به نام خدا

گزارش پروژه اول درس هوش مصنوعی

دانشگاه صنعتی شریف

دانشکده مهندسی کامپیوتر

استاد گرامی: جناب آقای دکتر عبدی

نام و نام خانوادگی: امیرمهدی نامجو

کتابخانه‌های مورد نیاز: numpy – sickitlearn (برای یکسری محاسبه مربوط مقایسه داده‌هایی که برنامه با آن‌ها تابع را کشف کرده و یک سری داده تست برای این که ببینیم تا چه حد تابع خوبی کشف شده) – graphviz (در صورتی که بخواهید گراف رسم شود. اگر نخواهید، می‌توانید بدون این کتابخانه هم کار کنید). در فایل requirements.txt هم نام و ورژن این توابع قرار دارد.

ابتدا در مورد فایل کد داده شده توضیح می‌دهم و سپس به توضیح الگوریتم می‌پردازم.

کد من از چهار فایل اساسی به همراه یک فایل Main برای تست تشکیل شده است.

فایل Function، شامل یک کلاس است که به نوعی توابعی قابل استفاده در آن قرار دارند و در آخر نام آن‌ها در یک دیکشنری قرار گرفته است. توابع پیاده سازی شده شامل جمع، ضرب، تفریق، تقسیم، جذر، توان نمایی، لگاریتم طبیعی، سینوس، کسینوس، تانژانت، معکوس و قدر مطلق و تابع سیگموید هستند. در مورد این که چرا تابع توان را قرار نداده‌ام، در ادامه توضیح می‌دهم.

درون فایل Fitness دو کلاس هست. کلاس Fitness که درون آن سه تابع مختلف برای محاسبه Fitness تعریف شده است. این سه تابع شامل mean absolute error و mean square error و root mean square error هستند که هر سه، هر چه قدر کمتر باشند، بهتر است و به نوعی می‌توان آن‌ها را تابع هزینه دانست. در هنگام اجرای برنامه می‌توان تعیین کرد که از کدام یک استفاده بشود ولی به طور کلی و پیش‌فرض از mean absolute error استفاده می‌کنم (یعنی میانگین مجموع قدر مطلق تفاضل مقدار به دست آمده و مقدار واقعی) --- کلاس دیگر هم FitnessCoutner نام دارد و یک کلاس Singleton است که وظیفه شمارش تعداد دفعات فراخوانی تابع‌های Fitness را برعهده دارد و درون تابعی به نام raw_fitness که در DataStructure قرار دارد، استفاده شده است. به علت ناشناخته‌ای نتوانستم آن را به صورت یک المان Global تعریف کنم و به این صورت کار نمی‌کرد و برای این که ساختار توابع خودم را خیلی بهم ریخته نکنم که مدام یک عدد ثابت را به همدیگر پاس بدهند، تصمیم گرفتم از یک کلاس سینگلتون برای ذخیره سازی مقادیر دفعات فراخوانی استفاده بکنم. بدین ترتیب، این کلاس در هر جا که از آن آبجکت ساخته شود، همگی به یک آبجکت اشاره خواهند کرد و به خوبی امکان استفاده از متغیر تعریف شده درون آن وجود خواهد داشت.

درون فایل DataStructure، کلاسی با نام Chromosome وجود دارد که در اصل مربوط به نحوه پیاده سازی یک کروموزوم و ذخیره عبارت ضربی می شود. توابعی مربوط به crossover و mutation هم در این جا قرار دارند.

در نهایت درون فایل Genetic، کلاسی به نام FunctionRegressor قرار دارد. کار اصلی این تابع این است که براساس تعداد داده شده، جمعیت اولیه را تولید کرده و عملیات‌های مربوط به جهش و کراس آور و... را روی آن‌ها فراخوانی کرده و در نهایت هر بار Fitness ها را هم چک کند. در صورتی که تعداد نسل‌ها از عدد خاصی بالاتر رفت و یا این که به fitness مدنظر رسیدیم، عملیات را متوقف کند. این کلاس، از کلاس BaseEstimatior و RegressorMixin کتابخانه sickitlearn ارث بری می‌کند. اصلی‌ترین دلیل این موضوع هم برای این است که بتوان از تابع مربوط به score که در sickitlearn تعریف شده و براساس مجموعه داده ورودی و مقادیر انتظار رفته، میزان خوب بودن تابع تخمین زده شده را حساب می‌کند، استفاده کنیم. این تابع عددی بین 0 تا 1 می دهد و در اصل بیانگر R^2 یا به عبارت دیگر ضریب تشخیص (Coefficient of Determination) است که هر چه به یک

نزدیک‌تر باشد، یعنی تابع تخمین زده شده بهتر است. یک تابع به نام population_evolver هم قرار دارد که فرآیندهای اصلی تکامل نظیر کراس اور و جهش درون آن به طور دقیق هندل می‌شوند و کلاس FunctionRegressor در مواقع مورد نیاز آن را فراخوانی می‌کند.

در نهایت در تابع Main، ورودی‌ها از کاربر گرفته می شود. یکسری ورودی پیش‌فرض هم در کد قرار داده شده که در صورتی که کاربر نخواهد ورودی بدهد، صرفاً با آن‌ها کار کند.

ورودی‌هایی که می‌توانند داده بشوند:

Population: اندازه جمعیت اولیه

generation: تعداد نسل‌هایی که جلو خواهیم رفت و در نهایت جواب را چاپ می‌کنیم.

stop_limit: معیاری که زمانی که fitness به این مقدار یا کمتر برسد برنامه متوقف می شود و خروجی را مشاهده کنیم. (چون fitness های تعریف شده به نوعی تابع هزینه هستند، این حد هم به صورت کمتر از یک عدد مشخص تعریف شده است)

p_crossover و p_mutation1 و p_mutation2 و p_mutation3: احتمال‌های مربوط به جهش و کراس اور هستند. چون سه نوع جهش تعریف شده است، سه عدد احتمالاتی گرفته می شود. توجه کنید که جمع این اعداد نباید از 1 بیش‌تر شود.

long_answer_penalty_multiplier: این یک ضریب است که در صورتی که طول تابع زیاد باشد، بسته به این که هر چه قدر طول بیش‌تر باشد و تابع فاصله بیش‌تری با جواب داشته باشد، آن را جریمه می‌کند. در نتیجه تا حدی (نه خیلی زیاد) جلوی جواب‌های خیلی بزرگ گرفته می شود.

function_set: در این مورد کاربر می‌تواند تعریف کند که از چه تابع‌هایی در طول عملیات می‌خواهد استفاده کند و یا این که شکل پیش‌فرضی که در کد قرار دارد، استفاده بشود.

به طور کلی توابع در دسترسی این‌ها هستند:

add: جمع دو متغیره sub: تفریق دو متغیره mul: ضرب دو متغیره div: تقسیم دو متغیره

abs: قدر مطلق تک متغیره inv: معکوس تک متغیره neg: منفی تک متغیره sin: سینوس تک متغیره

cos: کسینوس تک متغیره tan: تانژانت تک متغیره exp: تابع نمایی با مبنای e، تک متغیره

log: لگاریتم طبیعی، تک متغیره sqrt: تابع جذر (ریشه مثبت دوم)، تک متغیره sig: تابع سیگموید، تک متغیره

به عنوان مثال اگر قصد استفاده از جمع و تفریق و سینوس را داشته باشید و بخواهید که در جواب فقط از این توابع استفاده شود، باید عبارت زیر را به عنوان ورودی بدهید:

add,sub,sin

مورد دیگر تعداد نقاطی است که قرار است برای آموزش و پیدا کردن فرمول در اختیار کد قرار بگیرد و بازه ابتدا و انتهای آن‌هاست.

number_of_points: تعداد نقاط

low: کمترین عدد

high: بیش‌ترین عدد

با استفاده از این موارد، می‌توان بازه‌ای که اعداد از آن انتخاب می‌شوند را مشخص کرد. خود اعداد، از طریق np.rand.uniform از یک توزیع یکنواخت تولید می‌شوند.

پس از آن باید مشخص کنید که قصد استفاده از تابع نوشته شده توسط خودتان را برای اجرای عملیات روی آن و محاسبه جواب و... دارید یا از چند حالت پیش‌فرض نوشته شده استفاده می‌کنید.

حالت‌های پیش‌فرض به ترتیب این موارد هستند.

1. $x^2 + 2x + 1$
2. $\sin(x) + 2$
3. $\log(x)$
4. $x\log(x)$
5. $2x^3 + 5x + 3$
6. $\sqrt{x}$
7. $\frac{1}{x+5}$

در صورتی که عبارت نامعتبر به جز 1 تا 7 وارد شود، تابع f(x) = x در نظر گرفته می شود.

اگر هم بخواهید می‌توانید خودتان تابع مدنظرتان را ورودی بدهید تا اعداد روی آن evaluate شده و برنامه هم سعی در حدس تابعی مانند آن خواهد داشت.

برای این کار، باید توجه کنید که تنها می‌توان از متغیر x استفاده کرد. هر چند کدی که نوشته شده و طراحی کرده‌ام، با توجه به استفاده از توابع آرایه‌ای numpy باید توانایی استفاده از چند متغیر را هم داشته باشد اما برای این کار نیاز به تغییراتی در هنگام ورودی دادن و reshape کردن آرایه داد که هر چند برای دو متغیر موفق به انجام آن شده‌ام اما چون خیلی روی نتایج آن مطمئن نبودم و به علاوه برای تعداد بالاتر به دلیل ددلاین امکان دیباگ فراهم نبود و در نتیجه، صرفاً فعلاً به صورت تک متغیره

نوشته‌ام و reshape آن‌ها به صورت یک آرایه تک ستونه است؛ اما در کل خود کد تابع‌های تخمین زننده، توانایی ایجاد چند متغیر در درخت عبارت را دارند.

برای وارد کردن فرمول، باید توجه کنید که برای یکسری موارد خاص که استثنا دارد، باید از شکل دیگری استفاده کنید.

مثلاً برای تقسیم، باید به صورت protected_div، برای توان نمایی به صورت protected_exp، برای لگاریتم به صورت protected_log، برای جذر به صورت protected_sqrt، برای سیگموید به صورت sigmoid_، برای تابع معکوس به صورت protected_inverse و برای توابعی نظیر سینوس و کسینوس به صورت np.sin نوشته شوند. موارد protected برای این هستند که حالت‌های خاصی نظیر تقسیم بر صفر و یا رادیکال منفی و... هندل شده باشند.

دلیل این نوع نوشتن این است که در نهایت از تابع eval خود python برای محاسبه حالت‌های وارد شده توسط کاربر استفاده شده است و در نتیجه باید عبارتی که وارد می شود، عبارت معتبر پایتونی باشد.

مثلاً:

```
protected_sqrt(x) + np.sin(x) * x
```

پس از آن امکان مشخص کردن این که از چه تابعی برای fitness (یا در اصل هزینه) استفاده شود، وجود دارد. سه حالت ممکن برای آن به ترتیب:

1. Mean absolute error
2. Mean square error
3. Root mean square error

هستند. در صورتی که عدد یا عبارت غیرمجاز وارد شود، حالت اول در نظر گرفته می شود و به طور کلی حالت default، میانگین مجموع تفاضل قدر مطلق مقدار به دست آمده از مقدار مورد انتظار است.

در نهایت، تابع اجرا شده و پس از نمایش خروجی آن، امکان رسم گرافیکی نتیجه هم وجود دارد.

خروجی تابع مثلاً به این فرم خواهد بود که در اصل فرم نمایش پیشوندی است.

```
add(x0,sub(mul(x0,2),sin(x0)))
```

دلیل استفاده از این روش این است که اگر خواستیم در آینده توابع چند متغیره مثلاً توابع سه متغیره که سه ورودی می‌گیرند و ... را اضافه کنیم، نمایش آن‌ها به فرم میانوندی دشوار خواهد بود ولی به فرم پیشوندی، هر تابعی قابل نمایش است.

برای رسم گرافیکی، توجه کنید که باید کتابخانه graphviz رسم بوده و خود نرم افزار graphviz هم نصب باشد. برای نصب نرم افزار graphviz می‌توانید از سایت آن یعنی https://graphviz.gitlab.io/ استفاده کنید. ضمناً توجه کنید در نسخه ویندوز، باید به محل نصب آن رفته و پوشه bin را در PATH سیستم قرار بدهید. (خودم روی ویندوز تست کردم ولی فکر کنم برای سایر سیستم‌عامل‌ها هم باید این کار انجام شود). مثلاً اگر مسیر پیش‌فرض نصب آن را در ویندوز 10 با معماری 64 بیتی انتخاب کنید، این آدرس باید وارد PATH سیستم بشود:

```
C:\Program Files (x86)\Graphviz2.38\bin
```

برای تولید شکل نهایی گراف، از یکسری کد مخصوص این نرم افزار درون پایتون استفاده می شود که براساس چند کد آماده که در github و stackoverflow پیدا کردم، کد آن را متناسب با برنامه خودم تغییر دادم. این کدها در یک string ذخیره شده و سپس از طریق توابع Source و render از کتابخانه پایتونی graphviz امکان رسم آن مهیا می شود. در اصل نرم افزار graphviz خود سینتکس مخصوصی دارد که من کامل به آن مسلط نیستم و صرفاً با تحقیق فهمیدم که برای رسم گرافیکی درخت عبارت، گزینه خوبی است و از چند کد آماده و تغییر دادن آن‌ها متناسب با برنامه‌ام استفاده کردم. چون هدف این پروژه هم رسم گرافیکی درخت عبارت نبوده است که بخواهم به سینتکس این نرم افزار به طور کامل مسلط بشوم.

یک نکته قابل ذکر هم این است که در حین اجرای هر الگوریتم، شماره هر نسل، طول بهترین تابع تولید شده در آن و fitness بهترین تابع تولید شده در آن ذکر می شود. منظور از طول تعداد node هاست و بهترین تابع هم با توجه به fitness های تعریف شده، تابعی است که کمترین عدد fitness را داشته باشد. چون fitness هایی که تعریف کرده‌ام، در اصل به نوعی تابع هزینه هستند.

موضوع دیگر مربوط به ساخت جمعیت اولیه است که در تابع build_chromosome در کلاس Chromosome انجام می شود. ایده کلی کار این است که براساس عدد تصادفی رندوم تولید شده، مشخص می شود که باید تابع ایجاد کنیم یا متغیر/عدد. متغیرها و اعداد را طبق چیزی که از کتاب A Field Guide to Genetic Programming خواندم، terminal می‌نامند زیرا شاخه درخت بعد از آن‌ها ادامه پیدا نمی‌کند و به نوعی آن شاخه را ترمینیت می‌کنند. براساس عدد رندوم تولید شده، از میان توابعی که در function_set قرار دارد، یا تولید عدد ثابت، یا تولید متغیر (که البته در این جا فقط یک متغیر داریم) یک مورد انتخاب می شود. عدد رندوم هم از بازه‌ای به نام const_range ایجاد می شود که به طور پیش‌فرض، بین منفی 1 و 1 قرار داده‌ام اما می‌توان در هنگام ساختن کلاس FunctionRegeressor مقدار آن را تغییر داد. در مورد توابع، وقتی ایجاد بشوند، بسته به این که نیاز به چند ورودی دارند، در یک استک به نام terminal_stack تعداد آرگومان‌های آن اضافه می شود. پس از آن، هر بار که یکی از آرگومان‌های آن پر شود، از عدد درون انتهای terminal_stack یک واحد کم می شود و هر درایه terminal_stack که صفر بشود، از استک پاپ می شود. بدین ترتیب، مشکل این که چه طور با اضافه شدن توابع با تعداد متغیرهای جدید، بتوانیم اعداد توابع قبلی را هم حفظ کنیم، برطرف می شود؛ زیرا به دلیل ساختار استک مانند آن، مثلاً برای چنین تابعی:

add(sin(x) , sub(x,2))

به ترتیب مقادیر درون استک به این صورت می شود: (انتهای استک سمت راست است)

2 -> add

1,1 -> add(sin(

1,0 ->add(sin(x) ,

Pop 0

0,2 -> add(sin(x) , sub(

0,1 -> add(sin(x) , sub(x,

0,0 -> add(sin(x) , sub(x,2))

Pop 0

Pop 0

در مورد چالش‌های پیاده سازی، یکی از چالش‌های اساسی، نحوه ذخیره درخت عبارت است که در این جا من از یک لیست پایتونی استفاده کرده‌ام و فرمول را به صورت پیشوندی ذخیره می‌کند، در کروموزوم استفاده کرده‌ام. مثلاً عبارت زیر به این صورت ذخیره می شود:

add(cos(sub(sin(X0), sin(X0))), add(sin(sin(0.942)), sub(sin(X0), sin(-0.289))))

function add
function cos
function sub
function sin
int 0
function sin
int 0
function add
function sin
function sin
float 0.942
function sub
function sin
int 0
function sin
float -0.289

توجه کنید که int ها بیانگر شماره متغیر هستند و برای حالاتی هستند که بتوانیم از چند متغیر استفاده کنیم و برای متغیرهای بیش‌تر اعداد دیگر استفاده می شود. هر چند همان طور که گفتم، از آن جایی که در مورد کارکرد درست الگوریتم برای بیش از یک متغیر مطمئن نبودم، ورودی‌ها به صورتی هستند که تک متغیره برداشت می‌شوند (چون تنها یک ستون دارند) و در نتیجه در این جا چون همواره یک متغیر داریم، همه int ها صفر شده‌اند. اگر خود عدد صفر را بخواهیم، به صورت float 0. ذخیره می شود.

با توجه به این که در کلاس function مشخص شده که هر تابع چند ورودی می‌خواهد، می‌توان با این نحوه ذخیره سازی، به شکل یک تا توابع را ذخیره کرد. بدین ترتیب، امکان انجام عملیات‌های جهش و... هم به خاطر ذخیره شدن این ساختار درختی در آرایه، بهتر مهیا می شود.

الگوریتم‌های اساسی استفاده شده برای Crossover و Mutation به این صورت بوده‌اند:

ابتدا باید ذکر کنم که منظور از مسابقه در جمله‌های بعدی چیست. در بخش‌هایی از الگوریتم، یک زیرمجموعه از جمعیت با سایز مشخص (به طور پیش‌فرض در کد 20 فرض شده ولی قابل تغییر است)، انتخاب شده و Fitness آن‌ها حساب شده و بهترینشان انتخاب می شود. این کار را مسابقه می‌گوییم.

برای Crossover: ابتدا برنده یک مسابقه به عنوان گیرنده انتخاب می شود. سپس برنده یک مسابقه دیگر هم به عنوان اهدا کننده انتخاب می شود. سپس یک قسمت (زیردرخت) از گیرنده برای جایگزینی به صورت رندوم مشخص می شود. سپس یک قسمت (زیردرخت) هم از اهدا کننده انتخاب می شود و زیردرخت اهدا کننده، جایگزین زیردرخت گیرنده می شود.

برای Mutation با تحقیقاتی که من کردم، نظیر کتابخانه‌هایی که تقریباً همین روش‌های تخمین تابع را پیاده سازی کرده‌اند و همچنین کتاب A Field Guide to Genetic Programming، متوجه شدم که به شکل‌های مختلفی می‌توان این کار را انجام داد. از آن جایی که دیدم در بسیاری از کتابخانه‌ها، سه روش Hoist، Subtree و Point برای جهش انتخاب شده‌اند، این سه روش را پیاده سازی کردم.

روش اول mutation1 به صورت Subtree Mutation است. در این روش ابتدا در طی یک مسابقه که در یک زیرمجموعه رندوم از توابع انجام شده، بهترین تابع انتخاب شده. سپس یک زیردرخت آن انتخاب گردیده و یک زیر درخت به صورت کاملا رندوم در آن نقطه تولید شده و جایگزین آن می شود.

روش دوم mutation2 به صورت Hoist Mutation است. در این روش، ابتدا در طی یک مسابقه که در یک زیرمجموعه رندوم از توابع انجام می‌گیرد، بهترین تابع انتخاب می شود. سپس یک زیر درخت از آن انتخاب می شود. سپس یک زیر درخت از این زیردرخت (یعنی از جای پایین‌تر) انتخاب شده و جایگزین کل زیر درخت می شود. در اصل این روش، باعث می شود که جلوی ایجاد توابع خیلی بزرگ هم تا حدی گرفته شود.

روش آخر mutation3، همان روش معمول Point Mutation است. در این روش، ابتدا در طی یک مسابقه که در یک زیرمجموعه رندوم از توابع انجام می‌گیرد، بهترین تابع انتخاب می شود. سپس براساس احتمال تعیین شده با نام p_point_replace، یکسری از نقاط و Node های رندوم در درخت انتخاب می‌شوند (هر Node به اندازه p_point_replace احتمال انتخاب دارد. این احتمال به صورت پایه 0.05 است ولی می‌توان در هنگام ساختن آبجکت از کلاس FunctionRegeressor آن را تغییر داد، اما به هر حال من امکان ورودی دادن این مورد را توسط کاربر ایجاد نکردم، چون عددهای دیگر ممکن است الگوریتم را با مشکل مواجه کند). بعد از انتخاب این Node های رندوم، آن‌ها با موارد دیگری جایگزین می‌شوند. اگر عدد باشند، با یک عدد ثابت دیگر. اگر تابع باشند، با یک تابع دیگر که همان تعداد ورودی می‌گیرد و اگر هم متغیر باشند، با یک متغیر دیگر. البته چون در کدی که زدم، فعلاً امکان انتخاب تنها یک متغیر وجود دارد، این مورد بی‌اثر

است اما کد به طور کلی پتانسیل استفاده از چند متغیر را هم دارد ولی چون از کارکرد صحیح آن مطمئن نبودم، از این مورد در تست‌ها استفاده نکرده‌ام.

یکسری احتمالات برای Crossover و Mutation های 1 تا 3 در ابتدای برنامه وارد می شود. در صورتی که جمع آن‌ها کمتر از یک باشد، بقیه احتمال، مربوط به حالت Reproduction می شود که در این حالت یک تابع، به طور مستقیم به نسل بعد منتقل می شود.

در مورد شرایط اتمام، همان طور که پیش‌تر گفته شد، براساس دو متغیر generation و stop_limit است. در صورتی که تعداد نسل‌ها از generation بیش‌تر شود و یا این که fitness از stop_limit کمتر شود (به fitness ایده آل برسیم)، برنامه به پایان می‌رسد و بهترین تابع را بر می‌گرداند. (با بهترین fitness که با توجه به این که توابع تعریف شده که پیش‌تر گفته شد، همگی به نوعی تابع هزینه و دور بودن از هدف هستند، کمترین عدد fitness بهترین است).

در مورد چالش‌ها به طور کلی پیش‌تر توضیحاتی دادیم. فقط یک چالش می‌ماند و آن هم مسائل مربوط به تقسیم بر 0 و... هستند. به دلیل تولید رندوم، ممکن است در بعضی جاها با چنین مواردی رو به رو شویم. برای این کار توابعی را با نامی که در پیشوند کلمه protected آمده است، تعریف کرده‌ایم.

تابعی نظیر protected_div در صورتی که مخرج 0 باشد، خروجی را 1 بر می‌گرداند. همچنین اگر قدر مطلق عدد کمتر از 0.001 باشد هم 1 بر می‌گرداند. (در اصل به نوعی این اعداد حکم overflow دارند)

برای protected_sqrt، قدر مطلق مقدار را در نظر می‌گیریم.

برای protected_log، اگر عدد از 0.001 کوچک‌تر باشد، مقدار 0 برگردانده می شود. (در اصل حکم Overflow منفی را دارد). همچنین قدر مطلق مقدار درون آن در نظر گرفته می شود تا درون log منفی نشود.

برای protected_exp هم اگر توان از 10 بزرگ‌تر شود، مقدار آن به ازای 10 برگردانده می شود. این کار نیز برای جلوگیری از overflow بوده است.

تابع سیگموید را هم طبق تحقیقاتی که در اینترنت داشتم و نمونه‌های عملیاتی از مواردی مشابه این برنامه را که دیدم، در بعضی اوقات که جواب‌هایی معمول خوب نیستند، این تابع جواب‌های خوبی به دست می دهد و برای همین آن را هم در لیست توابع قرار داده‌ایم. فرمول این تابع به صورت:

$$Sigmoid(x) = \frac{1}{1 + e^{-x}}$$

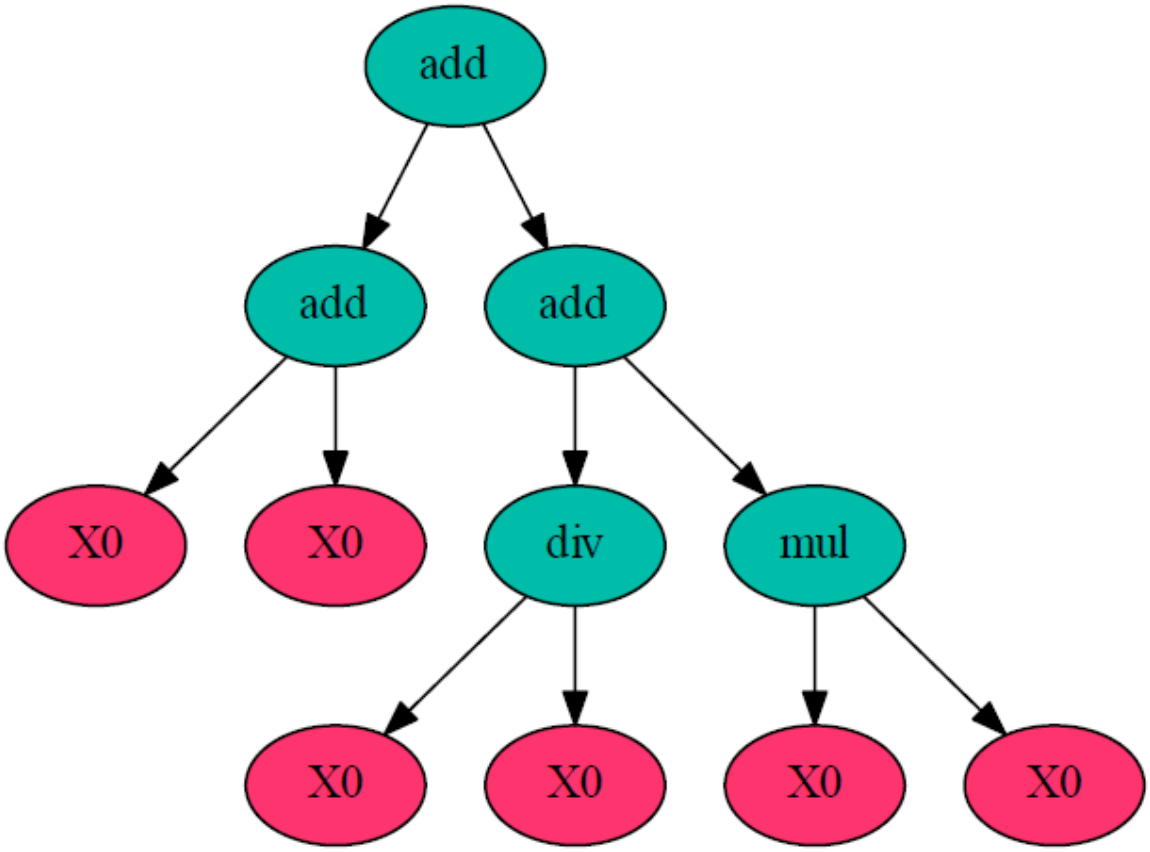
است. هر چند خودم روی این تابع به دلیل ددلاین، فرصت نکردم که تست‌های زیادی انجام دهم و در حالت ***default*** هم در لیست function_set از آن استفاده نمی‌کنم، چون در مورد خوب یا بد بودن آن براساس پیاده سازی که انجام داده‌ام مطمئن نیستم. فرم قابل استفاده آن برای numpy را هم که مشکلات مربوط به overflow نداشته باشد در اینترنت پیدا کردم ولی همان طور که گفتم، به طور کلی این تابع را تست نکردم و از عملکرد درست و صحیح آن مطمئن نیستم.

مسئله آخر مربوط به این است که من تابع توان را قرار ندادم. از یک طرف توان‌های معمول و طبیعی از طریق عبارات ناشی از ضرب قابل ساخت هستند. ولی اگر تابع توان را به صورت کلی قرار می‌دادم، نیاز به شرط‌های بسیار زیادی برای هندل کردن حالت‌های خاص آن به وجود می‌آمد. مثلاً این که توان منفی باشد و متغیر یا عدد پایه 0 بشود و یا این که توان اعشاری باشد و پایه منفی بشود و مشکل ایجاد اعداد موهومی به وجود می‌آید، یا این که مثلاً دو عدد نسبتاً بزرگ برای توان و پایه ایجاد بشود و نتیجه بسیار بزرگ ایجاد بشود که کل الگوریتم را دچار اختلال کند. به دلیل همین موضوع و این که هندل کردن این موارد، موجب ایجاد ناپایداری در الگوریتم می شود، کلاً توان را در تابع قرار ندادم. مثلاً یکی از راه‌های هندل کردن آن این است که در حالت‌های نامعتبر، خروجی 0 بشود ولی آیا این که 0 را جایگزین یک عدد خیلی بزرگ کنیم، کار درستی است؟ این کار را البته

در *inv* انجام داده‌ایم اما تکرار زیاد این حالت‌های استثنا که برای هر حالت خاص یک عدد دیگر بگذاریم، ممکن است موجب ایجاد فرمولی غیر منطبق بر واقعیت بشود و برای همین توان را در این جا قرار نداده‌ام.

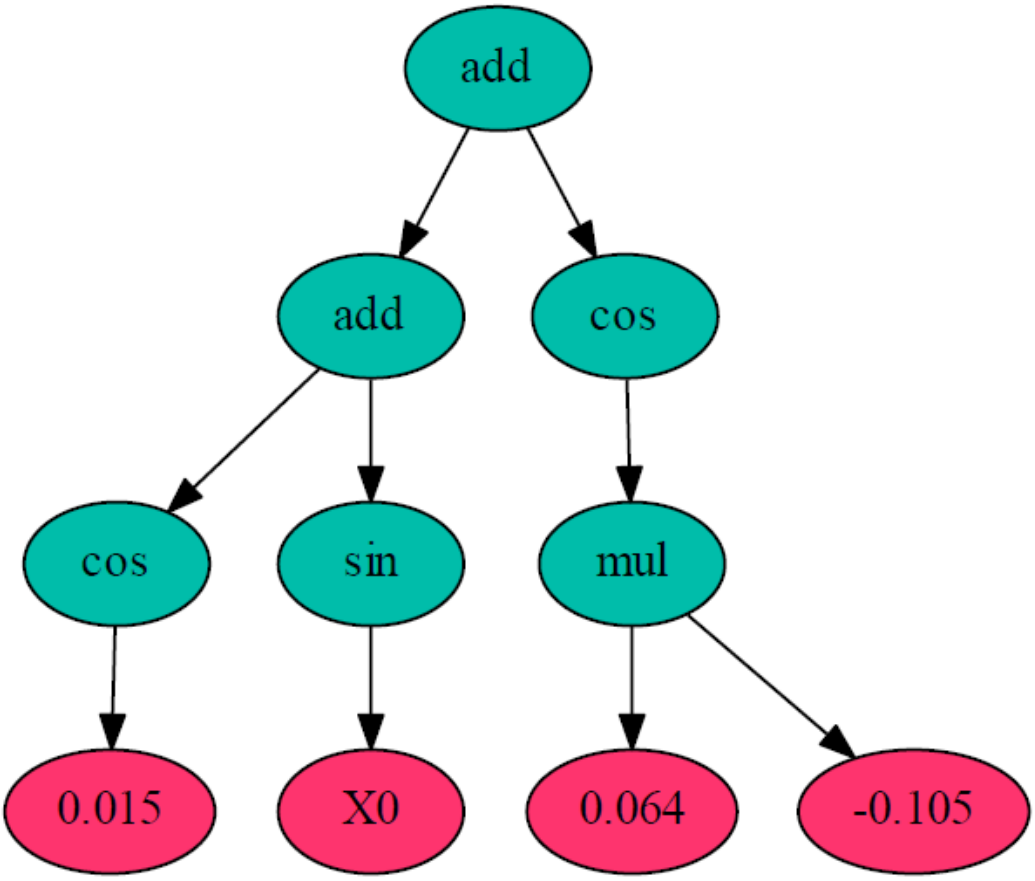
در مورد هفت تابع پیش‌فرض قرار داده شده، نتیجه گراف تولید شده بدین صورت است:

1. $x^2 + 2x + 1$



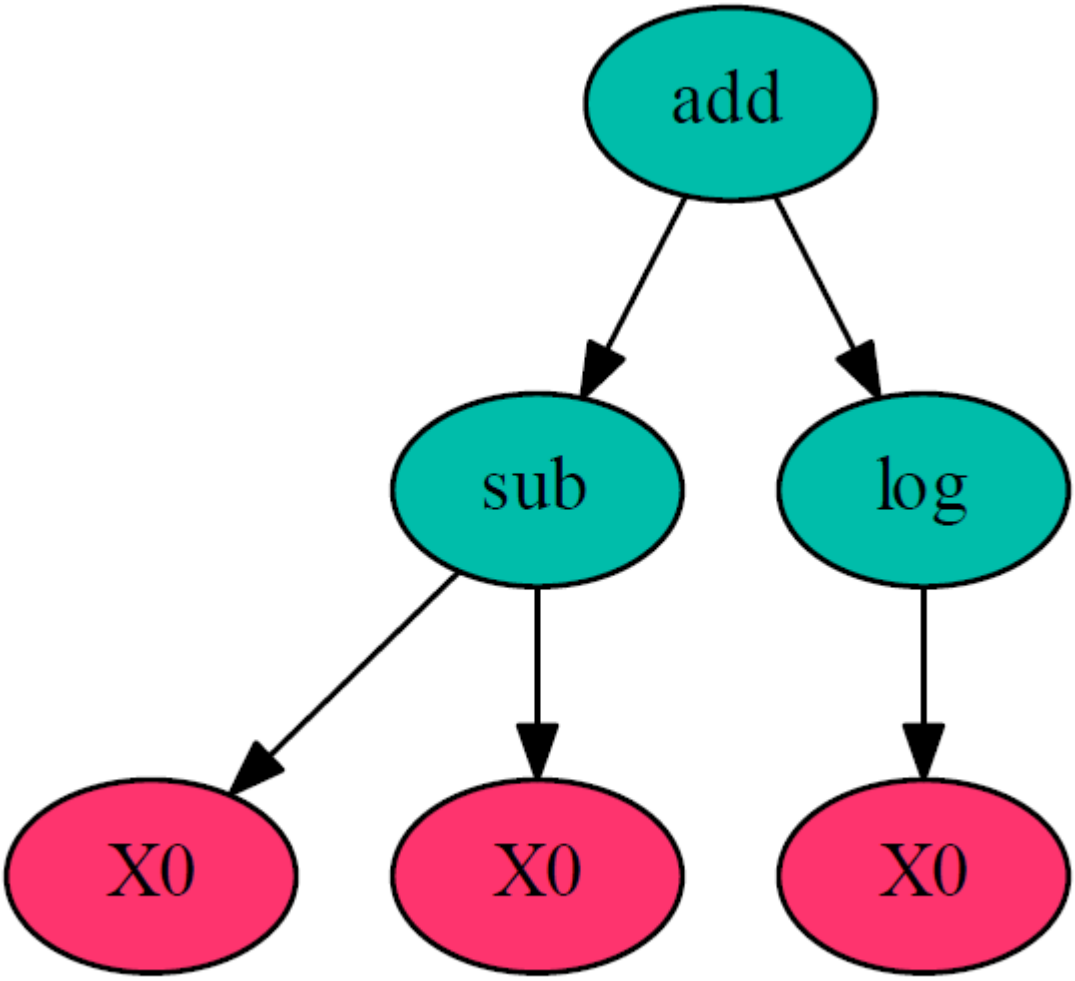
نتیجه نسبتاً خوب شده ولی شمال عبارت‌های بی‌فایده نظیر تقسیم یک عدد بر خودش است که برطرف کردن آن نیازمند زمان بیش‌تر برای نوشتن کدی است که ساده سازی عبارات را هم انجام دهد.

2. $\sin(x) + 2$



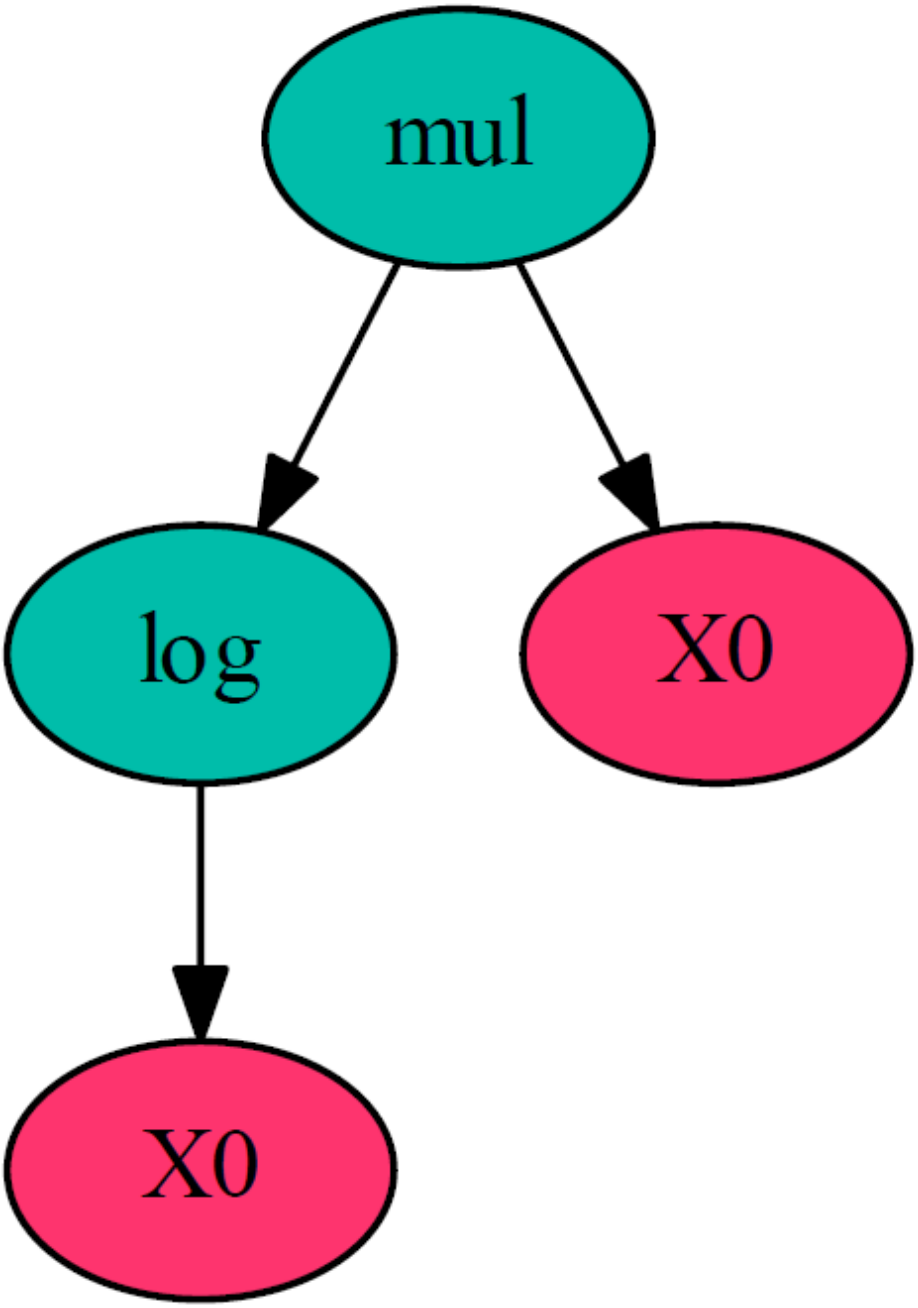
نتیجه نسبتاً خوبی به دست آمده ولی شاهد تشکیل عدد 2 به شکل عجیبی از یکسری کسینوس هستیم.

3. $\log(x)$



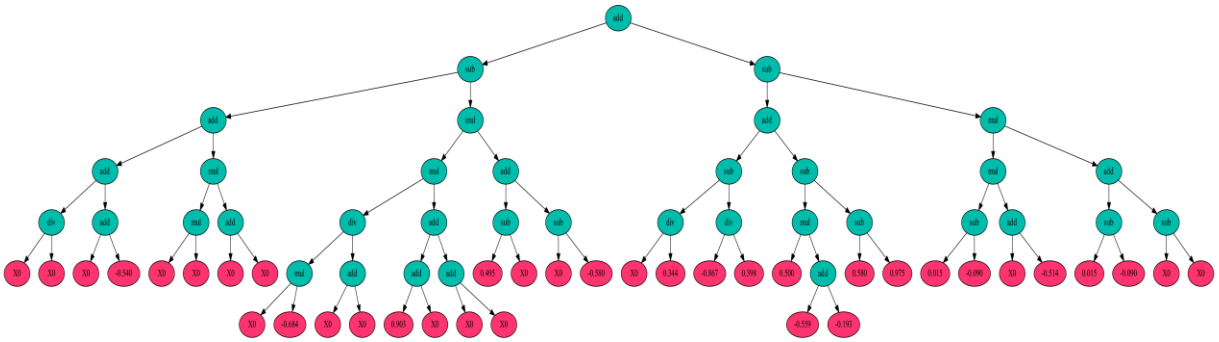
نتیجه شامل عملیات‌های بیهوده نظیر کم کردن یک متغیر از خودش بود. چنین مواردی در صورتی که وقت خیلی بیش‌تری روی کد بگذاریم، قابل رفع هستند ولی به هر حال از مشکلات کلی الگوریتم هستند که باید از طریق روش‌های جداگانه برطرف شوند و برطرف کردن آن‌ها در حین اجرای خود الگوریتم (لااقل الگوریتمی که من استفاده کرده‌ام) کار چندان آسانی نیست.

4. $x\log(x)$



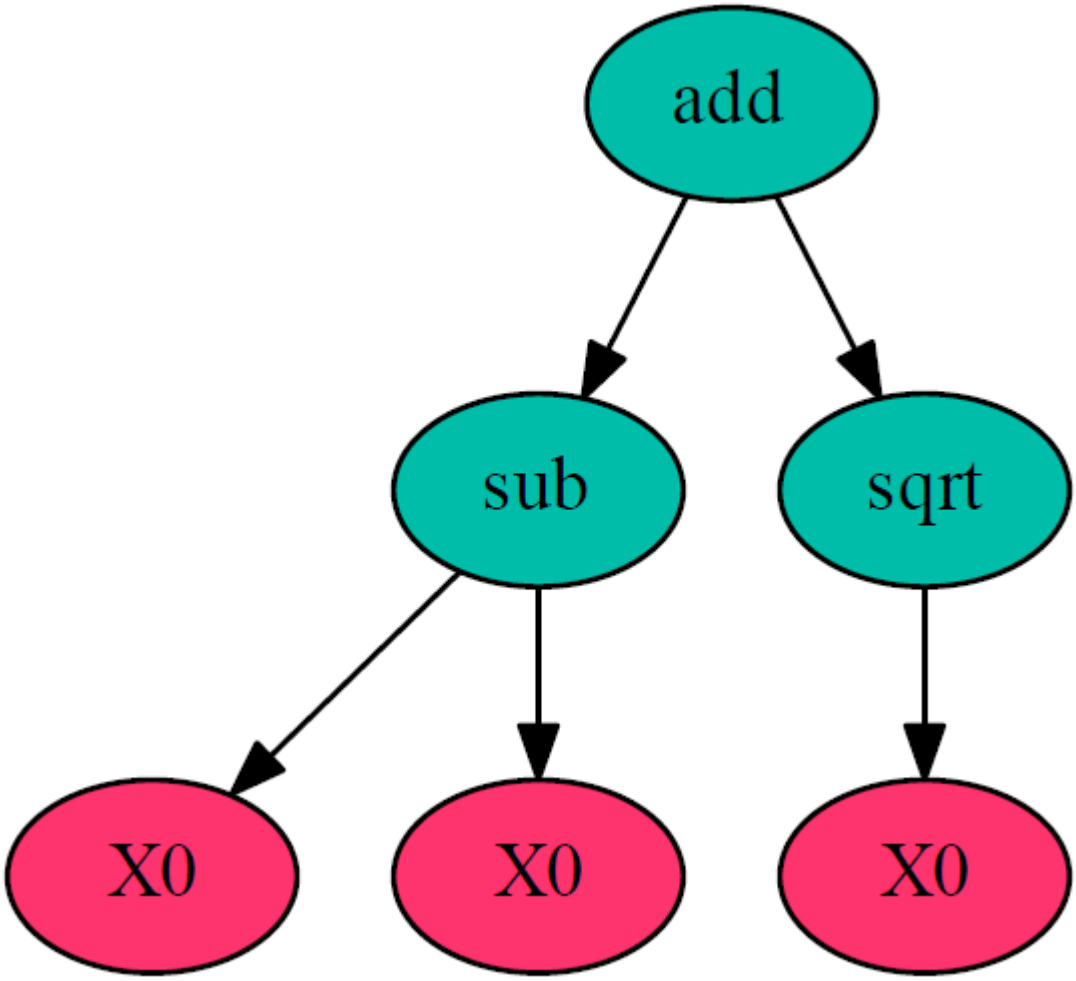
نتیجه بسیار خوب و دقیق به دست آمد.

$2x^3 + 5x + 3$ .5



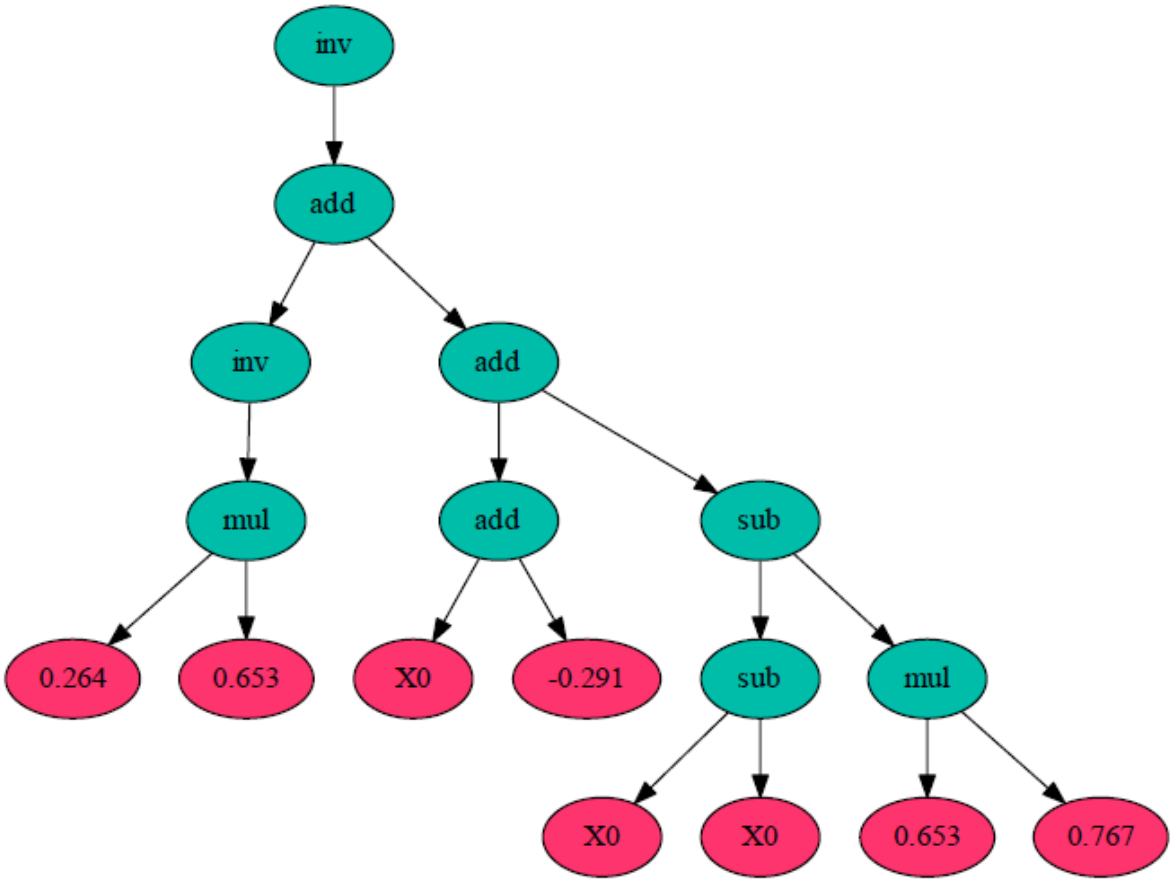
مشاهده می شود که نتیجه بسیار بزرگی تولید شده. هر چند در بعضی اجراهای دیگر، نتایج نسبتاً بهتری به دست آمدند؛ اما این مورد را به عنوان نمونه‌ای از نتایج نسبتاً طولانی که گاهی اوقات در الگوریتم پیش می‌آید، قرار داده‌ام.

6. $\sqrt{x}$



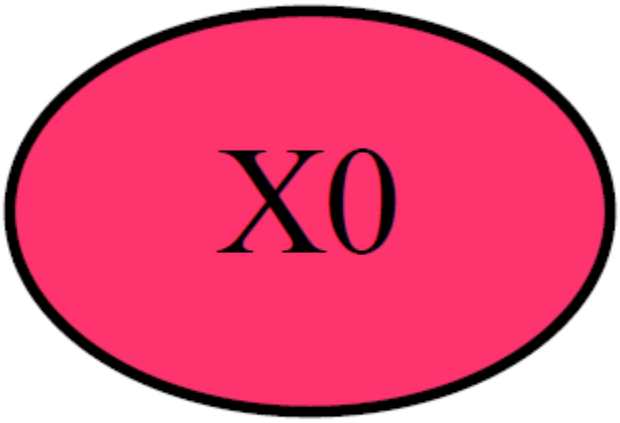
نتیجه نسبتاً خوب به دست آمده ولی همچنان شامل یک عبارت بیهوده است.

7. $\frac{1}{x+5}$



نتیجه به نسبت طولانی شده است ولی در تست انجام شده با آن روی مجموعه تست که جداگانه در کد ساخته می شود و تابع برای آن بهینه نشده است، نتایج خوب و قابل قبولی به دست آمد.

حالت پیش‌فرض در صورت وارد شدن عدد نامعتبر f(x) = x



با تمام این موارد و چیزی که پیاده سازی کردم و نمراتی که بعضاً برای توابع پیچیده توسط تابع score کتابخانه sickitlearn داده می‌شد، متوجه شدم که Genetic Programming روشی خوب برای حل مسائلی نظیر این مورد است که حالت‌های بسیار زیادی برای جواب‌های آن وجود دارد. در این روش در اکثر اوقات اگر بدشانس نباشیم و ناگهان الگوریتم در یکسری تابع بد رفتار گیر نیفتد، به جواب‌های قابل قبولی می‌توان دست یافت. به علاوه طوری که من کد زدم، امکان اضافه کردن توابعی با چند آرگومان مثلاً تابعی با سه یا چهار آرگومان و بیش‌تر هم وجود دارد که هر چند عملکرد الگوریتم را روی آن‌ها تست نکردم ولی به نظر اگر توابعی که در نظر گرفته شده، تابع مناسبی برای ایجاد توابع دیگر باشد، می‌توان به دست آوردن نتایج خوب امیدوار بود.

با این وجود یکی از ضعف‌های این الگوریتم که شاید برطرف کردن آن هم چندان ساده نباشد، بعضاً تولید جواب‌های بسیار طولانی است. در بعضی از موارد شاهد هستیم که با توابعی بسیار غول‌پیکر برای توصیف یک تابع ساده مواجه می‌شویم. بخشی از این مشکل با ساده‌سازی‌ها و مثلاً انجام عدد یکسری بخش‌ها نظیر توابعی که روی عدد ثابت اعمال شده‌اند، قابل رفع است که به دلیل زمان پروژه، امکان زدن کدی که ساده‌سازی‌ها را هم انجام بدهد برای من مهیا نبود. ولی با این وجود، این مشکل بزرگ شدن توابع همچنان وجود دارد. هر چند من یک ضریب هم اضافه کردم که توابعی متناسب با طولشان تا حدی جریمه بشود تا در میان موارد نزدیک به هم، تابع با طول کمتر انتخاب شود اما به هر حال همچنان شاهد توابع بزرگ هستیم. این مشکلی است که در اکثر برنامه‌های Genetic Programming وجود دارد و رفع آن نیازمند تلاش بسیار زیاد و مضاعف است. با این وجود کارآمدی این روش برای ایجاد توابع قابل قبول و قابل استفاده برای تخمین، بسیار بالاست و امکان استفاده عملی از آن در بسیاری از حوزه‌ها وجود دارد.