Ансамблі моделей

Припустимо, що складне питання задається тисячам випадкових людей і потім їх відповіді агрегуються. За деяких умов виявляється, що така агрегована відповідь є кращою, ніж відповідь експерта. Це називається *колективним розумом (wisdom of the crowd)*. Аналогічно, якщо агрегуються прогнози групи прогнозаторів, таких як класифікатори або регресори, то за певних умов можна отримати кращий агрегований прогноз, ніж прогноз від найкращого індивідуального прогнозатора. Група прогнозаторів називається *ансамблем (ensemble)*; а методи носять назву методів *ансамблевого навчання (ensemble learning)*.

Наприклад, можна навчати групу класифікаторів на основі дерев прийняття рішень, коли кожне дерево навчається на різному піднаборі – випадковому піднаборі навчального набору. В цьому випадку для формування прогнозів беруться прогнози всіх індивідуальних дерев і агрегованим прогнозом є клас, який став володарем більшості голосів. Такий ансамбль дерев прийняття рішень називається випадковим лісом (random forest) і, незважаючи на свою простоту, є одним з найбільш потужних алгоритмів машинного навчання на сьогоднішній день.

Часто ансамблеві методи використовуються, коли декілька гарних прогнозаторів вже побудовано, для їх об'єднання в ще кращий прогнозатор. В подальшому розглянемо найбільш популярні ансамблеві методи, в тому числі беггінг (bagging), бустінг (boosting), стекінг (stacking), випадкові ліси та декілька інших.

Класифікатори з голосуванням

Припустимо, що проведено навчання декількох класифікаторів, і кожен з них забезпечує правильність близько 80%. Цими класифікаторами можуть бути логістична регресія, класифікатори SVM, випадковий ліс, k найближчих сусідів та інші.

Простий спосіб створення ще кращого класифікатора передбачає агрегування прогнозів всіх класифікаторів і прогнозування класу, який отримує найбільшу кількість голосів. Такий мажоритарний класифікатор називається класифікатором *з жорстким голосуванням (hard voting classifier)*.

Виявляється, що даний класифікатор з голосуванням часто досягає більшої правильності, ніж найкращий класифікатор в ансамблі. Припустимо, що кожний класифікатор є *слабким учнем (weak learner)*, тобто він є лише трохи кращим за випадкове вгадування. В цьому випадку ансамбль може бути *сильним учнем (strong learner)*, забезпечувати високу правильність, за умови, що є достатня кількість слабких учнів і вони досить різноманітні.

Для обгрунтування цього розглянемо закон великих чисел та наступний модельний приклад. Розглянемо злегка несиметричну монету, яка у 51% випадків падає на лицьову сторону (орел) і у 49% випадків - на зворотну сторону (решка). Відомо, що ймовірність отримання більшості орлів після 1000 кидків близька до 75% . При 10 000 кидків ця імовірність вже перевищує 97%. Тобто, чим більше разів ми кидаємо монету, тим вищою буде вказана ймовірність. Таким чином, продовжуючи кидати монету, отримаємо що частка випадання орлів стає все ближче і ближче до ймовірності орлів (51%).

Аналогічно припустимо, що будується ансамбль, який містить 1000 класифікаторів, які окремо правильні тільки у 51% випадків, тобто тільки трохи кращі за випадкове вгадування. Якщо ми прогнозуємо мажоритарний клас, то можемо сподіватися на правильність до 75%! Однак це справедливо, тільки якщо всі класифікатори повністю незалежні. А у нас цього немає, тому що класифікатори навчалися на однакових даних.

Ансамблеві методи працюють краще, коли індивідуальні прогнозатори є якомога більш незалежними один від одного. Один із способів отримати несхожі класифікатори полягає в тому, щоб навчати їх із застосуванням дуже різних алгоритмів. Тоді більш імовірно, що вони будуть робити помилки різних типів, що, в свою чергу, підвищує правильність ансамблю.

На практиці, при використанні бібліотеки Scikit-Learn, якщо всі класифікатори мають метод predict_proba() і можуть оцінювати ймовірності класів, то можна виконати *м'яке голосування (soft voting)* – це прогнозування класу з найвищою імовірністю класу, усередненої за всіма індивідуальними класифікаторами. М'яке голосування часто буває більш ефективним, ніж жорстке, тому що надає більшої ваги голосам з високою довірою. Проте, треба слідкувати за тим, щоб класифікатор міг оцінювати ймовірності класів. Наприклад, в класі SVC по умовчанню такої можливості немає. Тому потрібно буде встановити його гіперпараметр probability рівним True. Це змусить клас SVC застосовувати перехресну перевірку для оцінки ймовірностей класів та додати метод predict_proba ().

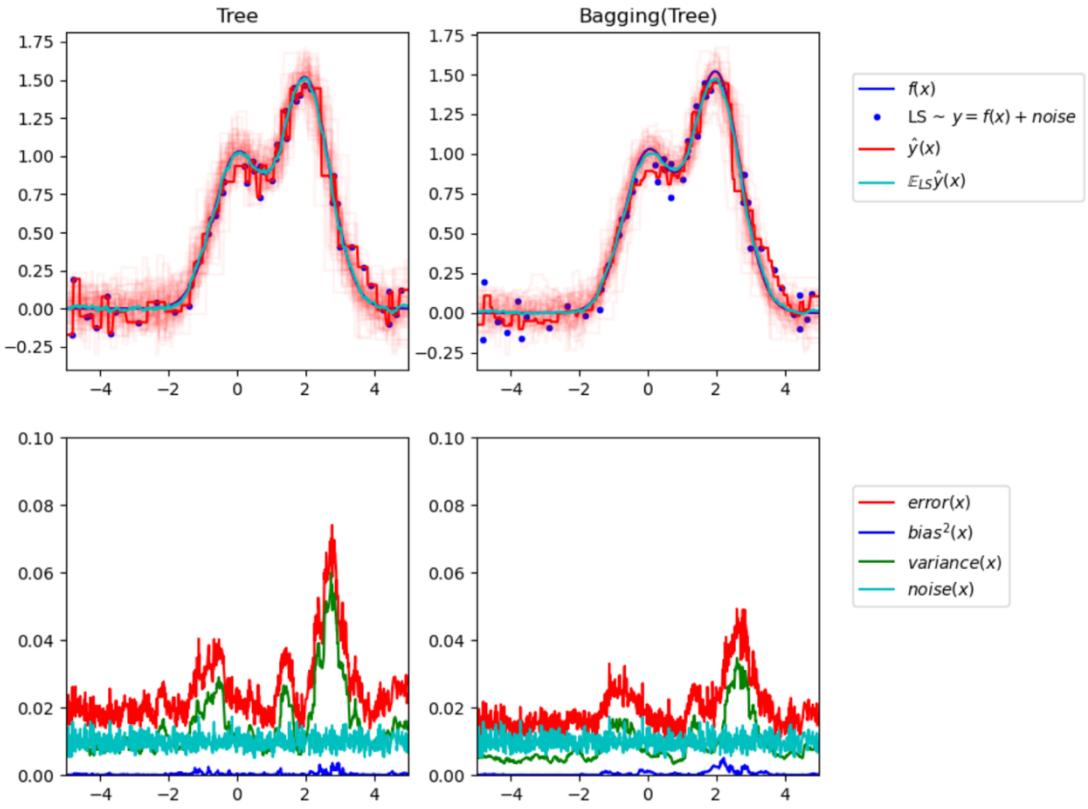
Беггінг і вставка

Один із способів отримати набори несхожих класифікаторів полягає у застосуванні дуже різних алгоритмів навчання. Інший підхід – *беггінг* (*bagging* – скорочення від bootstrap aggregating (бутстреп-агрегування)) передбачає використання для кожного прогнозатора одного і того ж алгоритму навчання, але навчання прогнозатора виконується на різних випадкових піднаборах навчального набору, причому *вибірка піднаборів* виконується *із заміною*.

Коли вибірка виконується без заміни, такий метод називається вставкою або вклеюванням (pasting). Таким чином, і беггінг, і вставка дозволяють множині прогнозаторів формувати піднабори навчальних прикладів по кілька разів. Але тільки беггінг дозволяє здійснювати вибірку навчальних прикладів по кілька разів одним і тим же прогнозатором.

Припустимо, що виконано навчання всіх індивідуальних моделей і тепер потрібно зробити прогноз. *Прогноз* для нового прикладу виробляється ансамблем шляхом агрегування прогнозів від усіх прогнозаторів. Функція агрегування – це зазвичай *мода* для класифікації (тобто прогноз, який найчастіше має місце) або *середнє значення* для регресії. Кожний індивідуальний прогнозатор має більш високе зміщення, ніж якби він навчався на початковому навчальному наборі. Але агрегування скорочує дисперсію. На практиці спостерігається такий результат, що ансамбль має схоже зміщення, але меншу дисперсію, ніж одиночний прогнозатор, який навчений на початковому навчальному наборі.

Це можна проілюструвати на такому рисунку (див.документацію scikit-learn):




```
Tree: 0.0255 (error) = 0.0003 (bias^2)  + 0.0152 (var) + 0.0098 (noise)

Bagging(Tree): 0.0196 (error) = 0.0004 (bias^2)  + 0.0092 (var) +
0.0098 (noise)
```

Ансамбль `Bagging(Tree)` має схоже зміщення, рівне 0.0004, але меншу дисперсію, рівну 0.0092, ніж одиночний прогнозатор `Tree`.

Тобто, ансамбль допускає приблизно таку саму кількість помилок на навчальному наборі, але межа рішень decision boundary характеризується меншою нерегулярністю.

Всі прогнозатори на основі беггінгу можуть навчатися паралельно, навіть на різних серверах. Аналогічно паралельно можуть вироблятися і прогнози. Тому беггінг і вставка є такими популярними методами: вони *дуже добре масштабуються*.

Розглянемо як беггінг і вставка працюють в *Scikit-Learn*.
Є два класи в sklearn.ensemble:

- BaggingClassifier для класифікації та
- BaggingRegressor для регресії.

Наприклад,

```
model = BaggingClassifier ( DecisionTreeClassifier ( ) , n_estimators=500, max_samples=100,
bootstrap=True, n_jobs=-1) .
```

В цьому коді навчили ансамбль з 500 класифікаторів на основі дерев рішень. Кожне дерево навчалося на 100 прикладах, випадково вибраних з навчального набору із заміною (це беггінг). Якщо ми хочемо застосувати вставку, тоді треба задати параметр bootstrap = False. Параметр n_jobs повідомляє про кількість процесорних ядер для використання при навчанні та прогнозуванні. Якщо n_jobs = - 1, то це говорить про участь всіх доступних ядер.

Клас BaggingClassifier автоматично виконує м'яке голосування замість жорсткого голосування, тільки за умови що базовий классифікатор може оцінювати ймовірності класів, тобто якщо він має метод predict proba (). Класифікатор на основі дерев прийняття рішень має метод predict proba ().

Порівняємо беггінг і вставку. Бегінг вносить трохи більше розходження в піднабори, на яких навчається кожен прогнозатор, і тому беггінг в результаті дає трохи більший зсув, ніж вставка. Але це також означає, що прогнозатори будуть менш залежними один від одного, та дисперсія ансамблю буде меншою. В цілому вважається, що моделі на основі беггінга є кращими, і часто використовують саме їх. Проте, розв'язуючи конкретну практичну задачу, можна застосувати перехресну перевірку для оцінки беггінга і вставки і вибрати підхід, який для даної задачі працює краще.

Розглянемо поняття *out-of-bag прикладів* – це ті приклади, що не використовуються при беггінгу.

Клас BaggingClassifier по умовчанню робить вибірку з *n* навчальних прикладів із заміною (bootstrap = True), де *n* - розмір навчального набору. Тому при беггінгу деякі приклади можуть бути обрані по декілька разів, бути присутніми по декілька разів у вибірці для будь-якого заданого прогнозатора, тоді як інші можуть не бути вибраними зовсім. Вони називаються невикористовуваними (*out-of-bag*, *oob*) прикладами.

Це означає, що при бегінгу для кожного прогнозатора буде вибиратися в середньому тільки близько 63% навчальних прикладів. Решта 37% навчальних прикладів – ті, що не вибираються, oob приклади. Зверніть увагу, що такі 37% прикладів не однакові для всіх прогнозаторів.

Оскільки прогнозатор ніколи не бачить oob приклади під час навчання, то прогнозатор можна оцінювати на oob прикладах. Тоді немає необхідності формувати окремий перевірочний набір або проводити перехресну перевірку. Натомість, можна оцінювати ансамбль, усереднюючи оцінки на oob прикладах від кожного прогнозатора.

Параметр oob_score в BaggingClassifier, встановлений рівним oob_score=True, свідчить про те що буде виконуватися автоматична оцінка якості на oob прикладах після навчання. Результуюча сума оцінки зберігається в *атрибуті oob_score_* (це й буде оцінкою правильності ансамблю на перевірочному наборі):

```
model = BaggingClassifier (DecisionTreeClassifier (), n_estimators = 500, bootstrap = True, n_jobs = -1, oob_score = True)

model. fit (X_train, y_train)
model.oob_score_
```

Клас BaggingClassifier має атрибут *oob_decision_function_*, який повертає значення функції рішень oob для кожного навчального прикладу, а саме, ймовірності класів для кожного навчального прикладу за умови що базовий оцінювач має метод predict_proba(). У наступному прикладі на основі оцінки oob встановлено, що перший навчальний приклад з ймовірністю 68.25% належить позитивному класу і з ймовірністю 31.75% - негативному класу:

```
bag_clf.oob_decision_function_

array([[ 0.31746032, 0.68253968],
       [ 0.34117647, 0.65882353],
       …
       [0 . 03108808, …],
       [0.57291667, …] ])
```

Методи випадкових ділянок (random patches method**) і випадкових підпросторів** (random subspaces method)

У класі BaggingClassifier можна робити *вибірку ознак*, тобто дозволити кожному прогнозатору навчатися на випадковому піднаборі вхідних ознак. Для цього є два гіперпараметри: max_features і bootstrap_features, які працюють аналогічно до max_samples і bootstrap, але призначені для вибірки ознак, а не вибірки прикладів.
Вибірка ознак забезпечує більшу несхожість прогнозаторів. Результат роботи ансамблю показує нижчу дисперсію, правда за рахунок трохи вищого значення зміщення.

Вибірка ознак особливо корисна, коли ми маємо справу з початковими даними високої розмірності, наприклад, зображеннями.

Методом *випадкових ділянок (random patches method)* називається одночасна вибірка навчальних прикладів і ознак (Ж.Люпп і П.Гер, 2012).

Методом *випадкових підпросторів (random subspaces method)* називається виконання вибірки ознак, але збереження всіх навчальних прикладів, тобто вибірка прикладів не виконується (Т.Хо, 1998).
Для збереження всіх навчальних прикладів встановлюються значення bootstrap = False і max_samples = 1. Для виконання вибірки ознак встановлюються значення параметрів bootstrap_features = True і / або max_features < 1.0.

Випадковий ліс

Випадковий ліс (Random Forest, Т. Хо, 1995) – це ансамбль дерев прийняття рішень, які навчаються зазвичай методом беггінга або іноді вставки і, як правило, з параметром max_samples рівним розміру навчальної вибірки.

Клас RandomForestClassifier є більш зручним і оптимізованим для дерев прийняття рішень порівняно з використанням BaggingClassifier з параметром DecisionTreeClassifier(). Побудуємо модель класифікації на основі випадкового лісу зі 100 деревами, де кожне дерево має максимум 18 вузлів:

```
model = RandomForestClassifier (n_estimators=100,  max_leaf_nodes=18,  n_jobs=-1)
```

В основному, клас RandomForestClassifier має всі гіперпараметри класу DecisionTreeClassifier для управління ростом дерев та всі гіперпараметри класу BaggingClassifier для управління самим ансамблем.
Виключення:

- відсутні гіперпараметри splitter (примусово встановлено в "random"),
- presort (примусово встановлено в False),
- max_samples (примусово встановлено рівним 1) і
- base_estimator (примусово встановлено в DecisionTreeClassifier з наданими гіперпараметрами).

Для того щоб класифікатор BaggingClassifier був приблизно еквівалентний попередньому класифікатору RandomForestClassifier потрібно задати наступні його гіперпараметри:

```
bagging_model= BaggingClassifier (
        DecisionTreeClassifier (splitter="random", max_leaf_nodes=18),
        n_estimators=100, max_samples=1, bootstrap=True, n_jobs=-1)
```

Алгоритм випадкового лісу вводить додаткову випадковість у дерева за рахунок того що при розщепленні вузла він шукає найкращу ознаку у *випадковому піднаборі ознак*, замість пошуку найкращої ознаки серед усіх ознак. В результаті виходять більш відмінні, несхожі одне на одне дерева. За рахунок цього випадковий ліс дає результат з більш низькою дисперсію, при трохи вищому значенні зміщення. В цілому модель випадкового лісу RandomForestClassifier() вважається кращою за класифікатор BaggingClassifier() з параметром DecisionTreeClassifier().

Особливо випадкові дерева (extremely randomized trees)

Як зазначалося вище, випадковий ліс вводить додаткову випадковість у дерева за рахунок того що при розщепленні вузла він шукає найкращу ознаку у випадковому піднаборі ознак, замість пошуку найкращої ознаки серед усіх ознак. В результаті виходять більш відмінні, несхожі одне на одне дерева. Можна зробити дерева ще більш випадковими за рахунок застосування випадкових порогів для кожної ознаки замість пошуку найкращих можливих порогів, як це робиться у звичайних деревах прийняття рішень.

Ліс з такими надзвичайно випадковими деревами називається *ансамблем особливо випадкових дерев* (extremely randomized trees ensemble або просто extra-trees).

Переваги особливо випадкових дерев:

- має місце більш низька дисперсія при трохи вищому значенні зміщення;
- вони навчаються набагато швидше, ніж традиційні випадкові ліси, оскільки знаходження найкращого можливого порога для кожної ознаки в кожному вузлі є однією з найбільш часовитратних задач під час навчання дерева.

Для створення класифікатора на основі особливо випадкових дерев використовується клас ExtraTreesClassifier з Scikit-Learn. Його API-інтерфейс ідентичний API-інтерфейсу класу RandomForestClassifier. Подібним чином клас ExtraTreesRegressor має такий же API-інтерфейс, як у класу RandomForestRegressor.

Щоб встановити, чи буде модель на основі ExtraTreesClassifier кращою за RandomForestClassifier, як правило, треба спробувати обидві ці моделі і порівняти їх шляхом перехресної перевірки, налаштовуючи гіперпараметри з використанням решітчастого пошуку.

Значущість ознак

Ще одна чудова якість випадкових лісів полягає в тому, що за їх допомогою легко виміряти відносну значущість кожної ознаки. Випадкові ліси вважаються дуже зручними для швидкого розуміння того, які ознаки дійсно мають значення, особливо, якщо необхідно здійснювати вибір ознак.

Бібліотека Scikit-Learn вимірює значущість ознаки шляхом з'ясування, наскільки вузли дерева, які застосовують цю ознаку, знижують забрудненість в середньому (по всім деревам у лісі).

Значущість ознаки – це зважене середнє, де вага кожного вузла дорівнює кількості навчальних прикладів, які потрапили до нього. Значущість зберігається в атрибуті feature_importances_. Ця значущість підраховується в Scikit-Learn автоматично для кожної ознаки після навчання, і значення нормуються так, що сума всіх значущостей дорівнює одиниці.
Наприклад, виконаємо класифікацію набору даних iris і знайдемо значущість кожної ознаки, використовуючи RandomForestClassifier:

```
>>> from sklearn.datasets  import load_iris
>>> iris = load_iris ( )
>>> model = RandomForestClassifier (n_estimators=500, n_jobs=-1)
>>> model. fit(iris["data"], iris["target"])
>>> for name, score in zip (iris["feature_names"], model. feature importances ) :
        print (name, score)
```

sepal length (cm) 0 . 112492250999
sepal width (cm) 0 . 0231192882825
petal length (cm) 0.441030464364
petal width (cm) 0.423357996355

Найважливішими ознаками виявилися довжина пелюстки в сантиметрах (petal length (cm)) (44%) і ширина пелюстки в сантиметрах (petal width (cm)) (42%), в той час як значущість довжини чашолистка в сантиметрах (sepal length (cm)) і ширини чашолистка в сантиметрах (sepal width (cm)) в порівнянні з ними мають набагато нижчу значущість (відповідно 11% і 2%).

Бустинг

Бустинг – це будь-який ансамблевий метод, який здатний комбінувати декількох слабких учнів в одного сильного учня. В бустингу відбувається послідовне навчання прогнозаторів, коли кожен з них намагається виправити свого попередника.

Доступно багато методів бустинга, але найпопулярнішими є AdaBoost (Adaptive Boosting – адаптивний бустинг) і градієнтний бустинг (Gradient Boosting).

AdaBoost

В методі AdaBoost новий прогнозатор виправляє свого попередника у наступний спосіб: трохи більше уваги приділяється навчальним прикладам, на яких у попередника було недонавчання. Так, при побудові класифікатора метод AdaBoost спочатку навчає перший базовий класифікатор (нехай це буде дерево рішень), який використовується для побудови прогнозів на навчальному наборі. Шукаються некоректно класифіковані ним навчальні приклади і їх відносна вага збільшується. Другий класифікатор навчається вже із застосуванням оновлених ваг, після чого він використовується для вироблення прогнозів на навчальному наборі, ваги знову оновлюються і т.д. Після того як всі прогнозатори навчені, ансамбль виробляє прогнози аналогічно до того як це роблять беггінг або вставка за винятком того, що прогнозатори мають різні ваги в залежності від їх загальної правильності на зваженому навчальному наборі.

У AdaBoost – методі послідовного навчання є один важливий недолік: він не допускає розпаралелювання (або тільки частково допускає), оскільки кожен прогнозатор можна навчати лише після того, як був навчений і

оцінений попередній прогнозатор. В результаті AdaBoost не масштабується настільки добре, як беггінг або вставка.

Докладніше розглянемо алгоритм AdaBoost. Вага кожного навчального прикладу $w^{(i)}$ спочатку встановлюється рівною $1/m$. Виконується навчання першого прогнозатора і підраховується його зважена частота помилок r_1 на навчальному наборі за наступною формулою:

$$r_j = \frac{\sum_{i=1,\, \hat{y}_j^{(i)} \neq y^{(i)}}^{m} w^{(i)}}{\sum_{i=1}^{m} w^{(i)}},$$

де r_j – *зважена частота помилок j–го прогнозатора*, $\hat{y}_j^{(i)}$ - прогноз *j*–го прогнозатора для *i*-го прикладу.

Потім *обчислюється* a_j – *вага прогнозатора*:

$$a_j = \eta \ln \frac{1 - r_j}{r_j},$$

де η – гіперпараметр швидкості навчання, стандартне його значення= 1. Чим більш правильним є прогнозатор, тим вищою буде його вага. Якщо прогнозатор всього лише випадково вгадує, тоді його вага буде близькою до нуля. Однак якщо прогнозатор майже завжди помиляється, тобто його правильність нижча за випадкове вгадування, то його вага буде від'ємною.

Далі *ваги прикладів оновлюються*:

$$w^{(i)} := \begin{cases} w^{(i)}, & \text{якщо } \hat{y}_j^{(i)} = y^{(i)} \\ w^{(i)} \exp(a_j), & \text{якщо } \hat{y}_j^{(i)} \neq y^{(i)} \end{cases}, \qquad i = 1, \ldots, m$$

В результаті ваги некоректно класифікованих прикладів зростають.

Після цього *ваги прикладів нормуються* шляхом ділення на суму $\sum_{i=1}^{m} w^{(i)}$.

На наступному етапі новий прогнозатор навчається з використанням оновлених ваг, після чого весь процес повторюється: знову обчислюється вага нового прогнозатора, оновлюються ваги прикладів, виконується навчання ще одного прогнозатора і т.д.). Алгоритм зупиняється, коли досягнута бажана кількість прогнозаторів або знайдено досконалий прогнозатор.

Етап *прогнозування* в алгоритмі AdaBoost полягає у підрахунку прогнозів за всіма прогнозаторами, і виконується їх зважування із застосуванням ваг прогнозаторів a_j. Спрогнозованим класом буде той, який отримує більшість зважених голосів:

$$\hat{y}(x) = \arg max_k \sum_{j=1,\ \hat{y}_j(x)=k}^{N} a_j \ ,$$

N – кількість прогнозаторів.

Scikit-Learn використовує багатокласову версію алгоритму AdaBoost, яка називається ***SAMME*** (*Stagewise Additive Modeling using a Multiclass Exponential loss function* – поетапне, ступінчасте адитивне моделювання із застосуванням багатокласової експоненційної функції втрат). Для випадку двох класів алгоритм SAMME еквівалентний до AdaBoost.

Якщо прогнозатори здатні оцінювати ймовірності класів, тобто мають метод predict_proba(), Scikit-Learn може використовувати модифікований алгоритм ***SAMME.R*** (R означає Real), який в якості прогнозів використовує ймовірності класів, і в цілому виконується краще.

Розглянемо приклад. Виконаємо навчання класифікатора AdaBoost, заснованого на 100 деревах рішень глибини 2, із застосуванням класу AdaBoostClassifier з Scikit-Learn:

```
from sklearn.ensemble import AdaBoostClassifier
ada_model = AdaBoostClassifier (
        DecisionTreeClassifier (max_depth=2) , n_estimators=100, algorithm=" SAMME. R" ,
        learning_rate=0.5 )
ada_model. fit(X_train, y_train)
```

Стандартний базовий оцінювач для класу AdaBoostClassifier – це *пеньок рішення (Decision Stump)* – дерево прийняття рішень з глибиною рівною 1, max_depth = 1. Це дерево, що складається з одного вузла рішення і двох листових вузлів. Попередній приклад із навчанням класифікатора AdaBoost, заснованого на 100 пеньках рішень виглядає наступним чином:

```
ada_model = AdaBoostClassifier (
        DecisionTreeClassifier (max_depth=1) , n_estimators=100, algorithm=" SAMME. R" ,
        learning_rate=0.5 )
ada_model. fit(X_train, y_train)
```

У випадку якщо ансамбль AdaBoost перенавчається навчальним набором, виконують наступне:

- скорочують кількість оцінювачів n_estimators,
- більш строго регуляризують базовий оцінювач.

Градієнтний бустинг

Інший популярний алгоритм бустинга – це градієнтний бустинг (Gradient Boosting, Л.Брейман, 1997). Подібно до AdaBoost градієнтний бустинг послідовно додає прогнозатори до ансамблю, причому кожний прогнозатор коригує свого попередника. Відмінність від AdaBoost полягає в тому, що замість налаштування ваг прикладів на кожній ітерації, алгоритм градієнтного бустингу налаштовує новий прогнозатор на залишкові помилки (residual errors), допущені попереднім прогнозатором.

Прогноз на новому прикладі знаходять як суму прогнозів за всіма моделями.

Розглянемо приклад регресії, в якому метод опорних векторів використовується в якості базового прогнозатора:

1. Спочатку алгоритм регресії на основі методу опорних векторів SVR навчається на заданому наборі даних:

```
svr_reg_1 = SVR()
svr_reg_1. fit (X, y)
```

2. Далі навчається другий регресор SVR на залишкових помилках, допущених першим прогнозатором:

```
y2 = y - svr_reg_1. predict(X)
svr_reg_2 = SVR ()
svr_reg_2. fit(X, y2)
```

3. Далі навчається третій регрессор SVR на залишкових помилках, допущених другим прогнозатором:

```
y3 = y2 - svr_reg_2 . predict (X)
svr_reg_3 = SVR ()
svr_reg_3. fit(X, y3)
```

4. Отримали ансамбль, який містить три моделі. Прогноз на новому прикладі можна знайти як суму прогнозів за всіма цими моделями:

```
y_pred = sum ( model.predict(X_new)  for model in ( svr_reg_1, svr_reg_2, svr_reg_3) )
```

Існують також:

- градієнтний бустинг на основі дерев рішень – *gradient tree boosting*,
- дерева регресії з градієнтним бустингом – *gradient boosted regression tree* – GBRT,

Більш простий спосіб навчання ансамблів GBRT полягає у застосуванні класу GradientBoostingRegressor з Scikit-Learn.

GradientBoostingRegressor (аналогічно до класу RandomForestRegressor) має гіперпараметри:

- для управління ростом дерев рішень, такі як max_depth, min_samples_leaf і т.д., а також
- для управління навчанням ансамбля, такі як n_estimators – кількість дерев та інші.

Наприклад, створення ансамблю на основі градієнтного бустингу, який містить три дерева рішень глибиною 2 кожне, задається таким кодом:

```
from sklearn.ensemble import GradientBoostingRegressor
gbrt = GradientBoostingRegressor (max_depth=2, n_estimators=3,  learning_rate=1.0)
gbrt. fit(X, y)
```

Гіперпараметр *швидкості навчання* learning_rate встановлює ступінь вкладу кожного дерева. Якщо встановити *низьке значення learning_rate*, наприклад, learning_rate=0.1, то треба включити до ансамблю *більшу кількість дерев*, щоб забезпечити гарне наближення навчальних даних. Але прогнози зазвичай будуть *краще узагальнюватися*. Керування швидкістю навчання забезпечує *регуляризацію*, яка називається *сжиманням (shrinkage).*

Щоб знайти оптимальну кількість дерев, використовують *ранню зупинку.* Її можна реалізувати з використанням методу **staged_predict()**: він повертає ітератор за прогнозами, які виробляються ансамблем на кожній стадії навчання (з одним деревом, двома деревами і т.д.).

Розглянемо приклад навчання ансамблю GBRT, який містить 160 дерев. Потрібно:

- виміряти помилку перевірки на кожній стадії навчання для знаходження оптимальної кількості дерев,

- навчити ансамбль GBRT, використовуючи знайдену оптимальну кількість дерев.

При такому підході спочатку навчається велика кількість дерев, а потім повертаються назад в пошуках оптимальної їх кількості.

```
import numpy as np
from sklearn.model_selection import train_test_split

from sklearn.metrics import mean_squared_error

X_train, X_val, y_train, y_val = train_test_split (X, y)
gbrt = GradientBoostingRegressor (max_depth=2, n_estimators=l60)

gbrt . fit (X_train, y_train)

errors = [mean_squared_error (y_val, y_pred) for y_pred in gbrt. staged_predict (X_val) ]

bst_n_estimators = np.argmin(errors)

gbrt_best = GradientBoostingRegressor (max_depth=2 , n_estimators=bst_n_estimators )

gbrt_best . fit (X_train, y_train)
```

Ранню зупинку також можна реалізувати шляхом *дострокової зупинки навчання*. Для цього встановлюють значення **warm_start**=True, що спонукає бібліотеку Scikit-Learn зберігати існуючі дерева, коли викликається метод fit( ). Це дає можливість виконувати поступове навчання.

Розглянемо приклад, в якому навчання зупиняється коли помилка перевірки не покращується для п’яти ітерацій:

```
gbrt = GradientBoostingRegressor(max_depth=2, warm_start=True)

min_val_error = float ( "inf" ) error_going_up = 0
for n_estimators in range (l, 120) :

    gbrt . n_estimators = n_estimators
    gbrt . fit (X_train, y_train)
    y_pred = gbrt . predict(X_val)
    val_error = mean_squared_error (y_val, y_pred)

    if val error < min val error:

        min val error = val error

        error_going_up = 0
```

```
else :

    error_going_up += 1
    if error_going_up == 5 :

        break # рання зупинка
```

Клас GradientBoostingRegressor також має гіперпараметр **subsample**, який вказує долю навчальних прикладів, які будуть використані для навчання кожного дерева. Наприклад, якщо subsample=0.25, тоді кожне дерево навчається на 25% навчальних прикладів, які вибираються випадковим чином. Такий прийом називається ***стохастичним градієнтним бустингом*** *(stochastic gradient boosting)*. Він значно прискорює процес навчання. А також обмінює більш високе зміщення на більш низьку дисперсію.

Градієнтний бустинг можна застосовувати з іншими функціями втрат, для цього є гіперпараметр loss .

Стекінг

Ансамблевий метод, який називається стекінгом (stacking – скорочено від "stacked generalization", стекове узагальнення, Уолперт, 1992) базується на наступній ідеї: для агрегування прогнозів усіх прогнозаторів в ансамблі використовувати не тривіальні функції, як при жорсткому прогнозуванні, а навчити для виконання агрегування певну модель. Стекінг реалізується у два етапи:

1) на першому етапі кожний індивідуальний прогнозатор (він ще називається ***прогнозатор першого рівня***) дає своє значення прогнозу, в загальному випадку відмінне від значень інших прогнозаторів,
2) на другому етапі заключний прогнозатор, який називається ***змішувачем (blender)*** або ***мета-учнем (meta learner)*** отримує на вході такі прогнози і виробляє остаточний прогноз.

Навчальний набір розбивається на два піднабори:

1. Перший піднабір використовується для навчання прогнозаторів першого рівня.
2. Прогнозатори першого рівня виконують прогнозування на другому, так званому ***утриманому (hold-out)*** піднаборі, і виробляються ***прогнози першого рівня***. Слід зазначити, що приклади з утриманого піднабору не використовувалися при навчанні прогнозаторів.

В результаті для кожного прикладу утриманого піднабору є множина спрогнозованих значень.

3. Далі створюється новий навчальний набір, використовуючи прогнози першого рівня в якості вхідних ознак (тому новий навчальний набір стає багатовимірним) і зберігаючи значення цільової змінної. ***Змішувач навчається*** на цьому новому навчальному наборі, навчається прогнозувати цільові значення на основі прогнозів першого рівня.

Таким способом можна навчити декілька різних змішувачів побудувати *багаторівневий ансамбль зі стекінгом*. Наприклад, декількома різними змішувачами можуть бути: який використовує випадковий ліс для класифікації, ще один змішувач, який використовує опорні вектори і т.д.

Можна виконати розбиття навчального набору на три піднабори:
- перший піднабір використовується для навчання першого рівня,
- другий піднабір використовується для створення навчального набору, який використовується для навчання другого рівня, використовуючи прогнози зроблені прогнозаторами першого рівня,
- третій піднабір використовується для створення навчального набору, який використовується для навчання третього рівня, використовуючи прогнози зроблені прогнозаторами другого рівня.
- заключний прогноз для нового прикладу отримують послідовно проходячи за всіма цими рівнями.