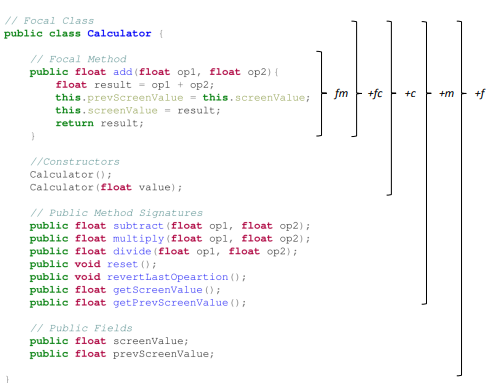
Дипломна работа

# Анализ на съществуващи решения

## AthenaTest [https://arxiv.org/pdf/2009.05617.pdf%20%E2%80%93%20unit%20Test%20Case%20Generation%20with%20Transformers%20and%20Focal%20Context]

Генерация на модулно тестване използвайки трансформатори и фокусен контекст е разработка на Мишел Туфано (Michele Tufano), Даун Дрейн (Dawn Drain), Алексей Святковски (Alexey Svyatkovski), Шао Кун Денг (Shao Kun Deng) и Нийл Съндерсън (Neel Sundersan).

Тази разработка се различава от други с това, че не е базирана на метрика за покритие на тестовете, а на контекста, който трябва да бъде валидиран от тест. При тази разработка подходът е от тип “sequence to sequence”, който се състои от процес с две стъпки при тренирането на модела. Първата стъпка е тренирането на модел върху голям обем от Java код, който после да бъде настроен фино (fine-tuning) за задачата на генериране на код. За целта е и създаден най-големият набор от данни, който е ѝ именуван, за модулни тестове с техните контексти – тестван метод, дефиниция на класа на тествания метод, конструктор на тествания клас, методи на тествания клас и полета. Данните са кръстени Methods2Test и могат да бъдат намерени и използвани свободно в GitHub.



Целият процес за генериране на тестове е както следва:

1. Събират се данни – съществуващи тестове и контексти,
2. Тестовете се асоциират с техните контексти
3. Използва се общият модел
4. Създава се фината настройка, след което вече се генерират тестовете

Процеса съществува на фигурата отдолу:

Graphical user interface, diagram

Description automatically generated

## GitHub Copilot [https://copilot.github.com]

GitHub Copilot е приложение, чиято цел е да бъде пълноценен помощник, който да се държи като втори програмист, който програмира с потребителя. Самото приложение е имплементирано като притурка (plugin) на средата за разработка Visual Studio Code. GitHub Copilot работи по механизъм подобен на този за обикновеното допълване при въвеждане на код, само че вместо да допълва една дума може да допълни цяла функция или метод на база на дадено описателно име и/или коментар, който описва действието на съответния фрагмент от код.

Реално GitHub Copilot представлява генератор на код, който генерира код от дадено описание. Това е постигнато с помощта на изкуствен интелект и по-конкретно с модел разработен от OpenAI в колаборация с Microsoft. Моделът се казва Codex и е базиран на огромния модел с 12 милиарда параметъра трансформатор (transformer) GPT-3. Моделът е трениран използвайки хиляди публични хранилища за код включително и всички такива от GitHub. Моделът, съответно и GitHub Copilot, се справя най-добре с код на Python, тъй като най-много такива данни са присъствали при тренирането на модела.

За да може да генерира предложения GitHub Copilot използва отворения файл в Visual Studio Code, както и другите файлове, които са част от отвореното работно пространство. С този контекст моделът може да направи предложение, което е конкретно за контекста на проекта.

На практика продуктът работи най-добре при писане на малки функции с описателни имена и точно описание на какво трябва да извърши функцията.



## EvoSuite [https://www.evosuite.org/evosuite/]

EvoSuite е система, която може автоматично да генерира модулни тестове за класове написани на Java. Това се постига, чрез използване на хибриден подход, който генерира и оптимизира цели тестови пакети, които се стремят към постигане на високо покритие върху тествания клас. Самите тестове са малки по размер и ефективни използвайки достатъчно клаузи от тип “assert”, които да са достатъчни за описанието и поддръжката на съществуващата логика. Системата има за цел да предоставя тестове, на които разработчиците да разчитат при по нататъшни пренаписвания на различни фрагменти. Тези тестове също се очаква да гарантират сегашната работа на софтуера.

Системата има следните функционалности:

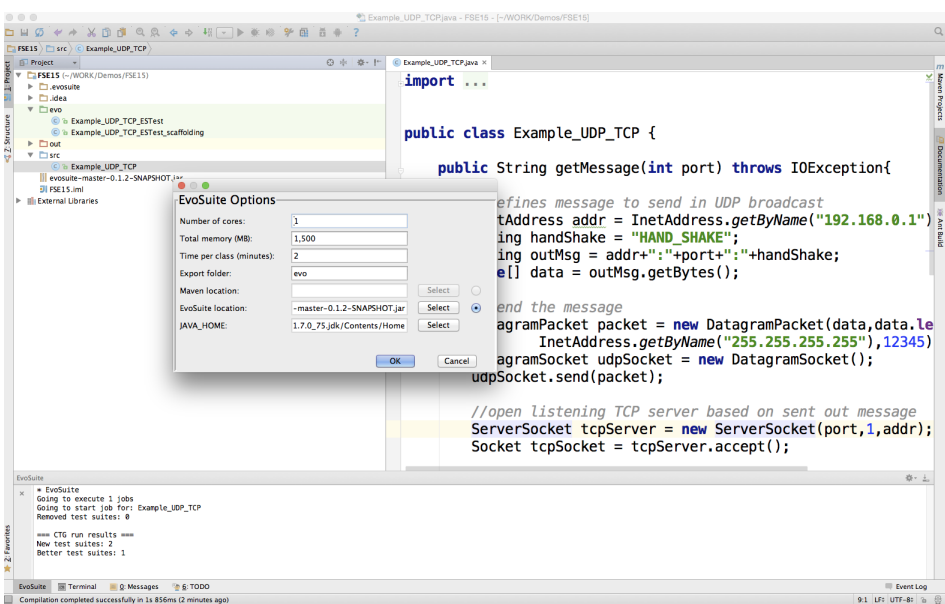
* Генериране на Junit4 тестове за избрани класове
* Оптимизация на тестове за конкретна критерия на покритие – линии, разклонения, изходи и тестове за мутации
* Генерираните тестове се минимизират – размерът им се свежда до минимум запазвайки същото ниво на покритие
* Генериране на Junit проверяващи клаузи, които да проверяват моментното състояние на класа
* Пускане на тестовете във виртуална среда за по-голяма сигурност
* Виртуална файлова система
* Виртуални компютърна мрежа

Системата може е достъпна по няколко начина:

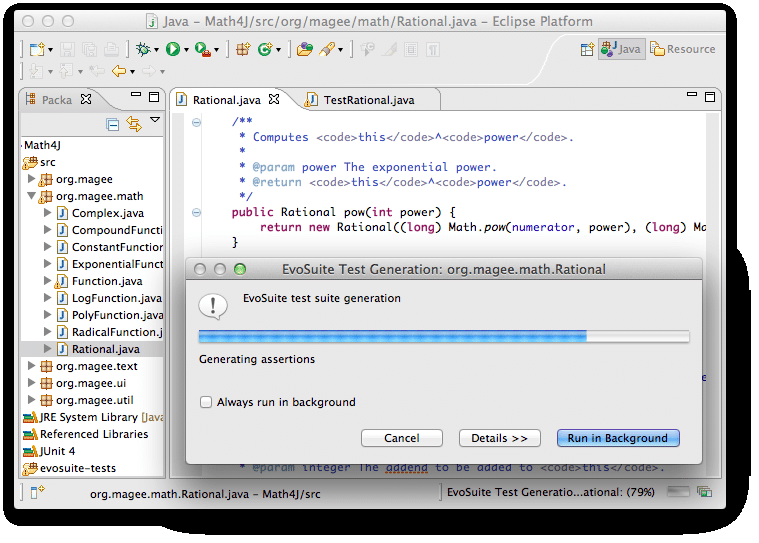
Първият, от които е като инструмент с конзолен интерфейс, за да се използва по този начин се сваля .jar файл, чрез който се използва системата.

Вторият вариант е като притурка (plugin) към средата за разработка. Към момента такива притурки съществуват за InteliJ IDEA и Eclipse.

След като се инсталира нужната притурка за InteliJ IDEA, тестове се генерират като се избере клас, група от класове или пакет и чрез десен клик на мишката се избере генериране на EvoSuite Test. Потребителският интерфейс изглежда по следния начин:



При Eclipse приложението работи по аналогичен начин:



Освен тези два плъгина съществува такъв и за системата за непрекъсната интеграция – Jenkins. С нея може да се добави стъпка, която да генерира и изпълнява тестове за всички новодобавени класове.

## Randoop[https://randoop.github.io/randoop/]

Randoop е система за генериране на модулни тестове за Java. Системата генерира тестове с Junit. Част от функционалността на Randoop съществува и за .Net.

Алгоритъмът, който използва Randoop за генериране на тестове е чрез итеративно генерация на произволни тестове, като се използва обратна връзка от генерирания тест, за да се подобри и стигне до финалната му форма. На практика се генерират псевдо-произволни извиквания към методите и конструкторите на тествания клас, след което генерираният тест се пуска и спрямо резултатите от извиканите методи се създават проверки.

Тази система се използва главно за намиране на бъгове в съществуващите системи. Често пъти генерираните тестове не са лесно четими за програмистите и за това се използват основно за намиране на регресии и бъгове.

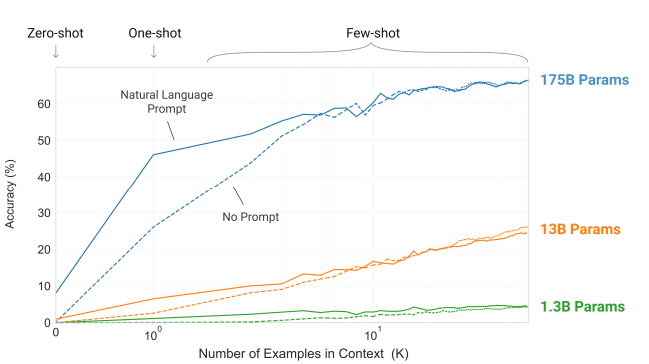
## Проучване

### Language Models are Few-Shot Learners[https://arxiv.org/pdf/2005.14165.pdf]

В този научен труд се изследва как мащабирането на модел от тип трансформатор взаимодейства с неговата продуктивност. Като се изследват няколко модела с основна разлика в броя параметри. Крайният продукт е наречен GPT-3 и използва 175 милиарда параметъра, като е обучаван върху 400 милиарда „думи“ (token-а). Разглежда се и концепцията за даване на 0, 1 или няколко примера на модела – изводите са, че моделът може се справя изключително добре за много задачи, без да има нужда от допълнително трениране (fine-tuning).

Практически статията не показва нова архитектура на модела, тя е базирана на стария модел разработен от OpenAI – GPT-2, основната разлика е в големината – GPT-3 е високо мащабирана версия на своя предшественик.

На графиката отдолу могат да бъдат видени как с увеличаване на броя параметри се подобрява се увеличава производителността на модела за контекстни задачи. От това, че с увеличаването на дадените примери се увеличава и производителността следва и че създаденият модел разбира от дадената му допълнително контекстната информация.

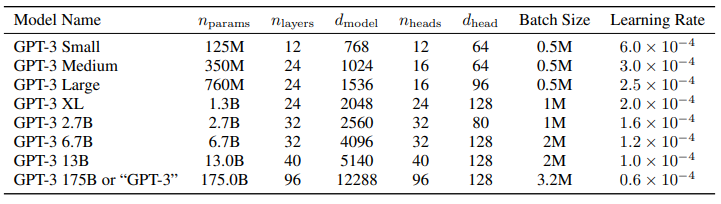


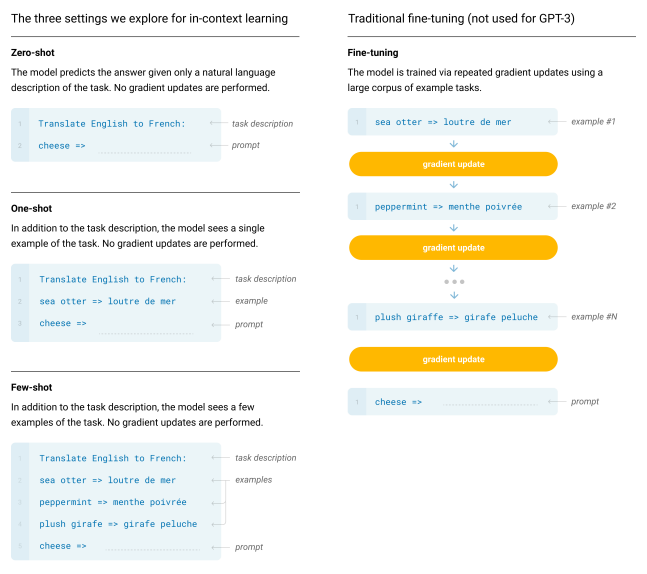
В статията е отделено специално внимание на ученето чрез 0 или няколко примера.

1. Допълнително-обучение (Fine-tuning):  
   Това е един от най-популярните похвати през последните години. Идеята зад него е да се използва голям съществуващ модел, чиито тежести да бъдат актуализирани чрез „допълнително трениране“ на модел с стотици хиляди надписани примери. Основният плюс на този подход е, че дава много добри резултати за конкретни тестове, основният минус е, че за всяка задача са нужни големи количества качествени данни, които са надписани. Което значи и че моделът може да изгуби своите „общи“ знания, което е сходно с това моделът да е твърде специфичен към данните, с които е обучаван (over fit).
2. Няколко-изстрела (Few-shot) – няколко примера:  
   Този термин се използва при настройка на модела при която се дават няколко демонстрационни примера заедно със задачата, която моделът трябва да изпълни. Важно е да се отбележи, че при това извиване не се променят тежестите на мрежата.
3. Един-изстрел (One-shot) – един пример:  
   Този термин, носи същото значение като няколкото-изстрела, единствената разлика е, че този път се дава само един пример за задачата.
4. Нула-изстрела (Zero-Shot) – нула примера:  
   При този вид извикване на модела, не се дават никакви примери и се изследва как моделът се справя без да му се дават никакви примери.

Всички тези извиквания на модела могат да бъдат видени на долната фигура.

Резултати и сравнение на това как моделът се подобрява с мащабирането си могат да бъдат видени в таблица 1.





### Evaluating Large Language Models Trained on Code [https://arxiv.org/pdf/2107.03374.pdf]

Evaluating Large Language Models Trained on Code е статия описваща процеса по създаване и оценка на най-големия модел за генериране на код от OpenAI – Codex. Codex е базиран на най-големия езиков модел на компанията GPT-3. Тъй като GPT-3 показва не лоши резултати в генерацията на код на Python, докато моделът не е трениран за това компанията смята, че при допълнително трениране на модела е напълно възможно да се постигнат много добри резултати.

За целта се прави допълнително трениране върху GPT-12B. Използваните данни за тази операция са от основно от хранилища за код в GitHub, данните съдържат в себе си 179 GB уникални python файлове, като всеки файл е не повече от 1 MB. След което е направена допълнителна филтрация, която цели да премахне всички генерирани файлове резултата е 159 GB данни.

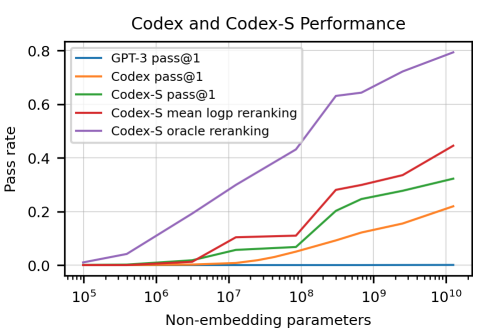
След тренирането на Codex разбира се идва и най-важната част – оценката на модела. Това е трудна и сложна задача, защото генерирането на решение не може да бъде оценено както генерацията на текст. Един от най-популярните подходи за оценка на генериран текст и код е оценката BLEU.

BLEU е абревиатура за bilingual evaluation understudy – двуезична оценка. Тази оценка се появява като решение за оценяването на качеството на генериран текст при машинни преводи. Тоест оценката дава количествена репрезентация на качеството на превод направен от модел в контекста на моделите базирани на машинно обучение. Идеята на тази оценка е да даде колко близък е преводът направен от машината до този направен от човек, който е професионален преводач. От тук следва и това че оценката винаги е между 0 и 1. [https://dl.acm.org/doi/10.3115/1073083.1073135]

Начин на работа на BLEU се базира на два входа: низ кандидат и референтен низове. Резултатът би бил 1 ако кандидат низът е сходен до референтните низове и 0 ако сходство отсъства. Тъй като в естествения език думите имат различно значение според контекста си BLEU поддържа и така наречените n-gram-и, които представляват дума заедно със съседните ѝ няколко (общо n на брой). [https://dl.acm.org/doi/10.3115/1073083.1073135]

BLEU работи изключително добре в контекста на преводите, но за жалост при генерирането на код, чиято идея е да решава даден проблем не е най-добрият начин за оценка. Програмистите оценяват даден метод, клас, модул спрямо това дали той работи правилно или не, а не по това колко сходно е написаното решение до такова, което вече съществува. Тази оценка за това дали даден код работи правилно или не типично се автоматизира чрез писането на автоматични тестове. Този вид оценяване гарантира качество на кода и затова дори е създаден начин на работа базиран на него – Разработка чрез тестове (Test Driven Development).

От тук следва, че е много по-логично оценката на модела да се случва по същия начин – дали за генерирания код минават написани предварително тестове. За целта и OpenAI създават 164 проблема заедно с модулни тестове за всеки. Тези проблеми са така направени, че да могат да тестват различни специфики на езика, алгоритми и проста математика, които са нужни за решенията. Тези данни могат да са свободно достъпни и могат да бъдат използвани от всекиго. Върху тези проблеми Codex се справя изключително добре както се вижда от графика 1.



За жалост резултатите не са толкова добри при тестове на реални проблеми. За това се налага и нов подход при оценяването на модела.

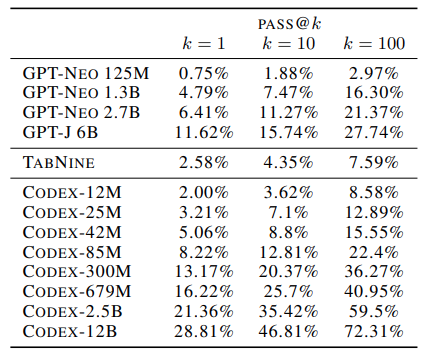
Както вече бе описано метрики базирани на точни съвпадения са неподходящи за оценка на генериран код, защото не обхващат комплексната природа на решенията на проблем.

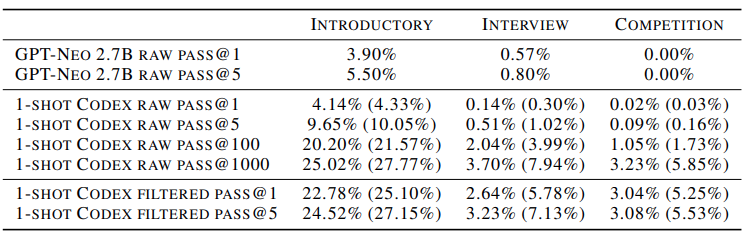
За това и изключително по-добър вариант в случая е оценка, чрез тестове. Тъй като моделът може да генерира различни решения то се вкарва в употреба и така нареченото оценяване eval@k. [https://www.researchgate.net/publication/361359280\_FixEval\_Execution-based\_Evaluation\_of\_Program\_Fixes\_for\_Competitive\_Programming\_Problems]. За този подход се генерират k примера за всеки проблем, като проблем се смята за решен ако за един от генерираните примери минат всички тестове.

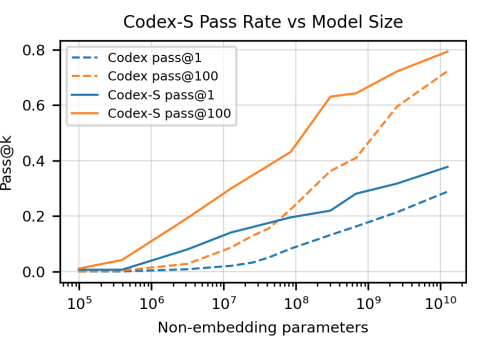
Тъй като изискването за това един пример да реши всички тестове не е съвсем точна оценка за това дали Codex генерира работещи решения е избрана вариация на този подход, при която се генерират n > k примера за всяка задача и след това се взима бройката на тези решения, за които минават всички тестове.

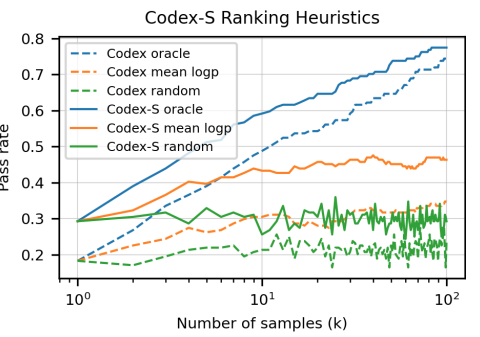
На практика формулата за оценка е следната:

Този подход е много по-подходящ и показателен за това как се справя моделът с генерирането на работещи решения по зададено описание.

сравнение на GPT-NEO, GPT-J (модели базирани на GPT-3 https://github.com/eleutherai/gpt-neo), TabNine и Codex модели върху HumanEval данни.

Сравнение на GPT-NEO и Codex върху данните от статията APPS [https://paperswithcode.com/paper/measuring-coding-challenge-competence-with] [https://paperswithcode.com/dataset/apps]





Оценка спрямо големината на модела.

Моделът, който предоставят OpenAI е с изключително добри възможности в генерирането на код. Освен това статията дава и добри насоки за това как трябва да се оценява генериран код, този вид оценки се използват

От описаните графики се вижда, че моделът на Codex е с изключително добри възможности в генерирането на код.

### Abstract Syntax Trees [https://en.wikipedia.org/wiki/Abstract\_syntax\_tree, https://hillside.net/plop/plop2003/Papers/Jones-ImplementingASTs.pdf]

В компютърните науки абстрактните синтактични дървета, още познати и като синтактични дървета са структура от данни, която репрезентира абстрактната синтактична структура на даден текст, който използва формален език - най-често сорс код. Всеки възел в дървото е конструкция, която участва в анализирания файл – if, while, дефиниция на променлива, return и т.н.

[ https://en.wikipedia.org/wiki/File:Abstract\_syntax\_tree\_for\_Euclidean\_algorithm.svg]

Абстрактното синтактично дърво се различава от конкретното синтактично дърво по това, че абстрактното дърво не съдържа в себе си подробности от типа на какви кавички се използват за дефиницията на низове и други. Конкретните синтактични дървета често се използват за контекстен анализ на кода. Често пъти контекстните синтактични дървета се построяват като се обогати модела на абстрактното синтактично дърво.

Абстрактното синтактично дърво от своя страна се използва от много компилатори, които създават тази структура от данни като резултат от една от фазите на компилатора. Често пъти се използва за междинна стъпка на репрезентацията на програмата, която се използва от другите фази на компилатора.

Абстрактното синтактично дърво може да бъде променяно и използвано за оптимизации на кода без да се променя самият сорс код. По този начин компилаторът може да оптимизира програмата използвайки дървото за анализ и в последствие променяйки го, но в същото време запазвайки непроменен сорс кода на програмата. Друго много важно предимство на дървото е, че то не съдържа в себе си ненужни пунктуационни детайли – точки и запетайки за край на реда, скоби, кавички и други. Важно е да се отбележи и че във възлите на дървото се съдържа информация свързана с това къде се намира дадения възел, с помощта на тази информация компилаторите могат да дават конкретна информация за това къде и какъв проблем е възникнал по време на компилация на кода.

За всяко абстрактно синтактично дърво има следните изисквания:

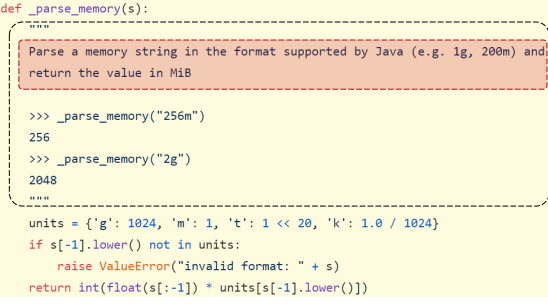
1. Типовете на променливите трябва да се запазват заедно с тяхното местоположение.
2. Редът на изпълнимият код трябва да бъде указан експлицитно, ясно и еднозначно
3. Левите и десните компоненти на булеви операции трябва да се запазят в правилен ред.
4. Идентификаторите и стойностите, които са им присвоени трябва да бъдат запазени

В контекста на дипломната работа структурата от данни се използва за анализ и оценка на различни сорс код файлове.

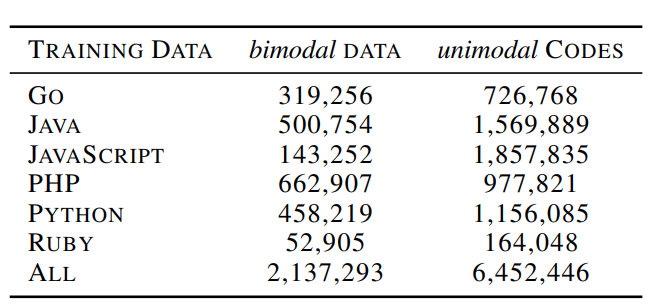
### A Pre-Trained Model for Programming and Natural Languages [https://arxiv.org/pdf/2002.08155.pdf]

Статията на тема предварително трениран модел за програмни и естествени езици се представя моделът CodeBERT, който е предварително трениран модел върху бимодални данни за програмни езици и естествени такива.

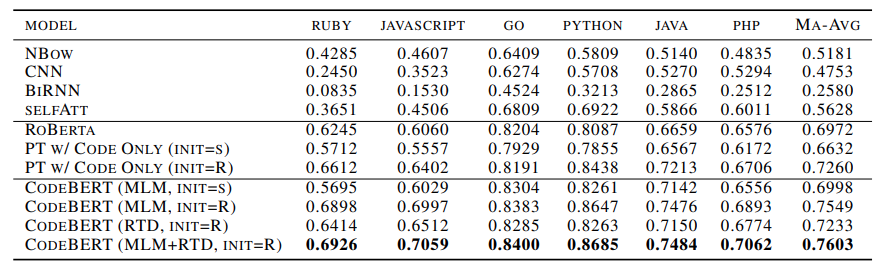
Идеята на модела е даде генерализирана репрезентация за връзката между програмните езици и естествените в контекста на техническа документация, търсене на функционалност в кода и т.н.

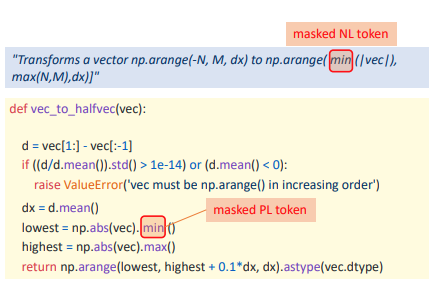
CodeBERT е с архитектура базирана на трансформаторите при невроните мрежи. Моделът е трениран върху бимодални и унимодални данни. Бимодални данни означава, че съществуват програмен код и свързаната с него техническа документация (фигура). Докато унимодлни данни са тези, които имат или само код или съответно само текст написан на естествен език (английски).

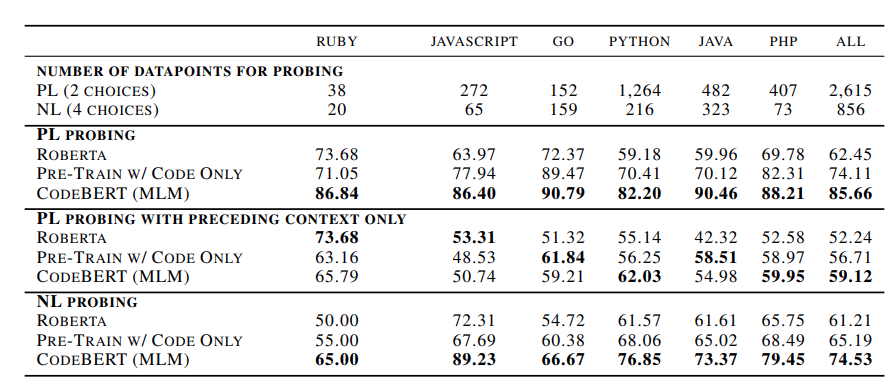
Моделът е обучен върху милиони примери, техният обем може да бъде видян в таблица .



Моделът в предварително-обучената си форма се справя най-добре със задачи от тип локализация на код по дадено описание Таблица и генерация на документация от код. Освен това моделът се изследва и за това колко добре може да предположи какъв низ трябва да се сложи при маскиран такъв в код или документация. За изследването са маскирани думите min и max в различни места от кода и документацията му (фигура). Резултати и сравнение от различнимодели могат да бъдат намерени в таблица.







Всички показни данни от експериментите са направени след допълнително-обучение на модела, от това следва и нуждата от добри данни, с които моделът да бъде допълнително-обучен за конкретният проблем.

### Модели

От горе изнесената информация се вижда, че големите проучвания на тематиката с генерация на текст имат своите разлики – основно в обем и данни, с които са тренирани, също така и за задачите към които се стремят. Техните прилики, разлики и специфики са описани в долната таблица

|  |  |  |  |
| --- | --- | --- | --- |
| Модел (име): | Архитектура | Мащаб | Цел |
| CodeBERT | Базирана на трансформатори | Параметри – 125 милиона  Данни – 8 милиона примера | Генерация на документация и локализация на код по описание.  Предвиден за допълнително-обучение за конкретни задачи свързани с генерация на код от текст и съответно генерация на текст от код. |
| GPT-3 | Базирана на трансформатори | Параметри – 175 милиарда  Данни – 300 милиарда думи (tokens) | Модел целящ се да има общо разбиране за естествения език. Най-големият модел за обработка на естествен език към момента, който се справя изключително добре на много и различни задачи свързани с естествения език особено ако бъде използван подхода за задаване на вход с няколко-изстрела.  Не е предвиден за разбиране и допълване на код без нуждата от допълнително-трениране, за което ще са нужни няколко стотин хиляди примера |
| Codex | Базирана на трансформатори | Параметри -12 Милиарда  Данни – 159 GB сорс код | Допълнително-трениран модел базиран на GPT-3, чиято цел е генерирането на код от описание на естествен език. Основната цел е да генерира функции, които извършват описаното в документиращ низ (doc string). |

# Функционално описание на разработката

## Цел

Целта на настоящата дипломна работа е разработката на система, която да генерира тестове базирани на критериите описани в потребителската история, която е разработвана от програмиста. Генерираните тестове са функционални, интеграционни и в някои случаи на цялата система – end to end. Идеята да съществува такъв инструмент е да се подобри продуктивността на програмистите в ежедневната им работа като се намали времето, което те инвестират в писане на автоматични тестове, а освен това и да се подобри качеството на разработвания продукт.

Идеята за автоматична генерация на код не е нова, както е описано в първа глава на дипломната работа съществуват разработки с различни подходи на тази тематика. От използване на изкуствен интелект до системи, които генерират псевдо произволни извиквания на съществуващи методи.

Важно е да се отбележи, че разработките, които съществуват на тематиката за генерация на тестове се фокусират основно и само върху генерация на модулни тестове. При тези проекти най-важно е разбирането за модула (класа) и няма нужда от знание за цялата системата докато в системата, която се разработва за тази дипломна работа е важно да се тества цяла функционалност, а не конкретен модул.

Решението, което е разработено в тази дипломна работа е добра демонстрация на концепцията – използване на дълбоко обучение за генериране на тестове. Към момента то може да се използва като начална версия на тестове, които да бъдат доразработени от програмиста или съответно инженерите по качеството.

За програмната реализация на проблема с генериране на код за функционални тестове са използвани редица технологии. Основните технологии използвани за продукта са Python и модели разработени от OpenAI, както и популярният стек за работа с големи обеми от данни за Python, а именно Numpy, Pandas и популярната библиотека за работа с естествени езици NLTK. Освен тези библиотеки са използвани и синтактични дървета както и библиотеката PyDriller за по-лесна обработка на хранилища за код от тип git.

## Програмен език

Езикът за програмиране, който е избран за реализацията на дипломната работа е python. За разработката на този софтуер това е един от най-добрите избори, защото е най-популярният сред обществото, което се занимава с изкуствен интелект и анализ на данни. Нормалното следствие от популярността на езика в тези среди е и изключително добрата му поддръжка на тематиката изразяваща се в множество различни библиотеки, които се разработват за анализ и обработка на данни както и такива свързани с машинно обучение.

## Библиотеки

### NLTK

Библиотеката, която е избрана за обработка на естествен език е NLTK. NLTK е една от водещите платформи за разработка на програми на Python, които работят с данни от естествен език. Библиотеката предоставя лесни за използване интерфейси включвайки в себе си множество ресурси от данни. Библиотеката съдържа в себе си множество под-библиотеки за класификация, токенизация, коренуване, парсиране, определяне на частите на речта, определяне на „емоцията “ и други. Основните причини за избора на тази библиотека е това, че тя е създадена за Python и в себе си съдържа всичко, което би потрябвало в разработката на дипломната работа и далеч повече освен това е една от най-използваните, с което идват и множеството ресурси свързани с нея.

### Pandas [https://www.nobledesktop.com/learn/python/pandas-overview, https://pandas.pydata.org/about/]



Pandas е библиотека с отворен код, чието начало е поставено през 2008 година като през 2009 кодът на библиотеката става публично достъпен. Името на библиотеката идва от думите на английски за панел и данни - “**Pan**el” и “**Da**ta”. Библиотеката предоставя функционалности за обработка на големи масиви от данни. Основни нейни функционалности са:

* Бързи и ресурс ефективни обекти от тип рамки за данни (DataFrame), които са предвидени за манипулация и индексация
* Инструменти за сериализация и десериализация – запазване на данните в често използвани формати като CSV, Microsoft Excel, SQL и HDF5
* Автоматична подредба на данните като се използват етикети, а не само цифрови индекси. Библиотеката следи за липсващи данни.
* Размера на матрицата с данни може да бъде сменян лесно и бързо. Т.е. операции с модификация на колони и редове са позволени и работят бързо.
* Мощен механизъм за групиране (филтриране) на данните по критерии
* Висока производителност при сливане на големи обеми от данни
* Висока производителност на библиотеката – имплементирана е с технологии от ниско ниво, които да подсигурят тази производителност, а именно C и Cython.
* Една от най-използваните библиотеки в академичните среди. Широко използвана в много индустрии – Финанси, Икономика, Невронаука, Статистика, Реклама, Анализ на големи данни.

Тази библиотека бе избрана за разработката на дипломната работа тъй като реализацията на генериране на автоматични тестове изисква работа с голям обем от данни – съществуващ и генериран код.

### PyDriller [https://pydriller.readthedocs.io/en/latest/]

PyDriller е библиотека, която дава абстракция и интерфейс върху командите и работата с Git хранилище. Идеята на библиотеката е да предостави възможност на различните разработчици да анализират и обработват данни от различни хранилища на код по лесен начин.

Библиотеката се състои от 3 основни обекта – хранилище (Repository), ревизия/версия (Commit) и модифициран файл (Modified File).

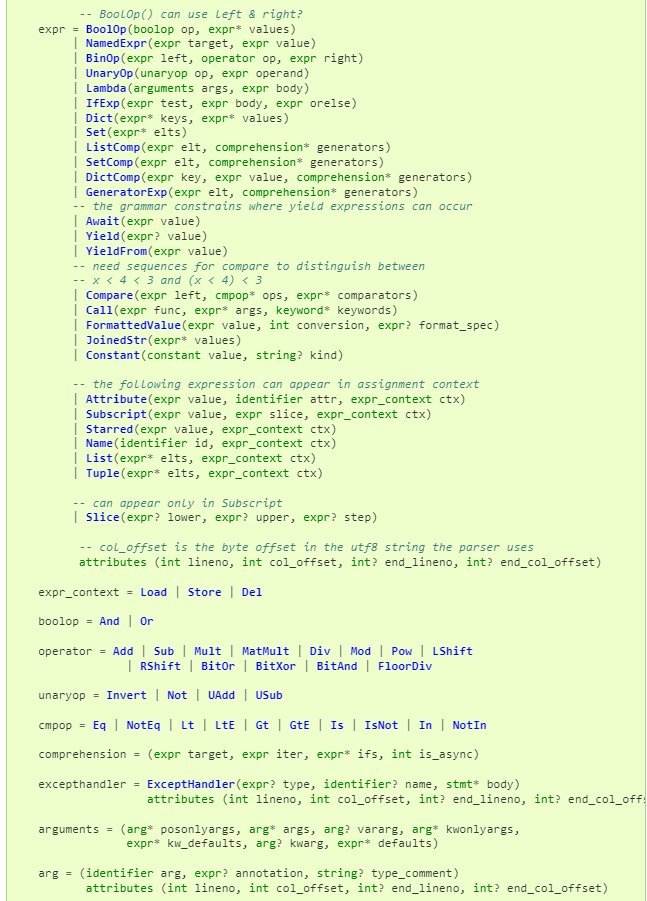
За целта на дипломната работа се използват различните ревизии и съответно от всяка се разграничават по-съществените файлове, тези които са медиатори и дефинират конкретна функционалност от тези, които са листа на дървото на зависимостите на софтуера.

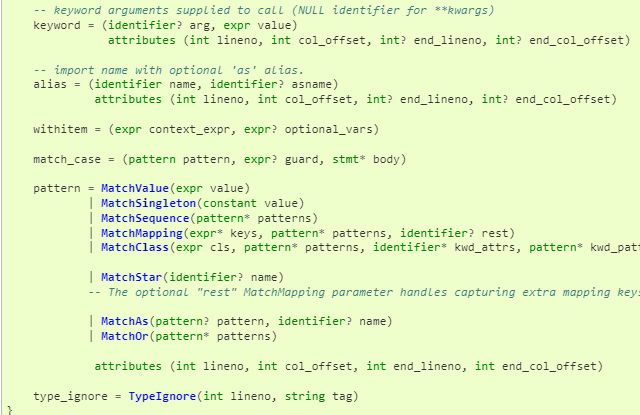
### AST [https://docs.python.org/3/library/ast.html]

AST – на български абстрактно синтактично дърво е модул вграден в езика Python, който се използва във всички програми написани на езика. Неговата цел е да разбира и работи със синтактическата граматика на езика.

Модулът дава следната абстрактна граматика върху езика:







## Open AI

Open AI е компания, която се занимава с проучване и създаване на системи базирани на изкуствен интелект. Целта на компанията е да постигна изкуствен “общ”интелект, който да помага на човечеството като цяло.

Open AI предоставя множество и различни модели, като интернет услуги. Достъпът и използването им става посредством REST и HTTP. За по-лесна употреба е създадена библиотека за Python, която обгръща тези заявки в удобен за използване Python интерфейс.

За целите на дипломната работа се използват основно два модела – GPT-3 и Codex.

Избрана е тази библиотека, защото тя има най-добро представяне в генерирането на код. Работата с тези модели е от наречения от някои Software 3.0, където вече важно е какъв вход ще се подаде на модела, за да може той да генерира резултати.

Конкурентите на тези модели нямат достатъчно добро представяне без нуждата от допълнителна настройка (fine-tuning), което е трудна и скъпа задача.

## Jupyter [https://jupyter.org]

Jupyter е проект с идеална цел, той е с отворен код. Създаден е през 2014 година като прераства от проекта IPython, който е проект за работа с езика python в команден ред (read-eval-print loop). Това, с което Jupyter надгражда и в частност тетрадките, които предлага е възможност за писане на код в така наречените блокове код, които могат да бъдат изпълнени директно в тетрадката, като контекста на изпълнения фрагмент остава в паметта. Практически тетрадката наподобява спряна с дебъгер програма, която може да бъде дописвана и изпълнявана блок по блок, като може предишни блокове да се преизпълняват. Това позволява изключително бърза итерация на различни експерименти, като се запазва състояние, което се смята за стабилно от разработчика. Точно това прави и платформата изключително популярна в света на машинното обучение, защото то изисква работа с големи обеми от данни върху, които да бъдат правени експерименти. С помощта на тази система, няма нужда големите обеми от данни да се зареждат в паметта многократно, също така ако е нужда някаква обработка тя може да бъде извършена един път и след това да се експериментира с обработените данни.

За целта на тази дипломна работа има нужда от сканиране, преглеждане и анализ на големи обеми код, което отнема съществено време. За това и е избрана тази система, с нейна помощ процеса по изграждането на инфраструктурата за анализ на хранилища за код бе изградена много по-бързо тъй като е намали многократно времето, което би отнело да се изследват различни решения, които минават и обработват хранилищата за код.

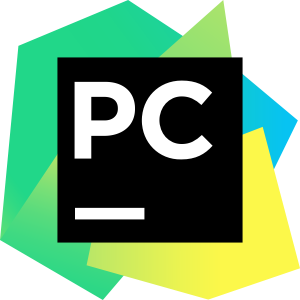
## Anaconda

Анаконда е водещата платформа за пакетен мениджър в изчислителните науки за езиците Python и R. За разработката на тази дипломна работа голяма част от използваните библиотеки са точно от този домейн на програмирането, което поставя платформата пред най-големият пакетен мениджър в Python света – pip. Друг бонус на платформата е, че се поддържа от PyCharm, както и прави опит за платформена независимост. Тази независимост се изразява в записването на библиотеките, на които зависи нашата среда по начин, чрез който при смяна на системата например от Windows на Linux ще потърси дали библиотеката съществува за Linux и ако има валидна версия, ще инсталира нея.

В разработката на системата за генериране на тестове анаконда се използва както за пакетен мениджър така и за виртуална среда. Виртуалната среда дава възможност за избор на конкретна версия на език за разработка, като средата е напълно базисна – има само базовите библиотеки по начало. Това позволява инсталация само на нужните за проекта библиотеки, което в последствие улеснява преноса на системата – на нова машина се създава същата виртуална среда и проекта е готов за изполозване.

## Интегрирана среда за разработка

Изборът на интегрирана среда за разработка бе сведен до два софтуера – PyCharm и Visual Studio Code. От двете среди е избрана PyCharm. Причината за този избор е по-добрата поддръжка за езика без нуждата от допълнителни модули и настройка на всеки от тях.

PyCharm поддържа Python както и виртуални среди, което прави разработката изключително лесна. Средата за разработка има и дебъгер в себе си, който е изключително функционален и лесен за работа. Освен това в PyCharm има и всички очаквани функционалности като допълване, преименуване на тип, метод и клас заедно с всички негови референции. Автоматичната създаване на модули в проекта също е изключително полезно, в себе си средата съдържа и графичен интерфейс за Git както и много други функционалности.

Друга важна функционалност е поддръжката на Jupyter тетрадки директно в интерфейса на средата. По този начин тетрадката използва същата виртуална среда като целия проект, освен това използва и същия контекст – т.е. вижда и може да работи с модулите от проекта.

Всичко, което е изброено се поддържа и от Visual Studio Code, чрез инсталиране на допълнителни плъгини, като всеки трябва да се настрои отделно. За това и PyCharm бе избран за разработката на тази дипломна работа.

# Описание на реализацията на разработката

Реализацията на системата представлява система, която генерира тестове базирани на променените файлове и съобщението, което е предоставено към новата версия в git. Системата има за цел да генерира тестове с приоритети както следва:

1. Генерираният код да се компилира (интерпретира) успешно.
2. Генерираният код да използва библиотека за тестване.
3. Генерираният код да може да бъде пуснат като тест и да минава успешно.
4. Генерираният код да осъществява възможно най-голямо покритие на добавения код.

За да може да се оцени работата на системата следва да се изгради нужната инфраструктура и за това и системата се разделя условно на две части: основна част, която се грижи за генерацията на код и подсистема, която има за цел да оценява генерираният код, резултатите от тази подсистема ще бъдат показани в глава IV.

По време на разработката на дипломната работа е използван IPython и Jupyter тетрадка за бързи и малки експерименти, чрез които по-бързо да се пробват различни решения свързани с генерирането на код и избора на подходящ формат за входа, който се подава на модела.

# Използвани данни и проведени експерименти

# Примерна употреба и валидация на системата