Supplementary Material for "DiSciPLE: Learning Interpretable Programs for Scientific Visual Discovery"

A. Additional Results

A.1. Interpretability with DiSciPLE

We qualitatively show an example run of our AGB estimator over a large area wherever ground truth is available. Fig. 1 shows such a run of our discovered function near Massachusetts. Due to the interpretable nature of our function, we can visualize intermediate features used by our method and understand their influence on the final prediction.

A.2. Performance with iterations

To study if $T = 15$ generations of programs are enough to get good solutions, we plot the R2-score with respect to the number of generations on different problems in Fig. 2. The R2-score is measured on the training set for the best program till the generation iteration.

We can observe two things. First, the programs improve over generations, proving that zero-shot inference is not enough to retrieve good programs and evolution with evaluation is necessary. Second, the programs start to converge in terms of performance by iteration 10, hence $T > 15$ is not needed.

A.3. More qualitative comparisons

In Fig. 3 we show more qualitative examples comparing population density predictions of DiSciPLE and the baselines to the true population density. Again, It is abundantly clear that DiSciPLE outperforms the baselines in modeling the fine-grained population changes in unseen regions.

B. DiSciPLE on More Challenging Demography indicators

As mentioned in the main paper we evaluated DiSciPLE on a set of more challenging indicators. Tab. 1 enumerates demographic indicators with their ACS Community Survey Code (left) and detailed description of their meaning.

In the main paper, we only reported the average performance of these models. In Fig. 4 we also report the performance of DiSciPLE compared to the baselines on each indicator individually. On many challenging indicators, where DiSciPLE cannot find a good program such as SE_A02001_002 and SE_A02001_003, the performance of all models is similar to the mean. However in many other cases such as SE_A00003_002 or SE_A01004_001, DiSciPLE can find a program that is better than the mean prediction performance.

C. DiSciPLE on Non-visual domains

While in this paper we primarily apply DiSciPLE to a set of visual benchmarks, high-dimensional problems also exist in other scientific applications. One such example is time-series forecasting. As a proof-of-concept, we apply DiSciPLE on one such climate-related time-series problem called Contiguous Solar Induced Chlorophyll Fluorescence forecasting.

C.1. Contiguous Solar Induced Chlorophyll Fluorescence (CSIF)

Observation Dataset: CSIF is used to monitor the photosynthetic activity of terrestrial ecosystems. Accurate forecasting of CSIF is important in many applications such as understanding the growing seasons of crops or seasonal changes to forests such as the arrival of fall colors [10]. The goal is to forecast CSIF values, by observing past CSIF and environmental values. Unlike other problems, which could make use of spatial information and thus require satellite images, CSIF forecasting is done without using satellite images. On the other hand, since this is a forecasting problem, it uses past CSIF values as well as

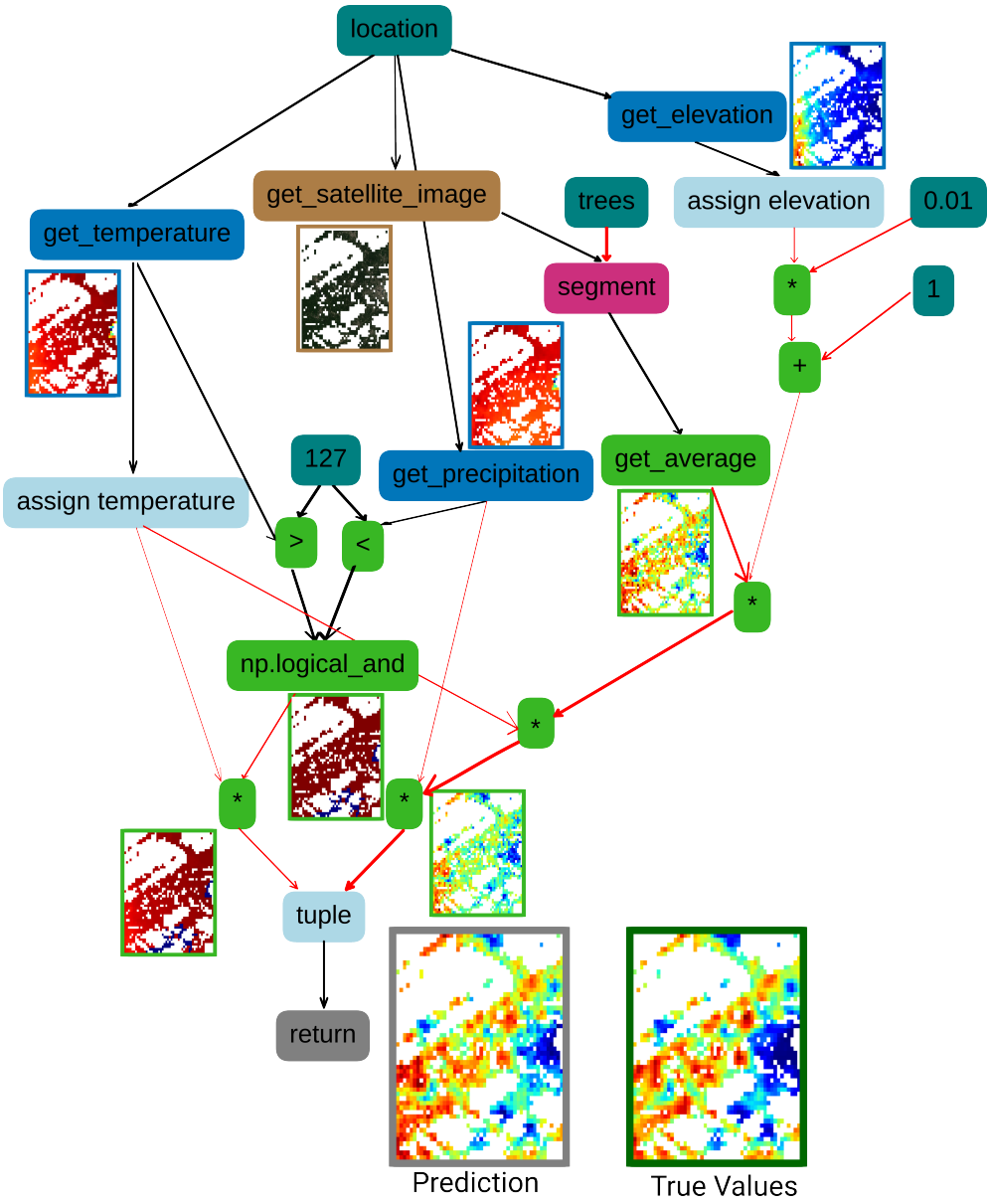

Figure 1. **Interpretable programs with DiSciPLE.** Visualization of intermediate features generated by the program for AGB estimation. The program allows us to look at intermediate features that the model is looking to make predictions, allowing an expert a deeper understanding.

past and current values of environmental variables. The observation variables are an input location and output CSIF values. The data comes from ERA5 climate data store [2].

Metric and Primitives: We use L2 error for each location as the evaluation metric, along with the mathematical and logical operations. The primitive allows obtaining present environmental variables as well as a time series of past environmental variables such as minimum and maximum temperature of past months, or soil moisture index (see Appendix I.3).

Baselines: For deep baselines we use LSTMs (small) [3] and time series transformers (large) [9]. For the concept bottleneck baseline, we learn a linear classifier over the present environment variable and the mean of past environment variables along with CSIF.

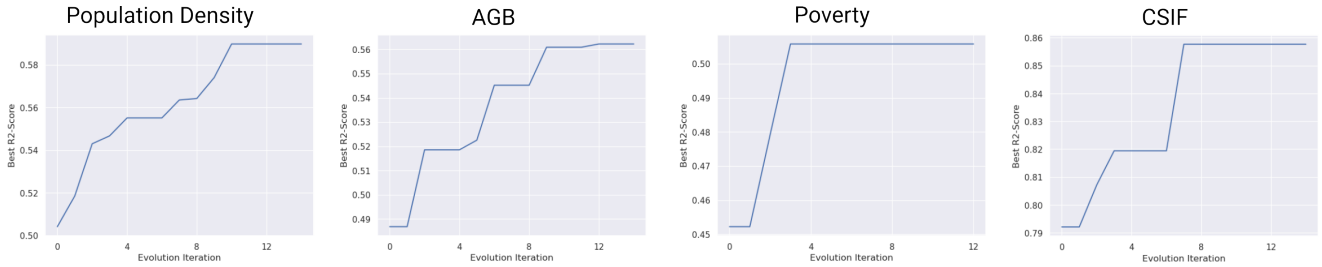

Figure 2. Performance of the best programs generated by our method in terms of R2-Score with respect to the number of evolution generations (iterations). For most of the problems, the programs tend to improve for the first 10 iterations but converge to a good solution after that.

Table 1. List of 34 demography indicators.

ACS Survey Code	Description
SE_A00003_002	amount of Total Area that is Land Area
SE_A00003_003	amount of Total Area that is Water Area
SE_A02001_002	amount of Total Population that is Male
SE_A02001_003	amount of Total Population that is Female
SE_B01001_002	amount of Total Population that is Under 18 Years
SE_B01001_003	amount of Total Population that is 18 to 34 Years
SE_B01001_004	amount of Total Population that is 35 to 64 Years
SE_B01001_005	amount of Total Population that is 65 and Over
SE_A01004_001	Median Age
SE_A10008_002	amount of Households that are Family Households
SE_A10008_007	amount of Households that are Nonfamily Households
SE_A12001_002	amount of Population 25 Years and Over that is Less than High School
SE_A12001_003	amount of Population 25 Years and Over that is High School Graduate or More (Includes Equivalency)
SE_A12001_005	amount of Population 25 Years and Over that is Bachelor"s Degree or More
SE_A12001_006	amount of Population 25 Years and Over that is Master"s Degree or More
SE_A12003_002	amount of Civilian Population 16 to 19 Years that is Not High School Graduate, Not Enrolled (Dropped Out)
SE_A12003_003	amount of Civilian Population 16 to 19 Years that is High School Graduate, or Enrolled (In School)
SE_A17002_002	amount of Population 16 Years and Over that is In Labor Force
SE_A17002_007	amount of Population 16 Years and Over that is Not in Labor Force
SE_A14001_002	amount of Households earning less than $60,000
SE_A14001_012	amount of Households earning more than $60,000
SE_A14006_001	Median Household Income (In 2022 Inflation Adjusted Dollars)
SE_A10011_002	amount of Households that are With Earnings
SE_A10011_003	amount of Households that are No Earnings
SE_A10060_002	amount of Occupied Housing Units that are Owner Occupied
SE_A10060_003	amount of Occupied Housing Units that are Renter Occupied
SE_A09005_002	amount of Workers 16 Years and Over that are Car, Truck, or Van
SE_A09005_003	amount of Workers 16 Years and Over that are Public Transportation (Includes Taxicab)
SE_A09005_006	amount of Workers 16 Years and Over that are Walked
SE_A10030_002	amount of Occupied Housing Units that have no vehicle available
SE_A10030_003	amount of Occupied Housing Units that have 1 vehicle available
SE_A10030_004	amount of Occupied Housing Units that have more than 1 vehicle available
SE_A10066_002	amount of Occupied Housing Units that are less than 4-Person Household
SE_A10066_005	amount of Occupied Housing Units that are more than 3-Person Household

C.2. Results on time-series forecasting

We apply DiSciPLE on this CSIF forecasting task. Tab. 2 shows the performance of our method compared to the baseline on CSIF forecasting on both in-distribution and out-of-distribution data. The function discovered by DiSciPLE is second best behind a deep model on in-distribution data. DiSciPLE can generalize to OOD data significantly better than the deep

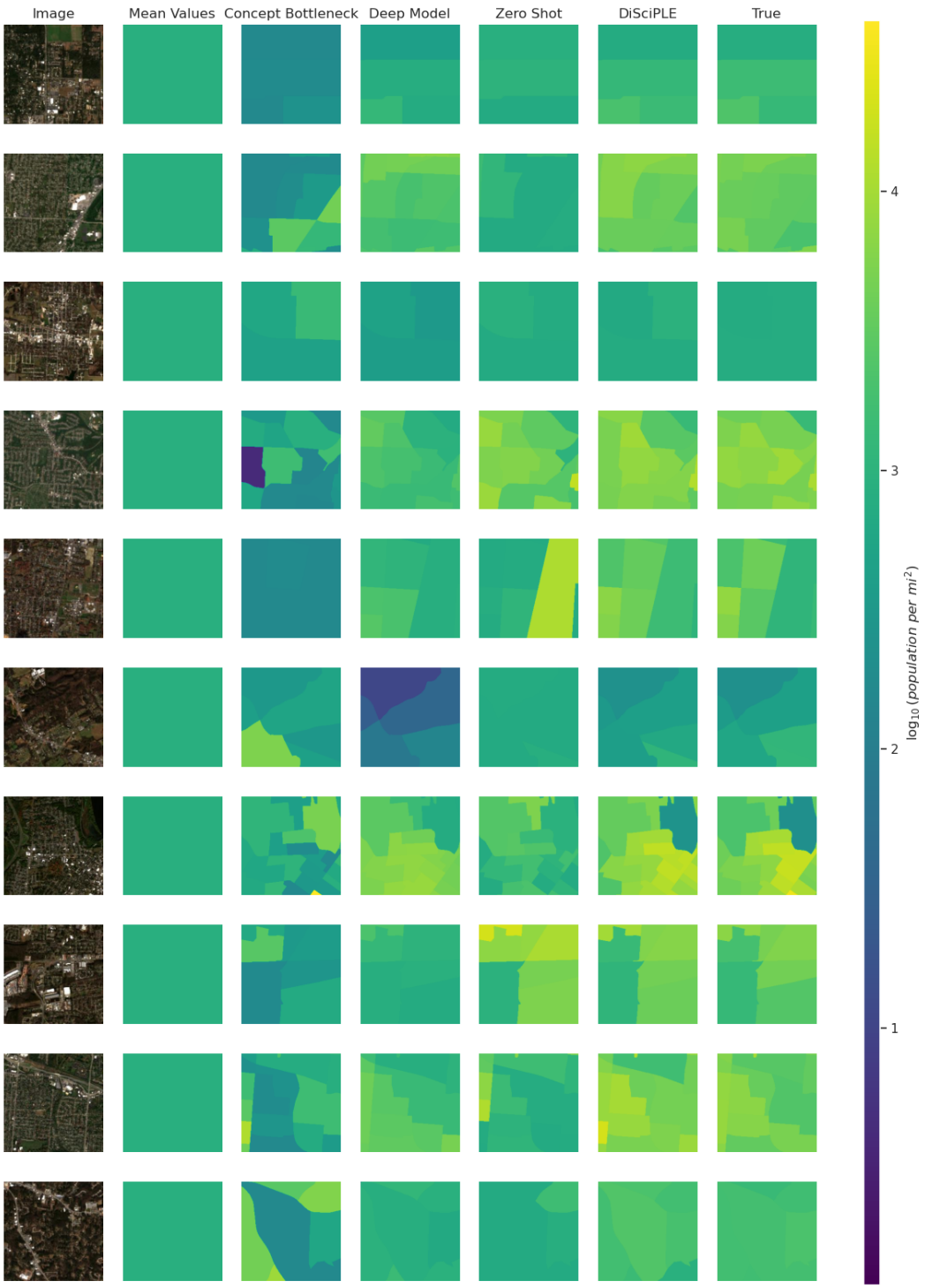

Figure 3. More examples of predictions from DiSciPLE and other models on population density estimation task on unseen locations. The maps display population density as the base-10 log of people per square mile.

models. Fig. 5 shows the program generated DiSciPLEon this task. This shows that DiSciPLE can be applied to obtain interpretable and accurate programs for other high-dimensional non-visual problems such as time series forecasting.

Figure 4. Performance of DiSciPLE and baselines individually on the 34 demography indicators.

D. Prompts for Crossover, Mutation and Others

For crossover, we take two programs and their corresponding scores. We use the following prompt for crossover:

Table 2. Performance of our programs on in-distribution (left) and out-of-distribution (right) observations on CSIF time series forecasting. DiSciPLE produces better programs (**red** is best and *blue* is second best).

	In Distribution		OOD	
	L1	RMSE	L1	RMSE
Mean	0.2521	0.3050	0.2393	0.3018
Concept Bottleneck	0.1474	0.1835	0.1565	0.2029
Deep Model - Small	**0.0503**	**0.0727**	*0.1061*	*0.1614*
Deep Model - Large	0.1487	0.1895	0.1815	0.2435
Zero-shot	0.2134	0.2610	0.2190	0.2814
Ours	*0.0902*	*0.1260*	**0.0882**	**0.1256**

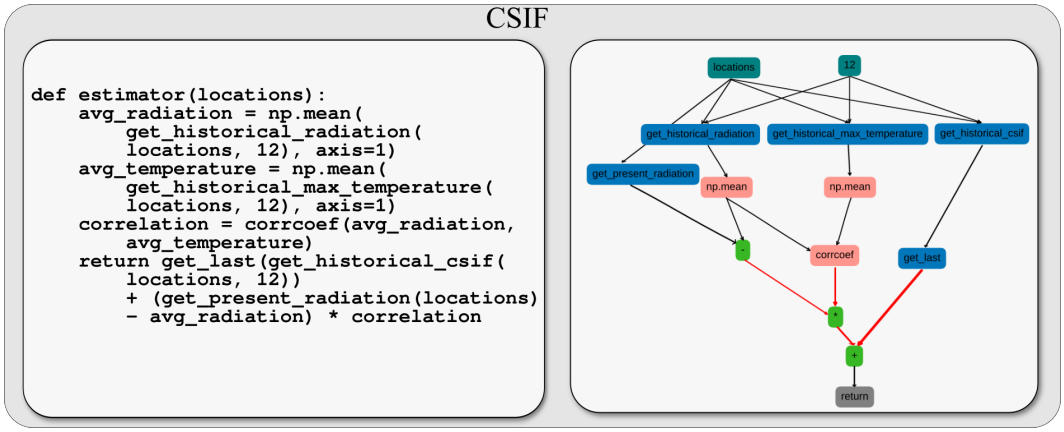

Figure 5. Program generated by DiSciPLE for CSIF forecasting task

$\{\{h_1\}\}$
`This program has a score of` $\{\{s(h_1)\}\}$.
$\{\{h_2\}\}$
`This program has a score of` $\{\{s(h_2)\}\}$.
`Can you write a function that gives a higher score? Feel free to combine elements that worked from both programs. Only give me the code.`

And similarly, we use the following prompt for random mutation.

$\{\{h\}\}$
`This program has a score of` $\{\{s(h)\}\}$.
`Can you edit this code to write a better function for the problem? Only give me code.`

For critic, we first extract a list of categories the existing program performs worse on C. We then list these categories with the following prompt.

`The program you generated is bad when the satellite image contains the following categories: {{C}}. Generate a program that also works on these categories.`

For the "no problem context" ablation we replace the problem objective prompt with a generic prompt.

`Write a function that takes in an image and returns a useful map using it.`

For the "no common-sense" ablation we replace the API string and the name of the functions. For example, `elementwise_max` is replaced like this:

```
def elementwise_max(matrix1, matrix2):
    """
    Compute the element-wise maximum of two matrices.

```

```
    Parameters:
        matrix1 (numpy.ndarray): First input matrix.
        matrix2 (numpy.ndarray): Second input matrix.

    Returns:
        numpy.ndarray: Element-wise maximum of the input matrices.
    """


# replace with a function with arbitrary name and no description.
def deep_shade(input1, input2):
    """
    Parameters:
        input1 (numpy.ndarray): First input.
        input2 (numpy.ndarray): Second input.
    Returns:
        numpy.ndarray
    """
```

E. More Examples

E.1. Illustration of Simplifcation

For the following program, we show the steps of simplification. **Program after crossover**

```
def estimator(im):
    building_mask = segment(im, "residential building")
    nr_mask = segment(im, "non-residential buildings")
    vegetation_mask = segment(im, "forest")
    water_mask = segment(im, "lake")
    road_mask = segment(im, "highway")

    building_distance = min_pixel_distance_to_mask(building_mask)
    nr_distance = min_pixel_distance_to_mask(nr_mask)
    vegetation_distance = min_pixel_distance_to_mask(vegetation_mask)
    water_distance = min_pixel_distance_to_mask(water_mask)
    road_distance = min_pixel_distance_to_mask(road_mask)

    return building_distance, nr_distance, vegetation_distance, road_distance
```

In the first step, as can be seen in Fig. 6 (top-left graph), at the bottom right there is a leaf node that is not a return node. We remove that node and other leaf nodes recursively resulting in a graph-like top-right.

Using regression weights our method figures out that the leftmost branch or `building_distance` is not a useful value to be returned. So in the third step, we remove that and recursively all the leaf nodes.

E.2. More Examples of Crossover

Parent 1

```
def estimator(location):
    images = get_satellite_image(location)
    temperature = get_temperature(location)
    precipitation = get_precipitation(location)
    nightlight = get_nightlight_intensity(location)
    return temperature, precipitation, elevation, nightlight
```

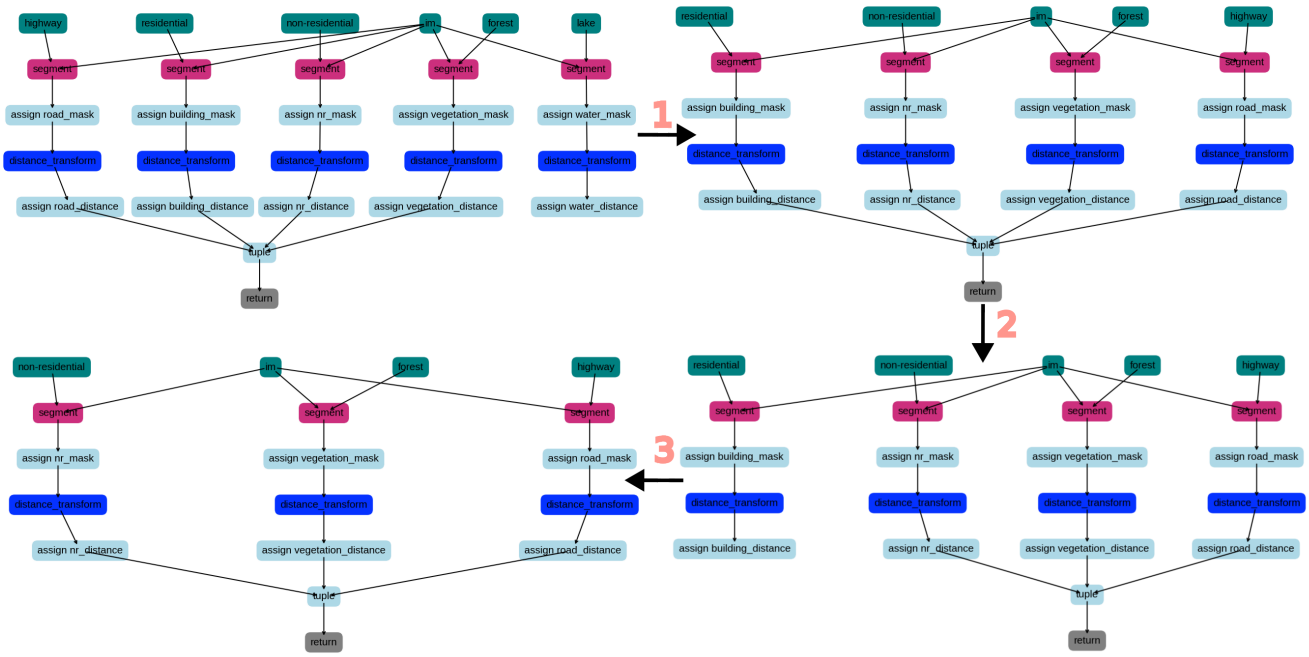

Figure 6. Process of simplification illustrated over a function.

Parent 2

```
def estimator(location):
    images = get_satellite_image(location)
    roads = segment(images, 'roads')
    avg_roads = get_average(roads)
    avg_nightlight = get_nightlight_intensity(location)
```

Program after crossover

```
def estimator(location):
    images = get_satellite_image(location)
    roads = segment(images, 'roads')
    avg_roads = get_average(roads)
    avg_nightlight = get_nightlight_intensity(location)
    avg_precipitation = get_precipitation(location)
    avg_temperature = get_temperature(location)
    return (avg_roads * avg_precipitation * avg_nightlight * avg_temperature)
```

Parent 1

```
def estimator(location):
    images = get_satellite_image(location)
    forests = segment(images, 'forests')
    avg_forests = get_average(forests)
    roads = segment(images, 'roads')
    avg_roads = get_average(roads)
    elevation = get_elevation(location)
    nightlight_intensity = get_nightlight_intensity(location)
    return avg_forests, avg_roads, 1 / (1 + elevation), 1 / (1 + nightlight_intensity)
```

Table 3. different LLMs

LLM	Test L2 log	Test L1 log	OOD L2 log	OOD L1 log	Program Error Percent
Qwen2.5-7b-instruct	0.2950	0.4039	0.4263	0.4716	44.3
llama-3-8b-instruct	0.2626	0.3842	0.3946	0.4514	56.5
llama-3.1-8b-instruct	0.2771	0.3872	0.3950	0.4518	49.2
llama-3.1-70b-instruct	0.2896	0.3958	0.4223	0.4663	40.2
Deep Model - Large	0.3974	0.4843	0.4460	0.5115	-

Parent 2

```python
def estimator(location):
    images = get_satellite_image(location)
    poverty_mask = segment(images, 'poverty')
    return get_average(poverty_mask)
```

Program after crossover

```python

def estimator(location):
    images = get_satellite_image(location)
    poverty_mask = segment(images, 'poverty')
    education_mask = segment(images, 'education')
    health_mask = segment(images, 'health')
    avg_poverty = get_average(poverty_mask)
    avg_education = get_average(education_mask)
    avg_health = get_average(health_mask)
    elevation = get_elevation(location)
    nightlight_intensity = get_nightlight_intensity(location)
    return avg_poverty, avg_education, avg_health, 1 / (1 + elevation), 1 / (1 +
    ↪  nightlight_intensity)
```

F. Additional Ablations

F.1. Using different LLMs

A key contribution of our work is to leverage the common-sense knowledge in LLMs to improve evolutionary search. So, it is natural to question whether (a) LLMs (with similar capacity) trained with a different large corpus of text would generate programs with different levels of reliability and (b) LLMs with larger capacity would produce more reliable programs. We answer these questions by testing recent LLMs: Qwen-2.5/7b [7], llama-3/8b, llama-3.1/8b, llama-3.1/70b. For ease of experimentation, we reduce the number of generations to 10 (as results in Appendix A.2 suggests) and the population size to 60. We report the results in Tab. 3. DiSciPLE works robustly with various LLMs and could generate more reliable models than the Deep Model. While the performance of the programs varies, we do not observe any discernible difference among the various programs.

Note that the main reason why there isn't a discernible difference in performance when using different LLMs is that, several components of DiSciPLE make up for the weaknesses of LLMs. For example, if an LLM is relatively bad at generating correct code, DiSciPLE rejects those solutions as we also have evaluation in the loop. Similarly, if an LLM produces verbose code, DiSciPLE has the ability to simplify and reduce the code to useful components. Therefore, as long as an LLM has good common sense ability and the ability to produce diverse programs, DiSciPLE can support any LLM.

While the final performance of functions discovered by all programs is similar across LLMs, language models with better coding ability lead to a lower percentage of buggy programs. As a result, the evolutionary search for working programs is faster for LLMs with less error rate. We also report these error percentages in Tab. 3.

We also tested DiSciPLE with GPT-3.5 and GPT-4o-mini, however, we observed that these models care too much about the objective prompt p_o, and as a result, the generated programs lack diversity. The lack of diversity prevents evolution from gaining momentum. In future, we plan to perform a better hyperparameter search to enable diverse program generation and as a result better evolution with GPT models.

F.2. Albation on noisy/unreliable primitives

To investigate how accurate/robust should the underlying black-box model be?, we corroded the OSM maps with a 3x3 convolution and ran DiSciPLE to generate a new program. With the corroded OSM maps, we observe an L2 log error of 0.3713 on the test set — a large degradation in performance compared to clean OSM maps (L2 log error: 0.2626).

G. Experimental Setup

G.1. More details on concept bottleneck baselines

For all the tasks in our benchmark, the bottleneck features are 42 categories of segments obtained from either OSM or GRAFT and the environmental variable. Moreover, for fairness, we also concatenate the environmental variables used by DiSciPLEas input to the model. So both DiSciPLEand concept bottleneck baselines get the same information.

For the CSIF task, the concept bottleneck features are the average of past CSIF and environmental variables and the current environmental variable.

Note that we call this method concept bottleneck due to its similarity to the original work [4] in terms of execution *i.e.* linear model on a feature set. However, the way we obtain concepts is much more similar to follow-up works [5, 8].

G.2. More details on deep models baseline

For spatial tasks, we use a ResNet-18 [1] based U-Net [6] (Deep Model-Large). Since our model also needs to be data-efficient, to prevent overfitting, we also try a smaller backbone of 4-layer fully convolutional network(Deep Model-Small).

We also tried larger ResNets and transformers, however, these models underfit and as a result, fail to generalize robustly to OOD as well as the in-distribution test set. We posit that this is due to the limited amount of training data (at most 4k observations).

For fair comparison, we also concatenate the environment variable in the intermediate layer of the deep model, so that they can make use of the same amount of information as DiSciPLE.

We also, compare a deep model baseline where we concatenate the RGB satellite images with the 42 binary segmentation masks. This 45-channel input is projected to 3 dimensions and passed through a ResNet and 4-layer models of the same size. The performance of this model was about the same as the RGB model, therefore we only report the RGB model's performance. However, this shows that with the same amount of input information DiSciPLEperforms better than the deep baselines.

H. More details

H.1. List of 42 land-use concepts

Table 4 show the list of 42 concepts extracted from OpenStreetMaps and also used in GRAFT to get partitions for critic.

tennis courts	skate park	american football field	swimming pool	cemetery	pond
golf course	roundabout	parking lot	supermarket	school	marina
baseball	waterfall	multi-storey parking garage	airport	beach	bridge
religious building	residential building	university building	office	farmland	warehouse
forest	lake	nature reserve	park	sandy area	soccer field
equestrian center	shooting range	non residential buildings	commercial area	garden	dam
railroad	highway	river or stream	wetland	ice-rink	coastline

Table 4. List of concepts extracted from OSM and also used via GRAFT for critic data stratification.

I. Problem Specific Primitive Description

I.1. Primitives and their descriptions for Population Density

```python

def elementwise_max(matrix1, matrix2):
    """
    Compute the element-wise maximum of two matrices.

    Parameters:
        matrix1 (numpy.ndarray): First input matrix.
        matrix2 (numpy.ndarray): Second input matrix.

    Returns:
        numpy.ndarray: Element-wise maximum of the input matrices.
    """

def elementwise_min(matrix1, matrix2):
    """
    Compute the element-wise minimum of two matrices.

    Parameters:
        matrix1 (numpy.ndarray): First input matrix.
        matrix2 (numpy.ndarray): Second input matrix.

    Returns:
        numpy.ndarray: Element-wise minimum of the input matrices.
    """

def elementwise_sum(matrix1, matrix2):
    """
    Compute the element-wise sum of two matrices.

    Parameters:
        matrix1 (numpy.ndarray): First input matrix.
        matrix2 (numpy.ndarray): Second input matrix.

    Returns:
        numpy.ndarray: Element-wise sum of the input matrices.
    """

def elementwise_product(matrix1, matrix2):
    """
    Compute the element-wise product of two matrices.

    Parameters:
        matrix1 (numpy.ndarray): First input matrix.
        matrix2 (numpy.ndarray): Second input matrix.

    Returns:
        numpy.ndarray: Element-wise product of the input matrices.
    """

def elementwise_division(matrix1, matrix2):
    """
    Compute the element-wise division of two matrices.

    Parameters:
```

```
        matrix1 (numpy.ndarray): First input matrix.
        matrix2 (numpy.ndarray): Second input matrix.

    Returns:
        numpy.ndarray: Element-wise division of the input matrices.
    """

def matrix_scalar_multiplication(matrix, scalar):
    """
    Perform matrix scalar multiplication.

    Parameters:
        matrix (numpy.ndarray): Input matrix.
        scalar (int or float): Scalar value.

    Returns:
        numpy.ndarray: Result of matrix scalar multiplication.
    """

def elementwise_log(matrix):
    """
    Compute the element-wise logarithm of a matrix.

    Parameters:
        matrix (numpy.ndarray): Input matrix.

    Returns:
        numpy.ndarray: Element-wise logarithm of the input matrix.
    """

def elementwise_exponentiate(matrix, base):
    """
    Compute the exponentiation of each element of a matrix with a given base.

    Parameters:
        matrix (numpy.ndarray): Input matrix.
        base (int or float): Base value.

    Returns:
        numpy.ndarray: Exponentiated matrix.
    """

def min_pixel_distance_to_mask(mask):
    """
    Compute the minimum pixel distance from each pixel to a mask.

    Parameters:
        mask (numpy.ndarray): Binary mask array where 1 represents the mask and 0
        ↪  represents the background.

    Returns:
        numpy.ndarray: Minimum pixel distance to the mask.
    """

def segment(im, text_prompt="trees"):
    """
    Segments a satellite image based on a text prompt. The text prompt can only take one
    ↪  concept at a time.
```

```
    Parameters:
        im (numpy.ndarray): An rgb satellite image.
        text_prompt (str): a text prompt

    Returns:
        mask (numpy.ndarray): Binary mask array where 1 represents the mask and 0
        ↪  represents the background.
    Example:
    text_prompt can be but not limited to these:
    ["tennis", "skate park", "football field", "swimming pool", "cemetery", "multi-storey
    ↪  garage", "golf", "roundabout", "parking lot", "supermarket", "school", "marina",
    ↪  "baseball field", "fall", "pond", "airport", "beach", "bridge", "religious
    ↪  building", "residential building", "warehouse", "office building", "farmland",
    ↪  "university building", "forest", "lake", "nature reserve", "park", "sand", "soccer
    ↪  field", "equestrian club", "shooting range", "ice-rink", "commercial area",
    ↪  "garden", "dam", "railroad", "highway", "river", "wetland", "non-residential
    ↪  buildings", "coastline"]
    """
```

I.2. Primitives and their descriptions for AGB and Poverty prediction

The evolutionary search uses all the above defined functions, plus the following:

```


def get_satellite_image(location):
    """
    Get the satellite image for a given location.
    Parameters:
        location (tuple): Tuple containing the latitude and longitude of the location.
    Returns:
        numpy.ndarray: Satellite image for the location.

    Can be used for segmentation ONLY.
    """

def get_temperature(location):
    """
    Get the average annual temperature for a given location.
    Parameters:
        location (tuple): Tuple containing the latitude and longitude of the location.
    Returns:
        float: Temperature for the location normalized between 0 and 255.
    """

def get_precipitation(location):
    """
    Get the average annual precipitation for a given location.
    Parameters:
        location (tuple): Tuple containing the latitude and longitude of the location.
    Returns:
        float: Precipitation for the location between 0 and 255.
    """

def get_elevation(location):
    """
```

```
    Get the elevation for a given location.
    Parameters:
        location (tuple): Tuple containing the latitude and longitude of the location.
    Returns:
        float: Digital Elevation for the location (scaled 0-8000) to 0-255.
    """

def get_nightlight_intensity(location):
    """
    Get the average annual nightlight intensity for a given location.
    Parameters:
        location (tuple): Tuple containing the latitude and longitude of the location.
    Returns:
        float: Nightlight intensity for the location (between 0 and 1).
    """

def get_average(segmented_image):
    """
    Get the average pixel value of a segmented image.

    Parameters:
        segmented_image (numpy.ndarray): Segmented image.

    Returns:
        float: Average pixel value of the segmented image.

```

I.3. Primitives and their descriptions for CSIF Forecasting

For CSIF forecast, the API borrows mathematical and logical functions from above. Additionally it has the following to obtain more environmental variables.

```


def get_historical_csif(locations, num_months=36):
    """
    Get historical CSIF (contiguous solar induced chlorophyll fluorescence) time-series
    ↪ data for the last given number of months.

    Parameters:
        locations: a list of locations in a specific format.
        num_months (int): Last number of months to get time-series data for.
    Returns:
        numpy.ndarray: Historical time series data for the given locations. size =
        ↪ (len(locations), num_months)
    """

def get_historical_min_temperature(locations, num_months=36):
    """
    Get historical minimum temperature time-series data for the last given number of
    ↪ months.

    Parameters:
        locations: a list of locations in a specific format.
        num_months (int): Last number of months to get time-series data for.
    Returns:
        numpy.ndarray: Historical time series data for the given locations. size =
        ↪ (len(locations), num_months)
```

```
    """


def get_historical_max_temperature(locations, num_months=36):
    """
    Get historical maximum temperature time-series data for the last given number of
    ↪  months.

    Parameters:
        locations: a list of locations in a specific format.
        num_months (int): Last number of months to get time-series data for.
    Returns:
        numpy.ndarray: Historical time series data for the given locations. size =
        ↪  (len(locations), num_months)
    """


def get_historical_radiation(locations, num_months=36):
    """
    Get historical solar radiation time-series data for the last given number of months.

    Parameters:
        locations: a list of locations in a specific format.
        num_months (int): Last number of months to get time-series data for.
    Returns:
        numpy.ndarray: Historical time series data for the given locations. size =
        ↪  (len(locations), num_months)
    """

def get_historical_precipitation(locations, num_months=36):
    """
    Get historical precipitation time-series data for the last given number of months.

    Parameters:
        locations: a list of locations in a specific format.
        num_months (int): Last number of months to get time-series data for.
    Returns:
        numpy.ndarray: Historical time series data for the given locations. size =
        ↪  (len(locations), num_months)
    """


def get_historical_photoperiod(locations, num_months=36):
    """
    Get historical photoperiod time-series data for the last given number of months.

    Parameters:
        locations: a list of locations in a specific format.
        num_months (int): Last number of months to get time-series data for.
    Returns:
        numpy.ndarray: Historical time series data for the given locations. size =
        ↪  (len(locations), num_months)
    """


def get_historical_swvl1(locations, num_months=36):
    """
```

```
    Get historical soil water content in the first layer time-series data for the last
    ↪ given number of months.

    Parameters:
        locations: a list of locations in a specific format.
        num_months (int): Last number of months to get time-series data for.
    Returns:
        numpy.ndarray: Historical time series data for the given locations. size =
        ↪ (len(locations), num_months)
    """


def get_present_min_temperature(locations):
    """
    Get present minimum temperature data for the given locations.

    Parameters:
        locations: a list of locations in a specific format.
    Returns:
        numpy.ndarray: Present minimum temperature data for the given locations. size =
        ↪ (len(locations),)
    """

def get_present_max_temperature(locations):
    """
    Get present maximum temperature data for the given locations.

    Parameters:
        locations: a list of locations in a specific format.
    Returns:
        numpy.ndarray: Present maximum temperature data for the given locations. size =
        ↪ (len(locations),)
    """

def get_present_radiation(locations):
    """
    Get present solar radiation data for the given locations.

    Parameters:
        locations: a list of locations in a specific format.
    Returns:
        numpy.ndarray: Present solar radiation data for the given locations. size =
        ↪ (len(locations),)
    """

def get_present_precipitation(locations):
    """
    Get present precipitation data for the given locations.

    Parameters:
        locations: a list of locations in a specific format.
    Returns:
        numpy.ndarray: Present precipitation data for the given locations. size =
        ↪ (len(locations),)
    """

def get_present_photoperiod(locations):
```

```
    """
    Get present photoperiod data for the given locations.

    Parameters:
        locations: a list of locations in a specific format.
    Returns:
        numpy.ndarray: Present photoperiod data for the given locations. size =
        ↪  (len(locations),)
    """

def get_present_swvl1(locations):
    """
    Get present soil water content in the first layer data for the given locations.

    Parameters:
        locations: a list of locations in a specific format.
    Returns:
        numpy.ndarray: Present soil water content in the first layer data for the given
        ↪  locations. size = (len(locations),)
    """


```

References

[1] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016. 10

[2] Hans Hersbach, Bill Bell, Paul Berrisford, Shoji Hirahara, András Horányi, Joaquín Muñoz-Sabater, Julien Nicolas, Carole Peubey, Raluca Radu, Dinand Schepers, et al. The era5 global reanalysis. *Quarterly Journal of the Royal Meteorological Society*, 146(730): 1999–2049, 2020. 2

[3] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997. 2

[4] Pang Wei Koh, Thao Nguyen, Yew Siang Tang, Stephen Mussmann, Emma Pierson, Been Kim, and Percy Liang. Concept bottleneck models. In *ICML*, 2020. 10

[5] Sachit Menon and Carl Vondrick. Visual classification via description from large language models. *International Conference on Learning Representations*, 2023. 10

[6] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *Medical image computing and computer-assisted intervention–MICCAI 2015: 18th international conference, Munich, Germany, October 5-9, 2015, proceedings, part III 18*, pages 234–241. Springer, 2015. 10

[7] Qwen Team. Qwen2.5: A party of foundation models, 2024. 9

[8] Yue Yang, Artemis Panagopoulou, Shenghao Zhou, Daniel Jin, Chris Callison-Burch, and Mark Yatskar. Language in a bottle: Language model guided concept bottlenecks for interpretable image classification. In *CVPR*, 2023. 10

[9] George Zerveas, Srideepika Jayaraman, Dhaval Patel, Anuradha Bhamidipaty, and Carsten Eickhoff. A transformer-based framework for multivariate time series representation learning. In *Proceedings of the 27th ACM SIGKDD conference on knowledge discovery & data mining*, pages 2114–2124, 2021. 2

[10] Yao Zhang, Joanna Joiner, Seyed Hamed Alemohammad, Sha Zhou, and Pierre Gentine. A global spatially contiguous solar-induced fluorescence (csif) dataset using neural networks. *Biogeosciences*, 15(19):5779–5800, 2018. 1