Вычислительная сложность алгоритмов Machine Learning

Лобов Валерий

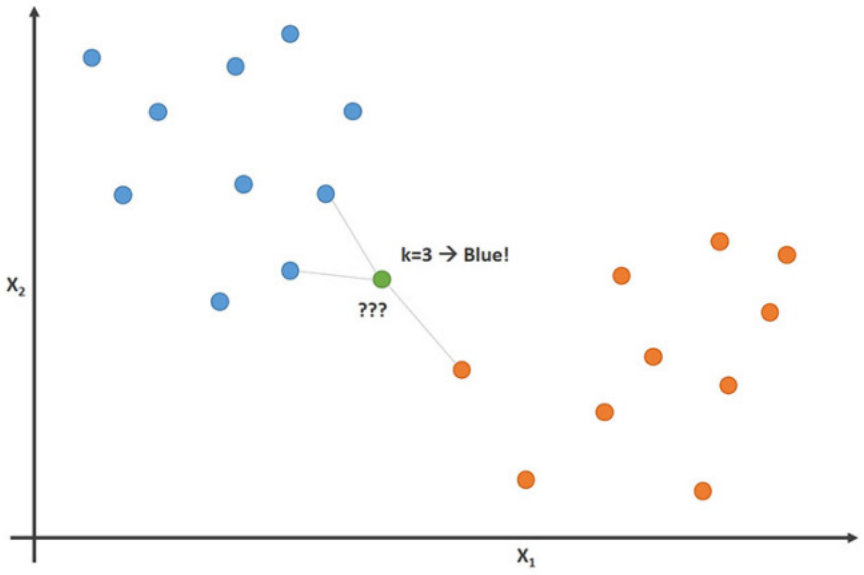
1 Постановка задачи

Имеется несколько моделей машинного обучения. Будем считать, что:
n - длина входных данных (количество строк в обучающем датафрейме),
p - количество признаков в обучающем датафрейме,
n_{trees} - количество деревьев в Decision Tree алгоритмах,
n_{sv} - количество саппорт векторов.

Необходимо построить обзорную картину классических алгоритмов машинного обучения и проанализировать их вычислительную сложность.

2 k-Nearest Neighbours

2.1 Описание алгоритма



Метод ближайшего соседа является, пожалуй, самым простым алгоритмом классификации. Классифицируемый объект x относится к тому классу y_i, которому принадлежит ближайший объект обучающей выборки x_i.

В методе k ближайших соседей для повышения надёжности классификации объект относится к тому классу, которому принадлежит большинство из его соседей — k ближайших к нему объектов обучающей выборки x_i. В задачах с двумя классами число соседей берут нечётным, чтобы не возникало ситуаций неоднозначности, когда одинаковое число соседей принадлежат разным классам.

Основная формула

Пусть задана обучающая выборка пар «объект-ответ» $X^n = \{(x_1, y_1), \ldots, (x_n, y_n)\}$.

Пусть на множестве объектов задана функция расстояния $\rho(x, x')$. Эта функция должна быть достаточно адекватной моделью сходства объектов. Чем больше значение этой функции, тем менее схожими являются два объекта x, x'.

Для произвольного объекта u расположим объекты обучающей выборки x_i в порядке возрастания расстояний до u:
$\rho(u, x_{1;u}) \leq \rho(u, x_{2;u}) \leq \cdots \leq \rho(u, x_{n;u})$, где через $x_{i;u}$ обозначается тот объект обучающей выборки, который является i-м соседом объекта u.

Аналогичное обозначение введём и для ответа на i-м соседе: $y_{i;u}$. Таким образом, произвольный объект u порождает свою перенумерацию выборки. В наиболее общем виде алгоритм ближайших соседей есть $a(u) = \arg\max_{y \in Y} \sum_{i=1}^{n} [x_{i;u} = y] w(i, u)$, где $w(i, u)$ — заданная весовая функция, которая оценивает степень важности i-го соседа для классификации объекта u. Естественно полагать, что эта функция неотрицательна и не возрастает по i.

2.2 Анализ сложности

Метод k ближайших соседей предполагает несколько алгоритмических реализаций, благодаря чему время работы может варьироваться между $O(np+kn)$, $O(npk)$ и $O(np)$, где $O(p)$ - время, занимаемое на вычисление расстояния между двумя конкретными точками.

Рассмотрим **O(np+kn)** алгоритм:

1. Инициализируем переменную $selected_i = 0$ для всех точек i в обучающей выборке.
2. Для точки из обучающей выборки i, вычислим $dist_i$, расстояние между каждой другой точкой и рассмотренной i.
3. **for j=1 to k:** Цикл по всем точкам из обучающей выборки, выбираем индекс i, для которого значение $dist_i$ минимально и $selected_i$=0. Выбираем эту точку и ставим $selected_i$=1.
4. Возвращаем набор из k точек.

Каждое вычисление расстояния требует $O(p)$ времени, поэтому шаг 2 занимает $O(np)$. В третьем шаге, мы совершаем $O(k)$ операций, проходя по каждой точке из обучающей выборки, поэтому суммарно этот шаг требует $O(nk)$ времени. Первый и четвертый шаги занимают $O(n)$, поэтому в итоге время работы **O(np+kn)**.

Теперь рассмотрим **O(npk)** алгоритм:

1. Инициализируем переменную $selected_i = 0$ для всех точек i в обучающей выборке.

2. **for j=1 to k:** Для каждой точки i из обучающей выборки, вычислим $dist_i$ - расстояние между каждой другой точкой и рассмотренной i. Выбираем точку, для которого значение $dist_i$ минимально и $selected_i$=0. Ставим $selected_i$=1.

3. Возвращаем набор из k точек для некоторой рассматриваемой точки.

На каждой итерации во втором шаге, мы вычисляем расстояние между рассматриваемой точкой и каждой другой точкой в обучающей выборке, это занимает $O(np)$ времени и в итоге получаем сложность **O(npk)**.

Разница между двумя алгоритмами в том, что первый просчитывает и хранит расстояния (требуя $O(n)$ дополнительной памяти), а второй - нет. Однако, исходя из того, что мы уже храним всю обучающую выборку за $O(np)$ памяти и храним вектор $selected$ за $O(n)$ памяти, расход памяти у обоих алгоритмов асимптотически совпадает. Поэтому, время работы для $k > 1$ делает первый алгоритм более привлекательным.

Стоит отметить, что возможно достичь времени работы **O(np)**, используя следующий алгоритм:

1. Для точки i из обучающей выборки, рассчитаем $dist_i$, расстояние от новой рассматриваемой точки до i.

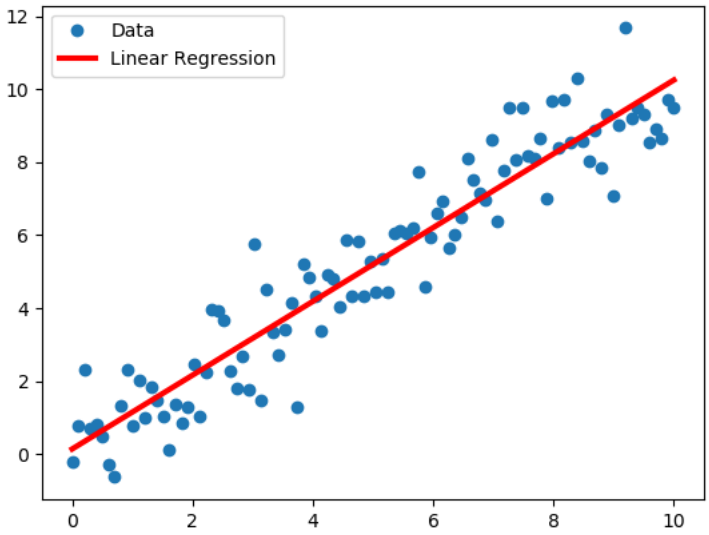
2. Запускаем quickselect алгоритм, чтобы посчитать k-ое наименьшее расстояние за $O(n)$.

3. Возвращаем все точки с посчитанным расстоянием не более чем k-ое наименьшее расстояние.

Этот подход выигрывает тем, что использует эффективный метод, чтобы найти k-ое наименьшее значение в несортированном массиве.

3 Linear Regression

3.1 Описание алгоритма



Используется для оценки реальных значений (стоимость домов, количество вызовов, общий объем продаж и т.д.) на основе непрерывной переменной. Здесь мы устанавливаем связь между независимыми и зависимыми переменными, устанавливая лучшую линию взаимосвязи. Эта линия наилучшего соответствия известна как линия регрессии и представлена линейным уравнением $Y = aX + b$.

3.2 Анализ сложности

Суть поиска весов в модели линейной регрессии сводится к решению матричного уравнения $\beta = (X^T X)^{-1} X^T Y$.

Самая вычислительно сложная часть в данном алгоритме это вычисление произведения матриц $X^T X$, которое можно оценить как $O(p^2 n)$. После чего происходит обращение матрицы за $O(p^3)$ операций.

Несмотря на то, что большинство реализаций линейной регрессии используют градиентный спуск для решения уравнения $(X^T X)\beta = X^T Y$, вычислительная сложность остается такой же.

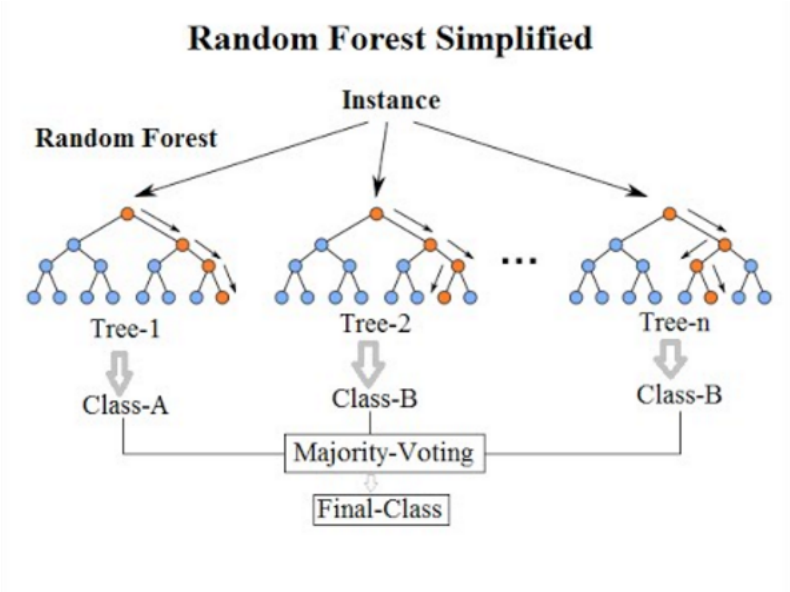
Получаем итоговую сложность алгоритма равную $\boldsymbol{O(p^2 n + p^3)}$.

Стоит отметить, что реализация линейной регрессии в популярной Python библиотеке $scikit - learn$ имеет сложность приблизительно $\boldsymbol{O(n^{0.72} p^{1.3})}$. Достигается это благодаря одной из следующих возможных причин. Первое: ни одна более-менее нормальная реализация линейной регресии не использует полное обращение матрицы. Вместо этого используется решение уравнение с помощью градиентного спуска. Вторая причина: используются другие методы для калибровки матрицы весов линейной регрессии.

4 Decision Tree based models

4.1 Random Forest

4.1.1 Описание алгоритма



Решающие деревья являются хорошим семейством базовых классификаторов для бэггинга, поскольку они достаточно сложны и могут достигать нулевой ошибки на любой выборке. Метод случайных подпространств позволяет снизить коррелированность между деревьями и избежать переобучения. Базовые алгоритмы обучаются на различных подмножествах признакового описания, которые также выделяются случайным образом. Ансамбль моделей, использующих метод случайного подпространства, можно построить, используя следующий алгоритм:

Пусть количество объектов для обучения равно N, а количество признаков D.

Выбрать L как число отдельных моделей в ансамбле.

Для каждой отдельной модели l выбрать dl как число признаков для l. Обычно для всех моделей используется только одно значение dl.

Для каждой отдельной модели l создать обучающую выборку, выбрав dl признаков из D, и обучить модель.

Теперь, чтобы применить модель ансамбля к новому объекту, нужно объединить результаты отдельных L моделей мажоритарным голосованием или путем комбинирования апостериорных вероятностей.

Алгоритм

Алгоритм построения случайного леса, состоящего из деревьев, выглядит следующим образом:

Для каждого $n = 1..N$:

Сгенерировать выборку X_n с помощью бутстрэпа;

Построить решающее дерево b_n по выборке X_n:

— по заданному критерию мы выбираем лучший признак, делаем разбиение в дереве по нему и так до исчерпания выборки

— дерево строится, пока в каждом листе не более n_min объектов или пока не достигнем определенной высоты дерева

— при каждом разбиении сначала выбирается m случайных признаков из n исходных, и оптимальное разделение выборки ищется только среди них.

Итоговый классификатор $a(x) = \frac{1}{N}\sum_{i=1}^{N} b_i(x)$, простыми словами — для задачи классификации мы выбираем решение голосованием по большинству, а в задаче регрессии — средним.

Рекомендуется в задачах классификации брать $m = \sqrt{n}$, а в задачах регрессии — $m = \frac{n}{3}$, где n— число признаков. Также рекомендуется в задачах классификации строить каждое дерево до тех пор, пока в каждом листе не окажется по одному объекту, а в задачах регрессии — пока в каждом листе не окажется по пять объектов.

4.1.2 Анализ сложности

Оценим вычислительную сложность операции predict в случайных лесах. Каждая операция включает в себя обход каждого из M деревьев, от корня до одного из листьев. Поэтому вычислительная сложность напрямую зависит от средней глубины листьев. Имея ввиду, что данные из обучающей и тестовой выборок имеют одинаковое распределение, анализ сложности процедуры предсказывания сводится к анализу средней глубины листьев, порожденных обучающей выборкой L. Будем рассматривать полное решающее дерево, где каждый лист содержит ровно один элемент из L. Рассмотрим среднюю глубину $D(N)$ листьев в лучшем, худшем и среднем случаях.

Теорема 1. В наилучшем случае средняя глубина листьев оценивается как $\Theta(logN)$.

Доказательство. Для идеально сбалансированного решающего дерева, вершины рекурсивно разделяются на на два подмножества с $N_t/2$ элементами в каждой вершине, из которого следует данное реккурентное соотншение (в лучшем случае):

$$\begin{cases} \mathrm{D}(1) = 1 \\ D(N) = 1 + 0.5D(N/2) + 0.5(N/2) = 1 + D(N/2). \end{cases}$$

Тогда можем оценить $D(N)$ как $D(N) = \Theta(log^{k+1}N) = \Theta(logN)$.

Теорема 2 В худшем случае средняя глубина листьев оценивается как $\Theta(N)$.

Доказательство. В худшем случае, реккурентное соотношение для средней глубины листьев выглядит следующим образом:

$$\begin{cases} \mathrm{D}(1) = 1 \\ D(N) = 1 + (1/N)D(1) + (N-1/N)D(N-1). \end{cases}$$

Введем величину $S(N) = ND(N)$. Из предыдущего равенства получаем $S(N) = N + 1 + S(N-1) = 1 + \sum_{i=2}^{N}(i+1) = (N^2)/2 + 3N/2$

Так как $D(N) = S(N)/N$, имеем $D(N) = 1/2(N - 2/N + 3) = \Theta(N)$.

Теорема 3. В большинстве случаев, средняя длина листьев оценивается как $\Theta(logN)$.

Доказательство. В среднем случае, реккурентное соотношение для средней глубины листьев выглядит следующим образом:

$$\begin{cases} \mathrm{D}(1) = 1 \\ D(N) = 1 + 1/(N-1)\sum_{i=1}^{N-1}((i/N)D(i) + (N - i/N)D(N-i)). \end{cases}$$

Домножая обе стороны на $N(N-1)$, получаем $N(N-1)D(N) = N(N-1)+2\sum_{i=1}^{N-1}(iD(i))$.

Для $N > 3$, заменяя N на $N-1$ аналогично получаем $(N-1)(N-2)D(N-1) = (N-1)(N-2) + 2\sum_{i=1}^{N-2}(iD(i))$.

Вычитая последнее равенство из предыдущего, после упрощения и деления обоих сторон на $N(N-1)$:

$D(N) = D(N-1)+2/N = 1+\sum_{i=2}^{N}(1/i) = 2H_N - 1$ где H_N это N_{th} гармоническое число (член гармонического ряда). Используя $H_N \approx logN+\gamma+12N+O(1/N^2)$ как аппроксимацию (где γ - постоянная Эйлера-Маскерони), мы получаем $D(N) = \Theta(logN)$.

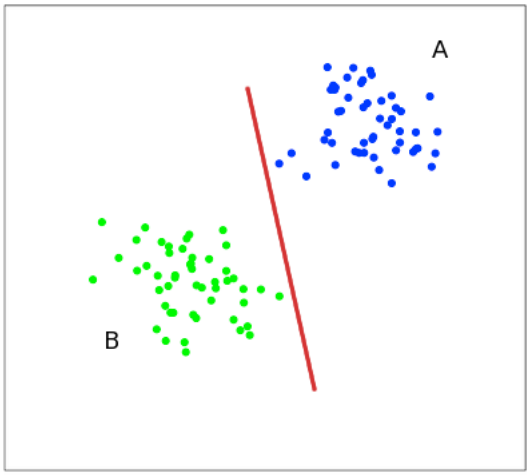
Из теорем 1-3, получаем, что суммарная вычислительная сложность для операции predict в случайном лесу из M деревьев оценивается как $\Theta(MlogN)$ в лучшем случае и $\Theta(MN)$ в худшем. В соответствии с предыдущими результатами, анализ среднего случая показал, что его вычислительная сложность все таки приближается к таковой в наилучшем случае.

5 Support Vector Machine (SVM)

5.1 Описание алгоритма

Основная идея метода — перевод исходных векторов в пространство более высокой размерности и поиск разделяющей гиперплоскости с максимальным зазором в этом пространстве. Две параллельных гиперплоскости строятся по обеим сторонам гиперплоскости, разделяющей классы. Разделяющей гиперплоскостью будет гиперплоскость, максимизирующая расстояние до двух параллельных гиперплоскостей. Алгоритм работает в предположении, что чем больше разница или расстояние между этими параллельными гиперплоскостями, тем меньше будет средняя ошибка классификатора.

Доступным языком:



Идею метода удобно проиллюстрировать на следующем простом примере: даны точки на плоскости, разбитые на два класса (рис. выше). Проведем линию, разделяющую эти два класса (красная линия на рисунке). Далее, все новые точки (не из обучающей выборки) автоматически классифицируются следующим образом:

Точка выше прямой попадает в класс **A**,

Точка ниже прямой — в класс **B**.

5.2 Анализ сложности

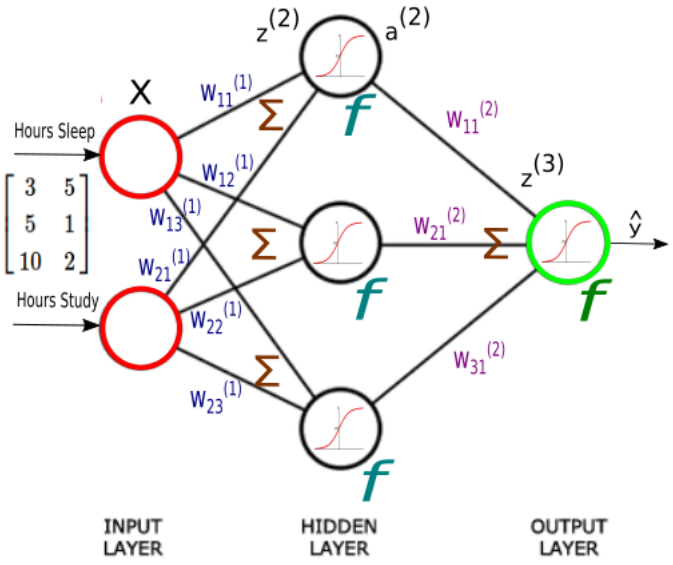
В процедуре обучения классическим алгоритмам требуется высчитать матрицу ядра K, матрица, где каждый элемент представим как $K(x_i, x_j)$, где K - некоторое заранее заданное ядро.

Предполагается, что вычисление матрицы занимает $O(p)$ времени, по крайней мере, это верно для всех наиболее популярных и известных ядер.

Тогда работа алгоритма эквивалентна инвертированию квадратной матрицы размера n, а эта операция имеет сложность $O(n^3)$.

6 Нейронные сети

6.1 Описание алгоритмов



Оценим вычислительную сложность Forward propagation и back propagation.

Для начала рассмотрим момент получения входных данных нейронной сетью, здесь работает алгоритм **forward propagation**.

Прежде всего, нужно ознакомиться с самим алгоритмом. Возьмем такие входные данные $x \in \mathbb{R}^N, x_0 = 1$

Это может быть представлено как вектор, где каждая координата имеет индекс (номер слоя), то есть, $x = a^{(0)}$

Тогда про forward propagation мы можем написать следующее:

$z^{(k)} = \theta^{(k)} a^{(k-1)}$

$z^{(k)} \in \mathbb{R}^{1*(m|\theta^{(k)} \in \mathbb{R}^{m*n})}$

$a^{(k)} = g(z^k)$,
где $g(x)$ - функция активации, которая высчитывается поэлементно. При этом мы знаем, что $a^{(k)}$ имеет такие же размеры, как и $z^{(k)}$

Система условий для **backpropagation**.

$$\text{time}_{\text{error}} = \begin{cases} \alpha \cdot \beta | \nabla_a^{(L)} \in \mathbb{R}^{\alpha \times \beta} & \text{if} k = L \\ \left(\theta^{(k+1)^T} \delta^{(k+1)}\right) \odot g'(z^{(k)}) \end{cases}$$ (стандартная нотация для нейронных сетей)

6.2 Анализ сложности

Forward propagation

На каждом слое вычисляем матричное произведение и функцию активации. Ранее показывали, что матричное произведение имеет сложность $O(n^3)$ и т.к. $g(x)$ функция, которая вычисляется поэлементно, ее время работы равно $O(n)$.

$a = n^{(k)}$

$z = n^{(k-1)}$

$d = n^{(k-2)}$

Далее получим n_{mul} - число всех вычисленных произведений, и n_g - количество раз, когда мы вычисляем функцию активации g.

$$n_{mul} = \sum_{k=2}^{n_{layers}} \left(n^{(k)} n^{(k-1)} n^{(k-2)}\right) + \left(n^{(1)} n^{(0)} 1\right)$$

$$n_g = \sum_{k=1}^{n_{layers}} \left(n^{(k)}\right)$$

Получаем

$$\text{time} = n_{mul} + n_g \Leftrightarrow$$

$$\text{time} = \sum_{k=2}^{n_{layers}} \left(n^{(k)} n^{(k-1)} n^{(k-2)}\right) + \left(n^{(1)} n^{(0)} 1\right) + \sum_{k=1}^{n_{layers}} \left(n^{(k)}\right)$$

Имеем ввиду, что матрицы везде квадратные, количество строк и столбцов совпадает, поэтому $n_{mul} = n_{layers} \cdot n^3$

Если брать в расчет то, что на каждом слое количество нейронов одинаковое, и что количество слоев равно количеству нейронов на каждом слое:

$n_{mul} = O(n \cdot n^3) = O(n^4)$

Аналогично для функции g:

$n_g = n_{layers} \cdot n = O(n^2)$

Итоговая вычислительная сложность forward propagation: $O(n^4 + n^2) \Leftrightarrow$
$O(n^4) \because \forall n \geq 1 | n^4 + n^2 \leq 2n^4$

Back propagation

Если на каждом слое ровно n нейронов:
$$n^2 = O\left(\alpha \cdot \beta | \nabla_a^{(L)} \in \mathbb{R}^{\alpha \times \beta}\right)$$
$$n^3 = O\left(\left(\theta^{(k+1)^T} \delta^{(k+1)}\right) \odot g'(z^{(k)})\right)$$

Итоговое время работы дельта ошибки: $O(\text{time}_{\text{error}}) = n^2 + n^3(L-1)$

Имея ввиду, что всего n слоев, $O(\text{time}_{\text{error}}) = n^4$

Нам нужно просчитать все веса, поэтому: $O(\text{time}_{\text{weights}}) = O(\text{time}_{\text{error}}) + n^3 = n^4$

Используем градиентный спуск: $\text{time}_{\text{gradient descent}} = n_{\text{gradient iterations}} \cdot \text{time}_{\text{weights}}$

Если у градиентного спуска ровно n итераций, получаем $O(\text{time}_{\text{gradient descent}}) = n \cdot n^4 = n^5$

Итоговая вычислительная сложность back propagation равна $O(n^5)$

Единственный способ, как можно увеличить производительность этих алгоритмов, это использование параллельных вычислений, к примеру, вычислять произведение матриц на GPU.

GPU изначально созданы для параллельного вычисления матричных произведений, т.к. 3D геометрия и анимация могут быть представлены, как серия линейных трансформаций (произведения матриц).

7 *Заключение*

В заключении стоит сказать, что сложность вычислений и машинное обучение в современном мире практически не связаны друг с другом, но проанализировать их взаимосвязь определенно стоит. К примеру, у нас есть модель. Обучение на n элементах занимает x минут. Тогда, что произойдет, если мы обучим модель на kn элементах? Если время обучения равняется kx, тогда время обучения линейно. Иногда, это не так. Новое время обучения может быть k^2x. В данном случае, время обучения будет называться квадратичным (благодаря размеру обучающей выборки). Если это время довольно большое для нескольких тысяч элементов в обучающей выборке, не стоит ожидать, что этот метод справится с задачей на миллионе точек.

На самом деле, определить вычислительную сложность алгоритма машинного обучения не так просто, как может показаться, особенно если алгоритм зависит от реализации. Кроме того, свойства данных могут приводить к изменению времени обучения, да и вообще, оно часто зависит от параметров, переданных в алгоритм (гиперпараметров). Ну и последнее, алгоритмы машинного обучения могут быть чрезвычайно сложны и включать в себя композицию других алгоритмов.

Поэтому, я считаю, что задача анализа вычислительной сложности алгоритмов ML является крайне интересной и может привести к неожиданным и практически важным, полезным выводам.

Список литературы

[1] Gilles Louppe *Understanding Random Forests*,

[2] Kasper Fredenslund, *Computational Complexity Of Neural Networks*