Поиск недостающих вызовов библиотечных функций с использованием машинного обучения

И.А.. Якимов <ivan.yakimov.research@yandex.ru>
А.С. Кузнецов <ASKuznetsov@sfu-kras.ru>
Институт космических и информационных технологий,
Сибирский федеральный университет,
660074, Россия, г. Красноярск, ул. Академика Киренского, д. 26

Аннотация. Разработка программного обеспечения является сложным и подверженным ошибкам процессом. В целях снижения сложности разработки ПО создаются сторонние библиотеки. Примеры исходных кодов для популярных библиотек доступны в литературе и интернет-ресурсах. В данной работе представлена гипотеза о том, что большинство подобных примеров содержат повторяющиеся шаблоны. Более того, данные шаблоны могут быть использованы для построения моделей, способных предсказать наличие (либо отсутствие) недостающих вызовов определенных библиотечных функций с использование машинного обучения. В целях проверки данной гипотезы была реализована система, реализующая описанный функционал. Экспериментальные исследования, проведенные на примерах для библиотеки OpenGL, говорят в поддержку выдвинутой гипотезы. Точность результатов достигает 80%, при условии рассмотрения уже первых 4-х ответов, предлагаемых системой. Можно сделать вывод о том, что данная система при дальнейшем развитии может найти индустриальное применение.

Ключевые слова: ОpenGL; качество программного обеспечения; рекомендательные системы; машинное обучение; нейронные сети;

DOI: 10.15514/ISPRAS-2017-29(6)-6

Для цитирования: Якимов И.А., Кузнецов А.С. Поиск недостающих вызовов библиотечных функций с использованием машинного обучения. Труды ИСП РАН, том 29, вып. 6, 2017 г., стр. 117-134. DOI: 10.15514/ISPRAS-2017-29(6)-6

1. Введение

Разработка программного обеспечения (ПО) является не только дорогостоящим, но и сложным процессом. Для упрощения разработки ПО создаются библиотеки функций, скрывающие сложность за программным интерфейсом (*application programming interface* — API). При использовании сторонних библиотек программисты зачастую применяют некоторые готовые

решения — *шаблоны*, полученные из ресурсов интернета и литературы. При этом начинающие программисты имеют тенденцию пропускать различные части данных шаблонов при реализации собственных приложений.

Ключевая идея. В предложенной работе исследуется проблема восстановления пропущенных частей шаблонов, используемых программистами при применении сторонних библиотек. В основу данного исследования положена гипотеза о том, что большая часть корректно работающих приложений, использующих сторонние библиотеки, основана на шаблонах, свойства которых поддаются анализу средствами машинного обучения. Данная гипотеза основана на результатах работ, ранее проведенных в смежных областях, таких как автоматическая генерация патчей [1] и автодополнение кода [2].

1.1 Обзор литературы

Статический анализ и динамические символьные вычисления. Во время динамических символьных вычислений [3,4,5] систематически исследуются различные пути выполнения программы. При этом, либо производится многократный перезапуск целевой программы, либо происходит параллельное выполнения различных ее путей посредством интерпретатора с копированием состояний. Статический анализ [6] в свою очередь производится без запуска программы. Результатом статического анализа является абстрактное представление, содержащее информацию о множестве различных путей выполнения программы.

Информация, полученная средствами динамического и статического анализа, используется для поиска программных ошибок. Данные инструменты нацелены на поиск низкоуровневых ошибок, таких как переполнение буфера, разыменовывание нулевого указателя и т.д. При условии моделирования работы внешних функций также может проводиться проверка корректности алгоритма использования библиотек (например, память, выделенная malloc, должна быть высвобождена с помощью free), предупреждение использования небезопасных функций (например, strlen вместо strnlen). Статический и динамический анализ может также применяться для проверки моделей и решения задач формальной верификации.

Таким образом, средства статического и динамического анализа предоставляют мощный инструментарий для проверки корректности программ. Однако в данной работе акцент смещен в сторону обнаружения недостающих вызовов библиотечных функций. Дополнительным требованием является то, чтобы разработчик не задавал правила обнаружения недостающих вызовов вручную. Система должна самостоятельно извлечь необходимые знания из подготовленных для нее примеров.

Автоматическая генерация патчей. Задача автоматической генерации патчей близка к задаче, решаемой в данном исследовании. Генерация патчей предполагает автоматическое исправление ошибок в программах. Семантика верных программ, используемая при оценке качества патчей строится

средствами машинного обучения [1]. Однако для проверки патчей все же предполагается наличие тестов, позволяющих определить их корректность. В данной работе для оценки верности предлагаемых системой рекомендаций не предполагается наличие тестов, поэтому системы автоматической генерации патчей не будут детальнее рассматриваться.

Рекомендательные системы. Наиболее близкими к предлагаемой в данной работе системе являются рекомендательные системы, построенные с использованием машинного обучения [7], в частности системы для улучшения механизма автодополнения кода.

В случае контекстно-зависимых вероятностных моделей автодополнения происходит анализ всех случаев применения объекта и затем моделируется распределение вероятности для следующего вызова [8, 9]. Развитием данной концепции является применение n-граммной языковой моделей исходного кода [10] для предоставления контекста автодополнения. Возможно также использование n-граммной модели с поддержкой кэша использованных токенов [11], а также дополнение языковой модели исходного кода статистической семантикой [12]. Помимо языковых моделей завершения кода на основе токенов также применяются вероятностные модели на основе абстрактных синтаксических деревьев [13,14].

Альтернативным подходом является автодополнение в указанных пользователем местах, в которых предположительно расположены недостающие фрагменты кода. В данном случае производится не только подбор нужного токена, но синтез целой строки кода. Пользователь помечает (пустые) строки, в которых предположительно должен располагаться фрагмент кода. В указанных строках система синтезируются наиболее вероятный недостающий фрагмент [15]. При этом используется языковая модель исходного кода, построенная на основе рекуррентной нейронной сети.

Таким образом, в существующих на данный момент рекомендательных системах пользователь должен в явном, либо неявном виде указывать конкретные участки кода, в которых предполагается применение автодополнения. Однако в данной работе фокус смещен на способность системы самостоятельно определять факт наличия (либо отсутствия) пропущенных вызовов библиотечных функций без активного участия пользователя.

Обучение с учителем. В первом приближении, машинное обучение с учителем предполагает построение моделей, тренеруемых устанавливать соответствие между некоторым входом и выходом на основе тренеровочной выборки примеров [16]. Одной из разновидностей моделей, применяемых в задачах машинного обучения является искусственная нейронная сеть (artificial neural network — ANN). В данной работе применяется рекуррентная нейронная сеть, построенная на ядре LSTM (long-short term memory) [17]. Архитектура LSTM была выбрана из-за своей широкой применимости в задачах статистического моделирования языка. Рекуррентные нейронные сети способны

аппроксимировать работу алгоритмов [18], что важно при построении системы, выявляющей закономерности в исходном коде.

1.2 Постановка задачи

Пусть дана некоторая пользовательская процедура, содержащая вызовы функций сторонней библиотеки. С помощью заранее заданного алгоритма из данной процедуры извлекается последовательность вызовов функций w = f1,f2,...,fn, где fi — имя функции. Предполагается, что в данной последовательности либо пропущена одна функция, необходимая для завершения определенного шаблона, либо пропущенные функции отсутствуют. Необходимо построить модель, обладающую следующими свойствами. Если в последовательности недостает вызова функции, завершающего некоторый шаблон, система должна восстановить имя пропущенной функции. В том же случае, когда пропуски отсутствуют, и последовательность представляет собой завершенный шаблон использования сторонней библиотеки, система должна установить данный факт.

2. Методы

2.1 Обзор системы

В качестве целевой была выбрана библиотека OpenGL [19]. Данный выбор обусловлен тем, что для OpenGL существует большое количество примеров с открытым исходным кодом, которые являются сравнительно однотипными, что делает их хорошим материалом для решения задач машинного обучения.

В законченном виде система предоставляет достаточно простой для использования интерфейс. Пользователь подает ей на вход исходный код программы, получая на выходе отчет. Отчет содержит в себе записи о каждой из пользовательских процедур. Каждая запись содержит несколько предположений системы о том, какая функция была пропущена. Особым видом предположения является отсутствие пропущенных функций.

Пример. Рассмотрим несколько примеров, полученных в ходе экспериментальных исследований работы системы. В первом примере приведен код процедуры по отображению сцены на экране. В данной процедуре был пропущен вызов функции glPushMatrix. Пропущенный вызов помещен в комментарий в целях улучшения наглядности данного примера. Без предварительных знаний о свойствах библиотечных функций и закономерностях их вызовов, система выучила шаблон, согласно которому за вызовом glPushMatrix должен следовать вызов glPopMatrix. Система предоставляет пользователю распределение вероятностей, содержащее имена 10 наиболее вероятных недостающих функций. Следует отметить, что в задачу системы не входит определение местоположения пропущенной функции. Среди предложенных ответов присутствует и верный — по оценке системы, с

вероятностью 98,84% была пропущена функция glPushMatrix. Специальный токен __none__ в данном распределении символизирует отсутствие пропусков. В примере с пропуском функций, реализующих логику push-pop, можно предложить простую реализацию алгоритма обнаружения пропущенных вызовов. Достаточно использовать счетчик push- и pop-вызовов и затем сравнить результат. Однако во многих случаях система должна быть способна распознавать сложные алгоритмы использования библиотечных функций, и обойтись алгоритмом со счетчиком вызовов крайне затруднительно. Во втором примере приведена процедура Reshape, в которой отсутствуют пропуски вызовов функций, то есть подаваемый на вход системе алгоритм реализован верно. Данный факт подтверждается в отчете, предлагаемом системой, согласно которому с вероятностью 55.27% пропуски отсутствуют.

Табл. 1. Пример работы системы
Table 1. Example

Исходный код	Вывод системы	
void display(void) { glClear (GL_COLOR_BUFFER_BIT); glColor3f (1.0, 1.0, 1.0); **// glPushMatrix(); - недостающая функция** glutWireSphere(1.0, 20, 16); glRotatef ((GLfloat) year, 0.0, 1.0, 0.0); glTranslatef (2.0, 0.0, 0.0); glRotatef ((GLfloat) day, 0.0, 1.0, 0.0); glutWireSphere(0.2, 10, 8); glPopMatrix(); glutSwapBuffers(); }	**glPushMatrix** glRotatef __none__ glFlush gluPerspective glFrustum glScalef glEnable gluOrtho2D glNewList	**98,84%** 0,22% 0,10% 0,09% 0,09% 0,08% 0,07% 0,07% 0,06% 0,06%
// пропущенных вызовов нет static void Reshape(int width, int height){ glViewport(0, 0, width, height); glMatrixMode(GL_PROJECTION); glLoadIdentity(); glFrustum(-2.0, 2.0, -2.0, 2.0, 0.8, 10.0); gluLookAt(7.0, 4.5, 4.0, 4.5, 4.5, 2.5, 6.0, -3.0, 2.0); glMatrixMode(GL_MODELVIEW); }	**__none__** glLoadIdentity glOrtho gluOrtho2D glEnable glTranslatef gluPerspective glScalef glMatrixMode glDisable	**55,27%** 18,47% 6,72% 4,67% 3,04% 1,64% 1,25% 1,22% 0,94% 0,76%

2.2 Архитектура

Рассмотрим внутреннее устройство обученной системы, сам процесс обучения будет показан в следующем разделе. На вход системы подается файл с исходным кодом на языке Си. Данный файл передается трассировщику. На выходе трассировщика получается n трасс вызовов библиотечных функций «трасса 1», «трасса 2», …, «трасса n», где n — число пользовательских процедур, содержащих вызовы библиотечных функций. Каждая трасса представляет собой последовательность имен библиотечных функций, расположенных в порядке их вызова внутри одной пользовательской процедуры. Полученные трассы далее поступают на вход препроцессору, который преобразует их в матрицу, содержащую признаки каждой отдельной трассы. Далее полученная матрица поступает на вход классификатору. Структурная схема изображена на рис. 1.

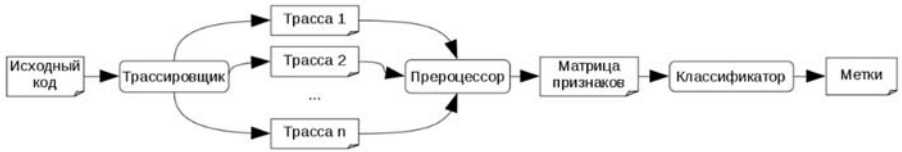

Рис. 1. Архитектура системы
Fig. 1. Architecture of the system

Остановимся подробнее на задаче, решаемой классификатором. В конечном итоге, классификатор должен определять пропущенные в трассе функции, либо же отсутствие пропуска. Иначе говоря, он должен поставить в соответствие каждой трассе метку, которая либо хранит имя пропущенной функции, либо информацию о том, что она отсутствует. Следует отметить, что в случае обучения классификатора препроцессор не только преобразует трассы к требуемому формату, но и генерирует на их основе примеры.

2.3 Анализ исходных данных

Для тренировки классификатора средствами машинного обучения необходим соответствующий набор данных. Для получения набора данных были использован примеры исходных кодов программ из следующих источников [20], далее данный набор будет именоваться *бенчмарком*. Примеры содержат исходный код учебных OpenGL программ. Далее понятия OpenGL-функция и функция будут применяться как взаимозаменяемые там, где это не противоречит контексту. В общей сложности было использовано 214 файлов с исходным кодом программ. Отметим, что данные примеры изначально не предназначены для решения задач машинного обучения и не являются частью готового набора данных.

В данной работе применяется обучение с учителем, и набор данных разбивается на тренировочную и тестовую выборку. Таким образом, обучение классификатора производится на тренировочной выборке, каждый пример из которой содержит корректный ответ в виде метки. Проверка, в свою очередь, производится на тестовой выборке. Таким образом, набор данных должен быть размечен, и состоять из матрицы признаков и вектора меток.

Объектами, подлежащими классификации, являются пользовательские процедуры. Единственным признаком, характеризующим каждую процедуру, является полученная из нее трасса. Соответственно, метка, которая классификатором ставится в соответствие каждой процедуре, содержит либо имя пропущенной функции, либо «__none__» — запись о том, что пропущенные функции отсутствуют. Таким образом, признаки являются списками имен функций, а метки — либо именами функций, либо __none__.

В данной работе используется следующий принцип получения примеров: пусть дана трасса w = f1,f2,...,fn, содержащая последовательность из n вызовов функций. Для данной трассы может быть получен n+1 пример. Первые n примеров содержат трассы, в которых последовательно пропущены вызовы функций. Последний пример содержит исходную версию трассы, в которой пропуски отсутствуют. Набор данных формируется из большого количества примеров, получаемых из трасс. Из принципа построения набора данных следует, что для каждой функции число порождаемых примеров равно числу вызовов данной функции внутри бенчмарка. В общей сложности, в бенчмарке производятся вызовы 329 различных функций, некоторые из них более популярны, некоторые — менее. При этом, если функция glEnable была вызвана ~700 раз, то менее популярные функции, такие как glBitmap вызываются менее 30 раз. Таким образом, число вызовов распределено неравномерно между различными функциями. Однако в целях улучшения обучаемости классификатора желательно, чтобы число примеров для каждой из функций было примерно одинаковым. Таким образом, необходимо произвести анализ распределения частоты вызовов для отдельных функций. Данный анализ был проведен, и его результаты показали, что распределение вызовов обладает свойствами, схожими с принципом Парето. В частности, первые 20% вызванных функций обеспечивают 82% всех вызовов функций.

В табл. 2. приведено распределение частоты вызовов функций. В первом столбце указана взятая рассматриваемая доля наиболее популярных функций. Во втором столбце указана общая доля вызовов данных функций среди всех вызовов функций в бенчмарке.

Табл. 2. Распределение частоты вызовов функций
Table. 2. Function calls frequency distribution

Процент самых популярных функций	Процент вхождений в примеры
5%	45%
10%	68%
20%	83%
30%	90%

40%	94%
50%	96%

Можно судить о наличии проблемы неравномерности распределения количества примеров использования различных функций. В целях ее решения наименее популярные функции были сгруппированы в наборы. Для этого было задано отображение, переводящее имя каждой отдельной функции в имя соответствующего ей предопределенного набора. Разбиение функций в наборы было произведено эвристическим способом. При этом мы опирались на назначение функции (например, работа с текстурами) и распределение вызовов функций. Результаты классификации не учитывались при разбиении. Далее набор функций будет называться «метафункцией», в целях удобства.

Используя описанное выше отображение, исходный набор данных может быть поэлементно отображен в новый набор. В новом наборе все функции, входящие состав трасс и меток, отображаются на соответствующие им метафункции. Классификатор, обученный на наборе данных, составленном из метафункций, также для удобства назовем «метаклассификатором». В свою очередь классификатор, который обучен на наборе, построенном из необработанных функциях, назовем «наивным классификатором».

Каждая метафункция имеет определенную мощность — число функций, которые в нее отображаются. Мощность может быть равна нулю, единице, либо некоторому целому n > 1. *Пустой* назовем метафункцию, в которую не отображается ни одна из функций; данная метафункция используется как метка __none__. *Именные* метафункции соответствуют единственной функции. В *обобщенную* метафункцию отображается несколько функций. Еще одной важной характеристикой является вес метафункции – число примеров, в которые она входит в качестве метки.

Разберем введенное для удобства понятие метафункции на примере. Пустая метафункция __none__ не содержит ни одной функции, ее мощность равна нулю, а вес равен числу трасс, в которых не пропущена ни одна функция. Выделенная метафункция *glEnable* содержит единственную функцию — собственно glEnable, ее мощность равна единице, и вес равен числу вызовов glEnable в исходном коде программы. Обобщенная метафункция *draw* включает множество функций по отрисовке, выбранных эвристическим путем, ее мощность больше 1 т. к. она включает в себя несколько функций, а вес равен суммарному количеству вызовов входящих в нее функций.

В общей сложности, эвристическим путем было выделено 50 непустых метафункций. Среди данных метафункций 35 являются *именными*, и 15 *обобщенными*, соответственно.

2.4 Создание набора данных

Набор данных для обучения классификатора получается из бенчмарка путем ряда преобразований. Первым этапом является получение трасс вызовов функций внутри пользовательских процедур. При этом условные операторы и циклы не учитываются. Для получения трасс используется интерпретатор IR-кода [21]. Интерпретатор работает со списком пользовательских функций, объявленных внутри модуля, генерируемого clang [22] из файла с исходным кодом. Для каждой функции производится обход входящих в нее базовых блоков. Порядок следования базовых блоков определяется clang-ом на этапе трансляции в IR-код. Алгоритм получения трасс представлен на листинге 1.

Алгоритм: построение трасс вызовов OpenGL-функций
Вход: исходный код
Выход: набор трасс пользовательских процедур
для каждой пользовательской процедуры *proc*:
  trace = новая пустая трасса
  для каждого базового блока *block* процедуры function:
    для каждого вызова внешней функции *func* внутри блока *block*:
    если *func* — OpenGL-функция, добавить ее имя в трассу

Листинг 1. Построения трасс вызовов функций
Algorithm 1. Function calls tracing

Суммарная длинна полученных трасс составляет 15182 вызовов, что теоретически позволяет получить 15 тысяч примеров для каждой функции.

Отсечение трасс. Трассы, полученные описанным выше методом, отличаются по длине, то есть имеют различное количество вызовов. Разброс значения длины лежит в диапазоне от 1 до 338 вызовов. Обучение классификатора на основе рекуррентной нейронной сети на последовательностях различной длины вызывает затруднения. По данной причине из полученных трасс были отобраны трассы длиной от 5 до 25 вызовов, что составляет ~50% всех трасс. В результате, количество примеров сократилось до 7531.

Отображение функций в метафункции. Для обучения классификатора используется два набора данных. В первом наборе использованы оригинальные имена функции. Таким образом, поскольку число меток в наборе данных равно числу различных функций плюс дополнительная пустая функция __none, то мы имеем 329 + 1 метку. Во втором случае имена функций отображены на соответствующие им метафункции. В результате получается 50 + 1 метка, соответственно.

Группировка трасс. Данный шаг предшествует шагу получения примеров из каждой трассы. Трассы группируются так, чтобы получить n наборов трасс приблизительно равного размера, где n = 10. Далее n-1 набор составит тренировочную выборку и 1 набор — тестовую выборку, соответственно. Таким

образом, тестовая выборка составляет ~10% от объема полного набора данных. Обязательным условием получения тестовой выборки является то, чтобы в нее входили случайные примеры (примеры будут получены из трасс на последнем шаге). Однако полностью случайный выбор примеров невозможен. Примеры, полученные из одного файла с исходным кодом, не должны попадать в разные наборы — это нарушило бы достоверность проверки работы классификатора на тестовой выборке. По этой причине, при формировании наборов трасс случайным образом компонуются только трассы, полученные из одного файла с исходным кодом. Случайность тестовой выборки обеспечивается за счет случайного выбора наборов трасс.

Получение примеров и заполнение. Следующей фазой является получение примеров из каждой трассы с помощью описанного ниже алгоритма. На вход ему подается трасса из n символов, на выходе массив из n+1 примера. В основном цикле (строки 5-9) из трассы длиной производится n примеров. Далее добавляются исходная трасса и соответствующий ей пропуск (строки 10-11).

1. **Алгоритм**: получение примеров из трассы
2. **Вход**: трасса *trace* длинной n вызовов
3. **Выход**: n+1 пример
4. examples = []
5. для каждого *k* в диапазоне от 0 до длина(*trace*)-1:
6. *current_trace* = клонировать(*trace*)
7. удалить *current_trace*[*k*]
8. y = *trace*[*k*]
9. добавить (current_trace, *y*) к *examples*
10. xs = *trace*
11. y = '_' // пропуск означает отсутствие пропущенных функций
12. добавить (*xs, y*) к *examples*

Листинг 2. Получение примеров из трассы
Algorithm 2. Example extraction

После того как набор данных получен, все имена функций заменяются целочисленными идентификаторами — id, где id > 0. Следует отметить, что вследствие различий в длине трасс, входящих в состав примеров, их необходимо выравнивать по длине. Это достигается за счет добавления слева нужного числа нулей.

2.5 Архитектура классификатора

Готовый набор данных используется для обучения и тестирования классификатора. Нами был использован классификатор, построенный на базе фреймворка keras [23]. Данный классификатор состоит из нескольких слоев. Структурная схема классификатора изображена на рис. 2. При обсуждении

архитектуры классификатора понятия функции и метафункции считаются взаимозаменяемыми.

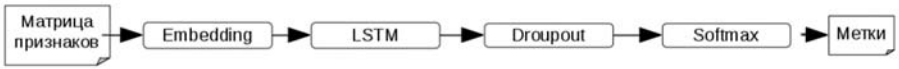

Рис. 2. Архитектура классификатора
Fig. 2. Classifier

Рассмотрим каждый из слоев более подробно. **Embedding**, или встраивание — преобразует поступающие в него трассы в векторное представление. Векторизация позволяет улучшить обучаемость нейронной сети при работе с последовательностями. С помощью векторизации каждый id преобразуется в вектор длиной в 32 элемента. **LSTM** — слой, реализующий рекурсивную нейронную сеть. Данный слой построен на базе модуля долгой краткосрочной памяти (Long short-term memory, LSTM). Отметим, в данной работе в качестве функции активации использована тангенциальная функция. **Dropout —** слой прореживания, необходимый для регуляризации нейросети с целью снижения эффектов переобучения. **Softmax** — функция активации, вычисляющая мягкий максимум. Данная функция преобразует выходы от предыдущего слоя в распределение вероятностей, описывающего вероятность принадлежности трассы к каждому из классов. Далее полученное распределение используется для определения класса, к которому принадлежит трасса.

3. Результаты

3.1 Описание эксперимента

Обучение классификатора было произведено на выборке, состоящей из 6623 примеров. Размер тестовой выборки, в свою очередь, составил 833 примеров. Число циклов (эпох) обучения нейронной сети равняется 100 циклам. При анализе выхода классификаторов была использована оценка точности top-k. На рис. 3 показан график, отображающий результаты проведенного эксперимента. По оси абсцисс отложено число k — количество взятых за рассмотрение ответов. Для каждого классификатора по оси ординат отложена линия, обозначающая график некоторой зависящей от k функции, соответствующей доле верных ответов при рассмотрении первых k ответов.

Линия S(k) на данном графике соответствует выходу классификатора, обученного на наборе данных из необработанных функций. Линии M(k) и M*(k) соответствуют выходу метаклассификатора, то есть классификатору, обученному на наборе данных из метафункций. При этом линия M*(k) соответствует доле корректно классифицированных *именных* метафункций.

3.2 Обсуждение

Анализ результатов позволяет выявить определенные закономерности. Так при k = 1 (то есть когда рассматривается только первое предположение о пропущенной функции), оба классификатора приблизительно в половине случаев дают верные ответы. Однако при значениях k > 1 проявляется выгода от использования метаклассификатора, то есть S(k) < M(k) для k = 2..10. В случае метаклассификатора доля верных ответов достигает 80% уже при k = 4; в свою очередь, в случае же простого классификатора — при k = 7. Таким образом, пользователь может с большей уверенностью полагаться на ответы системы в режиме метаклассификатора, так как чаще получает верную информацию о наличии (либо отсутствии) пропусков.

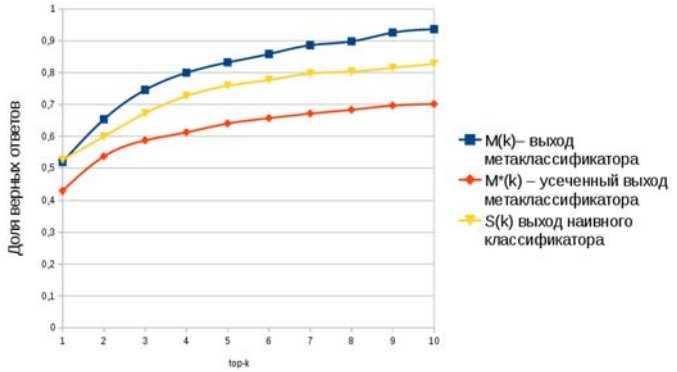

Рис. 3. Результаты
Fig. 3. Results

Однако, график, соответствующий доле верно классифицированных именных метафункций, лежит ниже графика, полученного для выхода наивного классификатора, то есть для всех k=1..10 M*(k)<S(k). Это вызвано тем, что метаклассификатор в качестве ответа приводит не только именные, но и обобщенные метафункции. При этом зачастую имя пропущенной функции не может быть точно определено, как в случае обобщенных метафункций. Таким образом, метаклассификатор чаще выдает верные ответы, однако наивный классификатор чаще выдает имена конкретных функций (строго говоря, он всегда выдает только имена конкретных функций). Следовательно, отображение редко используемых функций на наборы (метафункции) увеличивает число верных ответов, но при этом снижает среди них долю ответов с определенным именем функции. Это является закономерным, так как в данном случае многие функций объединяются в метафункции. Например, метаклассификатор может корректно определить факт отсутствия функции по работе с текстурами, но без

уточнения того, какой именно функции из набора не хватает — пользователю будет предоставлен список возможных вариантов.

3.3 Достоверность

Выбранная в качестве целевой, библиотека OpenGL является удобным объектом для данного исследования. Во-первых, вызовы входящих в нее функций образуют шаблоны, и, во-вторых, для данной библиотеки доступен широкий набор примеров в открытых источниках. Однако не все библиотеки обладают тем свойством, что их вызовы образуют шаблоны; примером может служить часть стандартной библиотеки языка Си по работе со строками — cstring. Также не для всех библиотек доступен столь же широкий набор примеров, как для OpenGL. Однако, на наш взгляд, предлагаемый метод может быть распространен на достаточно большой набор библиотек, для чего требуются дальнейшие экспериментальные и теоретические исследования.

Необходимо отметить ряд факторов, которые могли оказать влияние на достоверность результатов. Во-первых, в набор данных были включены трассы длинной от 5-25 функций. Однако на наш взгляд в большинстве случаев этого достаточно. Так, например, если функция содержит более 25 вызовов, то пользователь может произвести ее декомпозицию, сократив число вызовов; функция с 4 и менее вызовами маловероятно будет содержать какой-либо шаблон. Так или иначе, пользователю доступна информация об ограничении количества вызовов. Также следует отметить, что проверка классификатора была произведена на единственной тестовой выборке. Более надежным методом является перекрестная проверка, однако данный метод является слишком ресурсозатратным. Так как наборы трасс для тестовой выборки были отобраны случайным образом, можно с уверенностью утверждать, что результаты достаточно надежны и воспроизводимы. Далее, на данной стадии исследований в систему не заложен функционал по определению нескольких пропусков вызовов библиотечных функций. Однако данный недостаток может быть устранен при инкрементальном сценарии использовании системы, параллельно с разработкой программы. Например, на законченном фрагменте программного кода пользователь может его проверять на наличие пропуска и лишь затем приступать к новому фрагменту.

4. Заключение

В данной работе была разработана система, позволяющая определять наличие (либо отсутствие) пропусков вызовов библиотечных функций. Данная системы была апробирована на наборе данных, полученном из бенчмарка, включающего 239 файлов с исходным кодом примеров для OpenGL. Полученная система демонстрирует достаточно высокую точность работы. Так, в случае группировки редко используемых библиотечных функций в наборы, доля верных ответов равная 80% достигается уже при рассмотрении первых четырех ответов (top-4) системы. В базовом случае, аналогичная точность, равная 80%,

достигается для top-7 ответов. Несмотря на определенные ограничения, уже на данном этапе полученная система может быть использована в качестве ассистента при разработке программ с использованием библиотеки OpenGL, например, при обучении студентов. В настоящий момент проводится работа по увеличению набора данных, используемого при обучении классификатора, улучшению архитектуры положенной в его основу нейронной сети, а также добавлению возможности по определению наличия большего числа пропусков вызовов библиотечных функций.

Список литературы

[1]. Long F., Rinard M. Automatic patch generation by learning correct code. Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2016, pp 298-312. DOI: 10.1145/2837614.2837617

[2]. Bruch M., Bodden E., Monperrus M., Mezini M. 2010. IDE 2.0: collective intelligence in software development. Proceedings of the FSE/SDP workshop on Future of software engineering research, 2010, pp. 53-58. DOI: 10.1145/1882362.1882374

[3]. Cadar C., Godefroid P., Khurshid S., Păsăreanu C.S., Sen K., Tillmann N., Visser W. Symbolic execution for software testing in practice: preliminary assessment. Proceedings of the 33rd International Conference on Software Engineering, 2011, pp 1066-1071. DOI: https://doi.org/10.1145/1985793.1985995

[4]. Anand S., Burke E.K., Chen T.Y., Clark J., Cohen M.B., Grieskamp W., Harman M., Harrold M.J., Mcminn P. 2013. An orchestrated survey of methodologies for automated software test case generation. J. Syst. Softw. 86, 8 (August 2013), pp 1978-2001. DOI: 10.1016/j.jss.2013.02.061

[5]. Вертанов С.П., Герасимов А.Ю. Динамический анализ программ с целью поиска ошибок и уязвимостей при помощи целенаправленной генерации входных данных. Труды ИСП РАН, том 26, вып. 1, 2014 г., стр 375-394. DOI: 10.15514/ISPRAS-2014-26(1)-15

[6]. Герасимов А.Ю. Обзор подходов к улучшению качества результатов статического анализа программ. Труды ИСП РАН, том 29, вып. 3, 2017 г., стр. 75-98. DOI: 10.15514/ISPRAS-2017-29(3)-6

[7]. Allamanis M., Barr E.T., Devanbu P., Sutton C. A Survey of Machine Learning for Big Code and Naturalness. Размещено на сайте arxiv.org 18 сентября 2017 г. Режим доступа: https://arxiv.org/abs/1709.06182

[8]. Bruch M., Monperrus M., Mezini M. Learning from examples to improve code completion systems. Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on foundations of software engineering, 2009, pp. 213-222. DOI: 10.1145/1595696.1595728

[9]. Proksch S., Lerch J., Mezini M. 2015. Intelligent Code Completion with Bayesian Networks. ACM Trans. Softw. Eng. Methodol. 25, 1, Article 3 (December 2015), 31 pages. DOI: 10.1145/2744200

[10]. Hindle A., Barr E.T., Su Z., Gabel M., Devanbu P. On the naturalness of software. Proceedings of the 34th International Conference on Software Engineering, 2012, pp. 837-847.

[11]. Franks C., Tu Z., Devanbu P., Hellendoorn V. CACHECA: a cache language model based code suggestion tool. Proceedings of the 37th International Conference on Software Engineering - Volume 2, 2015, pp. 705-708.

[12]. Nguyen T.T., Nguyen A.T., Nguyen H.A., Nguyen T.N. A statistical semantic language model for source code. Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, 2013, pp. 532-542. DOI: 10.1145/2491411.2491458
[13]. Bielik P., Raychev V., Vechev M. PHOG: probabilistic model for code. Proceedings of the 33rd International Conference on International Conference on Machine Learning,Vol. 48, 2016, pp. 2933-2942.
[14]. Maddison C.J., Tarlow D. Structured generative models of natural source code. Proceedings of the 31st International Conference on International Conference on Machine Learning,Vol. 32, 2014, II-649-II-657.
[15]. Raychev V., Vechev M., Yahav E. Code completion with statistical language models. Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2014, pp. 419-428. DOI: 10.1145/2594291.2594321
[16]. Smola Alex., Vishwanathan S.V.N. Introduction to Machine Learning. Cambridge University Press (2008).
[17]. Hochreiter S., Schmidhuber J. Long Short-Term Memory. Neural Comput. 9, 8 (November 1997), 1997, 1735-1780. DOI: 10.1162/neco.1997.9.8.1735
[18]. Siegelmann HT. Computation beyond the turing limit. Science. 1995 Apr 28; 268 (5210):545-8. DOI: 10.1126/science.268.5210.545
[19]. Библиотека OpenGL. Режим доступа: https://www.opengl.org/
[20]. Примеры использования C API OpenGL. Режим доступа: https://www.khronos.org/opengl/wiki/Code_Resources
[21]. LLVM — компиляторная инфраструктура. Режим доступа: https://llvm.org/
[22]. Clang — фронт-енд для семейства Си-подобных языков. Режим доступа: https://clang.llvm.org/
[23]. Keras - фреймворк для создания нейронных сетей. Режим доступа: https://keras.io/

Searching for missing library function calls using machine learning

I.A. Yakimov <ivan.yakimov.research@yandex.ru>
A.S. Kuznetsov <ASKuznetsov@sfu-kras.ru>

Institute of space and informatics technologies, Siberian Federal University, Akademika Kirenskogo 26 st., Krasnoyarsk, 660074, Russia

Abstract. Software development is a complex and error-prone process. In order to reduce the complexity of software development, third-party libraries are being created. Examples of source codes for popular libraries are available in the literature and online resources. In this paper, we present a hypothesis that most of these examples contain repetitive patterns. Moreover, these patterns can be used to construct models capable of predicting the presence (or absence) of missing calls of certain library functions using machine learning. To confirm this hypothesis, a system was implemented that implements the described functional. Experimental studies confirm the hypothesis. The accuracy of the results reaches 80% with a top-4 accuracy. It can be concluded that this system, with further development, can find industrial application.

Keywords: OpenGL; software quality; recommender systems; machine learning; neural networks;

DOI: 10.15514/ISPRAS-2017-29(6)-6

For citation: Yakimov I.A., Kuznetsov A.S. Searching for missing library function calls using machine learning. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 6, 2017. pp. 117-134 (in Russian). DOI: 10.15514/ISPRAS-2017-29(6)-6

References

[1]. Long F., Rinard M. Automatic patch generation by learning correct code. Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2016, pp 298-312. DOI: 10.1145/2837614.2837617
[2]. Bruch M., Bodden E., Monperrus M., Mezini M. 2010. IDE 2.0: collective intelligence in software development. Proceedings of the FSE/SDP workshop on Future of software engineering research, 2010, pp. 53-58. DOI: 10.1145/1882362.1882374
[3]. Cadar C., Godefroid P., Khurshid S., Păsăreanu C.S., Sen K., Tillmann N., Visser W. Symbolic execution for software testing in practice: preliminary assessment. Proceedings of the 33rd International Conference on Software Engineering, 2011, pp 1066-1071. DOI: https://doi.org/10.1145/1985793.1985995
[4]. Anand S., Burke E.K., Chen T.Y., Clark J., Cohen M.B., Grieskamp W., Harman M., Harrold M.J., Mcminn P. 2013. An orchestrated survey of methodologies for automated software test case generation. J. Syst. Softw. 86, 8 (August 2013), pp 1978-2001. DOI: 10.1016/j.jss.2013.02.061
[5]. Vartanov S. P., Gerasimov A. Y. Dynamic program analysis for error detection using goal-seeking input data generation. Trudy ISP RAN / Proc. of ISP RAS, vol. 26, issue 1, 2014, pp. 375-394 (in Russian). DOI: 10.15514/ISPRAS-2014-26(1)-15
[6]. Gerasimov A.Y. Survey on static program analysis results refinement approaches. Trudy ISP RAN / Proc. of ISP RAS, vol. 29, issue 3, 2017, pp. 75-98. DOI: 10.15514/ISPRAS-2017-29(3)-6 (in Russian)
[7]. Allamanis M., Barr E.T., Devanbu P., Sutton C. A Survey of Machine Learning for Big Code and Naturalness. Размещено на сайте arxiv.org 18 сентября 2017 г. Режим доступа: https://arxiv.org/abs/1709.06182
[8]. Bruch M., Monperrus M., Mezini M. Learning from examples to improve code completion systems. Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on foundations of software engineering, 2009, pp. 213-222. DOI: 10.1145/1595696.1595728
[9]. Proksch S., Lerch J., Mezini M. 2015. Intelligent Code Completion with Bayesian Networks. ACM Trans. Softw. Eng. Methodol. 25, 1, Article 3 (December 2015), 31 pages. DOI: 10.1145/2744200
[10]. Hindle A., Barr E.T., Su Z., Gabel M., Devanbu P. On the naturalness of software. Proceedings of the 34th International Conference on Software Engineering, 2012, pp. 837-847.
[11]. Franks C., Tu Z., Devanbu P., Hellendoorn V. CACHECA: a cache language model based code suggestion tool. Proceedings of the 37th International Conference on Software Engineering - Volume 2, 2015, pp. 705-708.
[12]. Nguyen T.T., Nguyen A.T., Nguyen H.A., Nguyen T.N. A statistical semantic language model for source code. Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, 2013, pp. 532-542. DOI: 10.1145/2491411.2491458

[13]. Bielik P., Raychev V., Vechev M. PHOG: probabilistic model for code. Proceedings of the 33rd International Conference on International Conference on Machine Learning,Vol. 48, 2016, pp. 2933-2942.
[14]. Maddison C.J., Tarlow D. Structured generative models of natural source code. Proceedings of the 31st International Conference on International Conference on Machine Learning,Vol. 32, 2014, II-649-II-657.
[15]. Raychev V., Vechev M., Yahav E. Code completion with statistical language models. Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2014, pp. 419-428. DOI: 10.1145/2594291.2594321
[16]. Smola Alex., Vishwanathan S.V.N. Introduction to Machine Learning. Cambridge University Press (2008).
[17]. Hochreiter S., Schmidhuber J. Long Short-Term Memory. Neural Comput. 9, 8 (November 1997), 1997, 1735-1780. DOI: 10.1162/neco.1997.9.8.1735
[18]. Siegelmann HT. Computation beyond the turing limit. Science. 1995 Apr 28; 268 (5210):545-8. DOI: 10.1126/science.268.5210.545
[19]. OpenGL: https://www.opengl.org/
[20]. OpenGL code resource: https://www.khronos.org/opengl/wiki/Code_Resources
[21]. LLVM — compiler Infrastructure: https://llvm.org/
[22]. Clang — a C language family frontend for LLVM: https://clang.llvm.org/
[23]. Keras – the Python Deep Learning library: https://keras.io/