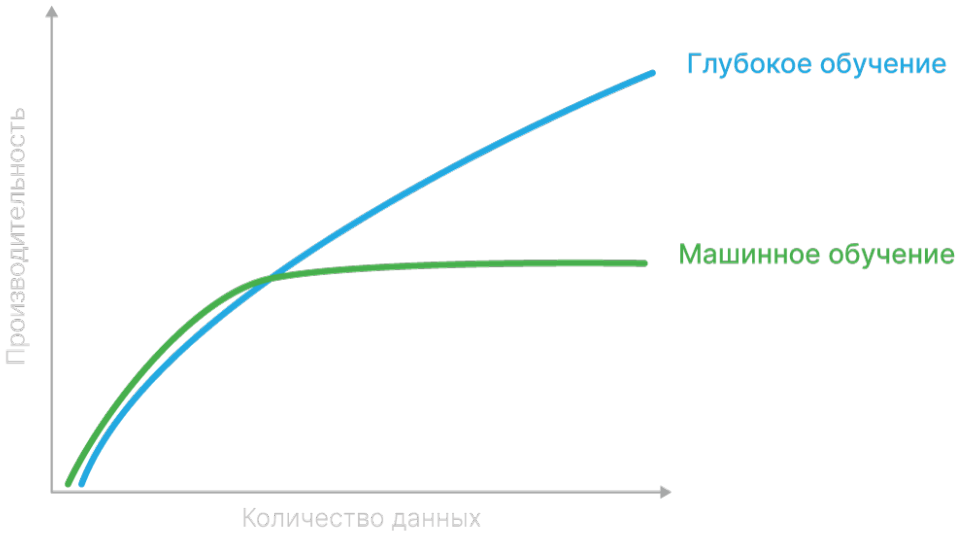
И очевидно, на больших объемах данных, когда в принципе можно использовать очень малоинформативные признаки (данных-то много, все коэффициенты при них оценим), которые и не придут исследователю в голову, классическое машинное обучение будет проигрывать.

Глубокое машинное обучение

В глубоком же машинном обучение признаки за нас выделяет нейросеть.



Это позволяет нейросети самой решать, какие признаки для данных объектов и для данной задачи важны и итеративно выделять сначала простые признаки, а затем комбинировать их в более сложные. За счет этого, в то время как качество подходов классического машинного обучения, по мере увеличения размера обучающей выборки со временем выходит на плато, для нейросетей это плато наступает сильно позже или вообще не наступает.



Потенциально такой подход позволяет почти не применять человеческую экспертизу при построении модели под задачу. Однако в реальности, человеческая экспертиза теперь переходит на уровень подбора архитектуры нейросети.

Необходимость методов классического машинного обучения

Казалось бы, если все так классно, то зачем нам вообще знать и использовать стандартные методы машинного обучения?

Оказывается, что существует целый спектр задач, в которых либо объем данных недостаточен для использования нейронных сетей, либо сама структура данных не предполагает каких-либо внутренних повторяющихся паттернов, на которые и заточены нейронные сети.

В ситуации, когда у вас есть просто таблица с признаками каждого объекта (**данные табулярного формата**), каждый признак не особо связан как-то с другими и взаиморасположение признаков в таблице не играет никакой роли:

1. По крайней мере в начале вам придется использовать полносвясзные слои, что очень сильно увеличивает число параметров нейросети
2. Полученные нейросети как правило не сильно превосходят методы классического обучения, а могут и вести себя хуже (из-за переобучения)

Кроме того, стоит учитывать, что часто для того, чтобы достигнуть качества top-of-the-art методов классического машинного обучения, необходимо долгое время подбирать архитектуры нейросети и настраивать ее параметры.

Время же получение решений при помощи классических методов часто исчисляется часами. Таким образом, даже в худшем случае, вы сможете оценить, насколько проблема хорошо решаема, и получить хороший бейзлайн для сравнения.

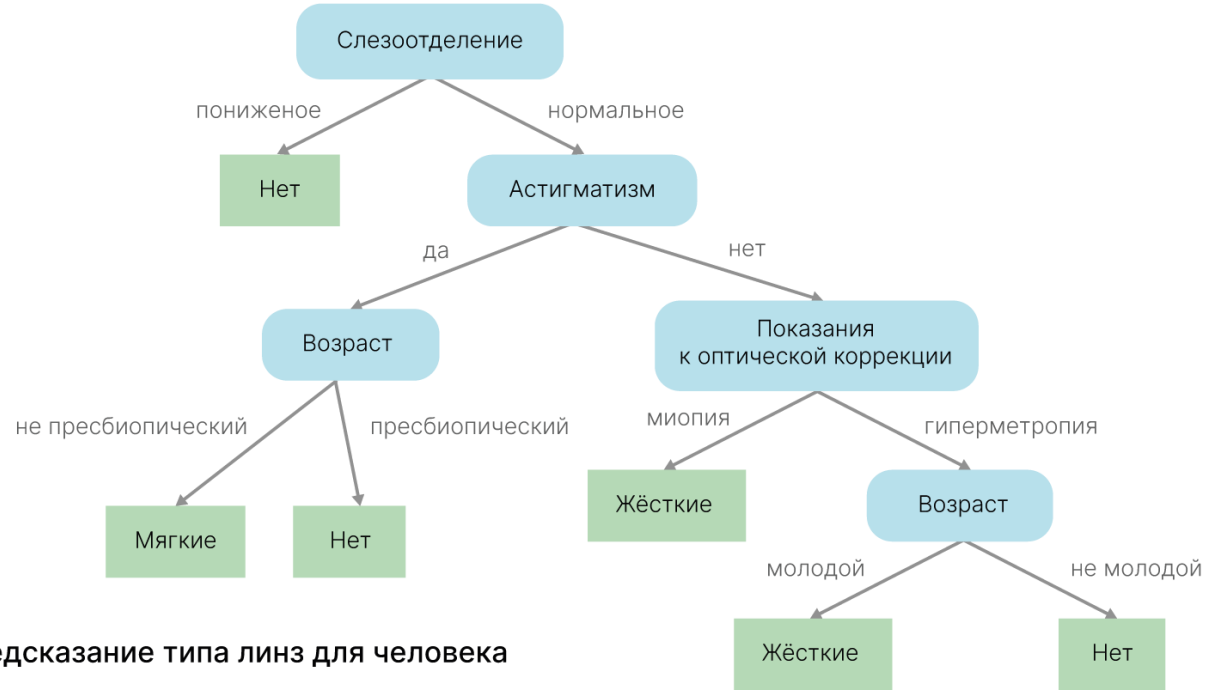
Помимо этого, методы машинного обучения дают более трактуемые предсказания - как правило, из них легче получить важность каждого отдельно взятого признака.

Кроме того, в современных задачах часто используются комбинации нейронных сетей и алгоритмов классического машинного обучения. Например, можно нейронные сети в качестве генераторов признаков для методов классического обучения.

Деревья решений

Деревья решений — это одна из первых моделей машинного обучения, которая была известна человеку. Изначально их строили без специальных алгоритмов, а просто вручную.

Когда требовалось принять решение по проблеме, для которой построено дерево, человек брал и проходился по этому дереву.



Предсказание типа линз для человека

Как оно устроено? В каждом узле есть какой-то вопрос. Например, нормальное ли у человека слезоотделение, есть ли него астигматизм и так далее. И, отвечая на каждый из этих вопросов, мы подбираем человеку линзы. Опять же, такое дерево решений можно построить без использования моделей машинного обучения, просто на основании опыта многих врачей. По сути это как раз те самые экспертные системы.

Понятно, что вручную такие деревья строить тяжело, для большого объема данных их руками и не построишь. Также возникает вопрос: зачем вообще нужна такая старая модель?

Оказывается, что эти модели могут быть неожиданно эффективны и их можно автоматически строить с помощью специально разработанных алгоритмов. Причем, это можно делать достаточно быстро даже на больших объемах данных.

How to win a Kaggle competition?

Anthony Goldbloom

"According to Anthony, in the history of Kaggle competitions, there are only two Machine Learning approaches that win competitions:
"Handcrafted & Neural Networks."

По словам Энтони Голдблум, одного из наиболее известных участников kaggle, на соревнованиях побеждают hand-crafted алгоритмы и нейронные сети. Под hand-crafted Энтони понимает алгоритмы, основанные на деревьях решений.

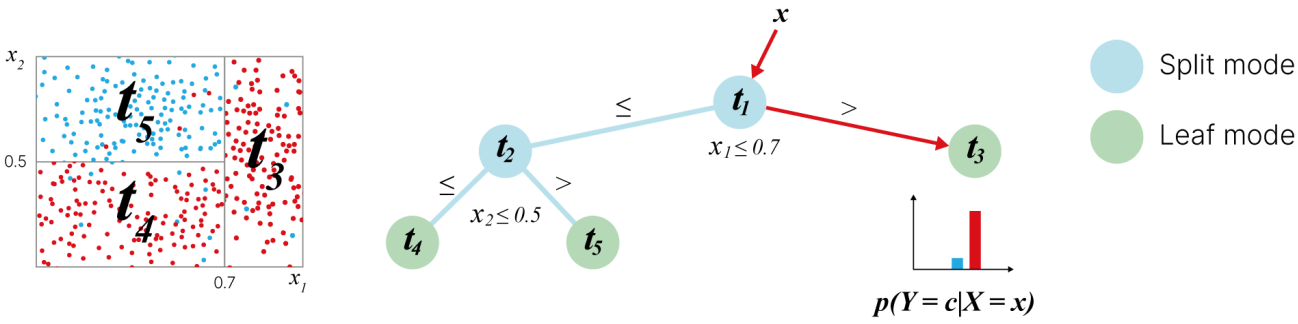
Где побеждают ансамбли деревьев решений?

- Recomendation systems (Netflix Prize 2009);
- Learning to rank (Yahoo Learning to rank challenge 2010);
- Crowdflower Search Results Relevance (2015);
- Avito Context Ad Clicks (2015);
- Везде :)

*"As long as Kaggle has been around, Anthony says, it has almost always been **ensembles of decision trees that have won competitions**."*

В принципе, до недавнего времени деревья решений побеждали практически везде — в рекомендательных системах, в задачах ранжирования, задачах релевантной выдачи для поиска, на предсказаниях, какую рекламу выберет пользователь и так далее. В любой задаче, где нет какой-то локальный связанности - которая есть в изображениях, текстах и других областях, захваченных нейронными сетями.

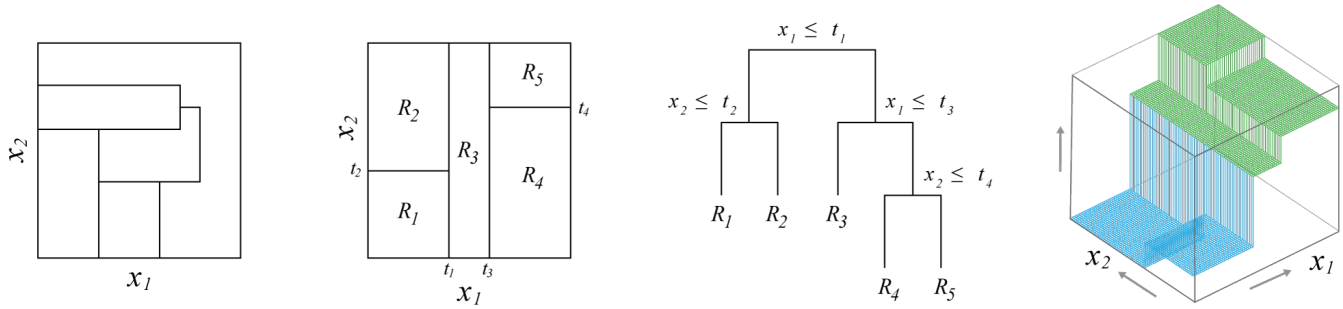
Принцип работы дерева решений



Принцип работы дерева решений можно проиллюстрировать на данном примере. Есть 2 признака, в данном случае вещественных. Для каждой точки мы создаем вопрос: 1

признак — больше 0.7 или меньше? Если больше 0.7, то это красная точка. Если меньше 0.7, то идем во второй внутренний узел T2 и спрашиваем: x2 меньше 0.5 или больше? Если меньше 0.5, то точка будет красная, в другом случае — синяя.

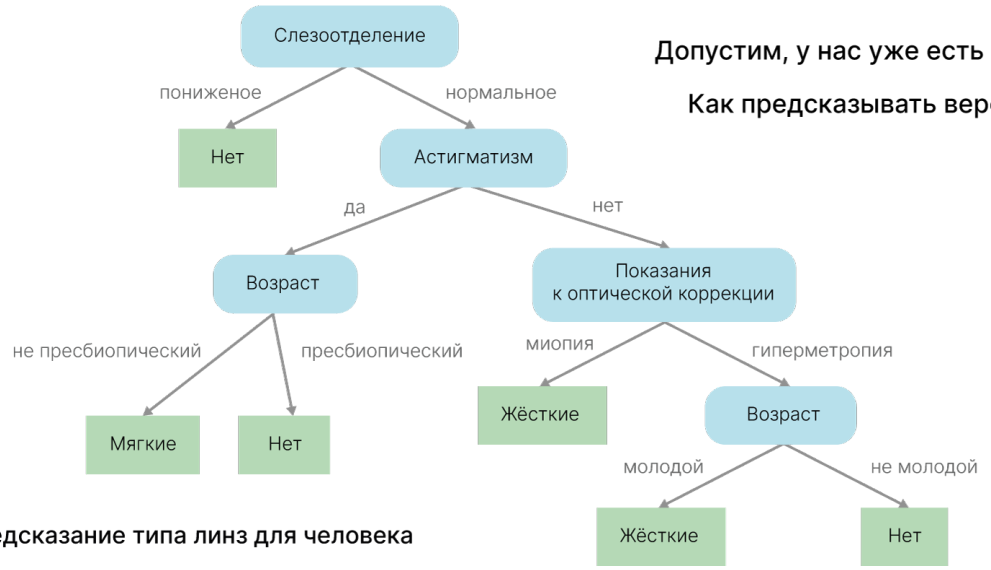
Разбиение пространства



Фактически, дерево решений бьет пространство признаков с помощью плоскостей на области, и в каждой из этих областей предсказывается какая-то константная величина.

Деревья решений (классификация)

Деревья решений, как правило, используются при решении 2-х типов задач — классификации и регрессии. Первая — классификация. Например, предсказание типа линз.

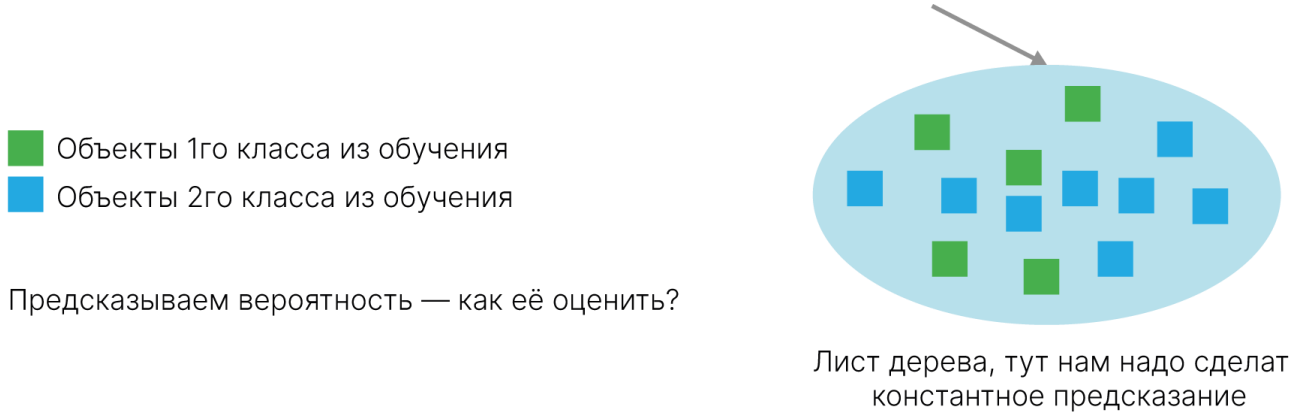


Допустим, у нас уже есть построенное дерево.

Как предсказывать вероятности классов?

Предсказание типа линз для человека

Особенно актуальны модели, которые могут предсказывать вероятности классов. Жесткие предсказания — не самый удачный вариант, лучше оставить человеку возможность выбирать. К примеру, лекарства всегда действует с какой-то вероятностью, потому что невозможно учесть все факторы.



На самом деле в деревьях решений в каждом листе находятся разные объекты, то есть каждому листу могут соответствовать объекты разных классов, а предсказать нужно один. Как предсказать вероятность этого класса? Нужно понять, что дерево решений выделило область пространства и пытается эту область описать эффективно (с точки зрения решения текущей задачи).

У нас есть небольшая выборка, и мы хотим оценить вероятность. Как это сделать: число представителей данного класса делим на общее число объектов в клетке.

Это согласуется с тем, как мы делаем в статистике. При построении дерева мы видим только обучающую выборку. А хотим делать выводы о генеральной совокупности. Конкретно здесь мы оцениваем, какая доля из объектов, попадающих в данный узел, принадлежит нужному классу.

Оценка доли объектов классов в популяции - здесь

$$\hat{p} = \frac{n}{N}$$

Эта оценка имеет дисперсию - величину, в некотором роде характеризующую то, насколько сильно мы можем ошибиться с оценкой доли.

$$D(\hat{p}) = \frac{p \cdot (1-p)}{N}$$

Это сразу дает нам одну важную интуицию - желательно, чтобы в листе дерева было не очень мало объектов. Чем объектов меньше - тем больше дисперсия и больше возможная ошибка.

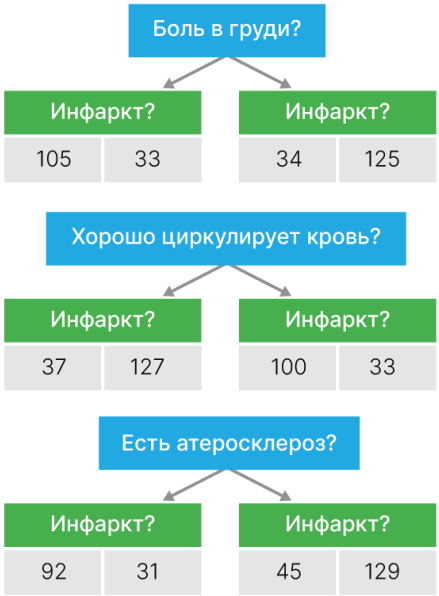
Как построить дерево решений?

Для бинарных признаков

Как строить дерево?

Перебираем таким образом все признаки

Боль в груди	Хорошо циркулирует кровь	Есть атеросклероз	Инфаркт
Нет	Нет	Нет	Нет
Да	Да	Да	Да
Да	Да	Нет	Нет
Да	Нет	Да	Да
...	...	...	...



Допустим, у нас есть табличка, в ней пока нет никаких вещественных признаков. Также есть много данных. Мы хотим построить дерево решений по этой табличке. Предсказывать будем, есть у человека инфаркт или нет. Возьмем в качестве признака боль в груди. В 1-ом и во 2-ом листе получается разное распределение людей. В левом листе инфаркт более вероятен, в правом — менее вероятен. Другим признаком может быть “как хорошо циркулирует кровь”, в таком случае тоже получится неплохое разделение. 3 признак — “есть ли атеросклероз”.

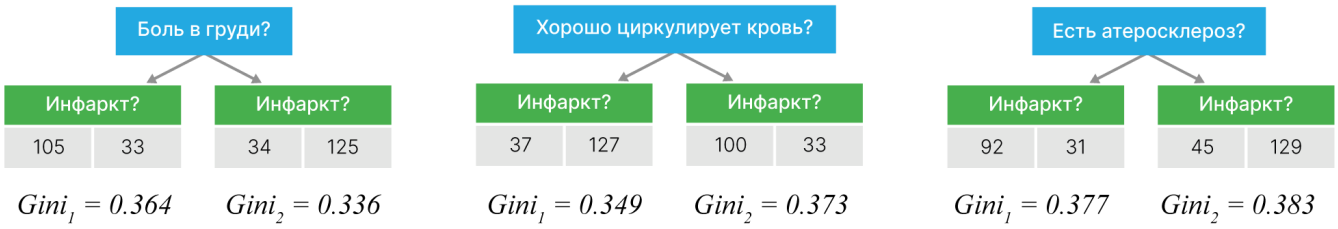
В результате все при разбиении выглядят хорошо, кроме последнего, но вдруг мы что-то не учитываем.

Что с этим делать? Как из этих 3 разделений что-то выбрать? Логично выбрать такое разбиение, дает нам "хорошие" узлы - те, в которых преимущественно сосредоточенны объекты одного класса

Одна из используемых метрик называется Gini. Она считается по следующей формуле:

$$Gini = 1 - \sum_i p_i^2$$

Фактически, она показывает, какова вероятность того, что два объекта, попавшие в один узел, принадлежат разным классам. Соответственно, чем она ниже, тем лучше получившийся узел.



$Gini_1 = 0.364$ $Gini_2 = 0.336$ $Gini_1 = 0.349$ $Gini_2 = 0.373$ $Gini_1 = 0.377$ $Gini_2 = 0.383$

Такую метрику можно посчитать для каждого узла. Дальше можно посчитать метрику для узла, который мы били. Дальше посчитать, насколько стал лучше результат в зависимости от используемого признака

$$Impurity_decrease = Gini_0 - \left(\frac{n_1}{n_1+n_2}Gini_1 + \frac{n_2}{n_1+n_2}Gini_2\right)$$

n1, n2 - число объектов в листьях
Gini_0 - чистота исходного узла

$$p = \frac{105+33}{105+33+34+125} = 0.880$$ **Доля объектов 0класса до разбиения**

$$Gini_0 = 0.498$$

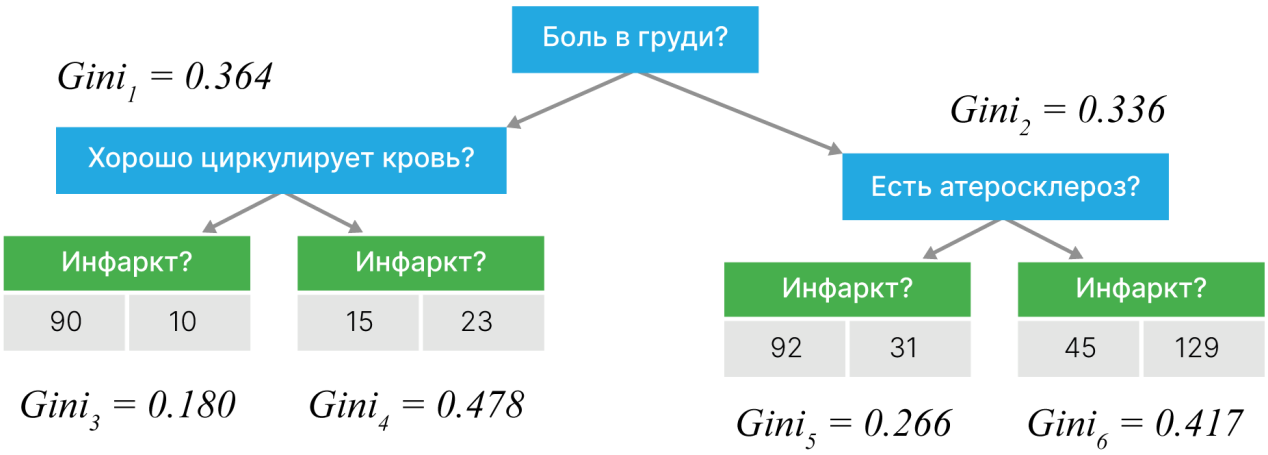
$$Gini_1 = 0.364 \qquad Gini_2 = 0.336$$

$$Impurity_decrease = 0.149$$

Выбираем это разбиение как приводящее к наибольшему уменьшению impurity

Оказывается, что наибольший impurity_decrease в признаке “боль в груди”. Значит, мы возьмем “боль в груди”, как признак, на основании которого продолжим строить дерево.

Теперь каждое из этих листьев, которые у нас получились, мы можем разбить. Имеет ли смысл их бить, используя тот же признак (“боль в груди”)? Очевидно, нет, потому что мы по нему уже все разделили. Но можно использовать 2 других признака.



Каждый раз мы будем выбирать новые признаки. В одном листе один признак лучше бьет его на 2, в другом — другой. В результате получим итоговое дерево, где все признаки

используются по одному разу.

Можно ли остановиться раньше? В принципе да. Например, когда у нас получатся листья, в которых есть объекты только 1 класса. В этом случае не имеет смысла использовать другие разбиения. Либо может сложиться ситуация, когда разбиение по признаку, которое мы дополнительно взяли, ситуацию никак не улучшает, и дерево предсказывает настолько же плохо, как если бы мы его не использовали. Но это редкая ситуация.

Для вещественных признаков

Как работать с вещественными переменными?

Допустим, у нас есть вещественные примеры (например, измерения). Что делать с ними? Берем сетевой угол переменной, для каждого порога делаем разбиение (как делали ранее для бинарного признака) и считаем impurity_decrease. Из всех этих разбиений можно выбрать то, которое дает наилучший результат.

Дальше мы среди всех вещественных признаков можем выбрать тот, у которого наилучшее разбиение дает наилучший impurity.

Для категориальных признаков

Мы разобрали бинарные и вещественные признаки.

Также существуют категориальные признаки — это когда признаки принимают несколько значений, но при этом мы не можем их как-то упорядочить. Например, цвет, порода собаки, город и так далее. Нельзя сказать, что зеленый больше красного. Но с такими признаками тоже надо как-то работать.

Нейронные сети работают с категориальными признаками плохо, обычно необходимо специальным образом предобработать данные, чтобы нейронная сеть смогла с ними работать. Напрямую закодировать

Если у нас есть категориальной признак, у которого N возможных значений (например, N цветов), то есть мы можем разбить наши объекты на 2 группы 2^N возможных способов. То есть красный и синий идут в одну сторону, а белый и желтый в другую. Таким образом, мы можем выбрать, идет какой-то цвет в определенный лист или нет. Отсюда получается

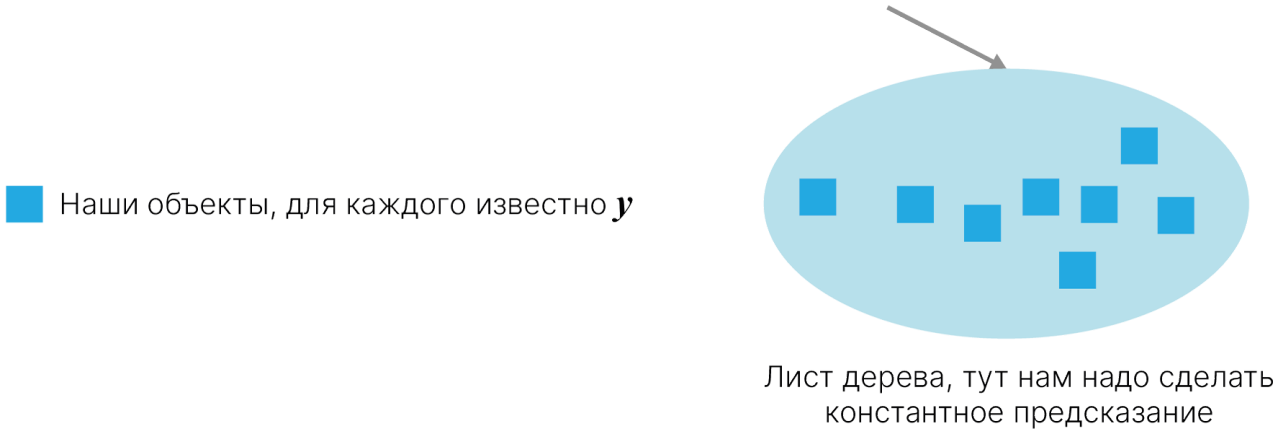
порядка 2^N возможных разбиенией. Перебирать все возможные разбиения - долго и для категориальных признаков с большим числом возможных категорий неприменимо.

$$E_{x_i==k}(y)$$

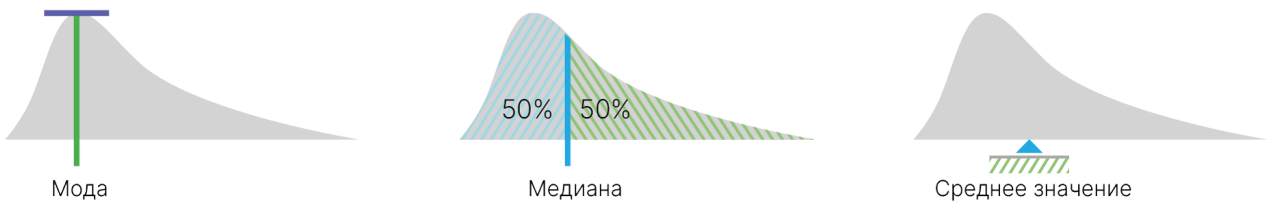
Среднее значение y для объектов, у которых категориальная переменная x_i равно данному значению

Но мы не предсказываем категориальный признак, а используем категориальный признак для предсказания чего-то. Можем взять и посчитать для каждой категории категориального признака среднее значение целевой переменной. Дальше отсортировать эти категории по полученному среднему значению. Окажется, что оптимальное разбиение будет одним из тех, которые мы получим в данном случае. Все остальные разбиения гарантированно не являются оптимальными. Получается, что для категориальных переменных мы легко находим правильный способ разбиения.

Деревья решений (Регрессия)



Для решения задач регрессии дерево строится практически точно так же, но есть несколько нюансов. Во-первых, в листе нужно предсказывать не класс, а какое-то значение. Тут опять возникает вопрос: в области пространства мы знаем какие-то объекты. Им соответствует какое-то значение. Как лучше всего одним числом охарактеризовать все эти объекты?



Метрики регрессии

Если мы вспомним статистику, у нас нет однозначного ответа. То есть мы можем для распределения всех этих "y" предсказывать наиболее частое значение, медиану, среднее значение. Обычно предсказывают среднее значение, потому что с ним легче всего работать и, опять же, среднее чаще всего используют в статистике.

Опять же, фактически хотим оценить среднее всех объектов генеральной совокупности, попадающих в данный узел, используя объекты из тренировочной выборки, попавшие в данный узел.

$$\overline{x} = \frac{\sum_i X_i}{n}$$

Опять же, у нашей оценки будет дисперсия, которая показывает, насколько сильно мы можем ошибаться по сравнению с реальным средним.

$$D(\overline{x}) = \frac{\sigma^2}{n}$$

Опять же, полученная формула говорит нам о том, что желательно иметь в каждом листе достаточное число объектов, чтобы дисперсия оценки была меньше

Теперь надо сформулировать критерий качества узла для регрессионного дерева. Что мы хотим? Мы хотим, чтобы значения в узле отличались как можно меньше. В этом случае наше среднее будет предсказывать значение для объекта, попавшего в узел, с меньшей ошибкой. В идеале, мы хотим, чтобы все значения были одинаковы.

Какую меру можно использовать? Для этого подумаем, а как мы в итоге обычно оцениваем качество регрессии? При помощи MSE. MSE на тренировочной выборке, если мы предсказываем ее среднее, будет таким:

$$\frac{1}{n}\sum(X_i - \overline{X})^2 = \frac{n}{n-1}D(X)$$

То есть, фактически, нам надо минимизировать дисперсию выборки. Разбиение, которое лучше всех уменьшает дисперсию выборки - то, что нам нужно. Единственное - будем взвешивать дисперсии на размер узла. Иначе самыми выгодными будут разбиения, отправляющие в один из узлов только один объект. Те из вас, кто помнит формулу объединения дисперсии двух выборок, могут найти в таком взвешивании и статистический смысл.

Разбиение регрессионного дерева

$$R_1(j,s) = \{X | X_j \leq s\} \text{ and } R_2(j,s) = \{X | X_j > s\}.$$

Мера качества узла - дисперсия оценки R0. Остальное - так же, как с классификацией

$$\frac{D_{R_1} \cdot N_1 + D_{R_2} \cdot N_2}{N_1 + N_2} < D_{R_0}$$

Деревья решений и работа с пропущенными значениями

Passengerid	Survived	Pclass	Sex	Age	SibSp	Ticket	Cabin	Embarked
1	0	3	male	22	1	A/5 21171		S
2	1	1	female	38	1	PC 17599	C85	C
3	1	3	female	26	0	STON/O2. 3101282		S
4	1	1	female	35	1	113803	C123	S
5	0	3	male	35	0	373450		S
6	0	3	male		0	330877		Q

Пропущенные значения

Очень сложно, в том числе и в нейронных сетях, бороться с пропущенными значениями. Если значение какого-то признака неизвестно, то любая непрерывная модель, в том числе и нейронные сети, будет без дополнительных ухищрений вести себя очень плохо. Потому что надо как-то объяснить модели, что значения какого-то признака нет.

Это сложно сделать, поскольку с этим признаком должны производиться некие математические операции. И потому обычно отсутствующий признак нельзя оставить так, как есть. Обычно его:

1. на место пропущенных значений записать какие-то числа (заполнить пропущенные значения);
2. либо удаляют все объекты с пропущенными значениями.

Чаще всего в науке так сделать нельзя, потому что данные стоят очень дорого, и потому выбрасывают объекты только потому, что у них нет какого-либо признака, достаточно редко.

Потому обычно прибегают к первому варианту. Как это можно сделать? Можно предположить, что данные не очень сложные, признаки не зависят друг от друга, и заполнить пропущенные значения средними значениями соответствующего признака.

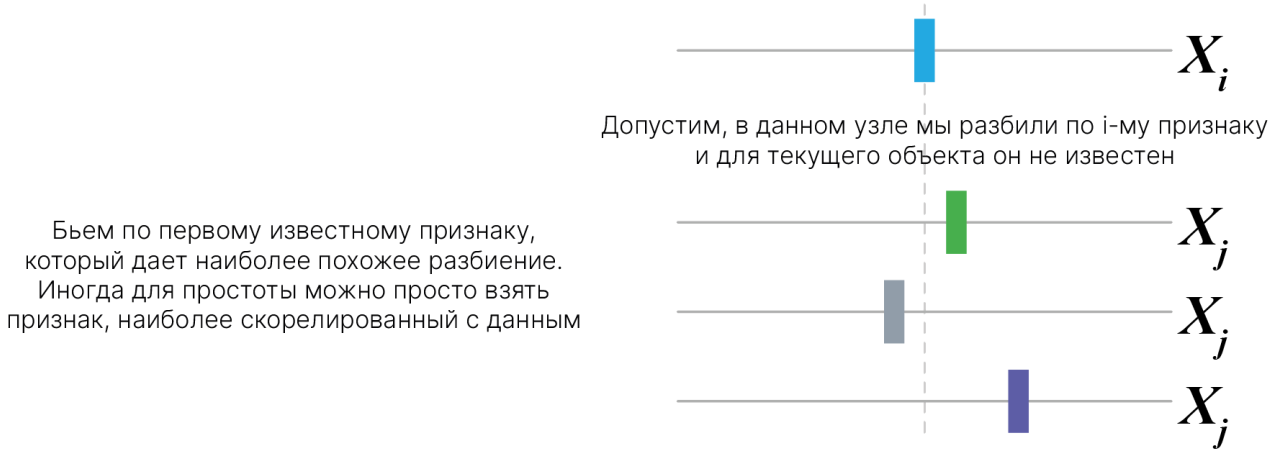
Например, если средний возраст людей 30 лет, то для людей, чей возраст неизвестен, будем писать это же значение.

Но бывают случаи, когда этот способ (среднее, медиана или любая другая взятая величина, которую мы считаем по известным значениям признаков) не работает.

Можно сделать иначе - накидываем дополнительные блоки / модели, которые будут предсказывать пропущенные значения. Если пропущено много признаков, то возникает много проблем, связанных с тем, что надо обучить много моделей: каждая из них совершает ошибки, в итоговым датасете получается много зашумленных данных, не факт, что модель вообще обучится.

Третий вариант — использовать алгоритм, который умеет справляться с пропуском. Один из таких алгоритмов — KNN. Можно брать ближайших соседей по известным признакам и на место неизвестного признака поставить среднее значение. Можно сделать нейросеть, которая будет устойчива к отсутствию какого-то признака. Способ рабочий.

Деревья решений могут бороться с этой проблемой двумя способами. Первый — мы не можем сказать, что это значение какое-то особое для нейронных сетей. Засунуть None в нейросеть нельзя. Дереву можно сказать, что для какого-то значения признака нет, трактуя это как особое правило (выделим такие объекты в особую группу). Это



Дерево решений - суррогатные сплиты

Второй вариант: если мы в этом узле разбивали объекты по данному признаку и для текущего объекта он неизвестен, то можно поискать среди нашей обучающей выборки признаки, которые дают разбиение, похожее на признак с неизвестным значением. Это не надо делать вручную, в некоторых пакетах это реализовано за вас. Это работающее решение, которое явно лучше, чем заполнение пропусков вручную.

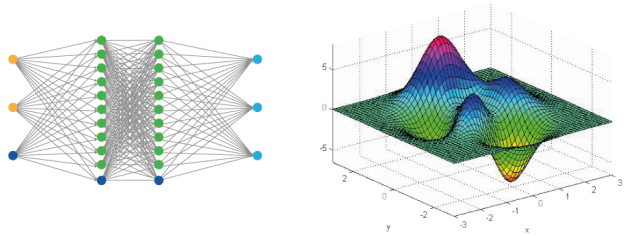
Преимущества и недостатки деревьев решений

Почему деревья - очень мощный метод?

Взгляд точки зрения функционального анализа

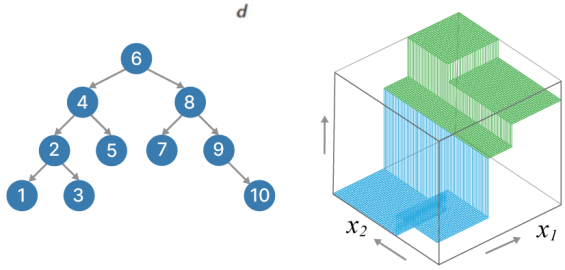
Гладкие функции

$$h(\mathbf{x}) = \sum \sigma(\ldots \sum \sigma(\mathbf{w}^T \mathbf{x}))$$



Кусочно-постоянные функции

$$h(\mathbf{x}) = \sum_d c_d I\{\mathbf{x} \in R_d\}$$



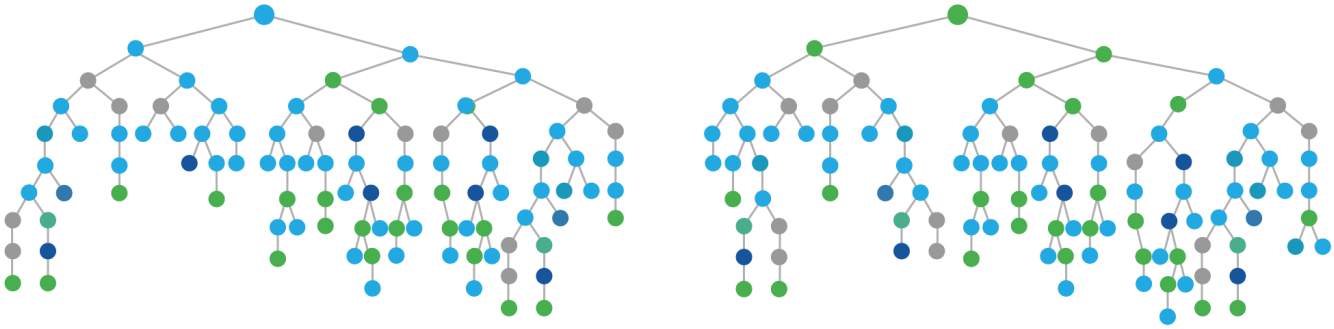
У деревьев решений есть еще одно хорошее свойство. На первом занятии вы познакомились с замечательный теоремой об универсальном аппроксиматоре. Ее суть в том, что если взять нейросеть, и у нее будет 2 скрытых слоя, то она сможет аппроксимировать в любую заданную гладкую функцию.

Для деревьев решений есть аналогичная теорема, говорящая о том, что дерево может аппроксимировать в любую заданную кусочно-постоянную функцию.

Дерево решений, в отличие от нейронной сети, может адаптироваться к выборке любого размера, любому количеству признаков. Для нейронной сети эта теорема предполагает, что можно бесконечно увеличивать скрытые слои. В реальной жизни так не делают.

В случае же с деревьями решений лист разбивается еще на 2 части. Это можно делать бесконечно до тех пор, пока в каждом листе не окажется по одному объекту. Это приведет к сильному переобучению, но в принципе, если бы у нас была вся генеральная совокупность, мы бы выучили ее гораздо легче деревом решений, чем нейронной сетью.

Неустойчивость деревьев решений



Деревья решений не используется в чистом виде, потому что они неустойчивы. Если у нас есть данные, и мы выкинем из данных 2 объекта, то дерево решений может очень сильно поменяться. Красивого дерева, как на примере в самом начале, у нас не

получится. Будет мешанина, которая может в любой момент поменяться, если мы

Продемонстриуем неустойчивость решения, получаемого при помощи деревьев решений на примере датасета iris (ирисы Фишера).

Ирисы Фишера состоят из данных о 150 экземплярах ириса, по 50 экземпляров из трёх видов — Ирис щетинистый (Iris setosa), Ирис виргинский (Iris virginica) и Ирис разноцветный (Iris versicolor)

Для каждого экземпляра измерялись четыре характеристики (в сантиметрах):

1. Длина наружной доли околоцветника (англ. sepal length);
2. Ширина наружной доли околоцветника (англ. sepal width);
3. Длина внутренней доли околоцветника (англ. petal length);
4. Ширина внутренней доли околоцветника (англ. petal width).

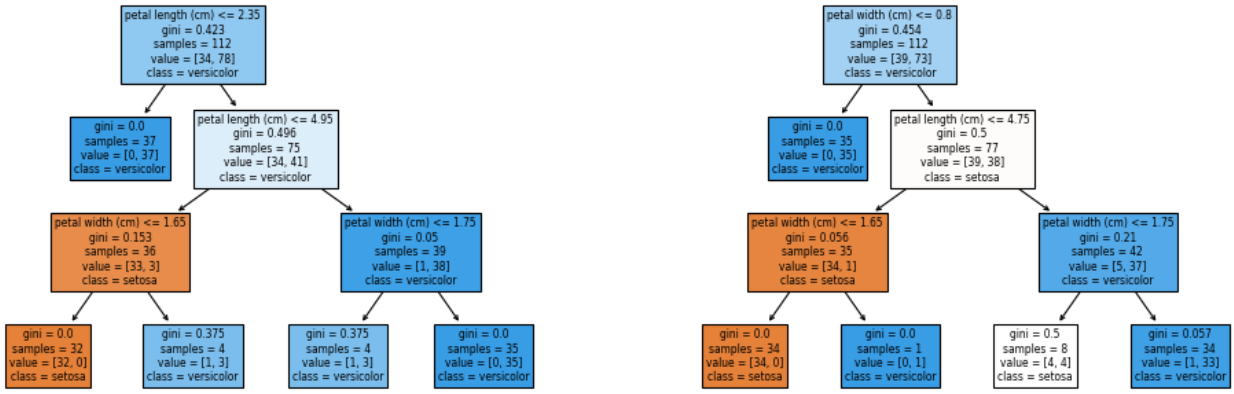
Будем учиться отделять Ирис виргинаский (versicolor) от остальных видов.

```
from sklearn.datasets import load_iris
import pandas as pd

dataset = load_iris()
df = pd.DataFrame(dataset.data, columns=dataset.feature_names)
df['target'] = dataset.target != 1 # 0 for setosa, 1 - versicolor, 2 - virginica
```

Сделаем два разных разбиения на обучение и тест. И посмотрим, будут ли отличаться деревья, построенные для данных разбиений.

```
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
import matplotlib.pyplot as plt
from sklearn import tree

# first set of points
X_train1, X_test1, Y_train1, Y_test1 = train_test_split(df[dataset.feature_names], df['
clf1 = DecisionTreeClassifier(max_depth = 3)
clf1.fit(X_train1, Y_train1)

# second set of points
X_train2, X_test2, Y_train2, Y_test2 = train_test_split(df[dataset.feature_names], df['
clf2 = DecisionTreeClassifier(max_depth = 3)
clf2.fit(X_train2, Y_train2)


fn=['sepal length (cm)','sepal width (cm)','petal length (cm)','petal width (cm)']
cn=['setosa', 'versicolor', 'virginica']
fig, axes = plt.subplots(nrows = 1, ncols = 2, figsize = (15,5), dpi=80)
tree.plot_tree(clf1,
               feature_names = fn,
```

```
                class_names = cn,
                filled = True, ax=axes[0])
tree.plot_tree(clf2,
                feature_names = fn,
                class_names = cn,
                filled = True, ax=axes[1])
plt.show()
```

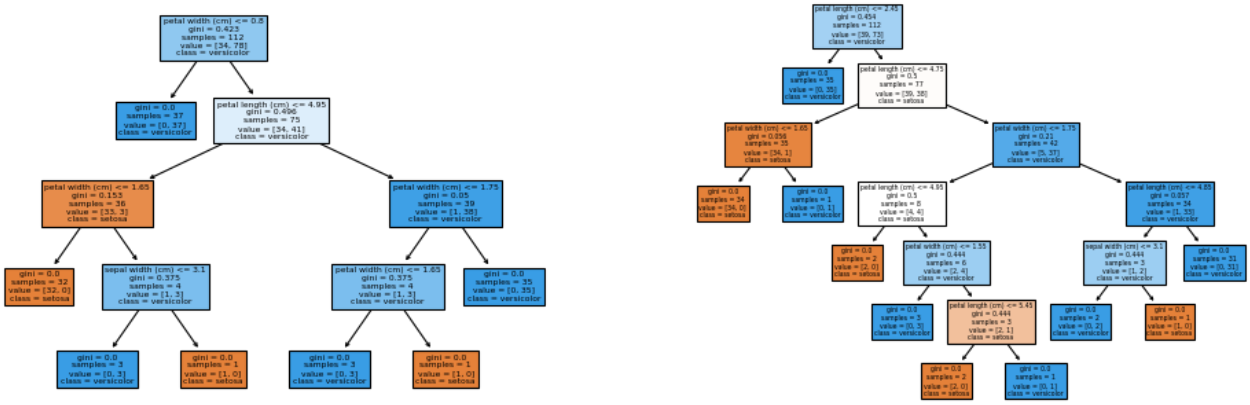


Видим, что даже деревья максимальной глубины 3 уже не совпадают между собой. Конечно, у нас маленький датасет - как правило, чем датасет больше, тем устойчивее будет получаться дерево на первых уровнях. Но часто и деревья используют куда большей глубины.

Если использовать деревья бОльшей глубины, то и структура деревьев (то, как они выглядят, даже если не обращать внимания на конкретные признаки в узлах), будет отличаться

```
# first set of points
clf1 = DecisionTreeClassifier(max_depth = 10, random_state = 0)
clf1.fit(X_train1, Y_train1)

# second set of points
clf2 = DecisionTreeClassifier(max_depth = 10, random_state = 42)
clf2.fit(X_train2, Y_train2)


fn=['sepal length (cm)','sepal width (cm)','petal length (cm)','petal width (cm)']
cn=['setosa', 'versicolor', 'virginica']
fig, axes = plt.subplots(nrows = 1, ncols = 2, figsize = (15,5), dpi=80)
tree.plot_tree(clf1,
                feature_names = fn,
                class_names=cn,
                filled = True, ax=axes[0])
```

```
tree.plot_tree(clf2,
               feature_names = fn,
               class_names=cn,
               filled = True, ax=axes[1])
plt.show()
```



Переобучение деревьев

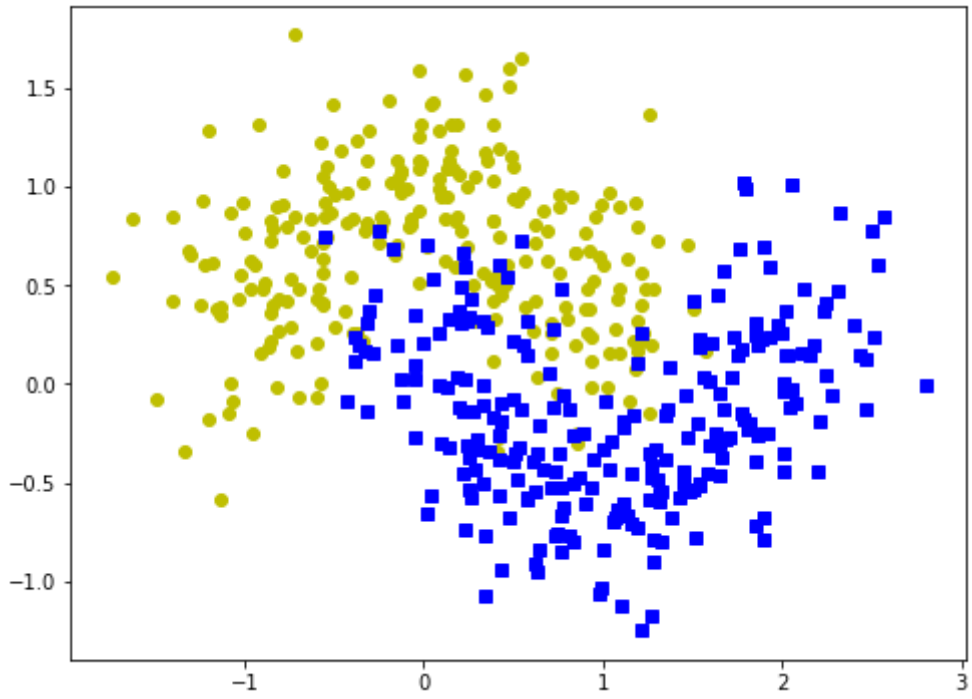
Если алгоритм при небольшом изменении признаков сильно меняет свое решение, это, как правило, указывает на переобучение. Алгоритм сильно реагируют на любой шум в данных - доверять его решениям опасно.

Покажем это на синтетическом датасете

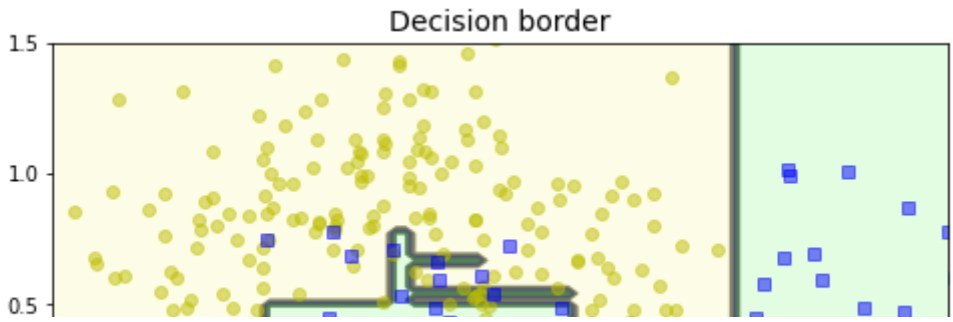
```
#handson-ml
import numpy as np
from matplotlib.colors import ListedColormap

def plot_decision_boundary(clf, X, Y, axes=[-1.5, 2.5, -1, 1.5], alpha=0.5, contour=Tru
    x1s = np.linspace(axes[0], axes[1], 100)
    x2s = np.linspace(axes[2], axes[3], 100)
    x1, x2 = np.meshgrid(x1s, x2s)
    X_new = np.c_[x1.ravel(), x2.ravel()]
    Y_pred = clf.predict(X_new).reshape(x1.shape)
    custom_cmap = ListedColormap(['#fafab0','#9898ff','#a0faa0'])
    plt.contourf(x1, x2, Y_pred, alpha=0.3, cmap=custom_cmap)
    if contour:
        custom_cmap2 = ListedColormap(['#7d7d58','#4c4c7f','#507d50'])
        plt.contour(x1, x2, Y_pred, cmap=custom_cmap2, alpha=0.8)
    plt.plot(X[:, 0][Y==0], X[:, 1][Y==0], "yo", alpha=alpha)
    plt.plot(X[:, 0][Y==1], X[:, 1][Y==1], "bs", alpha=alpha)
    plt.axis(axes)
```

```
    plt.xlabel(r"$x_1$", fontsize=18)
    plt.ylabel(r"$x_2$", fontsize=18, rotation=0)
```

```
import sklearn

X, Y = sklearn.datasets.make_moons(n_samples=500, noise=0.30, random_state = 42)

plt.figure(figsize=(8, 6))
plt.plot(X[:, 0][Y==0], X[:, 1][Y==0], "yo")
plt.plot(X[:, 0][Y==1], X[:, 1][Y==1], "bs")
plt.show()
```




```

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, random_state = 42)

plt.figure(figsize=(8,6))
clf = DecisionTreeClassifier(max_depth = 20, random_state = 42)
clf.fit(X_train, Y_train)
plot_decision_boundary(clf, X, Y)
plt.title("Decision border", fontsize=14)
plt.show()
```

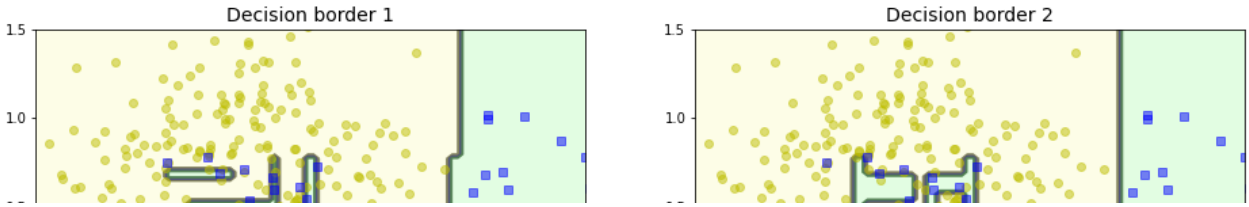


В областях, покрашенных в желтый, модель будет классифицировать точки как точки из 0 класса. В зеленый - как точки из 1го класса.

Обратите внимание на странные, рваные области на рисунке. В этих областях из-за шума, присутствующего в данных, оказались точки неправильного класса. Дерево не смогло их распознать как неправильные и просто выучило, очевидно, неверные правила.

Что произойдет, если мы возьмем точки из того же датасета, но другие?

```
# first set of points
X_train1, X_test1, Y_train1, Y_test1 = train_test_split(X, Y, random_state = 1)
clf1 = DecisionTreeClassifier(max_depth = 20, random_state = 42)
clf1.fit(X_train1, Y_train1)

# second set of points
X_train2, X_test2, Y_train2, Y_test2 = train_test_split(X, Y, random_state = 2)
clf2 = DecisionTreeClassifier(max_depth = 20, random_state = 42)
clf2.fit(X_train2, Y_train2)

plt.figure(figsize=(16,6))
plt.subplot(121)
plot_decision_boundary(clf1, X, Y)
plt.title("Decision border 1", fontsize=14)
plt.subplot(122)
plot_decision_boundary(clf2, X, Y)
plt.title("Decision border 2", fontsize=14)
plt.show()
```



Границы решений поменялись, причем сильно. Исчезли одни "рваные" границы и появились другие. Опять же, наше дерево неустойчиво, из-за малейшего шума в данных оно может поменять свое предсказание. Оно переобучается на шум в данных.

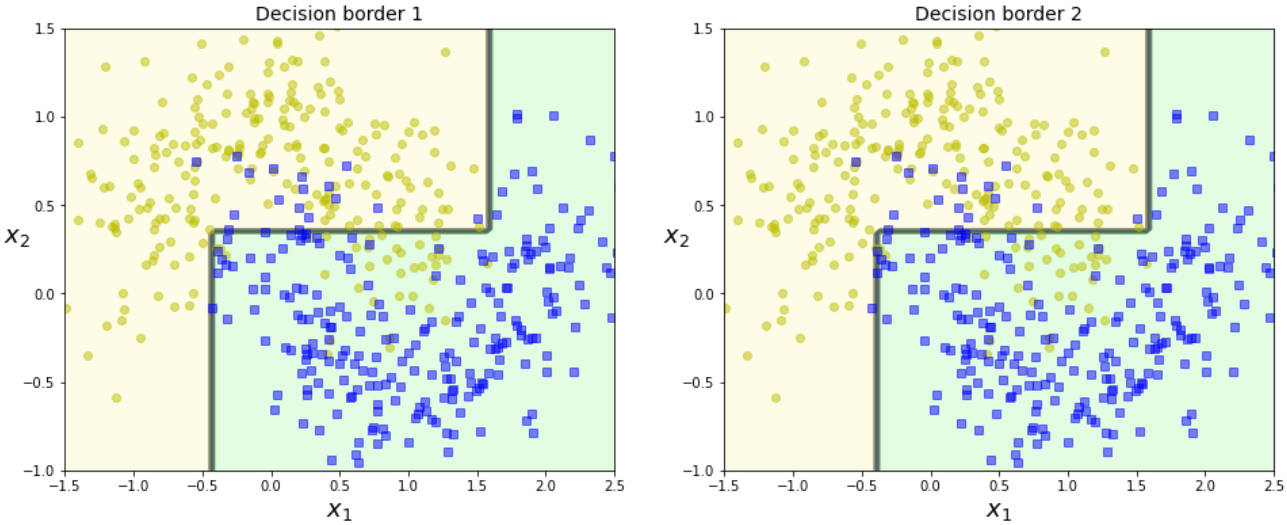
Говорят, что у нашего дерева высокий **variance**.

Можно ли что-то поправить? Ну, у нас в настройках максимальная глубина дерева поставлена равной 20. Это очевидно много. Давайте сделаем дерево глубины 1

```
# first set of points
clf1 = DecisionTreeClassifier(max_depth = 1, random_state = 42)
clf1.fit(X_train1, Y_train1)

# second set of points
clf2 = DecisionTreeClassifier(max_depth = 1, random_state = 42)
clf2.fit(X_train2, Y_train2)

plt.figure(figsize=(16,6))
plt.subplot(121)
plot_decision_boundary(clf1, X, Y)
plt.title("Decision border 1", fontsize=14)
plt.subplot(122)
plot_decision_boundary(clf2, X, Y)
plt.title("Decision border 2", fontsize=14)
plt.show()
```

Или глубины 2

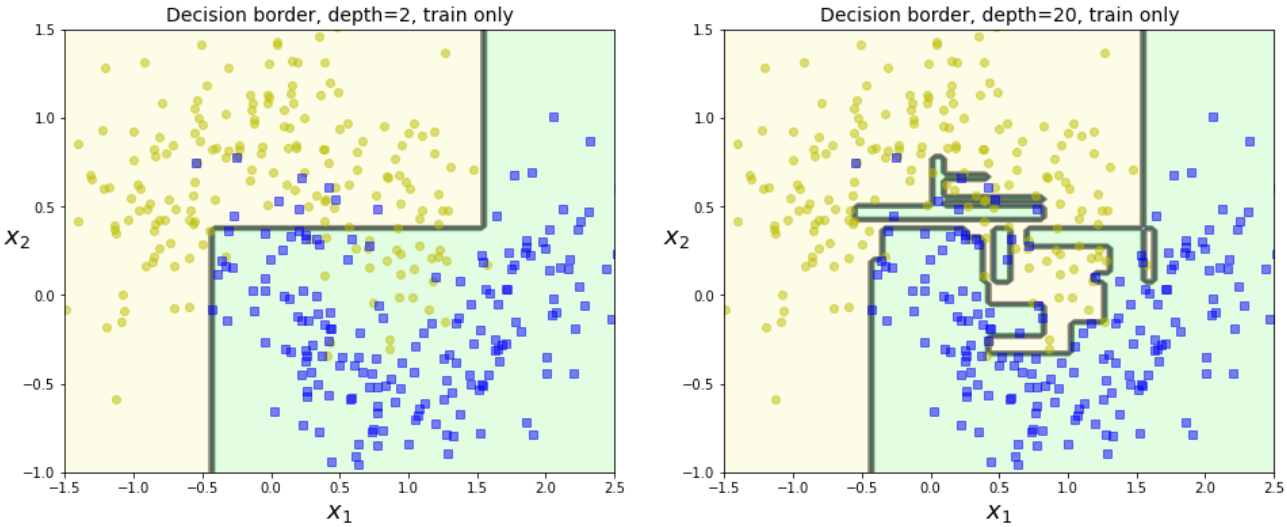
```
# first set of points
clf1 = DecisionTreeClassifier(max_depth = 2, random_state = 42)
clf1.fit(X_train1, Y_train1)

# second set of points
clf2 = DecisionTreeClassifier(max_depth = 2, random_state = 42)
clf2.fit(X_train2, Y_train2)

plt.figure(figsize=(16,6))
plt.subplot(121)
plot_decision_boundary(clf1, X, Y)
plt.title("Decision border 1", fontsize=14)
plt.subplot(122)
plot_decision_boundary(clf2, X, Y)
plt.title("Decision border 2", fontsize=14)
plt.show()
```



Теперь полученные границы решений (почти) совпадают. Но что мы видим? Наше дерево абсолютно не в состоянии (в обоих случаях) уловить закономерность в исходных данных. Если мы отложим только тренировочный датасет, то увидим следующее

```
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, random_state = 42)

clf1 = DecisionTreeClassifier(max_depth = 2, random_state = 42)
clf1.fit(X_train, Y_train)

```

```
clf2 = DecisionTreeClassifier(max_depth = 20, random_state = 42)
clf2.fit(X_train, Y_train)

plt.figure(figsize=(16,6))
plt.subplot(121)
plot_decision_boundary(clf1, X_train, Y_train)
plt.title("Decision border, depth=2, train only", fontsize=14)
plt.subplot(122)
plot_decision_boundary(clf2, X_train, Y_train)
plt.title("Decision border, depth=20, train only", fontsize=14)
plt.show()
```



Мы видим, что в то время как дерево большой глубины выучило нашу тренировочную выборку почти идеально, дерево малой глубины для многих объектов из тренировочной выборки предсказывает не тот класс. Причем, оно не может исправиться просто в силу ограничения на глубину.

В случае дерева с малой глубиной нам не хватает **сложности модели**, чтобы уловить **внутреннюю структуру данных**. Говорят, что у нашей модели высокий **bias**.

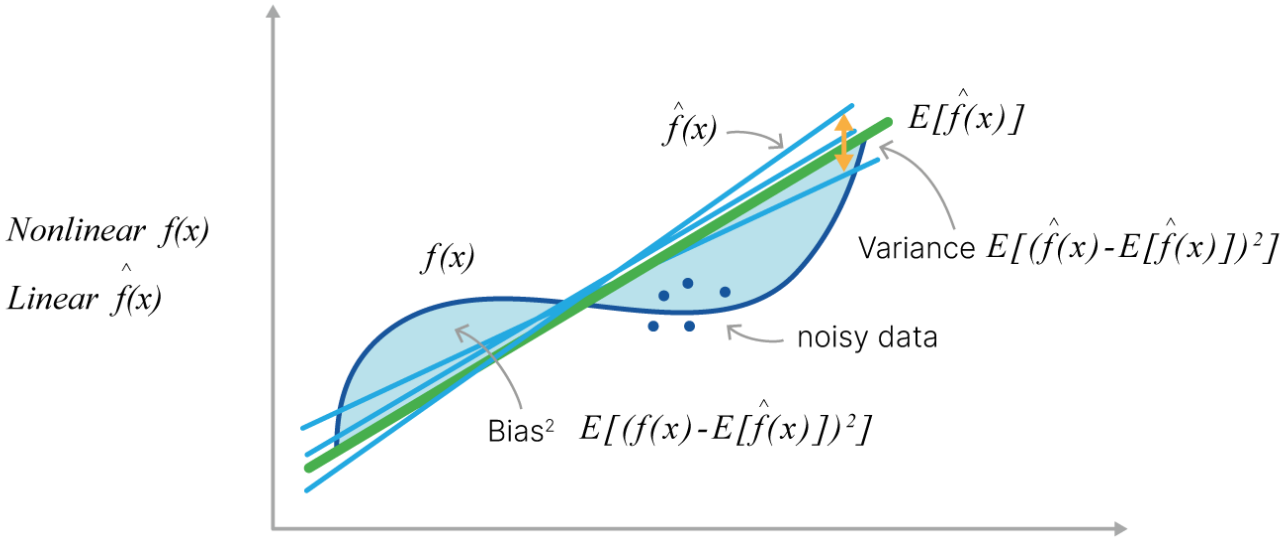
Bias, Variance, Irreducible error

Можно показать, что ошибка любой модели раскладывается в сумму трех компонент:

$$Model_error = Bias^2 + Variance + Irreducible_error$$

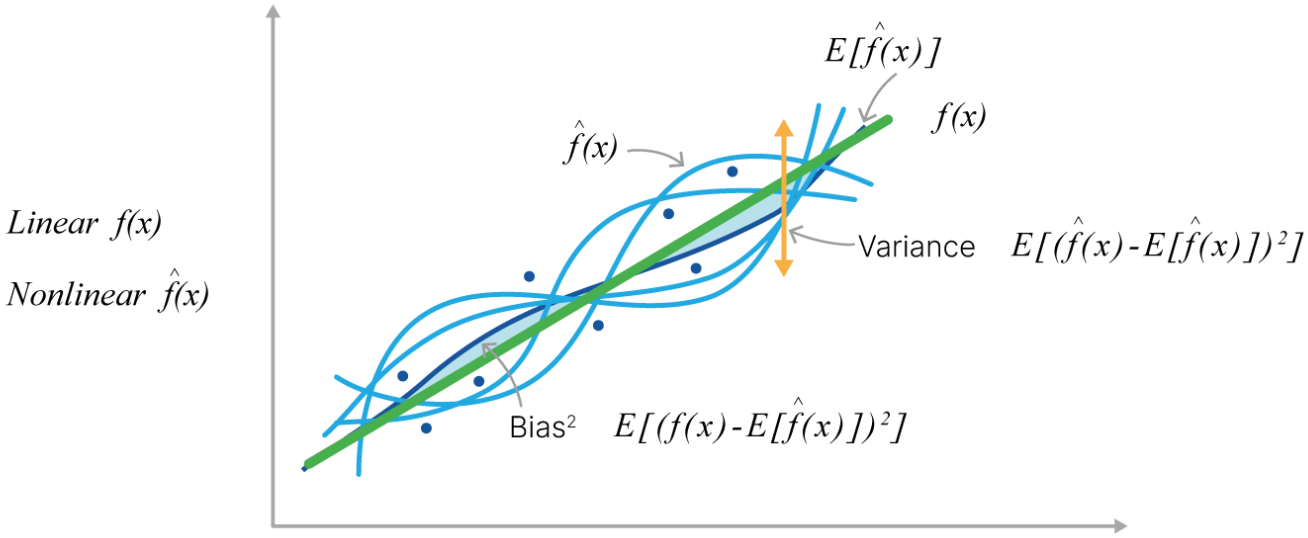
Bias

Обычно, высокий bias имеют недообученные модели. Например, реальная зависимость, которую мы наблюдаем - нелинейная, а мы пытаемся аппроксимировать ее линией. В этом случае наше решение заведомо смещено (biassed) в сторону линейной модели и мы всегда будем ошибаться в сравнении с реальной моделью данных



Variance

Можно получить и обратную ситуацию. Реальная закономерность имеет линейный вид, а мы пытаемся ее аппроксимировать нелинейной моделью. В этом случае мы будем выучивать любой шум в данных и пытаться объяснить его нашей моделью. Малое изменение в данных будет приводить к большим изменениям в прогнозе модели.



Иногда bias и variance представляет еще таким образом:

1. можно быть очень точным и попадать всегда в центр мишени - это соответствует низкому bias и низкому variance;

2. можно попадать примерно в центр мишени, но при этом с большим разбросом - низкий bias, но высокий variance;
3. можно стрелять кучно, но не туда) - это высокий bias и низкий variance;
4. ну а можно просто стрелять наугад, куда душа зовет - это высокий bias и высокий variance.

Irreducible error

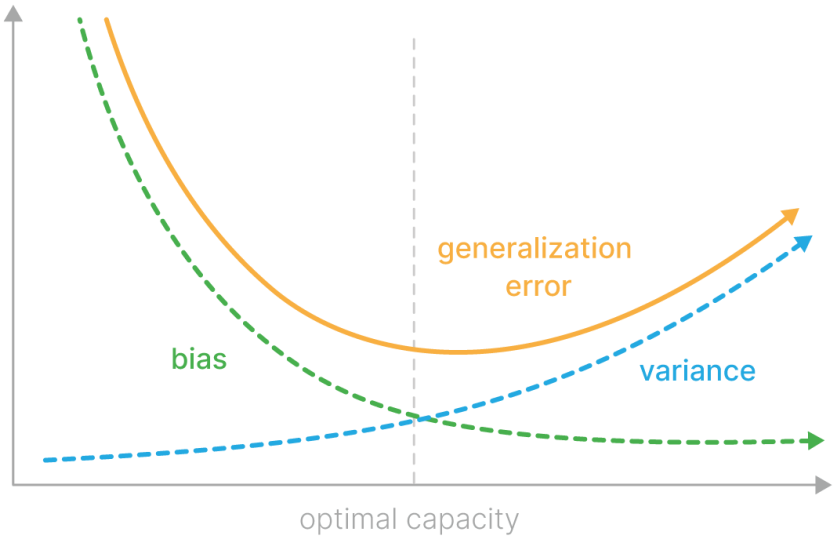
В идеальном для нас случае - когда мы угадали с моделью наших данных, гипотетически можно получить.

$$Bias = 0, Variance = 0$$

Однако, у нас есть ошибки в измерении самой предсказываемой величины. Из-за этого наша модель всегда будет иметь некий уровень ошибки, ниже которого опуститься нельзя.

Bias vs variance

В реальности же, когда реальную модель данных угадать в точности почти невозможно, есть bias-variance tradeoff - нельзя бесконечно уменьшать и Bias, и Variance. Есть какая-то точка оптимума. С какого-то момента при уменьшении Bias начнет увеличиваться Variance, и наоборот. При этом, можно построить связь этих величин с увеличением сложности модели (capacity)



Применительно к деревьям

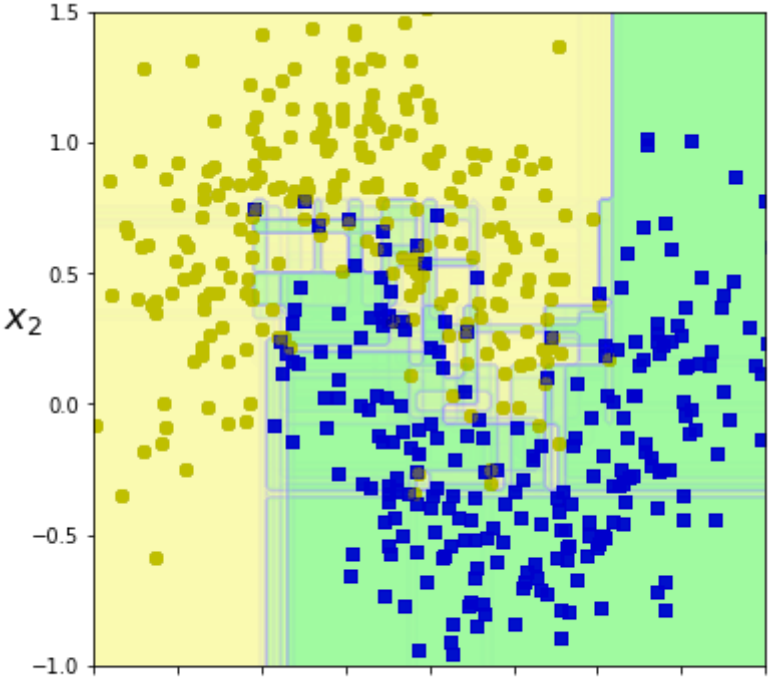
Дерева малой глубины имеет малую сложность - и высокий bias. Дерево большой глубины имеет высокую сложность - и высокий variance.

Можно подобрать для дерева идеальную capacity, когда Bias и Variance будут суммарно давать наименьший вклад в ошибку. Этим мы занимаемся при подборе параметров. Но, оказывается, есть и другие способы борьбы с variance и/или bias, которые мы разберем позже.

Заметим, что если бы могли говорить не просто решение дерева, а привязывать к этому какую-то статистику, например, сколько деревьев, построенных по подобной процедуре, приняли такое же решение - было бы легче.

Если наложить решающие границы 100 решающих деревьев, построенных на разных выборках из X, y, то мы увидим, что "хорошие области", соответствующие реальному разделению данных будут общими между деревьями, а плохие - индивидуальны. К сожалению, в реальности, мы не можем брать бесконечное число наборов данных из генеральной совокупности (представленной в данном случае X, y)

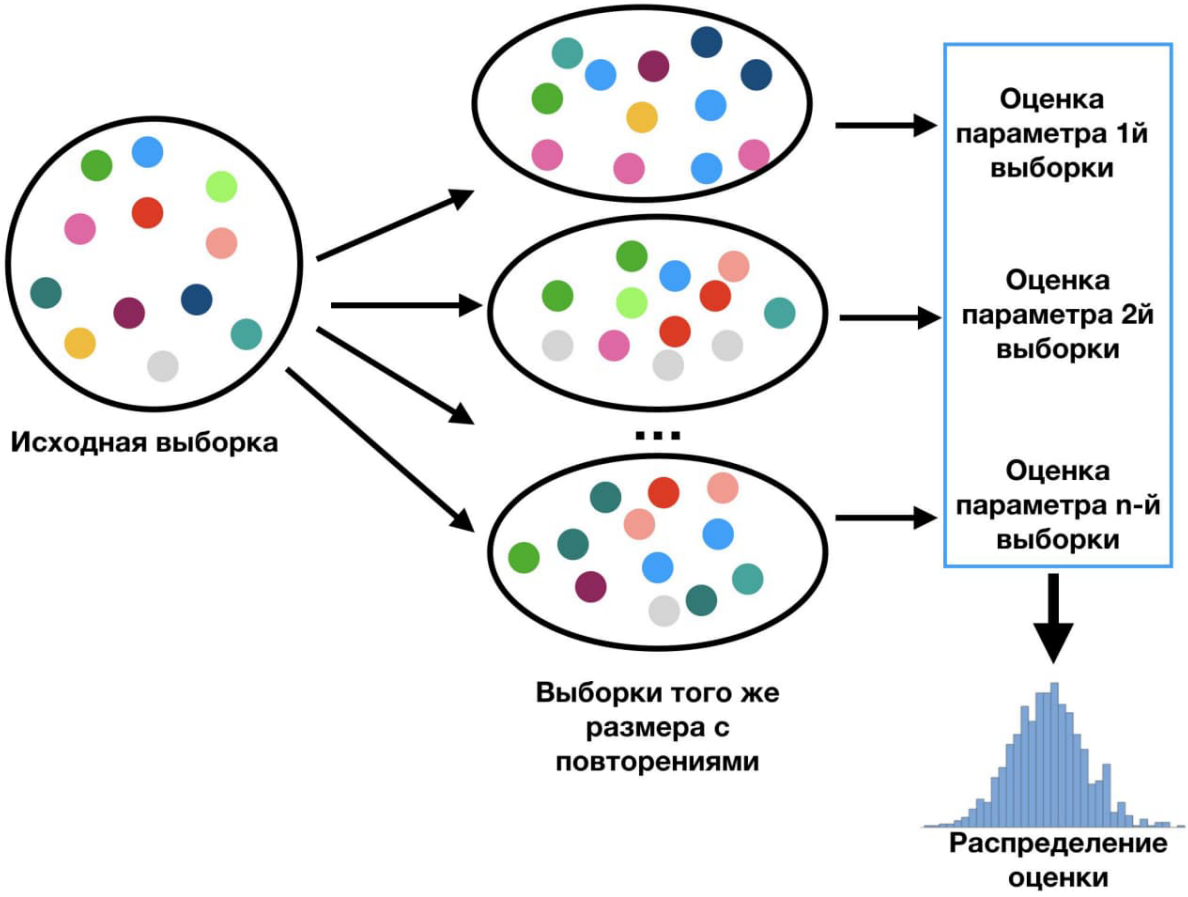
```
plt.figure(figsize=(6,6))

for i in range(1,101):
    X_train, X_test, Y_train, Y_test = train_test_split( X, Y, random_state = i)
    clf = DecisionTreeClassifier(max_depth = 20, random_state = 0)
    clf.fit(X_train, Y_train)
    plot_decision_boundary(clf, X, Y, alpha = 0.02, contour=False)

plt.show()
```



Бутстрэп

Часто мы хотим получить какое-то представление о точности какой-либо нашей оценки - медианы выборки, качества модели, корреляции между двумя переменными и тд. И мы не знаем, как распределена характеристика, которую мы оцениваем.

Есть много подходов к тому, как получить такую оценку и один из них - бутстрэп.



Что мы делаем:

1. Делаем из нашего исходного датасета N выборок такого же размера с повторениями.
2. Для каждой полученной выборки (обычно их называют **псевдовыборками**) считаем характеристику, для которой хотим получить оценку.
3. В результате такой процедуры получаем N значений характеристики. Строим гистограмму этих значений. Получили примерное распределение нашей характеристики.
4. Можем построить 95% доверительный интервал для нашей характеристики - для этого отрезаем 2.5% самых больших значений и самых малых.

Давайте попробуем сделать это на двух практических примерах

Корреляция и построение доверительного интервала для нее

Часто мы хотим понять, как взаимосвязаны две переменные. И часто для оценки этой взаимосвязи используется корреляция. Например, корреляция Пирсона позволяет оценить линейную зависимость между двумя переменными.

Пример 1

Подсчет корреляции Пирсона реализован в Python с помощью функции *scipy.stats.pearsonr*. Посчитаем при помощи этой функции корреляцию между длиной наружной доли околоцветника и внутренней доли околоцветника.

```
import pandas as pd
from sklearn.datasets import load_iris

dataset = load_iris()
df = pd.DataFrame(dataset.data, columns=dataset.feature_names)
df['target'] = dataset.target != 1
```

```
import scipy.stats

cor_value, pval = scipy.stats.pearsonr(df['sepal length (cm)'], df['petal length (cm)']
print(f"Correlation coefficient is {cor_value}")
print(f"P-value is {pval}")
```

```
Correlation coefficient is 0.8717537758865831
P-value is 1.0386674194498099e-47
```

Функция выдает нам одновременно и коэффициент, и его значимость - вероятность наблюдать такое или более критическое значение. Однако на основе чего она считала p-value?

Можно обратиться к документации

```
?scipy.stats.pearsonr
```

p-value считается в предположении, что наши переменные распределены нормально.

Аналогично, при тех же предположениях, при помощи приведенной ниже функции можно подсчитать доверительный интервал для нашего коэффициента корреляции, который дает нам интервальную, а не точечную оценку для значения корреляции.

```
import numpy as np

def pearsonr_ci(X, Y, alpha = 0.05):
    ''' calculate Pearson correlation along with the confidence interval using scipy an
    Parameters
    ----------
    X, Y : iterable object such as a list or np.array
      Input for correlation calculation
    alpha : float
      Significance level. 0.05 by default
    Returns
    -------
    r : float
      Pearson's correlation coefficient
    pval : float
      The corresponding p value
    lo, hi : float
      The lower and upper bound of confidence intervals
    '''

    r, p = scipy.stats.pearsonr(X, Y)
    r_z = np.arctanh(r)
    se = 1/np.sqrt(X.size - 3)
    z = scipy.stats.norm.ppf(1 - alpha / 2)
    lo_z, hi_z = r_z - z * se, r_z + z * se
    lo, hi = np.tanh((lo_z, hi_z))
    return lo, hi
```

```
lo, hi = pearsonr_ci(df['sepal length (cm)'], df['petal length (cm)'])

print(f'Lower bound of confidence interval: {lo}')
print(f'Upper bound of confidence interval: {hi}')
```

```
Lower bound of confidence interval: 0.8270363296643621
Upper bound of confidence interval: 0.9055080488214541
```

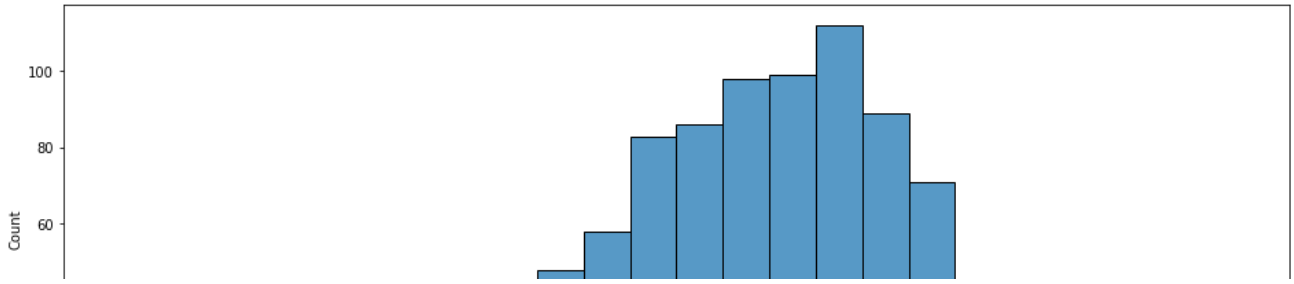
А что, если мы не хотим делать предположения о нормальности? Или знаем, что они не выполняются? Или не хотим писать сложных функций на каждый случай - если x распределено так-то, то считай так-то, если так-то, то так-то и т.д.?

В этом случае нам и поможет бутстрэп. Напишем функцию, которая будет нам для любой характеристики, считаемой для пар из x и y, выдавать значения этой характеристики, посчитанные на псевдовыборках.

```
def bootstrap_metric(X,
                     Y,
                     metric_fn,
                     samples_cnt = 1000,
                     random_state = 42):

    np.random.seed(random_state)
    b_metric = np.zeros(samples_cnt)
    for it in range(samples_cnt):
        poses = np.random.choice(X.shape[0], size=X.shape[0], replace=True)

        X_boot = X[poses]
        Y_boot = Y[poses]

        m_val = metric_fn(X_boot, Y_boot)
        b_metric[it] = m_val

    return b_metric
```

Посчитаем значение корреляции Пирсона на бутстрэп-репликах:

```
import matplotlib.pyplot as plt
import seaborn as sns

boot_cor = bootstrap_metric(X = df['sepal length (cm)'],
                            Y = df['petal length (cm)'],
                            metric_fn=lambda X, Y: scipy.stats.pearsonr(X, Y)[0])

# plot histogram of the obtained values:
plt.figure(figsize=(16, 6))
sns.histplot(boot_cor)
plt.show()
```



Чтобы получить bootstrap 1-alpha доверительный интервал, просто посчитаем alpha/2 и 1-alpha квантили в полученном массиве:

```
alpha = 0.05
lo_2, hi_2 = np.quantile(boot_cor, q=[alpha/2, 1 - alpha/2])

print(f'Lower bound of confidence interval: {lo}, bootstap: {lo_2}')
print(f'Upper bound of confidence interval: {hi}, bootstap: {hi_2}')
```

```
Lower bound of confidence interval: 0.8270363296643621, bootstap: 0.8361068014189094
Upper bound of confidence interval: 0.9055080488214541, bootstap: 0.9034859305858718
```

Получили доверительный интервал, полученный при помощи бутстрапа. Заметим, что точность интервала зависит от числа реплик, которые вы сделаете — чем больше, тем точнее интервал.

Сравнивая с интервалом, подсчитанным в предположении нормальности, видим, что они почти друг от друга не отличаются.

Пример 2

Загрузим данные о людях с подозрением на сердечные заболевания. Допустим, мы хотим узнать, есть ли связь между уровнем холестерина (колонка chol) и возрастом (age)

```
!wget -q https://edunet.kea.su/repo/EduNet-web_dependencies/L03/heart.csv
```

```
heart_dataset = pd.read_csv("/content/heart.csv")
```

```
lo, hi = pearsonr_ci(heart_dataset['age'], heart_dataset['chol'])

print(f'Lower bound of confidence interval: {lo}')
print(f'Upper bound of confidence interval: {hi}')
```

```
Lower bound of confidence interval: 0.10349166489938495
Upper bound of confidence interval: 0.3186831235829968
```

```
cor_value, p_value = scipy.stats.pearsonr(heart_dataset['age'], heart_dataset['chol'])

```

```
print(f'Correlation coefficient is: {cor_value}')
print(f'P-value: {p_value}')
```

```
Correlation coefficient is: 0.2136779565595619
```

Видим, что корреляция есть (доверительный интервал не включает 0). Или где-то есть подвох? На самом деле - да. Возраст точно не распределен нормально. Потому считать доверительный интервал с предположениями о нормальности переменных - не очень хорошо. Посмотрим, что скажет нам bootstrap-интервал:

```
boot_cor = bootstrap_metric(X=heart_dataset['age'],
                            Y=heart_dataset['chol'],
                            metric_fn=lambda X, Y: scipy.stats.pearsonr(X, Y)[0])
lo_2, hi_2 = np.quantile(boot_cor, q=[alpha/2, 1 - alpha/2])


print(f'Lower bound of confidence interval(bootstap): {lo_2}')
print(f'Upper bound of confidence interval(bootstap): {hi_2}')
```

```
Lower bound of confidence interval(bootstap): 0.10081424487923238
Upper bound of confidence interval(bootstap): 0.3125714457058485
```

Видим, что, действительно, полученная корреляция значима. Но теперь этот результат получен без очевидно неверных предположений.

Построение доверительного интервала для качества метрики

Представим теперь, что мы сравним поведение двух моделей на тестовом датасете. Допустим, нам интересует F1-score. Как нам это сделать?

Пример 1 (искусственный)

```
size = 1500
Y = np.random.choice([0, 1], size=size, replace=True)
print(f'shape Y: {Y.shape}')
print(f'First 10 values: {Y[0:10]}')
```

```
shape Y: (1500,)
First 10 values: [0 0 0 0 0 1 1 1 1 0]
```

Напишем функцию, которая имитирует поведение модели, которая угадывает правильный класс в p проценте случаев:

```
def guess_model(Y_real, p):
    guessed = np.random.choice([True, False], size=size, replace=True, p=[p, 1-p])
    Y_pred = np.zeros_like(Y_real)
```

```
    Y_pred[guessed] = Y_real[guessed]
    Y_pred[~guessed] = 1 - Y_real[~guessed]
    return Y_pred
```

Пусть у нас две модели обладают одинаковым качеством, а третья - лучшим:

```
model1 = lambda Y: guess_model(Y, p=0.7)
model2 = lambda Y: guess_model(Y, p=0.7)
model3 = lambda Y: guess_model(Y, p=0.75)

np.random.seed(42)
Y_pred1 = model1(Y)
Y_pred2 = model2(Y)
Y_pred3 = model3(Y)
```

Первый вариант - просто посчитать f1-score каждой модели и проранжировать их в соответствии с f1-score:

```
from sklearn.metrics import f1_score

qual1 = f1_score(y_true=Y, y_pred=Y_pred1)
qual2 = f1_score(y_true=Y, y_pred=Y_pred2)
qual3 = f1_score(y_true=Y, y_pred=Y_pred3)

print(f' qual1: {qual1}\n qual2: {qual2}\n qual3: {qual3}')
```

```
 qual1: 0.6943498978897209
 qual2: 0.697951090548579
 qual3: 0.764505119453925
```

Очевидная проблема - да, нужная нам модель выбрана, но почему мы считаем различие между первой и второй моделью значимыми, а между второй и третьей - нет?

```
print(f'qual2 - qual1: {qual2 - qual1}\nqual3 - qual2: {qual3 - qual2}')
```

```
qual2 - qual1: 0.0036011926588580545
qual3 - qual2: 0.06655402890534601
```

Получается, такой способ (при помощи точечной оценки), может привести к ошибке, так как не дает нам судить о **значимости** отличий.

Существует много способов посчитать значимость данного отличия. Мы рассмотрим способ сравнения на основе bootstrap. Об остальных можете почитать в обзоре, приведенном в списке литературы.

Способ, собственно состоит в применении бутстрэпа к предсказаниям модели и реальным меткам.

```
boot_f1score_m1 = bootstrap_metric(Y, Y_pred1, metric_fn=lambda X, Y: f1_score(y_true=X
boot_f1score_m2 = bootstrap_metric(Y, Y_pred2, metric_fn=lambda X, Y: f1_score(y_true=X
boot_f1score_m3 = bootstrap_metric(Y, Y_pred3, metric_fn=lambda X, Y: f1_score(y_true=X
```

Построим 90% доверительный интервал качества для каждой модели:

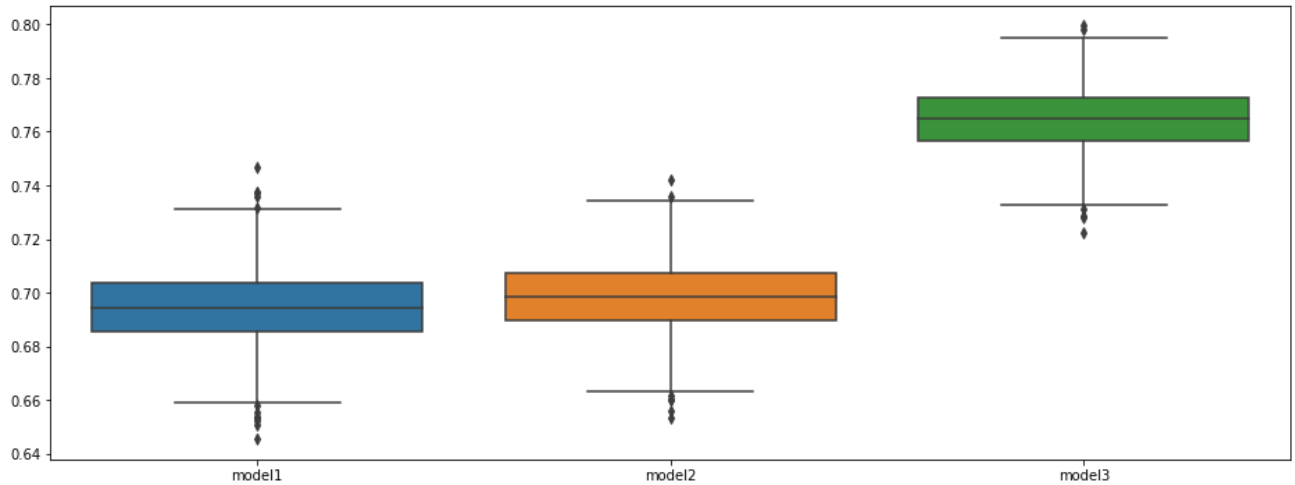
```
alpha = 0.10
print("F1 score for the 1st model: ", np.quantile(boot_f1score_m1, q=[alpha/2, 1 - alph
print("F1 score for the 2st model: ", np.quantile(boot_f1score_m2, q=[alpha/2, 1 - alph
print("F1 score for the 3st model: ", np.quantile(boot_f1score_m3, q=[alpha/2, 1 - alph
```

```
F1 score for the 1st model:  [0.6706605  0.71665785]
F1 score for the 2st model:  [0.67613252 0.71965134]
F1 score for the 3st model:  [0.74469469 0.78326985]
```

Теперь мы видим, что доверительные интервалы для качества первой и второй модели практически одинаковы, в то время как доверительный интервал для качества третьей модели от них сильно отличается и не пересекается.

Можем построить боксплот:

```
plt.figure(figsize=(16,6))
sns.boxplot(y=np.concatenate([boot_f1score_m1, boot_f1score_m2, boot_f1score_m3]),
            x=["model1"] * 1000 + ['model2'] * 1000 + ['model3'] * 1000)
plt.show()
```



Пример 2 (классифицируем людей с больным сердцем и нет)

```
from sklearn.model_selection import train_test_split
```

```
from sklearn.model_selection import train_test_split

X = heart_dataset.drop("target", axis=1)
Y = heart_dataset['target'] > 0
X_train, X_test, Y_train, Y_test = train_test_split(X, Y.values, random_state = 42)
```

```
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV

svc_model = GridSearchCV(SVC(), {'kernel':('linear',
                                            'rbf'),
                                 'C':[0.01, 0.1, 1, 10]}
                         ).fit(X_train, Y_train)

logr_model = GridSearchCV(LogisticRegression(solver='liblinear', max_iter=100000),
                          {'penalty':('l1',
                                      'l2'),
                            'C':[0.01, 0.1, 1, 10, 100]}
                          ).fit(X_train, Y_train)

# few objects in the leaf - poor estimates of class probabilities - the model is overtr
dt_model = GridSearchCV(DecisionTreeClassifier(),
                          {'max_depth':[1, 3, 5, 7, 10],
                           'min_samples_leaf': [1, 3, 5, 10]}
                        ).fit(X_train, Y_train)

```

```
from sklearn.metrics import average_precision_score # PR-AUC

Y_pred1 = svc_model.decision_function(X_test) # by default, SVM gives score to each obj
Y_pred2 = logr_model.predict_proba(X_test)[:, 1]
Y_pred3 = dt_model.predict_proba(X_test)[:, 1]

qual1 = average_precision_score(y_true=Y_test, y_score=Y_pred1)
qual2 = average_precision_score(y_true=Y_test, y_score=Y_pred2)
qual3 = average_precision_score(y_true=Y_test, y_score=Y_pred3)
```

```
print(f"Logistic regression pr-auc: {qual1:.03f}")
print(f"SVC pr-auc: {qual2:.03f}")
print(f"DecisionTreeClassifier pr-auc: {qual3:.03f}")
```

```
Logistic regression pr-auc: 0.893
SVC pr-auc: 0.902
DecisionTreeClassifier pr-auc: 0.801
```

Теперь подсчитаем бутстрэп-оценки. Обратите внимание - теперь мы передаем не предсказания, а вероятности.

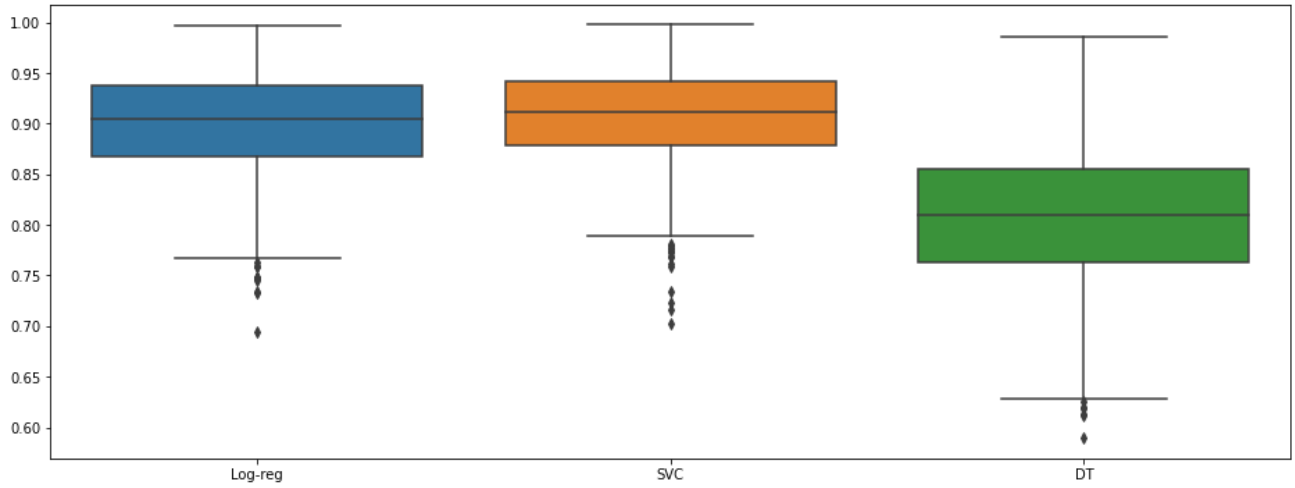
```
boot_score_logreg = bootstrap_metric(Y_test,
```

```
                                  Y_pred1,
                                  metric_fn=lambda X, Y: average_precision_score(y_tru
boot_score_svc = bootstrap_metric(Y_test,
                                  Y_pred2,
                                  metric_fn=lambda X, Y: average_precision_score(y_tru
boot_score_dt = bootstrap_metric(Y_test,
                                  Y_pred3,
                                  metric_fn=lambda X, Y: average_precision_score(y_tru

alpha=0.10
print("Logistic regression pr-auc 90%-ci: ", np.quantile(boot_score_logreg, q=[alpha/2,
print("SVC pr-auc 90%-ci:", np.quantile(boot_score_svc, q=[alpha/2, 1 - alpha/2]))
print("DecisionTreeClassifier pr-auc 90%-ci:", np.quantile(boot_score_dt, q=[alpha/2, 1
```

```
Logistic regression pr-auc 90%-ci:  [0.80793003 0.97363727]
SVC pr-auc 90%-ci: [0.8276533  0.97115084]
DecisionTreeClassifier pr-auc 90%-ci: [0.70049707 0.90513571]
```

Видим, что качество SVC и логистической регрессии почти не отличается, а дерево решений уступает обеим моделям.

```
plt.figure(figsize=(16,6))
sns.boxplot(y=np.concatenate([boot_score_logreg,
                              boot_score_svc,
                              boot_score_dt]),
            x=["Log-reg"] * 1000 + ['SVC'] * 1000 + ['DT'] * 1000)
plt.show()
```



Ансамбли

Корректирующий код

Пусть у нас есть сигнал:

1110110011

Но при передаче на другое устройство в нем могут возникать ошибки.

1010111011

Самое простое решение возникшей проблемы:

1. Шум, который вносит ошибки, скорее всего не зависит от места в сигнале;
2. Передадим 3 раза один и тот же сигнал;

1010111011

1110010011

1100110111

3. Усредним, что получилось (в каждом случае возьмем наиболее часто встречающуюся цифру);

1110110011

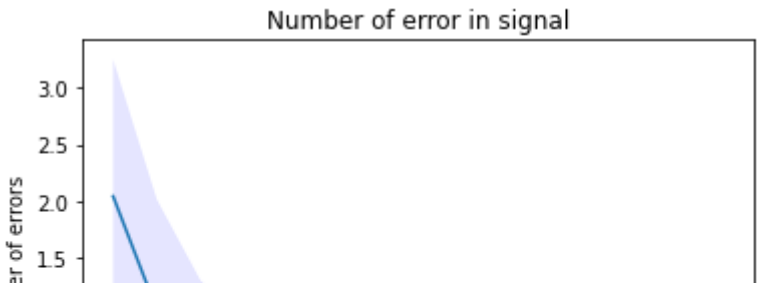
4. С большой долей вероятности итоговый сигнал восстановится;
5. Чем больше копий сигналов передастся, тем выше вероятность, что сигнал восстановится полностью корректно.

Напишем код, чтобы удостовериться в наших выводах.

```
import numpy as np

def get_signal(size, random_state = 42):
    signal = np.random.choice([0,1], size, replace=True)
```

```
    signal = np.random.choice([0,1], size, replace=True)
    return signal

def compare_signs(sig1, sig2):
    return (sig1 != sig2).sum()

def add_noise(sig, noise_p = 0.20):
    sig = sig.copy()
    changed = np.random.choice([True, False], sig.shape[0], replace=True, p = [noise_p,
    sig[changed] = 1 - sig[changed]
    return sig

def average_signals(sigs):
    sig = np.mean(sigs, axis=0)
    sig = np.round(sig, 0)
    return sig

def send_signal(signal, tries):
    passed_sigs = [add_noise(signal) for _ in range(tries)]
    fin_signal = average_signals(passed_sigs)
    return fin_signal
```

```
import matplotlib.pyplot as plt

np.random.seed(42)
repeats = 1000
signals_cnt_rng = range(1, 30, 2)

signal = get_signal(10)
mistakes = np.zeros((repeats, len(signals_cnt_rng)))

for j, sig_cnt in enumerate(signals_cnt_rng):
    for i in range(repeats):
        rec_sig = send_signal(signal, sig_cnt)
        mistakes[i, j] = compare_signs(rec_sig, signal)


mn = mistakes.mean(axis=0)
sd = mistakes.std(axis=0)
plt.title("Number of error in signal")
plt.ylabel("Number of errors")
plt.xlabel("Number of signals passed at once")
plt.plot(signals_cnt_rng, mn)
plt.fill_between(signals_cnt_rng, mn - sd, mn+sd,  facecolor='blue',
                 alpha=0.1)
plt.show()
```



Оказывается, это имеет отношение к проблеме, с которой мы столкнулись с деревьями решений.

Постановка задачи:

Есть 10 объектов, в реальности все принадлежат классу 1

1111111111

Пусть у нас есть три **независимых** классификатора A, B и C. Каждый предсказывает 1 в 70% случаев.

Мы хотим получить общий классификатор на основании этих трех.

Мы хотим получить предсказание базовых классификаторов, применить к ним какую-то функцию, которая выдаст итоговый ответ. Вид этой функции задается заранее.

Усреднение предсказания классификаторов

Будем просто усреднять предсказание наших классификаторов

$$h(x) = \frac{1}{T}\sum_{i=1}^{T} a_i(x)$$

Простое голосование

Допустим, у нас 3 классификатора, которые предсказывают независимо друг от друга.

Посчитаем вероятность того, что:

1. все три классификатора верны: 0.7 * 0.7 * 0.7 = 0.3429
2. два классификатора верны: 0.7 * 0.7 * 0.3 + 0.7 * 0.3 * 0.7 + 0.3 * 0.7 * 0.7 = 0.4409

Таким образом, если брать большинство голосов, то мы будем в 78% случаев предсказывать верно. Мы взяли 3 классификатора, которые сами по себе были не очень хорошими, и получили классификатор лучшего качества. Очевидно, что если взять 4, 5, 10 классификаторов, то ситуация будет становиться лучше.

Пусть теперь у нас три классификатора, выдающие следующие предсказания

1111111100 - 80% точность

1111111100 - 80% точность

1011111100 - 70% точность

Если объединим предсказания, то получим:

1111111100 - 80% точность

Потому что очень **высокая зависимость предсказаний**. Выше видно, что два классификатора предсказывают абсолютно одинаково. Вероятность, что они делают это случайно, очень мала.

А вот если возьмем такие классификаторы, то все получится:

1111111100 - 80% точность

0111011101 - 70% точность

1000101111 - 60% точность

Усреднение:

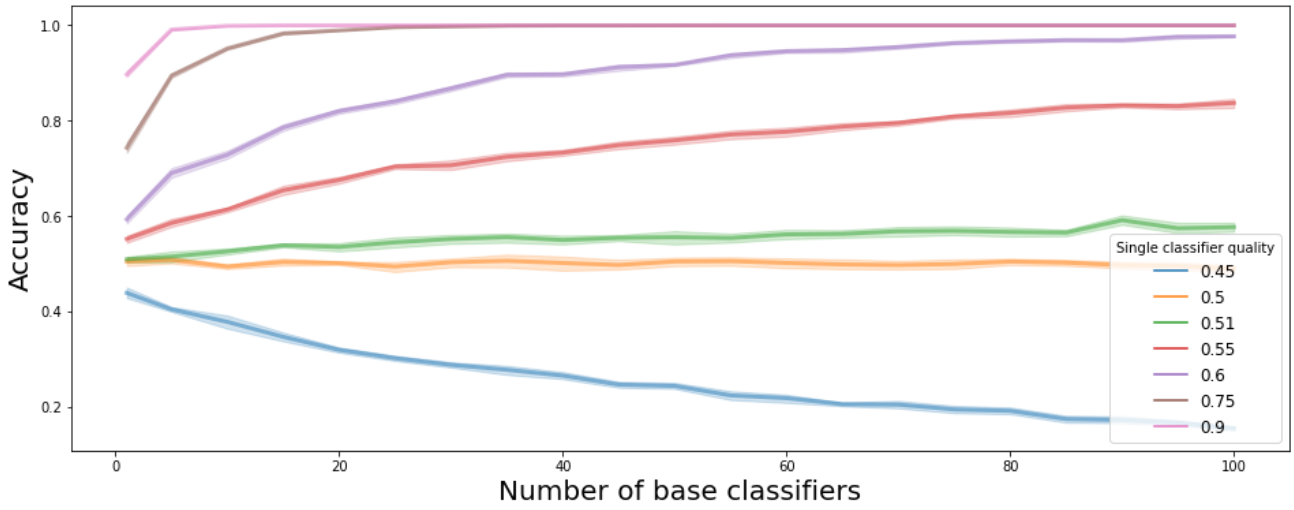
1111111101 - 90% точность

Эксперимент: зависимость качества ансамбля от качества индивидуального предсказателя и от числа предсказателей

```
def get_predictions(Y_real, p, cnt):
    size = Y_real.shape[0]
    guessed = np.random.choice([True, False], (cnt, size), p=[p, 1-p])
    Y = np.repeat(Y_real.reshape(1, -1), cnt, axis=0)
    Y[~guessed] = 1 - Y[~guessed]
    return Y
```

```
import pandas as pd
import seaborn as sns

size = 1000
reps = 10

cnt_base_predictors = [1] + list(range(5, 105, 5))
single_qual = [0.45, 0.5, 0.51, 0.55, 0.6, 0.75, 0.9]

dt = {"cnt":[], "single_qual":[], "accuracy":[]}

for i in range(reps):
    Y_real = np.random.choice([0,1], size)
    for cnt in cnt_base_predictors:
        for p in single_qual:

            preds = get_predictions(Y_real, p, cnt)
            voting =  np.round(preds.mean(axis=0))
            accuracy = (Y_real == voting).mean()
            dt['cnt'].append(cnt)
            dt['single_qual'].append(f"{p:.02}")
            dt['accuracy'].append(accuracy)

results = pd.DataFrame(dt)


plt.figure(figsize=(16,6))

sns.lineplot(data=results, x='cnt', y='accuracy', hue='single_qual', lw=3, alpha=0.5)
```

```
plt.xlabel("Number of base classifiers", size=20)
plt.ylabel("Accuracy", size=20)
plt.legend(loc='best', fontsize=12, title="Single classifier quality")
plt.show()
```



Видим:

1. чем лучше базовый классификатор, тем меньше нужно классификаторов при прочих равных для достижения большего качества;
2. если качество базового классификатора даже чуть больше 0.5, то качество ансамбля растет с увеличением числа моделей в ансамбле;
3. если качество базового классификатора неотличимо от случайного (0.5), то качество ансамбля будет оставаться равным 0.5;
4. если качество базового классификатора ниже случайного (0.5), то качество случайного классификатора стремится к 0.

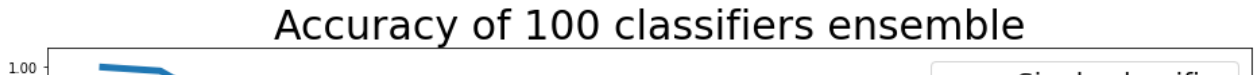
Эксперимент: коррелированность моделей

Посмотрим, как зависит качество предсказания от коррелированности предсказателей. Конкретно — от ожидаемой коррелированности вероятностей ошибиться на данном объекте для любой взятой пары классификаторов из ансамбля.

```
import scipy

def get_correlated_predictions(Y_real, p, cnt, r):
    size = Y_real.shape[0]
```

```
    x1 = np.random.uniform(0, 1, size)
    x2 = np.random.uniform(0, 1, (cnt, size))
    q = np.sqrt(r)
    Y = q * x1 + (1 - q ** 2) ** 0.5 * x2 # Y variables now correlated with correlation
    Y_mod = np.zeros_like(Y)
    for i in range(Y.shape[0]):
      Y_mod[i] = scipy.stats.rankdata(Y[i])

    Y =  Y_mod / size # back to uniform, slightly affects correlations

    Y_pred = np.repeat(Y_real.reshape(1, -1), cnt, axis=0)
    Y_pred[Y < 1-p ] = 1 - Y_pred[Y < 1-p ] # to predictions, affects correlations
    return Y_pred
```

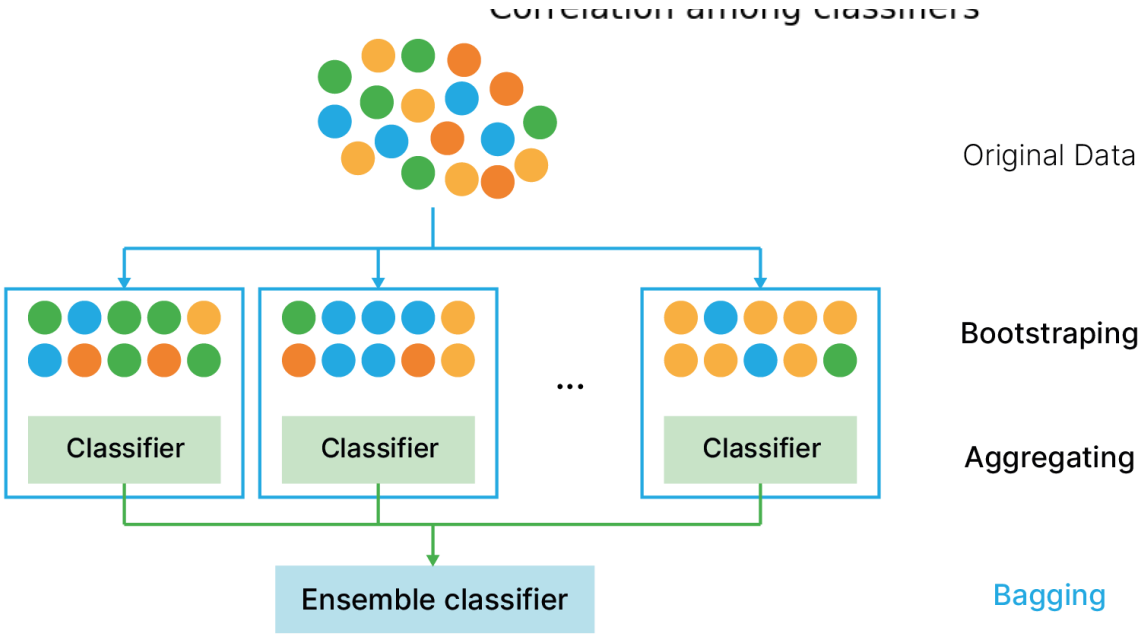
```
np.random.seed(42)
x = np.arange(0, 1, 0.05)
accuracy = np.zeros_like(x)
p = 0.7
cnt = 100
for ind, r in enumerate(x):
    preds = get_correlated_predictions(Y_real, p, cnt, r)
    voting =  np.round(preds.mean(axis=0))
    accuracy[ind] =  (Y_real == voting).mean()


plt.figure(figsize=(16,6))
plt.title(f"Accuracy of {cnt} classifiers ensemble", size=30)
plt.xlabel("Correlation among classifiers", size=25)
plt.ylabel("Accuracy",  size=25)
plt.axhline(y=p, color='red', lw=5, ls='--', label="Single classifier")
sns.lineplot(x=x, y=accuracy, lw=5, label='Ensemble')
plt.legend(fontsize=20)
plt.show()
```



Видим, что по мере увеличения коррелированности моделей, качество все больше и больше приближается к качеству одной базовой модели.

Bagging = Bootstrap aggregation

Нам надо откуда-то брать классификаторы, которые сами по себе предсказывают лучше, чем случайное число, при этом они должны быть не коррелированы. На самом деле это нетривиальная задача: откуда нам брать классификаторы, учитывая, что у нас 1 датасет?



Первый вариант — нам поможет уже рассмотренный bootstrap:

1. Делаем из нашего исходного датасета N выборок такого же размера с повторениями.
2. На каждой полученной выборке (обычно их называют **псевдовыборками**) строим отдельную модель. Чтобы полученным модели были слабо зависимы, будем использовать алгоритм, который чувствителен к небольшим изменениям в выборке.
3. Получаем N слабо зависимых моделей.
4. Когда нам нужно сделать предсказание для нового объекта, делаем предсказание каждой из N моделей, а затем усредняем предсказание.

В sklearn для bagging можно использовать класс BaggingClassifier из sklearn.ensemble. Мы напишем свой код для бэггинга для большей наглядности происходящего. Кроме того, это вам понадобится при выполнении заданий.

Пишем свой bagging

```
import sklearn

def get_bootstrap_sample(X, Y):
    size = X.shape[0]
    poses = np.random.choice(size,
                             size=size,
                             replace=True)
    X_boot = X[poses]
    Y_boot = Y[poses]
    return X_boot, Y_boot

class BaggingBinaryClassifierEnsemble():
    def __init__(self, base_classifier, ensemble_size, random_state=42):
        self. base_classifier =  base_classifier
        self.ensemble_size = ensemble_size
        self.random_state = random_state
        self.ensemble = []

    def fit(self, X, Y):
        np.random.seed(self.random_state)
        for est_id in range(self.ensemble_size):
            X_boot, Y_boot = get_bootstrap_sample(X, Y)
            model = sklearn.clone(self.base_classifier) # create new base model
            model.fit(X_boot, Y_boot)
            self.ensemble.append(model)

    def predict_proba(self, X):
        if not self.ensemble:
            raise Exception("Unfitted model")

        Y_pred = 0
        for est in self.ensemble:
            Y_pred += est.predict(X)
        Y_pred = Y_pred / self.ensemble_size
        return Y_pred

    def predict(self, X):
        Y_proba = self.predict_proba(X)
        Y_pred = np.round(Y_proba)
        return Y_pred
```

Эксперимент1: Классифицируем людей с больным сердцем и нет

Для простоты здесь и далее будем использовать параметры моделей, подобранные ранее, когда мы использовали их вне ансамбля. В данном случае это не имеет значения, но в случае разбираемых далее случайного леса и градиентного бустинга, параметры подбирают **вместе** с построением ансамбля.

```
import os
file_exists = os.path.exists("/content/heart.csv")
if file_exists == False:
    !wget -q https://edunet.kea.su/repo/EduNet-web_dependencies/L03/heart.csv
```

```
heart_dataset = pd.read_csv("/content/heart.csv")
```

```
from sklearn.model_selection import train_test_split

X = heart_dataset.drop("target", axis=1)
Y = heart_dataset['target'] > 0
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, random_state = 42)
```

```
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV

svc_model = GridSearchCV(SVC(), {'kernel':('linear',
                                            'rbf'),
                                 'C':[0.01, 0.1, 1, 10]}
                         ).fit(X_train, Y_train)

logr_model = GridSearchCV(LogisticRegression(solver='liblinear', max_iter=100000),
                          {'penalty':('l1',
                                      'l2'),
                            'C':[0.01, 0.1, 1, 10, 100]}
                          ).fit(X_train, Y_train)

# few objects in the leaf - poor estimates of class probabilities - the model is overtr
dt_model = GridSearchCV(DecisionTreeClassifier(),
                          {'max_depth':[1, 3, 5, 7, 10],
                           'min_samples_leaf': [1, 3, 5, 10]}
                        ).fit(X_train, Y_train)

```

Опробуем модель:

```
bagging_dt = BaggingBinaryClassifierEnsemble(dt_model.best_estimator_,
                                             ensemble_size=100)

bagging_logreg = BaggingBinaryClassifierEnsemble(logr_model.best_estimator_,
                                                 ensemble_size=100)

bagging_svc = BaggingBinaryClassifierEnsemble(svc_model.best_estimator_,
                                              ensemble_size=100)
```

```
bagging_dt.fit(X_train.values, Y_train.values)
bagging_logreg.fit(X_train.values, Y_train.values)
```

```
bagging_svc.fit(X_train.values, Y_train.values)
```

```
Y_pred_blr = bagging_logreg.predict_proba(X_test.values)
Y_pred_bsvc = bagging_svc.predict_proba(X_test.values)
Y_pred_bdt = bagging_dt.predict_proba(X_test.values)
```

```
from sklearn.metrics import average_precision_score # PR-AUC

qual_blr = average_precision_score(y_true=Y_test, y_score=Y_pred_blr)
qual_bsvc = average_precision_score(y_true=Y_test, y_score=Y_pred_bsvc)
qual_bdt = average_precision_score(y_true=Y_test, y_score=Y_pred_bdt)
print(f"Bagged Logistic regression pr-auc: {qual_blr:.03f}")
print(f"Bagged SVC pr-auc: {qual_bsvc :.03f}")
print(f"Bagged DecisionTreeClassifier pr-auc: {qual_bdt:.03f}")
```

```
Bagged Logistic regression pr-auc: 0.872
Bagged SVC pr-auc: 0.900
Bagged DecisionTreeClassifier pr-auc: 0.898
```

```
def bootstrap_metric(X,
                     Y,
                     metric_fn,
                     samples_cnt = 1000,
                     alpha = 0.05,
                     random_state = 42):
    size = len(X)

    np.random.seed(random_state)
    b_metric = np.zeros(samples_cnt)
    for it in range(samples_cnt):
        poses = np.random.choice(X.shape[0], size=X.shape[0], replace=True)

        X_boot = X[poses]
        Y_boot = Y[poses]

        m_val = metric_fn(X_boot, Y_boot)
        b_metric[it] = m_val

    return b_metric
```

```
boot_score_blogreg = bootstrap_metric(Y_test.values,
                                      Y_pred_blr,
                                      metric_fn=lambda X, Y: average_precision_score(y_tru
                                                                                     y_sco
boot_score_bsvc = bootstrap_metric(Y_test.values,
                                      Y_pred_bsvc ,
                                      metric_fn=lambda X, Y: average_precision_score(y_tru
                                                                                     y_sco
boot_score_bdt = bootstrap_metric(Y_test.values,
                                      Y_pred_bdt,
                                      metric_fn=lambda X, Y: average_precision_score(y_tru
                                                                                     y_sco
```

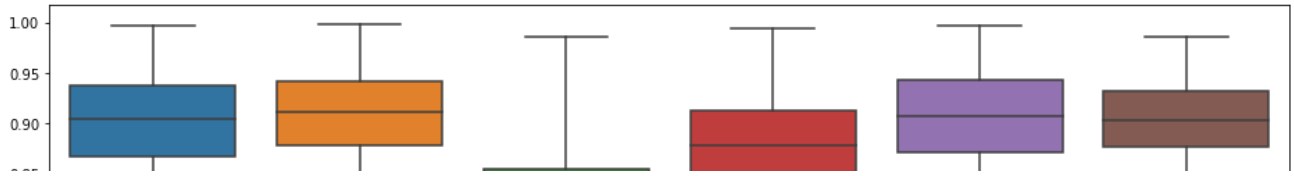
```
alpha=0.10
print("Bagged Logistic regression pr-auc 90%-ci: ", np.quantile(boot_score_blogreg, q=[
print("Bagged SVC pr-auc 90%-ci:", np.quantile(boot_score_bsvc, q=[alpha/2, 1 - alpha/2
print("Bagged DecisionTreeClassifier pr-auc 90%-ci:", np.quantile(boot_score_bdt, q=[al
```

```
Bagged Logistic regression pr-auc 90%-ci:  [0.78262498 0.95069418]
Bagged SVC pr-auc 90%-ci: [0.81964811 0.9761168 ]
Bagged DecisionTreeClassifier pr-auc 90%-ci: [0.83053272 0.95963424]
```

```
Y_pred1 = svc_model.decision_function(X_test) # by default, SVM gives score to each obj
Y_pred2 = logr_model.predict_proba(X_test)[:, 1]
Y_pred3 = dt_model.predict_proba(X_test)[:, 1]

qual1 = average_precision_score(y_true=Y_test, y_score=Y_pred1)
qual2 = average_precision_score(y_true=Y_test, y_score=Y_pred2)
qual3 = average_precision_score(y_true=Y_test, y_score=Y_pred3)
```

```
boot_score_logreg = bootstrap_metric(Y_test.values,
                                     Y_pred1,
                                     metric_fn=lambda X, Y: average_precision_score(y_tru
boot_score_svc = bootstrap_metric(Y_test.values,
                                     Y_pred2,
                                     metric_fn=lambda X, Y: average_precision_score(y_tru
boot_score_dt = bootstrap_metric(Y_test.values,
                                     Y_pred3,
                                     metric_fn=lambda X, Y: average_precision_score(y_tru
```

```
plt.figure(figsize=(16,6))
sns.boxplot(y=np.concatenate([boot_score_logreg,
                              boot_score_svc,
                              boot_score_dt,
                              boot_score_blogreg,
                              boot_score_bsvc,
                              boot_score_bdt
                             ]),
            x=["Log-reg"] * 1000 + ['SVC'] * 1000 + ['DT'] * 1000 + ["Bagged Logreg"]
            + ["Bagged SVC"] * 1000 + ["Bagged DT"] * 1000)
plt.show()
```



Внезапно, мы видим, что главному аутсайдеру — дереву решений, в случае bagging удается побить все остальные решения и улучшить получить качество сравнимое, если не лучше, с качеством других базовых моделей.

0.65

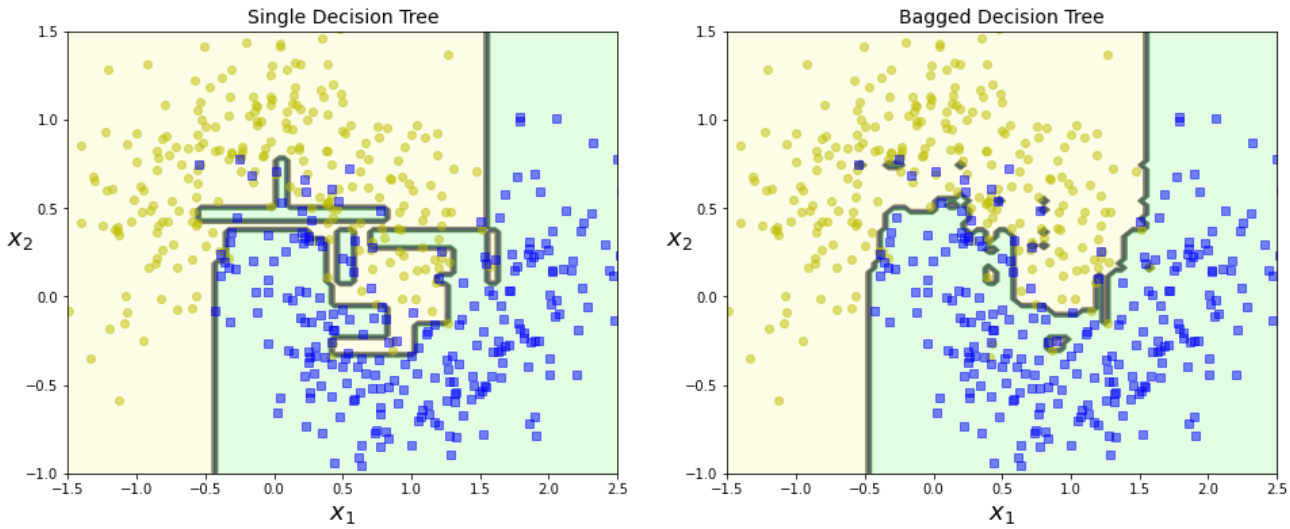
Эксперимент2: Сравним разделяющие плоскости дерева решений и бэггинга на деревьях решений

```
from sklearn import datasets

X, Y = sklearn.datasets.make_moons(n_samples=500, noise=0.30, random_state=42)
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, random_state=42)
```

```
def plot_decision_boundary(clf, X, Y, axes=[-1.5, 2.5, -1, 1.5], alpha=0.5, contour=Tru
    from matplotlib.colors import ListedColormap
    x1s = np.linspace(axes[0], axes[1], 100)
    x2s = np.linspace(axes[2], axes[3], 100)
    x1, x2 = np.meshgrid(x1s, x2s)
    X_new = np.c_[x1.ravel(), x2.ravel()]
    Y_pred = clf.predict(X_new).reshape(x1.shape)
    custom_cmap = ListedColormap(['#fafab0','#9898ff','#a0faa0'])
    plt.contourf(x1, x2, Y_pred, alpha=0.3, cmap=custom_cmap)
    if contour:
        custom_cmap2 = ListedColormap(['#7d7d58','#4c4c7f','#507d50'])
        plt.contour(x1, x2, Y_pred, cmap=custom_cmap2, alpha=0.8)
    plt.plot(X[:, 0][Y==0], X[:, 1][Y==0], "yo", alpha=alpha)
    plt.plot(X[:, 0][Y==1], X[:, 1][Y==1], "bs", alpha=alpha)
    plt.axis(axes)
    plt.xlabel(r"$x_1$", fontsize=18)
    plt.ylabel(r"$x_2$", fontsize=18, rotation=0)
```

```
plt.figure(figsize=(16,6))
plt.subplot(121)
clf = DecisionTreeClassifier(max_depth = 10,
                             random_state = 42)
clf.fit(X_train, Y_train)
plot_decision_boundary(clf, X, Y)
plt.title("Single Decision Tree", fontsize=14)


plt.subplot(122)
bagging_dt = BaggingBinaryClassifierEnsemble(DecisionTreeClassifier(max_depth=10),
                                             ensemble_size=100)
bagging_dt.fit(X_train, Y_train)
plot_decision_boundary(bagging_dt, X, Y)
```

```
plt.title("Bagged Decision Tree", fontsize=14)
plt.show()
```

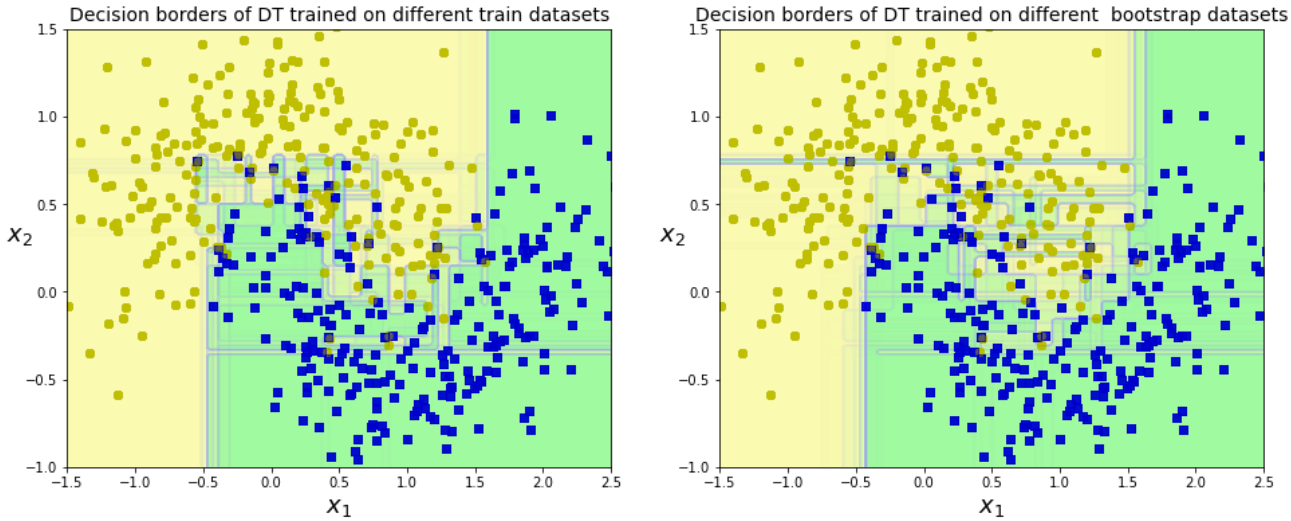


Видим, что разделяющая плоскость для дерева более гладкая и лучше отражает реальное разделение классов.

До этого мы говорили (обсуждая bias-variance trade-off), что хорошо бы взять решающие границы большого числа деревьев, построенных на разных тренировочных датасетах, полученных из генеральной совокупности, и усреднить эти границы, получив более хорошую решающую границу. К сожалению, мы не можем брать бесконечное число тренировочных датасетов. Но у нас есть бутстрэп. И он тоже достаточно хорошо аппроксимирует границу, которую мы бы получили, будь у нас много разных тренировочных датасетов:

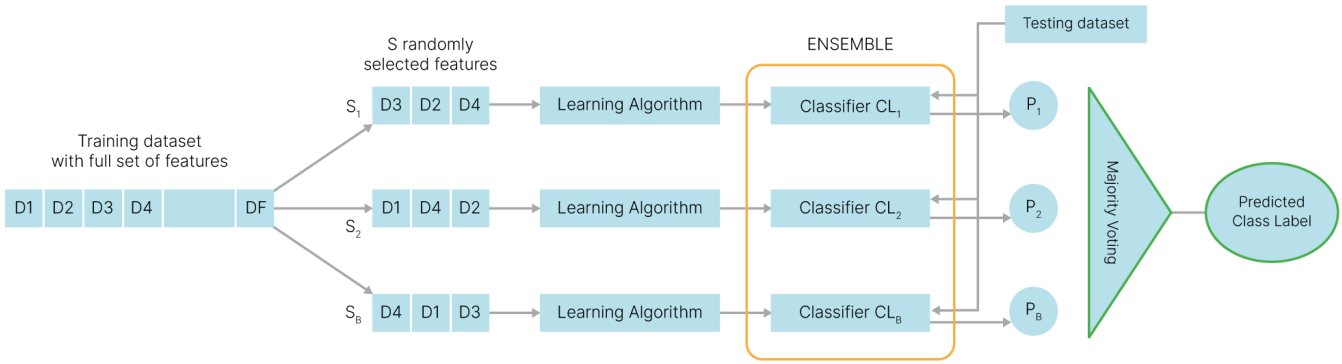
```
plt.figure(figsize=(16,6))
plt.subplot(121)
plt.title("Decision borders of DT trained on different train datasets", fontsize=14)
for i in range(100):
    X_train, X_test, Y_train, Y_test = train_test_split(X, Y, random_state=i)
    clf = DecisionTreeClassifier(max_depth = 20, random_state = 42)
    clf.fit(X_train, Y_train)
    plot_decision_boundary(clf, X, Y, alpha=0.02, contour=False)

plt.subplot(122)
plt.title("Decision borders of DT trained on different  bootstrap datasets", fontsize=1
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, random_state=42)
```

```
bagging_dt = BaggingBinaryClassifierEnsemble(DecisionTreeClassifier(max_depth=10),
                                             ensemble_size=100, random_state=42)

bagging_dt.fit(X_train, Y_train)

for base_dt in bagging_dt.ensemble:
    plot_decision_boundary(base_dt, X, Y, alpha=0.02, contour=False)
plt.show()
```



Метод случайных подпространств (RSM, random subspace method)

Второй вариант получения пссевдовыборок — мы можем сэмплировать не объекты, а признаки. При этом бесполезно иметь в выборке два одинаковых признака, потом мы делаем выборки меньшего размера, чем исходное число признаков и без повторений.



Обычно для каждой модели выбирают:

для задач классификации:

$$\sqrt{feature_cnt}$$

для задач регрессии :

$$\frac{feature_cnt}{3}$$

Хотя строгих правил нет, этот параметр можно подбирать на кросс-валидации.

Пишем свой RSM

Опять же, напишем метод случайных подпространств сами:

```
import sklearn

def get_rsm_sample(X, Y, f_num=None):
    size = X.shape[1]
    f_num = f_num or int(np.sqrt(size)) + 1
    f_num = min(size, f_num)

    f_poses = np.random.choice(size,
                               size=f_num,
                               replace=False)
    X_rsm = X[:, f_poses]
    Y_rsm = Y.copy()
    return X_rsm, Y_rsm, f_poses

class RSMBinaryClassifierEnsemble():
    def __init__(self, base_classifier, ensemble_size, random_state=42, max_features=No
        self. base_classifier =  base_classifier
        self.ensemble_size = ensemble_size
        self.random_state = random_state
        self.max_features = max_features

        self.ensemble = []
        self.feature_poses = []
        # we had to keep track of features selected. In sklearn Random Forest, discusse
        # another, more stable implementation is used.
        # they use `f_num` random features but in case no good split found, they try ot

    def fit(self, X, Y):
        np.random.seed(self.random_state)
        for est_id in range(self.ensemble_size):
            X_boot, Y_boot, f_poses = get_rsm_sample(X, Y, f_num=self.max_features)
            self.feature_poses.append(f_poses)
            model = sklearn.clone(self.base_classifier) # create new base model
            model.fit(X_boot, Y_boot)
            self.ensemble.append(model)

    def predict_proba(self, X):
        if not self.ensemble:
```

```
            raise Exception("Unfitted model")

        Y_pred = 0
        for ind, est in enumerate(self.ensemble):
            Y_pred += est.predict(X[:, self.feature_poses[ind]])
        Y_pred = Y_pred / self.ensemble_size
        return Y_pred


    def predict(self, X):
        Y_proba = self.predict_proba(X)
        Y_pred = np.round(Y_proba)
        return Y_pred

```

Эксперимент1: Классифицируем людей с больным сердцем и нет

```
X = heart_dataset.drop("target", axis=1)
Y = heart_dataset['target'] > 0
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, random_state=42)

rsm_dt = RSMBinaryClassifierEnsemble(dt_model.best_estimator_,
                                     ensemble_size=100)

rsm_logreg = RSMBinaryClassifierEnsemble(logr_model.best_estimator_,
                                         ensemble_size=100)

rsm_svc = RSMBinaryClassifierEnsemble(svc_model.best_estimator_,
                                      ensemble_size=100)
```

```
rsm_dt.fit(X_train.values, Y_train.values)
rsm_logreg.fit(X_train.values, Y_train.values)
rsm_svc.fit(X_train.values, Y_train.values)
```

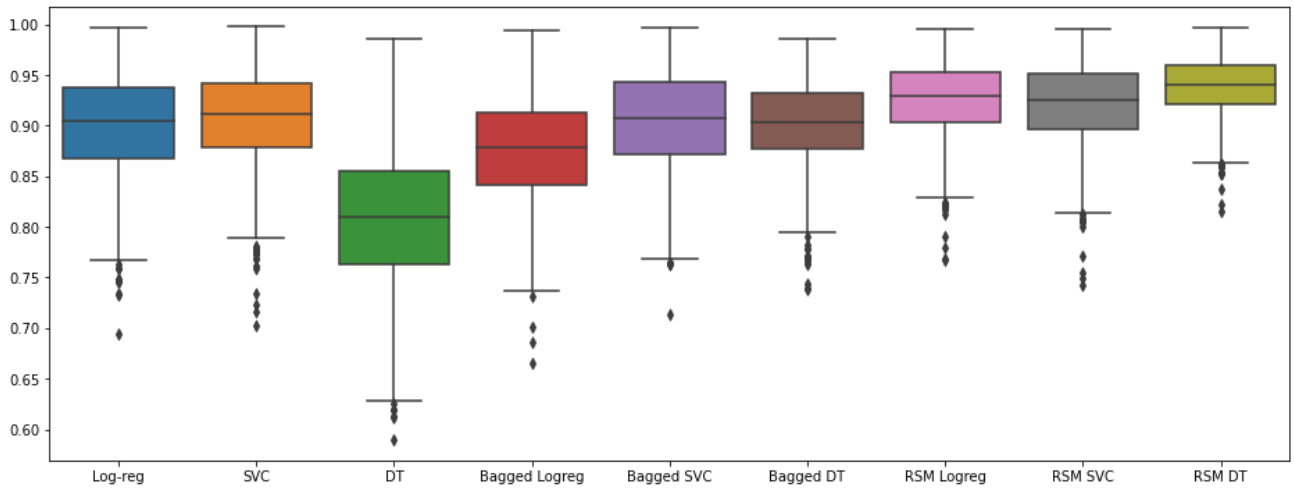
```
Y_pred_rlr = rsm_logreg.predict_proba(X_test.values)
Y_pred_rsvc = rsm_svc.predict_proba(X_test.values)
Y_pred_rdt = rsm_dt.predict_proba(X_test.values)
```

```
boot_score_rlogreg = bootstrap_metric(Y_test.values,
                                      Y_pred_rlr,
                                      metric_fn=lambda X, Y:
                                      average_precision_score(y_true=X, y_score=Y))

boot_score_rsvc = bootstrap_metric(Y_test.values,
                                   Y_pred_rsvc ,
                                   metric_fn=lambda X, Y:
                                   average_precision_score(y_true=X, y_score=Y))

boot_score_rdt = bootstrap_metric(Y_test.values,
                                  Y_pred_rdt,
```

```
                                metric_fn=lambda X, Y:
                                average_precision_score(y_true=X, y_score=Y))

alpha=0.10
print("RSM Logistic regression pr-auc 90%-ci: ", np.quantile(boot_score_blogreg, q=[alp
print("RSM SVC pr-auc 90%-ci:", np.quantile(boot_score_bsvc, q=[alpha/2, 1 - alpha/2]))
print("RSM DecisionTreeClassifier pr-auc 90%-ci:", np.quantile(boot_score_bdt, q=[alpha
```

```
RSM Logistic regression pr-auc 90%-ci:  [0.78262498 0.95069418]
RSM SVC pr-auc 90%-ci: [0.81964811 0.9761168 ]
RSM DecisionTreeClassifier pr-auc 90%-ci: [0.83053272 0.95963424]
```

```
plt.figure(figsize=(16,6))
sns.boxplot(y=np.concatenate([boot_score_logreg,
                              boot_score_svc,
                              boot_score_dt,
                              boot_score_blogreg,
                              boot_score_bsvc,
                              boot_score_bdt,
                              boot_score_rlogreg,
                              boot_score_rsvc,
                              boot_score_rdt
                             ]),
            x=[x for x in  ('Log-reg',
             'SVC',
             'DT',
             "Bagged Logreg",
             "Bagged SVC",
             "Bagged DT",
             "RSM Logreg",
             "RSM SVC",
             "RSM DT") for y in range(1000)])
plt.show()
```

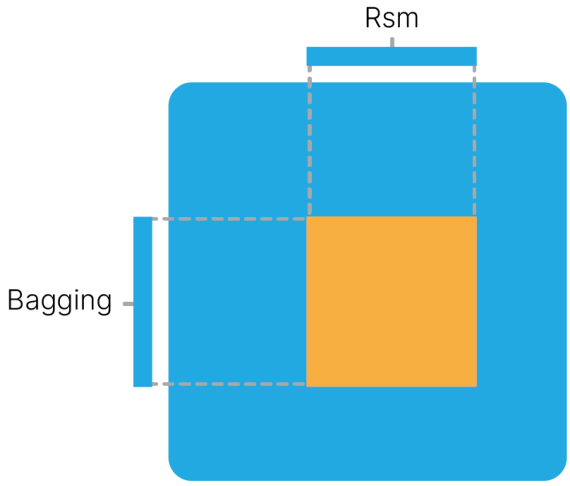


Опять же, видим, что RSM помогает только дереву решений. Остальные методы не улучшают своего качества.

Комбинация RSM и Bagging

Можно объединить оба способа: применяем bootstrap к объектам (получается выборка одного размера, но с повторяющимися объектами, а каких-то объектов не будет), и, кроме этого, выкидываем часть признаков. Зачем это нужно? В этом случае мы получим еще более сильно отличающиеся друг от друга случайные выборки.

sklearn.ensemble.BaggingClassifier и sklearn.ensemble.BaggingRegressor вопреки названию может поддерживать оба способа.



Почему для одних классов моделей работает, а для других - нет

Теперь будем использовать BaggingClassier из стандартной библиотеки:

```
from sklearn.ensemble import BaggingClassifier
models = {}


svc_model = GridSearchCV(SVC(), {'kernel':('linear',
                                            'rbf'),
                                 'C':[0.01, 0.1, 1, 10]}
                         ).fit(X_train, Y_train)
models['SVC'] = svc_model


logr_model = GridSearchCV(LogisticRegression(solver='liblinear', max_iter=100000),
                          {'penalty':('l1',
                                      'l2'),
                            'C':[0.01, 0.1, 1, 10, 100]}
```

```
                          ).fit(X_train, Y_train)
models['LogReg'] = logr_model

# few objects in the leaf - poor estimates of class probabilities - the model is overtr
dt_model = GridSearchCV(DecisionTreeClassifier(),
                          {'max_depth':[1, 3, 5, 7, 10],
                           'min_samples_leaf': [1, 3, 5, 10]}
                        ).fit(X_train, Y_train)
models['DT'] = dt_model


bagging_dt = BaggingClassifier(dt_model.best_estimator_,
                               n_estimators=100,
                               random_state=42)
models["Bagging DT"] = bagging_dt


bagging_logr = BaggingClassifier(logr_model.best_estimator_,
                                 n_estimators=100,
                                 random_state=42)
models["Bagging LogReg"] = bagging_logr


bagging_svc = BaggingClassifier(svc_model.best_estimator_,
                                n_estimators=100,
                                random_state=42)
models["Bagging SVC"] = bagging_svc


sqrt_features = int(np.sqrt(X.shape[1])) + 1


rsm_dt = BaggingClassifier(dt_model.best_estimator_,
                           n_estimators=100,
                           bootstrap=False,
                           max_features=sqrt_features,
                           random_state=42)
models["RSM DT"] =  rsm_dt


rsm_logreg = BaggingClassifier(logr_model.best_estimator_,
                               n_estimators=100,
                               bootstrap=False,
                               max_features=sqrt_features,
                               random_state=42)
models["RSM LogReg"] = rsm_logreg


rsm_svc = BaggingClassifier(svc_model.best_estimator_,
                            n_estimators=100,
                            bootstrap=False,
                            max_features=sqrt_features,
                            random_state=42)
models["RSM SVC"] = rsm_svc

```

```
# Both Bagging and RSM
bag_rsm_dt = BaggingClassifier(dt_model.best_estimator_,
                               n_estimators=100,
                               bootstrap=True,
                               max_features=sqrt_features,
                               random_state=42)
models["BagRSM DT"] = bag_rsm_dt


bag_rsm_logreg = BaggingClassifier(logr_model.best_estimator_,
                                   n_estimators=100,
                                   bootstrap=True,
                                   max_features=sqrt_features,
                                   random_state=42)
models["BagRSM LogReg"] = bag_rsm_logreg


bag_rsm_svc = BaggingClassifier(svc_model.best_estimator_,
                                n_estimators=100,
                                bootstrap=True,
                                max_features=sqrt_features,
                                random_state=42)
models["BagRSM SVC"] = bag_rsm_svc
```

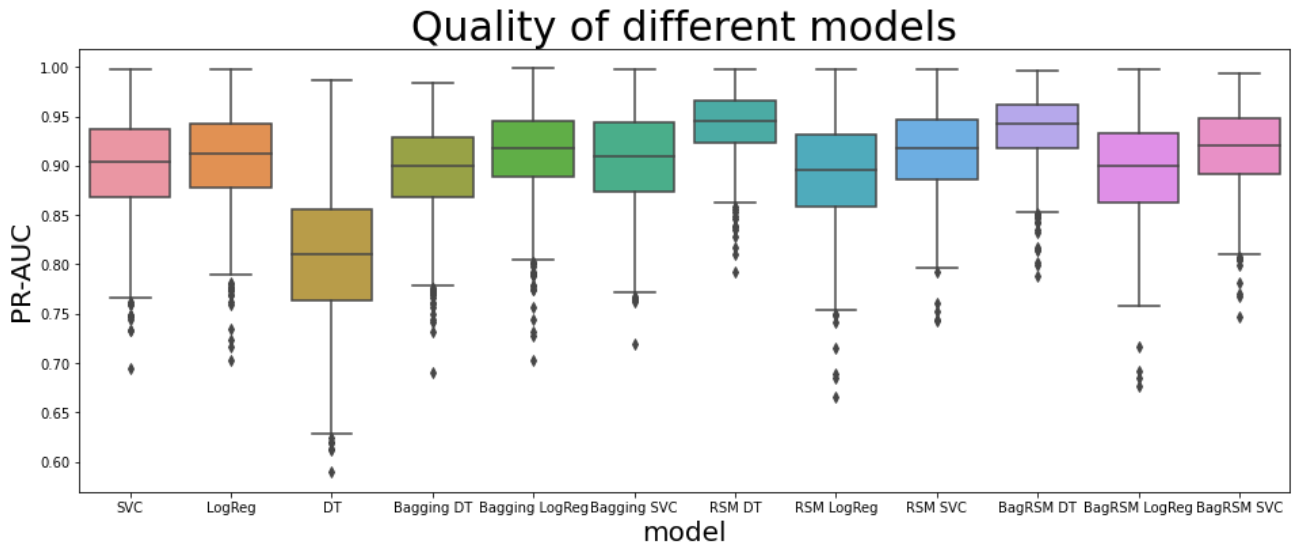
```
# Trainig models
for name, model in models.items():
    model.fit(X_train, Y_train)
```

```
predictions = {}

for name, model in models.items():
    if name != "SVC":
        Y_pred = model.predict_proba(X_test)[:, 1]
    else:
        Y_pred = model.decision_function(X_test)
    predictions[name] = Y_pred
```

```
boot_scores = {}

for name, Y_pred in predictions.items():
    boot_score = bootstrap_metric(Y_test.values,
                                  Y_pred,
                                  metric_fn=lambda X, Y:
                                 average_precision_score(y_true=X, y_score=Y))
    boot_scores[name] = boot_score
```

```
results = pd.DataFrame(boot_scores)
# cast to long format
results = results.melt(value_vars=results.columns,
                       value_name="prauc",
                       var_name="model")
```

```

plt.figure(figsize=(16,6))
sns.boxplot(data=results, y="prauc", x="model")
plt.xlabel("model", size=20)
plt.ylabel("PR-AUC", size=20)
plt.title("Quality of different models", size=30)
plt.show()
```



Видим, что, опять же, существенное улучшение качества наблюдается только, когда в качестве базовой модели мы используем дерево.

Мы говорили о том, что важное требование того, чтобы простое голосование работало - чтобы ошибки моделей не коррелировали между собой.

Попробуем оценить попарную корреляцию в ошибках базовых моделей в наших ансамблях.

```
import itertools
def base_model_pair_correlation(ensemble, X):
    corrs = []
    for (i, est1), (j, est2) in itertools.combinations(enumerate(ensemble.estimators_),
        Xi_test = X.values[:, ensemble.estimators_features_[i]]
        Xj_test = X.values[:, ensemble.estimators_features_[j]]


        if not isinstance(est1, sklearn.svm.SVC):
            Y_pred_t1 = est1.predict_proba(Xi_test)[:, 1]

            Y_pred_t2 = est2.predict_proba(Xj_test)[:, 1]
        else:
```

```
            Y_pred_t1 = est1.decision_function(Xi_test)
            Xj_test = X_test.values[:, ensemble.estimators_features_[j]]
            Y_pred_t2 = est2.decision_function(Xj_test)
        corrs.append(scipy.stats.pearsonr(Y_pred_t1, Y_pred_t2)[0])
    return np.array(corrs)
```

```
pair_correlations = {}
for name, model in models.items():
    if not "Bagging" in name and not "RSM" in name:
        continue
    pair_correlations[name] = base_model_pair_correlation(model, X_test)
```

```
cor_res = pd.DataFrame(pair_correlations)
cor_res = cor_res.melt(value_vars=cor_res.columns,
                       value_name="paircor",
                       var_name="model")
```

Кроме этого, давайте посчитаем качество базовых моделей, входящих в каждый из ансамблей:

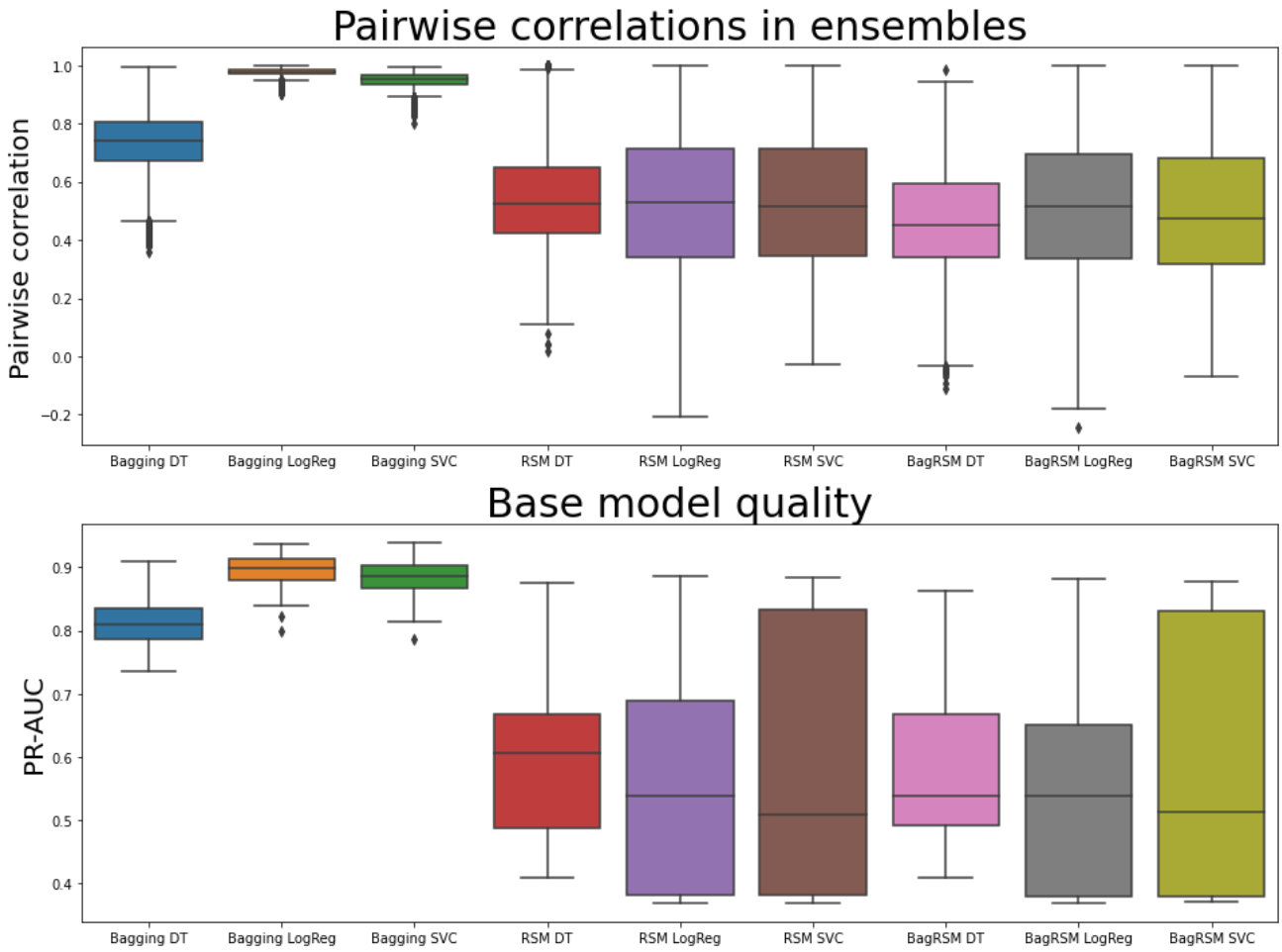
```
def base_model_prauc(ensemble, X, Y):
    qual = np.zeros(ensemble.n_estimators)
    for ind, est in enumerate(ensemble.estimators_):
        X_test = X.values[:, ensemble.estimators_features_[i]]
        if not isinstance(est, sklearn.svm.SVC):
            Y_pred = est.predict_proba(X_test)[:, 1]
        else:
            Y_pred = est.decision_function(X_test)
        qual[ind] = average_precision_score(y_score=Y_pred, y_true=Y)
    return qual
```

```
base_prauc = {}
for name, model in models.items():
    if not "Bagging" in name and not "RSM" in name:
        continue
    base_prauc[name] = base_model_prauc(model, X_test, Y_test)
```

```
base_prauc_res = pd.DataFrame(base_prauc)
base_prauc_res = base_prauc_res.melt(value_vars=base_prauc_res.columns,
                       value_name="prauc",
                       var_name="model")
```

```
plt.figure(figsize=(16,12))
plt.subplot(211)
sns.boxplot(data=cor_res, y="paircor", x="model")
plt.title("Pairwise correlations in ensembles", size=30)
plt.xlabel("", size=20)
plt.ylabel("Pairwise correlation", size=20)

plt.subplot(212)
```

```
sns.boxplot(data=base_prauc_res, y="prauc", x="model")
plt.title("Base model quality", size=30)
plt.xlabel("", size=20)
plt.ylabel("PR-AUC", size=20)
plt.subplots_adjust()
plt.show()
```



Какие можно сделать выводы из предложенных графиков?

1. Bagging, RSM и их комбинация помогают уменьшать корреляцию между базовыми моделями. Причем, комбинация Bagging и RSM работают лучшее, чем каждый из них поодиночке;
2. В случае просто Bagging дерево решений получает существенный прирост качества за счет того, что модели на его основе коррелированы значительно слабее, нежели

модели на основе логистической регрессии и машины опорных векторов;

3. В остальных случаях корреляция моделей на основе деревьев решений все равно немного ниже, чем у других моделей, при этом качество базовых моделей сравнимо. Из-за этого, опять же, ансамбль на основе дерева решений показывает лучший результат.

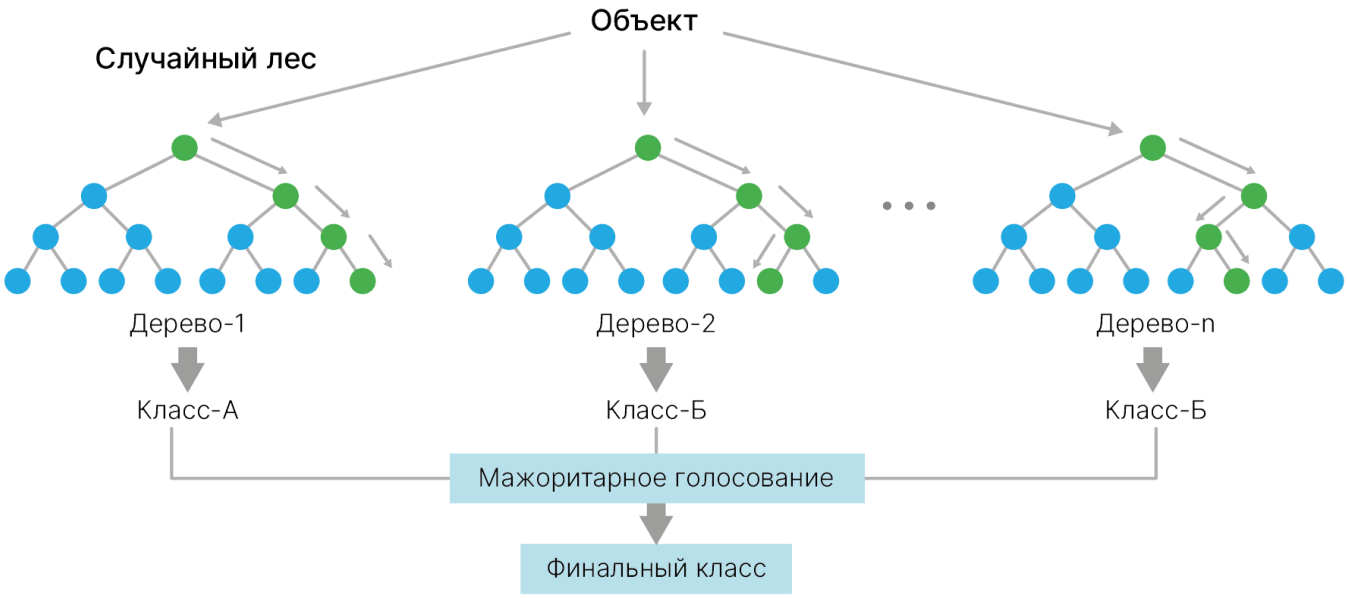
Случайный лес

Из предыдущей части мы увидели, что использование bagging или RSM для SVM или подобных моделей не несет большого смысла.

И если использовать bagging или RSM для SVM, то значимого улучшения качества не будет.

Но для деревьев решений это не так. Будем брать деревья большой глубины. Незначительные изменения в данных приводят к значительным изменениям в топологии таких деревьев. Таким образом, мы приходим к **случайному лесу** = Bagging + RSM над деревом решений.

При этом RSM в классическом случайном лесе делается не на уровне дерева, а на уровне узла. В каждом узле дерева, когда мы выбираем лучшее разбиение его на два дочерних, мы просматриваем не все признаки, а только определенное их количество.



Обычно случайный лес работает лучше, чем отдельно случайно взятое дерево. Но главное - случайный лес намного более устойчив к шуму. Оказывается, что это свойство до сих пор позволяет случайному лесу успешно использоваться в областях с шумными данными. Например, в биологии.

Зависимость качества случайного леса от числа деревьев

Для случайного леса верно следующее: когда мы берем множество базовых классификаторов (в данном случае деревьев) и усредняем их, то результат этих усреднений стремится к идеальному дереву решений, причем построенному на идеальных, а не на исходных признаках.

Посмотрим, действительно ли это так. Для того, чтобы не пересчитывать случайный лес для каждого числа деревьев решений, обучим один случайный лес с 1000 деревьев решений, а леса с меньшим числом решений будем

И будем работать для разнообразия — с задачей регрессии. Задача — предсказать цену на дом.

```
calif_housing = sklearn.datasets.fetch_california_housing()
X = calif_housing.data
Y = calif_housing.target
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, random_state=42)
```

И напишем функцию, которая будет автоматически обучать переданные ей модели и считать для них качество на тесте. Чтобы избавиться от необходимости копировать код между ячейками:

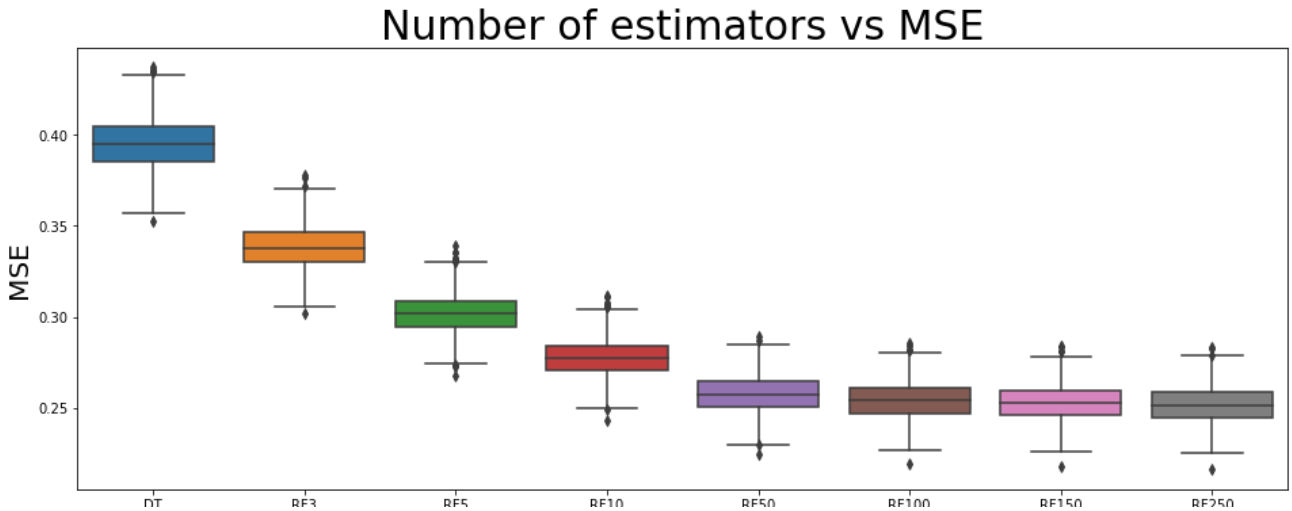
```
from sklearn.metrics import mean_squared_error

def train_and_test_regressor(models, X_train, Y_train, X_test, Y_test):
    for name, model in models.items():
        print(f"Fitting {name}")
        model.fit(X_train, Y_train)
    predictions = {}
    for name, model in models.items():
        Y_pred = model.predict(X_test)
        predictions[name] = Y_pred

    boot_scores = {}

    for name, Y_pred in predictions.items():
        print(f"Calculating bootstrap score for {name}")
        boot_score = bootstrap_metric(Y_test,
                                      Y_pred,
                                      metric_fn=lambda X, Y: mean_squared_error(y_true
                                                                                    y
        boot_scores[name] = boot_score


    results = pd.DataFrame(boot_scores)
    # cast to long format
```

```
    results = results.melt(value_vars=results.columns,
                           value_name="mse",
                           var_name="model")
    return results
```

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.tree import DecisionTreeRegressor
models_rf = {}

models_rf['DT'] =  GridSearchCV(DecisionTreeRegressor(),
                                {'max_depth':[1, 3, 5, 7, 10],
                                 'min_samples_leaf': [1, 3, 5, 10]}
                            )# add single decision tree for comparison

# this can be done faster, see warm_start parameter for this
# (https://stackoverflow.com/questions/42757892/how-to-use-warm-start)
for n_estimators in [3, 5, 10, 50, 100, 150, 250]:
    models_rf[f"RF{n_estimators}"] = RandomForestRegressor(n_estimators=n_estimators,
                                                           random_state=42,
                                                           n_jobs=-1) # run in parallel

# Trainig models
results_rf = train_and_test_regressor(models_rf, X_train, Y_train, X_test, Y_test)
```

```
Fitting DT
Fitting RF3
Fitting RF5
Fitting RF10
Fitting RF50
Fitting RF100
Fitting RF150
Fitting RF250
Calculating bootstrap score for DT
Calculating bootstrap score for RF3
Calculating bootstrap score for RF5
Calculating bootstrap score for RF10
Calculating bootstrap score for RF50
Calculating bootstrap score for RF100
Calculating bootstrap score for RF150
Calculating bootstrap score for RF250
```

```
plt.figure(figsize=(16,6))
sns.boxplot(data=results_rf, y="mse", x="model")
plt.xlabel("", size=20)
plt.ylabel("MSE", size=20)
plt.title("Number of estimators vs MSE", size=30)
plt.show()
```



Видим, что действительно, с какого-то момента число качество модели почти не изменяется, увеличение числа базовых классифакторов как минимум не ухудшает ее качество.

Зависимость качества случайного леса от глубины дерева

Мы уже обсуждали, что чем больше глубина дерева, тем большая нескорелированность базовых моделей будет получаться. По идее, в случайном лесе важно использовать именно глубокие деревья, причем, в большинстве случаев их глубину не надо ограничивать (или ограничивать большими значениями порядка 10-12). Проверим это.

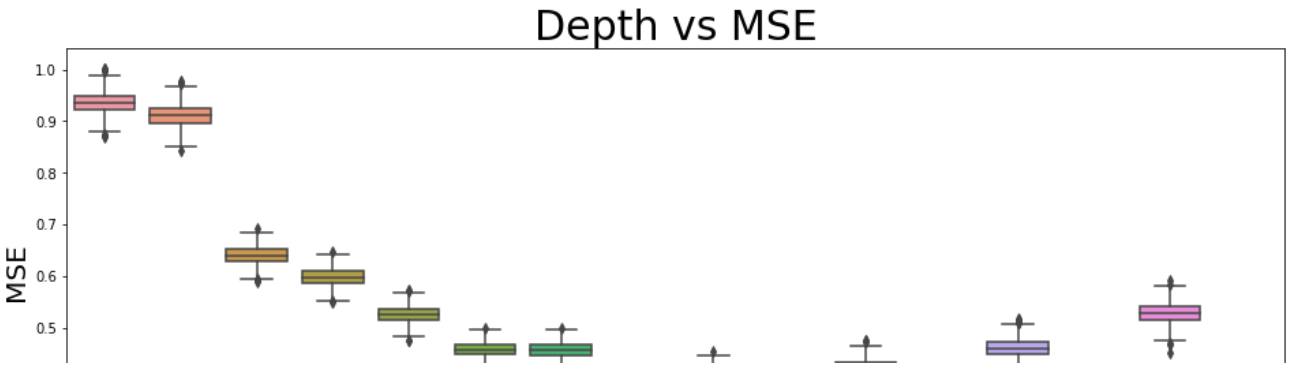
Кроме этого, будем добавлять кроме случайного леса с заданной глубиной еще одиночное дерево такой же глубины

```
from sklearn.tree import DecisionTreeClassifier

models_depth = {}

for depth in [1, 3, 5, 7, 9, 11, 13, None]:

    models_depth[f"DT_depth{depth}"] = DecisionTreeRegressor(max_depth=depth,
                                                             random_state=42)

    models_depth[f"RF_depth{depth}"] = RandomForestRegressor(n_estimators=100,
                                                             max_depth=depth,
                                                             random_state = 42,
                                                             n_jobs=-1) # run in parall

# Trainig models
results_depth = train_and_test_regressor(models_depth, X_train, Y_train, X_test, Y_test
```

```
Fitting DT_depth1
Fitting RF_depth1
Fitting DT_depth3
Fitting RF_depth3
Fitting DT_depth5
```

```
Fitting RF_depth5
Fitting DT_depth7
Fitting RF_depth7
Fitting DT_depth9
Fitting RF_depth9
Fitting DT_depth11
Fitting RF_depth11
Fitting DT_depth13
Fitting RF_depth13
Fitting DT_depthNone
Fitting RF_depthNone
Calculating bootstrap score for DT_depth1
Calculating bootstrap score for RF_depth1
Calculating bootstrap score for DT_depth3
Calculating bootstrap score for RF_depth3
Calculating bootstrap score for DT_depth5
Calculating bootstrap score for RF_depth5
Calculating bootstrap score for DT_depth7
Calculating bootstrap score for RF_depth7
Calculating bootstrap score for DT_depth9
Calculating bootstrap score for RF_depth9
Calculating bootstrap score for DT_depth11
Calculating bootstrap score for RF_depth11
Calculating bootstrap score for DT_depth13
Calculating bootstrap score for RF_depth13
Calculating bootstrap score for DT_depthNone
Calculating bootstrap score for RF_depthNone
```

```
plt.figure(figsize=(16,6))
ax = sns.boxplot(data=results_depth, y="mse", x="model")
plt.setp(ax.get_xticklabels(), rotation=90)
plt.xlabel("", size=20)
plt.ylabel("MSE", size=20)
plt.title("Depth vs MSE", size=30)
plt.xticks(size=20)
plt.show()
```



Действительно, в данном случае модель без ограничения на глубину дерева показывает наилучшие результаты. Модель же с малой глубиной дерева работает сильно хуже.

Качество случайного леса с малой глубиной дерева не отличается от качества одиночного дерева той же глубины. Опять же, деревья малой глубины слабо отличаются друг от друга, потому усреднение их предсказаний не дает почти никакого выигрыша.

Стоит все же отметить, что в случае малых выборок, незначительные ограничения на глубину дерева могут дать выигрыш.

Минимальное число объектов в листе

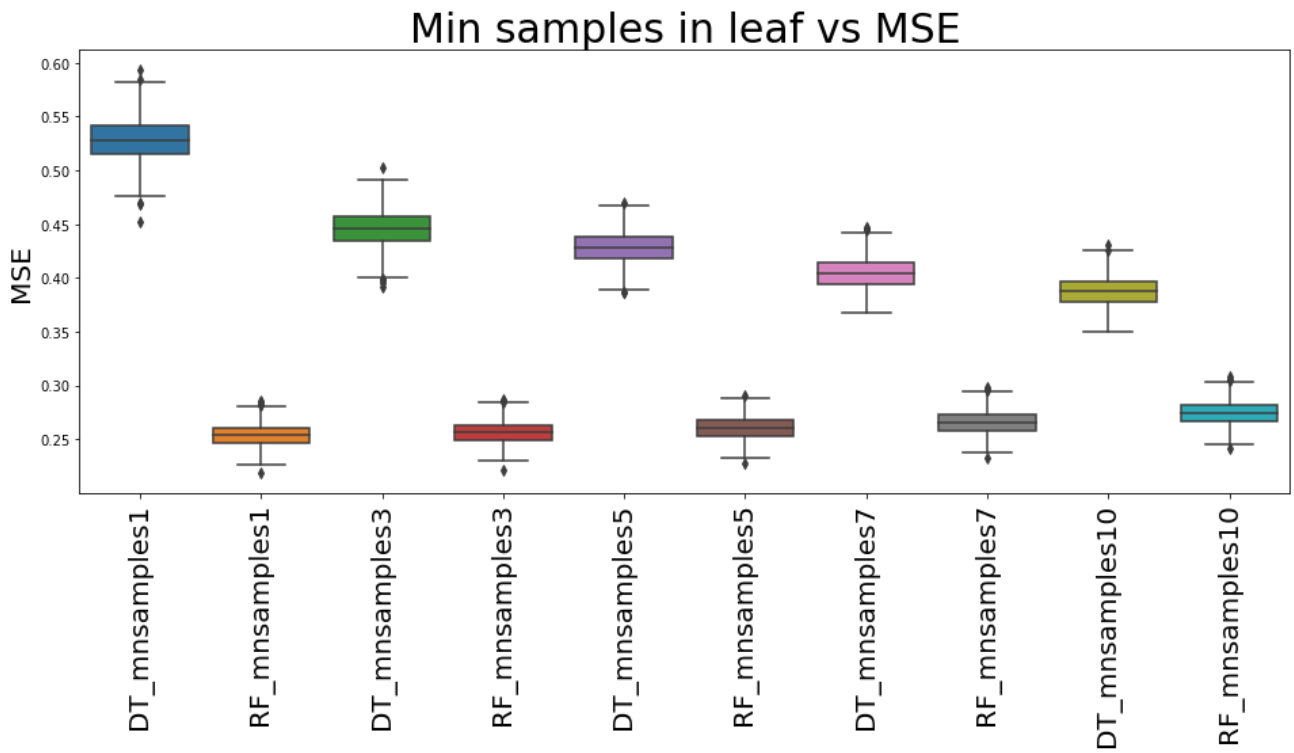
Так же иногда качество дополнительно улучшается, если поставить небольшое ограничение на минимальное число объектов в листе, чтобы запретить явно переобученные деревья.

```
models_min_samples= {}

for mn_sm in [1, 3, 5, 7, 10]:

    models_min_samples[f"DT_mnsamples{mn_sm}"] = DecisionTreeRegressor(max_depth=None,
                                                                        min_samples_leaf=mn_sm,
                                                                        random_state=42)

    models_min_samples[f"RF_mnsamples{mn_sm}"] = RandomForestRegressor(n_estimators=100
                                                                        max_depth=None,
                                                                        min_samples_leaf=mn_sm,
                                                                        random_state=42,
                                                                        n_jobs=-1) # run in paralle

# Trainig models
results_mn_samples = train_and_test_regressor( models_min_samples, X_train, Y_train, X_
```

```
Fitting DT_mnsamples1
Fitting RF_mnsamples1
Fitting DT_mnsamples3
Fitting RF_mnsamples3
Fitting DT_mnsamples5
Fitting RF_mnsamples5
Fitting DT_mnsamples7
Fitting RF_mnsamples7
```

```
Fitting DT_mnsamples10
Fitting RF_mnsamples10
Calculating bootstrap score for DT_mnsamples1
Calculating bootstrap score for RF_mnsamples1
Calculating bootstrap score for DT_mnsamples3
Calculating bootstrap score for RF_mnsamples3
Calculating bootstrap score for DT_mnsamples5
Calculating bootstrap score for RF_mnsamples5
Calculating bootstrap score for DT_mnsamples7
Calculating bootstrap score for RF_mnsamples7
Calculating bootstrap score for DT_mnsamples10
Calculating bootstrap score for RF_mnsamples10
```

```
plt.figure(figsize=(16,6))
ax = sns.boxplot(data=results_mn_samples, y="mse", x="model")
plt.setp(ax.get_xticklabels(), rotation=90)
plt.xlabel("", size=20)
plt.ylabel("MSE", size=20)
plt.title("Min samples in leaf vs MSE", size=30)
plt.xticks(size=20)
plt.show()
```



В данном случае, однако, мы это не наблюдаем. Более того, опять же, с увеличением числа ограничений на дерево, падает непохожесть деревьев друг на друга и качество случайного леса так же начинает падать.

Переобучается ли случайный лес?

Существует мнение, что случайный лес не переобучается. Это не так. Можно подобрать такие два набора параметров, что первый дает лучше качество на тренировочной выборке, а второй — на тестовой. При этом увеличение числа деревьев в ансамбле с этим справиться поможет слабо.

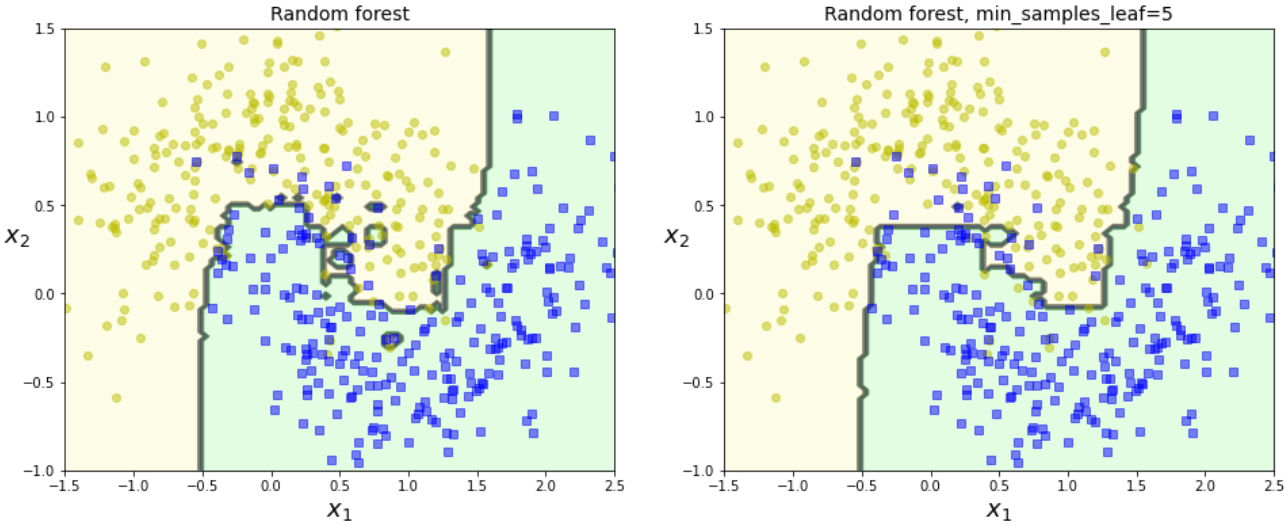
Возьмем, к примеру, наш игрушечный пример на плоскости.

```
from sklearn.ensemble import RandomForestClassifier

X, Y = sklearn.datasets.make_moons(n_samples=500, noise=0.30, random_state=42)
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, random_state=42)
plt.figure(figsize=(16,6))
plt.subplot(121)

clf = DecisionTreeClassifier(max_depth = 10, random_state = 42)
clf.fit(X_train, Y_train)
plot_decision_boundary(clf, X, Y)
plt.title("Single Decision Tree", fontsize=14)


plt.subplot(122)
rf = RandomForestClassifier(n_estimators=1000)
rf.fit(X_train, Y_train)
plot_decision_boundary(rf, X, Y)
plt.title("Random forest", fontsize=14)
plt.show()
```

Очевидно, видны области, в которых случайный лес переобучился.

Добавление ограничения на число объектов в листьях улучшает ситуацию.

```
plt.figure(figsize=(16,6))

plt.subplot(121)
rf1 = RandomForestClassifier(n_estimators=1000)
rf1.fit(X_train, Y_train)
plot_decision_boundary(rf1, X, Y)
plt.title("Random forest", fontsize=14)

plt.subplot(122)
rf2 = RandomForestClassifier(n_estimators=1000, min_samples_leaf=5)
rf2.fit(X_train, Y_train)
plot_decision_boundary(rf2, X, Y)
plt.title("Random forest, min_samples_leaf=5", fontsize=14)
plt.show()
```



Сравним качество на трейне — у первого леса оно больше, в то время как на тесте лучше качество у второго дерева.

```
Y_score = rf1.predict(X_train)
q = average_precision_score(y_true=Y_train, y_score=Y_score)
print(f"RF1 Train: {q:.02}")
Y_score = rf1.predict(X_test)
```

```
q = average_precision_score(y_true=Y_test, y_score=Y_score)
print(f"RF1 Test: {q:.02}")

Y_score = rf2.predict(X_train)
q = average_precision_score(y_true=Y_train, y_score=Y_score)
print(f"RF2 Train: {q:.02}")
Y_score = rf2.predict(X_test)
q = average_precision_score(y_true=Y_test, y_score=Y_score)
print(f"RF2 Test: {q:.02}")
```

```
RF1 Train: 1.0
RF1 Test: 0.87
RF2 Train: 0.91
RF2 Test: 0.91
```

Случайный лес и bias-variance tradeoff

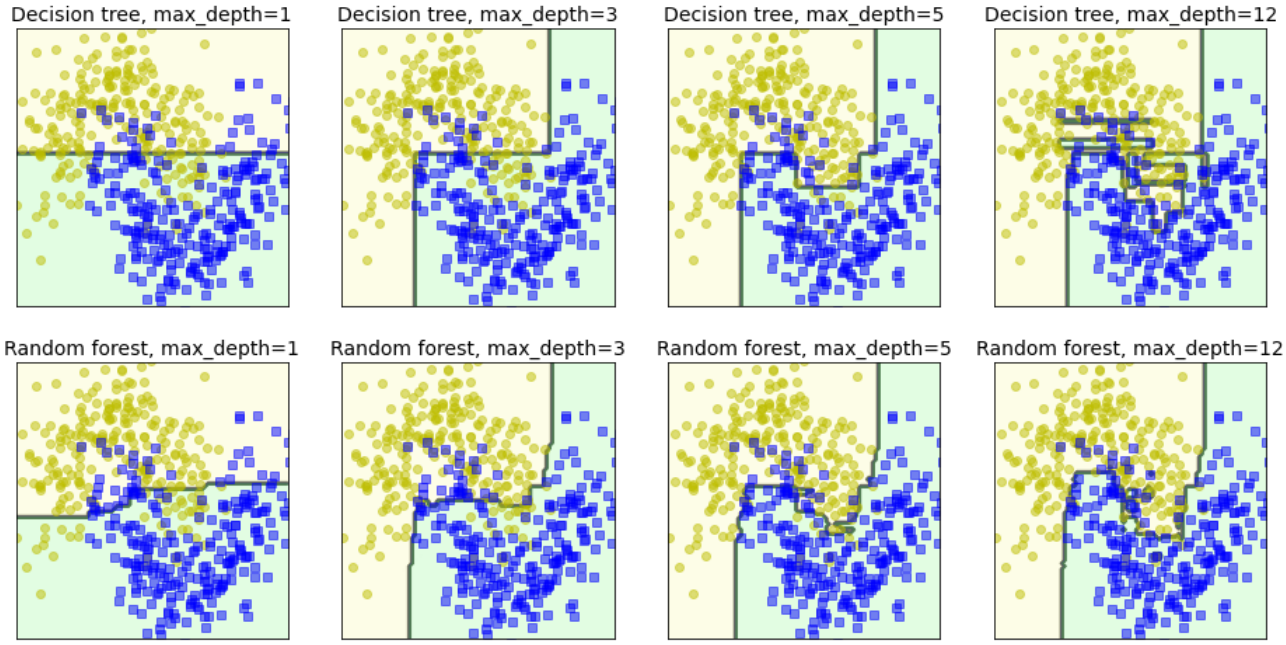
Случайный лес - инструмент для уменьшения variance нашей модели. Можно показать, что при стремлении числа моделей в ансамбле к бесконечности, а их коррелированности - к 0, variance ансамбля стремится к нулю. Однако при этом bias ансамбля будет равен bias базовой модели

Продемонстрируем это

```
plt.figure(figsize=(16,8))

plt.subplot(241)
dt1 = DecisionTreeClassifier(max_depth=1)
dt1.fit(X_train, Y_train)
plot_decision_boundary(dt1, X, Y)
plt.xticks([], [])
plt.yticks([], [])
plt.title("Decision tree, max_depth=1", fontsize=14)
plt.xlabel("")
plt.ylabel("")
plt.xticks([], [])
plt.yticks([], [])

plt.subplot(242)
dt3 = DecisionTreeClassifier(max_depth=3)
dt3.fit(X_train, Y_train)
plot_decision_boundary(dt3, X, Y)
plt.title("Decision tree, max_depth=3", fontsize=14)
plt.xlabel("")
plt.ylabel("")
plt.xticks([], [])
plt.yticks([], [])

plt.subplot(243)
dt5 = DecisionTreeClassifier(max_depth=5)
dt5.fit(X_train, Y_train)
plot_decision_boundary(dt5, X, Y)
plt.title("Decision tree, max_depth=5", fontsize=14)
```

```
plt.xlabel("")
plt.ylabel("")
plt.xticks([], [])
plt.yticks([], [])

plt.subplot(244)
dt12 = DecisionTreeClassifier(max_depth=12)
dt12.fit(X_train, Y_train)
plot_decision_boundary(dt12, X, Y)
plt.title("Decision tree, max_depth=12", fontsize=14)
plt.xlabel("")
plt.ylabel("")
plt.xticks([], [])
plt.yticks([], [])

plt.subplot(245)
rf1 = RandomForestClassifier(max_depth=1, n_estimators=500, n_jobs=-1)
rf1.fit(X_train, Y_train)
plot_decision_boundary(rf1, X, Y)
plt.title("Random forest, max_depth=1", fontsize=14)
plt.xlabel("")
plt.ylabel("")
plt.xticks([], [])
plt.yticks([], [])

plt.subplot(246)
rf3 = RandomForestClassifier(max_depth=3, n_estimators=500, n_jobs=-1)
rf3.fit(X_train, Y_train)
plot_decision_boundary(rf3, X, Y)
plt.title("Random forest, max_depth=3", fontsize=14)
plt.xlabel("")
plt.ylabel("")
plt.xticks([], [])
plt.yticks([], [])

plt.subplot(247)
rf5 = RandomForestClassifier(max_depth=5, n_estimators=500, n_jobs=-1)
rf5.fit(X_train, Y_train)
plot_decision_boundary(rf5, X, Y)
plt.title("Random forest, max_depth=5", fontsize=14)
plt.xlabel("")
plt.ylabel("")
plt.xticks([], [])
plt.yticks([], [])

plt.subplot(248)
rf12 = RandomForestClassifier(max_depth=12, n_estimators=500, n_jobs=-1)
rf12.fit(X_train, Y_train)
plot_decision_boundary(rf12, X, Y)
plt.title("Random forest, max_depth=12", fontsize=14)
plt.subplots_adjust()
plt.xlabel("")
plt.ylabel("")
plt.xticks([], [])
```

```
plt.yticks([], [])
plt.show()
```



Видим, что общая сложность решающей границы меняется не сильно - bias случайного леса по сравнению с bias дерева решений не меняется. А вот гладкость границы увеличивается, тем самым, уменьшается variance

Boosting

Взвешенное голосование

$$h(x) = \sum_i b_i a_i(x)$$

Не все модели предсказывают что-то одинаково хорошо. В случайном лесе у нас были деревья решений, которые мы обучали на равнозначных выборках, поэтому могли усреднять их предсказания.

Теперь представим, что у нас есть нейросеть, которую мы обучали на огромной выборке, есть линейная модель, которую мы учили на 200 объектах, есть случайный лес, который мы учили на выборке с множеством пропущенных признаков. Очевидно, что эти модели будут давать разное качество.

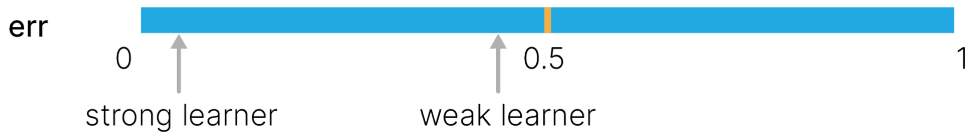
Разумно эти модели учитывать с каким-то весом класса. На основе этого построен метод под названием boosting.

Вы наверняка слышали о нем, потому что градиентный boosting — предмет большого количества мемов про машинное обучение, потому что он помогает получить решение, которое сложно побить другими видами моделей.

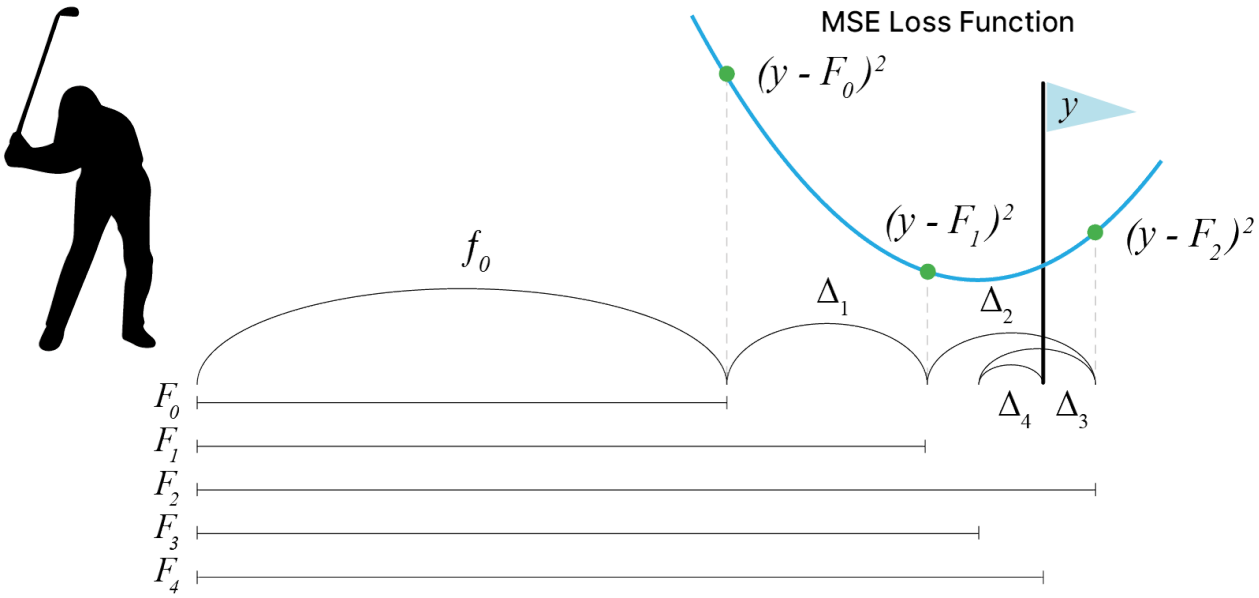
В то время как случайный лес представлял собой попытку стрелять по мишени много раз в надежде, что в среднем мы попадем в нее, идея бустинга больше похожа на игру в гольф. Пусть каждая модель будет представлять собой один слабенький удар клюшкой. А последующие удары — будут корректировать результаты этого удара, постепенно приближаясь к лунке.

Бустинг позволяет нам на основе большого числа слабых моделей получить одну сильную. Опять же, в качестве модели используется дерево решений, но не большой

высоты, а наоборот, маленькой. Чтобы оно само по себе в принципе не могло выучить выборку хорошо.



Цель i-й модели в ансамбле — скорректировать ошибки предыдущих i-1 моделей. В результате, когда мы суммируем вклады всех моделей ансамбля, получается хорошее предсказание.

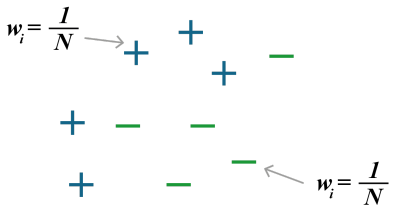


AdaBoost

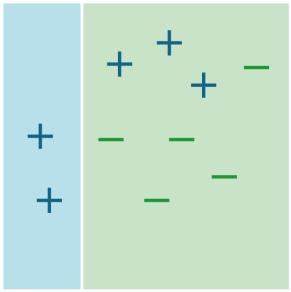
(Freund & Schapire, 1996)

Первый бустинг был предложен Фреундом и назывался AdaBoost. На нем легче всего объяснить всю идею бустинга.

Допустим, у нас есть выборка (плюсы и минусы). Мы хотим различать их между собой. Назначим вес каждому плюсу и минусу. Важно классифицировать их одинаково, поэтому их вес=1÷N, где N — число (суммарный объект).

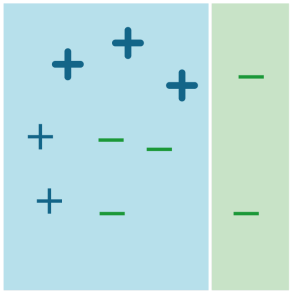


Построим первый предсказатель.

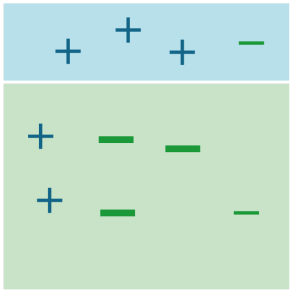


Он классифицировал объекты плохо, но лучше, чем если бы он это делал случайно. Некоторые объекты он определил верно, а на некоторых (три плюса) ошибся.

Тогда мы скажем, что эти плюсы важнее, чем все остальное, потому что на них мы уже научились всё предсказывать. Дальше обучим еще одну модель на той же выборке, но скажем, что эти плюсы важнее, чем все остальное объекты.

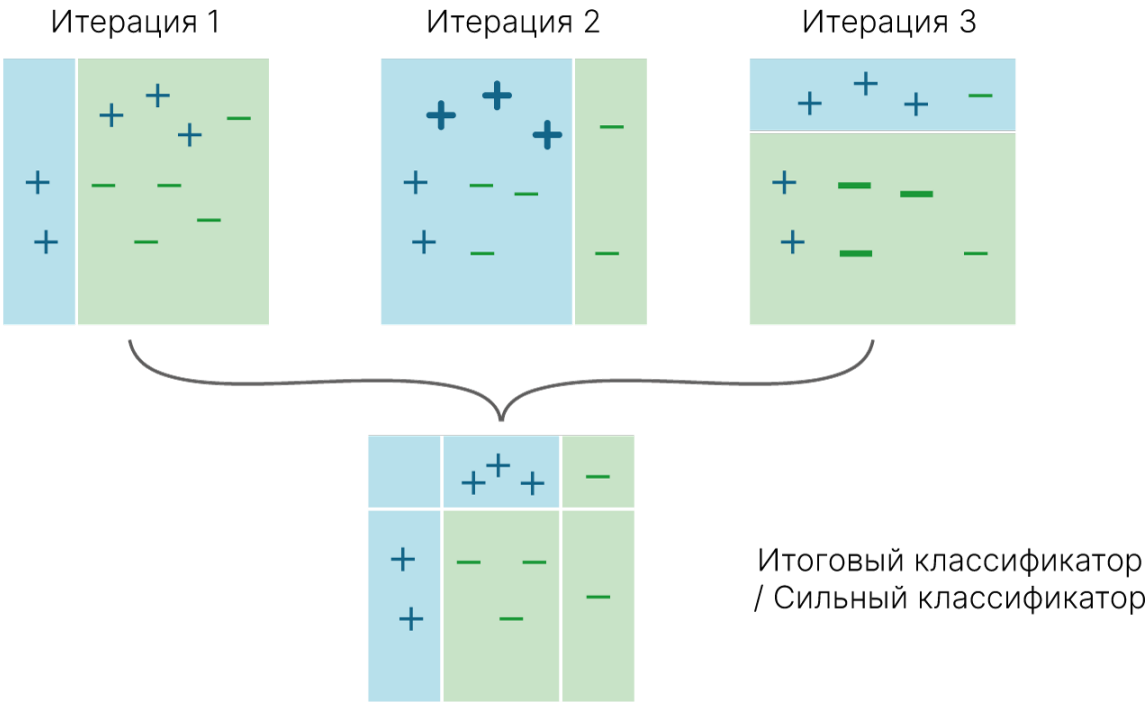


Новая модель будет ошибаться уже на минусах. Добавляем веса этим минусам.



Строим еще один классификатор, который правильно классифицирует важные для нас объекты, а на неважные не обращают внимания. Получилось три классификатора, которые по-разному ошибаются на обучающей выборке.

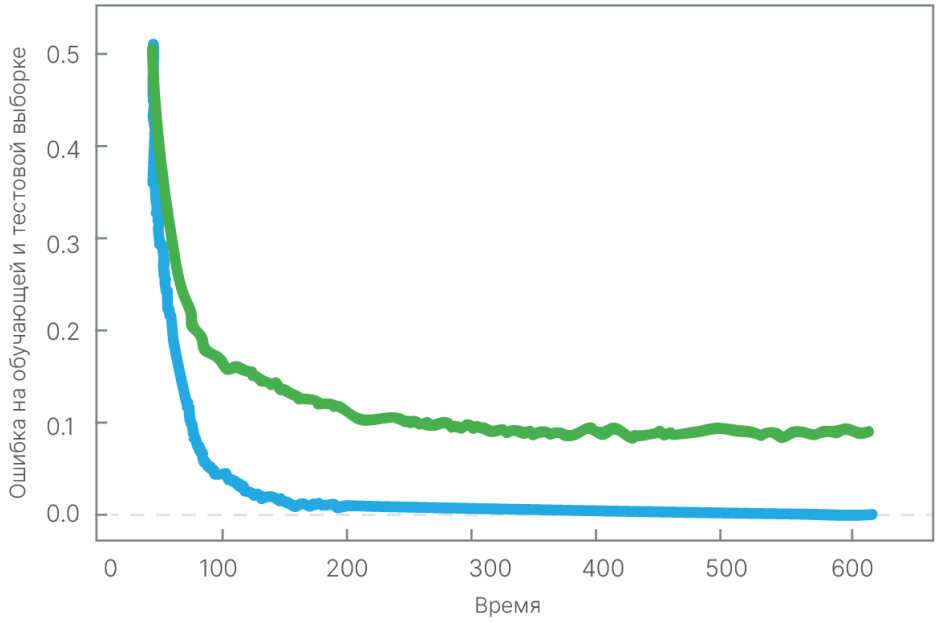
Но если сложить эти классификаторы с правильными весами, которые можно подсчитать по специальной формуле, то полученная модель будет решать нашу задачу классификации идеально.



Переобучается ли AdaBoost?

Сложился миф о том, что он не переобучается. На самом деле это нет так, просто когда он появился, для переобучения не хватало вычислительных мощностей (нужно сделать много итераций, чтобы переобучение увидеть).

Но у него было интересное свойство, которое унаследовали все последующие бустинги: вы можете идеально выучить обучающую выборку, но при этом вы добавляете новые деревья, и качество на тестовой выборке становится лучше. Это обусловлено тем, что каждое последующее дерево, которое мы добавляем, уточняет границу разделения, делает ее более плавной, и это зачастую помогает нашему классификатору.



Применимость AdaBoost

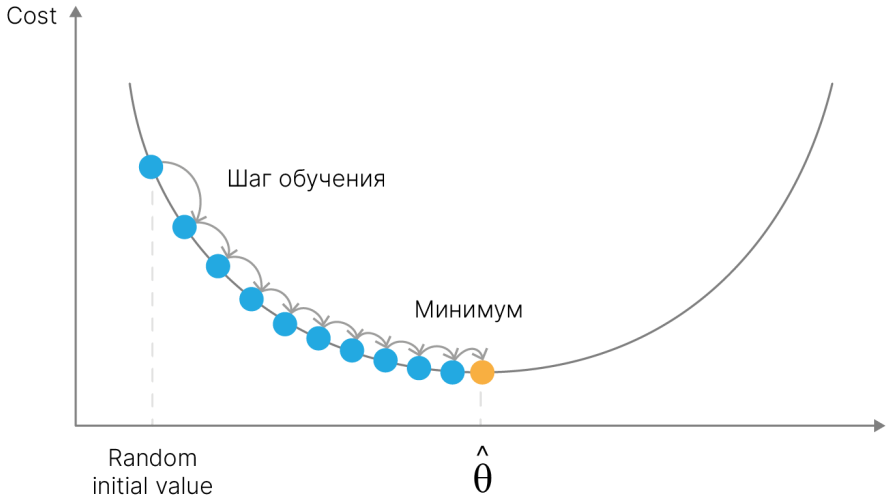
AdaBoost реализован в sklearn и он может использовать в качестве базовой любую модель, которая умеет работать с весами объектов в выборке.

На данный момент, тем не менее, AdaBoost уступил место градиентному бустингу и почти не используется. Мы обсудили его в образовательных целях — на нем легче всего понять основную идею бустинга.

Gradient boosting (градиентный бустинг)

Создатель продолжения AdaBoost — градиентного бустинга — Leo Breiman заметил, что бустинг можно рассматривать как итеративную минимизацию некой целевой функции ошибки.

Как нейронные сети в ходе своего обучения, так и бустинг (с помощью добавленных слабых моделей) минимизируют функции ошибки.



Как он их минимизирует? Точно так же, как нейронная сеть. Он использует градиентный спуск. Можно сделать следующее: у нас есть исходное приближение. Мы как-то ошибаемся. Считаем градиент ошибки, дальше пытаемся его аппроксимировать. Делаем новое приближение. Опять как-то ошибаемся — и повторяем все еще и еще.

Для объекта из тестовой выборки:

1. Первая модель обеспечивает нам исходное приближение.
2. Вторая модель пытается предсказать градиент ошибки. Зная градиент ошибки для тестового объекта, мы можем эту ошибку уменьшить, получив новое приближение.
3. Третья модель пытается предсказать градиент ошибки нового приближения. Опять, зная его, можем получить еще одно, более точное приближение.
4. Повторяем, пока не пройдем все N моделей.

Оказывается, такой подход позволяет достигать state-of-the art результатов при работе с табличными данными.

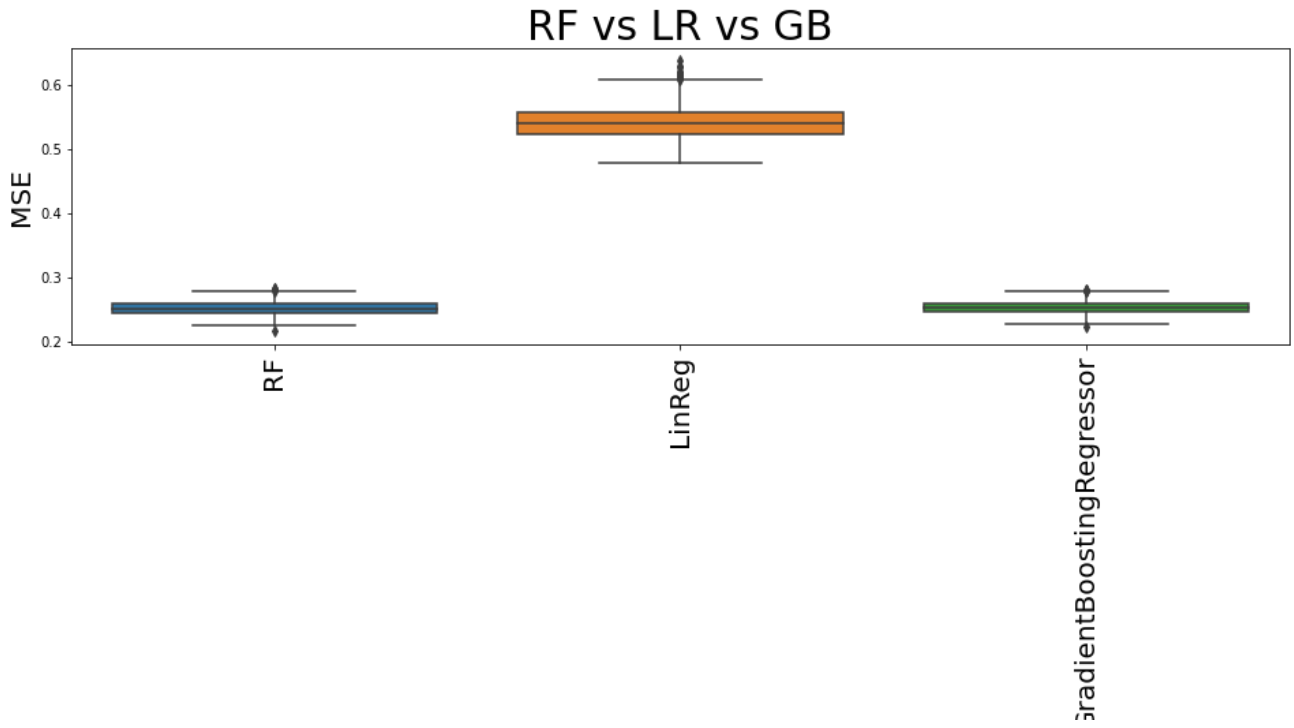
Например, запустим градиентный бустинг "почти" наугад:

```
X = calif_housing.data
Y = calif_housing.target
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, random_state=42)
```

```
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import GradientBoostingRegressor, RandomForestRegressor
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline

models = {}
models['RF'] = RandomForestRegressor(n_estimators=250,  # for better result set to 1000
                                     max_depth=None,
                                     min_samples_leaf=1,
                                     n_jobs=-1,
                                     random_state=42)
# make pipeline to normalization
models['LinReg'] = make_pipeline(StandardScaler(with_mean=False), LinearRegression())
models['GradientBoostingRegressor'] = GradientBoostingRegressor(learning_rate=0.1, # fo
                                                                n_estimators=250, # for
                                                                random_state=42)

# Training models
results_boost = train_and_test_regressor(models, X_train, Y_train, X_test, Y_test)
```

```
Fitting RF
Fitting LinReg
Fitting GradientBoostingRegressor
Calculating bootstrap score for RF
Calculating bootstrap score for LinReg
Calculating bootstrap score for GradientBoostingRegressor
```

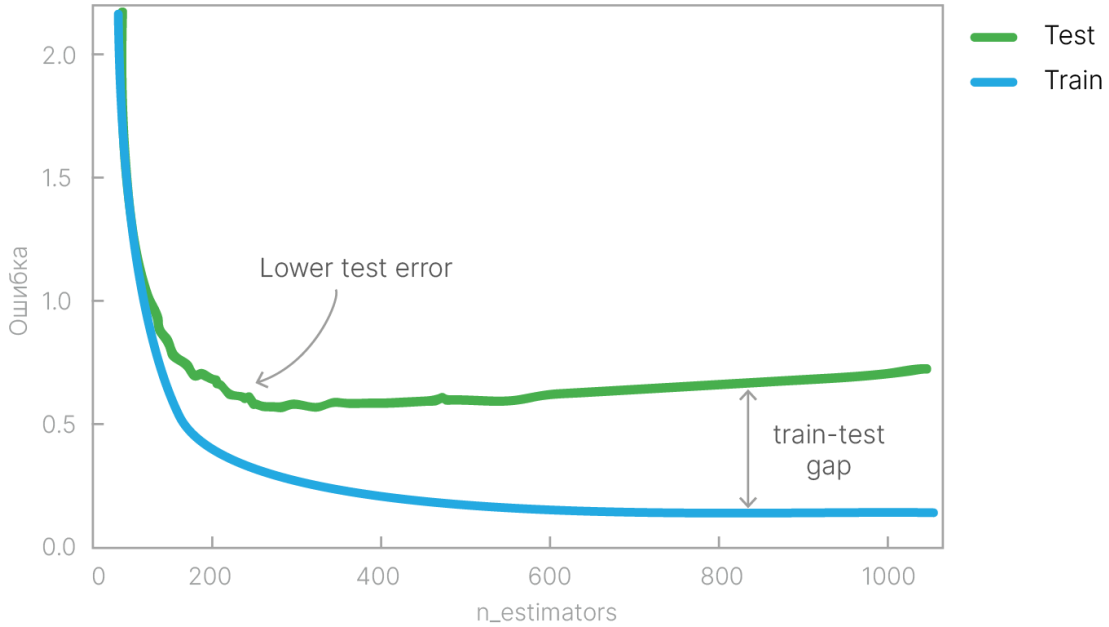
```
plt.figure(figsize=(16,4))
ax = sns.boxplot(data=results_boost , y="mse", x="model")
plt.setp(ax.get_xticklabels(), rotation=90)
plt.xlabel("", size=20)
plt.ylabel("MSE", size=20)
plt.title("RF vs LR vs GB", size=30)
plt.xticks(size=20)
plt.show()
```



Видим, что сходу мы побили случайный лес и линейную регрессию.

Переобучение

В то же время, **Gradient boosting**, в отличие от случайного леса, может сильно переобучиться. Это важно понимать. Для небольших датасетов часто может оказаться, что случайный лес дает более надежные результаты.

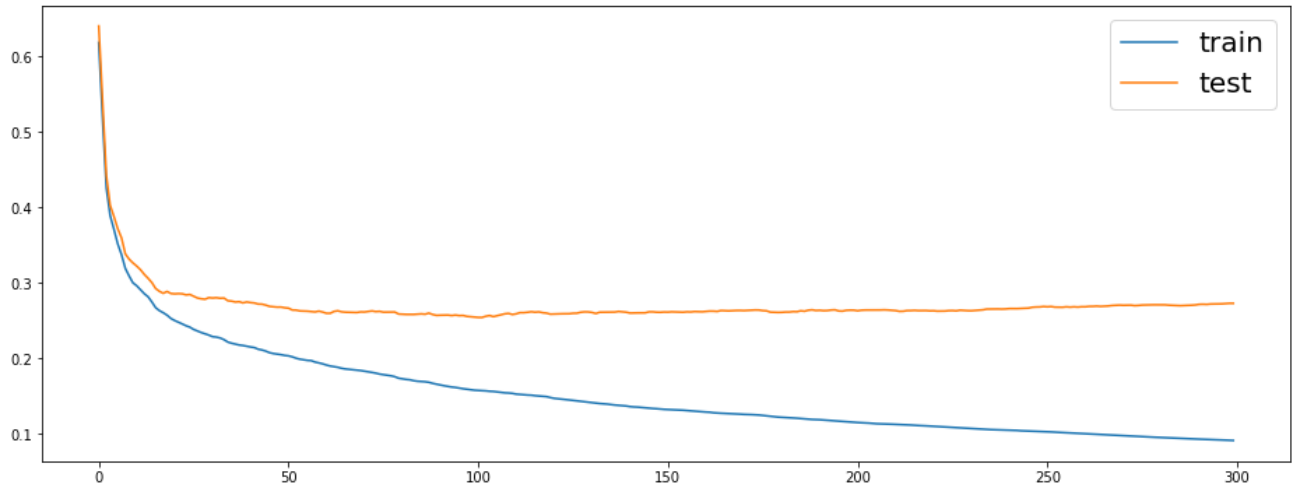



```
gbtree = GradientBoostingRegressor(n_estimators=300,
                                   learning_rate=1 # faster learning rate to force ovef
                                   )
gbtree.fit(X_train, Y_train)
```

```
GradientBoostingRegressor(learning_rate=1, n_estimators=300)
```

Используем метод, который есть во многих реализациях бустинга в том или ином виде — staged_predict, который получать предсказание от первых i деревьев ансамбля, что позволяет быстро строить график качества градиентного бустинга в зависимости от числа базовых моделей:

```
error_train = []
error_test = []
for it, (Y_train_pred, Y_test_pred) in enumerate(zip(gbtree.staged_predict(X_train),
                                                     gbtree.staged_predict(X_test))):
    ertr = mean_squared_error(y_true=Y_train, y_pred=Y_train_pred)
    error_train.append(ertr)
    erte = mean_squared_error(y_true=Y_test, y_pred=Y_test_pred)
    error_test.append(erte)


plt.figure(figsize=(16,6))
plt.plot(error_train, label='train')
plt.plot(error_test, label='test')
plt.legend(fontsize=20)
plt.show()
```



Здесь мы *немного* схитрили, сильно увеличив learning rate нашей модели — дополнительный множитель к весу, с которым добавляются в нее новые модели. Из-за этого даже на таком простом датасете мы в состоянии увидеть явное переобучение.

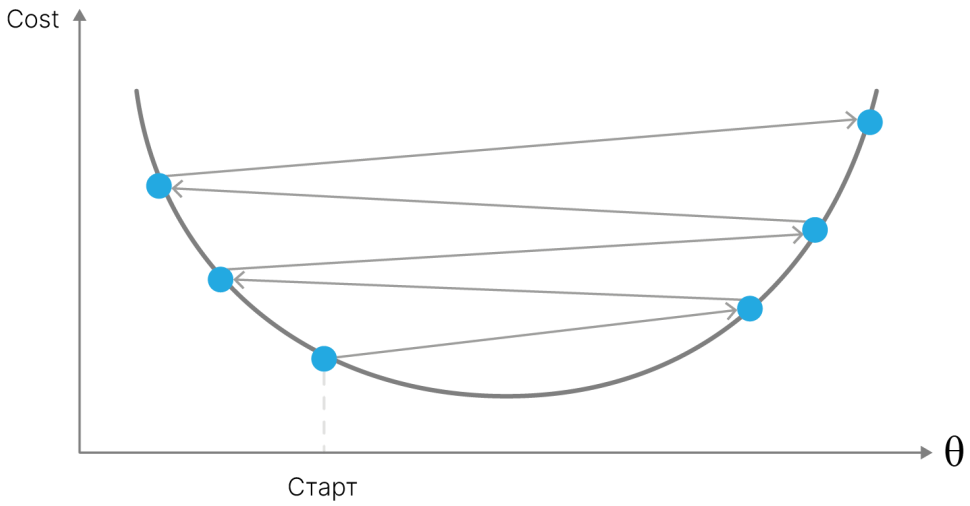
Давайте об этом learning rate и поговорим.

Shrinkage (learning rate)

Как у градиентного спуска есть learning rate, который определяет силу каждого нашего следующего шага, так и у градиентного бустинга есть параметр, который называется shrinkage или learning rate — это дополнительный параметр, на который мы домножаем вес, с которым мы добавляем новые модели в ансамбль.

Мы не хотим сильно скакать по пространству решений, мы хотим спускаться медленно и сойтись к какому-то хорошему минимуму. Потому мы берем и вес каждой модели, которую мы рассчитываем по специальному алгоритму, еще домножаем на маленький коэффициент, который называется shrinkage или learning rate. Фактически, мы берем и умножаем градиент на некий альфа (это то же, что было в градиентном спуске, ничего нового мы не добавили).

Если не домножать вес каждой модели дополнительно на этот параметр, то мы можем попасть в ситуацию, которая будем пролетать мимо минимума функции ошибки (та же опасность, что и в обычном градиентном спуске).

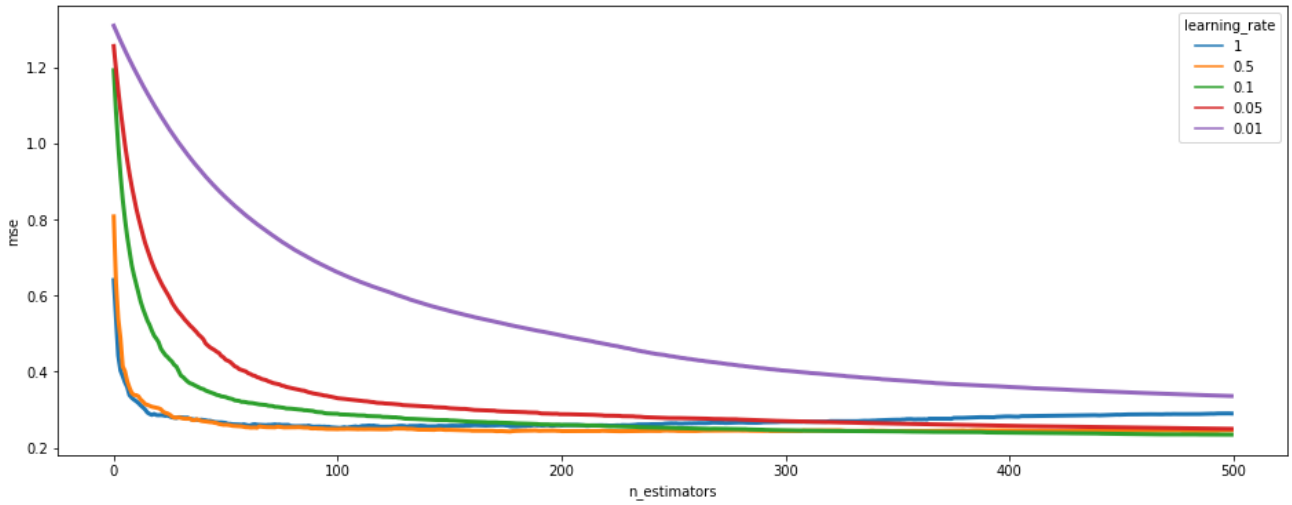


Как и в случае с градиентным спуском, **learning rate** влияет не только на то, как быстро мы станем переобучаться, но и на глубину минимума, который мы найдем.

```
# here anb below in the cell can be set to 1000 for better visualization
gbtree1 = GradientBoostingRegressor(n_estimators=500, learning_rate=1)
gbtree1.fit(X_train, Y_train)

gbtree2 = GradientBoostingRegressor(n_estimators=500, learning_rate=0.5)
gbtree2.fit(X_train, Y_train)

gbtree3 = GradientBoostingRegressor(n_estimators=500, learning_rate=0.1)
gbtree3.fit(X_train, Y_train)

gbtree4 = GradientBoostingRegressor(n_estimators=500, learning_rate=0.05)
gbtree4.fit(X_train, Y_train)

gbtree5 = GradientBoostingRegressor(n_estimators=500, learning_rate=0.01)
gbtree5.fit(X_train, Y_train)
```

```
GradientBoostingRegressor(learning_rate=0.01, n_estimators=500)
```

```
lr = []
step = []
mse = []
for gb_tree in (gbtree1, gbtree2, gbtree3, gbtree4, gbtree5):

    for it, Y_test_pred in enumerate(gb_tree.staged_predict(X_test)):

        erte = mean_squared_error(y_true=Y_test, y_pred=Y_test_pred)
        mse.append(erte)
        lr.append(str(gb_tree.learning_rate))
        step.append(it)

df = pd.DataFrame({"learning_rate":lr,
                   "n_estimators": step,
                   "mse": mse})

plt.figure(figsize=(16,6))
sns.lineplot(data=df, x='n_estimators', y='mse', hue='learning_rate', lw=3)
plt.show()
```



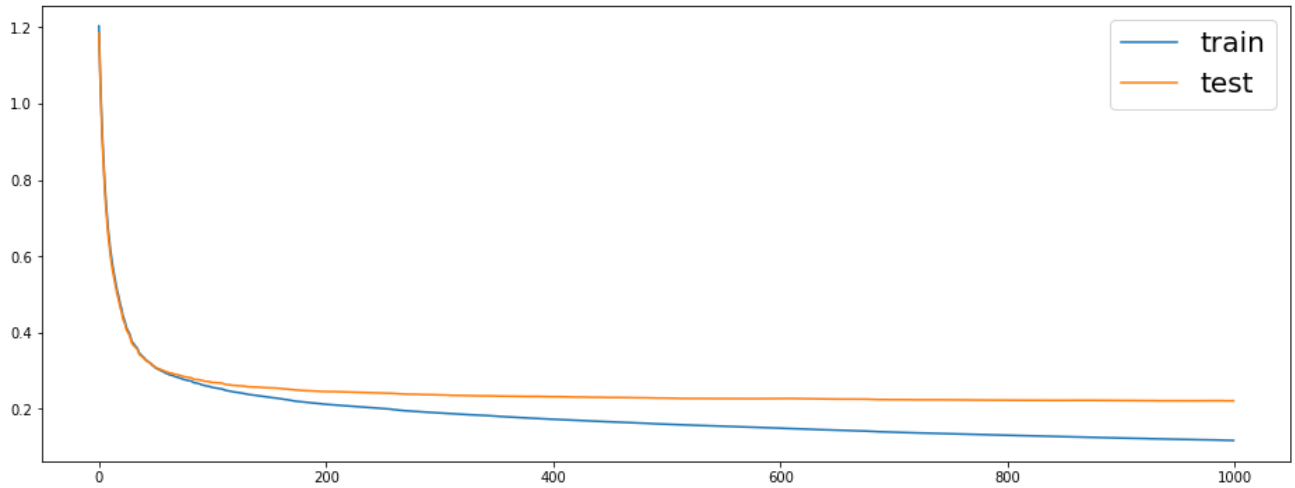
В данном случае мы видим, что значение learning rate, установленного по-умолчанию, позволяет получить наименьшую ошибку на валидации. В то же время слишком большие значения learning rate (1 и 0.5) приводят к тому, что мы не достигаем таких глубоких минимумов и, более того, начинаем переобучаться. При этом слишком маленькое значение learning rate может привести к тому, что нам понадобится очень большое число деревьев, чтобы достигнуть минимума (если мы его вообще достигнем).

Число деревьев в ансамбле

Число деревьев, как было видно из предыдущего графика, зависит от установленного learning rate. Потому обычно ставят learning_rate равным 0.1 и подбирают оптимальное значение числа деревьев в ансамбле. Делают это на кросс-валидации, но для экономии времени просто дополнительно разобьем train датасет:

```
X_learn, X_valid, Y_learn, Y_valid = train_test_split(X_train, Y_train, random_state=42
```

```
gbtree = GradientBoostingRegressor(n_estimators=1000, learning_rate=0.1)
gbtree.fit(X_learn, Y_learn)

error_train = []
error_test = []
for it, (Y_learn_pred, Y_valid_pred) in enumerate(zip(gbtree.staged_predict(X_learn),
                                                      gbtree.staged_predict(X_valid))):
    ertr = mean_squared_error(y_true=Y_learn, y_pred=Y_learn_pred)
    error_train.append(ertr)
    erte = mean_squared_error(y_true=Y_valid, y_pred=Y_valid_pred)
    error_test.append(erte)


plt.figure(figsize=(16,6))
plt.plot(error_train, label='train')
plt.plot(error_test, label='test')
plt.legend(fontsize=20)
plt.show()
```



Видим, что ошибка продолжает медленно уменьшаться, но в принципе увеличение идет очень маленькое, потому в целях демонстрации выберем число деревьев равным 500.

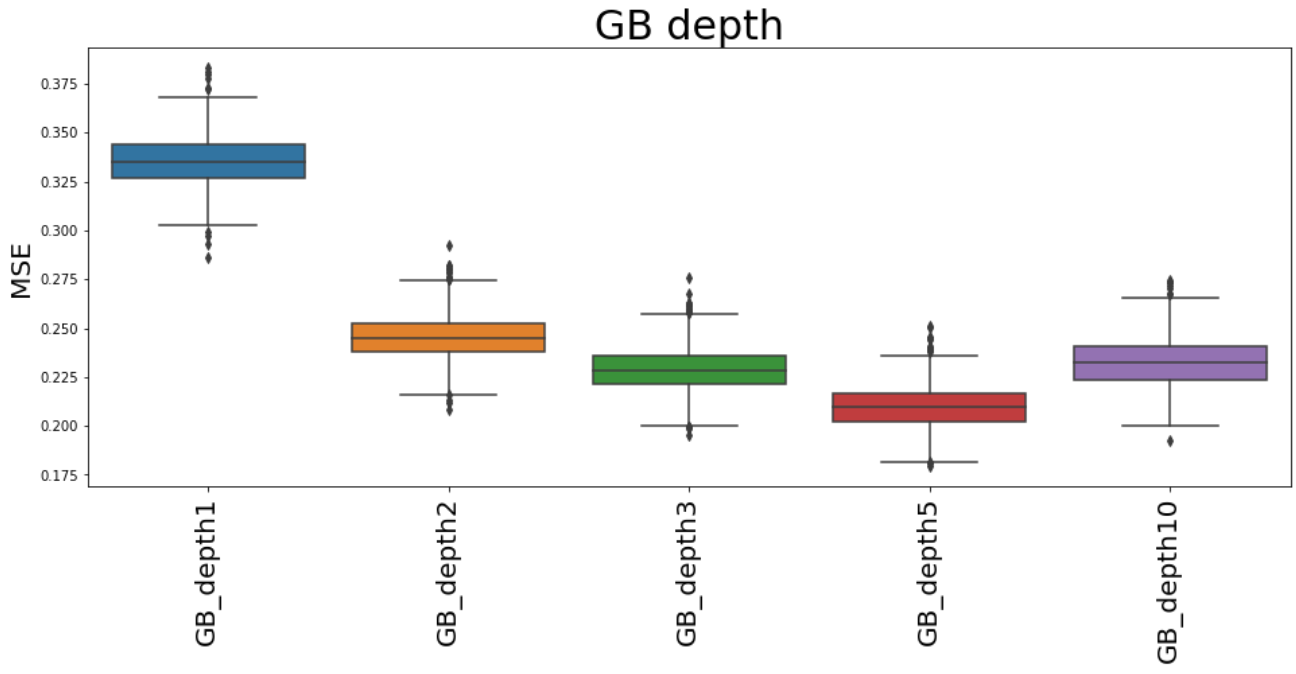
Глубина деревьев в градиентном бустинге

В случае бустинга нам нужны **слабые модели**. Потому очень глубокие деревья в бустинге встречаются редко — бустинг с ними проиграет по качеству. Обычно глубина дерева выбирается вместе с минимальным числом объектов в листе (min_samples_leaf) или весе листа (min_weight_fraction_leaf), так как эти параметры взаимосвязаны и вместе влияют на сложность полученных деревьев. Здесь же, опять, в целях демонстрации мы подберем сначала глубину дерева, а потом минимальное число объектов.

```
models = {}
for depth in (1, 2, 3, 5, 10):
    models[f"GB_depth{depth}"] = GradientBoostingRegressor(n_estimators=500,
                                    learning_rate=0.1,
                                    max_depth=depth,
                                    random_state=42)

# Train model
depth_boost = train_and_test_regressor(models, X_learn, Y_learn, X_valid, Y_valid)
```

```
Fitting GB_depth1
Fitting GB_depth2
Fitting GB_depth3
Fitting GB_depth5
Fitting GB_depth10
Calculating bootstrap score for GB_depth1
Calculating bootstrap score for GB_depth2
Calculating bootstrap score for GB_depth3
Calculating bootstrap score for GB_depth5
Calculating bootstrap score for GB_depth10
```

```
plt.figure(figsize=(16,6))
ax = sns.boxplot(data=depth_boost , y="mse", x="model")
plt.setp(ax.get_xticklabels(), rotation=90)
plt.xlabel("", size=20)
plt.ylabel("MSE", size=20)
plt.title("GB depth", size=30)
plt.xticks(size=20)
plt.show()
```

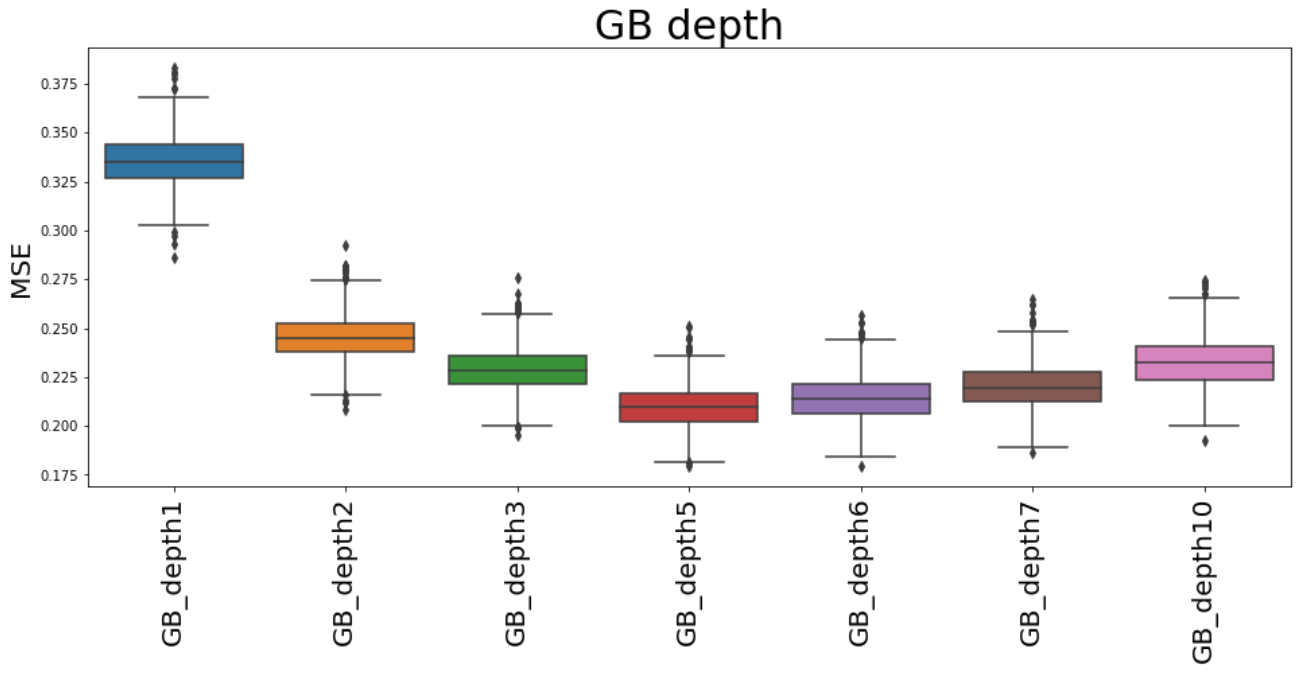


Искомая глубина находится между 3 и 10. Проверим 4, 6 и 7.

```
models_add = {}
for depth in (4, 6, 7):
    models_add[f"GB_depth{depth}"] = GradientBoostingRegressor(n_estimators=500,
                                     learning_rate=0.1,
                                     max_depth=depth,
                                     random_state=42)

# Train model
depth_boost_add = train_and_test_regressor(models_add, X_learn, Y_learn, X_valid, Y_val
```

```
Fitting GB_depth4
Fitting GB_depth6
Fitting GB_depth7
Calculating bootstrap score for GB_depth4
Calculating bootstrap score for GB_depth6
Calculating bootstrap score for GB_depth7
```

```
depth_boost_joined = pd.concat([depth_boost, depth_boost_add])

plt.figure(figsize=(16,6))
ax = sns.boxplot(data=depth_boost_joined , y="mse", x="model", order=[f"GB_depth{i}" fo
plt.setp(ax.get_xticklabels(), rotation=90)
plt.xlabel("", size=20)
plt.ylabel("MSE", size=20)
plt.title("GB depth", size=30)
plt.xticks(size=20)
plt.show()
```



Значит, оптимальная глубина — 5.

Минимальное число объектов в листе

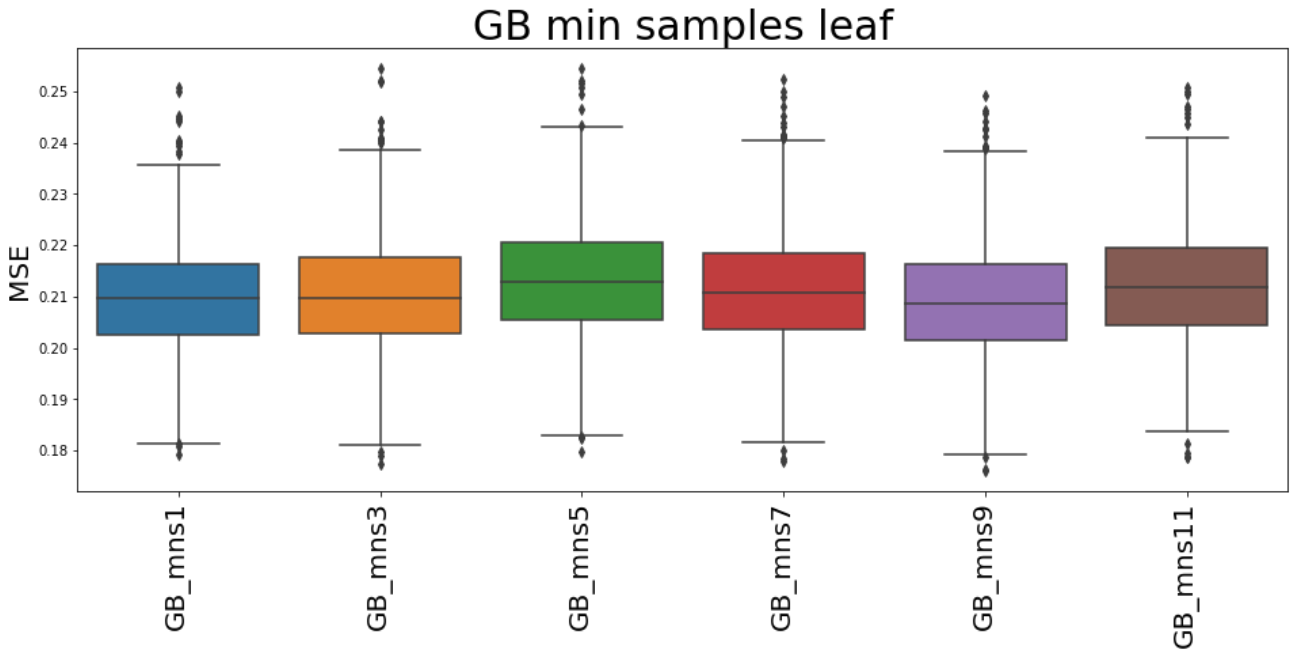
Опять же, этот параметр влияет на сложность полученных деревьев. Как уже отмечалось ранее, стоит подбирать его вместе с глубиной, но мы ограничены по времени.

```
models = {}
for min_samples_leaf in (1, 3, 5, 7, 9, 11):
    models[f"GB_mns{min_samples_leaf}"] = GradientBoostingRegressor(n_estimators=500,
                                    learning_rate=0.1,
                                    max_depth=5,
                                    min_samples_leaf=min_samples_leaf,
                                    random_state=42)

# Train model
mns_boost = train_and_test_regressor(models, X_learn, Y_learn, X_valid, Y_valid)
```

```
Fitting GB_mns1
Fitting GB_mns3
Fitting GB_mns5
Fitting GB_mns7
Fitting GB_mns9
Fitting GB_mns11
Calculating bootstrap score for GB_mns1
Calculating bootstrap score for GB_mns3
Calculating bootstrap score for GB_mns5
Calculating bootstrap score for GB_mns7
```

```
Calculating bootstrap score for GB_mns9
Calculating bootstrap score for GB_mns11
```

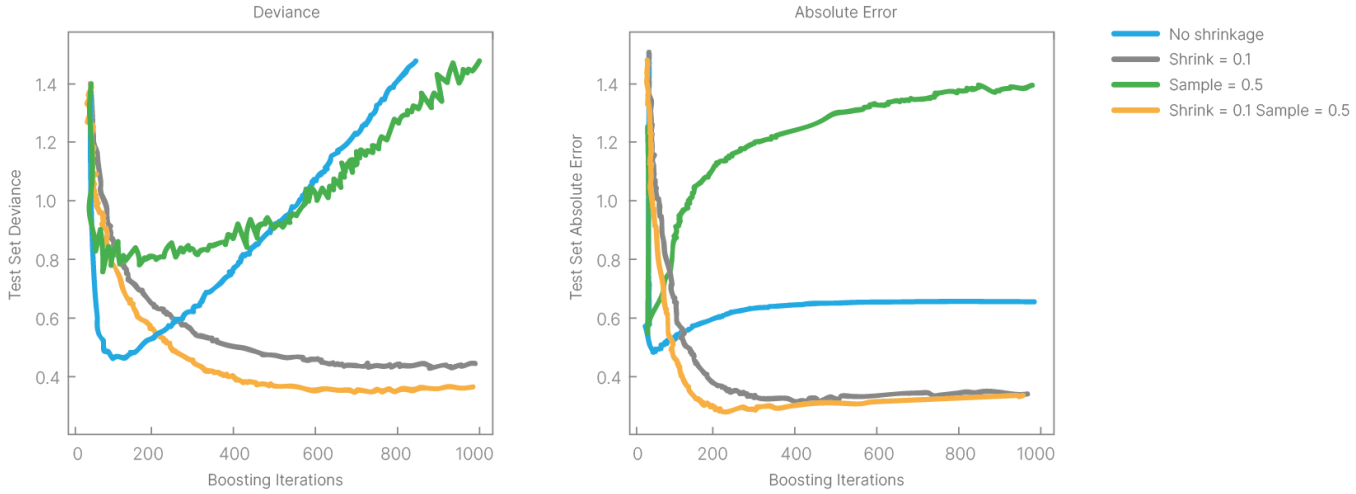
```
plt.figure(figsize=(16,6))
ax = sns.boxplot(data=mns_boost , y="mse", x="model")
plt.setp(ax.get_xticklabels(), rotation=90)
plt.xlabel("", size=20)
plt.ylabel("MSE", size=20)
plt.title("GB min samples leaf", size=30)
plt.xticks(size=20)
plt.show()
```



Поставим число объектов в листе равное 9.

Subsampling-and-max_features-and-max_features and max_features

Можно каждому дереву в ансамбле давать только часть объектов из выборки — получим стохастический градиентный бустинг. За это отвечает параметр subsample. Аналогично, можно давать каждому дереву в ансамбле лишь часть признаков — max_features. Это может давать дополнительный прирост качества. Сейчас мы этот этап опускаем, так как у нас мало признаков и не очень много объектов, а также в целях экономии времени.



Кроме того, мы можем каждое дерево еще немножко рандомизировать. Можем давать ему не всю выборку, не все признаки. Это работает не всегда, гиперпараметры, контролирующие это поведение, нужно подбирать.

Понижение learning rate

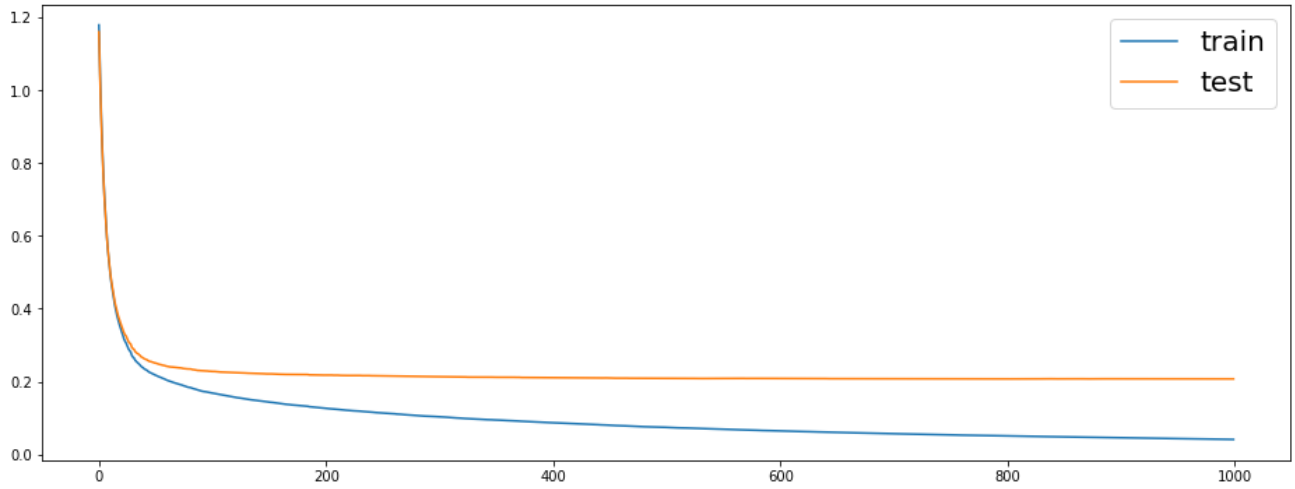
После того, как мы подобрали остальные параметры, можно попытаться выиграть дополнительное качество за счет понижения learning_rate и одновременное увеличение числа деревьев ансамбля.

Опять же, этот этап пропустим.

Построим график качества для текущего learning rate — мы же брали не оптимальное число предсказателей, посмотрим, можно ли уже на финальном этапе взять побольше.

```
gbtree = GradientBoostingRegressor(n_estimators=1000,
                                   max_depth=5,
                                   min_samples_leaf=9,
                                   learning_rate=0.1,
                                   random_state=42)
gbtree.fit(X_learn, Y_learn)

error_train = []
error_test = []
for it, (Y_learn_pred, Y_valid_pred) in enumerate(zip(gbtree.staged_predict(X_learn),
                                                      gbtree.staged_predict(X_valid))):
    ertr = mean_squared_error(y_true=Y_learn, y_pred=Y_learn_pred)
    error_train.append(ertr)
    erte = mean_squared_error(y_true=Y_valid, y_pred=Y_valid_pred)
    error_test.append(erte)


plt.figure(figsize=(16,6))
plt.plot(error_train, label='train')
plt.plot(error_test, label='test')
```

```
plt.legend(fontsize=20)
plt.show()
```

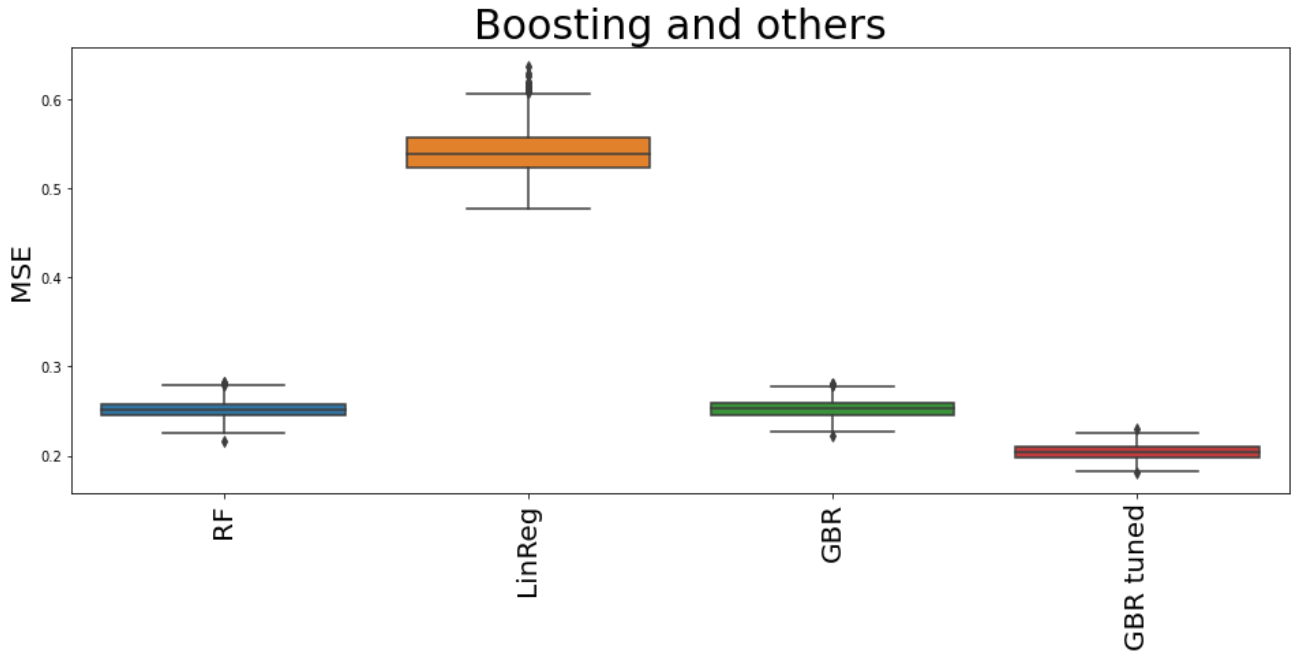


Ок, возьмем 500 деревьев.

```
models = {}

models['RF'] = RandomForestRegressor(n_estimators=250,
                                     max_depth=None,
                                     min_samples_leaf=1,
                                     n_jobs=-1,
                                     random_state=42)

models['LinReg'] = make_pipeline(StandardScaler(with_mean=False), LinearRegression())

models['GBR'] = GradientBoostingRegressor(learning_rate=0.1,
                                          n_estimators=250,
                                          random_state=42)

models['GBR tuned'] = GradientBoostingRegressor(learning_rate=0.1,
                                                n_estimators=500,
                                                max_depth=5,
                                                min_samples_leaf=9,
                                                random_state=42)

# Train models
tuned_boost = train_and_test_regressor(models, X_train, Y_train, X_test, Y_test)
```

```
Fitting RF
Fitting LinReg
Fitting GBR
Fitting GBR tuned
Calculating bootstrap score for RF
Calculating bootstrap score for LinReg
Calculating bootstrap score for GBR
Calculating bootstrap score for GBR tuned
```

```
plt.figure(figsize=(16,6))
ax = sns.boxplot(data=tuned_boost , y="mse", x="model")
plt.setp(ax.get_xticklabels(), rotation=90)
plt.xlabel("", size=20)
plt.ylabel("MSE", size=20)
plt.title("Boosting and others", size=30)
plt.xticks(size=20)
plt.show()
```



Видим, что мы дополнительно уменьшили ошибку нашего метода

Градиентный бустинг и bias-variance tradeoff

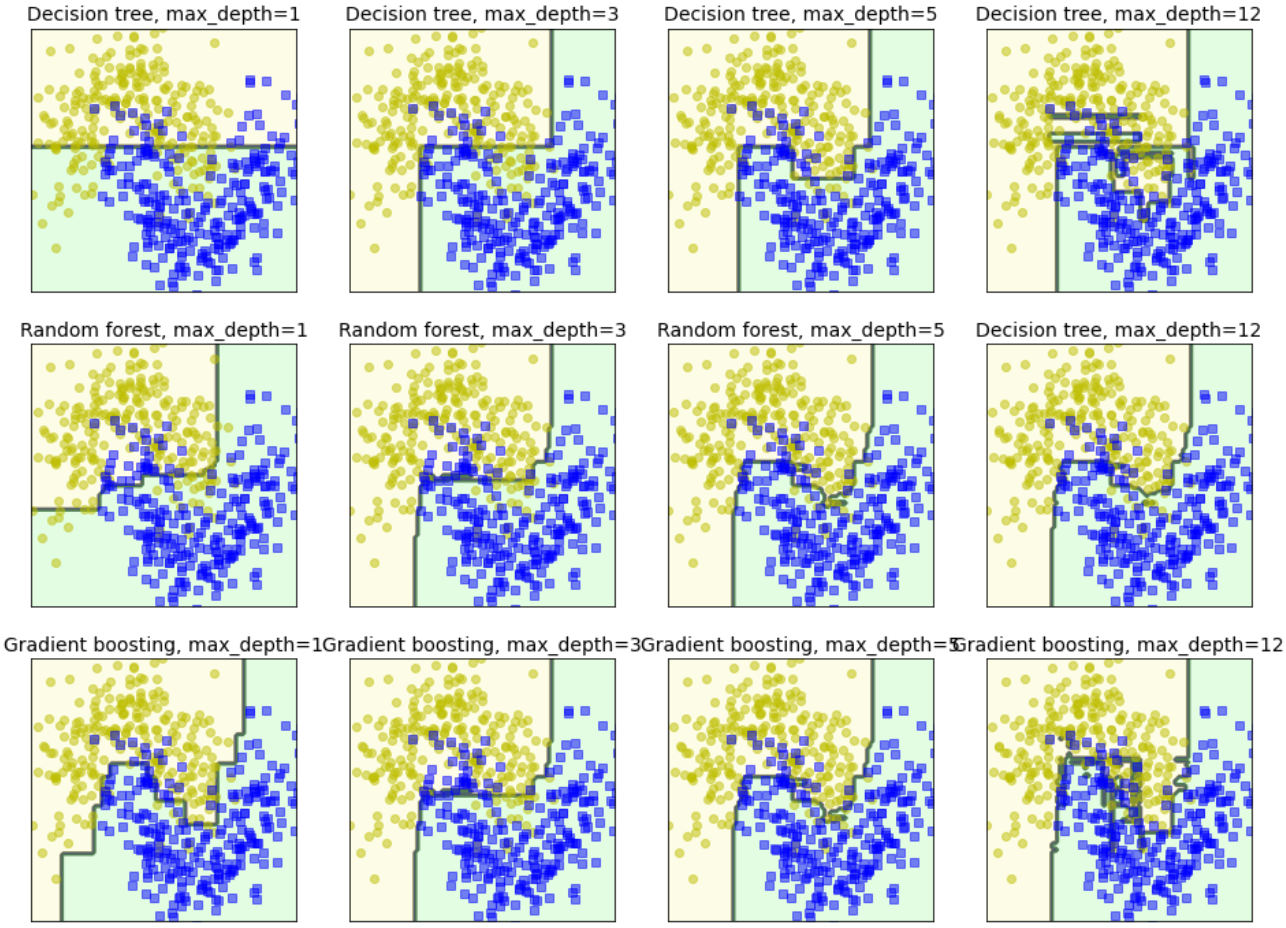
В то время как случайный лес только уменьшал variance модели, бустинг стремится уменьшить и bias, и variance.

```
from sklearn.ensemble import GradientBoostingClassifier

X, Y = sklearn.datasets.make_moons(n_samples=500, noise=0.30, random_state=42)
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, random_state=42)


plt.figure(figsize=(16,12))
plt.subplot(341)
```

```
dt1 = DecisionTreeClassifier(max_depth=1)
dt1.fit(X_train, Y_train)
plot_decision_boundary(dt1, X, Y)
plt.xticks([], [])
plt.yticks([], [])

plt.title("Decision tree, max_depth=1", fontsize=14)
plt.xlabel("")
plt.ylabel("")
plt.xticks([], [])
plt.yticks([], [])

plt.subplot(342)
dt3 = DecisionTreeClassifier(max_depth=3)
dt3.fit(X_train, Y_train)
plot_decision_boundary(dt3, X, Y)
plt.title("Decision tree, max_depth=3", fontsize=14)
plt.xlabel("")
plt.ylabel("")
plt.xticks([], [])
plt.yticks([], [])

plt.subplot(343)
dt5 = DecisionTreeClassifier(max_depth=5)
dt5.fit(X_train, Y_train)
plot_decision_boundary(dt5, X, Y)
plt.title("Decision tree, max_depth=5", fontsize=14)
plt.xlabel("")
plt.ylabel("")
plt.xticks([], [])
plt.yticks([], [])

plt.subplot(344)
dt12 = DecisionTreeClassifier(max_depth=12)
dt12.fit(X_train, Y_train)
plot_decision_boundary(dt12, X, Y)
plt.title("Decision tree, max_depth=12", fontsize=14)
plt.xlabel("")
plt.ylabel("")
plt.xticks([], [])
plt.yticks([], [])

plt.subplot(345)
rf1 = RandomForestClassifier(max_depth=1, n_estimators=500, n_jobs=-1)
rf1.fit(X_train, Y_train)
plot_decision_boundary(rf1, X, Y)
plt.title("Random forest, max_depth=1", fontsize=14)
plt.xlabel("")
plt.ylabel("")
plt.xticks([], [])
plt.yticks([], [])

plt.subplot(346)
rf3 = RandomForestClassifier(max_depth=3, n_estimators=500, n_jobs=-1)
rf3.fit(X_train, Y_train)
```

```
plot_decision_boundary(rf3, X, Y)
plt.title("Random forest, max_depth=3", fontsize=14)
plt.xlabel("")
plt.ylabel("")
plt.xticks([], [])
plt.yticks([], [])

plt.subplot(347)
rf5 = RandomForestClassifier(max_depth=5, n_estimators=500, n_jobs=-1)
rf5.fit(X_train, Y_train)
plot_decision_boundary(rf5, X, Y)
plt.title("Random forest, max_depth=5", fontsize=14)
plt.xlabel("")
plt.ylabel("")
plt.xticks([], [])
plt.yticks([], [])

plt.subplot(348)
rf12 = RandomForestClassifier(max_depth=5, n_estimators=500, n_jobs=-1)
rf12.fit(X_train, Y_train)
plot_decision_boundary(rf12, X, Y)
plt.title("Decision tree, max_depth=12", fontsize=14)
plt.subplots_adjust()
plt.xlabel("")
plt.ylabel("")
plt.xticks([], [])
plt.yticks([], [])

plt.subplot(349)
boost1 = GradientBoostingClassifier(max_depth=1, n_estimators=250)
boost1.fit(X_train, Y_train)
plot_decision_boundary(boost1, X, Y)
plt.title("Gradient boosting, max_depth=1", fontsize=14)
plt.xlabel("")
plt.ylabel("")
plt.xticks([], [])
plt.yticks([], [])

plt.subplot(3, 4, 10)
boost3 =  GradientBoostingClassifier(max_depth=3, n_estimators=250)
boost3.fit(X_train, Y_train)
plot_decision_boundary(rf3, X, Y)
plt.title("Gradient boosting, max_depth=3", fontsize=14)
plt.xlabel("")
plt.ylabel("")
plt.xticks([], [])
plt.yticks([], [])

plt.subplot(3, 4, 11)
boost5 =  GradientBoostingClassifier(max_depth=5, n_estimators=250)
boost5.fit(X_train, Y_train)
plot_decision_boundary(rf5, X, Y)
plt.title("Gradient boosting, max_depth=5", fontsize=14)
plt.xlabel("")
plt.ylabel("")
```

```
plt.xticks([], [])
plt.yticks([], [])

plt.subplot(3, 4, 12)
boost12 =  GradientBoostingClassifier(max_depth=5, n_estimators=250)
boost12.fit(X_train, Y_train)
plot_decision_boundary(boost12, X, Y)
plt.title("Gradient boosting, max_depth=12", fontsize=14)
plt.subplots_adjust()
plt.xlabel("")
plt.ylabel("")
plt.xticks([], [])
plt.yticks([], [])
plt.show()
```



Модификации градиентного бустинга

Есть много модификаций градиентного бустинга. В отличие от реализации в sklearn, большая часть из них умеет параллелиться на CPU или даже на GPU.

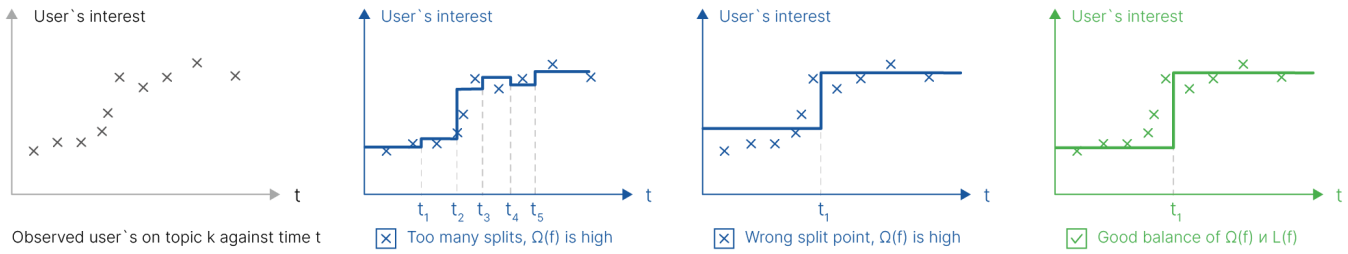
Потому при работе с реальными данными использовать градиентный бустинг из sklearn не стоит. XGBoost и/или LigthGBM дадут результат как правило лучше и быстрее.

XGBoost

Мы хотим спускаться медленно и не хотим, чтобы модели по пути были сложными. Пусть минимально отличающиеся от случайного, но при этом не очень сложными.

Поэтому XGBoost вводит специальный штраф за сложные деревья (большей глубины, чем 2 — 3). За счет того, что в градиентном бустинге можно минимизировать любую дифференцируемую функцию ошибок, мы просто добавляем штраф напрямую в функцию ошибок исходного градиентного бустинга.

Компромисс между функцией потерь и регуляризатором



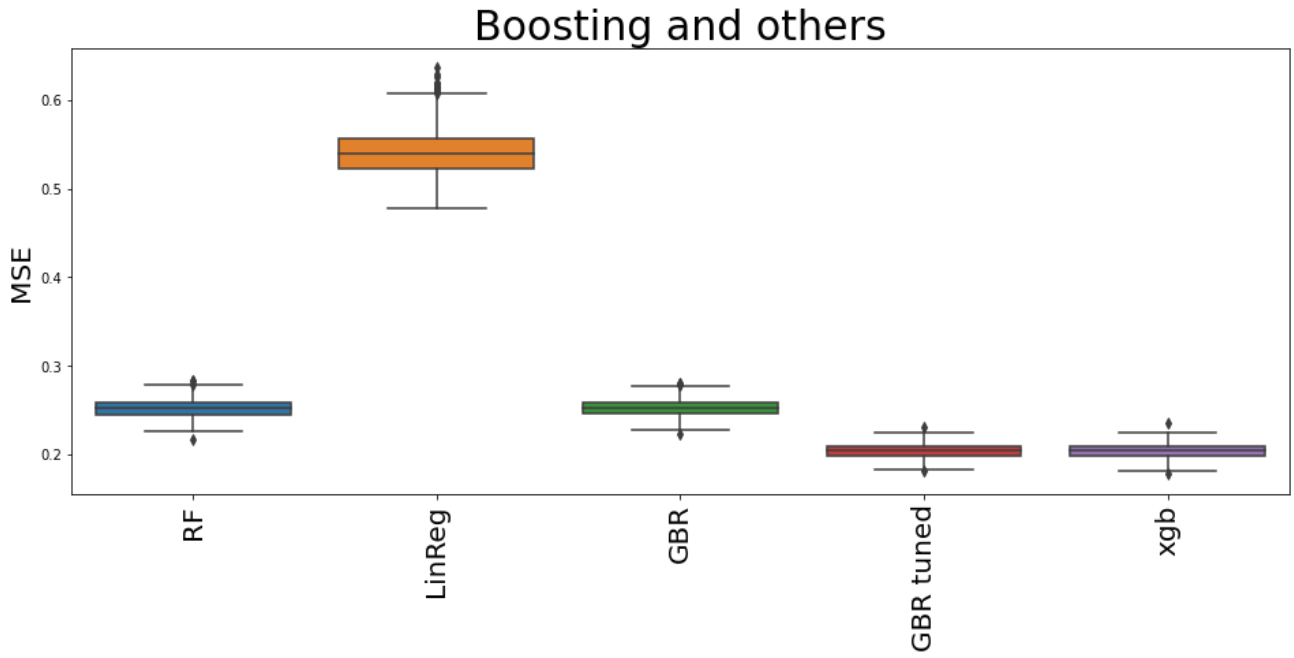

```
calif_housing = sklearn.datasets.fetch_california_housing()
X = calif_housing.data
Y = calif_housing.target
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, random_state=42)
```

Скопируем параметры, которые совпадают между градиентным бустингом и XGBoost. Посмотрим, вдруг получится еще улучшить качество.

```
import xgboost

models_add = {}
models_add['xgb'] = xgboost.XGBRegressor(n_estimators=500,
                     learning_rate=0.1,
                     max_depth=5,
                     random_state=42,
                     min_child_weight=9, # not exact analogue for min_samples_leaf
                     n_jobs=-1, # can be constructed in parrallel, much!!! faster)
                     objective ='reg:squarederror')

# Train models
xgb_add = train_and_test_regressor(models_add, X_train, Y_train, X_test, Y_test)
```

```
Fitting xgb
Calculating bootstrap score for xgb
```

```
plt.figure(figsize=(16,6))
ax = sns.boxplot(data=pd.concat([tuned_boost, xgb_add]) , y="mse", x="model")
plt.setp(ax.get_xticklabels(), rotation=90)
plt.xlabel("", size=20)
plt.ylabel("MSE", size=20)
plt.title("Boosting and others", size=30)
plt.xticks(size=20)
plt.show()
```



Пусть и незначимо, но xgboost с полоборота лучше GBR. И, главное — в разы быстрее.

min_child_weight

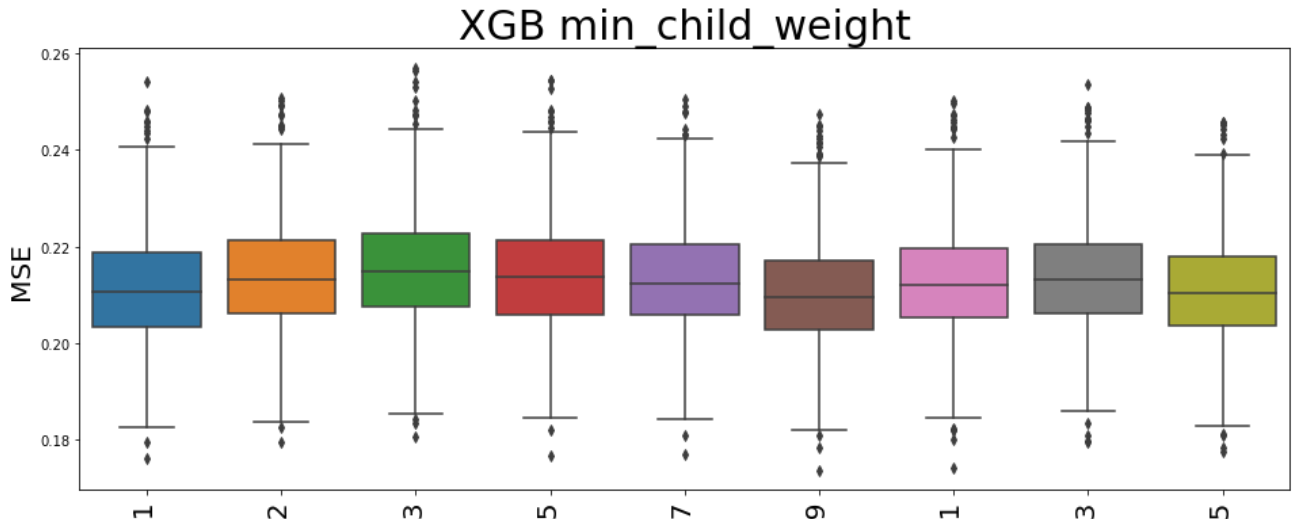
XGBoost немного иначе определяет важность листа — не насколько именно объектов в лист попало, а насколько объекты в листе имеют сильно разные предсказания. Из-за этого способа определения, min_child_weight может принимать нецелые значения, в том числе меньше 1 (к примеру, в случае задачи классификации)

Попробуем затюнить только этот параметр. Заметьте, мы можем спокойно использовать 2000 естиматоров, не боясь ждать результата рассчетов долгое время

```
models = {}
for min_child_weight in (1, 2, 3, 5, 7, 9, 11, 13, 15):
    models[f"XGB_mnw{min_child_weight}"] = xgboost.XGBRegressor(n_estimators=2000,
                        learning_rate=0.1,
                        max_depth=5,
                        random_state=42,
                        min_child_weight=min_child_weight,
                        n_jobs=-1,
                        objective ='reg:squarederror')

# Train models
xgb_mw = train_and_test_regressor(models, X_learn, Y_learn, X_valid, Y_valid)
```

```
Fitting XGB_mnw1
Fitting XGB_mnw2
Fitting XGB_mnw3
Fitting XGB_mnw5
Fitting XGB_mnw7
Fitting XGB_mnw9
Fitting XGB_mnw11
Fitting XGB_mnw13
Fitting XGB_mnw15
Calculating bootstrap score for XGB_mnw1
Calculating bootstrap score for XGB_mnw2
Calculating bootstrap score for XGB_mnw3
Calculating bootstrap score for XGB_mnw5
Calculating bootstrap score for XGB_mnw7
Calculating bootstrap score for XGB_mnw9
Calculating bootstrap score for XGB_mnw11
Calculating bootstrap score for XGB_mnw13
Calculating bootstrap score for XGB_mnw15
```

```
plt.figure(figsize=(16,6))
ax = sns.boxplot(data=xgb_mw, y="mse", x="model")
plt.setp(ax.get_xticklabels(), rotation=90)
plt.xlabel("", size=20)
plt.ylabel("MSE", size=20)
plt.title("XGB min_child_weight", size=30)
plt.xticks(size=20)
plt.show()
```

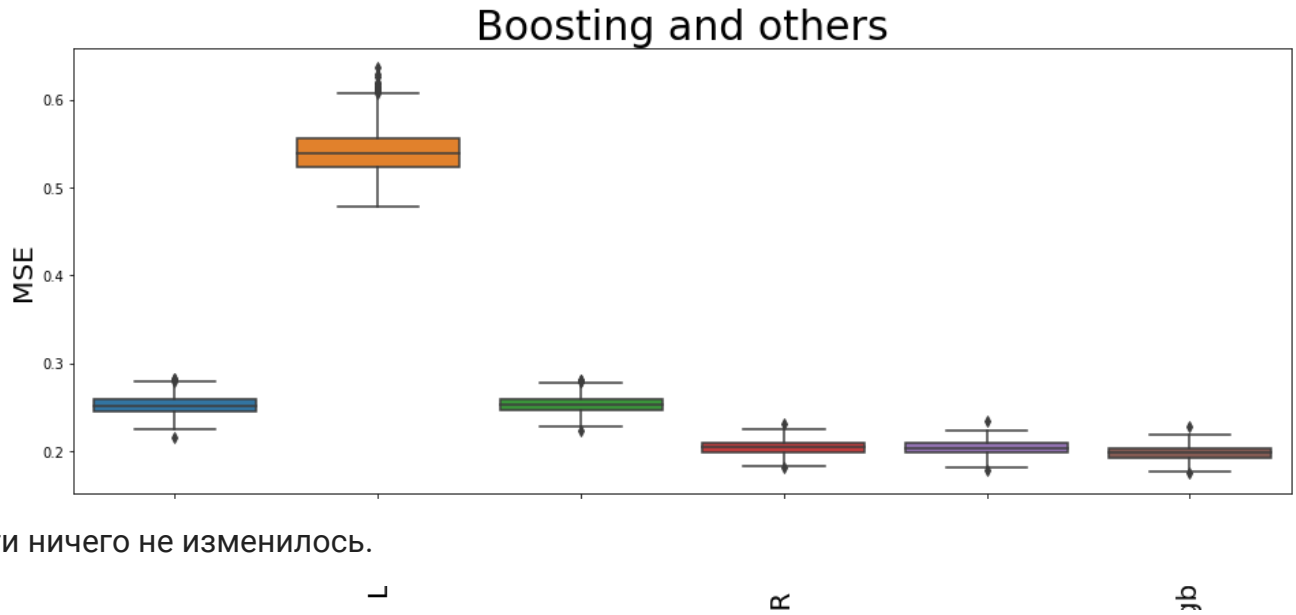


Возьмем равное 9 (в принципе, можно было бы подобрать чуть точнее)

```
models_add2 = {}
models_add2['xgb_mcw'] = xgboost.XGBRegressor(n_estimators=2000,
                     learning_rate=0.1,
                     max_depth=5,
                     random_state=42,
                     min_child_weight=9,
                     n_jobs=-1,
                     objective ='reg:squarederror')

# Train models
xgb_add2 = train_and_test_regressor(models_add2, X_train, Y_train, X_test, Y_test)
```

```
Fitting xgb_mcw
Calculating bootstrap score for xgb_mcw
```

```
plt.figure(figsize=(16,6))
ax = sns.boxplot(data=pd.concat([tuned_boost, xgb_add, xgb_add2]) , y="mse", x="model")
plt.setp(ax.get_xticklabels(), rotation=90)
plt.xlabel("", size=20)
plt.ylabel("MSE", size=20)
plt.title("Boosting and others", size=30)
plt.xticks(size=20)
plt.show()
```



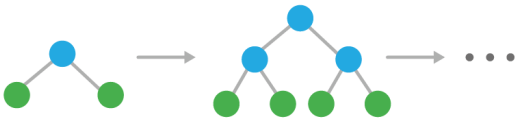
Почти ничего не изменилось.

LightGBM

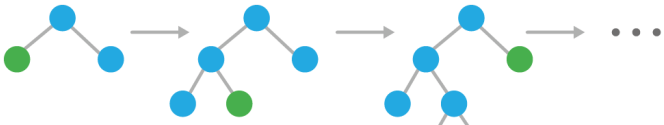
Он был изначально разработан для того, чтобы работать очень быстро. В него добавили много ухищрений, связанных с этим. Кроме этого, LightGBM по-умолчанию строит дерево немного иначе, нежели XGBoost.

XGBoost по-умолчанию строит дерево по уровням - на каждом уровне, если узел можно разбить так, чтобы улучшить значение функции ошибки, то мы это делаем. Ограничены мы только максимальной глубиной дерева.

LightGBM же строит дерево по-узлам. На каждом шаге бьется тот узел, разбиение которого сильнее всего минимизирует функцию ошибки. И ограничения на глубину нет. В LightGBM вводится ограничение не на глубину дерева, а на общее число листьев в итоговом дереве.



Поуровневое построение дерева



Полистовое построение дерева (первый — лучший)

По-умолчанию он **еще** быстрее (хотя в xgboost тоже есть опции для ускорения)

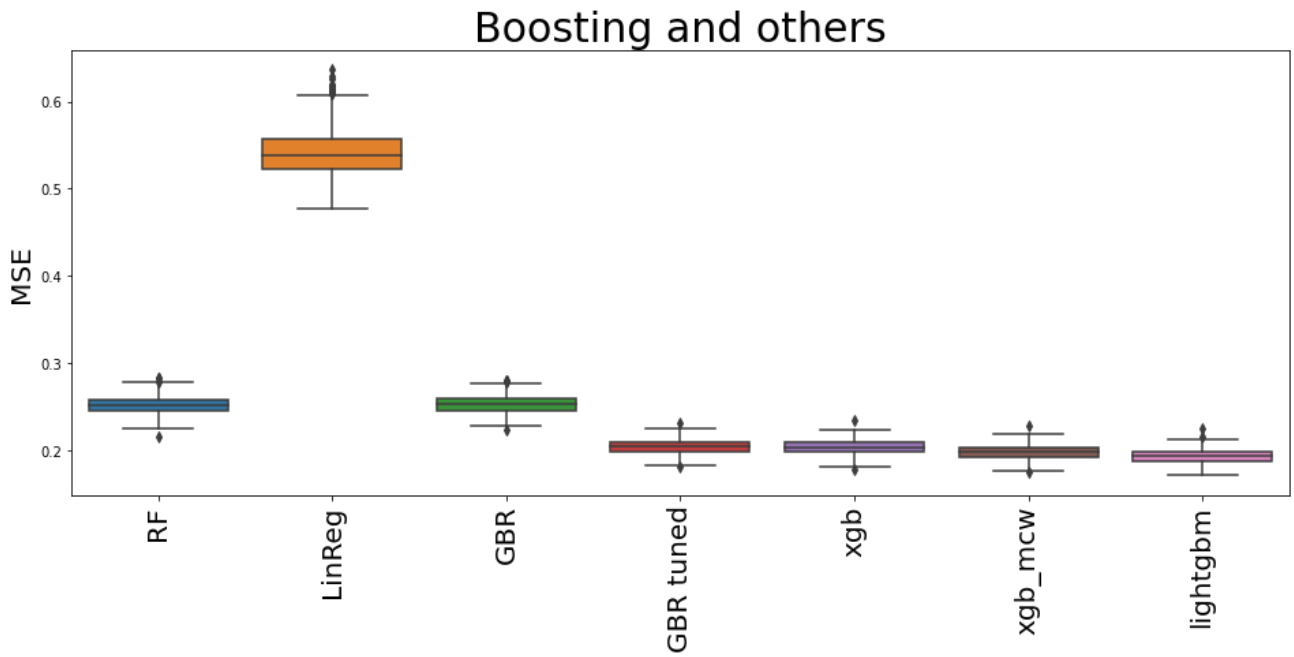
Из-за особенностей построения им деревьев, надо задавать не высоту дерева, а максимальное число листьев. Поставим пока так, чтобы число листьев было равно числу листьев в дереве высоты 6

```
import lightgbm

models_add3 = {}
```

```
models_add3['lightgbm'] = lightgbm.LGBMRegressor(n_estimators=2000, # can use more esti
                      learning_rate=0.1,
                      max_depth=-1,
                      num_leaves=2**5,
                      random_state=42,
                      min_child_weight=9,
                      n_jobs=-1)

# Train models
lgb_add = train_and_test_regressor(models_add3, X_train, Y_train, X_test, Y_test)
```

```
Fitting lightgbm
Calculating bootstrap score for lightgbm
```

```
plt.figure(figsize=(16,6))
ax = sns.boxplot(data=pd.concat([tuned_boost, xgb_add, xgb_add2, lgb_add]) , y="mse", x
plt.setp(ax.get_xticklabels(), rotation=90)
plt.xlabel("", size=20)
plt.ylabel("MSE", size=20)
plt.title("Boosting and others", size=30)
plt.xticks(size=20)
plt.show()
```



На уровне с xgboost. Зато быстрее)

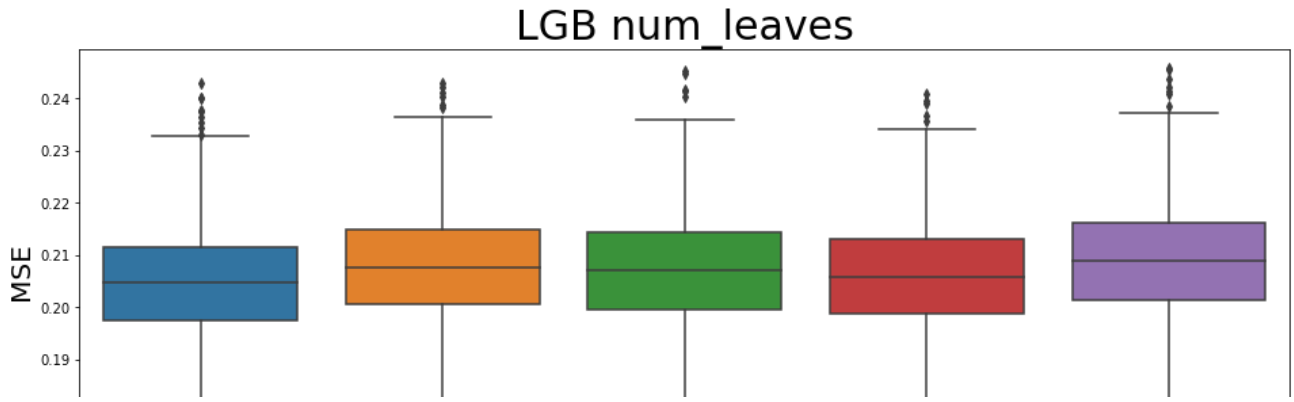
Максимальное число листьев в дереве

Попробуем все же отдельно подобрать ограничение на число листьев. По словам создателей, оно не должно быть больше числа листьев в соответствующем дереве глубины 5, то есть 32.

```
models = {}
for num_leaves in (8, 16, 24, 32, 40):
    models[f"LGB_lvn{num_leaves}"] =  lightgbm.LGBMRegressor(n_estimators=2000,
                        learning_rate=0.1,
                        max_depth=-1,
                        num_leaves=num_leaves,
                        random_state=42,
                        min_child_weight=10,
                        n_jobs=-1)

# Train models
lgb_nl = train_and_test_regressor(models, X_learn, Y_learn, X_valid, Y_valid)
```

```
Fitting LGB_lvn8
Fitting LGB_lvn16
Fitting LGB_lvn24
Fitting LGB_lvn32
Fitting LGB_lvn40
Calculating bootstrap score for LGB_lvn8
Calculating bootstrap score for LGB_lvn16
Calculating bootstrap score for LGB_lvn24
Calculating bootstrap score for LGB_lvn32
Calculating bootstrap score for LGB_lvn40
```

```
plt.figure(figsize=(16,6))
ax = sns.boxplot(data=lgb_nl, y="mse", x="model")
plt.setp(ax.get_xticklabels(), rotation=90)
plt.xlabel("", size=20)
plt.ylabel("MSE", size=20)
plt.title("LGB num_leaves", size=30)
plt.xticks(size=20)
plt.show()
```

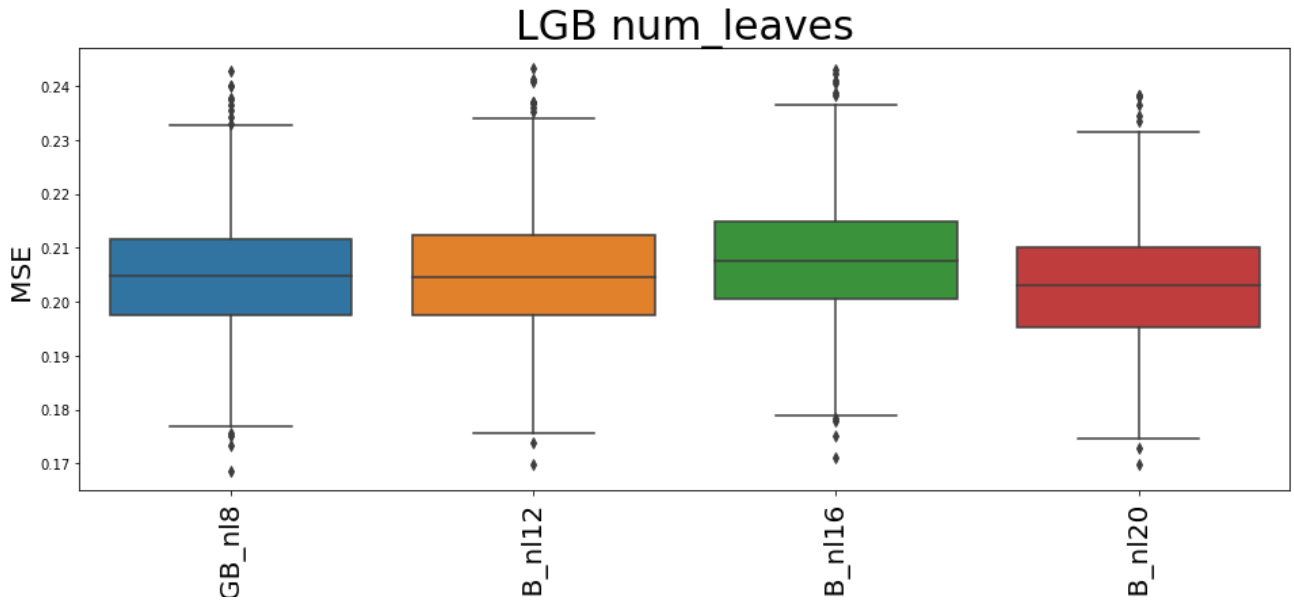


Исходя из этого графика, оптимальное число листьев - в районе 8

```
models = {}
for num_leaves in (8, 12, 16, 20):
    models[f"LGB_nl{num_leaves}"] =  lightgbm.LGBMRegressor(n_estimators=2000,
                        learning_rate=0.1,
                        max_depth=-1,
                        num_leaves=num_leaves,
                        random_state=42,
                        min_child_weight=10,
                        n_jobs=-1)

# Train models
lgb_nl = train_and_test_regressor(models, X_learn, Y_learn, X_valid, Y_valid)
```

```
Fitting LGB_nl8
Fitting LGB_nl12
Fitting LGB_nl16
Fitting LGB_nl20
Calculating bootstrap score for LGB_nl8
Calculating bootstrap score for LGB_nl12
Calculating bootstrap score for LGB_nl16
Calculating bootstrap score for LGB_nl20
```

```
plt.figure(figsize=(16,6))
ax = sns.boxplot(data=lgb_nl, y="mse", x="model")
plt.setp(ax.get_xticklabels(), rotation=90)
plt.xlabel("", size=20)
plt.ylabel("MSE", size=20)
plt.title("LGB num_leaves", size=30)
plt.xticks(size=20)
plt.show()
```

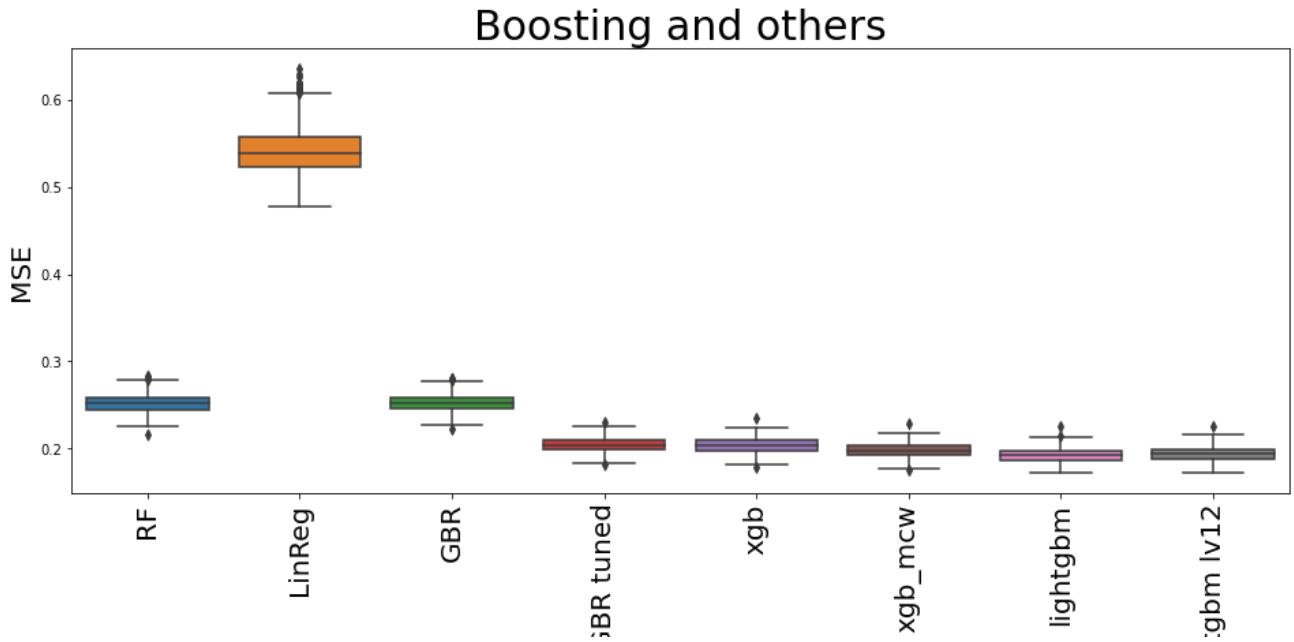


Выберем 12 листьев

```
models_add4 = {}
models_add4['lightgbm lv12'] = lightgbm.LGBMRegressor(n_estimators=2000,
                      learning_rate=0.1,
                      max_depth=-1,
                      num_leaves=12,
                      random_state=42,
                      min_child_weight=9,
                      n_jobs=-1)

# Train models
lgb_add2 = train_and_test_regressor(models_add4, X_train, Y_train, X_test, Y_test)
```

```
Fitting lightgbm lv12
Calculating bootstrap score for lightgbm lv12
```

```
plt.figure(figsize=(16,6))
ax = sns.boxplot(data=pd.concat([tuned_boost, xgb_add, xgb_add2, lgb_add, lgb_add2]) ,
plt.setp(ax.get_xticklabels(), rotation=90)
plt.xlabel("", size=20)
plt.ylabel("MSE", size=20)
plt.title("Boosting and others", size=30)
plt.xticks(size=20)
plt.show()
```



Качество почти не изменилось

CatBoost

Разработал Яндекс.

CatBoost:

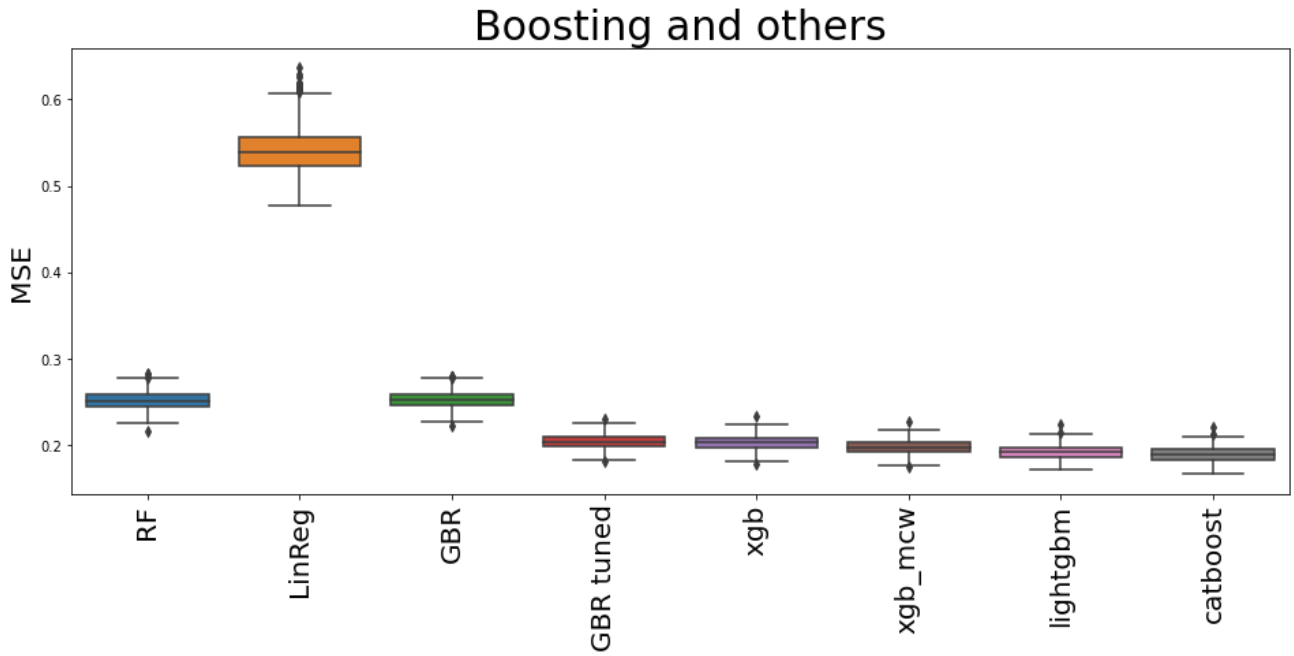
1. хорошо умеет работать с категориальными признаками. Если у вас много категориальных признаков - он может дать сушественный выигрыш.
2. по-умолчанию использует в качестве модели модификацию обычного дерева решения - Symmetric Tree, которое менее склонно к переобучению

```
from IPython.display import clear_output
!pip install catboost
clear_output()
```

```
import catboost

models_add4 = {}
models_add4['catboost'] = catboost.CatBoostRegressor(iterations=2000,
                      learning_rate=0.1,
                      depth=5,
                      random_state=42,
                      min_data_in_leaf=9,
                      verbose=0)
                      #task_type="GPU") # can use gpu, but no parallel-cpu option

# Train models
cat_add = train_and_test_regressor(models_add4, X_train, Y_train, X_test, Y_test)
```

```
Fitting catboost
Calculating bootstrap score for catboost
```

```
plt.figure(figsize=(16,6))
ax = sns.boxplot(data=pd.concat([tuned_boost, xgb_add, xgb_add2, lgb_add, cat_add]) , y
plt.setp(ax.get_xticklabels(), rotation=90)
plt.xlabel("", size=20)
plt.ylabel("MSE", size=20)
plt.title("Boosting and others", size=30)
plt.xticks(size=20)
plt.show()
```



Стало лучше

Про другие реализации случайного леса

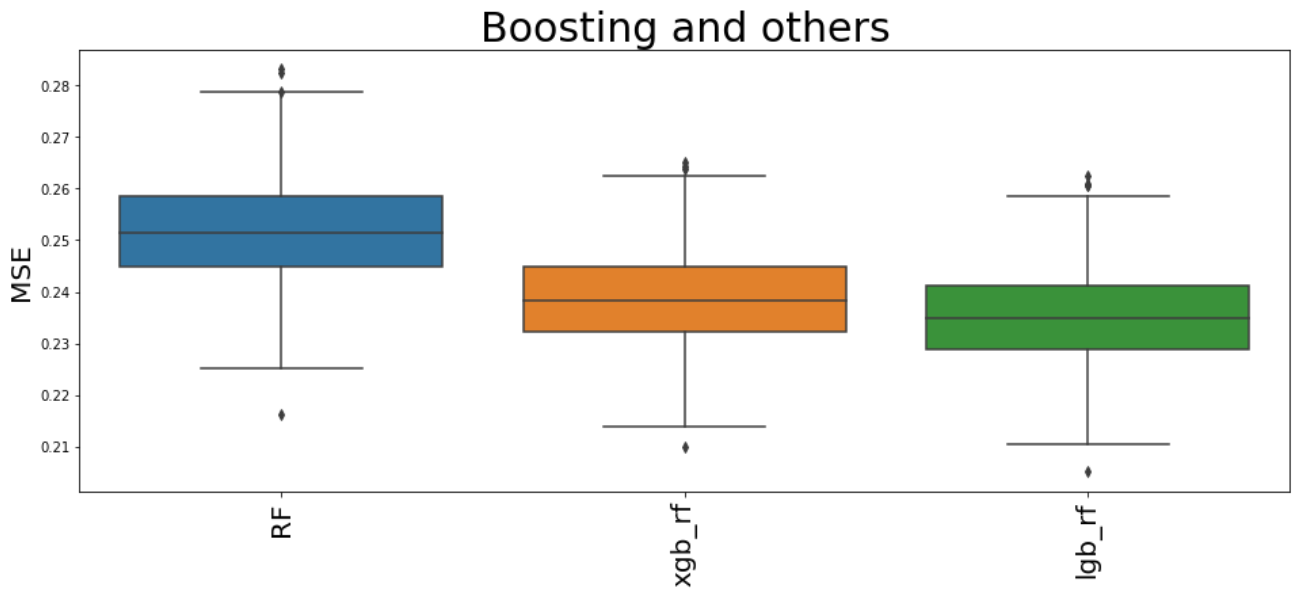
В xgboost и lightgbm есть свои, более быстрые и иногда более эффективные реализации случайного леса. Но их надо дополнительно настраивать.

```
models_rf = {}

models_rf["xgb_rf"] = xgboost.XGBRFRegressor(n_estimators=250,
                                             colsample_bytree=0.8,
                                             subsample=0.8,
                                             reg_lambda=0.001, # to get deeper, less re
                                             max_depth=20, # trees must be deep
                                             n_jobs=-1,
```

```
                                        objective ='reg:squarederror')

models_rf["lgb_rf"] = lightgbm.LGBMRegressor(n_estimators=250,
                            bagging_freq=1, # for lgb random forest must be set to 1
                            num_leaves=2**14, # don't forget to change the number of le
                            boosting_type="rf", # set boosteer type
                            feature_fraction=0.8, # for lgb random forest must be less
                            bagging_fraction=0.8, # for lgb random forest must be less
                            min_child_samples=1, # to get deeper trees
                            n_jobs=-1)

# Train models
rf_add = train_and_test_regressor(models_rf, X_train, Y_train, X_test, Y_test)
```

```
Fitting xgb_rf
Fitting lgb_rf
Calculating bootstrap score for xgb_rf
Calculating bootstrap score for lgb_rf
```

```
plt.figure(figsize=(16,6))
ax = sns.boxplot(data=pd.concat([tuned_boost.query('model == "RF"'), rf_add]) , y="mse"
plt.setp(ax.get_xticklabels(), rotation=90)
plt.xlabel("", size=20)
plt.ylabel("MSE", size=20)
plt.title("Boosting and others", size=30)
plt.xticks(size=20)
plt.show()
```



Заметьте, что в данном случае они показывают себя иначе, нежели sklearn RandomForest. Это частое явление - так как деревья в пакетах xgboost и lightgbm строятся отличным от sklearn образом. Это может как улучшать качество, так и ухудшать

Про подбор параметров модифицированных бустингов

Каждый из модифицированных бустингов имеет свои особые параметры и рекомендации от создателей и участников Kaggle, как и в каком порядке выбирать параметры для конкретного бустинга. Если вы решили использовать тот или иной вид бустинга в своей задаче - ОЗНАКОМЬТЕСЬ с этими советами.

Все три бустинга, к примеру, позволяют задавать не только число признаков, использующихся в каждом узле, но и число признаков, использующихся в каждом дереве в целом.

Кроме того, все три бустинга имеют настройки, ускоряющие их работу (у lightgbm эти настройки выставлены по-умолчанию) и все три бустинга умеют работать на GPU

Здесь мы бегло пробежались по самым верхам, копируя подобранные на предудущих этапах параметры

Блендинг и Стэкинг

Смесь экспертов

До этого мы использовали два подхода. Первый подход — берем много предсказаний большого количества моделей и усредняем (считаем, что модели равноправны).

$$h(x) = \frac{1}{k}\sum_{i=i}^{k} a_i(x)$$

Потом задумываемся о том, что каждая модель предсказывает в целом по-разному. Одна модель ошибается в 10% случаях, другая — в 15% и так далее. Неплохо бы к предсказаниям этих моделей подобрать веса. Так рождается идея того же бустинга.

$$h(x) = \frac{1}{k}\sum_{i=i}^{k} b_i a_i(x)$$

Следующий логичный шаг — подумать о том, что каждая модель может ошибаться на каких-то своих объектах. Например, при предсказании, насколько велика вероятность, что человек серьезно заболеет, одна модель будет плохо работать для европейцев, а другая — для жителей Новой Гвинеи.

$$h(x) = \frac{1}{k}\sum_{i=i}^{k} b_i(x) a_i(x)$$

Получается ситуация, в которой то, насколько хорошо будет предсказывать модель, зависит от самого объекта. Фактически это означает то, что мы бы хотели для каждого объекта получать какой-то вектор весов, а дальше суммировать предсказания моделей a_i, используя эти веса.

Как получать эти веса для каждого объекта? Обучим еще одну модель, которая явно или неявно предсказывает веса алгоритмов и агрегирует их предсказания. Называться она будет **метаалгоритмом**.

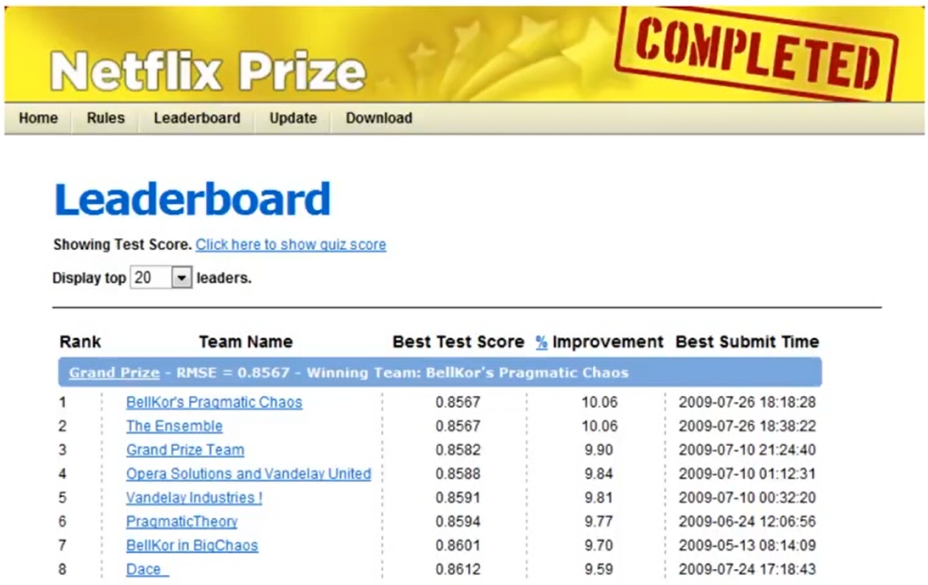
Обычно для метаалгоритма используется сравнительно простые модели, например, линейную модель (в этом случае веса предсказаний базовых моделей не будут зависеть от объекта) или

Как получить такую модель? Есть два распространенных подхода - **stacking** и **blending**

Blending (Блендинг)

На соревновании Netflix Prize была поставлена задача - предсказать, как люди оценят тот или иной фильм. Победил подход, основанный на таком объединении модели, при котором у вас каждая модель получает вес в зависимости от объекта. Подход называется **блендинг**.

Объединение моделей

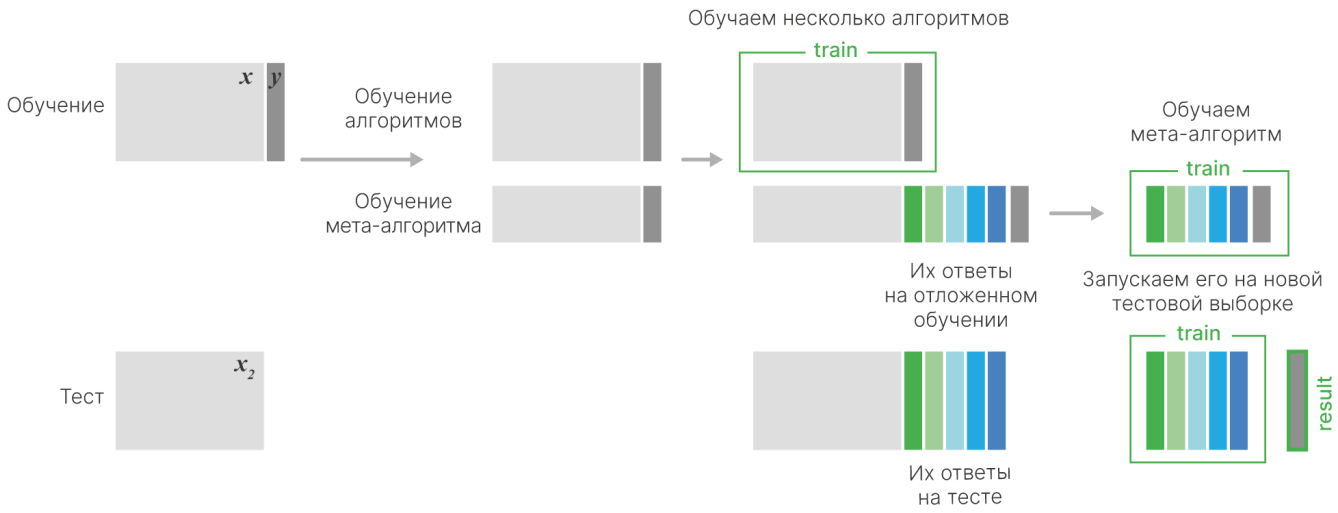


У нас есть **обучающая и тестовая выборка**.

1. Бьем обучающую выборку на две подвыборки - A (побольше) и B
2. Обучаем набор базовых алгоритмов на подвыборке A.

3. Получаем предсказания этих базовых алгоритмов на объектах из выборки B
4. Используем эти предсказания в качестве признаков для метаалгоритма, который будет предсказывать веса алгоритмов. Обучаем его на выборке B.
5. Для того, чтобы сделать предказание на тесте:
    1. Получаем предсказания базовых моделей на тесте
    2. Аггрегируем их при помощи метаалгоритма

Тестовую выборку не трогаем. Дополнительно **разбиваем обучение и обучаем алгоритмы на разбиении**, каждый алгоритм дает какое-то предсказание. На отложенной части обучения делаем то же предсказание, то есть используем предсказания алгоритмов как дополнительные признаки. Получается датасет, в котором есть изначальные признаки и предсказания моделей. Дальше можно **обучить мета-алгоритм либо на объединении всех этих признаков, либо только на этих признаках**. Получится, что алгоритм будет брать предсказания других алгоритмов, обрабатывать их внутри себя и выдавать итоговое предсказание.



Недостатком блендинга является то, что мы дополнительно разбиваем обучающую выборку - нельзя учить и базовые алгоритмы, и основной на одних и тех же данных.

Для улучшения качества можно сделать несколько блендингов (по-разному разбивая обучающую выборку на выборки A и B), а дальше усреднять предсказания разных блендингов.

Попробуем сделать простой блендинг. Единственное,немного схитрим. Для обучения метаалгоритма будем использовать датасет, который использовали при подборе гиперпараметров алгоритмов

```
class BlendingRegressor():
    def __init__(self,
                 first_layer_models,
                 second_layer_model):
```

```
        self.first_layer_models = {name : sklearn.clone(model) for name, model in first
        self.second_layer_model = sklearn.clone(second_layer_model)

    def fit_1st_layer(self, X, Y):
        for name, model in self.first_layer_models.items():
            print(f"Fitting {name}")
            model.fit(X, Y)

    def predict_1st_layer(self, X):
        features = np.zeros((X.shape[0], len(self.first_layer_models)) )
        for ind, model in enumerate(self.first_layer_models.values()):
            features[:, ind] = model.predict(X)
        return features

    def fit_2st_layer(self, X, Y):
        features = self.predict_1st_layer(X)
        self.second_layer_model.fit(features, Y)

    def predict(self, X):
        features = self.predict_1st_layer(X)
        Y_pred = self.second_layer_model.predict(features)
        return Y_pred
```

```
first_layer_models = {}
first_layer_models['linreg'] = make_pipeline(StandardScaler(with_mean=False), LinearReg


first_layer_models['rf'] = RandomForestRegressor(n_estimators=250,
                                  max_depth=None,
                                   min_samples_leaf=1,
                                   n_jobs=-1,
                                  random_state=42)

# too long
#first_layer_models['gbr'] = GradientBoostingRegressor(learning_rate=0.05,
#                                n_estimators=1000,
#                                random_state=42)
#first_layer_models['gbr_tuned'] = GradientBoostingRegressor(learning_rate=0.1,
#                                     n_estimators=2000,
#                                     max_depth=5,
#                                     min_samples_leaf=5,
#                                     random_state=42)


first_layer_models['xgb'] = xgboost.XGBRegressor(n_estimators=500,
                     learning_rate=0.1,
                     max_depth=5,
                     random_state=42,
                     min_child_weight=10,
                     n_jobs=-1,
                     objective ='reg:squarederror')


```

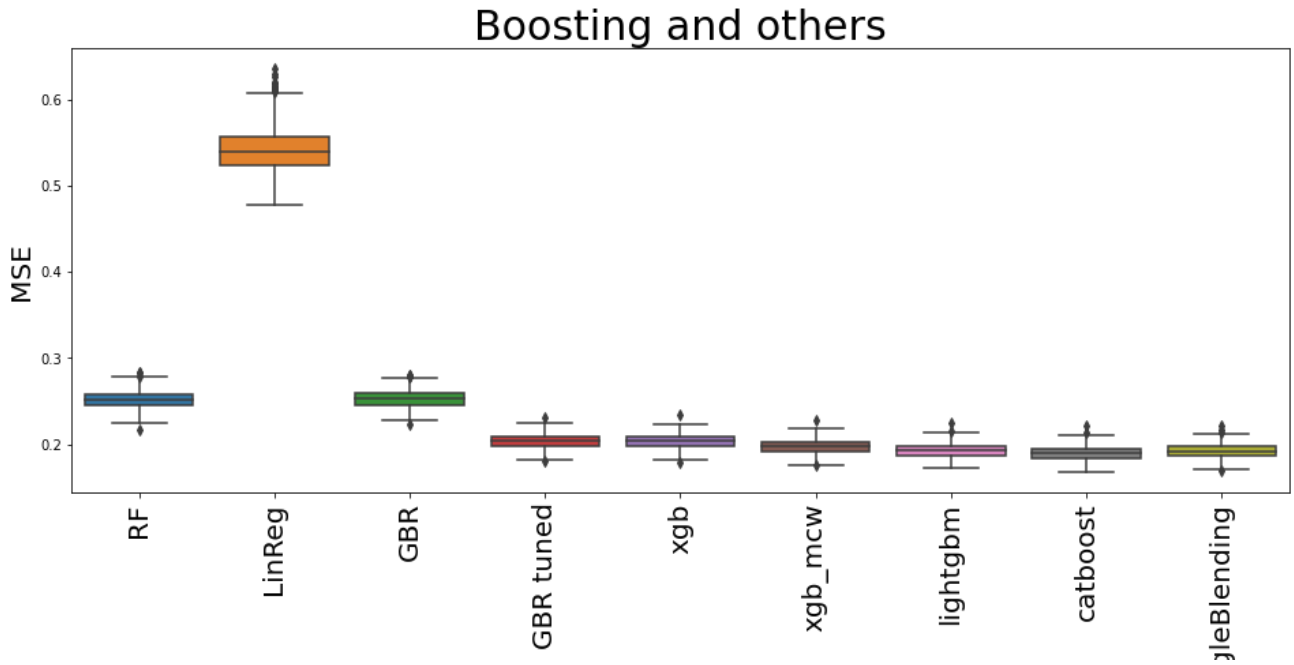
```
first_layer_models['lgb_tuned'] = lightgbm.LGBMRegressor(n_estimators=2000,
                      learning_rate=0.1,
                      max_depth=-1,
                      num_leaves=12,
                      random_state=42,
                      min_child_weight=7,
                      n_jobs=-1)

first_layer_models['catboost'] = catboost.CatBoostRegressor(iterations=2000,
                                                            verbose=0,
                                                            learning_rate=0.1,
                                                            depth=5,
                                                            random_state=42,
                                                            min_data_in_leaf=5)
```

```
X_learn, X_valid, Y_learn, Y_valid = train_test_split(X_train, Y_train, random_state=42
blend_reg = BlendingRegressor(first_layer_models, LinearRegression())

blend_reg.fit_1st_layer(X_learn, Y_learn)
blend_reg.fit_2st_layer(X_valid, Y_valid)
Y_pred = blend_reg.predict(X_test)
blend_boot = bootstrap_metric(Y_test,
                 Y_pred,
                 metric_fn=lambda X, Y: mean_squared_error(y_true=X, y_pred=Y))
```

```
Fitting linreg
Fitting rf
Fitting xgb
Fitting lgb_tuned
Fitting catboost
```

```
blend_data = pd.DataFrame({"mse": blend_boot})
blend_data['model'] = 'SingleBlending'
```

```
plt.figure(figsize=(16, 6))
ax = sns.boxplot(data=pd.concat([tuned_boost, xgb_add, xgb_add2, lgb_add, cat_add, blen
plt.setp(ax.get_xticklabels(), rotation=90)
plt.xlabel("", size=20)
plt.ylabel("MSE", size=20)
plt.title("Boosting and others", size=30)
plt.xticks(size=20)
plt.show()
```



Одиночный блендинг работает даже хуже, чем один catboost

Попробуем обучить 10 блендингов с разными разбиениями X_train на X_learn и X_valid

```
blending_ensemble = []

# takes some time to run
for i in range(1, 11):
    print(f"Training blender {i}")
    X_learn, X_valid, Y_learn, Y_valid = train_test_split(X_train, Y_train, random_stat
    blend_reg = BlendingRegressor(first_layer_models,
                                  LinearRegression())

    blend_reg.fit_1st_layer(X_learn, Y_learn)
    blend_reg.fit_2st_layer(X_valid, Y_valid)
    blending_ensemble.append(blend_reg)
    clear_output()
```

```
Y_pred = 0
for blend_reg in blending_ensemble:
    Y_pred += blend_reg.predict(X_test)
Y_pred /= len(blending_ensemble)


eblend_boot = bootstrap_metric(Y_test,
                 Y_pred,
                 metric_fn=lambda X, Y: mean_squared_error(y_true=X, y_pred=Y))
eblend_data = pd.DataFrame({"mse": eblend_boot})
eblend_data['model'] = 'EnsembleBlending'
```

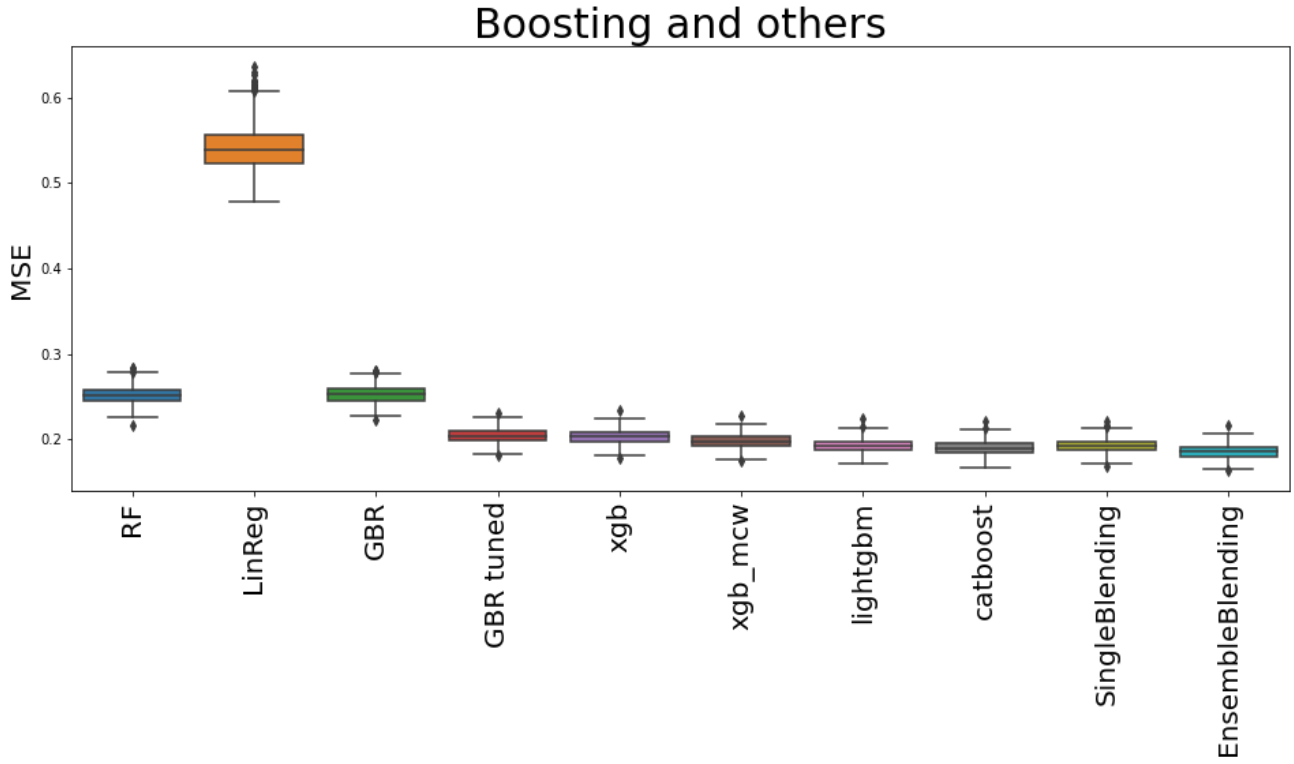
```
np.quantile(eblend_boot, q=[0.025, 0.975])
```

```
array([0.17014789, 0.20072306])
```

```
np.quantile(cat_add['mse'], q=[0.025, 0.975])
```

```
array([0.17445492, 0.20549425])
```

```
plt.figure(figsize=(16,6))
ax = sns.boxplot(data=pd.concat([tuned_boost, xgb_add, xgb_add2, lgb_add, cat_add, blen
plt.setp(ax.get_xticklabels(), rotation=90)
plt.xlabel("", size=20)
plt.ylabel("MSE", size=20)
plt.title("Boosting and others", size=30)
plt.xticks(size=20)
plt.show()
```



Качество по сравнению с лучшей моделью (catboost) существенно не отличается

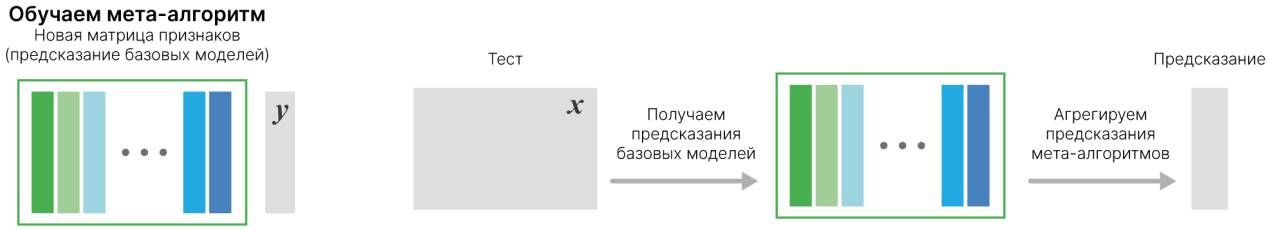
Стэкинг

Так же можно использовать другой способ, который позволяет использовать всю обучающую выборку

Вместо того, чтобы бить обучающую выборку на выборки А и Б случайным образом, давайте использовать кросс-валидацию.

- 1. Для каждого из базовых алгоритмов, которые хотим объединять:
  - 1.1 Бьем обучающую выборку на блоки
  - 1.2 На каждом кросс-валидационном разбиении учим базовый алгоритм. На блоке, который не вошел в обучение делаем предсказание.
  - 1.3. В результате получаем предсказания для всей обучающей выборки.
  - 1.4 Для предсказания на тестовой выборке, обучаем наш базовый алгоритм на всей выборке
- 2. Обучаем на полученных предсказаниях мета-алгоритм

- 3. Для того, чтобы сделать предсказание на тесте, делаем предсказания моделями, полученными на шаге 1.4. И аггрегируем их предсказания при помощи мета-алгоритма, полученного на шаге 2



Стэкинг, в отличие от блендинга, использует все обучение - мы не теряем часть объектов безвозвратно. Но этим мы вносим несколько проблем, одной из которых является то, что модели, на предсказаниях которых мы учили мета-алгоритм и модели, предсказаниях

которых мы в итоге им агрегируем, немного отличаются. Если в одном случае вероятность 0.2, предсказанная моделью, значила: «я не совсем уверена в предсказании», то теперь это может значить «я очень уверена в предсказании».

Кроме того, по наблюдениям, такой подход более склонен к переобучению. Для борьбы с ним можно, к примеру, подмешивать к предсказаниям базовых моделей шум.

Так же, по причине того, что нам надо делать предсказание на кроссвалидации, и работать этот метод будет дольше Blending. Зато иногда он работает лучше

В sklearn уже реализован свой StackingRegressor, потому в этой лекции мы его реализовывать не будем.

Можете посмотреть пример реализации, например, здесь. В принципе, если вам действительно понадобится стэкинг, полезно иметь свою собственную реализацию, чтобы легко менять в ней некоторые детали (вариантов стекинга - великое множество)

Возьмем в нашу новую модель только быстро учащиеся модели

```
base_models = []

base_models.append(["linreg", make_pipeline(StandardScaler(with_mean=False), LinearRegr

base_models.append(["xgb", xgboost.XGBRegressor(n_estimators=500,
                      learning_rate=0.1,
                      max_depth=5,
                      random_state=42,
                      min_child_weight=10,
                      n_jobs=-1,
                      objective ='reg:squarederror')])

base_models.append(["lgb", lightgbm.LGBMRegressor(n_estimators=2000,
                      learning_rate=0.1,
                      max_depth=-1,
                      num_leaves=12,
                      random_state=42,
                      min_child_weight=7,
                      n_jobs=-1)])

base_models.append(["cgb", catboost.CatBoostRegressor(iterations=2000,
                                                       verbose=0,
                                                       learning_rate=0.1,
                                                       depth=5,
                                                       random_state=42,
                                                       min_data_in_leaf=5)])
```

```
from sklearn.ensemble import StackingRegressor

stacking_reg = StackingRegressor(base_models,
                    LinearRegression(), # level-two model is Linear Regression
```

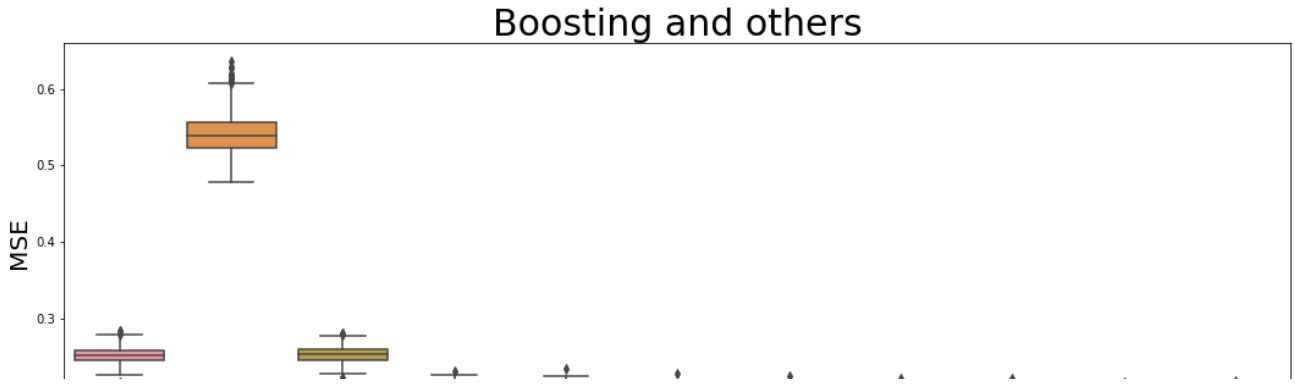
```
                    cv=3)

stacking_reg.fit(X_train, Y_train)
clear_output()
```

```
Y_pred = stacking_reg.predict(X_test)
print(mean_squared_error(y_true=Y_test, y_pred=Y_pred))
```

```
0.18633737632004774
```

```
stack_boot = bootstrap_metric(Y_test,
                 Y_pred,
                 metric_fn=lambda X, Y: mean_squared_error(y_true=X, y_pred=Y))

stack_data = pd.DataFrame({"mse": stack_boot})
stack_data['model'] = 'Stacking'
```

```
plt.figure(figsize=(18,6))
ax = sns.boxplot(data=pd.concat([tuned_boost,
                                 xgb_add,
                                 xgb_add2,
                                 lgb_add,
                                 cat_add,
                                 blend_data,
                                 eblend_data,
                                 stack_data]) , y="mse", x="model")
plt.setp(ax.get_xticklabels(), rotation=90)
plt.xlabel("", size=20)
plt.ylabel("MSE", size=20)
plt.title("Boosting and others", size=30)
plt.xticks(size=20)
plt.show()
```




```
np.quantile(stack_data['mse'],
            q=[0.025, 0.975])
```

```
array([0.17123641, 0.20178981])
```

```
np.quantile(cat_add['mse'],
            q=[0.025, 0.975])
```

```
array([0.17445492, 0.20549425])
```

Опять же, видим, что стэкинг на данной задаче не принес существенного улучшения

Стэкинг vs блендинг

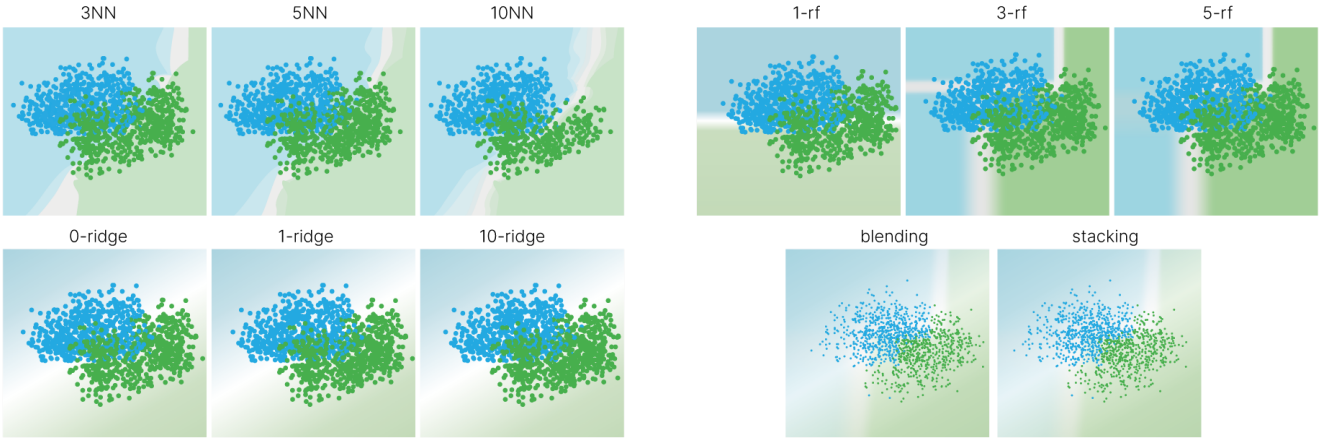
На самом деле, подходы часто объединяют в один или путают местами. Например, улучшая блендинг в сторону использования всей обучающей выборки можно получить стэкинг. И т.д.

Оба подхода имеют множество модификаций, к примеру, с добавлением к признаков-предсказаний базовых моделей изначальных признаков и тд. Оба подхода показывают схожие результаты. И оба подхода улучшают свое качество по мере увеличения размера обучающей выборки.

Пример того, как выглядит решающая граница для стэкинга и блендинга

Задача — разделить синее и красное облако точек. На изображении выше показано, как разные алгоритмы справляются с этим — есть K-Nearest Neighbors, ridge регрессия и случайный лес с разной глубиной деревьев. Все три алгоритма как-то справляются, но лучшие результаты показывает K-Nearest Neighbors. Ridge регрессия работает не очень хорошо потому, что разделение не линейное. Случайный лес угадал тенденцию, но ему либо недостаточно данных, либо надо добавить ещё деревья, либо поменять настройки параметров.

Но если мы берем блендинг и стэкинг этих моделей, то оба этих подхода проводят хорошую разделяющую границу (намного лучше, чем эти подходы по отдельности).



Известно, что нейронные сети хорошо работают с изображениями, зачастую поступают следующим образом: учат много разных нейросетей на изображениях, либо берут уже обученные архитектуры, затем делают с ними блендинг или стэкинг на какой-то отложенной выборке. Результаты получаются лучше, чем от использования одной только нейросети

Применение нейронных сетей к табличным данным

Есть попытки построить архитектуры нейронных сетей таким образом, чтобы они могли работать с табличными данными. Из недавних можно отметить архитектуру от Яндекса NODE и TabNet(TabNet git) от Гугл .

Вторая работа особенно удачна и уже активно используется в соревнованиях на Kaggle. Стоит отметить, что победы над древесными моделями не одержано. Но полученная модель иногда достигает их качества и обладает рядом интересных особенностей:

1. Как и любая другая нейросетевая архитектура - она дифференцируема. Можно вставить обученную нейросеть как часть другой и использовать transfer learning и прочее
2. Нейросеть автоматически выбирает признаки, потому feature importance можно получить без дополнительных ухищрений.
3. Ее можно обучать в semisupervised манере - сначала предобучить на неразмеченных данных, а потом уже использовать размеченные.

Установим пакет, реализующий эту нейросеть на pytorch

Так как это нейросеть, возможно вам захочется сначала поменять среду исполнения на среду с GPU (если поменять после установленные пакеты в Colab слетят)

```
from IPython.display import clear_output
```

```
!pip install pytorch-tabnet
clear_output()
```

```
import sklearn
import sklearn.datasets
from sklearn.model_selection import train_test_split

calif_housing = sklearn.datasets.fetch_california_housing()
X = calif_housing.data
Y = calif_housing.target
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, random_state=42)
X_learn, X_valid, Y_learn, Y_valid = train_test_split(X_train, Y_train, random_state=42
```

Запустим с параметрами по-умолчанию, передав датасет для валидации - нам нужно узнать, когда нейросеть начнет переобучаться.

Заметьте, что у xgboost и lightgbm тоже есть такая возможность (early_stopping), которая позволяет легко подбирать число базовых моделей. Подробнее - смотрите в документации.

Нейросеть принимает y в формате (предсказываемая величина, число предсказываемых величин), потому нам потребуется reshape.

```
from pytorch_tabnet.tab_model import  TabNetRegressor

tabnet = TabNetRegressor()
tabnet.fit(X_learn, Y_learn.reshape(-1, 1),
           eval_set = [(X_valid,  Y_valid.reshape(-1, 1))])
```

```
Device used : cuda
epoch 0  | loss: 3.53212 | val_0_mse: 2632.85901|  0:00:02s
epoch 1  | loss: 0.81726 | val_0_mse: 184.80552|  0:00:04s
epoch 2  | loss: 0.59724 | val_0_mse: 474.33028|  0:00:06s
epoch 3  | loss: 0.49879 | val_0_mse: 214.47596|  0:00:08s
epoch 4  | loss: 0.45785 | val_0_mse: 205.5777|  0:00:10s
epoch 5  | loss: 0.46268 | val_0_mse: 233.87317|  0:00:12s
epoch 6  | loss: 0.43407 | val_0_mse: 41.46188|  0:00:14s
epoch 7  | loss: 0.42183 | val_0_mse: 27.25452|  0:00:16s
epoch 8  | loss: 0.40557 | val_0_mse: 5.60085 |  0:00:18s
epoch 9  | loss: 0.39969 | val_0_mse: 7.62274 |  0:00:19s
epoch 10 | loss: 0.39477 | val_0_mse: 12.18945|  0:00:21s
epoch 11 | loss: 0.38229 | val_0_mse: 31.54496|  0:00:23s
epoch 12 | loss: 0.37447 | val_0_mse: 41.88723|  0:00:24s
epoch 13 | loss: 0.38751 | val_0_mse: 44.58281|  0:00:26s
epoch 14 | loss: 0.37774 | val_0_mse: 41.64264|  0:00:27s
epoch 15 | loss: 0.37614 | val_0_mse: 35.83064|  0:00:28s
epoch 16 | loss: 0.36313 | val_0_mse: 18.78655|  0:00:29s
epoch 17 | loss: 0.35879 | val_0_mse: 24.43299|  0:00:30s
epoch 18 | loss: 0.36055 | val_0_mse: 11.77047|  0:00:31s

Early stopping occurred at epoch 18 with best_epoch = 8 and best_val_0_mse = 5.60085
Best weights from best epoch are automatically used!
```

Обучим теперь на всем тренировочном датасете, поставив число эпох обучения равным тому, что нам сообщила выдача функции

```
tabnet = TabNetRegressor()
tabnet.fit(X_train, Y_train.reshape(-1, 1), max_epochs=53)
```

```
Device used : cuda
No early stopping will be performed, last training weights will be used.
epoch 0  | loss: 2.8148  |  0:00:01s
epoch 1  | loss: 0.76103 |  0:00:02s
epoch 2  | loss: 0.57694 |  0:00:03s
epoch 3  | loss: 0.51798 |  0:00:05s
epoch 4  | loss: 0.49104 |  0:00:07s
epoch 5  | loss: 0.45779 |  0:00:09s
epoch 6  | loss: 0.44154 |  0:00:11s
epoch 7  | loss: 0.4214  |  0:00:14s
epoch 8  | loss: 0.40573 |  0:00:17s
epoch 9  | loss: 0.40465 |  0:00:19s
epoch 10 | loss: 0.37902 |  0:00:21s
epoch 11 | loss: 0.38174 |  0:00:24s
epoch 12 | loss: 0.37445 |  0:00:26s
epoch 13 | loss: 0.38446 |  0:00:28s
epoch 14 | loss: 0.35961 |  0:00:31s
epoch 15 | loss: 0.35876 |  0:00:33s
epoch 16 | loss: 0.35983 |  0:00:36s
epoch 17 | loss: 0.3608  |  0:00:38s
epoch 18 | loss: 0.35865 |  0:00:40s
epoch 19 | loss: 0.36119 |  0:00:42s
epoch 20 | loss: 0.34588 |  0:00:44s
epoch 21 | loss: 0.33849 |  0:00:46s
epoch 22 | loss: 0.33432 |  0:00:48s
epoch 23 | loss: 0.32513 |  0:00:49s
epoch 24 | loss: 0.32134 |  0:00:51s
epoch 25 | loss: 0.32663 |  0:00:52s
epoch 26 | loss: 0.33068 |  0:00:53s
epoch 27 | loss: 0.32652 |  0:00:54s
epoch 28 | loss: 0.32386 |  0:00:55s
epoch 29 | loss: 0.32946 |  0:00:56s
epoch 30 | loss: 0.32495 |  0:00:57s
epoch 31 | loss: 0.3515  |  0:00:58s
epoch 32 | loss: 0.34008 |  0:00:59s
epoch 33 | loss: 0.32808 |  0:01:00s
epoch 34 | loss: 0.32982 |  0:01:01s
epoch 35 | loss: 0.34592 |  0:01:02s
epoch 36 | loss: 0.32083 |  0:01:03s
epoch 37 | loss: 0.31775 |  0:01:04s
epoch 38 | loss: 0.32159 |  0:01:05s
epoch 39 | loss: 0.31683 |  0:01:06s
epoch 40 | loss: 0.32536 |  0:01:07s
epoch 41 | loss: 0.3333  |  0:01:08s
epoch 42 | loss: 0.32961 |  0:01:09s
epoch 43 | loss: 0.32106 |  0:01:10s
epoch 44 | loss: 0.32337 |  0:01:11s
epoch 45 | loss: 0.32567 |  0:01:12s
epoch 46 | loss: 0.31576 |  0:01:13s
epoch 47 | loss: 0.32683 |  0:01:14s
epoch 48 | loss: 0.32691 |  0:01:15s
```

```
epoch 49 | loss: 0.32906 |  0:01:16s
epoch 50 | loss: 0.3236  |  0:01:17s
epoch 51 | loss: 0.3114  |  0:01:18s
epoch 52 | loss: 0.31451 |  0:01:19s
```

```
from sklearn.metrics import mean_squared_error

Y_pred = tabnet.predict(X_test)
print(mean_squared_error(y_true=Y_test, y_pred=Y_pred))
```

```
0.4466265954090426
```

Видим, что качество сильно хуже, чем у топ-моделей бустинга. Возможно, надо попробовать не параметры по-умолчанию нейросети, или нейронную сеть другой архитектуры!

Однако пока создатели не предоставили точных рекомендаций по подбору этих параметров. Можете попробовать улучшить качество модели для интереса.

По сообщениям участников Kaggle, модель все же чаще всего уступает хорошо затюненным бустингам(

Ссылки на материалы для самостоятельного изучения

Общие источники

Про разделение на классическое и глубокое машинное обучение

Один из лучших учебников по классическому машинному обучению

Хорошая книга по машинному обучению - можно скачать через http://libgen.rs

Про деревья решений

Деревья решений

Про ансамбли

https://mlwave.com/kaggle-ensembling-guide/

https://www.analyticsvidhya.com/blog/2018/06/comprehensive-guide-for-ensemble-models/

Цикл лекций по обучению ансамблей

Про стэкинг и блендинг

XGBoost

отличие XGBoost от обычного градиентного бустинга

оригинальная статья

как подбирать параметры xgboost

CatBoost

Лекции по правильному использованию CatBoost

Официальная документация

LightGBM

Очень подробная и удобная документация

Описание параметров

Новый бустинг с деревьями, содержащими в листах линейные регрессии

Дисбаланс классов

Обучение в случае дисбаланса классов

Bagging и случайные леса для обучения с имбалансом классов

Коэффициент корреляции Мэтьюса

Нейронные сети и бустинг

TabNet

TabNet, реализация на pytorch

NODE

0 сек. выполнено в 15:40