این قسمت یک کلاس به نام "Individual" را تعریف می‌کند، که یک نمونه فرد را در یک جمعیت ژنتیکی برای الگوریتم ژنتیک نشان می‌دهد. این کلاس دارای ویژگی‌های مختلفی است که در زیر توضیح داده شده است:

`chromosome` - یک آرایه که نشان دهنده ژنوم فرد است و شامل مقادیر 0 و 1 است.

`objectives` - یک لیست خالی که برای ذخیره هدف‌های فرد استفاده می‌شود. این هدف‌ها در طول اجرای الگوریتم محاسبه می‌شوند.

`dominated_by` - یک مجموعه از داده‌ها که توسط فرد فعلی غلبه شده‌اند. این مجموعه به طور پیش فرض خالی است.

`dominates` - تعداد داده‌هایی که توسط فرد فعلی غلبه شده‌اند.

`rank` - رتبه فرد در جمعیت ژنتیکی که توسط الگوریتم NSGA-II تعیین می‌شود.

در قسمت بعدی :

`initialize_population` - این تابع یک جمعیت اولیه از افراد را با اندازه و طول ژنوم مشخص شده ایجاد می‌کند. برای هر فرد، یک آرایه تصادفی از صفرها و یک‌ها به عنوان ژنوم ایجاد می‌شود و در جمعیت قرار می‌گیرد. سپس جمعیت ایجاد شده برگردانده می‌شود.

`df` - این خطوط کد یک (DataFrame) را از یک فایل CSV به نام "DS02.csv" می‌خوانند و آن را به صورت تصادفی مرتب می‌کنند.

قسمت بعدی از کد مقاله شامل دو تابع است: `flip` و `mutation` :

تابع `flip` یک آرگومان به نام `x` دریافت می‌کند و بررسی می‌کند که آیا `x` برابر با صفر است یا نه. اگر `x` برابر با صفر باشد، تابع مقدار یک را برمی‌گرداند، و در غیر این صورت (یعنی `x` برابر با یک است)، مقدار صفر را برمی‌گرداند. به عبارت دیگر، این تابع مقدار یک را به صفر تبدیل می‌کند و بالعکس.

تابع `mutation` یک آرگومان به نام `ind` (یک نمونه از کلاس Individual) و یک آرگومان به نام `m_rate` (نرخ جهش) دریافت می‌کند. این تابع یک کپی از ژنوم فرد را با نام `m_chromosome` ایجاد می‌کند. سپس به تمام عناصر `m_chromosome` (تک تک ژن‌ها) نگاه می‌کند. اگر عدد تصادفی تولید شده کمتر از `m_rate` باشد، یعنی با احتمال `m_rate`، آن ژن را تغییر می‌دهد. برای این منظور از تابع `flip` استفاده می‌کند تا ژن را تغییر دهد. سپس یک نمونه جدید از کلاس Individual با ژنوم جدید را برمی‌گرداند.

این قسمت از کد مقاله برای اعمال جهش به یک نمونه فرد استفاده می‌شود. در هر دوره از الگوریتم ژنتیک، با احتمال `m_rate`، هر ژن از ژنوم فرد تغییر می‌کند. این عملیات جهش به الگوریتم کمک می‌کند تا از محل میانبرها در فضای جستجوی بهتر بهره ببرد و به جواب‌های جدید و بهتر برسد.

این قسمت از کد مقاله به تابع `crowding_distance` اختصاص داده شده است.

تابع `crowding_distance` یک آرگومان به نام `front` دریافت می‌کند که یک لیست از افراد در یک (front) چند هدفه است. هدف این تابع، محاسبه فاصله برای هر فرد در front است که در ادامه توضیح داده می‌شود.

ابتدا، اندازه را با استفاده از تابع `len` و ذخیره در متغیر `front_size` محاسبه می‌کند. سپس یک آرایه صفر با اندازه `front_size` به نام `dis` ایجاد می‌شود که برای ذخیره فاصله هر فرد استفاده می‌شود. سپس، یک حلقه `for` برای هر هدف در هدف‌های داده‌ها ایجاد می‌شود. ابتدا front را بر اساس مقدار هدف مرتب می‌کند با استفاده از تابع `sorted` و با استفاده از لمبدا `lambda` و تابع `ind.objectives[m]` برای مقایسه داده‌ها بر اساس هدف `m`. سپس، مقدار بی‌نهایت (inf) به عنوان مقدار اول و آخر در آرایه `dis` قرار می‌گیرد.

سپس، یک حلقه `for` دیگر برای اینکه از دومین فرد تا قبل از آخرین فرد در front بگذرد، ایجاد می‌شود. در هر مرحله، فاصله داده بعدی نسبت به داده قبلی و داده بعدی در هدف `m` محاسبه می‌شود و به مقدار `dis[i]` اضافه می‌شود.

در نهایت، زوج‌هایی از داده و فاصله محاسبه شده با استفاده از تابع `zip` ایجاد شده و بر اساس فاصله به صورت نزولی مرتب می‌شوند (با استفاده از لمبدا `lambda x: x[1]`) و نتیجه برگردانده می‌شود.

به طور خلاصه، تابع `crowding_distance` برای محاسبه فاصله برای هر فرد در front استفاده می‌شود. این فاصله نشان می‌دهد که هر فرد در میان front چقدر به دیگر داده‌ها نزدیک است و در نهایت به الگوریتم انتخاب نمونه‌های برتر کمک می‌کند.

این قسمت از کد مقاله به تابع `environmental_selection` اختصاص داده شده است.

تابع `environmental_selection` دو آرگومان دارد: `pop` که یک لیست از جمعیت (افراد) است، و `pop_size` که اندازه جمعیت مورد نظر برای بازگشت به عنوان نتیجه است . ابتدا یک لیست خالی به نام `remaining_pop` ایجاد می‌شود که برای ذخیره افراد انتخاب شده است. همچنین یک متغیر `i` با مقدار صفر تعریف می‌شود. سپس یک حلقه `while` ایجاد می‌شود که تا زمانی که تعداد افراد در `remaining_pop` به همراه تعداد افراد در `pop[i]` کمتر یا مساوی `pop_size` باشد، ادامه می‌یابد. در هر مرحله، افراد front توسط `pop[i]` به لیست `remaining_pop` اضافه می‌شوند و مقدار `i` یک واحد اضافه می‌شود.

سپس یک شرط `if` بررسی می‌کند که آیا تعداد افراد در `remaining_pop` کمتر از `pop_size` است یا خیر. اگر کمتر باشد، ابتدا front را توسط `pop[i]` را بر اساس فاصله محاسبه شده با استفاده از تابع `crowding_distance` مرتب می‌کند و در متغیر `sorted_front` ذخیره می‌کند. سپس به لیست `remaining_pop` تعداد `pop_size - len(remaining_pop)` از افراد با بیشترین فاصله (با استفاده از لمبدا `sorted_front[j][0]`) اضافه می‌شوند.

در نهایت، لیست `remaining_pop` که شامل افراد انتخاب شده است را برمی‌گرداند.

به طور خلاصه، تابع `environmental_selection` برای انتخاب افراد جدید در جمعیت بر اساس فاصله و اندازه جمعیت مورد نظر استفاده می‌شود. این تابع ابتدا افرادی را که از front های قبلی انتخاب می‌شوند، به `remaining_pop` اضافه می‌کند و سپس اگر تعداد افراد به اندازه مورد نظر نباشد، افراد اضافی را بر اساس فاصله انتخاب می‌کند.

تابع `non_dominated_sorting` برای انجام مرتب‌سازی بدون اولویت بر روی یک جمعیت از افراد استفاده می‌شود. این تابع دو آرگومان دارد: `pop` که لیستی از افراد است و `fronts` که لیستی از (fronts) را برمی‌گرداند.

ابتدا یک لیست `fronts` با یک لیست خالی ایجاد می‌شود. سپس یک حلقه `for` برای هر فرد (`ind`) در جمعیت ایجاد می‌شود. در هر مرحله، متغیرهای `dominated_by` و `dominates` برای فرد `ind` ایجاد می‌شوند و از نوع `set` و `int` هستند، به ترتیب. سپس یک حلقه `for` دیگر برای هر فرد دیگر (`s`) در جمعیت ایجاد می‌شود. در این حلقه `for`، شرطی برای بررسی اینکه آیا فرد `ind` از فرد `s` تسلط می‌کند یا خیر، بررسی می‌شود. اگر فرد `ind` در همه‌ی هدف‌ها کمتر یا مساوی فرد `s` باشد، به مجموعه `dominated_by` فرد `ind` اضافه می‌شود. همچنین اگر فرد `ind` در همه‌ی هدف‌ها بزرگتر یا مساوی فرد `s` باشد، مقدار `dominates` فرد `ind` یک واحد افزایش می‌یابد. در نهایت، اگر مقدار `dominates` برابر با صفر باشد، رتبه (rank) فرد `ind` صفر قرار می‌گیرد و به `fronts[0]` اضافه می‌شود. سپس یک متغیر `i` با مقدار صفر تعریف می‌شود. سپس یک حلقه `while` ایجاد می‌شود که تا زمانی که جبهه `fronts[i]` خالی نباشد، ادامه می‌یابد. در هر مرحله، یک لیست خالی به نام `FN` ایجاد می‌شود. سپس یک حلقه `for` برای هر فرد (`ind`) در `fronts[i]` ایجاد می‌شود. در این حلقه `for`، برای هر فرد `ind`، حلقه `for` دیگری برای هر فردی (`s`) که توسط `ind` تسلط شده است ایجاد می‌شود. در این حلقه `for`، مقدار `dominates` فرد `s` یک واحد کاهش می‌یابد و اگر مقدار `dominates` برابر با صفر شود، رتبه (rank) فرد `s` به `i + 1` تنظیم می‌شود و به لیست `FN` اضافه می‌شود. سپس مقدار `i` یک واحد افزایش می‌یشود.

در نهایت، fronts ها که شامل افراد با رتبه‌های مختلف هستند، به جز fronts آخر (`fronts[-1]`) که ممکن است خالی باشد، برگردانده می‌شوند.

تابع `dominate` نیز یک تابع کمکی است که دو فرد (`ind1` و `ind2`) را دریافت می‌کند و بررسی می‌کند که آیا `ind1` از `ind2` تسلط می‌کند یا خیر. برای این منظور، بررسی می‌شود که آیا `ind1` در همه‌ی هدف‌ها کمتر یا مساوی `ind2` است یا خیر. اگر برقرار باشد، تابع `dominate` مقدار `True` برمی‌گرداند و در غیر این صورت، `False` برمی‌گرداند.

به طور خلاصه، تابع `non_dominated_sorting` برای انجام مرتب‌سازی بدون اولویت بر روی یک جمعیت از افراد استفاده می‌شود. این تابع ابتدا رتبه‌ها را بر اساس تسلط فرد بر دیگران محاسبه می‌کند و سپس fronts ها را بر اساس رتبه‌ها ساختاردهی می‌کند و برمی‌گرداند. تابع `dominate` نیز برای بررسی تسلط یک فرد بر دیگری استفاده می‌شود.

تابع `Roulette_wheel_selection` برای انتخاب یک عضو از مجموعه `Q` عضو با استفاده از روش چرخه یا چرخش رولت استفاده می‌شود. این تابع دو آرگومان دارد: `p` که لیستی از احتمالات مربوط به هر عضو در مجموعه است و `Q` که تعداد کل عناصر مجموعه را نشان می‌دهد. تابع `Roulette_wheel_selection` یک عدد صحیح بین ۱ و `Q` را به عنوان خروجی برمی‌گرداند.

ابتدا یک لیست به نام `probability_sum` ایجاد می‌شود که مجموع احتمالات تجمعی را برای هر عضو در مجموعه `p` محاسبه می‌کند. به عبارت دیگر، هر عضو `i` از `probability_sum` برابر با مجموع احتمالات عضوهای `p[0]` تا `p[i]` است.

سپس یک عدد تصادفی بین ۰ و ۱ با استفاده از تابع `np.random.rand()` تولید می‌شود و در متغیر `rand` ذخیره می‌شود.

سپس یک حلقه `for` ایجاد می‌شود که برای هر عضو در `probability_sum` اجرا می‌شود. در هر مرحله، اگر عدد تصادفی `rand` کمتر یا مساوی مقدار تجمعی بعدی (`next`) باشد، متغیر `o_idx` برابر با `i+1` قرار می‌گیرد و حلقه متوقف می‌شود.

در نهایت، مقدار `o_idx` که شماره عضو انتخاب شده است، برگردانده می‌شود.

به طور خلاصه، تابع `Roulette_wheel_selection` برای انتخاب یک عضو از مجموعه با استفاده از روش چرخه رولت استفاده می‌شود. این تابع ابتدا مجموع احتمالات تجمعی را محاسبه می‌کند و سپس با تولید یک عدد تصادفی بین ۰ و ۱، عضو متناظر با این عدد را انتخاب و برمی‌گرداند.

در قسمت بعدی از کد مقاله مربوط به عملیات تولید فرزند در الگوریتم ژنتیک است. در این قسمت، توابعی برای انجام عملیات جابجایی در ژنوم فرزندان تعریف شده‌اند. همچنین تابع `Crossover` نیز برای انتخاب نوع عملیات جابجایی بین فرزندان والدین به وسیله یک نمادگذاری switch-case تعریف شده است.

در ابتدا، تابع `initOSP(Q)` تعریف شده است که یک لیست از طول `Q` ایجاد کرده و هر عضو آن را برابر با ۱ به تقسیم بر `Q` قرار می‌دهد. این لیست مربوط به توزیع اولیه احتمالات استفاده در روش چرخه رولت است.

سپس توابع ``one_point_crossover`، `two_point_crossover`، `uniform_crossover`، `shuffle_crossover`` و `reduced_surrogate_crossover` تعریف شده‌اند که هر کدام یک نوع جابجایی مختلف را بین دو فرزند انجام می‌دهند. توابع `one_point_crossover` و `two_point_crossover` از روش‌های جابجایی با نقاط تقاطع یک‌تایی و دوتایی استفاده می‌کنند. تابع `uniform_crossover` از جابجایی تصادفی استفاده می‌کند و تابع `shuffle_crossover` از جابجایی با

مخلوط کردن دو فرزند استفاده می‌کند. در نهایت، تابع `reduced_surrogate_crossover` از جابجایی با استفاده از نقطه‌های مجاز تقاطع استفاده می‌کند.

در تابع `Crossover`، ابتدا با استفاده از یک عدد تصادفی، تصمیم می‌گیریم که آیا عملیات جابجایی بین فرزندان والدین انجام شود یا نه. اگر عدد تصادفی کوچکتر از نرخ جابجایی (`c_rate`) باشد، یکی از نوع‌های جابجایی انتخاب شده و براساس `idx`، تابع مربوطه صدا زده می‌شود و فرزندان جدید تولید می‌شوند. در غیر این صورت، فقط فرزندان والدین به عنوان خروجی برگردانده می‌شوند.

به طور خلاصه، این قسمت از کد مقاله، توابع مربوط به جابجایی در الگوریتم ژنتیک را شامل می‌شود. این توابع شامل انواع مختلف جابجایی هستند و تابع `Crossover` براساس یک عدد تصادفی و نرخ جابجایی، نوع جابجایی را انتخاب و اجرا می‌کند.

این قسمت از کد مقاله مربوط به بروزرسانی توزیع احتمالات محیطی (OSP) است. تابع `Update_OSP` با دریافت دو ماتریس `RD` و `PN` و یک عدد صحیح `Q`، توزیع احتمالات جدید را محاسبه و برمی‌گرداند.

در ابتدا، یک لیست خالی با نام `lst` ایجاد می‌شود. سپس با استفاده از حلقه `for` که بر روی اعداد صفر تا `Q-1` اجرا می‌شود، محاسباتی برای هر ستون (`q`) انجام می‌شود.

در هر مرحله، مجموع عناصر ستون `q` در ماتریس `RD` به وسیله تابع `sum` محاسبه می‌شود و در `S1_q` ذخیره می‌شود. همچنین مجموع عناصر ستون `q` در ماتریس `PN` را با استفاده از تابع `sum` محاسبه کرده و در `S2_q` ذخیره می‌کنیم.

سپس با استفاده از یک شرط، مقدار `S3_q` محاسبه می‌شود. اگر `S1_q` برابر با صفر باشد، `S3_q` برابر با ۰,۰۰۰۱ قرار می‌گیرد؛ در غیر این صورت، برابر با `S1_q` است. سپس `S4_q` محاسبه می‌شود، که نسبت `S1_q` به `(S2_q + S3_q)` است. مقادیر `S4_q` برای هر `q` در لیست `lst` ذخیره می‌شود.

در نهایت، لیست `lst` به عنوان توزیع احتمالات جدید `P_q` با تقسیم هر عضو به مجموع تمامی عناصر لیست (تابع `sum(lst)`) بازنویسی می‌شود.

به طور خلاصه، این قسمت از کد مقاله، تابع `Update_OSP` را شامل می‌شود که با استفاده از ماتریس‌های `RD` و `PN` و تعداد `Q`، توزیع احتمالات جدید را براساس محاسباتی از مقادیر ماتریس‌ها محاسبه می‌کند و برمی‌گرداند.

این قسمت از کد مقاله مربوط به ارزیابی تابع سلامت (fitness) برای یک جمعیت ژنتیکی است. تابع `fitness_evaluation` با دریافت یک عضو (ind) از جمعیت، تابع سلامت را محاسبه کرده و نتیجه را برمی‌گرداند.

در ابتدا، به عضو `ind` دسترسی داریم تا به کروموزوم آن (`ind.chromosome`) دسترسی پیدا کنیم. این کروموزوم برای تعیین ویژگی‌های استفاده شده در مدل‌سازی استفاده می‌شود.

سپس، با استفاده از لیست فهرستی (`cl`)، شاخص‌هایی که در کروموزوم برابر با 1 هستند را استخراج می‌کنیم و به لیست `cl` اضافه می‌کنیم. همچنین، یک شاخص دیگر با مقدار `df.shape[1]-1` به لیست `cl` اضافه می‌شود.

سپس، ماتریس داده (`df`) بر اساس شاخص‌های موجود در `cl`، به عنوان ماتریس داده جدید (`data`) انتخاب می‌شود.

سپس، مجموعه‌های آموزش و آزمون (`train_v` و `test_v`) به صورت خالی تعریف می‌شوند.

با استفاده از توابع `iloc` و `np.floor`، ماتریس داده به سه قسمت تقسیم می‌شود و هر قسمت به ترتیب به `folds` اضافه می‌شود. سپس، برای هر زوج از مجموعه‌های آموزش و آزمون، مراحل آموزش و آزمون مدل انجام می‌شود. ابتدا، داده‌های وابسته (`y_train` و `y_test`) جدا می‌شوند و سپس داده‌های مستقل (`X_train` و `X_test`) استخراج می‌شوند. سپس، داده‌های مستقل نرمال‌سازی می‌شوند و مدل KNN با تعداد همسایگان ۳ ساخته و روی داده‌های آموزش آموزش داده می‌شود. سپس، پیش‌بینی مدل بر روی داده‌های آزمون انجام شده و تعداد پیش‌بینی‌های اشتباه محاسبه می‌شود. در نهایت، نسبت تعداد پیش‌بینی‌های اشتباه به تعداد کل نمونه‌ها در داده‌های آموزش و آزمون محاسبه شده و به لیست `fold_result` اضافه می‌شود.

در پایان، میانگین ارزش خطا برای هر یک از مجموعه‌های آموزش و آزمون در `fold_result` محاسبه شده و در `f1` ذخیره می‌شود و همچنین، تعداد ویژگی‌های فعال در کروموزوم (`ind.chromosome`) را با استفاده از تابع `sum` محاسبه می‌کنیم و در `f2` ذخیره می‌کنیم. در نهایت، زوج `(f1, f2)` به عنوان نتیجه تابع سلامت برای عضو `ind` برگردانده می‌شود.

به طور خلاصه، تابع `fitness_evaluation` برای هر عضو در جمعیت، ماتریس داده را بر اساس کروموزوم تعریف شده در `ind` انتخاب کرده و مدل KNN را با استفاده از داده‌های آموزش و آزمون آموزش داده و عملکرد مدل را ارزیابی می‌کند. سپس نتیجه ارزیابی را برمی‌گرداند که شامل میانگین ارزش خطا برای هر زوج داده‌های آموزش و آزمون و تعداد ویژگی‌های فعال در کروموزوم است.

این قسمت از کد مقاله مربوط به عملیات جهش یکنواخت (uniform mutation) در جمعیت ژنتیکی است. تابع `uniform_mutation` با دریافت جمعیت `p` و نرخ جهش `m_rate`، جمعیت جدیدی را با اعمال جهش یکنواخت بر روی اعضای جمعیت اصلی ایجاد می‌کند و برمی‌گرداند.

ابتدا، یک لیست خالی به نام `lst` تعریف می‌شود. سپس، برای هر عضو `ind` در جمعیت `p`، یک لیست خالی به نام `tmp` ایجاد می‌شود.

سپس، برای هر عنصر در کروموزوم `ind.chromosome`، اگر یک عدد تصادفی کمتر از نرخ جهش (`m_rate`) باشد، عنصر جدید را با عکس بیت عنصر موجود در کروموزوم تولید می‌کنیم (`ind.chromosome[i] ^ True`) و در `tmp` اضافه

می‌کنیم. در غیر این صورت، عنصر موجود را بدون تغییر در `tmp` اضافه می‌کنیم. لیست `tmp` برای هر عضو جدید در لیست `lst` اضافه می‌شود.

سپس، با استفاده از لیست `lst`، لیست جدیدی از عناصر `Individual` ایجاد می‌شود. برای هر عضو در `lst`، یک عضو جدید `Individual` با کروموزوم متناظر ایجاد شده و به لیست `chromosome_list` اضافه می‌شود.

در نهایت، لیست `chromosome_list` حاوی جمعیت جدید با اعمال جهش یکنواخت بر روی اعضای جمعیت اصلی برگردانده می‌شود.

به طور خلاصه، تابع `uniform_mutation` با استفاده از نرخ جهش `m_rate`، برای هر عضو در جمعیت اصلی، جهش یکنواخت را انجام می‌دهد. در این جهش، هر بیت در کروموزوم با احتمال `m_rate` تغییر می‌کند و جمعیت جدید با اعمال جهش بر روی اعضای جمعیت اصلی ایجاد می‌شود.

این قسمت از کد مقاله شامل دو تابع است: `dominanceComparison` و `CreditAssignment`.

تابع `dominanceComparison` با دریافت یک جمعیت `p`، اعضایی را که توسط سایر اعضا در جمعیت (non-dominated) و اعضایی را که توسط سایر اعضا (dominated)، جدا می‌کند. برای هر عضو `j` در جمعیت `p`، اعضای دیگر جمعیت

را بررسی می‌کند و اگر `j` توسط همه اعضا در جمعیت non-dominated باشد (برای تمامی هدف‌ها)، آنگاه `j` به لیست `p_non_dominated` اضافه می‌شود. اگر `j` توسط همه اعضا در جمعیت dominated شده باشد (برای تمامی هدف‌ها)، آنگاه `j` به لیست `p_dominated` اضافه می‌شود. در نهایت، دو لیست `p_dominated` و `p_non_dominated` برگردانده می‌شوند.

تابع `CreditAssignment` با دریافت دو جمعیت `P` و `R`، نرخ پاداش (`nReward`) و نرخ تنبیه (`nPenalty`) را محاسبه می‌کند. ابتدا، نرخ پاداش و تنبیه را از آرایه‌های `np` و `nr` کپی می‌کند.

سپس، برای هر عضو `ind` در جمعیت `R`، اگر تعداد هدف‌های `ind` برابر 0 باشد، تابع `fitness_evaluation` را بر روی `ind` اجرا کرده و مقادیر هدف‌ها را محاسبه می‌کند.

سپس، جمعیت `P` را بررسی می‌کند. اگر تعداد اعضای `P` که تسلیم شده‌اند (`P_d`) برابر با صفر نباشد، آنگاه برای دو عضو اول در جمعیت `R`، بررسی می‌کند که آیا `P_d[0]` از `R[i]` ؛ dominated شده است یا خیر. اگر dominated شده باشد، نرخ تنبیه مربوط به این عملگر (`operator_idx-1`) افزایش می‌یابد، در غیر این صورت نرخ پاداش مربوط به این عملگر افزایش می‌یابد.

در غیر این صورت، برای دو عضو اول در جمعیت `R`، بررسی می‌کند که آیا `P[0]` و `P[1]` از `R[i]` تسلیم نشده‌اند یا خیر. اگر هیچکدام از آن‌ها dominated نشده باشند، نرخ پاداش مربوط به این عملگر افزایش می‌یابد، در غیر این صورت نرخ تنبیه مربوط به این عملگر افزایش میشود.

در نهایت، آرایه‌های نرخ پاداش (`nReward`) و نرخ تنبیه (`nPenalty`) به عنوان خروجی تابع برگردانده می‌شوند.

به طور خلاصه، تابع `CreditAssignment` با دریافت دو جمعیت `P` و `R`، اعضایی که در جمعیت `P` تسلیم نشده‌اند و تسلیم شده‌اند را تشخیص می‌دهد و بر اساس این تشخیص، نرخ پاداش و تنبیه را محاسبه می‌کند. این تابع برای هر عضو در جمعیت `R`، محاسبات لازم را انجام می‌دهد و مقادیر نرخ پاداش و تنبیه را برمی‌گرداند.

این قسمت از کد مقاله شامل تابع `MOBGA_AOS` است که الگوریتم بهینه‌سازی چند هدفه بر اساس الگوریتم ژنتیک چندهدفه با تخصیص اعتبار (MOBGA-AOS) را پیاده‌سازی می‌کند. الگوریتم به شکل زیر است:

1. متغیرهای `k`، `nFE`، `p_new`، `f0` را مقداردهی اولیه کنید.

2. جمعیت اولیه `p` را با ایجاد `N` عضو، هر کدام با `D` بیت تصادفی، ایجاد کنید.

3. برای هر عضو `i` در جمعیت `p`، تابع `fitness_evaluation` را اجرا کرده و مقادیر هدف‌ها را محاسبه کنید.

4. آرایه‌های `nReward` و `nPenalty` را با طول `Q` و مقدار صفر مقداردهی اولیه کنید. همچنین، آرایه‌های `RD` و `PN` را با ابعاد `LP×Q` و مقدار صفر مقداردهی اولیه کنید.

5. تا زمانی که تعداد مقادیر تابع هدف محاسبه شده (`nFE`) کمتر از حداکثر تعداد مقادیر تابع هدف مجاز (`maxFEs`) باشد، مراحل زیر را تکرار کنید:

- متغیر `p_new` را خالی کنید.

- برای `i` برابر با نصف تعداد جمعیت `N`، مراحل زیر را انجام دهید:

- میزان مساحت تحت منحنی (OSP) را با تابع `initOSP` محاسبه کنید.

- یک اندیس تابع ترکیب را توسط انتخاب چرخه‌ای رولت (Roulette wheel selection) با استفاده از مساحت تحت منحنی (OSP) و به تعداد تابع ترکیب `Q` انتخاب کنید.

- دو عضو تصادفی از جمعیت `p` را به عنوان والدین `P_p` انتخاب کنید.

- عملگر ترکیب را با تابع `Crossover` روی `P_p`، با استفاده از اندیس تابع ترکیب و نرخ ترکیب `crossover_rate`، اعمال کنید.

- عملگر جهش را با تابع `uniform_mutation` روی `P_c` و با استفاده از نرخ جهش `mutation_rate` اعمال کنید.

- برای هر عضو در `P_c`، تا زمانی که همه بیت‌های کروموزوم برابر 0 نباشند، عملگر جهش را اعمال کنید.

- تعداد مقادیر تابع هدف (`nFE`) را به ازای هر دو عضو در `P_c` به 2 افزایش دهید.

- با استفاده از تابع `CreditAssignment`، نرخ پاداش و تنبیه را بدر ادامه توضیح قسمت قبلی از کد مقاله را ادامه می‌دهیم:

- با استفاده از تابع `CreditAssignment`، نرخ پاداش و تنبیه را برای `P_p` و `P_c` محاسبه کنید و در آرایه‌های `nReward` و `nPenalty` به روز کنید.

- عضوهای جدید تولید شده `P_c` را به `p_new` اضافه کنید.

- مقادیر تابع هدف و تنبیه‌های جدید را در آرایه‌های `RD` و `PN` به عنوان سطر `k`-ام ذخیره کنید.

- اگر `k` برابر با `LP` شد، مساحت تحت منحنی (OSP) را با استفاده از تابع `Update_OSP` و با استفاده از آرایه‌های `RD` و `PN` به روز کنید و `k` را صفر کنید.

- جمعیت جدید `R` را با ادغام جمعیت اولیه `p` و جمعیت جدید `p_new` به دست آورید.

- با استفاده از تابع `non_dominated_sorting`، عضوهای غیرمهیا به همدیگر را مرتب کنید و پارتو فرانت را بدست آورید.

- با استفاده از تابع `environmental_selection`، جمعیت جدید را از پارتو فرانت انتخاب کنید و به عنوان جمعیت `p` به روز رسانی کنید.

- پارتو فرانت اولیه (`f0`) را به روز کنید.

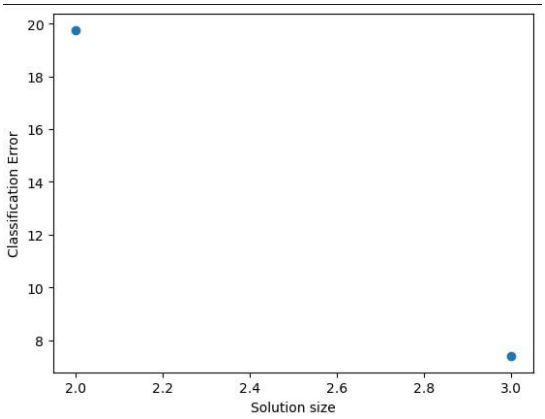
6. مجموعه زیرمجموعه ویژگی بهینه (`optimal_feature_subset`) را با تعداد اعضای پارتو فرانت بدست آورید.

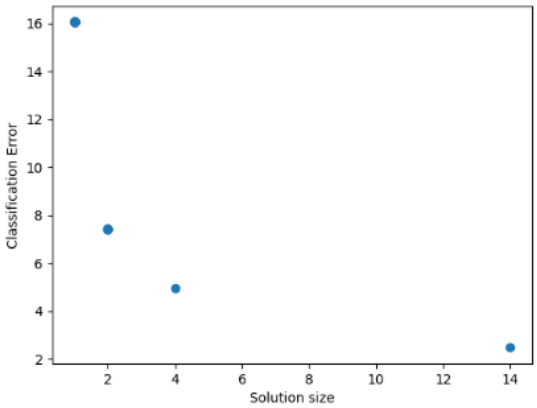
7. نمودار پراکندگی از مقادیر تابع هدف برای پارتو فرانت (`front_pareto`) رسم کنید.

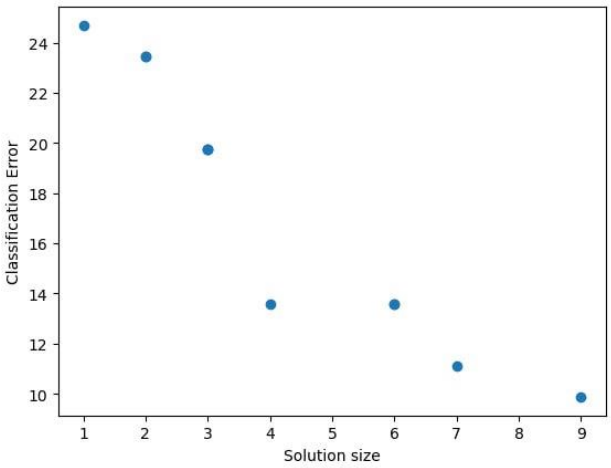
در این کد، الگوریتم MOBGA-AOS با استفاده از الگوریتم ژنتیک چندهدفه بهینه‌سازی می‌شود. جمعیت اولیه تصادفی ایجاد می‌شود و نرخ ترکیب و جهش نیز مقداردهی اولیه می‌شود. سپس الگوریتم به صورت تکراری عملیات ترکیب و جهش را انجام داده و جمعیت را به روز می‌کند تا مقادیر تابع هدف مجاز را برساند. در نهایت، پارتو فرانت و نمودار پراکندگی مقادیر تابع هدف رسم می‌شوند.

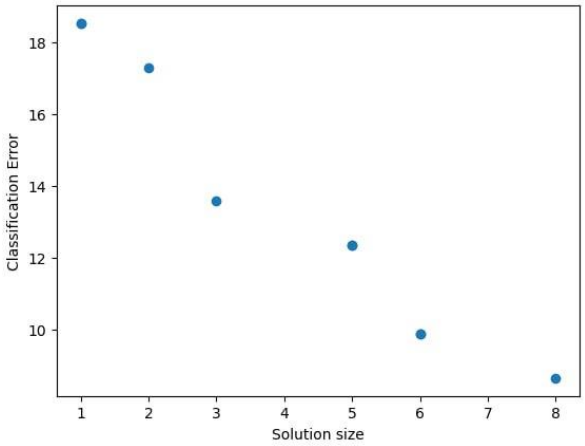
خروجی های کد به شکل زیر است :

dataset 2

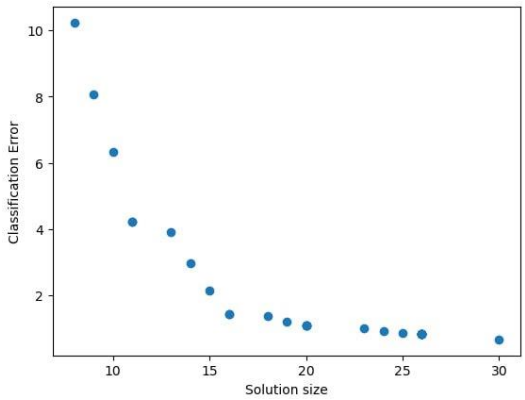


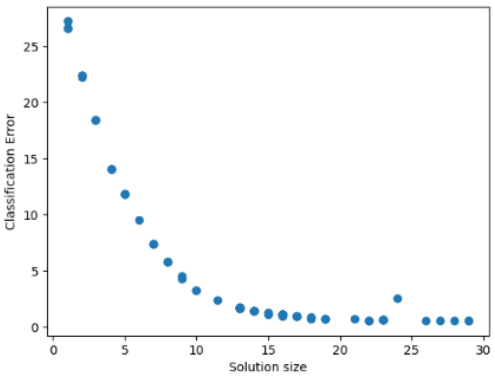


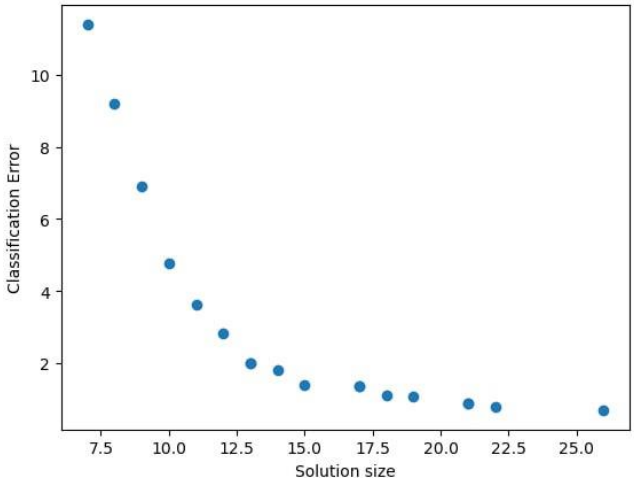




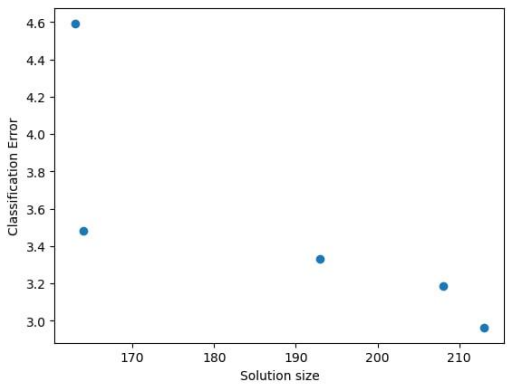
dataset 4

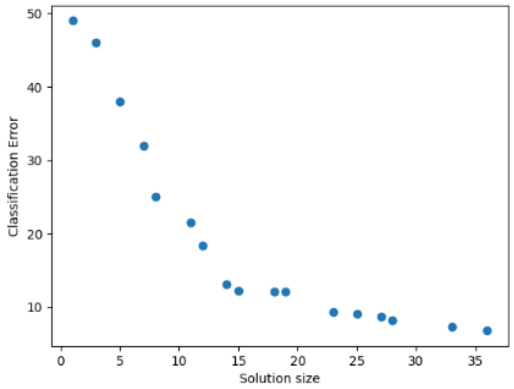


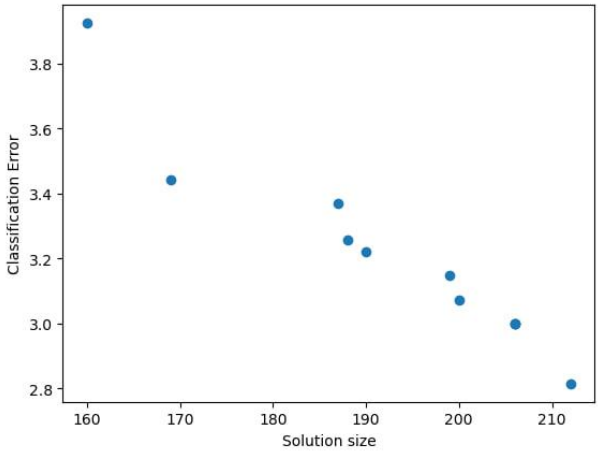



Classification Error
Solution size
10
8
6
4
2
7.5
10.0
12.5
15.0
17.5
20.0
22.5
25.0

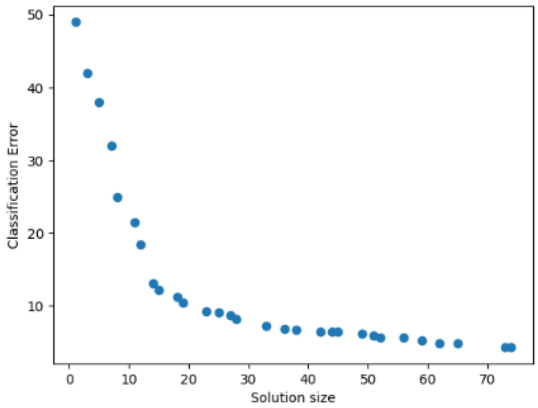
dataset 7


Classification Error
Solution size
4.6
4.4
4.2
4.0
3.8
3.6
3.4
3.2
3.0
170
180
190
200
210

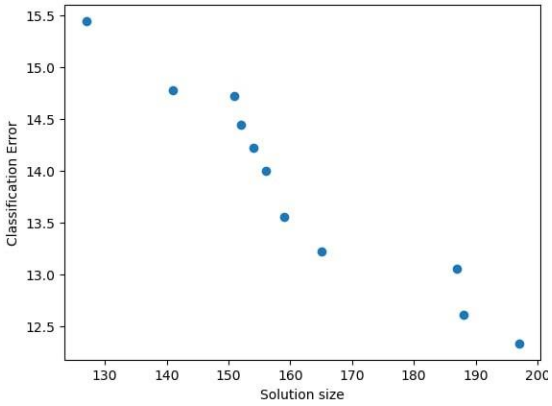

Classification Error
Solution size
50
40
30
20
10
0
5
10
15
20
25
30
35

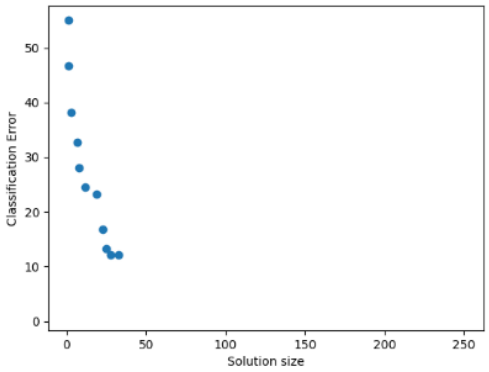

Classification Error
Solution size
3.8
3.6
3.4
3.2
3.0
2.8
160
170
180
190
200
210

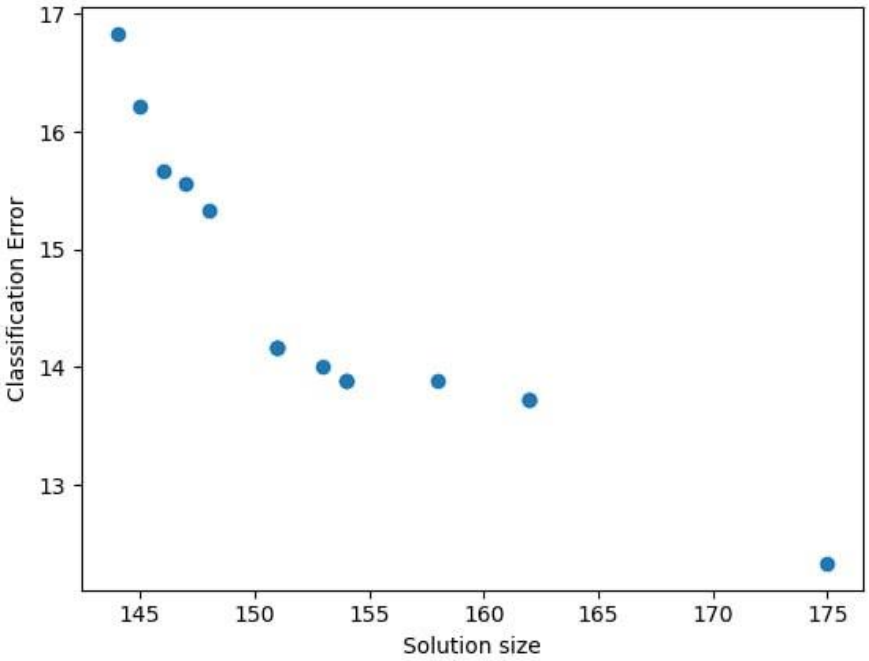
dataset 10- max-F - 80000 - run 1



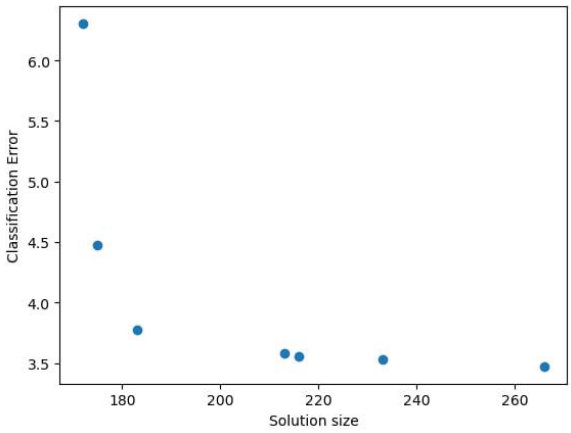
dataset 5

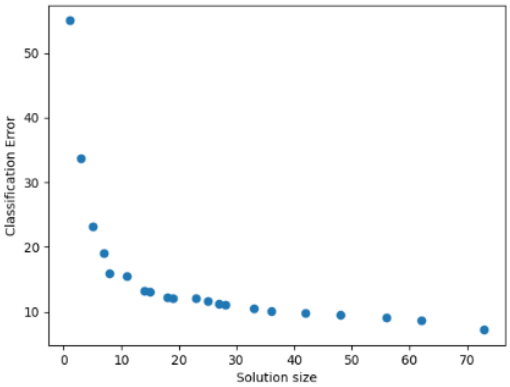


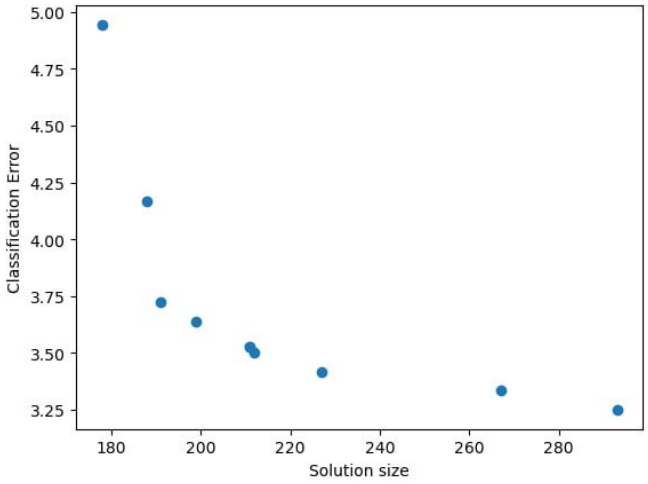



Classification Error
Solution size
17
16
15
14
13
145
150
155
160
165
170
175

dataset 8


Classification Error
Solution size
6.0
5.5
5.0
4.5
4.0
3.5
180
200
220
240
260


Classification Error
Solution size
50
40
30
20
10
0
10
20
30
40
50
60
70


5.00
4.75
4.50
4.25
4.00
3.75
3.50
3.25
Classification Error
180
200
220
240
260
280
Solution size