Это, кстати, именно тот принцип выбора точки отсечения, который мы упоминали в первом разделе работы когда речь шла построении классификаторов для данных примера. Только там удавалось добиться полной точности классификации, что было следствием $KS = 1$ в формуле (2.3), т.е. находилась точка, в тех обозначениях p' которая оставляла все наблюдения с проявлением исследуемого признака «позади», не достигнув еще ни одного из наблюдений без исследуемого признака.

Важным является вопрос: какую статистику лучше использовать в каком случае? В формуле для *AUC* используются FPR и TPR для всех точках отсечения, что характеризует эту статистику как описание классифицирующей способности для всего ряда вероятностей. *KS* же использует всего одну вероятность, и по ней нельзя сделать вывода, есть ли другие точки отсечения с приемлемыми свойствами, тем не менее, она указывает на статистически оптимальную точку отсечения. То, на какую статистику стоит обращать внимание при оценке классифицирующих свойств, сильно зависит от прикладной области, так в кредитном скоринге, при выборе точки отсечения, нередко опираются не столько на статистику, сколько на текущую политику банка, потому важнее иметь много допустимых точек отсечения, от сюда, больше внимания следует уделять статитике *AUC*. За *KS* остается еще одно очень важное преимущество – за ним стоит строгий статистический тест. В рамках выбора готовой модели это малое преимущество, но в рамках оценки классифицирующих свойств отдельных предикторов и, как следствие, их отборе, как будет показано далее, это становиться ключевой возможностью.

Все вышесказанное естественно для готовой модели, а у нас, на данном этапе, задача – отбор показателей в модель. Для оценки классифицирующей способности отдельного предиктора с помощью ROC анализа рассмотрим однофакторную модель вида:

$$\hat{y} = \begin{cases} N, x \in A_N; \\ P, x \in A_P. \end{cases} \quad (2.5)$$

где $\hat{y}$ – предсказанное значение оклика;

$x \in A_N \cup A_P$ – переменная классифицирующая способность которой оценивается;

A_N – множество значений x при котором предсказывается отсутствие исследуемого признака;

A_P – множество значений x при котором предсказывается наличие исследуемого признака.

На данном этапе можно логику разделить на две ветви – исследование числовых и исследование номинативных переменных. Начнем с рассмотрения числовых. Для названного случая формула (2.5) примет вид:

$$\hat{y} = \begin{cases} N, x \le x' \\ P, x > x'. \end{cases} \quad (2.6)$$

где x' – точка отсечения по переменной x.

В отношении кредитного скоринга получается модель со следующим принципом принятия решений: для всех наблюдений у которых наблюдаемое значение меньше либо равно некоторого порога x' принимаем решение о выдаче кредита, в противном случае, понимается решение об отказе.

Заметим, что рассуждения предыдущего абзаца очевидно справедливы для случая, когда с ростом исследуемого показателя ожидается увеличение проявления исхода с дефолтом (P). В действительности же, даже чаще, встречается обратная ситуация – с ростом исследуемой переменной ожидается падение проявления исходов с дефолтом. Например, с ростом уровня дохода клиента физического лица, справедливо ожидать от него большей стабильности платежей и, как следствие, меньшую вероятность выхода в дефолт.

В сущности, приведенное замечание не меняет основных идей анализа – если на рисунках Е.1, Е.2 поменять местами распределения P и N, то диаметрально поменяется порядок изменяя *FPT, TPR*, описанный выше. Что приведет к тому, что ROC кривая выгнется в обратную сторону. Такая ситуация представлена на рисунке Е.4.

Очевидно, что, в таком случае, нельзя сказать, что модель не имеет классифицирующей мощности, хотя AUC, указывает на то, что классифицирующая мощность модели ниже чем в случае случайного угадывания (нижняя часть рисунка Е.3).

Если бы такая модель использовалась для предсказаний, то логично было бы поменять знаки в выражении (2.6). Но, в нашем случае, мы лишь оцениваем классифицирующую способность предиктора с использованием AUC. Потому введем следующее окончательное определение AUC:

$$AUC = \begin{cases} AUC', AUC' \geq 0.5; \\ 1 - AUC', AUC' < 0.5, \end{cases}$$

где AUC' – эмпирическая функция распределения наблюдений с проявлением признака по PD';

AUC – эмпирическая функция распределения наблюдений без проявления признака по PD'.

Так же, заметим, что ранее речь шла о распределении наблюдений с различным состоянием признака по *PD*, а сейчас речь идет о распределении вдоль некоторой численной переменной *x*. В сущности, снова нет большой разницы – наблюденный *x* можно нормировать и, по смыслу, будет получен тот-же *PD*.

Более сложным является подобный анализ для номинативной переменной. Множества A_N и A_P становятся дискретными конечными множествами, в объединение которых включены все уровни исследуемой переменной.

Обозначим и пронумеруем все уровни исследуемой переменной как $t_i, i = \overline{1, m}$. Начинаем «движение» от наиболее лояльной ситуации, когда при любом

уровне исследуемого предиктора в модели (2.5) принимается решение о том, что $\hat{y} = N$, т.е. $A_N = \{t_1, t_2, \dots, t_m\}$, $A_P = \emptyset$. Далее в множество A_P в обратном порядке вводятся t_i, и выводятся из A_N – модель становиться строже. Так продолжается до тех пор пока мы не приходим к наиболее строгой ситуации, при которой, для любого уровня исследуемого предиктора принимается решение о том, что $\hat{y} = P$, т.е. $A_N = \emptyset$, $A_P = \{t_1, t_2, \dots, t_m\}$. В более понятной форме эта логика описана в таблице Ж.1.

Используя описанный механизм можно, строить ROC кривую и вычислять показатель *AUC*. На рисунке Ж.1 нанесена ROC кривая в общем виде и различными заполнениями (которые понадобиться в дальнейшем повествовании) обозначена площадь под ней. ROC кривая состоит из ряда линейных участков, каждый из которых знаменует переход к новому уровню предиктора и образует с осью абсцисс трапецию площадь которой обозначим S_q.

Может показаться, что просто суммируя S_q можно получать AUC для номинативного предиктора, однако, при использовании этого метода на практике, быстро выявляется нюанс описанный в следующем утверждении.

Утверждение 1. При изменении порядка включения предикторов в группу P вид ROC кривой может меняться и, как следствие, может меняться показатель *AUC*.

Это утверждение наталкивает на вопрос – какую последовательность включения уровней использоваться для вычисления статистики *AUC*? Для того, чтобы ответить на этот вопрос введем следующее утверждение.

Утверждение 2. Если строить ROC кривую по логике постепенного уменьшения лояльности модели (так как это записано в таблице Ж.1), то при включении уровней в порядке убывания частот проявления исследуемого признака в отклике *AUC* будет максимален.

В виду и так заметной перегруженности этого подраздела теоритическими сведениями не будем приводить строгого доказательства этого утверждения – его можно получить используя сведения школьного курса алгебры и геометрии. Оговоримся лишь, что доказательство строиться на том, что зеленая площадь рисунка Ж.1 не зависит от последовательности включения *i-го* уровня с меньшей частотой проявления исследуемого признака и *j-го* уровня с большей частотой проявления. Синяя тоже, но по другим причинам. Любая трапеция красной площади может быть представлена формулой:

$$S_s = \frac{n_s^N}{2N'P'}\left(2\left[\sum_{k=s+1}^{j-1} n_k^p + n_j^P + \sum_{k=j+1}^{m} n_k^p\right] + n_s^P\right), s = \overline{\imath + 1, j}.$$

В случае прямого включения уровней. А в случае, если поменять порядок включения *i-го* и *j-го* уровней между собой:

$$S'_s = \frac{n_s^N}{2N'P'}\left(2\left[\sum_{k=s+1}^{j-1} n_k^p + n_i^P + \sum_{k=j+1}^{m} n_k^p\right] + n_s^P\right), s = \overline{\iota+1, J}.$$

Очевидно, что два последних выражения отличаются на второе слагаемое в квадратных скобках. Учитывая вышесказанное, заметим, что $n_i^P < n_j^P$, от сюда и $S_s > S'_s$, что ведет к истинности второго утверждения.

В целом, результат рассуждения такой: для получения показателя классифицирующей способности *AUC* для номинативного предиктора, следует включать урони в порядке возрастания проявления исследуемого признака – в таком случае *AUC* будет максимальный. Нужно это, конечно, не для того, чтобы завысить показатель *AUC*, а чтобы из множества возможных оценок выбрать единственную.

У *AUC* в данном случае имеется один большой минус – нет критерия при котором можно считать, что показатель связан с объясняемой переменной или придерживаться обратного утверждения. Тут на помощь приходит введенная выше связанная статистика KS – она распределена в соответствии с распределением Колмогорова-Смирнова, от того возможность проведения статистического тестирования.

Нас, в данном случае, очевидно, интересует двухвыборочная вариация теста – сравнение выборок, полученных для наблюдений с проявлением исследуемого признака и без него. Такой тест имеет следующую постановку: пусть X и Y две независимые случайные переменные, чьи функции распределения соответственно F и G неизвестны. Выдвигается нулевая гипотеза [14]:

$$H_0: F(x) = G(x), \forall x \in R^d.$$

Против альтернативной гипотезы:

$$H_1: F(x) \neq G(x), \forall x \in R^d.$$

Представленная выше KS статистика распределена в соответствии с распределением Колмогорова-Смирнова с числом степеней свободы вычисляемым по формуле:

$$\frac{nm}{n+m},$$

где n – число наблюдений в выборке X;

m – число наблюдений в выборке Y.

Нулевая гипотеза принимается в том случае, если реальная статистика меньше теоритической вычисленной с заданной доверительной вероятностью. Или можно посчитать p-значение, как значение функции распределения Колмогорова-Смирнова с указанным числом степеней свободы в точке KS.

Все описанные принципы положены на практику классом python «classification_power_predictor», со всеми сопутствующими методами расположившемся в отдельном репозитории[2] в сети интернет. Он позволяет произвести подсчет *AUC, KS* и *p*-значений двувыборочного *KS* теста, для всех переменных переданного «pandas.DataFrame» и столбца отклика в виде «pandas.Series».

В конструктор этому классу следует первым аргументом предать набор предикторов и столбец отклик, например так:

```
import classification_power_predictor as cpp
my_cpp = cpp(
    data.drop('Y', axis = 1),
    data['Y'].replace({0:"Нет дефолта", 1:"Дефолт"})
),
```

где data – экземпляр класса «pandas.DataFrame» с структурой данных соответствующей той, что мы организовывали выше.

Заметим некоторую тонкость последней записи – столбец «Y» изначально идет как набор нулей и единиц, класс устроен так, чтобы выводить урони так как они названы в столбце отклике, потому используя метод «replace» мы сложно читаемые 0 и 1 превращаем в «Нет дефолта» и «Дефолт» соответственно.

После получения экземпляра класса «classification_power_predictor» обязательно нужно вызвать его метод «update_predictors». После того в объексте будет проинициализировано поле «result_DF», которое представляет собой «pandas.DataFrame» структуры, подобной той, что представлена в таблице И.1. В таблице И.1 первое поле указывает название предиктора, поля 2-4 – статистики вокруг которых строилось рассуждение выше: *AUC, KS* и *p*-значение двувыборочного теста Колмогорова-Смирнова. Поля 5-7 дают информацию о пропусках: «Пропуски» указывает количество пропусков в каждом столбце, последние два поля описываю структуру распределения пропусков по уровням отклика.

Заметим, что в целях размещения в работе, таблица была сильно урезана, так в оригинальном варианте такой набор колонок формируется для каждого уровня отклика. В данном случае, информации потеряно не было – в случае бинарной классификации *AUC* и *KS* для обоих классов будут одинаковые.

В поле «result_DF» располагается также информация о показателе *Gini,* который составляет альтернативу *AUC* и вычисляется по формуле:

$$Gini = (AUC - 0{,}5)/2.$$

Еще, по умолчанию, можно получить боле подробную информацию о наполненности данных и структуре пропусков.

[2] https://github.com/Dranikf/classification_power_predictor

Еще одно важное замечание – при вычислении статистик классифицирующей способности для номинативных переменных пропуск воспринимается как отдельный уровень. Это сделано, для того, чтобы не терять данных. Предполагается, что стремление кредитополучателя скрыть о данные о себе может быть признаком того, что эти данные могут занизить его шанс на получение кредита.

При более подробном рассмотрении таблицы И.1 можно найти ряд предикторов, для которых двувыборочный тест Колмогорова-Смирнова ведет к принятию нулевой гипотезы об отсутствии различий распределений наблюдений с дефолтом и без него, при доверительной вероятности в 0,01. В список таких показателей вошли: «Автомобиль год выпуска3», «Количество детей», «Отношение к банку», «Работа последнее место стаж лет», «Работа уровень дохода BYR», «Количество действующих договоров обеспечения», «Число авто» и «Есть авто».

Сразу было произведено удаление этих столбцов. Далее проводился более тонкий анализ для каждого из предикторов по отдельности. По результатам было принято решение произвести удаление столбца «Автомобиль год выпуска», так как он имел меньший показатель *AUC* по сравнению с базовыми для него столбцами «Автомобиль год выпуска 1» и «Автомобиль год выпуска 2». Аналогично «Ежедневный платеж» как отношение «Сумма договора» к «Срок депозита в днях» уступает базовому показателю по значению *AUC*.

Похожая ситуация с показателями «Столица», «Областной центр», «Адрес проживания – Населенный пункт». Несмотря на то, что они были созданы в отрыве от показателя «Тип населенного пункта», очевидно, что они неразрывно с ним связаны и, вместе с тем, заметно уступают ему по классифицирующей способности, потому были удалены.

Следующим шагом, после удаления показателей со слабой описательной способностью, будет борьба с пропусками, перечислим возможные пути решения и проблемы связанные с каждым из них:

- Удаление строк содержащих пропуски по каждому показателю. Если некоторый показатель слабо заполнен, то для его включения в модель понадобиться удалить много сток, и будет потеряна информация указанная по удаленным записям в других столбцах;
- Удаление столбцов с большим числом пропусков. Так же ведет к недостаточности данных, но в другом смысле – теряются «ракурсы» с которых мы можем смотреть на исследуемую проблему;
- Заполнение пропусков данных альтернативными значениями – ведет к искажению реальных закономерностях лежащих в данных может привести к улучшению модели «на бумаге», но в реальных условиях толку от нее будет мало;
- Рассмотрение пропусков в номинативных показателях как отдельных уровней. Так же может вести к искажениям в данных, но лучше применять такой подход чем выбрасывать показатели.

В общем нет универсального рецепта – всегда приходиться принимать решения опираясь на конкретные данные и, даже, возвращаться к этому этапу в процессе построения модели.

Логически получилось связать показатели «Воинская служба» и «Пол». Очевидно, что у большинства женщин в графе «Воинская служба» должно быть проставлено «невоеннообязанный» и если соответствующие пропуски заполнить этим значением в подавляющем большинстве случае это будет правильно.

Для любой числовой переменной, превращение ее в номинативную переменную и, как результат, сохранение наблюдений с пропуском в этом столбце можно было осуществить двумя путями.

Можно превратить в бинарные переменные с уровнями «нет данных»/ «есть данные».

Второй подход несколько сложнее – для каждой переменной подсчитаны статистики *KS*. Выше описывалось, как выбрать точку отсечения для готовой модели с использованием *KS* статистики по формуле (2.4). Можно попробовать использовать такой подход для того, чтобы определиться с точкой разделения численной переменной на два уровня номинативной переменной, и отдельно будет использован уровень «нет данных», таким образом получается номинативный показатель с тремя уровнями.

Еще один способ обойти проблемы связанные с пропусками в числовых предикторах – записать вместо пропусков число ноль. В некоторых случаях это может быть оправдано, например, в случае показателя «Работа последнее место стаж лет» некоторые кредитополучатели при заполнении анкеты могли решить, что в случае отсутствия стажа (0 лет) следует в этом поле ничего не указывать.

Применим все три операции к каждой переменной численного типа из оставшихся на данный момент в выборке. И потом, опираясь на здравый смысл и подсчитанные статистики классифицирующей способности, примем решения о том в какой форме та или иная переменная должна участвовать в модели и должна ли участвовать вообще. В таблице И.2 результаты применения класса «classification_power_predictor» к таким переменным. В ней каждая численная переменная исходного набора переменных содержит еще 3 дочерние переменные. Первая группа с припиской «(бинарная)» представляет разделение на «нет данных»/«есть данные», вторая группа с припиской «(пропуск => 0)» предполагает замену пропусков нулями и последняя отмечена как «(разбивка по KS)» предполагает, что для разделения использовалась формула (2.4).

Показатель «Автомобиль год выпуска1» по классифицирующей способности сам по себе достаточно хорош, но имеет очень много пропусков. Было принято решение использовать его вариацию с разделением по *KS* так как она в сравнении с остальными производными показателями сохраняет высокий *AUC*. Аналогичное рассуждение можно приложить к показателям: «Количество запросов в КБ за последние 30 дней», «Максимальный срок, на который заключался договор, в годах», «Общее количество запросов в КБ» и «Срок кредита в днях».

Показатель «количество действующих кредитных договоров» даже улучшил свой *AUC* при замене пропусков нулями. Такое преобразование имеет за собой логическое основание – при отсутствии кредитных договоров многим потенциальным кредитополучателям может показаться логичным не заполнять соответствующее поле, потому оставим именно этот вариант показателя. Аналогичное рассуждение можно приложить к показателям: «Максимальное количество дней просрочки» и «Сумма кредитных лимитов (пропуск => 0)».

Отдельного обсуждения заслуживает показатель «Сумма договора». Названный показатель несколько прибавляет в значении *AUC* в случае замены пропусков нулями, но вообще, сомнительно качество записей в которых не указана даже сумма договора – сами информационные системы банка должны сохранить это значение. В результате для этого показателя было принято решение сохранить исходную форму несмотря на потерю около 10% записей от исходного числа.

Все оставшиеся показатели было решено не использовать в модели – у них недопустимо много пропусков и показатели *AUC* очень слабые.

Последним шагом станет возвращение сохраненным производным показателям имен родительских для них показателей.

В этой главе рассмотрен полный цикл подготовки данных к моделированию: загрузка данных, изучение их структуры и приведение типов данных, исключение невозможных значений и выбросов, анализ классифицирующей способности и отбор предикторов, исключение пропусков.

Изучение структуры данных проводилось с использованием подготовленного инструмента – функции языка программирования python3 «get_data_frame_settings» который позволяет получать таблицы вида таблиц приложения Г. На основе такого анализа были выдвинуты предположения, которые легли в основу создания новых показателей. Развитие некоторых показателей показалось неперспективным направлением, потому они были удалены. Даты получили численный тип данных, как число дней от базовой даты, номинативные показатели с очень большим числом уровней либо были укрупнены либо удалены в связи со сложностью возможных алгоритмов укрупнения и логической неперспективности для решения поставленной задачи. В связи с оторванностью от остальных записей исследуемого набора данных были исключены записи содержащие информацию о кредитах типа овердрафт вместе с показателем позволяющим произвести соответствующее разделение.

Касательно вопроса недопустимых значений можно обратиться к таблице Д.2, а именно к ее строкам использующимся для описания показателей «Работа последнее место стаж лет», «Работа уровень дохода BYN», «Сумма договора» и «Ежедневный платеж», они по смыслу, очевидно, не могут содержать отрицательные значения, однако по проведенному исследованию получается, что есть записи с отрицательными числами. Записи с невозможными значениями были заменены на пропуски.

Так же наше внимание привлекли показатели с подозрительно большими размахами: «Количество фактов просрочки по основному долгу»,

«Максимальное количество дней просрочки», «Общее количество запросов в КБ», «Сумма кредитных лимитов», «Сумма договора» и «Ежедневный платеж». В подтверждение этой идеи мы приводим таблицу 2.2 – значение 75-ой персетиля и максимальное нередко различаются более чем в 100 раз. Выбросом было принято считать значение не попадающее в интервал (2.2). Выбросы были также заполнены пустыми значениями.

Второй подраздел данной главы был посвящен преимущественно анализу классифицирующей способности предикторов. Были описаны показатели *AUC* и *KS*. Главная цель этих показателей выбор готового классификатора, но мы предложили для каждого показателя рассмотреть однофакторную модель вида (2.5) и рассчитанные для нее показатели воспринимать как оценки классифицирующей способности соответствующих предикторов.

Для того, чтобы в 3 строки кода проводить описанный анализ был создан класс языка программирования python3 «classification_power_predictor». Результаты его работы выражаются в таблицах приложения И. В результате были исключены все показатели для которых двухвыборочный тест Колмогорова-Смирнова приводил к принятию нулевой гипотезы о том, что распределения наблюдений с дефолтом и без по исследуемому показателю не различаются. Из всех связанных между собой показателей были выбраны те, которые имеют лучшие оценки классифицирующей способности, остальные были удалены.

В контексте избавления от пропусков в основном использовался метод приведения численных показателей к номинативной форме, и для всех номинативных переменный пропущенное значение воспринималось как новый уровень переменной. Особо интересен подход к разбиению численной переменной с использованием точки соответствующей *KS* статистике (формула (2.4)).

Таким образом, мы пришли к структуре данных описанной таблицей Д.4, а классифицирующие способности отдельных предикторов представлены в таблице И.3.

К сожалению невозможно гарантировать, что во всех моделях будет использован именно точь в точь этот набор данных – нередко приходиться подгонять данные в зависимости от ситуации. Но по умолчанию используется именно этот набор данных, все преобразования будут кратко описаны.

3 Построение и валидация модели

3.1 Программное описание модели и алгоритма обучения

3.1.1 Основные элементы модели в pytorch

В этой главе мы сконцентрируемся на прикладных вопросах создания моделей искусственных нейронных сетей. На сегодняшний день нет готовых пакетов которые позволяют полноценно формировать модели нейронных сетей, вызвано это тем, что ходы которые делают инженеры соответствующей квалификации очень разнообразны и зачастую, даже, уникальны. Потому, удобнее всего использовать языки программирования для построения таких формул. Однако нам не придется все писать с нуля, разработаны и распространяются как open source проекты, ряд отличных библиотек языка программирования python3. Мы будем использовать разработанный кампанией facebook пакет-расширение pytorch [11]. Выбор пал на него благодаря его относительной простоте, по сравнению с разработанным в google пакетом tensor flow, и гибкости, по сравнению с надстройкой над tensor flow под названием keras.

Заметим, что эта глава будет содержать куда больше технических деталей, так как в ней описывается центральный для этой работы алгоритм.

Весь и самый актуальный код для построения модели распложен в указанном выше репозитории в папке «model». Ключевые его элементы будут представлены в приложении И и подробно описаны в этом разделе.

Любая искусственная сеть, формируемая в pytorch, обязательно должна содержать следующие элементы:

- Класс, наследник класса «torch.nn.Module» описывающий в общем виде нейронную сеть требуемой идентификационной формы и, соответственно, экземпляр этого класса;
- Целевая функция, в виде экземпляра одного из классов сформированных в модуле «torch.nn». Полный список готовых целевых функций представлен на сайте документации pytorch [15];
- Так называемый оптимизатор – алгоритм оценки параметров модели, в виде экземпляра одного из классов модуля «torch.optim». Полный список возможных оптимизаторов представлен на сайте документации pytorch [16].

В первой главе работы, касаемо идентификационной формы модели мы определились лишь с тем, что выходной слой представляет собой единственный сигмоидальный нейрон, а в срытых слоях содержатся ReLU нейроны. Неопределенными остаются число скрытых слоев нейронной сети и количесво нейронов в них входящее. Эти факты прямо повлияли на описание класса нейронной сети.

Названный класс представлен на листинге К.1. Подробно разберем этот класс. Метод «__init__» в python – зарезервированный метод под конструктор. Конструктором называются методы вызываемые при создании экземпляра класса. В нашем случае конструктор принимает список «neur_counts» в качестве

аргумента. Это должен быть список из целых чисел каждое число представляет собой число нейронов в соответствующем слое начиная с входного слоя. В нем не следует указывать выходной слой, это по умолчанию один нейрон. Так для создания экземпляра описывающего сеть с тремя входами и двумя скрытыми слоями, по 2 и 3 нейронами соответственно надо будет записать:

ResultNet([3,2,3]).

Такой подход обеспечит нам возможность создания моделей различных идентификационных форм «на лету» – можно будет в циклах перебирать различные идентификационные формы не прибегая к изменению кода класса.

Библиотека pytorch использует несколько другой тип формального представления сети. Нейроном называется то, что мы в торической главе называли сумматорной функцией, а потом к ним применяются такие преобразования, которые мы называли активационной функцией. В результате, никакой разницы, но сначала может создавать некоторый диссонанс теории и практики. Так конструктор создает словарь «layers» в котором последовательно указываются слои, как экземпляры класса «nn.Linear», реализующие линейную комбинацию (1.11). Для создания нужно указать число выходных активаций предыдущего слоя и число выходных активаций создаваемого слоя (число нейронов в этом слое). Созданный нами класс все это проделывает автоматически. Кроме того, мы заполнили веса и свободные члены каждого слоя случайными значениями, так формируется точка с которой алгоритм будет начинать процесс оптимизации. Лучшим решением для такой ситуации было бы указать примерное решение, но у нас нет идей как оно должно выглядеть, потому будет заполнение случайными весами.

Класс нейронной сети в pytorch должен содержать в себе метод «forward». Основная цель этого метода применять к матрице X входных данных нейронную сеть в текущем ее состоянии. Тут последовательно, в цикле перебираются слои созданные конструктором (все кроме последнего), на каждой итерации X подставляется в советующий слой, а результат в функцию «torch.nn.functional.relu», которая и представляем собой ReLU преобразование. Полученная матрица заменит X вышедший из предыдущего слоя. Последним действием применяется «torch.sigmoid», который реализует активацию выходного нейрона, полученное значение становиться возвращаемым значением метода «forward».

Что касается целевой функции, мы использовали бинарную кросс-энтропию [18]. Эта функция определяется следующим образом:

$$H(y,p) = \frac{1}{N}\sum_{i=1}^{n} y_i \log(p_i) + (1-y_i)\log(1-p_i),$$

где y_i – реально наблюденное значение признака, $y_i \in \{0,1\}, i = \overline{1,n}$ для i-*го* наблюдения;

p_i – предсказанная вероятность одного из уровней, $p_i \in [0,1]$, $i = \overline{1,n}$ для i-*го* наблюдения.

Эта функция рекомендована документацией pytorch к использованию в случае задач бинарной классификации [15]. Она реализуется как экземпляр класса «torch.nn.BCELoss». При создании не требуется никаких аргументов.

Оптимизаторы в pytoch наследуются от базового класса «torch.optim.Optimizer». Любому наследнику следует передавать первым аргументом параметры нейронной сети, обычно они идут в формате «torch.Tensor» и должны быть извлечены из экземпляра нейронной сети, то есть в нашем случае это должно выглядеть так:

```
model = ResultNet(layers_list)
optimizer = Some_optimizer_class(model.parameters()),
```

где layers_list – предварительно объявленный список слоев;

Some_optimizer_class – выбранный наследник.

Работает это следующим образом: model.parameters() возвращает параметры нейронной сети, а optimizer «запоминает» их и, по вызову его метода «step()», изменяет их в зависимости от ситуации. Более подробно этот механизм будет раскрыт далее, когда мы непосредственно коснемся вопроса обучения нейронной сети.

Конкретные алгоритмы оптимизации реализованы в наследниках. Нигде не удалось найти рекомендаций или указаний, какой вид алгоритма лучше всего использовать в нашем случае, мы всегда используем «torch.optim.Adam», потому как он нередко приводиться в примерах к задачам классификации. Важным параметром для оптимизатора является learning rate – это число на которое умножается градиент целевой функции, таким образом можно регулировать величину шага нелинейной оптимизации. Во всех готовых оптимизаторах и, в частности, в оптимизаторе Adam он указывается через именованный аргумент «lr» конструктора.

3.1.2 Реализация алгоритма обучения

В самом базовом случае для обучения модели нужны следующие шаги:

- Обнулить градиент оптимизатора с прошлой итерации используя метод «torch.optim.Optimizer.zero_grad»;
- Произвести прямое распространение активации используя описанный выше метод «forward» экземпляра модели. Нет необходимости вызвать метод напрямую, для «torch.nn.Module» произведено переопределение оператора «()», так что он вызывает «forward» автоматически для переданных аргументов. Соответственно, все наследники, если в них, конечно, оператор не переопределен повторно будут действовать также;

- Предсказанные результаты с реальными откликами передаются в экземпляр целевой функции, для которого также переопределен «()». Будет получена ошибка – объект класса «torch.tensor»;
- У полученной ошибки вызывают метод «baskward», который представляет собой ничто иное как реализацию обратного распространения ошибки, алгоритма о котором мы говорили в подразделе 1.4. В результате оптимизатор получает информацию о градиенте;
- В последнюю очередь вызывают метод «step» оптимизатора, тут то и происходит каждое смещение весов.

Описанные шаги повторяются много раз, так добиваются оптимизации весов. Данный подход самый базовый и, в большинстве случаев, его не достаточно. Оказалось недостаточно и в нашем, потому для обучения модели был создан класс «model_traner», представленный на листинге К.2. В названном классе использованы различные техники, позволяющие проводить обучение более эффективно.

Конструктор класса обязательно ожидает модель, оптимизатор и целевую функцию, кроме того имеется именованный аргумент «lr_scheduler» сущность которого будет раскрыта ниже, когда речь коснется соответствующей техники.

Обыденной практикой машинного обучения считается разбиение выборки на тренировочную и тестовую. Дело в том, что нейронная сеть – очень мощный метод аппроксимации, нередко получается так, что модель не улавливает закономерность, а просто запоминает выборку на которой училась, это ведет к тому, что она необоснованно ошибается на данных которых не видела, такую ситуацию в машинном обучении называют переобучением модели [19 с. 127]. Разбиение на тренировочную и тестовую выборки решает эту проблему – выгодно отслеживать целевую функцию и проводить валидацию модели на выборке которая не участвовала в обучении, несмотря на то, что для оценки коэффициентов будет задействован не весь массив данных. Мы рассказываем об этом именно тут потому как дальнейшее описание класса обучающего модель неразрывно связано с этой практикой.

Особого внимания заслуживает метод «fit» – центральный этого класса. В названном методе, по сути, и работают шаги обучения модели. Этот метод обязательно принимает аргументы «train_loader» и «test_loader». В терминах «tytorch» это загрузчики данных, соответственно получаем загрузчик обучающих и тестовых данных.

Загрузчики позволяют «элегантно» реализовать еще одну важную для данной работы технику – обучения модели по эпохам. При обучении модели по эпохам набор данных, выделенный для обучения, разбивают на равные части – так называемые батчи. Затем батчи перебирают и для каждого из них вычисляется градиент целевой функции и производится соответствующее смещение весов. Эпохой, применительно к алгоритмам обучения нейронных сетей, называют одну итерацию прохода по всем таким частям [20 с. 202]. Делают это для того, чтобы при вычислении градиента упрощать целевую функцию. А упрощение целевой функции, в свою очередь, приводит к тому, что

в задаче оптимизации становиться меньше локальных минимумов и алгоритм вероятнее сходится к глобальному минимуму или хотя-бы выбирает лучший локальный минимум.

Касательно батчей, мы провели отдельное небольшое исследование этого механизма. Его можно найти в ноутбуке «proc/batch_size_research/barch_size_research.ipynb» основного репозитория данной работы. Тут представим краткие результаты – мы создали случайный двумерный и набор данных с двумя классами, он представлен точками на рисунке Л.1. Специально пытались создать набор данных так, чтобы была группа точек «ломающая» основную статистическую закономерность (она в левом верхнем углу). Таким образом мы создали два здравых решения: левый рисунок представляет первое, правый – второе. Решения, для просторы, линейные и представляют собой двухфакторные логистические регрессии. Тепловой картой обозначены предсказания соответствующих моделей. А в заголовках к отдельным графикам представлены значение целевой функции бинарной кросс энтропии для данных предсказаний. Интересно, что мы рассчитывали, что закономерность заложенная в правый график будет побочной, но получилось так, что она стала основной (у нее ошибка ниже). Но так даже лучше, пример еще показателен в том плане, что нередко наиболее оптимальное решение оказывается неожиданным и, даже, нелогичным.

Как известно, двухфакторная логистическая регрессия, в классическом виде, требует три параметра: два при переменных и свободный член. Это не очень то удобно для визуализации целевой функции по параметрам, потому мы два параметра при переменных объединили в один, со смыслом поворота дискриминирующей прямой. Формула выражения под логитом приняла вид:

$$\cos(\alpha)\, x_1 + \sin(\alpha)\, x_2 + \beta,$$

где α – поворот дискриминирующей кривой в радианной мере;

β – смещение;

x_i – i-я переменная $i = \{1,2\}$.

Такое изменение немного отрывает нас от реального положения дел (модели так никогда не параметризуют), но, как увидите далее, для целей нашего примера такой поход хорош. Это нужно было для того чтобы нанести рисунок 3.1.

На рисунке 3.1 линии уровня целевой функции в зависимости от введенных параметров. Слева рисунок на полных данных – отчетливо выделяются два локальных минимума, но глобальный только правый. И если мы начнем оптимизацию с неудачных параметров, то мы рискуем прийти в левый. И тут это решается просто визуализацией, но, напомним, что в нашей задаче будет не менее 113 параметров. Не возникает сомнений, что подобная визуализация невозможна. Потому и пользуются всевозможными уловками для того, чтобы обходить локальные минимумы.