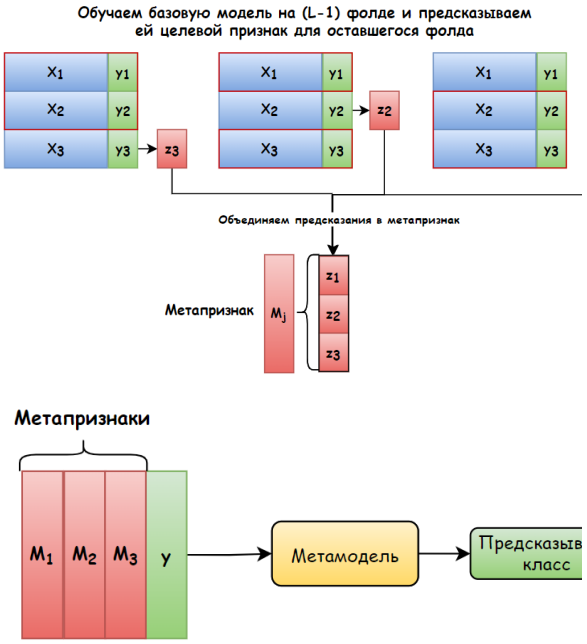
SkillFactory

Модели, используемые для ансамблирования, называются **базовыми.**
Существует три основных способа построения ансамблей:

- **Бэггинг** — алгоритм построения ансамбля путем параллельного обучения множества независимых друг от друга моделей. Ансамблирование для классификации происходит посредством **большинства голосов (Majority Vote).** Для задачи регрессии же ансамблирование происходит посредством **усреднения результата предсказания каждой базовой модели (Averaging)**.

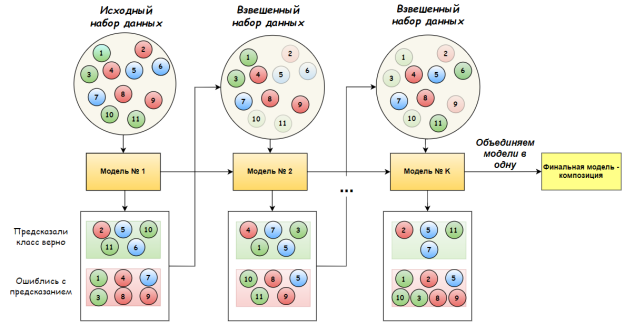
- **Стекинг** — алгоритм построения ансамбля, в котором параллельно и независимо друг от друга обучаются несколько ***базовых моделей*** (не обязательно одной природы), а их предсказания используются для обучения ***метамодели*** (финальная модель) как факторы.



Стекинг для задачи регрессии имеет реализацию в библиотеке Scikit-learn, реализовано в классе StackingRegressor, для задачи классификации класс StackingClassifier. На вход подается список базовых моделей (атрибут estimators) и мета-модель (атрибут final_estimator).

- **Бустинг** — алгоритм построения ансамбля, основанный на *последовательном* построении слабых моделей, причем, каждая новая модель пытается уменьшить ошибку предыдущей. После того как все модели обучены они объединяются в композицию.

Примечание:
Под слабыми моделями мы подразумеваем модели, точность которых не многим выше, чем случайное угадывание. Как правило, это короткие деревья решений, они обладают слабой предсказательной способностью.



Первая реализация бустинга называлась AdaBoost. Опишем алгоритм его работы на примере задачи классификации:

1. Пусть у нас есть набор данных $x = [x_1, x_2, ..., x_n]$, в котором n объектов размерности m (вектора в признаковом пространстве размера m) и метки класса $y \in \{-1, 1\}$, где -1 и 1 метки отрицательного и положительного класса соответственно.
2. Перед обучением базовой модели мы инициализируем веса объектов обучающей выборки как

$$w_0(x_i, y_i) = \frac{1}{n}$$

3. Далее мы выбираем j-й базовый классификатор $f_j(x)$ из всех базовых классификаторов (пусть их будет N) с наименьшей ошибкой, обозначим ее за ϵ_j.
4. Тогда вес предсказаний данного классификатора высчитываем по формуле:
$$\eta_j = \frac{1}{2}\ln(\frac{1-\epsilon_j}{\epsilon_j})$$
5. Формула веса становится нулевым только при случайном угадывании, т.е когда классификатор ошибается в половине меток, т.е работает ровно также как и подбрасывание монетки . Однако остальные будут вносить вес в итоговую модель с положительным или отрицательным знаком (легко убедиться, подставив вероятность ошибки отличную от 0.5). Тем самым мы исключаем возможность вклада случайных классификаторов в результирующую модель.
6. Тогда далее при каждом добавлении нового алгоритма в бустинг вес объекта обучающей выборки будет изменяться следующим образом:
$$w_{j+1} = \frac{w_j(x_i,y_i)e^{-\eta_j y_i f_j(x_i)}}{W_j}$$, где W_j – коэффициент нормировки.
7. Стоит отметить, что при неверной классификации, из формулы выше видно, что вес отдельного объекта начинает расти и модель начинает “зацикливаться” сильнее на неправильных предсказаниях, тем самым уточняя предсказания предыдущей модели.
8. Обучается базовый алгоритм, решаю задачу минимизации ошибки, учитывая веса объектов выборки:
$$L(y, f_j(x, W)) = \sum_i^n w_i[y_i f_i(x_i) < 0] \rightarrow mi$$
Пояснение: произведение $y_i f_i(x)$ отрицательно в случае неверной классификации объекта (x_i, y_i), т.е тем самым минимизируя эту ошибку мы уменьшаем количество неверных предсказаний, учитывая еще вес предсказания (чем он больше, тем больше модель стремится сделать его правильным, т.е происходит то, что объясняется в пункте 7 данного алгоритма)
9. После обучения, предсказание классификации берется по следующему правилу:
$$f(x) = sign(\sum_{i=1}^{N} \eta_i f_i(x))$$

Примечание: функция sign - функция взятия знака, принимает значения -1, если аргумент функции отрицательный, 1, если аргумент функции положительный.

В sklearn адаптивный бустинг над решающими деревьями реализован в модуле sklearn.ensemble в виде классов AdaBoostRegressor и AdaBoostClassifier для задач регрессии и классификации, соответственно.

Градиентный бустинг (Gradient Boosting, GB)

В GB принцип классического бустинга сохраняется - каждый последующий алгоритм улучшает предыдущий, но только в отличие эвристического "взвешивания" наблюдений градиентный бустинг использует информацию о функции потерь для построения нового алгоритма.

Допустим, у нас есть некоторая функция потерь $L(y, \hat{y})$. Она зависит от двух аргументов $y = y(x)$ - истинный ответ, а $\hat{y} = a(x)$ - прогноз модели $a(x)$. Причем не важно какая это функция потерь, почти :) (на самом деле ограничения есть - это дифференцируемость функции, то есть существование производной).

Для задачи регрессии это может быть, например, MSE:

$$L(y(x), f(x)) = -\frac{1}{n}\sum_{i}^{n}(y_i(x) - f_i(x))^2$$

Для задачи классификации это может быть, например, logloss:

$$L(y(x), f(x)) = -\sum_{i}^{n}(y_i(x)log(f_i(x)) + (1 -$$

Допустим, мы каким-то образом рассчитаем этот антиградиент на обучающей выборке $-\nabla_{f_k} L$. Это будет просто вектор-столбец из каких-то чисел. Ну числа предсказывать мы умеем, то есть будем решать задачу регрессии: будем с помощью модели $b(x)$ предсказывать координаты этого вектора антиградиента. (Например, можно построить такую модель $b(x)$, чтобы ее средний квадрат ошибки между ее ответами и антиградиентом (MSE) был минимален).

Когда мы построим модель новую $b(x)$, мы добавим ее ответы в композицию, но не все сразу, а умножим ответы на коэффициент η - темп обучения. Этот коэффициент влияет на то, какой вклад каждая следующая модель будет оказывать на общую композицию. То есть:

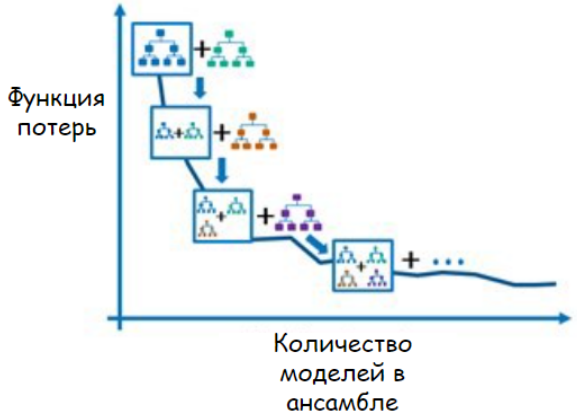
$$f_{k+1}(x) = f_k(x) + \eta\, b(x)$$

Где $f_k(x)$ - ансамбль, построенный на k-ом этапе, $f_{k+1}(x)$ - новый ансамбль, который будет построен на следующем этапе k+1.

Примечание:
Если принять, что предсказания $b(x)$ полностью совпадают с антиградиентом $b(x) = -\nabla_{a_k} L$, то получится формула градиентного спуска:

$$f_{k+1}(x) = f_k(x) - \eta \nabla_{f_k} L$$

Но только градиентный спуск происходит не в пространстве параметров, а в пространстве ответов ансамбля f_k. Что за пространство такое и как там считать градиенты, мы обсудим, когда будем разбирать градиентный бустинг по кусочкам.
Сейчас же важно понимать, что каждая новая модель бустинга будет строиться так, чтобы двигать всю композицию вниз по функции потерь — в сторону вектора антиградиента.



Бустинг, использующий в качестве базовой модели дерево решений, называется градиентным ***бустингом над деревьями решений (Gradient Boosting on Decision Trees, GBDT)***.
Градиентный бустинг над деревьями для решения задачи регрессии реализован в классе GradientBoostingRegressor. Для задачи классификации данный метод реализован в классе GradientBoostingClassifier.

Плюсы и минусы методов

Бэггинг

✅ **Плюсы**

Хорошо параллелится вычисление (модели обучаются параллельно)

Снижает дисперсию

⛔ **Минусы**

Предполагается использование одинаковых моделей

Необходимо использование глубоких деревьев

Плохо интерпретируемая

Стекинг

✅ **Плюсы**

Хорошо параллелится (модели обучаются параллельно)

Хорош для использования различных по природе базовых моделей

⛔ **Минусы**

Качество сильно зависит от качества базовых моделей

Плохо интерпретируемая

Бустинг

✅ **Плюсы**

Модели обучаются последовательно, уточняя друг друга

Снижает смещение

Базовые модели — неглубокие деревья

⛔ **Минусы**

Плохо параллелится вычисление

Плохо интерпретируемая

Пайплайны

Пайплайн — автоматизированный поэтапный процесс выполнения манипуляций с данными, включающий в себя сбор, обработку, генерацию и отбор признаков, обучение модели с последующей ее настройкой и проверкой качества.

Основные цели использования пайплайнов: автоматизация, ускорение вычислений с использованием многопоточности в Python и дальнейшее развертывание пайплайна для использования в периодических расчетах (сбор данных в режиме онлайн/ онлайн работа модели). Кроме того, пайплайны хороши в случае, когда надо подобрать оптимальные гиперпараметры для всего цикла обработки данных и последующего обучения.

В библиотеке scikit-learn пайплайны реализованы как класс sklearn.pipeline.Pipeline(). Данный класс может быть использован для сбора воедино отбора и обработки данных вместе с итоговой моделью. Важной особенностью применения пайплайна в реализации scikit-learn является обязательное наличие у каждого метода внутренних преобразований transform() и для финальной модели метод fit().

Пример пайплайна:

```
pipeline =
Pipeline([('scaler',
StandardScaler()), ('rf',
RandomForestRegressor())])
pipeline.fit(X_train,
y_train)
```

Примечание: *В качестве альтернативного метода задания пайплайна можно использовать метод make_pipeline, на вход которого подаются объекты, которые будут использованы в пайплайне. В нашем случае:*

```
from sklearn.pipeline import
make_pipeline
make_pipeline(StandardScaler
(), RandomForestRegressor())
```

Изменение параметра в пайплайне можно сделать, используя метод set_params(), на вход которого надо подавать конструкцию <название модуля>__<название параметра>.

```
pipeline.set_params(rf__n_es
timators=200)
```

ColumnTransformer

Например, у нас есть численный признак цена вина (Price) и категориальный регион производства (Region). Для первого надо применить StandardScaler(), для второго OneHotEncoder(). Мы можем сделать это следующим образом:

```
from sklearn.preprocessing
import StandardScaler,
OneHotEncoder
from sklearn.compose import
make_column_transformer
ct = make_column_transformer(
     (StandardScaler(),
['Price']),
    (OneHotEncoder(),
['Country']))
print(ct)
```

Примечание: *Если мы хотим применить, например, OneHotEncoder, к более, чем одному признаку, то просто достаточно добавить в список колонки, например* `['Country', 'Region']`. *Теперь OneHotEncoder будет работать не только на признаке Country, но и еще на Region.*

Примечание:

Можно также использовать фильтрацию по типу колонок используя метод make_column_selector() из sklearn.compose :

```
ct = ColumnTransformer([
        ('scale',
StandardScaler(),

make_column_selector(dtype_i
nclude=np.number)),
        ('onehot',
       OneHotEncoder(),

make_column_selector(dtype_i
nclude=object))])
```

Полученный трансформер можно использовать в качестве элемента пайплайна, например:

```
pipeline = Pipeline([('ct',
ct), ('rf',
RandomForestRegressor())])
```

Обученный пайплайн можно сохранить в формат pickle (подробнее по ссылке), например используя библиотеку joblib и использовать после десериализации как готовое решение “из коробки”, используя следующие методы.

Примечание*: Формат pickle реализует двоичный протокол для сериализации/десериализации объектов для сохранения и последующего использования без каких-либо дополнительных преобразований. Удобен для переноса обученных моделей, предобработки и тд.*

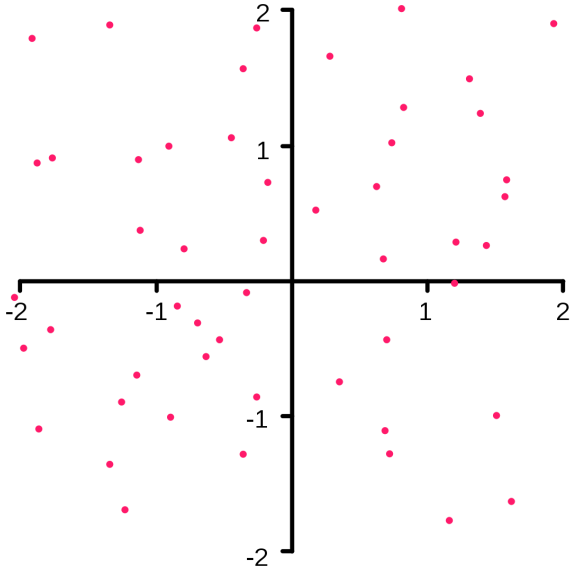
Сериализация *— процесс перевода структуры данных в последовательность битов.*
Десериализация *— процесс создания из последовательности битов структуры данных.*

```
!pip install joblib
import joblib
joblib.dump(pipeline,
'pipeline.pkl')
```

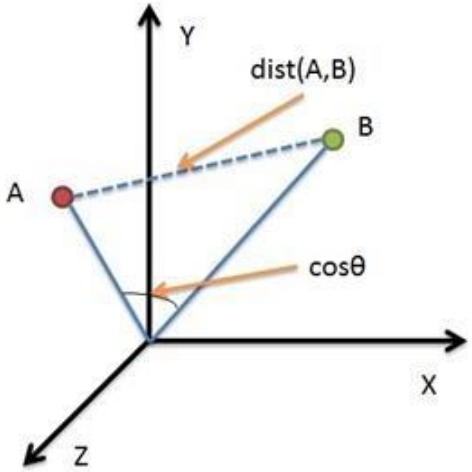
Metric Learning

Задачи metric learning основаны на измерении расстояния между объектами выборки. В отличие от задач обучения с учителем, такими как классификация и регрессия, такие задачи, чаще всего, не имеют явной и однозначной целевой метки, поэтому эти задачи по-другому называют **задачи без учителя (unsupervised learning).** К основным постановкам задачам metric learning относят задачи кластеризации, задачи поиска ближайших соседей, задачи снижения размерности и частичного восстановления данных.

Каждый объект выборки можно представить как точку в n-мерном пространстве, где n — количество признаков, которые описывают объект. Тогда расстояние есть некоторая мера дистанции между объектами в n-мерном пространстве (расстояние между n-мерными векторами, описывающими объекты). На иллюстрации приведена визуализация объектов, описываемых двухмерным пространством (количество признаков равно двум). Мы можем визуализировать также трехмерное пространство и одномерное, однако размерности выше n >3 визуализировать уже сложно.



Для понимания расстояния между двумя точками прибегнем к помощи еще одной иллюстрации:



На рисунке представлено расстояние между двумя точками А и Б уже в трехмерном пространстве и обозначено как dist(A,B).

Расстояние Махаланобиса, обозначаемое далее в формуле как D, между объектами x и x’ - объекты выборки, описываемые признаковым пространством (на практике это будут две строки в датасете, длиной равной количеству признаков), вводимое как:

$$D(x, x') = \sqrt{(Lx - Lx')^T (Lx - Lx')},$$

где L - матрица преобразование пространства, например понижение размерности.

Также в задачах metric learning используются альтернативные метрики, рассмотрим некоторые из них:

- Расстояние Миньковского, обозначим как Dm:
  - $$Dm(x, x') = (\sum_i^n |x_i - x'_i|^p)^{\frac{1}{p}}$$
    , где p некоторый параметр (p>1)
  - При $p \to \infty$ расстояние Миньковского превращается в расстояние Чебышева $Dm(x, x')_{p\to\infty} = \max_i |x_i - x'_i|$
  - При p=1 - расстояние Манхэттена
    $$Dm(x, x')_{p=1} = \sum_i^n |x_i - x'_i|$$
    или L1-расстояние.
  - При p=2 - евклидово расстояние, L2-расстояние
    $$Dm(x, x')_{p=2} = \sqrt{\sum_i^n |x_i - x'_i|^2}$$
- Косинусоидальное расстояние (далее обозначено как cosdist)- расстояние между векторами $\bar{u}$ и $\bar{v}$ описывающими объекты выборки x и x’
  $$cosdist(u, v) = 1 - \frac{u\ v}{||u||_2 ||v||_2}$$
  , где

$$||x||_2 = \sqrt{\sum_i^n |x_i|^2}$$ -

евклидова норма n-мерного вектора.

Кластеризация. Поиск ближайшего соседа

Кластеризация — задача разбиения на кластеры с последующим описанием и интерпретацией общих черт кластера.

Кластеризация k-means реализована в sklearn.cluster.

Основная сложность задачи кластеризации заключается в том, что не всегда известно истинное значение количества кластеров, иногда надо это исследовать.

Для этого нам поможет `Elbow Method` `реализованный в` `yellowbrick.cluster` `как KElbowVisualizer.` `Distortion score` есть квадрат расстояния от точки до ее центра кластера. Оптимальным на графике ниже количество кластеров считается, если после него `distortion score` падает практически линейно. Пример

```
Elbow_M = KElbowVisualizer(KMeans(),
k=10)
```

Поиск ближайших соседей состоит в определении уже существующего кластера, к которому относится объект.

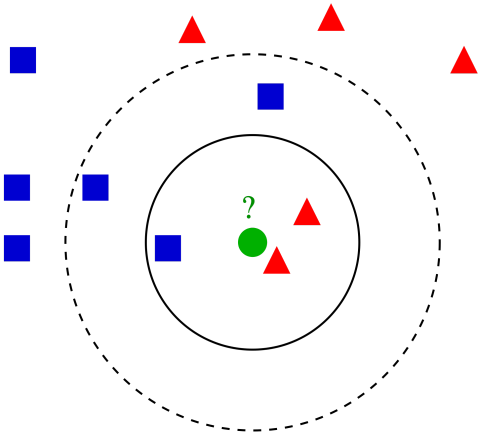
Задачи обучение с учителем, где целевые метки получены в результате кластеризации называются слабым обучением с учителем (weak supervised learning).

Алгоритм работы KNN

1. Зададим k — количество ближайший соседей, по которым принимается решение
2. Выберем расстояние, по которому будут определяться соседи. Обычно это евклидово расстояние, рассмотренное ранее.
3. Обучим модель на признаковом пространстве
   Как было сказано ранее, **обучение алгоритма KNN заключается в получении пар признаковое пространство-целевая метка.**
   Далее данные нормализуются с использованием MinMaxScaler(). Напомним его формулу:

$$X = \frac{X - min[X]}{max[X] - min[X]}$$

4. Получим предсказание модели
   Предсказание заключается в поиске k ближайших соседей на основе метрики, выбранной ранее. В sklearn реализован в модуле sklearn.neighbors как KNeighborsClassifier для задачи классификации и KNeighborsRegressor для задачи регресии. Для понимания работы получения предсказания, обратимся к иллюстрации:



На рисунке представлена задача бинарной классификации с двумя классами (синие квадраты и красные треугольники). Нам надо для нашего

объекта, отмеченного зеленым кругом определить класс.

Стоит отметить, что если k = 3 (в sklearn данный параметр главный и называется n_neighbors), то в радиус трех ближайших соседей (в модели он определяется евклидовым расстоянием от классифицируемого объекта, которое рассчитывается различными методами (в sklearn параметр algorigthm (см. параметры алгоритма) в том числе и жадным алгоритмом) попадают два красных треугольника и 1 квадрат (на рисунке это черная линия), т.е большинство представляют объекты класса красных треугольников и стоит принять класс нашего объекта за класс красных треугольников.В данном случае вероятность что мы определили верно класс равна 0.66 (2/3).

Однако, если изначально k = 5, то в этом радиусе 5 ближайших соседей (отмечено черным пунктиром) уже 3 квадрата и 2 треугольника, тогда класс объекта определяется как наиболее часто встречаемый класс среди соседей и объект вероятнее всего принадлежит квадратам. В данном случае с вероятность 0.6, если при ансамблировании не учитывать расстояние. В случае учета расстояния (параметр weights, значение distance) при k=5 может случиться так, что наш объект будет классифицирован как красный треугольник, так как у этих двух объектов будет больший вес по причине сильной близости к объекту.

KNN в sklearn реализован в
`sklearn.neighbors`
`KNeighborsClassifier`
`или KNeighborsRegressor`

Примечание:
В случае задачи регрессии, целевая метка нашего объекта определяется как среднее значение целевой метки по k ближайшим соседям.

Расчеты ведутся точно так же, только вместо вероятности предсказания класса высчитывается среднее по ближайшим соседям. Также, при использовании взвешенного подхода, среднее уже будет взвешенное в зависимости от расстояния до соседа, чем ближе к точке, тем больший вклад соседа.

Т.е предположим целевая метка у 3-х ближайших соседей в нашем случае равна 0.2, 0.3 и 0.8 соответственно, целевая метка без учета расстояния равна (0.2+0.3+0.8)/3 = 0.43.

Снижение размерности

Снижения размерности является также основной задачей metric learning. Основная суть данной задачи заключается в снижении размерности признакового пространства с сохранением информативности для дальнейшей работы с уменьшенным признаковым пространством в различных задачах, например задачах классификации или кластеризации. Самый популярный PCA

```
from sklearn.decomposition import PCA
pca = PCA(n_components=3)
pca.fit_transform(X)
```