Глава 1

Деревья принятия решений

§ 1.1. Зачем всё это надо

Деревья принятия решений (decision trees) обычно используются для решения задач классификации данных или, иначе говоря, для задачи *аппроксимации заданной булевской функции*. Ситуация, в которой стоит применять деревья принятия решений, обычно выглядит так: есть много случаев, каждый из которых описывается некоторым конечным набором дискретных атрибутов («конечный» и «дискретных» — разумеется, ключевые слова), и в каждом из случаев дано значение некоторой (неизвестной) булевской функции, зависящей от этих атрибутов. Задача — создать достаточно экономичную конструкцию, которая бы описывала эту функцию и позволяла классифицировать новые, поступающие извне данные.

Пример 1.1. Постановка задачи.

Предположим, что нас интересует, выиграет ли «Зенит» свой следующий матч. Мы знаем, что это зависит от ряда параметров; перечислять их все — задача безнадёжная, поэтому ограничимся основными:

— выше ли находится соперник по турнирной таблице;
— дома ли играется матч;
— пропускает ли матч кто-либо из лидеров команды;
— идёт ли дождь.

У нас есть некоторая статистика на этот счёт — см. табл. 1.1 (совпадения случайны). А теперь мы хотим понять, выиграет ли «Зенит» (например, нам на него деньги поставить хочется) в следующей игре.

Предположим, что сегодняшний соперник стоит ниже «Зенита», игра происходит на выезде, лидеры на месте, и дождя нет. Такой строки в таблице нет. Как предсказать исход? На этот вопрос могут ответить деревья принятия решений.

Разумеется, если мы сможем обучить компьютер обрабатывать данные и предсказывать результат в таких ситуациях, это будет типичной постановкой задачи машинного обучения. При этом алгоритмы наши должны быть тем лучше, чем больше информации предоставлено. Посмотрим, как деревья принятия решений справляются с такими задачами.

Таблица 1.1. Как играет «Зенит».

Соперник	Играем	Лидеры	Дождь	Победа
Выше	Дома	На месте	Да	Нет
Выше	Дома	На месте	Нет	Да
Выше	Дома	Пропускают	Нет	Да
Ниже	Дома	Пропускают	Нет	Да
Ниже	В гостях	Пропускают	Нет	Нет
Ниже	Дома	Пропускают	Да	Да
Выше	В гостях	На месте	Да	Нет
Ниже	В гостях	На месте	Нет	*???*

§ 1.2. Структура дерева принятия решений

Продолжим рассматривать пример 1.1 и попытаемся построить настоящее дерево принятия решений. В его узлах, не являющихся листьями, находятся атрибуты, по которым различаются случаи. В листьях находятся значения целевой функции. А по рёбрам мы будем спускаться, чтобы классифицировать имеющиеся случаи.

Для начала просто используем атрибуты в порядке, в котором они указаны в таблице. Тогда у нас получится структура, изображённая на рис. 1.1.

Использование дерева принятия решений для ответа на интересующий нас вопрос сводится к тому, чтобы пройтись по дереву сверху вниз и определить, в какой из листьев попадает интересующая нас ситуация.

Пример 1.2. Использование дерева принятия решений.

Вспомним условия примера 1.1. Соперник стоит ниже «Зенита» — следовательно, спускаемся налево, матч проходит в гостях — спускаемся направо и получаем, что «Зенит», судя по нашему дереву, этот матч должен проиграть.

Для того чтобы реализовать машинное обучение, нужно просто видоизменять дерево принятия решений в соответствии со вновь поступающей информацией. Например, если в вышеописанном случае «Зенит» проиграл, то ничего менять бы не понадобилось. Но если бы выиграл, то дерево принятия решений пришлось бы дополнить ещё одним узлом, и оно приняло бы вид, изображённый на рис. 1.2.

Построенное нами дерево далеко не идеально — например, его глубина равна четырём. Мы могли бы взять за основу другой атрибут. Например, давайте попробуем поместить в корень вопрос о том, идёт ли дождь во время матча. В случае, если дождя нет, следующим атрибутом будет положение соперника в турнирной таблице, а в случае, если дождь идёт, мы будем смотреть на то, проходит ли матч дома

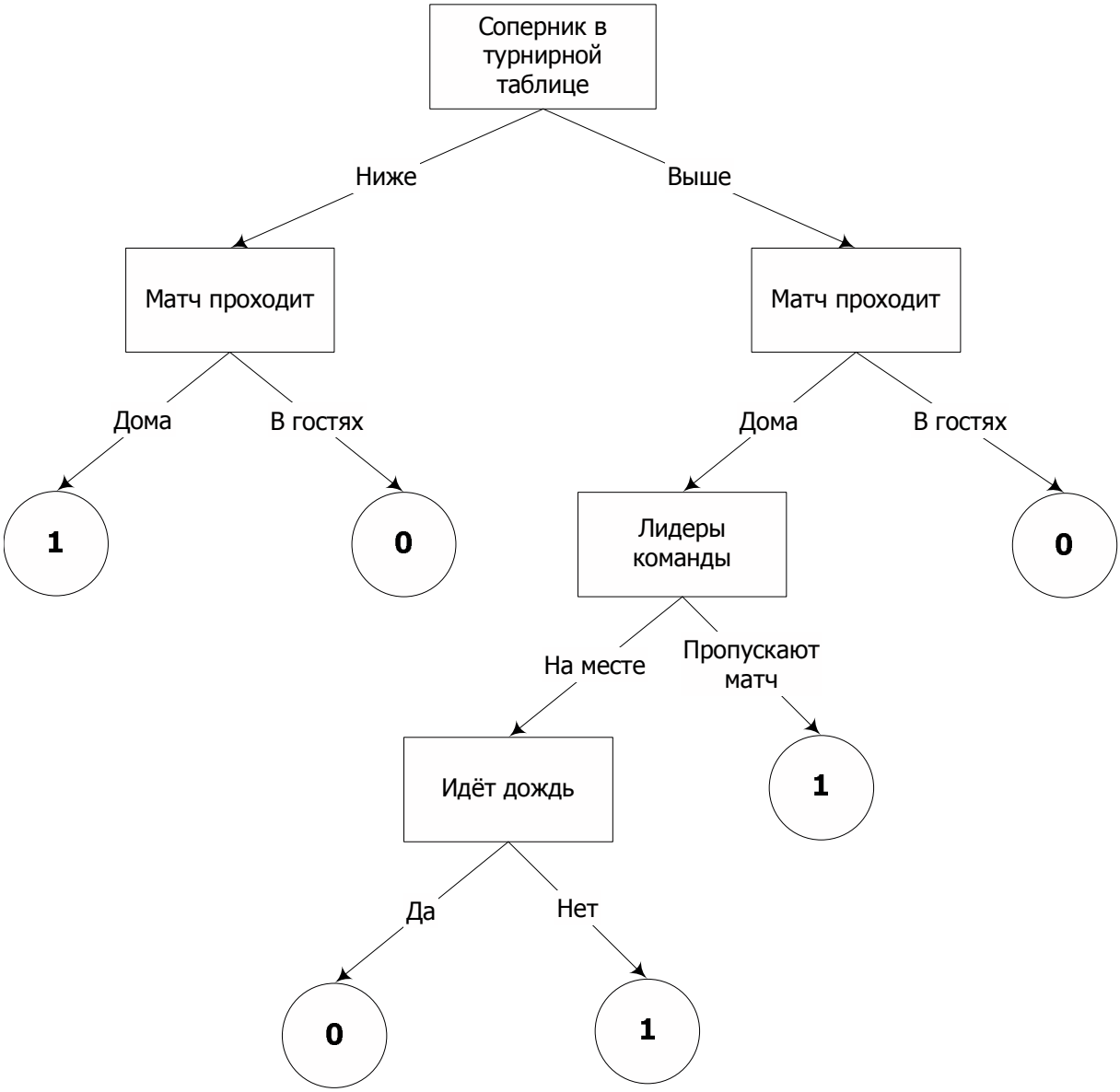

Рис. 1.1. Первый вариант дерева принятия решений.

или в гостях. В таком случае глубина нашего дерева будет равна всего лишь двум (см. рис. 1.3). Легко убедиться, что ни один атрибут сам по себе не может идеально разделить значения функции, поэтому изображённое на рис. 1.3 дерево оптимально (оптимальное дерево, конечно, не обязано быть единственным).

В следующем разделе мы увидим, как можно автоматически получать оптимальные деревья принятия решений.

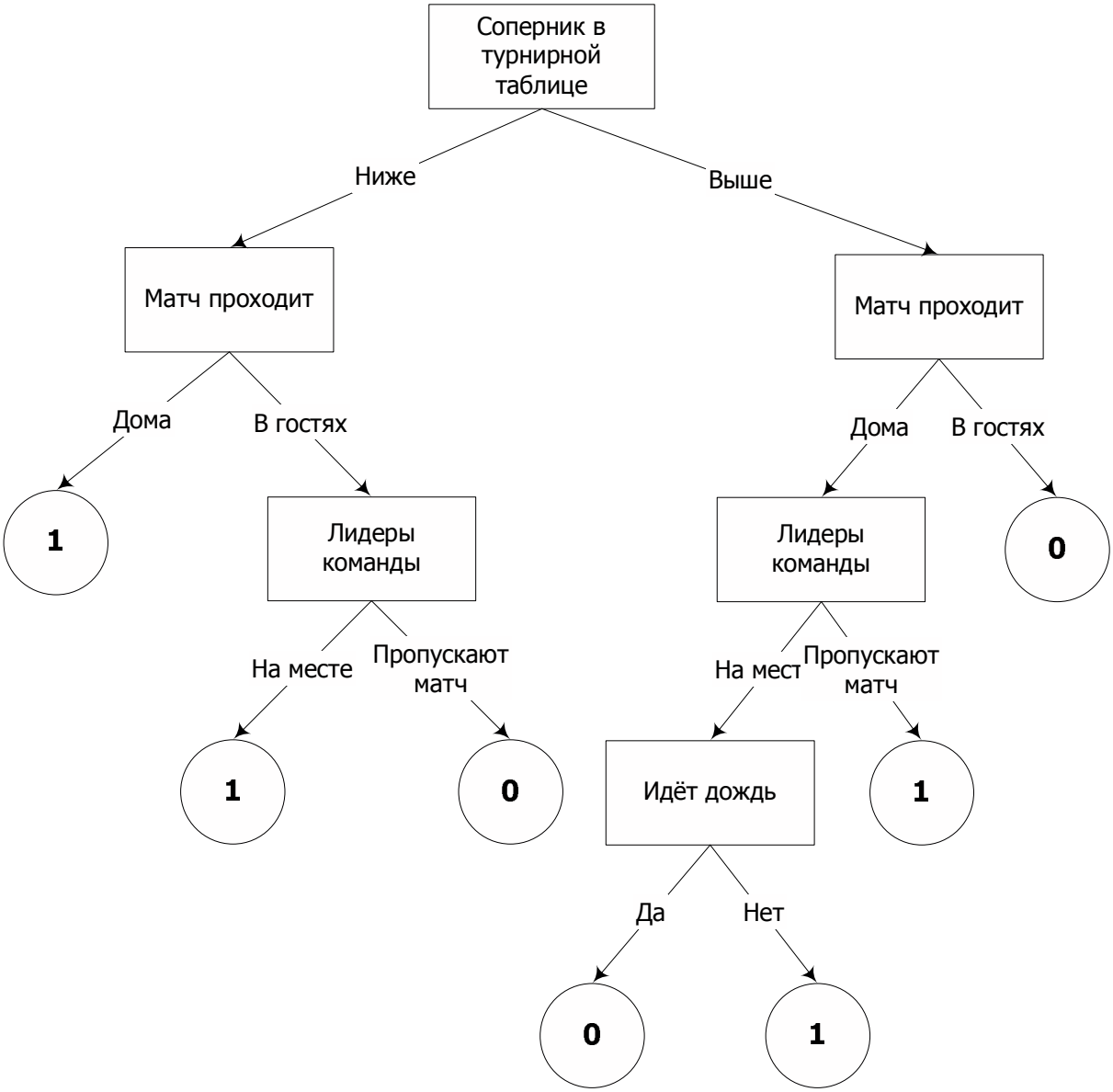

Рис. 1.2. Дерево после добавления ещё одного случая.

§ 1.3. Энтропия и прирост информации

Интуитивно понятно, что для того чтобы получить оптимальные деревья принятия решений, нужно на каждом шаге выбирать атрибуты, которые «лучше всего» характеризуют целевую функцию. Это требование формализуется посредством понятия *энтропии*.

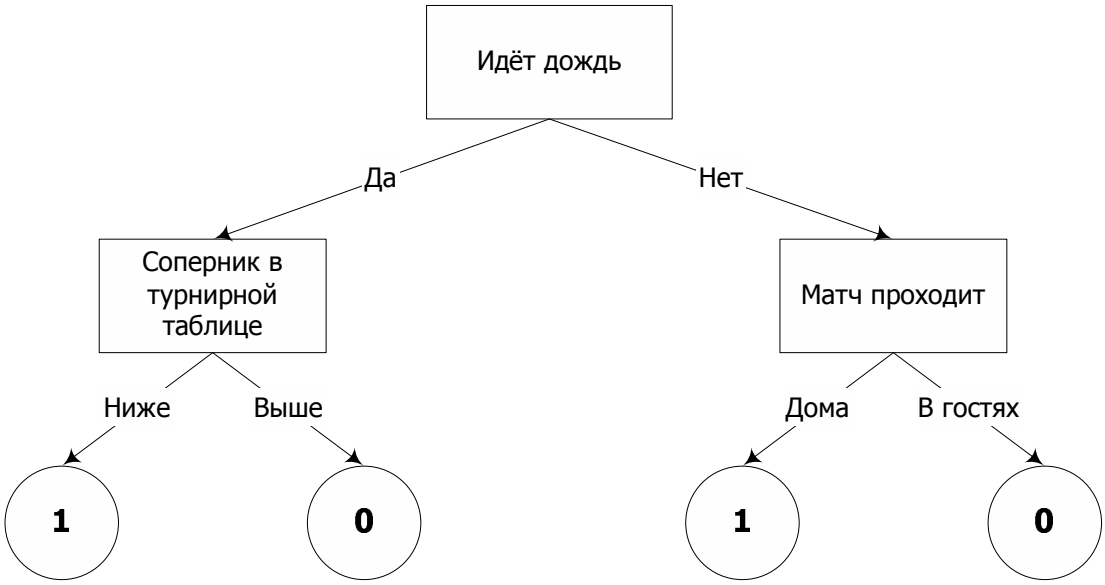

Рис. 1.3. Оптимальное дерево принятия решений.

Определение 1.1. *Предположим, что имеется множество* A *из* n *элементов,* m *из которых обладают некоторым свойством* S*. Тогда* энтропия *множества* A *по отношению к свойству* S *— это*

$$H(A,S) = -\frac{m}{n}\log_2\frac{m}{n} - \frac{n-m}{n}\log_2\frac{n-m}{n}.^{1}$$

Проще говоря, энтропия зависит от пропорции, в которой разделяется множество. По мере возрастания этой пропорции от 0 до $1/2$ энтропия тоже возрастает, а после $1/2$ — симметрично убывает.

Если свойство S не бинарное, а может принимать s различных значений, каждое из которых реализуется в m_i случаях, то энтропия обобщается естественным образом:

$$H(A,S) = -\sum_{i=1}^{s}\frac{m_i}{n}\log\frac{m_i}{n}.$$

Понятие энтропии тесно связано с теорией информации. Грубо говоря, энтропия — это среднее количество битов, которые требуются, чтобы закодировать атрибут S у элемента множества A. Если вероятность появления S равна $1/2$, то энтропия равна 1, и нужен полноценный бит; а если S появляется не равновероятно, то можно закодировать последовательность элементов A более эффективно.

Всё это приводит нас к мысли о том, что при выборе атрибута для классификации нужно выбирать его так, чтобы после классификации энтропия стала как можно

[1] Разумеется, $0\log 0 = 0$.

меньше (свойство S в данном случае — значение целевой булевской функции). Энтропия при этом будет разной в разных потомках, и общую сумму нужно считать с учётом того, сколько исходов осталось в рассмотрении в каждом из потомков. Общепринятое в теории деревьев принятия решений определение будет выглядеть так.

Определение 1.2. *Предположим, что множество A элементов, некоторые из которых обладают свойством S, классифицировано посредством атрибута Q, имеющего q возможных значений. Тогда* прирост информации *(information gain) определяется как*

$$\mathrm{Gain}(A, Q) = H(A, S) - \sum_{i=1}^{q} \frac{|A_i|}{|A|} H(A_i, S),$$

где A_i — множество элементов A, на которых атрибут Q имеет значение i.

На каждом шаге жадный алгоритм должен выбирать тот атрибут, для которого прирост информации максимален.

Пример 1.3. Вычисление энтропии и прироста информации.

Попытаемся определить оптимальный атрибут для примера 1.1. Вычислим исходную энтропию (для максимизации прироста это, конечно, не нужно, но всё же):

$$H(A, \text{Победа}) = -\frac{4}{7}\log_2\frac{4}{7} - \frac{3}{7}\log_2\frac{3}{7} \approx 0.9852.$$

Теперь вычислим приросты информации для различных атрибутов:

$$\mathrm{Gain}(A, \text{Соперник}) = H(A, \text{Победа}) - \frac{4}{7}H(A_{\text{выше}}, \text{Победа}) - \frac{3}{7}H(A_{\text{ниже}}, \text{Победа}) \approx$$
$$\approx 0.9852 - \frac{4}{7}\left(-\frac{1}{2}\log_2\frac{1}{2} - \frac{1}{2}\log_2\frac{1}{2}\right) - \frac{3}{7}\left(-\frac{2}{3}\log_2\frac{2}{3} - \frac{1}{3}\log_2\frac{1}{3}\right) \approx 0.0202.$$

Мы явно выбрали на рис. 1.1 не слишком удачный атрибут для корня дерева...

$$\mathrm{Gain}(A, \text{Играем}) = H(A, \text{Победа}) - \frac{5}{7}H(A_{\text{дома}}, \text{Победа}) - \frac{2}{7}H(A_{\text{в гостях}}, \text{Победа}) \approx 0.4696.$$

$$\mathrm{Gain}(A, \text{Лидеры}) = H(A, \text{Победа}) - \frac{3}{7}H(A_{\text{на месте}}, \text{Победа}) - \frac{4}{7}H(A_{\text{пропускают}}, \text{Победа}) \approx 0.1281.$$

$$\mathrm{Gain}(A, \text{Дождь}) = H(A, \text{Победа}) - \frac{3}{7}H(A_{\text{да}}, \text{Победа}) - \frac{4}{7}H(A_{\text{нет}}, \text{Победа}) \approx 0.1281.$$

Итак, вычисление прироста информации подсказывает нам, что нужно сначала классифицировать по тому, домашний ли матч или гостевой. Это выглядит логично: одна из веток оборвётся сразу («Зенит» в нашем примере ещё не выиграл ни одного гостевого матча), а во второй только в одном случае из пяти результат будет отличаться от других. Однако если мы воспользуемся нашим алгоритмом дальше, то глубина этого дерева будет равна трём: одного атрибута окажется недостаточно, чтобы выделить этот один случай из пяти. О том, как оптимизировать дерево в таких ситуациях, мы поговорим чуть ниже.

§ 1.4. Деревья принятия решений и булевские функции

Мы уже говорили о том, что основную задачу, которую решают деревья принятия решений, можно сформулировать как задачу аппроксимации некоторой не полностью заданной булевской функции. Но и наоборот, дерево принятия решений можно прочесть как булевскую функцию. Она будет естественным образом выражаться в КНФ. Например, дерево на рис. 1.1 соответствует функции

$$\big((\text{Соперник} = \text{Ниже}) \wedge (\text{Играем} = \text{Дома})\big) \vee$$
$$\vee \big((\text{Соперник} = \text{Выше}) \wedge (\text{Играем} = \text{Дома}) \wedge (\text{Лидеры} = \text{Пропускают})\big) \vee$$
$$\vee\big((\text{Соперник} = \text{Выше}) \wedge (\text{Играем} = \text{Дома}) \wedge (\text{Лидеры} = \text{На месте}) \wedge (\text{Дождь} = \text{Нет})\big).$$

А дерево на рис. 1.3 — функции

$$\big((\text{Дождь} = \text{Да}) \wedge (\text{Соперник} = \text{Ниже})\big) \vee \big((\text{Дождь} = \text{Да}) \wedge (\text{Играем} = \text{Дома})\big).$$

И обе эти функции получились вполне естественным путём из одного и того же набора данных.

§ 1.5. Алгоритм

Сейчас мы сведём всё то, о чём до сих пор говорили, в единый рекурсивный алгоритм для построения дерева принятия решений. Этот алгоритм носит название ID3 и был придуман Джоном Р. Квинланом (John R. Quinlan) [**3**].

Алгоритм выписан на рис. 1.4. В нём, по сравнению с предыдущими примерами, встречаются два тривиальных, но важных обобщения. Во-первых, алгоритм обрабатывает ситуацию, когда одному и тому же набору атрибутов соответствуют несколько случаев с разными исходами (например, «Зенит» сыграл в одних и тех же условиях два матча, но один проиграл, а другой выиграл). Тогда, когда мы дойдём до конца дерева, атрибутов, которые могли бы разделить исходы, уже не останется, а исходы всё ещё будут разными. Для этого служит пункт 4 алгоритма — решение будем принимать простым большинством.

Другое обобщение — у атрибута теперь может встретиться несколько возможных вариантов. Например, у атрибута «Дождь» могут быть варианты «Нет», «Слабый» и «Ливень». И, к тому же, может так случиться, что какой-то из этих вариантов не реализуется; в таком случае мы заполняем соответствующий лист в зависимости от того, каких исходов было больше в его предке (за это отвечает пункт 7b).

§ 1.6. Реализация ID3 на языке Python

В этом параграфе мы опишем программную реализацию алгоритма ID3 на языке Python. Программа достаточно большая, но значительную её часть занимает парсинг

$ID3(A, S, \mathcal{Q})$

1. Создать корень дерева.
2. Если S выполняется на всех элементах A, поставить в корень метку 1 и выйти.
3. Если S не выполняется ни на одном элементе A, поставить в корень метку 0 и выйти.
4. Если $Q = \emptyset$, то:
   а) если S выполняется на половине или большей части A, поставить в корень метку 1 и выйти;
   б) если S не выполняется на большей части A, поставить в корень метку 0 и выйти.
5. Выбрать $Q \in \mathcal{Q}$, для которого $\mathrm{Gain}(A, Q)$ максимален.
6. Поставить в корень метку Q.
7. Для каждого значения q атрибута Q:
   а) добавить нового потомка корня и пометить соответствующее исходящее ребро меткой q;
   б) если в A нет случаев, для которых Q принимает значение q (т.е. $|A_q| = 0$), то пометить этого потомка в зависимости от того, на какой части A выполняется S (аналогично пункту 4);
   в) иначе запустить $ID3(A_q, S, \mathcal{Q} \setminus \{Q\})$ и добавить его результат как поддерево с корнем в этом потомке.

Рис. 1.4. Алгоритм ID3.

исходного текстового файла и вывод в файл результатов. Тем не менее, мы приводим её целиком, для того чтобы читатель освоился с работой со строками и текстовыми файлами в Python, а также для того, чтобы привести пример полной, реально работающей программы.

Программа обрабатывает только бинарные атрибуты. Формат входного файла таков (построчно):

— число атрибутов n;
— n строк формата «[Название атрибута] [Положительное значение] [Отрицательное значение]»;
— название целевого атрибута;
— число тестовых примеров m;
— m строк формата «[атрибут$_1$=значение$_1$] ... [атрибут$_n$=значение$_n$]».

Имя файла подаётся в качестве первого параметра функции applyID3; второй её параметр — имя файла, куда программа запишет результат своей работы. Результатов является дерево, уровни которого отделяются табуляциями. Например, входной файл, соответствующий примеру 1.1, будет выглядеть так:

```
5
Opponent Higher Lower
Match Home Away
Leaders Play Pass
Rain Yes No
Victory Yes No
Victory
7
Opponent=Higher Match=Home Leaders=Play Rain=Yes Victory=No
Opponent=Higher Match=Home Leaders=Play Rain=No Victory=Yes
Opponent=Higher Match=Home Leaders=Pass Rain=No Victory=Yes
Opponent=Lower Match=Home Leaders=Pass Rain=No Victory=Yes
Opponent=Lower Match=Away Leaders=Pass Rain=No Victory=No
Opponent=Lower Match=Home Leaders=Pass Rain=Yes Victory=Yes
Opponent=Higher Match=Away Leaders=Play Rain=Yes Victory=No
```

А результат применения applyID3 к этому файлу выглядит так (как мы и предупреждали выше, дерево получится глубины 3):

```
Match=Home
    Leaders=Play
        Rain=Yes
            0
        Rain=No
            1
    Leaders=Pass
        1
Match=Away
    0
```

Теперь — собственно текст программы.

Листинг 1.1. Алгоритм ID3 в бинарном случае.

```
1   def ParseAttributes(infname):
        f = open(infname,'r')
        attr,attrnames,tests = {},[],[]
        attrnum = int(f.readline())
5       for i in xrange(attrnum):
            fWords = f.readline().split()
            attrnames.append(fWords[0])
```

```
            attr[fWords[0]] = [i,fWords[1],fWords[2]]
            attr[fWords[1]] = 1
            attr[fWords[2]] = 0
        num = attr[f.readline().strip()][0]
        testnum = int(f.readline())
        for i in xrange(testnum):
            fWords = f.readline().split()
            test = []
            for j in xrange(attrnum): test.append(0)
            for j in xrange(attrnum):
                attrib = fWords[j][:fWords[j].find('=')]
                value = fWords[j][fWords[j].find('=')+1:len(fWords[j])]
                test[ attr[attrib][0] ] = attr[value]
            tests.append(test)
        f.close()
        return [attrnum,attrnames,attr,tests,num]

    def entropy(tests,num):
        import math
        def log2(x): return math.log(x)/math.log(2)
        neg = float(len(filter(lambda x:(x[num]==0),tests)))
        tot = float(len(tests))
        if ((neg==tot) or (neg==0)): return 0
        return -(neg/tot)*log2(neg/tot)-((tot-neg)/tot)*log2(tot-neg)

    def gain(tests,attrnum,num):
        res = 0
        for i in xrange(2):
            arr = filter(lambda x:(x[attrnum]==i),tests)
            res += entropy(arr,num)*len(arr)/float(len(tests))
        return entropy(tests,num)-res

    def ID3(tests,num,f,tabnum,usedattr,attrnames,attr):
        def findgains(x):
            if usedattr[x]: return 0
            return gain(tests,x,num)
        if (len(tests)==0):
            f.write('\t'*tabnum+'1')
            return
        if len(filter(lambda x:(x[num]==0),tests))>len(filter(lambda x:(x[num]==1),tests)):
            majority = '0'
        else: majority = '1'
```

```
        gains = map(findgains,xrange(len(tests[0])))
        maxgain = gains.index(max(gains))
        if (gains[maxgain]==0):
            f.write('\t'*tabnum+majority+'\n')
            return
        arrpos=filter(lambda x:(x[maxgain]==1),tests)
        arrneg=filter(lambda x:(x[maxgain]==0),tests)
        newusedattr=usedattr
        newusedattr[maxgain]=True
        f.write('\t'*tabnum+attrnames[maxgain]+'='+attr[attrnames[maxgain]][1]+'\n')
        if (len(arrpos)==0):
            f.write('\t'*(tabnum+1)+majority+'\n')
        else:
            ID3(arrpos,num,f,tabnum+1,newusedattr,attrnames,attr)
        f.write('\t'*tabnum+attrnames[maxgain]+'='+attr[attrnames[maxgain]][2]+'\n')
        if (len(arrneg)==0):
            f.write('\t'*(tabnum+1)+majority+'\n')
        else:
            ID3(arrneg,num,f,tabnum+1,newusedattr,attrnames,attr)

    def applyID3(infname,outfname):
        bigarr = ParseAttributes(infname)
        attrnum,attrnames,attr,tests,num = bigarr[0],bigarr[1],bigarr[2],bigarr[3],bigarr[4]
        f = open(outfname,'w')
        usedattr=[]
        for i in xrange(attrnum): usedattr.append(i==num)
        ID3(tests,attrnum-1,f,0,usedattr,attrnames,attr)
```

§ 1.7. Проблемы с критерием прироста информации. Gain Ratio и GINI Index

У критерия прироста информации есть одна существенная проблема. Дело в том, что прирост информации часто будет выбирать атрибуты, у которых больше всего значений.

Пример 1.4. Проблема с критерием прироста информации.

Предположим, что в нашей таблице игр «Зенита» были записаны ещё и даты игр, причём у каждого матча, что вполне логично, своя дата. Вычислим в таком случае прирост информации:

$$\text{Gain}(A, \text{Дата}) = H(A, \text{Победа}) - \sum_{i=1}^{n} \frac{1}{n} H(A_{\text{Дата}=i}, \text{Победа}) = H(A, \text{Победа}),$$

потому что в каждой из веток только один случай, и энтропия в каждой ветке равна нулю. Таким образом, прирост информации в этом случае будет максимальным из возможных. Но полученное дерево принятия решений будет, конечно, абсолютно бесполезным — даже если бы дата матча имела отношение к силе команды «Зенит», всё равно уже прошедшие даты никогда не повторятся.

Что же делать? Очевидно, нужно модифицировать критерий прироста информации так, чтобы он смог справиться с этой проблемой. Надо как-то уменьшать желательность атрибута с ростом количества его значений.

В статье [3] был предложен критерий *Gain Ratio*. Идеологически он учитывает не только количество информации, требуемое для записи результата, но и количество информации, требуемое для разделения по текущему атрибуту. Эта поправка выглядит как

$$\mathrm{SplitInfo}(A, Q) = -\sum_{i=1}^{q} \frac{|A_q|}{|A|} \log_2 \frac{|A_q|}{|A|},$$

а сам критерий — как максимизация величины

$$\mathrm{GainRatio}(A, Q) = \frac{\mathrm{Gain}(A, Q)}{\mathrm{SplitInfo}(A, Q)}.$$

Ещё один часто используемый критерий [?] — так называемый *индекс Гини*. Для набора тестов A и свойства S, имеющего s значений, этот индекс вычисляется как

$$\mathrm{Gini}(A, S) = 1 - \sum_{i=1}^{s} \frac{|A_i|}{|A|}.$$

Соответственно, для набора тестов A, атрибута Q, имеющего q значений, и целевого свойства S, имеющего s значений, индекс вычисляется следующим образом:

$$\mathrm{Gini}(A, Q, S) = \mathrm{Gini}(A, S) - \sum_{j=1}^{q} \frac{|A_j|}{|A|} \mathrm{Gini}(A_j, S).$$

Индекс Гини отчасти компенсирует уклон критерия прироста информации в сторону выбора самых «развесистых» атрибутов. Однако в остальном они очень похожи; проведённое теоретическое исследование [4] показало, что индекс Гини и критерий прироста информации не согласуются только на приблизительно 2% возможных случаев.

§ 1.8. Оверфиттинг

Когда дерево принятия решений строится посредством алгоритма ID3 или аналогичного ему, полученное дерево всегда удовлетворяет *всем* имеющимся данным.

Однако на практике в результате часто получаются слишком детализированные деревья, и количество ошибок при дальнейшем использовании построенного дерева, наоборот, растёт. Это явление называется *оверфиттингом* (overfitting).

Пример 1.5. Когда возникает оверфиттинг.

Эффект оверфиттинга легко проиллюстрировать на следующем примере. Предположим, что игра нашего многострадального «Зенита» дома в ясную погоду вообще ни от чего больше не зависит: «Зенит» просто выигрывает в 90% случаев. Но среди исходных данных имеется одно домашнее поражение «Зенита» в ясную погоду. Тогда ID3 старательно учтёт все (не имеющие никакого отношения к сути дела) «причины» этого поражения и будет в дальнейшем предсказывать, что «Зенит» проиграет в аналогичных ситуациях. На самом же деле «Зенит», конечно, будет выигрывать всё с той же вероятностью 90%; таким образом, дерево принятия решений не будет справляться со своими задачами.

Можно, конечно, просто искусственно останавливать рост дерева в глубину. Но когда, на каком этапе? Обычно, чтобы избежать оверфиттинга, используют так называемое *обрезание* (pruning), то есть позволяют дереву вырасти дальше, а затем обрезают лишние ветки. Обрезание дерева в некотором узле заключается в том, что поддерево с корнем в этом узле удаляется из дерева, а в узел помещается метка с тем исходом, которых в этом узле большинство. Осталось только понять, какие ветки следует обрезать.

Разобьём, как в § 1.7, исходные данные на множество, на котором дерево будет строиться, и множество, на котором мы будем дерево тестировать. По первой части построим дерево принятия решений. Теперь у нас есть дерево и набор эталонных тестов, правильные ответы на которые нам известны.

Изложим один из возможных методов. Главной мерой успеха для нас будет то, насколько хорошо дерево справляется с тестами. Соответственно, можно просто рассмотреть каждую вершину как кандидата на обрезание и проверить, станет ли сокращённое дерево лучше справляться с тестами. На каждом шаге мы будем выбирать такую вершину, обрезание которой максимально улучшает поведение дерева принятия решений. Эти вершины будем оставлять обрезанными и продолжать до тех пор, пока обрезание любой вершины не будет приводить к более слабому результату на эталонных тестах.

Есть и другие методы борьбы с оверфиттингом...

§ 1.9. Упражнения и задачи

1. Нарисовать деревья принятия решений, соответствующие функциям:
   а) $x \vee (y \wedge \bar{z})$;

б) $(x \wedge \bar{y}) \vee (y \wedge \bar{z} \wedge t)$;
в) $(x \vee y) \wedge (\bar{y} \vee z)$.

2. Философский вопрос. ID3 в среднем предпочитает короткие гипотезы, небольшие деревья принятия решений. Иначе говоря, ID3 в каком-то смысле реализует бритву Оккама. Чем это хорошо? Часто говорят, что множество деревьев из трёх узлов гораздо меньше, чем множество деревьев из десяти узлов, поэтому найти подходящую под имеющиеся данные гипотезу из трёх узлов более удивительно и, следовательно, более полезно, чем из десяти. Какие логические изъяны есть в этом рассуждении?
3. В § 1.8 описан жадный алгоритм, на каждом шаге выбирающий «оптимальную» вершину, т.е. вершину, обрезание которой максимально улучшает поведение дерева принятия решений на эталонных тестах. Изменится ли результат, если позволить на каждом шаге выбирать любую вершину, хоть как-нибудь улучшающую поведение? Если нет, докажите, если да, приведите конкретный пример.
4. Модифицировать реализацию ID3 из листинга 1.1 так, чтобы она могла обрабатывать атрибуты с несколькими значениями.
5. Добавить в предыдущую реализацию автоматическое тестирование дерева на некоторой части входных данных и выбор наилучшего дерева в соответствии с этим критерием.
6. Добавить в предыдущую реализацию защиту от оверфиттинга посредством обрезания веток, в которых количество ошибок при этом уменьшается. Выбирать наилучшее дерево среди обрезанных.