Обучение

ИЦ «Гевисста»

Артем Груздев

Прогнозное моделирование в R и Python

Модуль 5. Построение случайного леса с помощью пакета R randomForest

Москва — 2017

СОДЕРЖАНИЕ

Модуль 5 Построение случайного леса с помощью пакета R randomForest

Лекция 5.1. Построение ансамбля деревьев классификации

5.1.1. Подготовка данных

Данные, которыми мы воспользуемся для построения случайного леса деревьев классификации, записаны в файле *Response.csv*. Исходная выборка содержит записи о 30259 клиентах, классифицированных на два класса: 0 — отклика нет (17170 клиентов) и 1 — отклик есть (13089 клиентов). По каждому наблюдению (клиенту) фиксируются следующие переменные (характеристики):

- номинальный предиктор *Ипотечный кредит [mortgage]*;
- номинальный предиктор *Страхование жизни [life_ins]*;
- номинальный предиктор *Кредитная карта [cre_card]*;
- номинальный предиктор *Дебетовая карта [deb_card]*;
- номинальный предиктор *Мобильный банк [mob_bank];*
- номинальный предиктор *Текущий счет [curr_acc]*;
- номинальный предиктор *Интернет-доступ к счету [internet]*;
- номинальный предиктор *Индивидуальный займ [perloan]*;
- номинальный предиктор *Наличие сбережений [savings]*;
- номинальный предиктор *Пользование банкоматом за последнюю неделю [atm_user]*;
- номинальный предиктор *Пользование услугами онлайн-маркетплейса за последний месяц [markpl]*;
- количественный предиктор *Возраст [age]*;
- количественный предиктор *Давность клиентской истории [cus_leng]*;
- номинальная зависимая переменная *Отклик на предложение новой карты [response]*.

Мы уже работали с этими данными в рамках лекция 3.2. *Построение и интерпретация дерева классификации CART*, поэтому ограничимся тем, что приведем программный код, выполняющий необходимые преобразования.

```
# загружаем данные
data <- read.csv2("C:/Trees/Response.csv")

# выполняем необходимые преобразования
data[, -c(12:13)] <- lapply(data[, -c(12:13)], factor)
set.seed(42)
data$random_number <- runif(nrow(data),0,1)
development <- data[which(data$random_number > 0.3), ]
holdout <- data[ which(data$random_number <= 0.3), ]
development$random_number <- NULL
holdout$random_number <- NULL
```

5.1.2. Построение модели и получение OOB оценки качества

Подготовив данные, необходимо загрузить пакет `randomForest`, который будет использоваться для построения случайного леса.

```
# загружаем пакет randomForest
library(randomForest)
```

Поскольку случайный лес использует рандомизацию для генерирования выборок и отбора переменных, необходимо установить стартовое значение генератора случайных чисел, чтобы получить воспроизводимые результаты.

```
# задаем стартовое значение генератора случайных
# чисел для воспроизводимости результатов
set.seed(152)
```

Чтобы построить модель случайного леса, необходимо вызвать функцию `randomForest`. Функция `randomForest` имеет общий вид:

```
randomForest(formula, data, ntree, mtry, classwt, cutoff,
             nodesize, maxnodes, do.trace=FALSE, importance=FALSE)
```

где

`formula=`	Задает формулу в формате *Зависимая переменная ~ Предиктор1 + Предиктор2 + Предиктор3* + др.
`data=`	Задает таблицу данных для анализа. Если таблица данных названа `data`, можно просто указать `data`
`ntree=`	Задает количество деревьев в ансамбле. Значение по умолчанию 500
`mtry=`	Задает m – количество переменных (признаков), случайно отбираемых при разбиении. Значение по умолчанию различно для классификации и регрессии. Для классификации $m = \sqrt{M}$, для регрессии $m = M/3$, где M – общее число предикторов в наборе

`classwt=`	Задает априорные вероятности. Обратите внимание, что они вовсе не обязательно должны быть в сумме равны 1, поскольку алгоритмом предусмотрена нормализация. Используется только для классификации
`strata=`	Задает переменную-фактор, которая используется для стратифицированного семплинга. Работает вместе с параметром `samplesize`

`samplesize=`	Задает количество случайно извлекаемых наблюдений для каждого класса. Работает вместе с параметром `strata`. Например, если мы зададим
	
	`model<-randomForest(response ~., development,`
	`strata=development$response, sampsize=c(100, 50))`
	или
	
	`model<-randomForest(response ~., development,`
	`strata=development$response,`
	`sampsize=c('NoResponse'=100, 'Response'=50))`
	
	это будет обозначать, что мы случайным образом отбираем 100 примеров из класса 0 и 50 примеров из класса 1 (с возвращением) для построения каждого дерева

`cutoff=`	Задает пороговое значение вероятности для прогнозирования класса. По умолчанию равен $1/k$, где k – количество классов. Например, в бинарной классификации он равен 0,5. Побеждает класс, который получает вероятность больше 0,5. Используется только для классификации
`norm.votes=`	Задает нормализацию частот голосов деревьев, т.е. количество голосов деревьев, поданных за каждый класс, делится на количество деревьев и умножается на 100. По умолчанию задано значение `TRUE`. Если задано значение `FALSE`, выводятся частоты голосов в исходном виде. Используется только для классификации
`nodesize=`	Задает минимальный размер терминальных узлов. Большее значение приводит к построению деревьев меньшего размера (и снижает время вычислений). Значение по умолчанию различно для классификации и регрессии. Для классификации оно равно 1, для регрессии – 5
`maxnodes=`	Задает максимальное количество терминальных узлов, которое могут иметь деревья в случайном лесе. Если не задано, то деревья строятся вплоть до достижения максимально возможного количества терминальных

	узлов (ограничением служит только значение `nodesize`). Если задано значение, превышающее максимально возможное количество терминальных узлов, выдается предупреждение
`do.trace=`	`TRUE` – выводит по каждому дерева в ансамбле для классификации – ошибки классификации по методу OOB (общую ошибку классификации и ошибки классификации по категориям зависимой переменной), для регрессии – среднеквадратичную ошибку и процент необъясненной дисперсии зависимой переменной по методу OOB. Также можно задать конкретное значение. Например, `do.trace=100` обозначает, что по каждым построенным 100 деревьям будет выводится общая ошибка классификации и ошибки классификации по категориям зависимой переменной
`localImp=`	Задает вычисление индивидуальных важностей предикторов. Если задано значение `TRUE`, будет вычислена матрица p на n, содержащая индивидуальные важности, каждый элемент $[i, j]$ – это значение важности i-той переменной для j-го наблюдения
`importance=`	`TRUE` – вычисляет значения важности для предикторов. Обратите внимание, что при запросе важностей для деревьев классификации будет выведена таблица с `nclass + 2` столбцами, а для деревьев регрессии – таблица с двумя столбцами. Для классификации `nclass` столбцы – это меры важности на основе усредненного уменьшения правильности, вычисленные по наблюдениям одного конкретного класса. `nclass+1`-й столбец – это усредненное уменьшение правильности, вычисленное по всем классам. Последний столбец – усредненное уменьшение неоднородности (используется мера Джини). Для регрессии первый столбец – это усредненное увеличение среднеквадратичной ошибки, а второй столбец – усредненное уменьшение неоднородности (используется сумма квадратов остатков)

Поскольку наша зависимая переменная *response* является категориальной, будет построен случайный лес деревьев классификации. Итак, построим случайный лес деревьев классификации по всем исходным предикторам на обучающей выборке, вычислив важность предикторов.

```
# строим случайный лес деревьев классификации
model<-randomForest(response ~., development, importance=TRUE)
```

Построив модель, выведем информацию о ее качестве.

```
# выводим информацию о качестве модели
print(model)
```

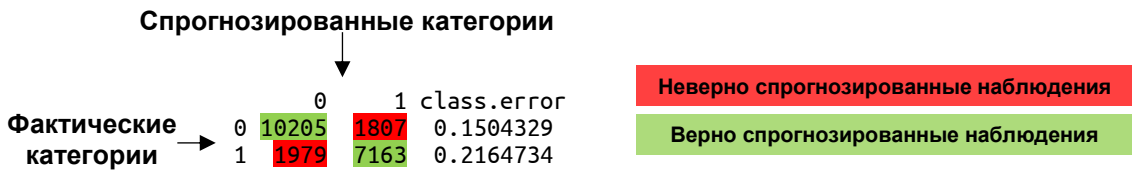
Сводка 5.1. Оценка качества модели (используется ошибка классификации по методу OOB)

```
Call:
 randomForest(formula = response ~ ., data = development, importance = TRUE)
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 3

        OOB estimate of  error rate: 17.9%
Confusion matrix:
      0    1 class.error
0 10205 1807   0.1504329
1  1979 7163   0.2164734
```

Параметр `Number of trees` показывает количество деревьев в ансамбле. `No. of variables tried at each split` показывает mtry – количество случайно отбираемых предикторов при каждом разбиении. Наш набор состоит из 13 предикторов, по умолчанию для классификации используется корень от общего количества предикторов, поэтому $mtry = \sqrt{13} \approx 3{,}6$, округляем в меньшую сторону и получаем 3. `OOB estimate of error rate` – это общая ошибка классификации по методу OOB или количество наблюдений, ошибочно классифицированных деревьями по out-of-bag выборкам, взятое от общего числа наблюдений. Она равна (1807+1979)/21154=0,179 или 17,9%. `Confusion matrix` – матрица OOB ошибок, допущенных при классификации. Рассмотрим матрицу OOB ошибок подробнее.

Сводка 5.2. Матрица OOB ошибок



Согласно матрице OOB ошибок (сводка 5.2), из 12012 неоткликнувшихся клиентов (сумма наблюдений по строке `0`) классификатор правильно выявил 10205, а 1807 неоткликнувшихся клиентов ошибочно отнес к откликнувшимся. Поэтому ошибка классификации по методу OOB для неоткликнувшихся клиентов

составила 1807/12012=0,15 или 15%. Из 9142 откликнувшихся клиентов (сумма наблюдений по строке 1) классификатор правильно выявил 7163, а 1979 откликнувшихся клиентов ошибочно отнес к тем, кто не отреагировал на маркетинговое предложение. Поэтому ошибка классификации по методу OOB для откликнувшихся клиентов составила 1979/9142=0,216 или 21,6%. Для получения матрицы OOB ошибок еще можно воспользоваться следующим программным кодом:

```
# матрицу ошибок по методу OOB
# можно еще вывести так
table(development$response, predict(model))
```

Теперь выведем график зависимости ошибок классификации по методу OOB от количества деревьев в ансамбле (рис. 5.1).

```
# строим график зависимости ошибок классификации по методу OOB
# от количества случайно отбираемых предикторов
plot(model)
```

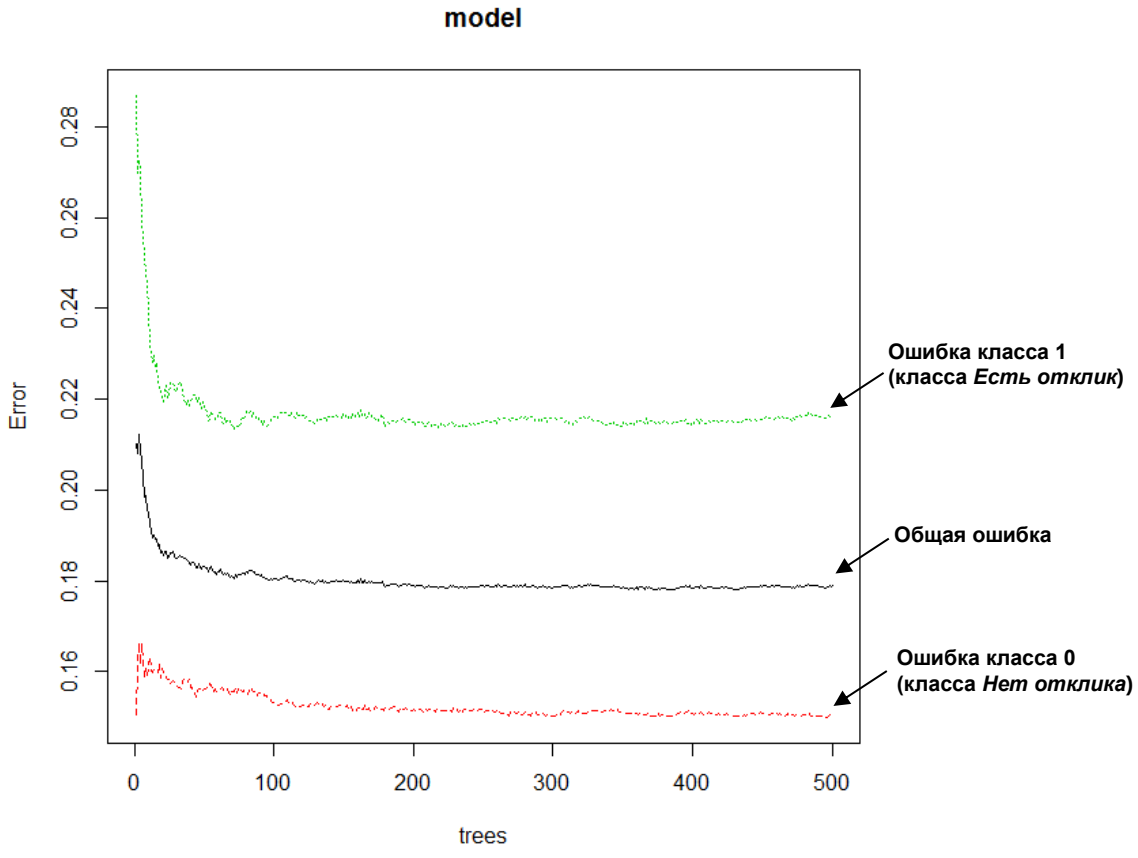


Рис. 5.1 График зависимости OOB ошибок классификации от количества деревьев в ансамбле

График показывает зависимость ошибок классификации по методу OOB (речь идет об общей доле ошибочно классифицированных наблюдений и доле ошибочно классифицированных наблюдений по каждому классу зависимой переменной) от количества деревьев в ансамбле. Видно, что

ошибка классификации уменьшается по мере построения определенного числа деревьев, затем перестает уменьшаться и стабилизируется.
Повысить качество модели можно также за счет настройки оптимального количества случайно отбираемых предикторов. Для этого необходимо воспользоваться функцией `tuneRF`. Функция `tuneRF` имеет общий вид:

```
tuneRF(x, y, mtryStart, ntreeTry, plot=TRUE)
```

где

`x=`	Задает матрицу или таблицу данных с предикторами
`y=`	Задает вектор значений зависимой переменной (для классификации – фактор, для регрессии – числовой вектор)
`mtryStart=`	Задает стартовое значение `mtry` (по умолчанию используется то же самое значение, что и в модели случайного леса)
`ntreeTry=`	Задает количество деревьев, используемых при настройке `mtry`
`plot=`	Печатает график зависимости OOB ошибок от количества переменных, случайно отбираемых в качестве кандидатов при каждом разбиении

Итак, попробуем вычислить оптимальное значение `mtry`.

```
# настраиваем оптимальное значение mtry
set.seed(152)
tuneRF(development[,1:13], development[,14], ntreeTry=500, trace=FALSE)
```

В результате получаем сводку, в которой приводятся ошибки классификации по методу OOB для разных значений `mtry`. Для задачи классификации в качестве таких значений используются корень от общего количества предикторов, поделенный пополам, корень от общего количества предикторов, удвоенный корень от общего количества предикторов.

Сводка 5.3. Зависимость ошибки классификации по методу OOB от количества случайно отбираемых предикторов для разбиения

```
-0.01553858 0.05
-0.0524098 0.05
      mtry  OOBError
2.OOB    2 0.1822823
3.OOB    3 0.1794932
6.OOB    6 0.1889004
```

Кроме того, для визуализации результатов генерируется график зависимости правильности модели от количества переменных для разбиения (рис. 5.2).

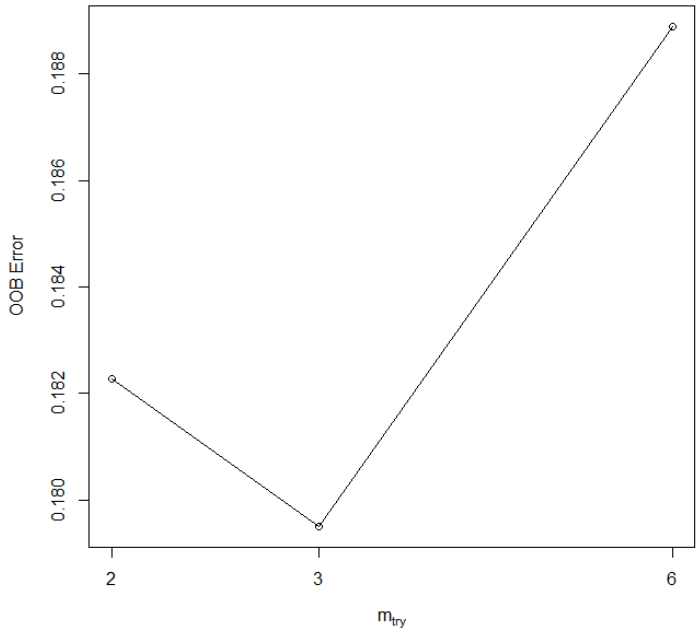


Рис. 5.2 График зависимости ошибки классификации, вычисленной по методу OOB, от количества переменных для разбиения

Наименьшее значение OOB ошибки наблюдается для `mtry=3`. Оно не отличается от значения `mtry=3`, автоматически выбранного в ходе построения случайного леса.

5.1.3. Получение информации о деревьях случайного леса

Пакет `randomForest` позволяет получить информацию о каждом дереве случайного леса. Это можно сделать с помощью функции `getTree`. Функция `getTree` имеет общий вид:

```
getTree(rfobj, k, labelVar=FALSE)
```

где

`rfobj=`	Задает модельный объект
`k=`	Задает номер дерева случайного леса
`labelVar=`	Выводит метки переменных разбиения и метки спрогнозированных классов

Функция `getTree` возвращает матрицу (или датафрейм, если `labelVar=TRUE`), состоящий из 6 столбцов и *n* строк, где *n* – общее количество узлов в дереве. Шесть столбцов включают в себя:

- `left daughter` – номер строки, соответствующий левому дочернему узлу (`0`, если узел является терминальным);

- `right daughter` – номер строки, соответствующий правому дочернему узлу (`0`, если узел является терминальным);
- `split var` – переменная, использованная для разбиения узла (`0`, если узел является терминальным);
- `split point` – лучшая точка расщепления (`0`, если узел является терминальным);
- `status` – статус узла: терминальный (`-1`) или нет (`1`);
- `prediction` – прогноз для узла (`0`, если узел не является терминальным, для бинарной классификации 1 будет означать отрицательный класс, 2 – положительный класс).

Все наблюдения, у которых значение количественного предиктора меньше или равно точке расщепления, относятся в левый дочерний узел. Все наблюдения, у которых значение количественного предиктора больше точки расщепления, относятся в правый дочерний узел. Для категориального предиктора разделяющее значение представлено целочисленным значением, двоичное разложение которого определяет, куда будут отправлены категории – в левый или правый дочерний узел. Например, если предиктор имеет 4 категории, точка расщепления будет равна 13. Если выполнить двоичное разложение числа 13, мы получим 1011, потому что $13 = 1 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 1 \times 2^3$ (обратите внимание, любое число, кроме нуля, возведенное в нулевую степень, будет равно единице), видим, что наблюдения с категориями 1, 3 и 4 отправлены в левый узел, наблюдения с категорией 2 отправлены в правый узел.

Давайте выведем информацию о последних 15 узлах дерева №1 случайного леса, не отображая метки переменных, использованных для разбиения, и метки спрогнозированных классов.

```
# выведем информацию о последних 15 узлах
# дерева №1 случайного леса, не отображая
# метки переменных расщепления и метки
# спрогнозированных классов
info_tree1 <- getTree(model, k=1, labelVar=F)
tail(info_tree1, 15)
```

Сводка 5.4. Последние 15 узлов дерева №1 случайного леса (метки переменных разбиения и метки спрогнозированных классов не выводятся)

	left daughter	right daughter	split var	split point	status	prediction
1021	0	0	0	0.0	-1	2
1022	1032	1033	9	1.0	1	0
1023	0	0	0	0.0	-1	1
1024	0	0	0	0.0	-1	2
1025	0	0	0	0.0	-1	2
1026	0	0	0	0.0	-1	1
1027	0	0	0	0.0	-1	2
1028	0	0	0	0.0	-1	2
1029	1034	1035	12	51.5	1	0
1030	0	0	0	0.0	-1	1
1031	0	0	0	0.0	-1	2
1032	0	0	0	0.0	-1	2
1033	0	0	0	0.0	-1	1
1034	0	0	0	0.0	-1	2
1035	0	0	0	0.0	-1	1

номер строки, соответствующий левому дочернему узлу

номер строки, соответствующий правому дочернему узлу

переменная расщепления

точка расщепления

статус узла (-1 – терминальный)

прогноз для узла (1 – отрицательный класс, 2 – положительный класс)

Мы можем сразу сказать, что дерево состоит из 1035 узлов. Возьмем узел 1021, у него нет дочерних узлов (значение `0` в столбцах `left_daughter` и `right_daughter`), нет переменной расщепления (значение `0` в столбце `split_var`), нет точки расщепления (значение `0` в столбце `split_point`). Это объясняется тем, что он является терминальным (значение `-1` в столбце `status`). Поскольку узел является терминальным, для него вычисляется прогноз (в столбце `prediction` значение `1` обозначает отрицательный класс или класс *Нет отклика*, значение `2` – положительный класс или класс *Есть отклик*). Теперь рассмотрим узел 1022. Его левый дочерний узел расположен в 1032-й строке, правый дочерний узел расположен в 1033-й строке, переменной расщепления стала переменная с индексом `9` (нумерация начинается с 1), то есть бинарная переменная *savings*. Точка расщепления будет равна `1`, категория 1 будет отправлена в левый узел, а категория 2 – в правый узел. Узел 1022 не является терминальным (значение `1` в столбце `status`) и поэтому для него прогноз не записывается (значение `0` в столбце `prediction`).

Теперь выведем информацию о последних 15 узлах дерева №1 случайного леса, отобразив метки переменных, использованных для разбиения, и метки спрогнозированных классов.

```
# выведем информацию о последних 15 узлах
# дерева №1 случайного леса, отобразив
# метки переменных расщепления и метки
# спрогнозированных классов
info_tree1 <- getTree(model, k=1, labelVar=T)
tail(info_tree1, 15)
```

Сводка 5.5. Последние 15 узлов дерева №1 случайного леса (с выводом меток переменных разбиения и меток спрогнозированных классов)

	left daughter	right daughter	split var	split point	status	prediction
1021	0	0	<NA>	0.0	-1	1
1022	1032	1033	savings	1.0	1	<NA>
1023	0	0	<NA>	0.0	-1	0
1024	0	0	<NA>	0.0	-1	1
1025	0	0	<NA>	0.0	-1	1
1026	0	0	<NA>	0.0	-1	0
1027	0	0	<NA>	0.0	-1	1
1028	0	0	<NA>	0.0	-1	1
1029	1034	1035	age	51.5	1	<NA>
1030	0	0	<NA>	0.0	-1	0
1031	0	0	<NA>	0.0	-1	1
1032	0	0	<NA>	0.0	-1	1
1033	0	0	<NA>	0.0	-1	0
1034	0	0	<NA>	0.0	-1	1
1035	0	0	<NA>	0.0	-1	0

номер строки, соответствующий левому дочернему узлу

номер строки, соответствующий правому дочернему узлу

переменная расщепления

точка расщепления

статус узла (-1 – терминальный)

прогноз для узла (0 – отрицательный класс, 1 – положительный класс)

5.1.4. Важности предикторов

Теперь выведем важности предикторов:

```
# выводим важности предикторов
importance(model)
```

Сводка 5.6. Важность переменных

	0	1	MeanDecreaseAccuracy	MeanDecreaseGini
mortgage	24.617204206	-2.500776	24.001405	117.04399
life_ins	19.757225517	2.748487	23.308543	97.15749
cre_card	33.941396685	28.623893	46.825814	459.41408
deb_card	5.193765254	8.158659	10.635994	63.48066
mob_bank	35.287746430	15.943149	38.418814	143.44418
curr_acc	38.270624615	17.627133	46.052467	282.88614
internet	-1.083234424	30.406735	28.994728	60.70255
perloan	-1.730117992	6.174774	3.299139	52.97052
savings	0.001114391	26.678706	29.229889	67.80267
atm_user	43.577559721	85.786279	86.132606	898.75617
markpl	18.871003112	38.330576	37.550018	420.53910
age	50.874401907	81.729307	94.138122	1161.73868
cus_leng	50.405824472	82.491582	88.871511	1945.48159

В сводке 5.6 первые два столбца – это меры важности на основе усредненного уменьшения качества модели, вычисленные по наблюдениям одного конкретного класса. Третий столбец – это усредненное уменьшение качества модели, вычисленное по всем классам. Метрикой качества выступает правильность. Последний столбец – усредненное уменьшение неоднородности (используется мера Джини). Напомним, что важность на основе уменьшения правильности позволяет судить о силе взаимосвязи между предиктором и зависимой переменной.

Здесь мы отвечаем на вопрос, насколько сильно упадет качество модели, если мы переставим значения предиктора, по сути устранив взаимосвязь между предиктором и зависимой переменной. Важность на основе уменьшения неоднородности позволяет сделать вывод о том, насколько данный предиктор позволяет уменьшить неоднородность (для ансамбля деревьев классификации – уменьшить неоднородность распределения категорий зависимой переменной в узлах-потомках, для ансамбля деревьев регрессии – уменьшить разброс значений зависимой переменной относительно ее среднего значения в узлах-потомках).
Наибольший интерес представляют последние два столбца. Важность на основе уменьшения правильности показывает, что сильнее всего на зависимую переменную влияют предикторы *Возраст [age]*, *Давность клиентской истории [cus_leng]* и *Пользование банкоматом за последнюю неделю [atm_user]*. Что касается уменьшения неоднородности, то здесь наиболее важными предикторами стали переменные *Давность клиентской истории [cus_leng]* и *Возраст [age]*.

Для наглядности визуализируем важности (рис. 5.3).

```
# выводим график важности предикторов
varImpPlot(model)
```

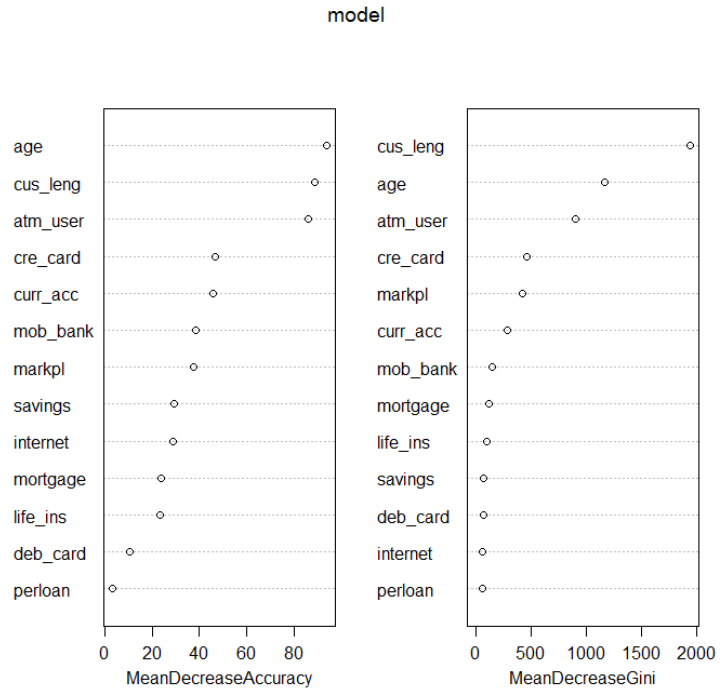


Рис. 5.3 График важности переменных

Кроме вычисления важности переменной можно посмотреть частоту использования переменной в качестве предиктора разбиения по ансамблю в целом. Для этого необходимо воспользоваться функцией `varUsed`. Функция `varUsed` имеет общий вид:

```
varUsed(x, by.tree=FALSE, count=TRUE)
```

где

x=	Задает модельный объект
by.tree=	Задает способ вывод частот использования переменных в качестве предикторов разбиения: по ансамблю в целом (значение FALSE) или по каждому дереву случайного леса (значение TRUE). По умолчанию задано значение FALSE
count=	Задает вывод частот использования переменных в качестве предикторов разбиения. По умолчанию задано значение TRUE

```
# вычисляем частоты использования переменных
# в качестве предикторов разбиения
freq <- varUsed(model, by.tree=FALSE, count=TRUE)
# извлекаем названия предикторов
names <- colnames(development[,-14])
# сопоставляем названия предикторов
# с частотами
names(freq) <- names
# выводим результаты сопоставления
freq
```

Сводка 5.7. Частоты использования переменных

```
mortgage life_ins cre_card deb_card mob_bank curr_acc internet  perloan  savings
   24627    15005    16907    37883    23568    18179    29243    33254    31567

atm_user   markpl      age cus_leng
   13017    19038   106547    27174
```

5.1.5. Графики частной зависимости

Теперь поглубже заглянем в нашу модель «черного ящика». Проанализируем взаимосвязи между зависимой переменной и некоторыми ключевыми предикторами. В этом нам помогут графики частной зависимости. Напомню, что график частной зависимости показывает, как значение интересующего предиктора влияет на прогнозы модели при том, что все остальные переменные фиксируются и рассматриваются как константы. По оси x откладываются значения интересующего предиктора, а по оси y – прогнозы модели. Для количественной зависимой переменной таким прогнозом будет усредненное значение зависимой переменной. Для категориальной зависимой переменной прогнозом становится разность между логарифмом доли голосов, поданных деревьями за интересующий класс зависимой переменной, и усредненной суммой логарифмов голосов,

поданных деревьями за каждый класс. Эта разность вычисляется по формуле:

$$f(x) = log[p_k(x)] - \frac{1}{K}\sum_{j=1}^{K} log\left[p_j(x)\right]$$

где:
x – предиктор, для которого строится график частной зависимости;
K – количество классов;
k – интересующий класс;
p_j – доля голосов, поданных за класс j.

Чтобы построить график частной зависимости, необходимо вызвать функцию `partialPlot`. Функция `partialPlot` имеет общий вид:

```
partialPlot(x, pred.data, x.var, which.class, plot=TRUE)
```

где

`x=`	Задает модельный объект
`pred.data=`	Задает таблицу данных для анализа. Если таблица данных названа `data`, можно просто указать `data`
`x.var=`	Задает интересующий предиктор
`which.class=`	Задает интересующий класс, по умолчанию используется первый класс. Обратите внимание, нумерация классов начинается с `0` (т.е. `0` означает первый класс, `1` – второй класс и т.д.). Для бинарной классификации класс `0` будет означать отрицательный класс, класс `1` будет означать положительный класс
`plot=`	`TRUE` – выводит график частной зависимости

Построим график частной зависимости для предиктора *Возраст [age]*. Обратите внимание, что интересующим классом будет класс 1 (класс *Есть отклик*).

```
# строим график частной зависимости для переменной age,
# интересующий класс - класс 1 (класс Есть отклик)
# значение по оси ординат - разность между логарифмом
# доли голосов, поданных деревьями за интересующий класс
# зависимой переменной, и усредненной суммой логарифмов
# голосов, поданных деревьями за каждый класс
partialPlot(model, development, age, 1)
```

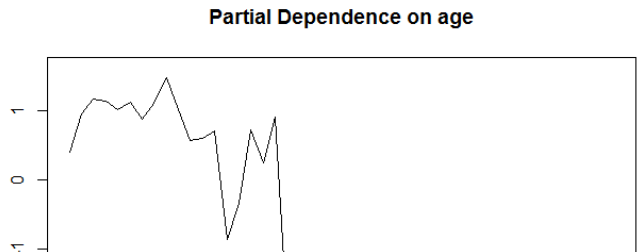


Рис. 5.4 График частной зависимости для переменной age (интересующий класс *Есть отклик*)

На рис. 5.4 видно, что в целом более высокие значения возраста слабее влияют на прогнозирование класса *Есть отклик* (меньшее число деревьев голосует за класс *Есть отклик* при высоких значениях возраста). При этом предполагается, что все остальные предикторы фиксируются и рассматриваются как константы. При работе с графиками частной зависимости придерживайтесь следующих правил интерпретации. Отрицательные значения по оси y означают, что интересующий класс менее вероятен для данного значения предиктора (меньшее количество деревьев склоняется в пользу интересующего класса). Положительные значения по оси y говорят о том, что интересующий класс более вероятен для данного значения предиктора (большее количество деревьев склоняется в пользу интересующего класса). Нулевое значение означает, что данное значение предиктора не влияет на прогнозирование интересующего класса.

Теперь построим график частной зависимости для предиктора *Давность клиентской истории [cus_leng]*.

```
# строим график частной зависимости для переменной cus_leng,
# интересующий класс - класс 1 (класс Есть отклик)
partialPlot(model, development, cus_leng, 1)
```

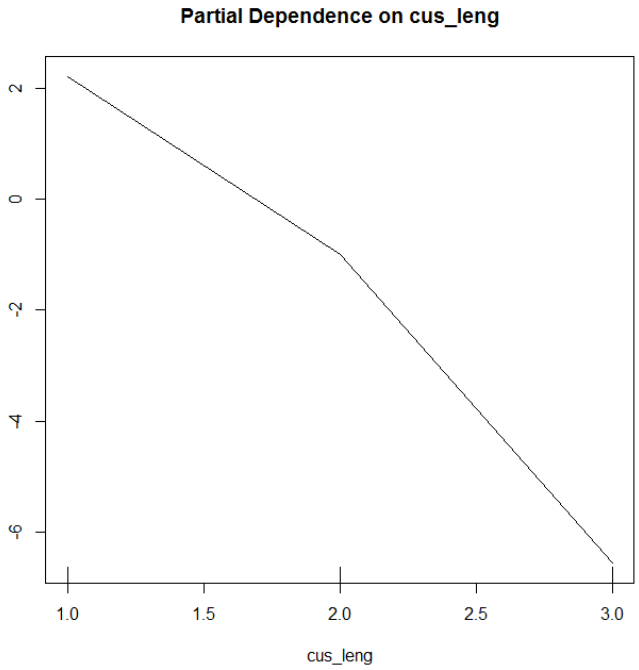


Рис. 5.5 График частной зависимости для переменной cus_leng (интересующий класс *Есть отклик*)

На рис. 5.5 картина более очевидна. Мы видим практически линейную взаимосвязь между давностью клиентской истории и откликом. Более высокие значения переменной *Давность клиентской истории [cus_leng]* в гораздо меньшей степени влияют на прогнозирование класса *Есть отклик*.

Наконец, построим график частной зависимости для бинарного предиктора *Пользование банкоматом за последнюю неделю [atm_user]*.

```
# строим график частной зависимости для переменной atm_user,
# интересующий класс - класс 1 (класс Есть отклик)
partialPlot(model, development, atm_user, 1)
```

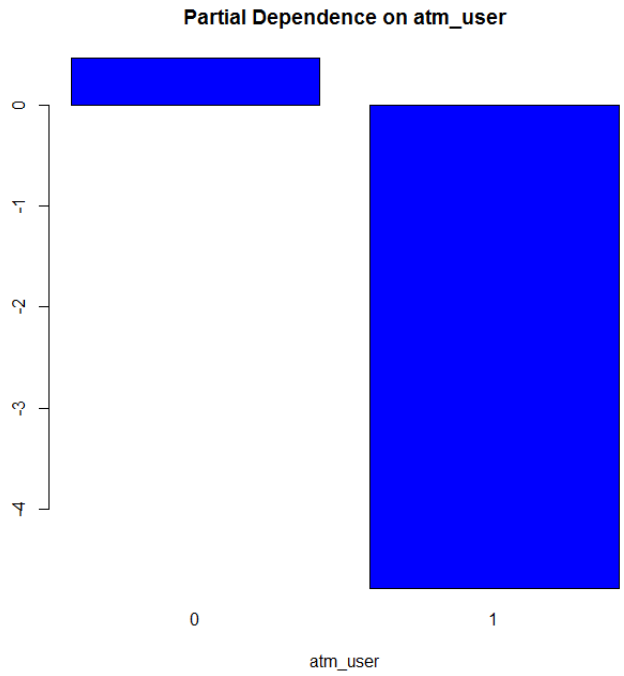


Рис. 5.6 График частной зависимости для переменной atm_user (интересующий класс *Есть отклик*)

На рис. 5.6 видно, что класс *Есть отклик* более вероятен для значения предиктора *Пользование банкоматом за последнюю неделю [atm_user]*, равного 0.

Выше уже говорилось, что в случае бинарной классификации график частной зависимости для одного класса зависимой переменной является зеркальным отражением графика частной зависимости для другого класса. Проверим это. Построим график частной зависимости для бинарного предиктора *Пользование банкоматом за последнюю неделю [atm_user]*, только теперь интересующим классом будет класс *Нет отклика*.

```
# строим график частной зависимости для переменной atm_user,
# интересующий класс - класс 0 (класс Нет отклика)
partialPlot(model, development, atm_user, 0)
```

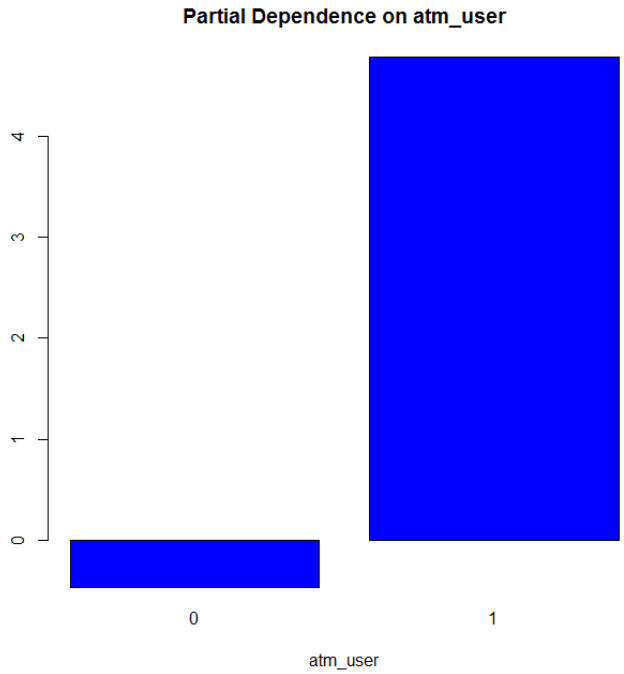


Рис. 5.7 График частной зависимости для переменной atm_user (интересующий класс *Нет отклика*)

Рис. 5.7 подтверждает вышесказанное.

5.1.6. Вычисление вероятностей классов

Теперь переходим к вычислению вероятностей классов. Для этого нам потребуется функция `predict`. Функция `predict` имеет общий вид

```
predict(object, data, type="response",
        norm.votes=TRUE, predict.all=FALSE,
        proximity=FALSE,
        nodes=FALSE)
```

где

`object=`	Задает модельный объект

`data=`	Задает таблицу данных для вычисления прогнозов. При наличии аргумента, задающего выборку, значения (классы) зависимой переменной, вероятности классов зависимой переменной для каждого наблюдения выборки прогнозируются обычным способом.
	выборка
	↓
	`prob_dev <- predict(model, holdout, type="prob")`
	`prob_dev <- predict(model, holdout, type="response")`

	Например, у нас есть наблюдение из выборки новых данных. Мы предъявляем наше наблюдение каждому дереву, которое построено по бутстреп-выборке (бутстреп-выборки в свою очередь сгенерированы на основе обучающей выборки). Для задачи классификации дерево проверяет наблюдение на соответствие своим правилам классификации, вычисляет согласно этим правилам листовые вероятности классов и соответствующий класс. И так по каждому дереву классификации. В итоге побеждает класс, за который проголосовало большинство деревьев (в бинарной классификации побеждает класс, за который проголосовало больше половины деревьев), а доли деревьев, проголосовавших за тот или иной класс, становятся итоговыми вероятностями классов. Для задачи регрессии дерево проверяет наблюдение на соответствие своим правилам прогнозирования, вычисляет среднее значение зависимой переменной в листе. И так по каждому дереву регрессии. В итоге усредняем наши средние значения и получаем итоговое среднее значение.
	Если аргумент, задающий выборку, будет пропущен, то для каждого наблюдения обучающей выборки будут спрогнозированы вероятности классов и класс зависимой переменной по методу OOB.
	
	`prob_dev <- predict(model, type="prob")`
	`prob_dev <- predict(model, type="response")`
	
	Например, у нас есть наблюдение обучающей выборки. Мы предъявляем наше наблюдение каждому дереву, построенному по out-of-bag выборке – бутстреп-выборке, в котором отсутствовало данное наблюдение. Для задачи классификации дерево проверяет наблюдение на соответствие своим правилам классификации, вычисляет согласно этим правилам листовые вероятности классов и соответствующий класс. И так по каждому дереву классификации, построенному по out-of-bag выборке. В итоге побеждает класс, за который проголосовало большинство деревьев, построенных по out-of-bag выборкам, а доли таких деревьев, проголосовавших за тот или иной класс, становятся итоговыми вероятностями классов. Для задачи регрессии дерево проверяет наблюдение на

	соответствие своим правилам прогнозирования, вычисляет среднее значение зависимой переменной в листе. И так по каждому дереву регрессии, построенному по out-of-bag выборке. В итоге усредняем наши средние значения, полученные по out-of-bag выборкам, и получаем итоговое среднее значение.
`type=`	Задает тип прогнозов. Для классификации можно задать значения: • `"prob"` – будут вычислены вероятности классов зависимой переменной; • `"response"` – будут спрогнозированы классы зависимой переменной; • `"vote"` – будут вычислены частоты голосов деревьев по каждому классу. По умолчанию они выводятся в виде процентных долей и совпадают с вероятностями классов. Если необходимо вывести голоса деревьев в виде исходных частот, при построении модели для параметра `norm.votes` нужно задать значение `FALSE`, а при работе с функцией `predict` задать `norm.votes=TRUE`). Для регрессии можно задать значения: • `"response"` – будут спрогнозированы значения зависимой переменной.
`norm.votes=`	Задает нормализацию частот голосов деревьев, т.е. количество голосов деревьев, поданных за каждый класс, делится на количество деревьев и умножается на 100. По умолчанию задано значение `TRUE` (частоты голосов выводятся в виде процентных долей). Используется только для классификации.
`proximity=`	Задает вычисление близостей. В итоге будет возвращена сводка, состоящая из двух разделов: • `predicted`, в котором будут записаны спрогнозированные классы, вероятности классов или частоты голосов (в зависимости от значения, заданного для параметра `type`); • `proximity`, в котором будет записана матрица близостей. Используется только для классификации.
`predict.all=`	Задает вывод прогнозов каждого дерева леса. В итоге будет возвращена сводка, состоящая из двух разделов: • `aggregate`, в котором будут записаны спрогнозированные значения или классы,

	вероятности классов или частоты голосов (в зависимости от значения, заданного для параметра `type`); • `individual`, в котором будет записана матрица со спрогнозированными значениями или классами, вычисленными отдельными деревьями.
`nodes=`	Задает вычисление номеров терминальных узлов. В итоге возвращает матрицу $k \times n$, где k – количество наблюдений, n – количество деревьев в ансамбле

Давайте спрогнозируем вероятности классов для наблюдений обучающей выборки обычным методом и по методу OOB, а затем запишем спрогнозированные вероятности в объекты `prob_dev` и `prob_dev_oob` соответственно.

```
# вычисляем вероятности классов для обучающей выборки
# обычным методом
prob_dev <- predict(model, development, type="prob")
# вычисляем вероятности классов для обучающей выборки
# по методу OOB
prob_dev_oob <- predict(model, type="prob")
```

Выведем спрогнозированные вероятности для последних 5 наблюдений обучающей выборки, полученные по обычному методу.

```
# выводим вероятности для последних 5 наблюдений
# обучающей выборки, вычисленные по обычному методу
tail(prob_dev, 5)
```

Сводка 5.8. Спрогнозированные вероятности по последним 5 наблюдениям обучающей выборки (обычный метод)

```
          0     1
30254 0.986 0.014
30256 0.392 0.608
30257 1.000 0.000
30258 1.000 0.000
30259 1.000 0.000
```

Выведем теперь спрогнозированные вероятности для последних 5 наблюдений обучающей выборки, полученные по методу OOB.

```
# выводим вероятности для последних 5 наблюдений
# обучающей выборки, вычисленные по методу OOB
tail(prob_dev_oob, 5)
```

Сводка 5.9. Спрогнозированные вероятности по последним 5 наблюдениям обучающей выборки (метод OOB)

```
              0          1
30254 0.9732620 0.02673797
30256 0.5989848 0.40101523
30257 1.0000000 0.00000000
```

```
30258 1.0000000 0.00000000
30259 1.0000000 0.00000000
```

5.1.7. Оценка дискриминирующей способности модели с помощью ROC-кривой

Теперь оценим дискриминирующую способность модели случайного леса на обучающей и контрольной выборках, построив ROC-кривые и вычислив значения AUC.

```
# загружаем пакет pROC для построения ROC-кривых
library(pROC)
# строим ROC-кривую для обучающей выборки (на основе
# вероятностей, вычисленных обычным способом)
roc_dev<-plot(roc(development$response, prob_dev[,2], ci=TRUE), percent=TRUE,
              print.auc=TRUE, col="#1c61b6")
# вычисляем вероятности классов для контрольной выборки
prob_hold <- predict(model, holdout, type="prob")
# добавляем ROC-кривую для контрольной выборки
roc_hold<-plot(roc(holdout$response, prob_hold[,2], ci=TRUE), percent=TRUE,
          print.auc=TRUE, col="#008600", print.auc.y= .4, add=TRUE)
# создаем легенды к ROC-кривым
legend("bottomright", legend=c("Обучающая выборка (обычный метод)",
                               "Контрольная выборка"),
       col=c("#1c61b6", "#008600"), lwd=2)
```

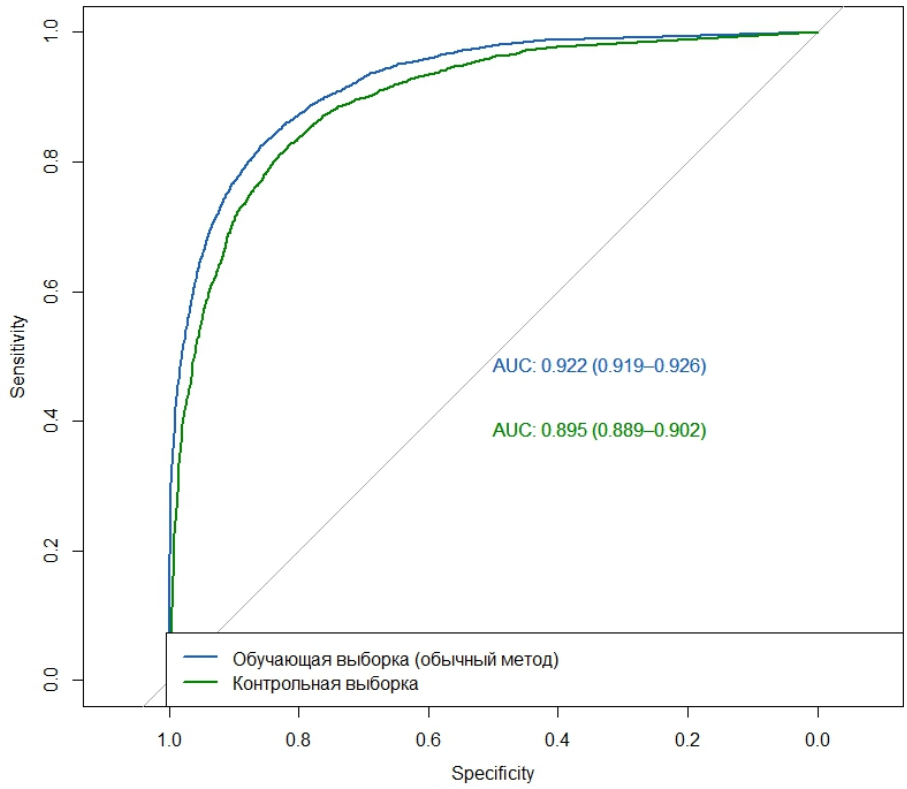


Рис. 5.8 ROC-кривые для обучающей и контрольной выборок, вероятность положительного класса для обучающей выборки вычислена обычным способом

Как видно на рис. 5.8, дискриминирующая способность модели случайного леса на контрольной выборке ниже, чем на обучающей

выборке, наблюдается эффект переобучения. При этом вероятность положительного класса для обучающей выборки была вычислена обычным способом. Теперь построим ROC-кривую на основе вероятности положительного класса для обучающей выборки, вычисленной по методу OOB, и снова сравним с ROC-кривой для контрольной выборки.

```
# строим ROC-кривую для обучающей выборки (на основе
# вероятностей, вычисленных по способу OOB)
roc_dev<-plot(roc(development$response, prob_dev_oob[,2], ci=TRUE), percent=TRUE,
              print.auc=TRUE, col="#1c61b6")
# добавляем ROC-кривую для контрольной выборки
roc_hold<-plot(roc(holdout$response, prob_hold[,2], ci=TRUE), percent=TRUE,
         print.auc=TRUE, col="#008600", print.auc.y= .4, add=TRUE)
# создаем легенды к ROC-кривым
legend("bottomright", legend=c("Обучающая выборка (метод OOB)",
                               "Контрольная выборка"),
       col=c("#1c61b6", "#008600"), lwd=2)
```

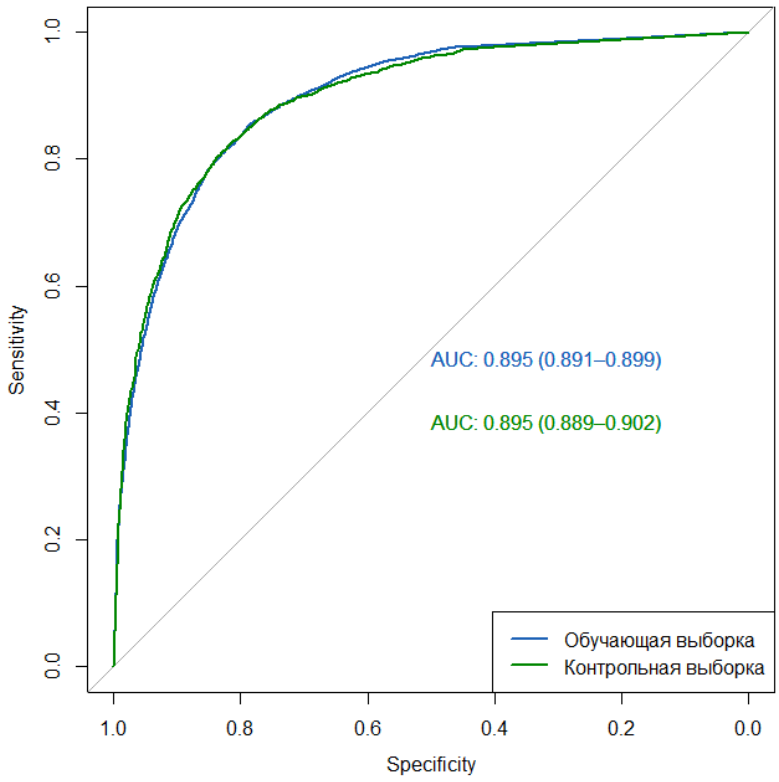


Рис. 5.9 ROC-кривые для обучающей и контрольной выборок, вероятность положительного класса для обучающей выборки вычислена по методу OOB

Теперь мы видим, что дискриминирующая способность на обучающей выборке оценена более «пессимистично». Как правило, при работе с незначительно зашумленными данными OOB оценка качества модели практически совпадает с оценкой качества модели на контрольной выборке, что в принципе позволяет даже отказаться от разбиения данных на обучающую и контрольную выборки и перекрестной проверки.

Однако мы не рекомендуем этого делать, потому что при решении задач с сильно разбалансированными классами зависимой переменной и большим количеством шумовых объектов OOB оценка качества модели все же имеет тенденцию быть более оптимистичной по сравнению с оценкой качества на контрольной выборке.

А теперь сравним дискриминирующую способность модели случайного леса с дискриминирующей способностью модели дерева CART на контрольной выборке.

```
# загружаем пакет rpart
library(rpart)
# подгоняем модель CART
set.seed(42)
model_cart <-rpart(response ~., development)
# записываем вероятности, спрогнозированные деревом CART
# для контрольной выборки, в объект prob_hold_cart
prob_hold_cart <- predict(model_cart, holdout, type="prob")
# визуализируем обе ROC-кривые
rf<-plot(roc(holdout$response, prob_hold[,2], ci=TRUE),
             percent=TRUE, print.auc=TRUE, col="#1c61b6")
cart<-plot(roc(holdout$response, prob_hold_cart[,2], ci=TRUE), percent=TRUE,
           print.auc=TRUE, col="#008600", print.auc.y= .4, add=TRUE)
# создаем легенды к ROC-кривым
legend("bottomright", legend=c("Случайный лес", "Дерево CRT"),
       col=c("#1c61b6", "#008600"), lwd=2)
```

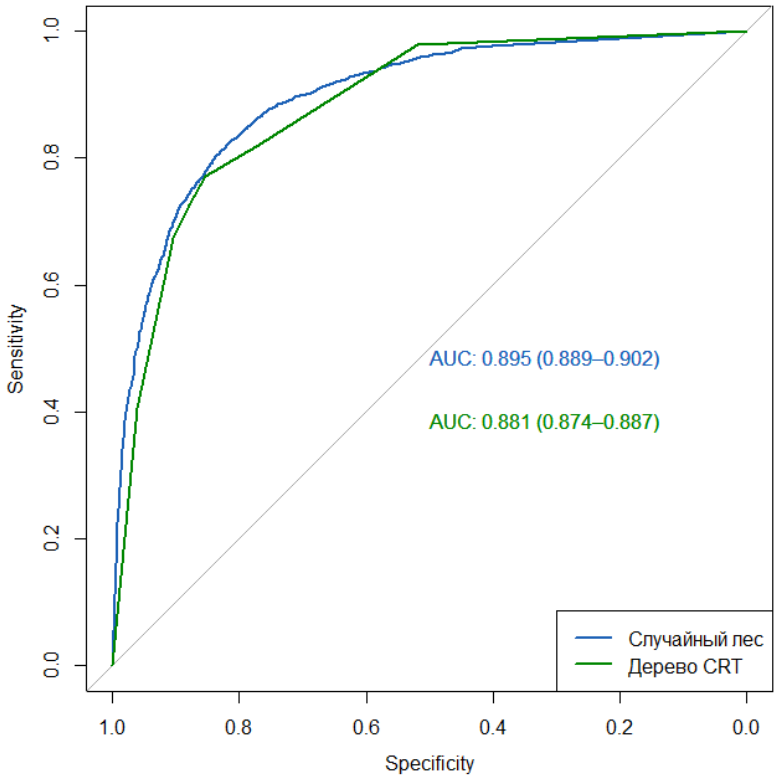


Рис. 5.10 Сравнение ROC-кривых модели случайного леса и модели CART на контрольной выборке

На рис. 5.10 видно, что модель случайного леса имеет более высокую дискриминирующую способность по сравнению с моделью CART. Кроме того, сравнительный анализ доверительных интервалов AUC показывает,

что применение случайного леса дало статистически значимое улучшение дискриминирующей способности.

5.1.8. Получение спрогнозированных классов зависимой переменной

Теперь спрогнозируем для каждого наблюдения класс зависимой переменной. Поскольку вероятности – это числа с плавающей точкой, довольно редко бывает ситуация, когда они обе будут точно равны 0.5. Однако, если это произойдет, то прогноз будет осуществлен случайным образом. Поэтому для получения воспроизводимых результатов классификации рекомендуется задать стартовое значение генератора случайных чисел.

```
# задаем стартовое значение генератора
# случайных чисел
set.seed(152)
```

Для каждого наблюдения обучающей выборки спрогнозируем класс зависимой переменной. Сделаем это обычным способом.

```
# вычисляем классы зависимой переменной
для обучающей выборки обычным способом
resp_dev <- predict(model, development, type="response")
```

Выведем спрогнозированные классы зависимой переменной по последним 5 наблюдениям обучающей выборки (сводка 5.10).

```
# выводим классы зависимой переменной
# для последних 5 наблюдений обучающей
# выборки, вычисленные по обычному методу
tail(resp_dev, 5)
```

Сводка 5.10. Спрогнозированные классы зависимой переменной по последним 5 наблюдениям обучающей выборки, прогнозы получены обычным способом

```
30254 30256 30257 30258 30259
    0     1     0     0     0
Levels: 0 1
```

А теперь вычислим матрицу ошибок для обучающей выборки, используя полученные прогнозы.

```
# выводим матрицу ошибок для обучающей выборки
# на основе классов, вычисленных обычным методом
table(development$response, resp_dev)
```

Сводка 5.11. Матрица ошибок для обучающей выборки, прогнозы получены обычным способом

Полученная матрица ошибок (сводка 5.11) отличается от той, что приведена в сводке 5.2. Это обусловлено тем, что сейчас мы получили ошибку классификации по обычному методу, используя бутстреп-выборки (каждое наблюдение классифицировалось с использованием деревьев, построенных по всем бутстреп-выборкам), а для вычисления ошибки классификации по методу OOB использовались прогнозы, которые вычислялись по out-of-bag выборкам (то есть тогда каждое наблюдение классифицировалось с использованием деревьев, построенных по бутстреп-выборкам, в которых оно отсутствовало). Ошибка классификации, вычисленная обычным методом, равна (1554+1761)/21154=0.157 или 15.7% (сравните с ошибкой классификации по методу OOB, равной 17.9%). Как правило, ошибка классификации, вычисленная по всем бутстреп-выборкам, меньше ошибки классификации, вычисленной по out-of-bag выборкам. Таким образом, используя метод OOB, мы получаем более реалистичную оценку качества классификации.

Теперь спрогнозируем класс зависимой переменной для каждого наблюдения контрольной выборки и, как в случае с обучающей выборкой, вычислим матрицу ошибок для контрольной выборки.

```
# задаем стартовое значение генератора
# случайных чисел
set.seed(152)
# вычисляем классы зависимой переменной
для контрольной выборки
resp_hold <- predict(model, holdout, type="response")
# выводим матрицу ошибок для контрольной выборки
table(holdout$response, resp_hold)
```

Сводка 5.12. Матрица ошибок для контрольной выборки

```
  resp_hold
       0    1
  0 4381  777
  1  852 3095
```

Общая ошибка классификации для контрольной выборки равна (777+852)/(4381+777+852+3095)=1629/9105=0.179 или 17.9%. Обратите внимание, что она совпадает с ошибкой классификации по методу OOB, также равной 17.9%.

5.1.9. График зазора прогнозов

В пакете `randomForest` прогнозы модели можно визуализировать.

```
plot(margin(model))
```

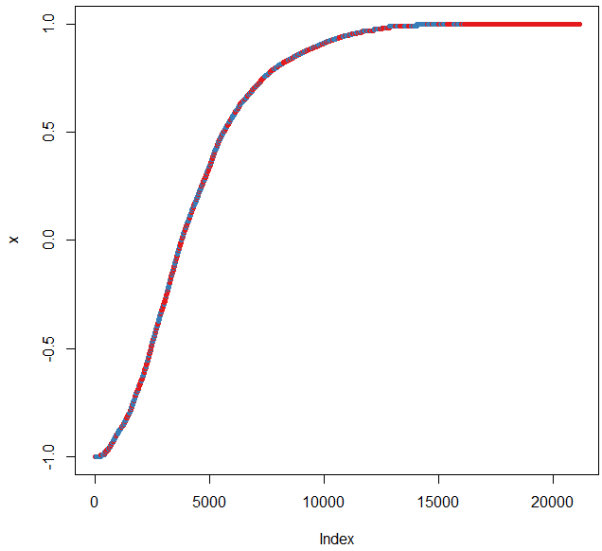


Рисунок 5.11 График зазора прогнозов

На рисунке 5.11 показан зазор прогноза для каждого наблюдения. Зазор определяется как доля голосов, поданных за правильный класс, минус максимальная доля голосов, поданных за другие классы. Например, возьмем наблюдение *x*, принадлежащее классу *K*. Из 100 деревьев 80 верно проголосовало за класс *K*, а остальные 20 деревьев проголосовали за класс *L*. По правилу большинства для наблюдения *x* верно спрогнозирован класс *K*. Зазор равен (80-20)/100=0.6. Нетрудно догадаться, что положительные значения по вертикальной оси означают корректные прогнозы, а отрицательные значения – ошибочные.

Лекция 5.2. Построение ансамбля деревьев регрессии

5.2.1. Подготовка данных

Теперь применим случайный лес для решения регрессионной задачи. Данные записаны в файле *Creddebt.csv*. Исходная выборка содержит записи о 5000 клиентах. По каждому наблюдению (клиенту) фиксируются следующие переменные (характеристики):

- количественный предиктор *Возраст в годах [age]*;
- порядковый предиктор *Уровень образования [ed]*

- количественный предиктор *Срок занятости на последнем месте работы в месяцах [employ]*;
- количественный предиктор *Срок проживания по последнему адресу [address]*;
- количественный предиктор *Ежемесячный доход в тысячах рублей [income]*;
- количественный предиктор *Отношение суммы обязательств по розничным кредитам к доходу [debtinc]*;
- количественная зависимая переменная *Задолженность по кредитной карте в тысячах рублей [creddebt]*.

Необходимо предсказать размер задолженности по кредитной карте в зависимости от характеристик клиента.
Запишем данные в объект `data`, чтобы начать работу с ними.

```
# загружаем данные
data <- read.csv2("C:/Trees/Creddebt.csv")
```

Поскольку пример подготовки этих данных уже приводился (см. лекцию 3.3 *Построение и интерпретация дерева регрессии CART*), мы ограничимся здесь программным кодом, который выполняет необходимые преобразования.

```
# выполняем необходимые преобразования
data$ed <- ordered(data$ed, levels = c("Неполное среднее", "Среднее", "Среднее специальное",
                                       "Незаконченное высшее", "Высшее, ученая степень"))
set.seed(100)
ind <- sample(2,nrow(data),replace=TRUE,prob=c(0.7,0.3))
development <- data[ind==1,]
holdout <- data[ind==2,]
```

5.2.2. Построение модели и получение OOB оценки качества

Подготовив данные, можно приступать к моделированию. Поскольку наша зависимая переменная *creddebt* является количественной, будет построен случайный лес деревьев регрессии. Давайте построим случайный лес деревьев регрессии по всем исходным предикторам на обучающей выборке, вычислив важность предикторов.

```
# задаем стартовое значение генератора случайных
# чисел для воспроизводимости результатов
set.seed(152)

# строим случайный лес деревьев регрессии
model<-randomForest(creddebt ~., development, importance=TRUE)
```

Выводим информацию о качестве модели

```
# выводим информацию о качестве модели
print(model)
```

Сводка 5.13. Оценка качества модели: используются среднеквадратичная ошибка и коэффициент детерминации, вычисленные по методу OOB

```
Call:
 randomForest(formula = creddebt ~ ., data = development, importance = TRUE)
               Type of random forest: regression
                     Number of trees: 500
No. of variables tried at each split: 2

          Mean of squared residuals: 6.75674
                    % Var explained: 38.95
```

В сводке 5.13 мы видим уже знакомые нам параметры. Параметр `Number of trees` показывает количество деревьев в ансамбле. `No. of variables tried at each split` показывает количество переменных, рассматриваемых в качестве кандидатов при каждом разбиении. Однако для оценки качества ансамбля деревьев регрессии используются уже другие показатели. `Mean of squared residuals` – это среднеквадратичная ошибка по методу OOB. Меньшее значение указывает на лучшее качество модели. `% Var explained` – коэффициент детерминации или доля объясненной дисперсии зависимой переменной по методу OOB. Большее значение указывает на лучшее качество модели.

Теперь выведем график зависимости среднеквадратичной ошибки по методу OOB от количества деревьев в ансамбле.

```
# строим график зависимости среднеквадратичной ошибки по методу OOB
# от количества деревьев в ансамбле
plot(model)
```

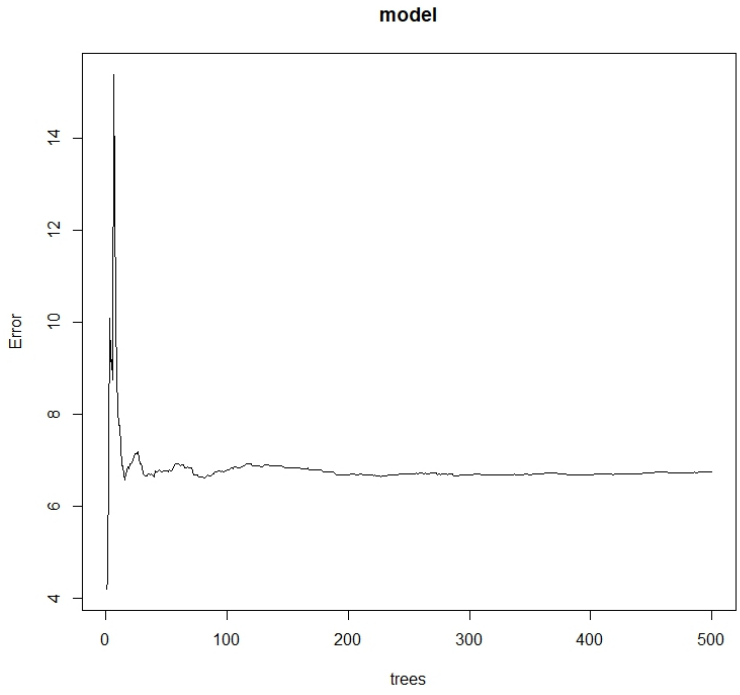


Рисунок 5.12 График зависимости среднеквадратичной ошибки по методу OOB от количества деревьев в ансамбле

Мы видим, как с увеличением числа деревьев среднеквадратичная ошибка падает (обратите внимание на всплески) и стабилизируется.

5.2.3. Важности предикторов

Теперь выведем важности предикторов:

```
importance(model)
```

Сводка 5.14. Важность переменных

```
           %IncMSE IncNodePurity
age      2.5184497      2442.251
ed       1.0674845      3390.937
employ   4.6913157      3214.523
address  0.8050128      1779.784
income  28.3667328     18122.800
debtinc 17.1006399      7343.865
```

Сводка 5.14 состоит из двух столбцов. Первый столбец – это усредненное уменьшение качества модели (используется среднеквадратичная ошибка), а второй столбец – усредненное уменьшение неоднородности (используется сумма квадратов остатков). Важность на основе уменьшения правильности показывает, что сильнее всего на зависимую переменную влияют предикторы *Ежемесячный доход в тысячах рублей [income]* и *Отношение суммы обязательств по розничным кредитам к доходу [debtinc]*.
Что касается уменьшения неоднородности, то здесь наиболее важными предикторами вновь стали переменные *Ежемесячный доход в тысячах рублей [income]* и *Отношение суммы обязательств по розничным кредитам к доходу [debtinc]*.
Для наглядности визуализируем важности (рисунок 5.13).

```
varImpPlot(model)
```

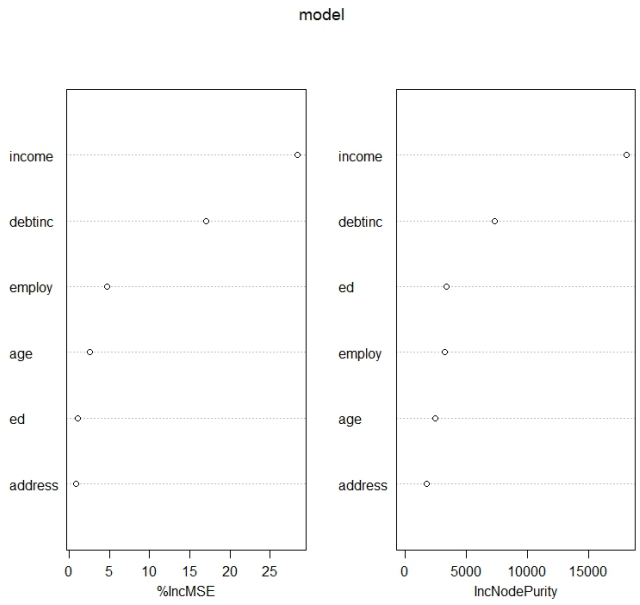


Рисунок 5.13 График важности переменных

5.2.4. Графики частной зависимости

Теперь с помощью графиков частной зависимости проанализируем взаимосвязи между зависимой переменной и ключевыми предикторами. Построим график частной зависимости для предиктора *Ежемесячный доход в тысячах рублей [income].*

```
# строим график частной зависимости для переменной income
partialPlot(model, development, income)
```

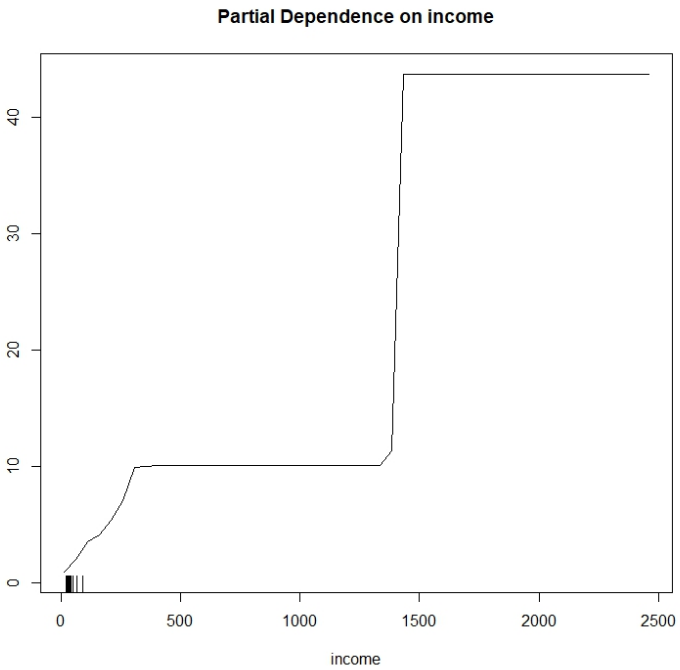


Рисунок 5.14 График частной зависимости для переменной income

На рисунке 5.14 мы видим причудливый ступенчатый график. По мере увеличения значений дохода до 350 тыс. деревья прогнозируют все больший размер задолженности по кредитной карте. Для значений дохода от 350 до 1450 тыс. рублей прогнозируется одинаковый уровень задолженности по кредитной карте, составляющий примерно 10 тыс. рублей, для значений дохода свыше 1450 тыс. рублей прогнозируется резко возросший одинаковый уровень задолженности по кредитной карте, составляющий теперь около 43 тыс. рублей. При этом сразу нужно оговориться, что значения по оси y не следует рассматривать как абсолютные, потому что при усреднении прогнозов по всем наблюдениям специального набора происходит их смещение к среднему.

Теперь построим график частной зависимости для предиктора *Отношение суммы обязательств по розничным кредитам к доходу [debtinc]*.

```
# строим график частной зависимости для переменной debtinc
partialPlot(model, development, debtinc)
```

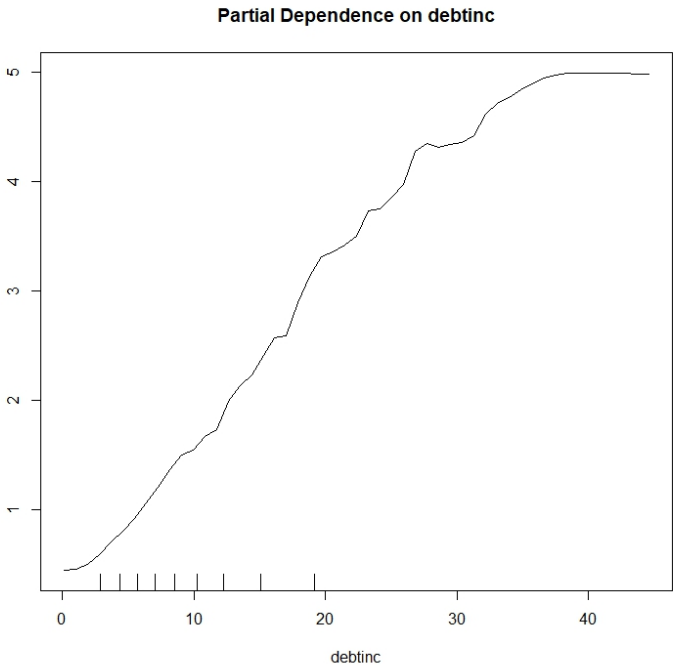


Рисунок 5.15 График частной зависимости для переменной debtinc

Глядя на рисунок 5.15, можно сказать, что в целом более высоким значениям переменной *Отношение суммы обязательств по розничным кредитам к доходу [debtinc]* соответствуют более высокие значения задолженности по кредитной карте.

5.2.5. Получение спрогнозированных значений зависимой переменной

Теперь давайте спрогнозируем значения зависимой переменной для наблюдений обучающей выборки обычным методом и по методу OOB, а

затем запишем спрогнозированные вероятности в объекты `predvalue_dev` и `predvalue_dev_oob` соответственно. Напомню, что при наличии аргумента, задающего выборку, функция `predict` вернет для каждого наблюдения выборки значение, спрогнозированное обычным способом, а если опустить этот аргумент, то будут возвращены значения, спрогнозированные по методу OOB. Сначала спрогнозируем значения зависимой переменной для обучающей выборки обычным способом.

```
# прогнозируем значения зависимой переменной
# для обучающей выборки обычным способом
predvalue_dev <- predict(model, development)
```

Для компактности выведем значения зависимой переменной, вычисленные обычным способом, по последним 5 наблюдениям обучающей выборки.

```
# выводим значения зависимой переменной
# для последних 5 наблюдений обучающей
# выборки, вычисленные по обычному методу
tail(predvalue_dev, 5)
```

Сводка 5.15. Спрогнозированные значения зависимой переменной по последним 5 наблюдениям обучающей выборки, прогнозы получены обычным способом

```
     4993      4994      4995      4999      5000
0.7738926 1.0425989 0.7552551 0.5881830 0.8830769
```

Теперь вычислим среднеквадратичную ошибку и R-квадрат для обучающей выборки обычным способом.

```
# вычисляем среднеквадратичную ошибку для обучающей выборки по обычному методу,
# для этого сумму квадратов разностей между фактическими и спрогнозированными
# значениями зависимой переменной делим на количество наблюдений, при этом
# каждое спрогнозированное значение - результат усреднения средних
# значений, вычисленных деревьями по всем бутстреп-выборкам
MSE_dev <- sum((development$creddebt-predvalue_dev)^2)/nrow(development)

# вычисляем сумму квадратов отклонений фактических значений
# зависимой переменной в обучающей выборке от ее среднего значения
TSS <- sum((development$creddebt-(mean(development$creddebt)))^2)
# вычисляем сумму квадратов отклонений фактических значений
# зависимой переменной в обучающей выборке от спрогнозированных,
# при этом каждое спрогнозированное значение - результат усреднения
# средних значений, вычисленных деревьями по всем бутстреп-выборкам
RSS <- sum((development$creddebt-predvalue_dev)^2)
# вычисляем R-квадрат для обучающей выборки по обычному методу
R2_dev <- (1-(RSS/TSS))*100

# печатаем результаты
output <- c(MSE_dev, R2_dev)
names(output) <- c("MSE", "R2")
output
```

Сводка 5.16. Среднеквадратичная ошибка и R-квадрат для обучающей выборки, вычисленные по обычному методу

```
      MSE        R2
```

```
 1.327817  88.002774
```

Полученные среднеквадратичная ошибка и R-квадрат разительно отличаются от среднеквадратичной ошибки и R-квадрата по методу OOB, приведенные в сводке 5.13. Это обусловлено тем, что для расчета среднеквадратичной ошибки и R-квадрата мы использовали прогнозы, полученные с помощью всех бутстреп-выборок, а для вычисления среднеквадратичной ошибки по методу OOB использовались прогнозы, которые вычислялись по out-of-bag выборкам (то есть тогда каждое наблюдение прогнозировалось с использованием только тех выборок, в которых оно отсутствовало). В данном случае среднеквадратичная ошибка равна 1,33 (сравните со среднеквадратичной ошибкой по методу OOB, равной 6,76), а R-квадрат равен 88 (сравните с R-квадратом по методу OOB, равном 38,95). Как правило, среднеквадратичная ошибка, вычисленная по всем бутстреп-выборкам, меньше среднеквадратичной ошибки, вычисленной по out-of-bag выборкам. R-квадрат, вычисленный по всем бутстреп-выборкам, наоборот, больше R-квадрата, вычисленного по out-of-bag выборкам. Таким образом, случайный лес, используя метод OOB, позволяет получить более реалистичную оценку качества модели.
Теперь вручную вычислим среднеквадратичную ошибку и R-квадрат для обучающей выборки по методу OOB.

```
# прогнозируем значения зависимой переменной
# для обучающей выборки по методу OOB
oob_predvalue_dev <- predict(model)

# вычисляем среднеквадратичную ошибку для обучающей выборки по обычному методу,
# для этого сумму квадратов разностей между фактическими и спрогнозированными
# значениями зависимой переменной делим на количество наблюдений, при этом
# каждое спрогнозированное значение – результат усреднения средних
# значений, вычисленных деревьями по OOB выборкам
oob_MSE_dev <- sum((development$creddebt-oob_predvalue_dev)^2)/nrow(development)

# вычисляем сумму квадратов отклонений фактических значений
# зависимой переменной в обучающей выборке от ее среднего значения
TSS <- sum((development$creddebt-(mean(development$creddebt)))^2)
# вычисляем сумму квадратов отклонений фактических значений
# зависимой переменной в обучающей выборке от спрогнозированных,
# при этом каждое спрогнозированное значение – результат усреднения
# средних значений, вычисленных деревьями по OOB выборкам
RSS <- sum((development$creddebt-oob_predvalue_dev)^2)
# вычисляем R-квадрат для обучающей выборки по методу OOB
oob_R2_dev <- (1-(RSS/TSS))*100

# печатаем результаты
output <- c(oob_MSE_dev, oob_R2_dev)
names(output) <- c("MSE", "R2")
output
```

Сводка 5.17. Среднеквадратичная ошибка и R-квадрат для обучающей выборки, вычисленные вручную по методу OOB

```
     MSE       R2
 6.75674 38.95083
```

Среднеквадратичная ошибка и R-квадрат для обучающей выборки, вычисленные вручную по методу OOB, в точности совпадают со среднеквадратичной ошибкой и R-квадратом, приведенными в сводке 5.13.
Спрогнозируем значения зависимой переменной для контрольной выборки.

```
# прогнозируем значения зависимой переменной
# для контрольной выборки
predvalue_hold <- predict(model, holdout)
```

Для компактности выведем значения зависимой переменной по последним 5 наблюдениям контрольной выборки.

```
# выводим значения зависимой переменной
# для последних 5 наблюдений контрольной
# выборки
tail(predvalue_hold, 5)
```

Сводка 5.18. Спрогнозированные значения зависимой переменной по последним 5 наблюдениям контрольной выборки

```
     4988      4989      4996      4997      4998
3.5994980 1.0787324 0.9805083 0.5102470 1.3359230
```

Теперь вручную вычислим среднеквадратичную ошибку и R-квадрат для контрольной выборки.

```
# вычисляем среднеквадратичную ошибку для контрольной выборки
MSE_hold <- sum((holdout$creddebt-predvalue_hold)^2)/nrow(holdout)
# вычисляем сумму квадратов отклонений фактических значений
# зависимой переменной в контрольной выборке от ее среднего значения
TSS <- sum((holdout$creddebt-(mean(holdout$creddebt)))^2)
# вычисляем сумму квадратов отклонений фактических значений
# зависимой переменной в контрольной выборке от спрогнозированных
RSS <- sum((holdout$creddebt-predvalue_hold)^2)
# вычисляем R-квадрат для контрольной выборки
R2_hold <- (1-(RSS/TSS))*100
# печатаем результаты
output <- c(MSE_hold, R2_hold)
names(output) <- c("MSE", "R2")
output
```

Сводка 5.19. Среднеквадратичная ошибка и R-квадрат для контрольной выборки

```
      MSE        R2
 1.996315 61.663733
```

5.2.6. Улучшение качества прогнозов

Попробуем улучшить модель (т.е. минизировать среднеквадратичную ошибку на контрольной выборке), вычислив оптимальное значение `mtry`. Для задачи регрессии в качестве таких значений используются треть от общего количества предикторов, поделенная пополам, треть от общего

количества предикторов, удвоенная треть от общего количества предикторов и общее количество предикторов.

```
# настраиваем оптимальное значение mtry
set.seed(152)
tuneRF(development[,1:6], development[,7], ntreeTry=500, trace=FALSE)
```

Сводка 5.20. Зависимость среднеквадратичной ошибки по методу OOB от количества случайно отбираемых предикторов для разбиения

```
-0.1005529 0.05
0.09034864 0.05
0.04458031 0.05
  mtry OOBError
1    1 7.606072
2    2 6.911137
4    4 6.286725
6    6 6.006461
```

Помимо сводки генерируется график зависимости среднеквадратичной ошибки, вычисленной по методу OOB, от количества случайно отбираемых переменных для разбиения.

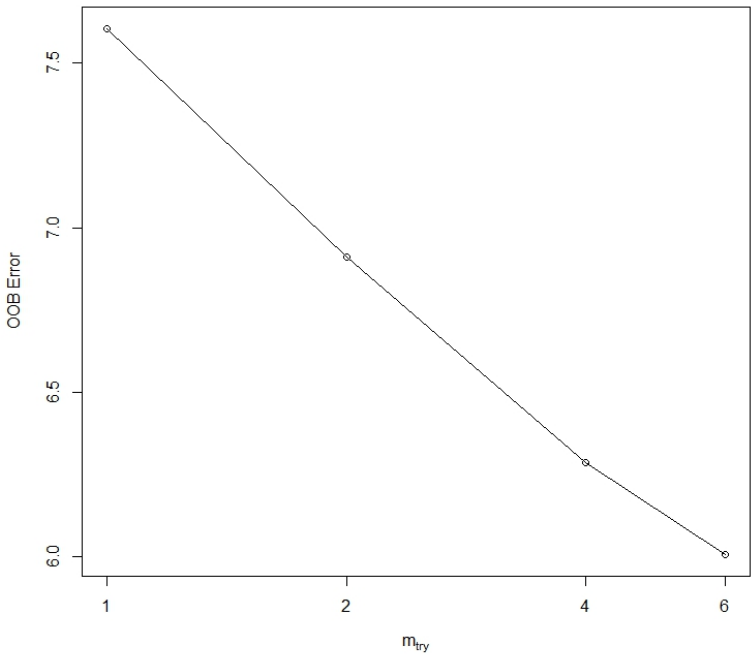


Рисунок 5.16 График зависимости среднеквадратичной ошибки, вычисленной по методу OOB, от количества переменных для разбиения

Наименьшее значение OOB ошибки наблюдается для `mtry=6`. Оно отличается от значения `mtry=2`, автоматически выбранного в ходе построения случайного леса.

Теперь мы построим случайный лес деревьев регрессии, использовав `mtry=6`.

```
# строим модель с новым значением mtry
set.seed(152)
model2<-randomForest(creddebt ~., development, mtry=6)
```

Выведем информацию о новой модели.

```
# выводим информацию о качестве модели
print(model2)
```

Сводка 5.21. Оценка качества модели, mtry=6

```
Call:
 randomForest(formula = creddebt ~ ., data = development, mtry = 6)
               Type of random forest: regression
                     Number of trees: 500
No. of variables tried at each split: 6

          Mean of squared residuals: 6.0699
                    % Var explained: 45.16
```

Сравнительный анализ моделей показывает улучшение показателей качества. По сравнению с предыдущей моделью теперь среднеквадратичная ошибка по методу OOB на обучающей выборке уменьшилась и равна 6,07 (сравните с 6,75 для модели случайного леса с `mtry=2`), а доля объясненной дисперсии увеличилась и равна 45,16% (сравните с 38,95%). Теперь вычислим среднеквадратичную ошибку и R-квадрат новой модели на контрольной выборке.

```
# прогнозируем значения зависимой переменной
# для контрольной выборки
predval_hold <- predict(model2, holdout)
# вычисляем среднеквадратичную ошибку для контрольной выборки
MSE_hold <- sum((holdout$creddebt-predval_hold)^2)/nrow(holdout)
# вычисляем сумму квадратов отклонений фактических значений
# зависимой переменной в контрольной выборке от ее среднего значения
TSS <- sum((holdout$creddebt-(mean(holdout$creddebt)))^2)
# вычисляем сумму квадратов отклонений фактических значений
# зависимой переменной в контрольной выборке от спрогнозированных
RSS <- sum((holdout$creddebt-predval_hold)^2)
# вычисляем R-квадрат для контрольной выборки
R2_hold <- (1-(RSS/TSS))*100
# печатаем результаты
output <- c(MSE_hold, R2_hold)
names(output) <- c("MSE", "R2")
output
```

Сводка 5.22. Среднеквадратичная ошибка и R-квадрат для контрольной выборки

```
      MSE        R2
 1.870343 64.082838
```

5.2.7. Получение более развернутого вывода о качестве модели

Обратите еще раз внимание на сводку 5.20. Если вы подгоняете модель случайного леса на обучающей выборке (когда используется разбиение

на обучающую и контрольную выборки), то оценка качества модели выводится только для обучающей выборки, а оценку качества модели для контрольной выборки нужно самостоятельно вычислять самостоятельно вручную. Чтобы получить более развернутый вывод о работе модели на обучающей и контрольной выборках, можно поступить следующим образом. Мы создадим четыре таблицы данных: `Xtrain` будет содержать значения предикторов для обучения, `ytrain` – значения зависимой переменной для обучения, `Xtest` – значения предикторов для контроля, `ytest` – значения зависимой переменной для контроля.

```
# создаем датафреймы
Xtrain <-development[,1:6]
ytrain <-development[,7]
Xtest <-holdout[,1:6]
ytest <-holdout[,7]
```

Строим случайный лес деревьев регрессии по всем исходным предикторам.

```
# строим модель, теперь мы получим
# более развернутые результаты
set.seed(152)
model2 <- randomForest(Xtrain, ytrain, Xtest, ytest, mtry=6)
```

Выводим информацию о модели.

```
# выводим информацию о качестве модели
print(model2)
```

Сводка 5.23. Оценка качества модели на обучающей и контрольной выборке

```
Call:
 randomForest(x = Xtrain, y = ytrain, xtest = Xtest, ytest = ytest,      mtry = 6)
               Type of random forest: regression
                     Number of trees: 500
No. of variables tried at each split: 6

          Mean of squared residuals: 6.0699
                    % Var explained: 45.16
                       Test set MSE: 1.87
                    % Var explained: 64.1
```

В сводке 5.23 мы видим уже знакомые нам показатели. Однако теперь они приводятся для обучающей и контрольной выборок.

Лекция 5.3. Поиск оптимальных параметров случайного леса с помощью пакета caret

5.3.1. Схема оптимизации параметров, реализованная в пакете caret

В предыдущих разделах мы строили модели случайного леса, применив настройки по умолчанию, а также воспользовавшись функцией `tuneRF` для подбора оптимального значения `mtry`. Однако в R имеются специальные пакеты, позволяющие автоматически подобрать оптимальные параметры модели. Одним из таких пакетов является пакет `caret` (сокращение от *Classification and Regression Training*), который представляет собой набор функций, призванный упростить процесс построения прогнозных моделей. Удобство пакета заключается в том, что в нем используется унифицированная методология настройки параметров модели, основанная на общепринятых подходах проверки модели и метриках качества.

Ключевой функцией пакета является функция `train`, которая позволяет:

- оценить с помощью алгоритмов создания повторных выборок (ресемплинга) влияние тех или иных настраиваемых параметров на качество модели;
- выбрать «оптимальную» модель с помощью этих параметров;
- оценить качество работы модели на обучающем наборе данных.

Для реализации этих задач применяется комбинированная проверка. В рамках этого метода набор данных сначала разбивается на обучающую и тестовую выборку. На обучающей выборке запускается процедура ресемплинга. Чаще всего в качестве такой процедуры используется перекрестная проверка. Например, при 10-блочной перекрестной проверке исходная обучающая выборка будет разбита на 10 блоков приблизительно равного объема, а затем 10 раз на девяти блоках будет выполнено обучение модели, а десятый, контрольный блок будет использован для проверки. Например, на первом проходе модель будет обучаться на блоках 2-10, а проверяться на блоке 1. На втором проходе модель будет обучаться на блоках 1 и 3-10, а проверяться на блоке 2 и так далее.

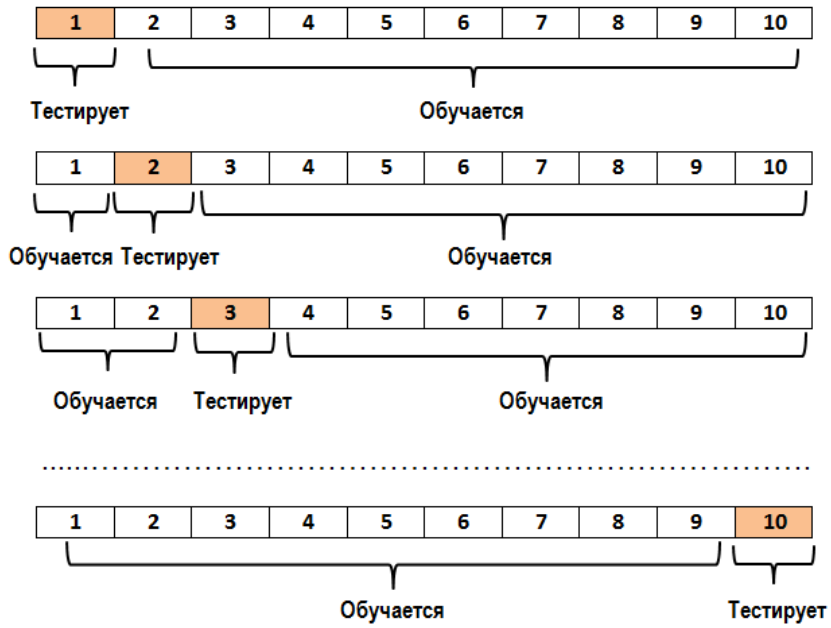


Рис. 5.17 Схема работы 10-блочной перекрестной проверки

В итоге вычисляется среднее значение метрики качества, полученное на 10 контрольных блоках перекрестной проверки. Допустим, у нас есть набор значений параметр(а, ов), метрикой качества является AUC и используется 10-блочная перекрестная проверка. Для каждого значения параметра (или комбинации значений, если используется несколько параметров) будет получено значение AUC, усредненное по 10 контрольным блокам перекрестной проверки. В итоге выбирается модель с таким значением параметра (или с такой комбинацией параметров), которое дает наибольшее значение AUC, усредненное по 10 контрольным блокам перекрестной проверки. Эта модель с наилучшим значением параметра (комбинацией значений параметров) подгоняется на исходной обучающей выборке и проверяется на тестовой выборке. Все вышесказанное можно записать в виде схемы:

1. Разбить набор данных на обучающую и тестовую выборки
2. Задать набор значений параметр(а, ов)
3. **for** каждого значения параметра (комбинации значений параметров) **do**
4. ¦ **for** для каждой итерации перекрестной проверки **do**
5. ¦ ¦ Сформировать обучающие блоки и контрольный блок перекрестной проверки
6. ¦ ¦ Подогнать модель на обучающих блоках перекрестной проверки
7. ¦ ¦ Вычислить метрику качества для контрольного блока перекрестной проверки
8. ¦ **end**
9. ¦ Вычислить метрику качества модели, усредненную по контрольным блокам
10. **end**
11. Определить оптимальное значение параметра (оптимальную комбинацию значений), дающее наилучшее качество
12. Подогнать модель на обучающей выборке с использованием оптимального значения параметра (оптимальной комбинации значений)

5.3.2. Настройка условий оптимизации

Перед обучением модели при помощи функции `train` необходимо задать нужный алгоритм и весь набор условий оптимизации. Этот набор условий создается с помощью вспомогательной функции `trainControl`. Вызвать эту функцию можно с помощью команды

```
trainControl(method, number, p, repeats, search, verboseIter=FALSE,
             summaryFunction, selectionFunction)
```

где:

`method=`	Задает метод создания повторных выборок: `"boot"`, `"boot632"`, `"cv"`, `"repeatedcv"`, `"LOOCV"`, `"LGOCV"` – для повторных разбиений на обучающую и контрольную выборки (по умолчанию используется `boot`); `"none"` – подгоняет модель на всем наборе; `"oob"` – предназначен для случайного леса, бэггинга и др.; `"adaptive_cv"`, `"adaptive_boot"` и `"adaptive_LGOCV"`.
`number=`	Задает количество блоков перекрестной проверки или количество итераций при создании повторных выборок.
`p=`	Задает долю обучающей выборки от общего объема данных при выполнении перекрестной проверки.
`repeats=`	Задает количество повторов для перекрестной проверки.
`search=`	Задает способ перебора параметров модели: `"grid"` – по предварительно заданной сетке; `"random"` – случайным образом.
`verboseIter=`	`TRUE` – печатает сообщения о ходе вычислений. По умолчанию сообщения не выводятся.
`summaryFunction=`	Задает функцию, вычисляющую сводную метрику качества модели на основе повторных выборок.
`selectionFunction=`	Задает функцию выбора оптимального значения настраиваемого параметра.

Например, создав объект `fitControl`

```
fitControl <- trainControl(method="repeatedcv",
                           number=10, repeats=10)
```

мы задаем следующие параметры перекрестной проверки:

- `method="repeatedcv"` означает, что необходимо выполнить повторную перекрестную проверку (кроме того, возможны варианты перекрестной проверки без повторов, проверки по одному наблюдению и др.);
- `number=10` означает, что в процессе перекрестной проверки исходные данные необходимо разбить на 10 (примерно) равных частей;
- `repeats=10` означает, что перекрестная проверка будет запущена 10 раз.

Теперь расскажем непосредственно о самой функции `train`. Для ее вызова необходимо воспользоваться командой

```
train(formula, data, method, preProcess, metric, maximize,
      trControl = trainControl(), tuneGrid, tuneLength)
```

`formula=`	Задает формулу в формате *Зависимая переменная ~ Предиктор1 + Предиктор2 + Предиктор3* + др.
`data=`	Задает таблицу данных для анализа
`method=`	Задает модель классификации или регрессии, которую необходимо построить и оптимизировать. Если выполнить команду `names(getModelInfo())`, то можно увидеть список из 233 доступных методов
`preProcess=`	Задает способ предварительной обработки предикторов. В настоящий момент доступны `"BoxCox"`, `"YeoJohnson"`, `"expoTrans"`, `"center"`, `"scale"`, `"range"`, `"knnImpute"`, `"bagImpute"`, `"medianImpute"`, `"pca"`, `"ica"` и `"spatialSign"`. По умолчанию предварительная обработка не используется
`metric=`	Задает сводную метрику качества, используемую для отбора оптимальной модели. По умолчанию для классификации используется правильность классификации и коэффицинт каппа (`"Accuracy"` и `"Kappa"`), а для регрессии – квадратный корень из среднеквадратичной ошибки и коэффициент детерминации (`"RMSE"` и `"Rsquared"`)
`maximize=`	Задает способ оптимизации сводной метрики качества (максимизацию или минимизацию)
`trControl=`	Задает набор условий оптимизации

`tuneGrid=`	Задает таблицу данных с возможными значениями настраиваемых параметров. Названия столбцов должны соответствовать названиям настраиваемых параметров
`tuneLength=`	Задает количество перебираемых значений настраиваемого параметра. По умолчанию этот аргумент равен количеству уровней для каждого настраиваемого параметра. Если функция `trainControl` использует случайный способ перебора настраиваемых параметров (`search="random"`), аргумент равен максимальному количеству комбинаций значений настраиваемых параметров, которое будет сгенерировано в ходе случайного поиска

Теперь устанавливаем developer-версию пакета `caret`:

```
# устанавливаем developer-версию пакета caret
# devtools::install_github("topepo/caret/pkg/caret")
```

Обратите внимание, дополнительно вам может потребоваться установка пакета `Rtools`.

5.3.3. Поиск оптимальных параметров для задачи классификации

Сначала попробуем найти оптимальную модель случайного леса для задачи классификации, применив с помощью пакета `caret` решетчатый поиск параметров. Обратимся к набору данных *Response.csv*, с которым мы работали в рамках лекции 5.1 *Построение ансамбля деревьев классификации*).

Загрузим данные и выполним предварительную подготовку.

```
# загружаем данные
data <- read.csv2("C:/Trees/Response.csv")

# выполняем необходимые преобразования
data[, -c(12:13)] <- lapply(data[, -c(12:13)], factor)
set.seed(42)
data$random_number <- runif(nrow(data),0,1)
training <- data[which(data$random_number > 0.3), ]
test <- data[which(data$random_number <= 0.3), ]
data$random_number <- NULL
training$random_number <- NULL
test$random_number <- NULL
```

Теперь загрузим установленный пакет `caret`.

```
# загружаем пакет caret
library(caret)
```

Также не забудьте загрузить пакет `randomForest`, если ранее он не был загружен. В нашем случае он уже загружен.

```
# не забудьте загрузить пакет randomForest,
# если он ранее не был загружен
# library(randomForest)
```

Поскольку поиск параметров – это довольно ресурсоемкая процедура, мы распараллелим вычисления.

```
# распараллеливаем вычисления
library(parallel)
library(doParallel)
# будем использовать 3 ядра процессора
cluster <- makeCluster(3)
registerDoParallel(cluster)
```

Теперь задаем набор условий оптимизации. Для выбора оптимальной модели применим 5-блочную перекрестную проверку и перебор параметров по предварительно заданной сетке.

```
# задаем набор условий оптимизации: применяем 5-блочную
# перекрестную проверку и перебор параметров
# по заданной сетке
control <- trainControl(method="cv", number=5,
                        search="grid", allowParallel=TRUE)
```

Теперь задаем сетку параметров для решетчатого поиска.

```
# задаем сетку параметров для решетчатого поиска
tunegrid <- expand.grid(.mtry=c(1:7))
```

В данном случае оптимизируем `mtry` – количество предикторов, случайно отбираемых для разбиения узлов. В итоге для каждого значения `mtry` будут вычислены значения правильности и коэффициента каппа, усредненные по 5 контрольным блокам перекрестной проверки.
Задав сетку параметров, можно приступать к построению моделей случайного леса.

```
# строим модели случайного леса и выбираем
# оптимальную с т.з. правильности
set.seed(152)
rf_gridsearch <- train(response ~ ., data=training, method="rf",
                       ntree=600, tuneGrid=tunegrid,
                       trControl=control)
```

Выводим результаты поиска оптимальных параметров.

```
# выводим результаты решетчатого поиска
print(rf_gridsearch)
```

Сводка 5.24. Значения правильности и каппа, усредненные по 5 контрольным блокам перекрестной проверки, параметр оптимизации – mtry

```
Random Forest

21154 samples
   13 predictor
    2 classes: '0', '1'

No pre-processing
Resampling: Cross-Validated (5 fold, repeated 1 times)
Summary of sample sizes: 16923, 16923, 16923, 16924, 16923
Resampling results across tuning parameters:

  mtry  Accuracy   Kappa
  1     0.8009826  0.5890654
  2     0.8168668  0.6258469
  3     0.8197505  0.6319160
  4     0.8177650  0.6275813
  5     0.8165832  0.6251015
  6     0.8123761  0.6162284
  7     0.8054744  0.6024593

Accuracy was used to select the optimal model using  the largest value.
The final value used for the model was mtry = 3.
```

В полученной сводке (сводка 5.24) сначала выводится информация о методе, общем размере обучающей выборки и количестве предикторов. Мы видим, что предварительная обработка данных не использовалась, а в качестве метода ресемплинга использовалась однократная 5-блочная перекрестная проверка. Далее приводятся размеры обучающих блоков, создаваемых в результате перекрестной проверки: 16923, 16923, 16923, 16924, 16923. Соответственно можно легко вычислить размеры контрольных блоков перекрестной проверки. Достаточно вычесть из размера исходной обучающей выборки размер обучающего блока, создаваемой процедурой перекрестной проверки, и получаем значения 4231, 4231, 4231, 4230, 4231. Как видим, для контроля каждый раз используется 20% данных, потому что использовалась 5-блочная перекрестная проверка. Затем приводится список значений `mtry` и соответствующих значений правильности и каппа, усредненных по 5 контрольным блокам однократной перекрестной проверки. Оптимальной моделью стала модель с наибольшим значением правильности, то есть модель с `mtry=3`.

Результаты, приведенные в сводке, можно визуализировать.

```
# визуализируем результаты решетчатого поиска
plot(rf_gridsearch)
```

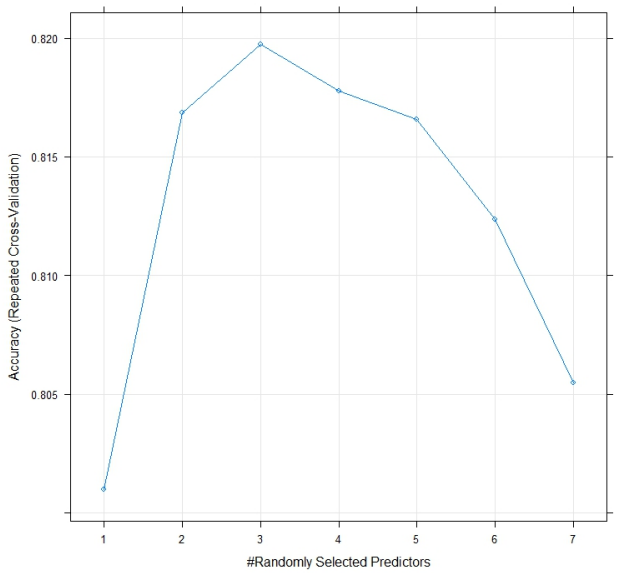


Рисунок 5.18 График зависимости правильности от значений mtry

График показывает, что хорошее качество модели дают не очень низкие и не очень высокие значения `mtry`. Вспомните, что на практике в качестве значения `mtry` используют примерно 20–40% от общего числа предикторов.

Вычислим правильность модели на тестовой выборке. Для этого достаточно вывести матрицу ошибок.

```
# вычисляем прогнозы для тестовой выборки
predval <- predict(rf_gridsearch, test)
# выводим матрицу ошибок
table(test$response, predval)
```

Сводка 5.25. Матрица ошибок оптимальной модели для тестовой выборки

```
  predval
      0    1
0 4380  778
1  848 3099
```

Правильность оптимальной модели равна (4380+3099)/9105=82,1%.

Вспомним, что правильность классификации – это не самая оптимальная метрика качества модели, потому что она зависит от конкретного порога отсечения. Более объективную информацию о качестве бинарного классификатора может дать ROC-кривая и значение AUC. Попробуем оптимизировать параметры с точки зрения ROC-кривой и AUC, а не правильности.

Обратите внимание, когда задана оптимизация по AUC, пакет `caret` требует, чтобы значениям зависимой переменной были присвоены символьные метки. Кроме того, для функции `trainControl` нужно задать значения параметров `classProbs=TRUE` и

`summaryFunction=twoClassSummary`, а для функции `train` задать значение параметра `metric="ROC"` (для удобства соответствующие участки программного кода выделены желтым фоном).

```
# присваиваем символьные метки значениям
# зависимой переменной
training$response<-factor(training$response, levels=c(0, 1),
                          labels=c("NoResponse","Response"),exclude=NULL)
test$response<-factor(test$response, levels=c(0, 1),
                      labels=c("NoResponse","Response"),exclude=NULL)

# задаем обновленный набор условий для оптимизации
control <- trainControl(method="cv", number=5, search="grid",
                        allowParallel=TRUE,
                        classProbs=TRUE,
                        summaryFunction=twoClassSummary)

# строим модели случайного леса и выбираем
# оптимальную с т.з. AUC
set.seed(152)
rf_gridsearch2 <- train(response ~ ., data=training, method="rf",
                        metric="ROC", ntree=600,
                        tuneGrid=tunegrid, trControl=control)
```

Выводим результаты поиска оптимальных параметров.

```
# выводим результаты решетчатого поиска
print(rf_gridsearch2)
```

Сводка 5.26. Значения AUC, усредненные по 5 контрольным блокам однократной перекрестной проверки, параметр оптимизации – mtry

```
Random Forest

21154 samples
   13 predictor
    2 classes: 'NoResponse', 'Response'

No pre-processing
Resampling: Cross-Validated (5 fold, repeated 1 times)
Summary of sample sizes: 16923, 16923, 16923, 16924, 16923
Resampling results across tuning parameters:

  mtry  ROC        Sens       Spec
  1     0.8835404  0.8663845  0.7150515
  2     0.8953186  0.8474017  0.7767463
  3     0.8968917  0.8483186  0.7822157
  4     0.8953430  0.8490674  0.7766364
  5     0.8934161  0.8485680  0.7745580
  6     0.8910624  0.8471528  0.7666824
  7     0.8883331  0.8383284  0.7623068

ROC was used to select the optimal model using  the largest value.
The final value used for the model was mtry = 3.
```

В сводке 5.26 мы видим значения `mtry` и соответствующие значения AUC, усредненные по 5 контрольным блокам однократной перекрестной проверки. Также приводятся значения чувствительности и

специфичности. Оптимальной моделью стала модель с наибольшим значением AUC, то есть модель с `mtry=3`.
Теперь выведем график.

```
# визуализируем результаты решетчатого поиска
plot(rf_gridsearch2)
```

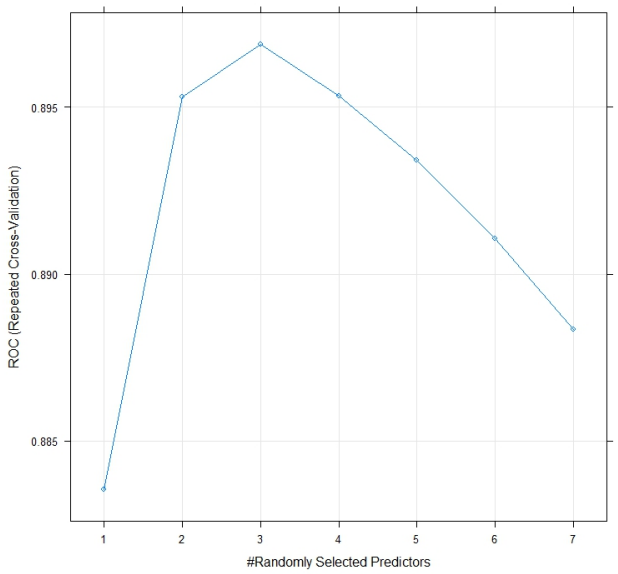


Рисунок 5.19 График зависимости AUC от значений mtry

Теперь вычислим значение AUC оптимальной модели для тестовой выборки.

```
# вычисляем AUC оптимальной модели
# на тестовой выборке
prob <- predict(rf_gridsearch2, test, type="prob")
roc(test$response, prob[,2], ci=TRUE)
```

Сводка 5.27. Значение AUC оптимальной модели, вычисленное для тестовой выборки

```
Call:
roc.default(response = test$response, predictor = prob[, 2],     ci = TRUE)

Data: prob[, 2] in 5158 controls (test$response NoResponse) < 3947 cases (test$response
Response).
Area under the curve: 0.896
95% CI: 0.8895-0.9024 (DeLong)
```

Кроме того, можно построить ROC-кривую оптимальной модели для тестовой выборки.

```
# строим ROC-кривую оптимальной модели
# на тестовой выборке
plot(roc(test$response, prob[,2], ci=TRUE))
```

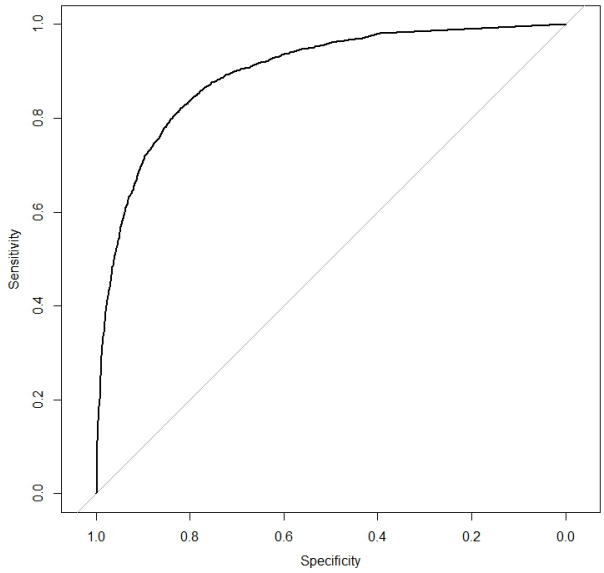


Рисунок 5.20 ROC-кривая оптимальной модели для тестовой выборки

После построения моделей отключаем кластер параллельных вычислений с помощью функции `stopCluster()` и переводим среду R в обычный режим с помощью функции `registerDoSEQ()`.

```
# отключаем кластер параллельных вычислений
stopCluster(cluster)
# переводим среду R в обычный режим
registerDoSEQ()
```

До этого момента мы оптимизировали лишь один параметр случайного леса – количество случайно отбираемых предикторов для разбиения (`mtry`). Теперь попробуем оптимизировать сразу два параметра – количество случайно отбираемых предикторов для разбиения (`mtry`) и минимальный размер терминальных узлов (`nodesize`).

```
# пишем собственную реализацию решетчатого
# поиска для случайного леса
customRF <- list(type = "Classification", library = "randomForest", loop = NULL)
customRF$parameters <- data.frame(parameter = c("mtry", "nodesize"),
                                  class = rep("numeric", 2), label = c("mtry", "nodesize"))
customRF$grid <- function(x, y, len = NULL, search = "grid") {}
customRF$fit <- function(x, y, wts, param, lev, last, weights, classProbs, ...) {
  randomForest(x, y, mtry=param$mtry, nodesize=param$nodesize, ...)
}
customRF$predict <- function(modelFit, newdata, preProc = NULL, submodels = NULL)
  predict(modelFit, newdata)
customRF$prob <- function(modelFit, newdata, preProc = NULL, submodels = NULL)
  predict(modelFit, newdata, type = "prob")
customRF$sort <- function(x) x[order(x[,1]),]
customRF$levels <- function(x) x$classes

control <- trainControl(method="cv", number=5, search="grid",
                        classProbs=TRUE, summaryFunction=twoClassSummary)
tunegrid <- expand.grid(.mtry=c(3:7), .nodesize=c(40, 50, 60))
set.seed(152)
custom <- train(response ~ ., ntree=600, data=training, method=customRF, metric="ROC",
                tuneGrid=tunegrid, trControl=control)
```

Выводим результаты поиска оптимальных параметров.

```
# выводим результаты решетчатого поиска
print(custom)
```

Сводка 5.28. Значения AUC, усредненные по 5 контрольным блокам однократной перекрестной проверки, параметры оптимизации – mtry и nodesize

```
21154 samples
   13 predictor
    2 classes: 'NoResponse', 'Response'

No pre-processing
Resampling: Cross-Validated (5 fold)
Summary of sample sizes: 16923, 16923, 16923, 16924, 16923
Resampling results across tuning parameters:

  mtry  nodesize  ROC        Sens       Spec
  3     40        0.8987762  0.8428236  0.7920598
  3     50        0.8976703  0.8426571  0.7923886
  3     60        0.8975018  0.8427401  0.7951230
  4     40        0.8994809  0.8439061  0.7898724
  4     50        0.8985472  0.8437394  0.7902004
  4     60        0.8984068  0.8437395  0.7904190
  5     40        0.8996168  0.8449884  0.7877938
  5     50        0.8990656  0.8463204  0.7893253
  5     60        0.8981473  0.8444057  0.7888881
  6     40        0.8997445  0.8452383  0.7888879
  6     50        0.8996327  0.8450717  0.7873561
  6     60        0.8986768  0.8463201  0.7856064
  7     40        0.9001374  0.8454880  0.7881221
  7     50        0.8994709  0.8459873  0.7867000
  7     60        0.8988534  0.8463203  0.7856064

ROC was used to select the optimal model using the largest value.
The final values used for the model were mtry = 7 and nodesize = 40.
```

Сводка 5.28 получилась довольно объемной, в ней приводятся комбинации значений `mtry` и `nodesize`, а также соответствующие этим комбинациям значения AUC, усредненные по 5 контрольным блокам однократной перекрестной проверки. Оптимальной моделью стала модель с `mtry=7` и `nodesize=40`.

Теперь визуализируем результаты, приведенные в сводке.

```
# визуализируем результаты решетчатого поиска
plot(custom)
```

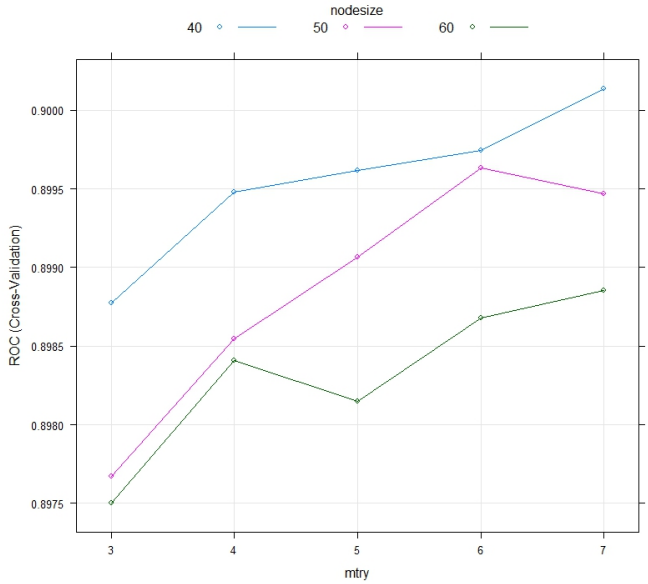


Рисунок 5.21 График зависимости AUC от значений mtry и nodesize

Вновь вычислим значение AUC оптимальной модели для тестовой выборки.

```
# вычисляем AUC оптимальной модели
# на тестовой выборке
score <- predict(custom, test, type="prob")
roc(test$response, score[,2], ci=TRUE)
```

Сводка 5.29. Значение AUC оптимальной модели, вычисленное для тестовой выборки

```
Call:
roc.default(response = test$response, predictor = score[, 2],     ci = TRUE)

Data: score[, 2] in 5158 controls (test$response NoResponse) < 3947 cases (test$response
Response).
Area under the curve: 0.8983
95% CI: 0.892-0.9047 (DeLong)
```

5.3.4. Поиск оптимальных параметров для задачи регрессии

Теперь с помощью пакета `caret` попробуем найти оптимальную модель случайного леса для задачи регрессии. Обратимся к набору данных *Creddebt.csv*, с которым мы работали в рамках лекции 5.2 *Построение ансамбля деревьев регрессии*).

Загрузим данные и выполним предварительную подготовку.

```
# загружаем данные
data <- read.csv2("C:/Trees/Creddebt.csv")

# выполняем необходимые преобразования
data$ed <- ordered(data$ed, levels = c("Неполное среднее", "Среднее", "Среднее специальное",
                                       "Незаконченное высшее", "Высшее, ученая степень"))
set.seed(100)
ind <- sample(2,nrow(data),replace=TRUE,prob=c(0.7,0.3))
tr <- data[ind==1,]
tst <- data[ind==2,]
```

В данном случае мы будем оптимизировать `mtry` – количество предикторов, случайно отбираемых для разбиения узлов, и `nodesize` – минимальный размер терминальных узлов. В итоге для каждой комбинации значения `mtry` и значения `nodesize` будут вычислены значения RMSE, R-квадрат и MAE, усредненные по 5 контрольным блокам перекрестной проверки.

RMSE – это корень из среднеквадратичной ошибки, вычисляется по формуле:

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}$$

где:

y_i – фактическое значение зависимой переменной в *i*-ном наблюдении;

$\hat{y}_i$ – спрогнозированное значение зависимой переменной в *i*-ном наблюдении;

N – общее количество наблюдений.

Чем меньше значение метрики, тем лучше качество модели. RMSE часто используется вместо MSE для того, чтобы получить ошибку такой же размерности, что и у зависимой переменной.

MAE – средняя абсолютная ошибка, вычисляется по формуле:

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i|$$

где:

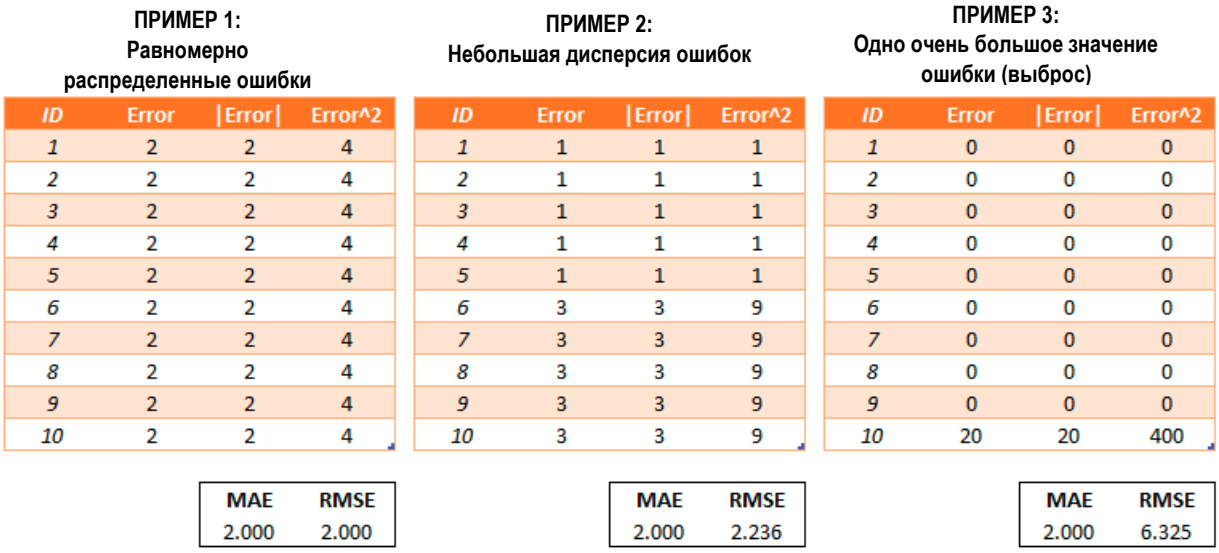
y_i – фактическое значение зависимой переменной в *i*-ном наблюдении;

$\hat{y}_i$ – спрогнозированное значение зависимой переменной в *i*-ном наблюдении.

Между RMSE и MAE много сходств. Обе метрики показывают усредненную ошибку прогноза модели, при этом они имеют ту же единицу измерения, что и у предсказываемой величины. Обе метрики могут варьировать от 0 до ∞ и игнорируют направление колебания данных. Меньшие значения этих метрик указывают на лучшее качество модели. Главное ограничение этих метрик заключается в том, что, будучи усредненными, они не показывают максимального отклонения прогноза от действительности, хотя часто именно максимальная ошибка важна для исследователя.

Однако есть между метриками и отличия. В RMSE ошибки возводятся в квадрат перед их усреднением, поэтому RMSE придает относительно большой вес крупным ошибкам. Это означает, что RMSE будет более полезной метрикой, когда для нас особенно нежелательны большие

ошибки. Ниже приведен рисунок, на котором как раз показана чувствительность метрик RMSE и MAE к большим значениям ошибок.



ПРИМЕР 1:
Равномерно распределенные ошибки

ID	Error	|Error|	Error^2
1	2	2	4
2	2	2	4
3	2	2	4
4	2	2	4
5	2	2	4
6	2	2	4
7	2	2	4
8	2	2	4
9	2	2	4
10	2	2	4

MAE	RMSE
2.000	2.000

ПРИМЕР 2:
Небольшая дисперсия ошибок

ID	Error	|Error|	Error^2
1	1	1	1
2	1	1	1
3	1	1	1
4	1	1	1
5	1	1	1
6	3	3	9
7	3	3	9
8	3	3	9
9	3	3	9
10	3	3	9

MAE	RMSE
2.000	2.236

ПРИМЕР 3:
Одно очень большое значение ошибки (выброс)

ID	Error	|Error|	Error^2
1	0	0	0
2	0	0	0
3	0	0	0
4	0	0	0
5	0	0	0
6	0	0	0
7	0	0	0
8	0	0	0
9	0	0	0
10	20	20	400

MAE	RMSE
2.000	6.325

Рисунок 5.22 Сравнение RMSE и MAE

Итак, запускаем решетчатый поиск.

```
# пишем собственную реализацию решетчатого
# поиска для случайного леса
customRF2 <- list(type = "Regression", library = "randomForest", loop = NULL)
customRF2$parameters <- data.frame(parameter = c("mtry", "nodesize"),
                                   class = rep("numeric", 2), label = c("mtry", "nodesize"))
customRF2$grid <- function(x, y, len = NULL, search = "grid") {}
customRF2$fit <- function(x, y, wts, param, lev, last, weights, classProbs, ...) {
  randomForest(x, y, mtry=param$mtry, nodesize=param$nodesize, ...)
}
customRF2$predict <- function(modelFit, newdata, preProc = NULL, submodels = NULL)
  predict(modelFit, newdata)
customRF2$prob <- function(modelFit, newdata, preProc = NULL, submodels = NULL)
  predict(modelFit, newdata, type = "prob")
customRF2$sort <- function(x) x[order(x[,1]),]
customRF2$levels <- function(x) x$classes

control <- trainControl(method="cv", number=5, search="grid")
tunegrid <- expand.grid(.mtry=c(1:6), .nodesize=c(1, 2, 3, 4))
set.seed(152)
custom2 <- train(creddebt ~ ., ntree=600, data=tr, method=customRF2,
                 tuneGrid=tunegrid, trControl=control)
```

Выводим результаты поиска оптимальных параметров.

```
# выводим результаты решетчатого поиска
print(custom2)
```

Сводка 5.30. Значения RMSE, R-квадрата и MAE, усредненные по 5 контрольным блокам однократной перекрестной проверки,
параметры оптимизации – mtry и nodesize

```
3510 samples
   6 predictor

No pre-processing
Resampling: Cross-Validated (5 fold)
Summary of sample sizes: 2807, 2809, 2807, 2809, 2808
Resampling results across tuning parameters:

  mtry  nodesize  RMSE      Rsquared   MAE
  1     1         2.614823  0.3942928  1.0211830
  1     2         2.615500  0.3964069  1.0264436
  1     3         2.633860  0.3761506  1.0274294
  1     4         2.624637  0.3853526  1.0253762
  2     1         2.371027  0.4867981  0.8515419
  2     2         2.382480  0.4784810  0.8545847
  2     3         2.385492  0.4778798  0.8556066
  2     4         2.370159  0.4857534  0.8535624
  3     1         2.283728  0.5202910  0.8112233
  3     2         2.287827  0.5188280  0.8122249
  3     3         2.299203  0.5105475  0.8149467
  3     4         2.290012  0.5172264  0.8140857
  4     1         2.245138  0.5370693  0.7997725
  4     2         2.241580  0.5397138  0.7982236
  4     3         2.249029  0.5356898  0.8008318
  4     4         2.248938  0.5358604  0.8005149
  5     1         2.214863  0.5513084  0.7934325
  5     2         2.231399  0.5456068  0.7973637
  5     3         2.226780  0.5456523  0.7968450
  5     4         2.218412  0.5508594  0.7953522
  6     1         2.203332  0.5586154  0.7928556
  6     2         2.209071  0.5550404  0.7931363
  6     3         2.196256  0.5610302  0.7924884
  6     4         2.201204  0.5594131  0.7933204

RMSE was used to select the optimal model using the smallest value.
The final values used for the model were mtry = 6 and nodesize = 3.
```

В сводке 5.30 приводятся комбинации значений `mtry` и `nodesize`, а также соответствующие этим комбинациям значения RMSE, R-квадрата и MAE, усредненные по 5 контрольным блокам однократной перекрестной проверки. Оптимальной моделью стала модель с `mtry=6` и `nodesize=3`. Обратите внимание, модель с оптимальным значением RMSE, вовсе не обозначает оптимальность по другим метрикам – R-квадрату и MAE.

Теперь визуализируем результаты, приведенные в сводке.

```
# визуализируем результаты решетчатого поиска
plot(custom2)
```

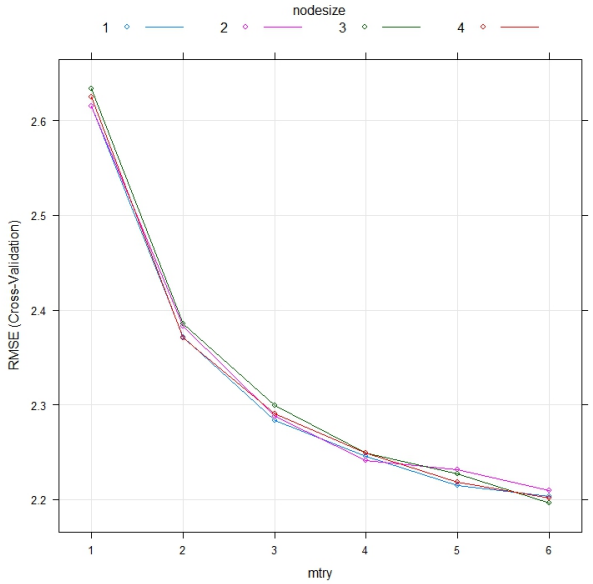


Рисунок 5.23 График зависимости RMSE от значений mtry и nodesize

Теперь вычислим значение RMSE, R-квадрат и MAE оптимальной модели для тестовой выборки.

```
# прогнозируем значения зависимой переменной
# для тестовой выборки с помощью
# оптимальной модели
predictions <- predict(custom2, tst)
# вычисляем корень из среднеквадратичной ошибки для тестовой выборки
RMSE <- sqrt(sum((tst$creddebt-predictions)^2)/nrow(tst))
# вычисляем сумму квадратов отклонений фактических значений
# зависимой переменной в тестовой выборке от ее среднего значения
TSS <- sum((tst$creddebt-(mean(tst$creddebt)))^2)
# вычисляем сумму квадратов отклонений фактических значений
# зависимой переменной в тестовой выборке от спрогнозированных
RSS <- sum((tst$creddebt-predictions)^2)
# вычисляем R-квадрат для тестовой выборки
R2 <- (1-(RSS/TSS))*100
# вычисляем среднюю абсолютную ошибку для тестовой выборки
MAE <- sum(abs(tst$creddebt-predictions))/nrow(tst)
# печатаем результаты
output <- c(RMSE, R2, MAE)
names(output) <- c("RMSE", "R2", "MAE")
output
```

Сводка 5.31. RMSE, R-квадрат и MAE оптимальной модели, вычисленное для тестовой выборки

```
      RMSE         R2        MAE
 1.3448210 65.2696000  0.7123731
```

Лекция 5.4. Улучшение интерпретабельности случайного леса с помощью пакета randomForestExplainer

Ансамблевые алгоритмы, классическим представителем которых является случайный лес, традиционно рассматриваются как модели «черного ящика»: в отличие от дерева решений такую модель нельзя простым и понятным способом визуализировать, она совсем не похожа на регрессионную модель, где по величине и знаку коэффициента можно судить о силе и направленности влияния предикторов. Пакет `randomForestExplainer` представляет собой попытку повысить интерпретируемость случайного леса путем получения различных оценок важности предикторов. Эти оценки также можно использовать на этапе отбора признаков (feature selection) для оптимальной модели.
Чтобы начать работу с пакетом `randomForestExplainer`, установим его с GitHub и загрузим:

```
# устанавливаем пакет randomForestExplainer
# devtools::install_github("MI2DataLab/randomForestExplainer")

# загружаем пакет randomForestExplainer
library(randomForestExplainer)
```

Обратите внимание, для корректной установки пакета `randomForestExplainer` требуется наличие установленных пакетов `DT` и `rmarkdown`.

Воспользуемся данными, с которыми мы работали в рамках лекции 5.1.

```
# загружаем данные
data <- read.csv2("C:/Trees/Response.csv")

# выполняем необходимые преобразования
data[, -c(12:13)] <- lapply(data[, -c(12:13)], factor)
set.seed(42)
data$random_number <- runif(nrow(data),0,1)
development <- data[which(data$random_number > 0.3), ]
holdout <- data[ which(data$random_number <= 0.3), ]
development$random_number <- NULL
holdout$random_number <- NULL
```

Теперь построим случайный лес, состоящий из 500 деревьев классификации (значение по умолчанию), при этом для параметра `localImp` задаем значение `TRUE`. Обратите внимание, мы используем заведомо избыточное количество деревьев для уменьшения неопределенности получаемых далее оценок параметров важности предикторов (мотивация здесь такая же, как и при увеличении объема выборки в выборочных исследованиях).

```
# задаем стартовое значение генератора
# случайных чисел
set.seed(152)
# строим случайный лес деревьев классификации
forest <-randomForest(response ~., development, localImp=TRUE)
```

5.4.1. Оценка важности предиктора с точки зрения минимальной глубины использования

Первым способом оценить важность предикторов является вычисление *минимальной глубины* (minimal depth), на которой каждый предиктор используется для создания очередного ветвления в деревьях случайного леса. Чем меньше значение данного показателя (т.е. чем ближе к корню он используется для разделения), тем более важным можно считать соответствующий предиктор. Для этого мы просто передаем функции `min_depth_distribution` пакета `randomForestExplainer` нашу модель случайного леса.

```
# мы передаем нашу модель случайного леса функции
# min_depth_distribution, чтобы получить информацию
# о значениях минимальной глубины для каждого
# предиктора по каждому дереву
min_depth_frame <- min_depth_distribution(forest)
# выводим информацию о значении минимальной глубины
# для всех предикторов по первому дереву
subset(min_depth_frame, min_depth_frame$tree == 1)
```

Сводка 5.32. Важность предикторов с точки зрения минимальной глубины (по данным дерева 1)

```
   tree variable minimal_depth
1     1      age             2
2     1 atm_user             2
3     1 cre_card             3
4     1 curr_acc             3
5     1 cus_leng             2
6     1 deb_card             3
7     1 internet             5
8     1 life_ins             0
9     1   markpl             2
10    1 mob_bank             1
11    1 mortgage             3
12    1  perloan             5
13    1  savings             4
```

В сводке 5.31 видим, что в дереве 1 наиболее важными предикторами стали переменные *Страхование жизни [life_ins]* и *Мобильный банк [mob_bank]*, наименее важным предиктором стала переменная *Интернет-доступ к счету [internet]* и *Индивидуальный займ [perloan]*.

Несколько более информативным подходом является построения графика распределения минимальной глубины с помощью функции `plot_min_depth_distribution`. Функция `plot_min_depth_distribution` имеет общий вид:

```
plot_min_depth_distribution(min_depth_frame, k = 10,
                            min_no_of_trees = 0,
                            mean_sample = "top_trees",
                            mean_scale = FALSE)
```

где

`min_depth_frame=`	Задает результат функции `min_depth_distribution`, полученный для модельного объекта или сам модельный объект
`k=`	Задает максимальное количество предикторов, отображаемых на графике
`min_no_of_trees=`	Задает минимальное количество деревьев, в котором должен использоваться предиктор, чтобы быть включенным в анализ. Рекомендуется использовать, когда модель включает большое количество признаков. Это позволяет избежать переменных, которые были выбраны по чисто случайным причинам – например, всего один раз во всем ансамбле, но при этом в корневом узле.
`mean_sample=`	Задает способ вычисления среднего значения минимальной глубины. Этот способ зависит от того, как мы будем учитывать деревья, вообще не использующие тот или иной предиктор Возможные значения: • `mean_sample="all_trees"` (заполнение пропущенных значений) – минимальная глубина для предиктора в дереве, которое этот предиктор не использует, принимается равной средней глубине деревьев в ансамбле (глубина дерева считается как самый длинный путь от корня к листу). • `mean_sample = "top_trees"` (ограничение выборки) – расчет среднего значения минимальной глубины осуществляется с учетом подвыборки $\tilde{B}$ из всех B деревьев, где $\tilde{B}$ равно наибольшему количеству деревьев, в которых используется любой из предикторов (пример: если предиктор X1 используется в 120 деревьях, X2 – в 150 и X3 – в 240, то $\tilde{B} = 240$). Для предикторов, используемых для разделения менее чем в $\tilde{B}$ деревьях, заполнение пропусков происходит так же, как при использовании значения `"all_trees"`. Этот вариант используется по умолчанию. • `mean_sample="relevant_trees"` (игнорирование пропусков) – среднее значение минимальной глубины рассчитывается без учета пропущенных значений, т.е.

	без деревьев, в которых не использовался тот или иной предиктор
`mean_scale=`	Нужно ли масштабировать средние значения минимальной глубины так, чтобы они находились в интервале от 0 до 1. По умолчанию используется значение `FALSE`

Давайте построим график распределения минимальной глубины для нашего примера.

```
# выводим график распределения минимальной глубины
plot_min_depth_distribution(min_depth_frame)
```

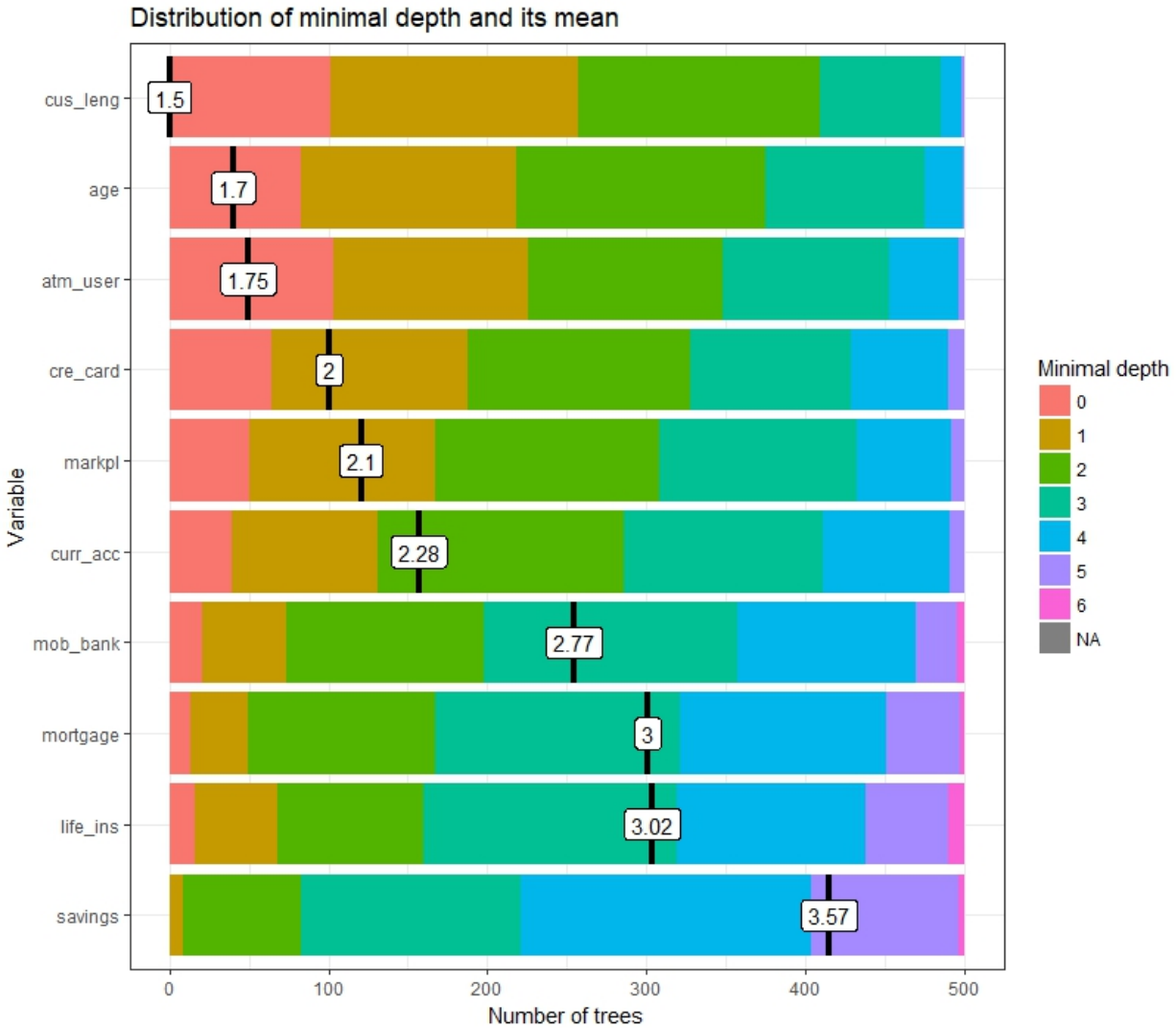


Рисунок 5.23 График распределения минимальной глубины и ее среднего

Этот график для каждого предиктора показывает количество деревьев, в которых этот предиктор использовался на минимальной глубине 0 (в корне дерева), на минимальной глубине 1 и так далее. Обратите

внимание, что один и тот же предиктор может использоваться при построении дерева более одного раза, но при анализе учитывается только его первое использование.

Если сравнить график, приведенный на рис. 5.23, с графиком важностей, вычисленных на основе уменьшения правильности и уменьшения неоднородности (рис. 5.3), можно увидеть много общего: согласно обоим графикам наиболее важными предикторами являются переменные *cus_leng*, *age* и *atm_user*.

Давайте ограничим максимальное количество отображаемых на графике предикторов до пяти:

```
# выводим график распределения минимальной глубины,
# ограничившись 5 наиболее важными предикторами
plot_min_depth_distribution(min_depth_frame,
                            mean_sample = "relevant_trees",
                            k = 5)
```

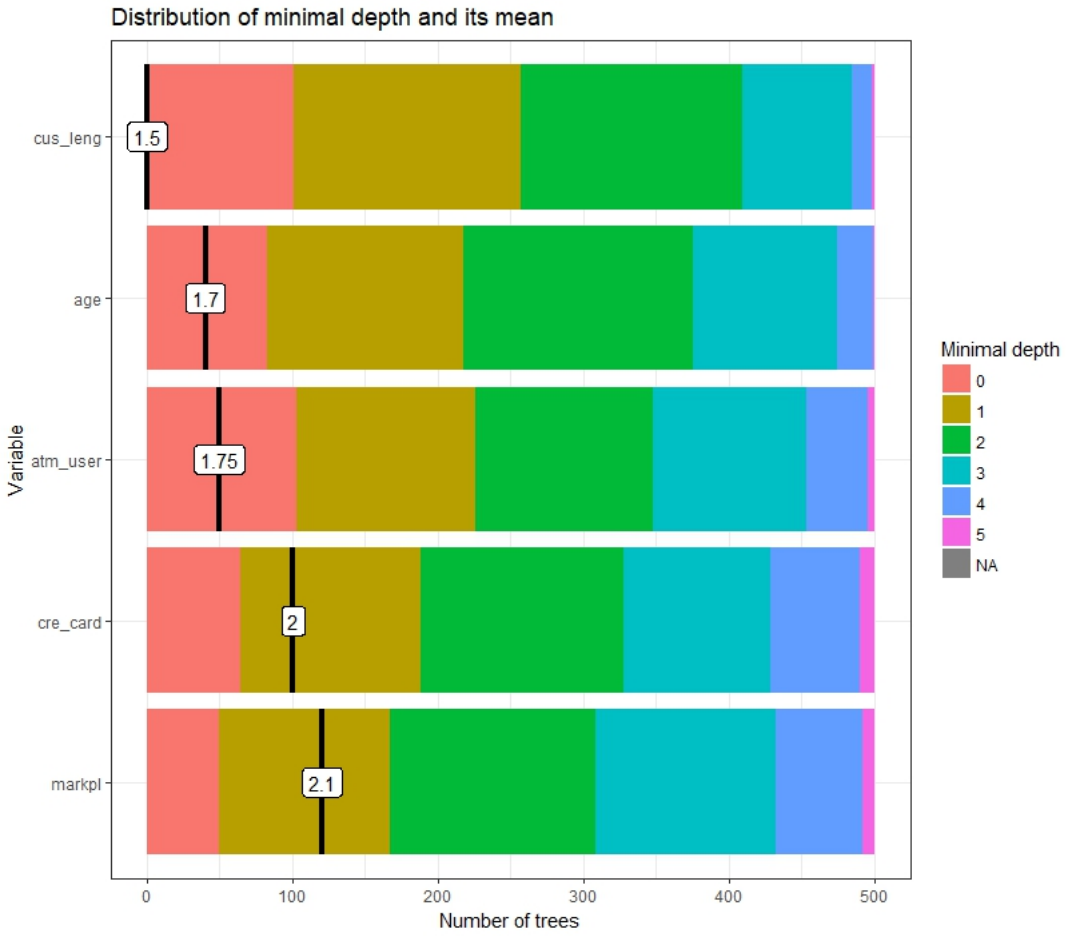


Рисунок 5.24 График распределения минимальной глубины и ее среднего (только пять самых важных предикторов)

Графики, приведенные на рис. 5.23 и рис. 5.24, позволяют составить представление о роли того или иного предиктора в ансамбле, но их трактовка содержит в себе элемент неоднозначности. Например, возможна ситуация, когда один предиктор имеет несколько меньшее

значение минимальной глубины, но при этом ни разу не используется для разделения в корневом узле, а другой имеет чуть большую минимальную глубину, но зато часто используется в корневом узле.

5.4.2. Альтернативные метрики важности

С помощью функции `measure_importance` можно вычислить девять метрик важности предикторов, включая рассмотренную выше минимальную глубину (часть из этих метрик имеет смысл для задачи регрессии, а часть – для задачи классификации). Функция `measure_importance` имеет общий вид:

```
measure_importance(forest, mean_sample = "top_trees",
                   measures = NULL)
```

где

`forest=`	Задает модельный объект
`mean_sample=`	Задает способ вычисления средней глубины. Возможные значения: `"all_trees"`, `"top_trees"` и `"relevant_trees"`.
`measures=`	Задает вычисление метрик важности. Если задано `NULL`, вычисляются все возможные показатели. Возможные значения: • `accuracy_decrease` – среднее уменьшение правильности после перестановок значений переменной X_j (только для классификации); • `gini_decrease` – среднее уменьшение индекса Джини (увеличение чистоты узлов) после ветвления с использованием X_j (только для классификации); • `mse_increase` – среднее увеличение MSE после перестановок значений переменной X_j (только для регрессии); • `node_purity_increase` – среднее увеличение чистоты узлов после ветвления с использованием X_j, измеряемое как уменьшение суммы квадратов остатков (только для регрессии); • `mean_minimal_depth` – среднее значение минимальной глубина, рассчитанное одним из трех вышерассмотренных способов (способ вычисления среднего значения минимальной глубины регулируется параметром `mean_sample`); • `no_of_trees` – общее количество деревьев, в которых встречается ветвление по X_j;

	• `no_of_nodes` – общее количество узлов с ветвлением по X_j (для неглубоких деревьев обычно равно значению `no_of_trees`); • `times_a_root` – общее количество деревьев, в которых ветвление в корневом узле происходит по X_j; • `p_value` – p-значение для одностороннего биномиального теста, использующего следующее распределение: $$Bin\left(no\_of\_nodes,\ P(\text{ветвление по } X_j)\right),$$ где вероятность ветвления по X_j рассчитывается, исходя из предположения, что переменная X_j случайным образом извлекается из r переменных-кандидатов (общее число переменных равно p): $$P(\text{ветвление по } X_j) = P(X_j \text{ является кандидатом}) \times P(X_j \text{ выбрана}) = \frac{r}{p} \times \frac{1}{r} = \frac{1}{p}$$ Результаты теста говорят нам, превышает ли наблюдаемое количество использований X_j для ветвления теоретическое число использований в ситуации, когда переменная X_j выбирается из всех переменных случайным образом. Метрика `p-value` дополняет метрику `times_a_root`. Некоторые предикторы часто бывают в корне и поэтому с точки зрения метрики `times_a_root` будут наиболее важными, в то же время в целом по дереву такие предикторы могут использоваться редко и с точки зрения метрики `p-value` будут неважными. С другой стороны, некоторые предикторы могут редко использоваться в корне и почти бесполезны с точки зрения метрики `times_a_root`, но часто могут использоваться ниже по дереву и метрика `p-value` это покажет. Метрики `accuracy_decrease`, `gini_decrease`, `mse_increase` и `node_purity_increase` рассчитываются пакетом `randomForest`, если задана настройка `localImp=TRUE`, поэтому они просто извлекаются из модельного объекта. Метрики `accuracy_decrease` и `mse_increase` основаны на ухудшении качества предсказаний модели после перестановки значений переменной, а метрики `gini_decrease` и `node_purity_increase` – на изменении чистоты узлов после разбиения по переменной. Последние четыре метрики основаны на структуре случайного леса

Давайте с помощью функции `measure_importance` вычислим все возможные метрики важности для нашего примера.

```
# вычислим альтернативные метрики важности
importance_frame <- measure_importance(forest)
importance_frame
```

Сводка 5.33. Альтернативные метрики важности

```
   variable mean_min_depth no_of_nodes accuracy_decrease gini_decrease no_of_trees
1       age          1.700      106547      0.0386026821    1161.73868         500
2  atm_user          1.748       13017      0.0414558367     898.75617         500
3  cre_card          2.002       16907      0.0108975520     459.41408         500
4  curr_acc          2.284       18179      0.0104337508     282.88614         500
5  cus_leng          1.500       27174      0.1094811467    1945.48159         500
6  deb_card          3.990       37883      0.0009721533      63.48066         500
7  internet          3.742       29243      0.0032034169      60.70255         500
8  life_ins          3.018       15005      0.0054717208      97.15749         500
9    markpl          2.102       19038      0.0111006310     420.53910         500
10 mob_bank          2.772       23568      0.0047657665     143.44418         500
11 mortgage          3.004       24627      0.0037050994     117.04399         500
12  perloan          4.000       33254      0.0002275767      52.97052         500
13  savings          3.574       31567      0.0036513316      67.80267         500

   times_a_root      p_value
1            83 0.000000e+00
2           103 1.000000e+00
3            64 1.000000e+00
4            39 1.000000e+00
5           101 1.000000e+00
6             0 0.000000e+00
7            10 1.000000e+00
8            16 1.000000e+00
9            50 1.000000e+00
10           21 1.000000e+00
11           13 1.000000e+00
12            0 6.635096e-61
13            0 2.872773e-11
```

На сводке 5.33 видно, что согласно большинству метрик наиболее важными предикторами стали *Возраст [age]*, *Пользование банкоматом за последнюю неделю [atm_user]* и *Давность клиентской истории [cus_leng]*.

5.4.3. Многомерные графики для оценки важности предикторов

С помощью функции `plot_multi_way_importance()` можно построить многомерный график для оценки важности предикторов. Функция `plot_multi_way_importance()` имеет общий вид:

```
plot_multi_way_importance(importance_frame,
                          x_measure = "mean_min_depth",
                          y_measure = "times_a_root",
                          size_measure = NULL,
                          min_no_of_trees = 0,
                          no_of_labels = 10)
```

где

`importance_frame=`	Задает результат функции `min_depth_distribution`, полученный для модельного объекта или сам модельный объект
`x_measure=`	Задает метрику важности, откладываемую по оси x (по умолчанию используется показатель `mean_min_depth`)
`y_measure=`	Задает метрику важности, откладываемый по оси y
`size_measure=`	Задает метрику важности, отображаемую в виде размера точек
`min_no_of_trees`	Задает минимальное количество деревьев, в котором должен использоваться предиктор, чтобы быть включенным в анализ. Тем самым можно ограничить количество точек на графике
`no_of_labels=`	Задает минимальное количество наилучших переменных для присвоения текстовых меток (имен переменных). По умолчанию метки задаются для 10 наилучших переменных. Наилучшие переменные изображаются в виде точек синего цвета. Переменные отбираются при помощи функции `important_variables()` путем ранжирования по метрикам важности, используемым на графике. Переменных может оказаться больше заданного количества, если несколько из них получают один и тот же ранг

Давайте построим многомерный график с настройками по умолчанию.

```
# постороим многомерный график для
# оценки важности предикторов
plot_multi_way_importance(importance_frame)
```

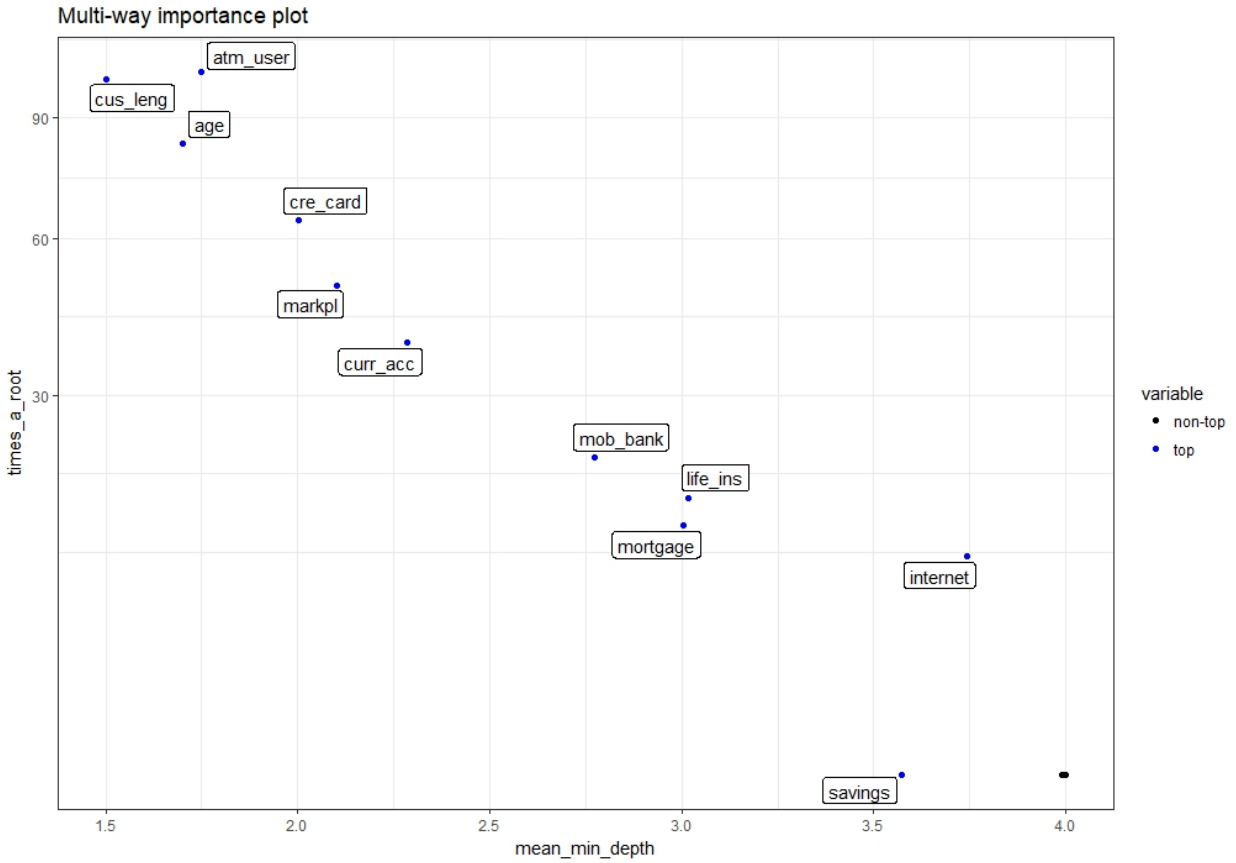


Рис. 5.25 Многомерный график для оценки важности предикторов (в качестве метрик важности используется частота использования предиктора в корне и среднее значение минимальной глубины)

На рис. 5.25 мы видим, что наиболее важные переменные *Давность клиентской истории [cus_leng]*, *Возраст [age]* и *Пользование банкоматом за последнюю неделю [atm_user]* сосредоточились в верхнем левом углу графика, им соответствуют высокие значения метрики `times_a_root` и низкие значения метрики `mean_min_depth`. Здесь мы видим вполне логичную зависимость: чем меньше средняя минимальная глубина ветвления по данному предиктору, тем чаще этот предиктор используется в корне дерева.

Теперь поменяем метрики важности, откладываемые по осям. Наш пример – это задача классификации, поэтому по оси x отложим `accuracy_decrease`, а по оси y – `gini_decrease` (для задачи регрессии по оси x мы могли бы отложить `mse_increase`, а по оси y – `node_purity_increase`). В качестве метрики важности, определяющей размер точек, выберем `times_a_root` – количество использований в корневом узле. Присвоим метки с именами переменных не всем, а только 5 наилучшим предикторам.

```
# строим вручную настроенный многомерный
# график для оценки важности предикторов
plot_multi_way_importance(importance_frame,
                          x_measure = "accuracy_decrease",
                          y_measure = "gini_decrease",
                          size_measure = "times_a_root",
                          no_of_labels = 5)
```

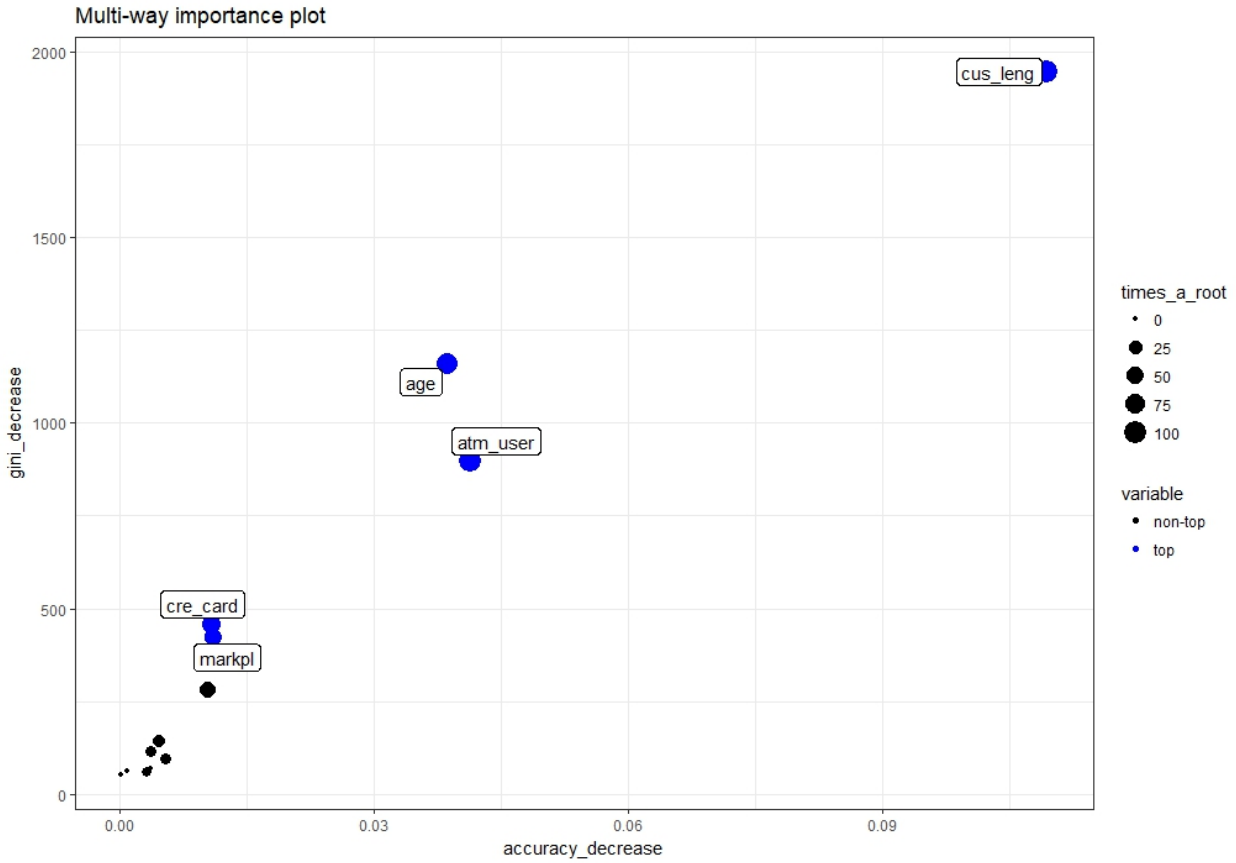


Рис. 5.26 Многомерный график для оценки важности предикторов (в качестве метрик важности используется уменьшение правильности и уменьшение меры Джини)

5.4.4. Парные графики для оценки корреляций между метриками важности

Многомерные графики допускают так много вариантов визуализации важности предикторов, что становится трудно выбрать оптимальный вариант. Преодолеть это затруднение можно путем изучения парных графиков, отображающих корреляции между отдельными метриками важности. Для построения таких парных графиков нужно воспользоваться функцией `plot_importance_ggpairs`.

```
# строим парные графики для оценки корреляций
# между метриками важности
plot_importance_ggpairs(importance_frame)
```

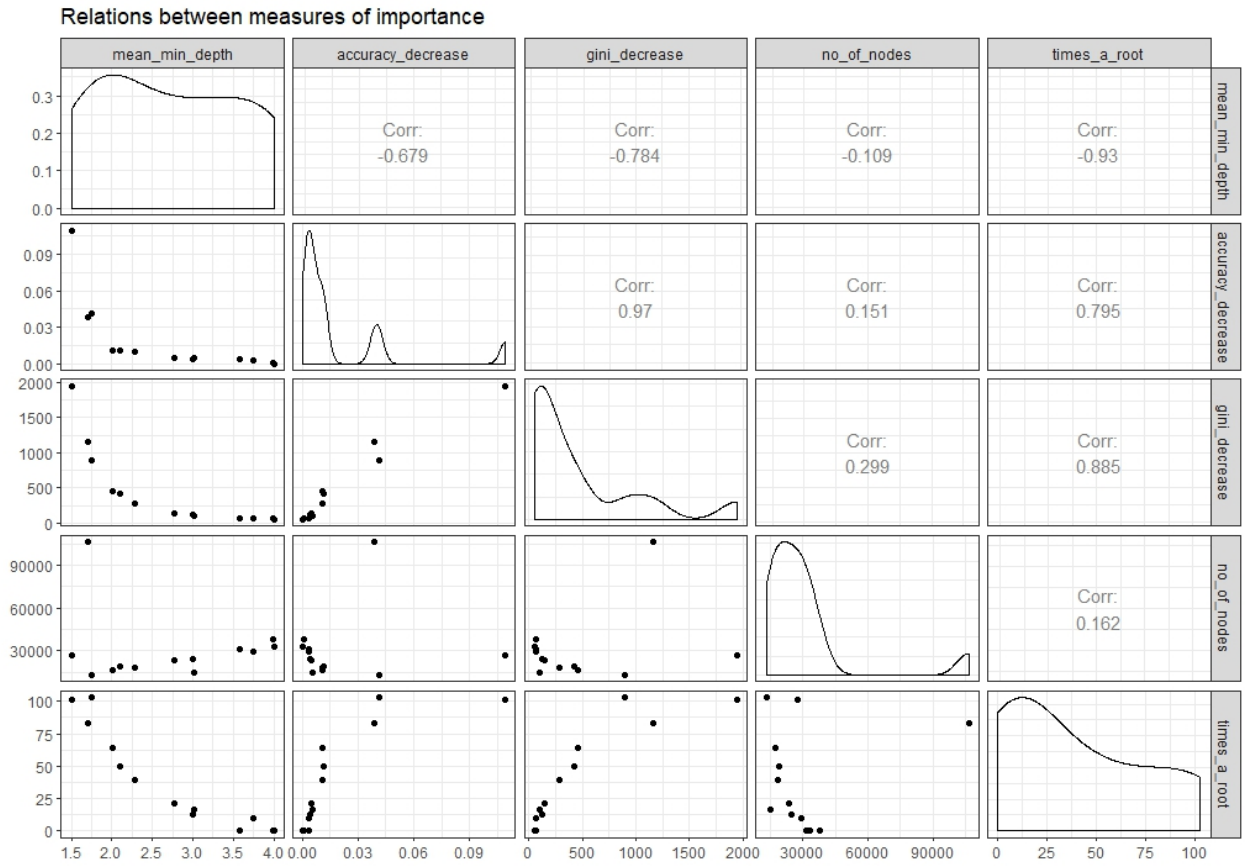


Рис. 5.27 Парные графики для оценки корреляций между метриками важности предикторов

На основе такого графика можно выбрать три наиболее согласованные метрики, а затем использовать их для построения многомерного графика (две метрики откладываем по осям x и y, а третья метрика задает размер точек). В нашем случае график показывает, что наиболее согласованными являются метрики `gini_decrease` и `accuracy_decrease`. Коэффициент корреляции между этими метриками равен 0,97. Более высоким значениям уменьшения меры Джини соответствуют более высокие значения уменьшения правильности. Также согласованными являются метрики `mean_min_depth` и `times_a_root`. У них – высокий отрицательный коэффициент корреляции, равный -0,93. Более высоким средним значениям минимальной глубины ветвления по данному предиктору соответствуют более низкая частота использования предиктора в корне. Еще одной парой согласованных метрик можно считать `gini_decrease` и `times_a_root` (коэффициент корреляции равен 0,885). Более высоким значениям уменьшения Джини соответствует более высокая частота использования предиктора в корне. Наконец, можно рассмотреть пару метрик `accuracy_decrease` и `times_a_root` (коэффициент корреляции равен 0,795). Более высоким значениям уменьшения правильности соответствует более высокая частота использования предиктора в корне.

Кроме точечных графиков и коэффициентов корреляции, функция `plot_importance_ggpairs()` также выводит кривые плотности распределения для каждой метрики, которые часто бывают очень асимметричными. Поэтому вместо исходных значений можно выполнить анализ рангов при помощи функции `plot_importance_rankings()`, которая добавляет на парные графики сглаживание LOESS.

```
# строим парные графики для оценки корреляций
# между метриками важности с добавлением
# сглаживания LOESS
plot_importance_rankings(importance_frame)
```

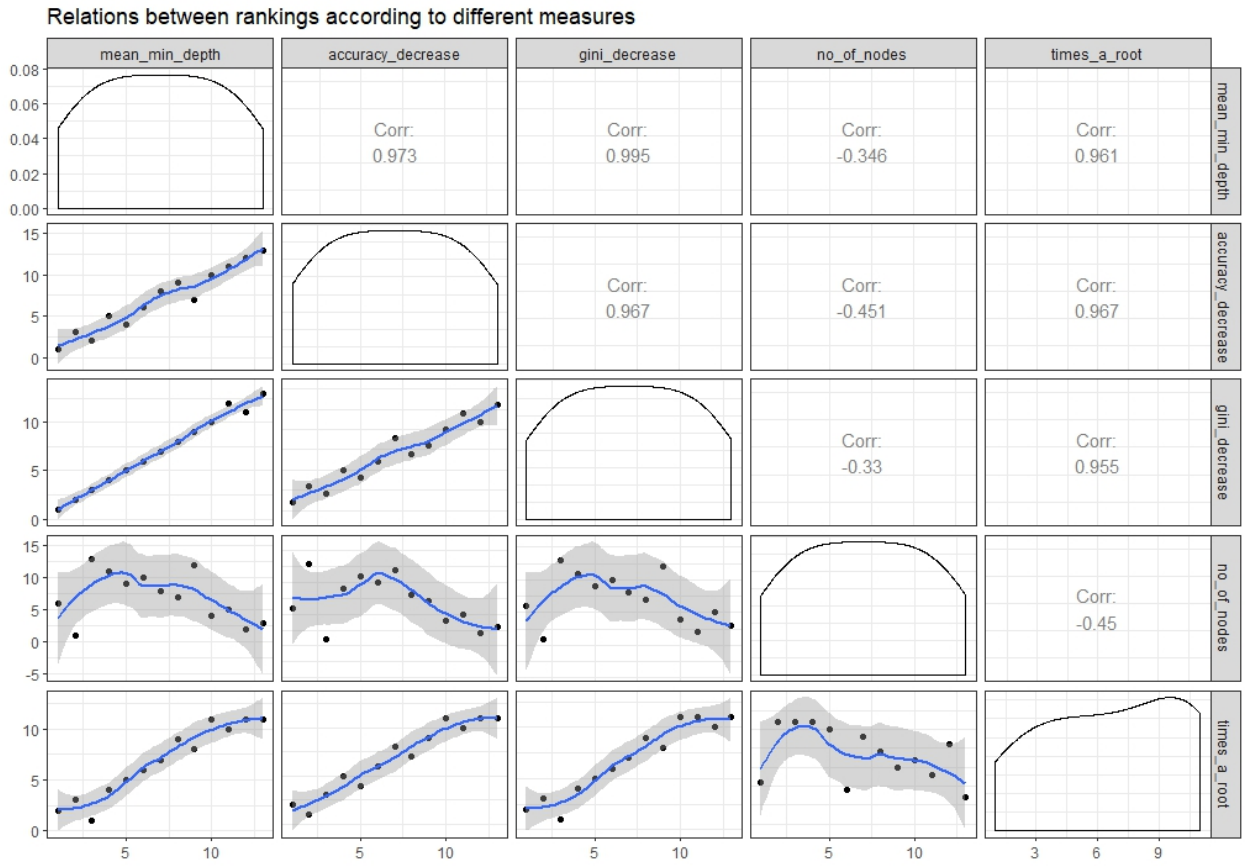


Рис. 5.28 Парные графики для оценки корреляций между метриками важности предикторов (используется сглаживание LOESS)

Вышеприведенные оценки плотности показывают, что мы получили более симметричные метрики важности (это не всегда возможно, например, для дискретных метрик важности типа `times_a_root` распределение будет по-прежнему асимметричным).

Сравнив рис. 5.27 и рис. 5.28, мы видим, что три пары метрик почти точно согласуются: `mean_min_depth` и `gini_decrease`, `mean_min_depth` и `accuracy_decrease`, `accuracy_decrease` и `gini_decrease`.

5.4.5. Графики взаимодействий между переменными

После выбора нескольких наиболее важных предикторов мы можем изучить взаимодействия имеющихся переменных с этими важными предикторами, то есть проанализировать ветвления, происходящие в поддеревьях, которые получили с помощью одного из выбранных предикторов. Мы вводим понятие «условная минимальная глубина». Условная минимальная глубина – это та же минимальная глубина, но посчитанная на поддеревьях, полученных после разбивки по другому предиктору, взаимодействие с которым нас и интересует.

Обычно рассматривают взаимодействия с наиболее важными предикторами. Чтобы извлечь имена 5 наиболее важных предикторов (важность в данном случае оценивается по безусловной средней минимальной глубине и количеству деревьев, в которых он участвует), воспользуемся функцией `important_variables`:

```
# извлекаем имена 5 наиболее
# важных предикторов
vars <- important_variables(importance_frame,
                            k = 5,
                            measures = c("mean_min_depth",
                                         "no_of_trees"))
```

Далее необходимо воспользоваться функцией `min_depth_interactions`. Она имеет общий вид:

```
min_depth_interactions(forest, vars,
                           mean_sample = "top_trees",
                           uncond_mean_sample = mean_sample)
```

где

`forest=`	Задает объект-модель
`vars=`	Задает переменные, перечисленные в `vars` (пропуски обрабатываются так же, как и при расчете безусловной минимальной глубины, способ контролируется параметром `mean_sample`). Если не задан параметр `vars`, вектор имен переменных будет определен путем вызова `important_variables(measure_importance(forest))`
`mean_ sample=`	Задает способ вычисления условной средней минимальной глубины. Возможные значения: `"all_trees"`, `"top_trees"` и `"relevant_trees"`.
`uncond_mean_ sample=`	Задает способ вычисления безусловной средней минимальной глубины. Возможные значения: `"all_trees"`, `"top_trees"` и `"relevant_trees"`.

Функция `min_depth_interactions` возвращает таблицу, состоящую из 6 столбцов:

- `variable` – предиктор набора данных;
- `root_variable` – предиктор, перечисленный в `vars` (обычно выбираем ограниченное количество наиболее важных предикторов);
- `mean_min_depth` – условная средняя минимальная глубина;
- `occurrences` – встречаемость взаимодействия (количество деревьев, где встретилось данное взаимодействие);
- `interaction` – взаимодействие предиктора, приведенного в столбце `variable`, с предиктором, приведенным в столбце `root_variable`;
- `uncond_mean_min_depth` – безусловная средняя минимальная глубина.

Итак, давайте построим таблицу взаимодействий для нашего примера и для компактности выведем первые 6 строк таблицы, предварительно отсортировав взаимодействия по встречаемости.

```
# строим таблицу взаимодействий
interactions_frame <- min_depth_interactions(forest, vars)

# выведем первые 6 строк таблицы
head(interactions_frame[order(interactions_frame$occurrences, decreasing = TRUE), ])
```

Сводка 5.34. Таблица взаимодействий предикторов (фрагмент)

	variable	root_variable	mean_min_depth	occurrences	interaction	uncond_mean_min_depth
2	age	atm_user	1.178357	499	atm_user:age	1.70
4	age	cus_leng	1.070140	499	cus_leng:age	1.70
3	age	cre_card	1.214345	496	cre_card:age	1.70
5	age	markpl	1.247848	496	markpl:age	1.70
29	deb_card	cus_leng	2.508329	495	cus_leng:deb_card	3.99
1	age	age	1.567174	494	age:age	1.70

Графически результаты можно представить следующим образом:

```
# визуализируем полученные данные
plot_min_depth_interactions(interactions_frame)
```

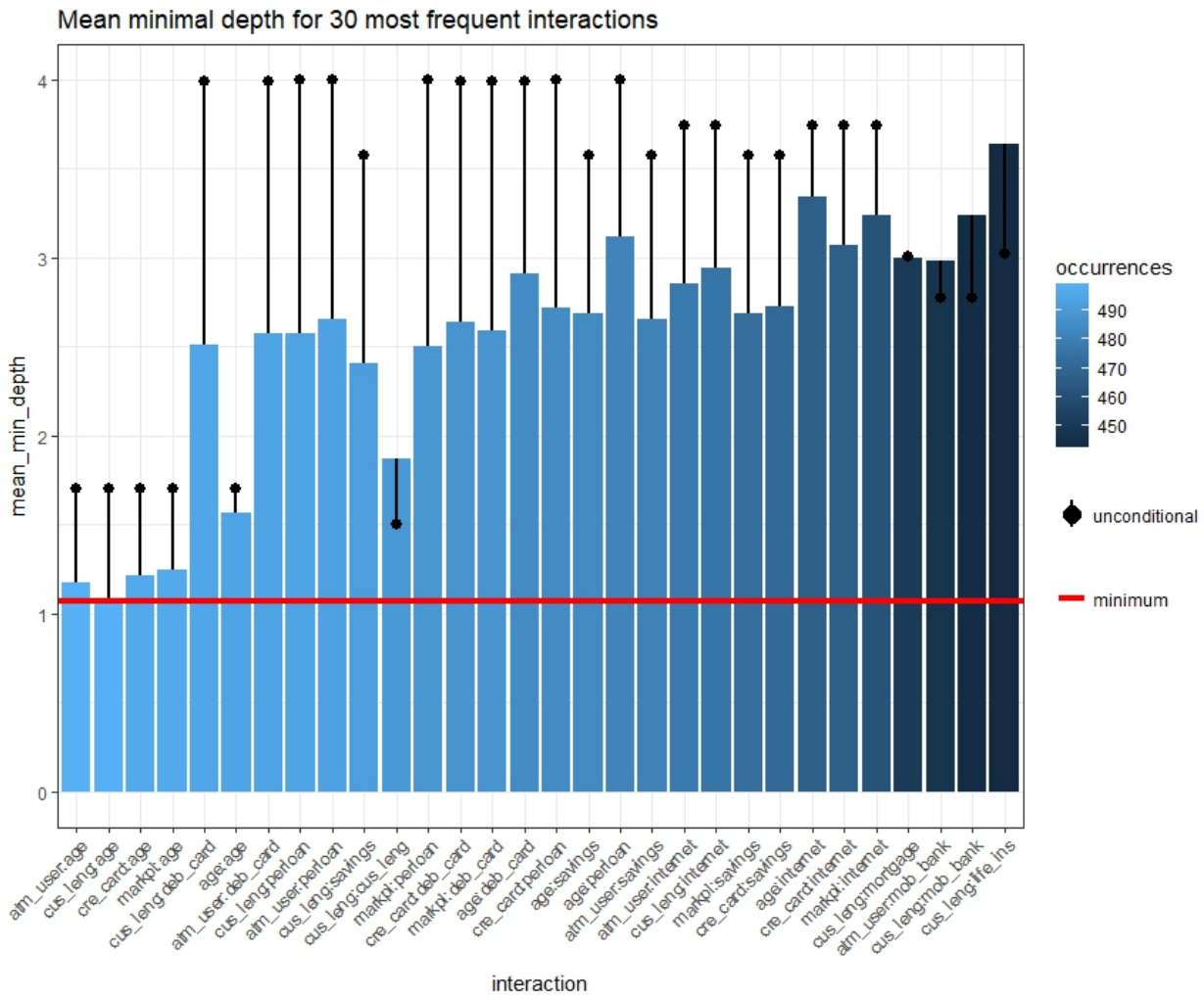


Рис. 5.28 График средней минимальной глубины для 30 самых часто встречающихся взаимодействий

Обратите внимание, что взаимодействия упорядочены в порядке уменьшения встречаемости. Наиболее часто встречающимися взаимодействиями являются *atm_user: age*, *cus_leng:age*, *cre_card:age* и *markpl:age*. Как правило, наиболее часто встречающиеся взаимодействия будут иметь наименьшие средние значения условной минимальной глубины.

Как правило, график содержит массу информации и его можно интерпретировать разными способами, но решающую роль имеет метод, использующийся для вычисления условной (параметр `mean_sample`) и безусловной (параметр `uncond_mean_sample`) средней минимальной глубины. Значение `"top_trees"`, используемое по умолчанию штрафует редко встречающиеся взаимодействия. Разумеется, можно попробовать значения `"all_trees"`, `"top_trees"` и `"relevant_trees"` для вычисления средних значений как условной, так и безусловной минимальной глубины.

5.4.6. Получение отчета по построенному случайному лесу

С помощью функции `explain_forest` можно автоматически сгенерировать отчет по построенному случайному лесу. Отчет будет записан в виде html-файла. Функция `explain_forest` имеет общий вид:

```
explain_forest(forest,
               interactions = FALSE,
               data = NULL, vars = NULL,
               no_of_pred_plots = 3,
               pred_grid = 100,
               measures = if (forest$type == "classification")
                 c("mean_min_depth",
                   "accuracy_decrease",
                   "gini_decrease",
                   "no_of_nodes",
                   "times_a_root")
               else c("mean_min_depth",
                      "mse_increase",
                      "node_purity_increase",
                      "no_of_nodes",
                      "times_a_root"))
```

`forest=`	Задает объект-модель
`interactions=`	Задает рассмотрение взаимодействий (может занять много времени)
`data=`	Задает данные, на которых был обучен лес, обязателен, если `interactions=TRUE`
`vars=`	Задает вектор имен переменных, взаимодействия с которыми нужно рассмотреть. Если не задан, вектор имен переменных будет определен путем вызова `important_variables(measure_importance(forest))`
`no_of_pred_plots=`	Задает количество самых часто встречающихся взаимодействий количественных переменных для визуализации прогнозов с помощью теплокарт
`pred_grid=`	Задает количество точек для построения теплокарт с помощью функции `plot_predict_interaction`
`measures=`	Задает вектор названий метрик важности для построения парных графиков

Решая задачу регрессии, вы можете построить тепловую карту зависимости прогнозного значения целевой переменной от двух переменных, взаимодействие между которыми нас интересует. Это можно сделать с помощью функции `plot_predict_interaction()`.