A DATASETS

The statistical information for all PMLB datasets used in this paper is provided in Figure 5 and Figure 6. Figure 5 depicts the basic properties of all datasets. It demonstrates that the amount of all experimental data spans from 200 to 9822, and the number of features ranges from 2 to 1000. Figure 6 displays the maximum number of classes and the imbalance score; the maximum number of classes is 18 and the maximum imbalance score is 0.93. The imbalance score is defined as the square deviation between the real label distribution $\frac{n_c}{n}$ and the ideal label distribution $\frac{1}{C}$, i.e., $\sum_{c=1}^{C}(\frac{n_c}{n} - \frac{1}{C})^2$, where n_c represents the number of instances of class c. This score is normalized by the imbalance score of a fake dataset with $n-1$ items as the first class sample and 1 item as the second class sample. All visualization results indicate that our experimental data encompasses a diverse variety of datasets with varying properties.

The statistical information of DIGEN datasets is not plotted because the size of each dataset is 1000 and each dataset has 10 features. Each dataset is a binary classification task with balanced labels. Although the number of instances and features is fixed, datasets still contain a high degree of diversity because all datasets are synthesized using a genetic programming method and manually selected by human specialists (Orzechowski & Moore, 2021). The optimization goal of DIGEN is to maximize the accuracy difference between two specific machine learning algorithms and simultaneously maximize the standard deviation of accuracy with respect to eight machine learning algorithms. This enables each dataset to have adequate complexity to differentiate the learning capabilities of different algorithms.

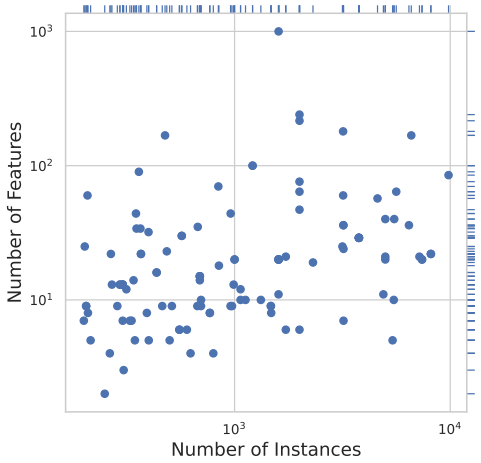

Figure 5: Properties of PMLB datasets.

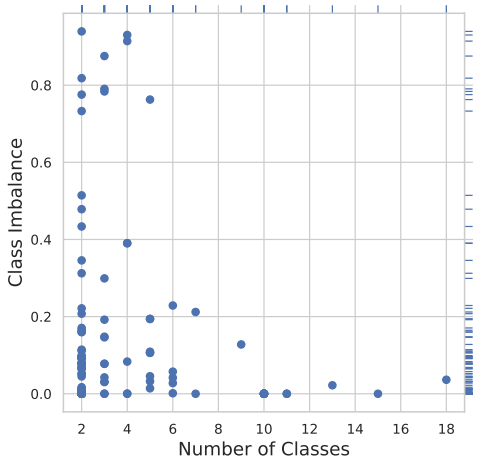

Figure 6: Properties of PMLB datasets.

B TIME CONSUMPTION COMPARISON

In addition to testing accuracy, time consumption is also a factor that should be considered. This section examines the time consumption of all algorithms. All experiments are carried out on the same sever with a Kunpeng 920 ARM CPU. For machine learning algorithms, the hyperparameter optimization time is taken into account. Figure 7 and Table 4 show the statistical data for training time consumption. The experimental results reveal that deep learning-based feature construction methods are slower than evolution-based feature construction methods. FTTransformer, the most accurate but slowest deep learning approach, is around one order of magnitude slower than EvoFeat. This confirms that the evolutionary algorithm, rather than deep learning, is preferable for feature construction on tabular data classification tasks. Furthermore, when compared to hyperparameter optimization on traditional machine learning algorithms, EvoFeat takes the same order of magnitude of time as LightGBM, but produces better prediction accuracy. Consequently, EvoFeat is worth trying first in a real-world scenario where computing resources are constrained.

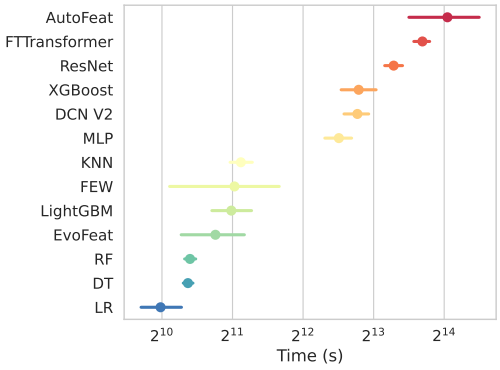

Figure 7: Wall-clock runtime for each methods in seconds.

Algorithm	Mean	Speedup
AutoFeat$^{\diamond}$	16913.69	9.77x
FTTransformer‡	13227.59	7.64x
ResNet‡	9973.16	5.76x
XGBoost†	7073.36	4.09x
DCN V2‡	6984.05	4.03x
MLP‡	5821.68	3.36x
KNN†	2223.24	1.28x
FEW$^{\diamond}$	2087.66	1.21x
LightGBM†	2023.95	1.17x
EvoFeat*	1731.26	1.0x
RF†	1348.87	0.78x
DT†	1321.54	0.76x
LR†	1010.16	0.58x

Table 4: Comparison of wall-clock time (seconds) over different algorithms. ($\star$: Our algorithm, $\dagger$: ML algorithms, $\ddagger$: DL algorithms, $\diamond$: FE algorithms)

C Instance Space Analysis

Based on the non-free-lunch theorem of machine learning algorithms, simply demonstrating that EvoFeat outperforms XGBoost and LightGBM on average may not be sufficient. It is worthwhile to investigate which classification tasks EvoFeat performs well on, as this will help determine whether to use EvoFeat for an unseen dataset. In this section, we employ the instance space analysis (ISA) (Muñoz et al., 2021) technique to assess the benefits of each learning algorithm.

First, we extract meta-features from all 119 PMLB datasets using an open-source meta-feature extraction tool (Alcobaça et al., 2020). These meta-features include basic dataset meta-features, statistical meta-features for numerical features, and information theory based meta-features for categorical features. After feature extraction, meta-features with null values are removed. This yields 63 meta-features for analysis, which are then combined with experimental results from 12 algorithms to form a meta-dataset for further investigation.

After obtaining the meta-dataset, ISA is used to transform the meta-feature space into a two-dimensional instance space. It first selects five features from all meta-features using neighborhood component analysis (NCA) (Yang et al., 2012). Then, the five-dimensional feature space is reduced to two dimensions using the prediction-based linear dimensionality reduction method (PBLDR) (Muñoz et al., 2018). Equation (1) shows the selected features as well as the transformation matrix. The selected features are the number of instances (nr_inst), the number of classes (nr_class), the attribute equivalence (eq_num_attr), the number of attributes (nr_attr), and the mean value of the maximum of each feature ($max.mean$). The attribute equivalence is defined as $nr\_attr * \frac{H(y)}{\sum_x MI(x,y)}$, where $H(y)$ represents the entropy of the target variable y and $MI(x, y)$ represents the mutual-information between an exploratory variable x and the target variable y. Based on the 2-D feature space, we could visualize the good instances of each algorithm. We consider an instance to be good on a specific algorithm if its testing accuracy is within 5% of the best algorithm. Based on this criterion, we plot Figure 8 to show the distribution of good instances.

The experimental results show that EvoFeat can effectively handle problem instances in the third quadrant, whereas deep learning methods struggle in this area. According to the transformation matrix, the instances in the third quadrant may have a high degree of attribute equivalence. In other words, the categorical variables in these cases have poor mutual information with the target variable, and thus high-order feature crosses may be required for better predictive accuracy. Consequently, the poor performance of deep learning on these datasets may confirm the poor ability of deep learning to construct high-order cross-features (Wang et al., 2021). One thing to keep in mind is that LightGBM complements EvoFeat in the fourth quadrant. Because the problem instances in the fourth quadrant have a limited number of data items, using LightGBM with simpler features may be preferable to

EvoFeat in some cases. Based on these findings, we can conclude that EvoFeat is a viable method for developing high-order features, but feature construction is not required in all scenarios. In some cases, using original features may be a better option, and an external validation set should be used to determine whether high-order features should be constructed in real-world applications.

$$\begin{bmatrix} -0.15 & -0.19 \\ -0.16 & 0.36 \\ 0.05 & -0.25 \\ -0.41 & -0.43 \\ -0.20 & 0.20 \end{bmatrix}^{\mathrm{T}} \begin{bmatrix} \text{nr\_class} \\ \text{nr\_inst} \\ \text{max.mean} \\ \text{eq\_num\_attr} \\ \text{nr\_attr} \end{bmatrix} \tag{1}$$

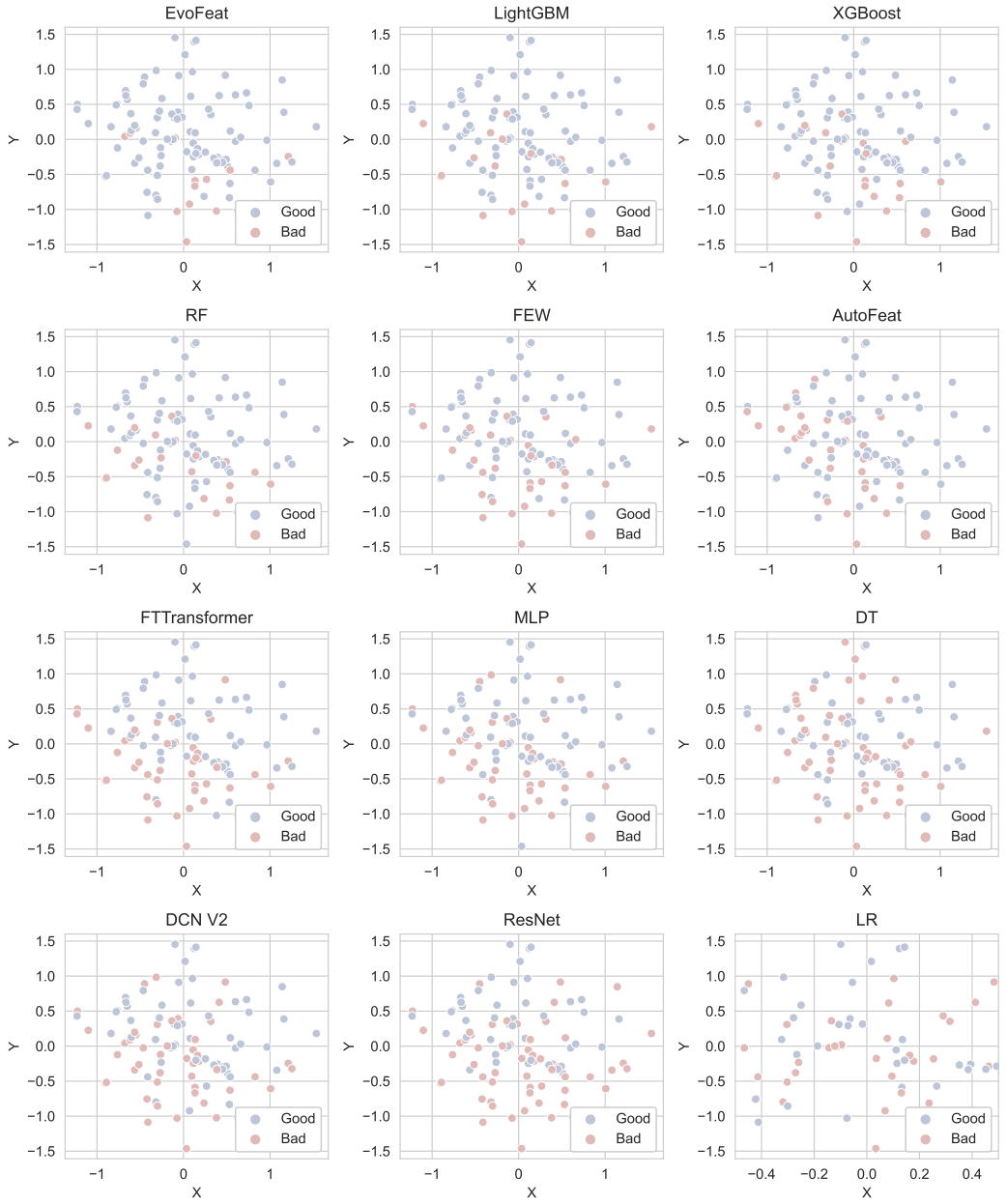

Figure 8: Distribution of good or bad problem instances on the projected instance space.

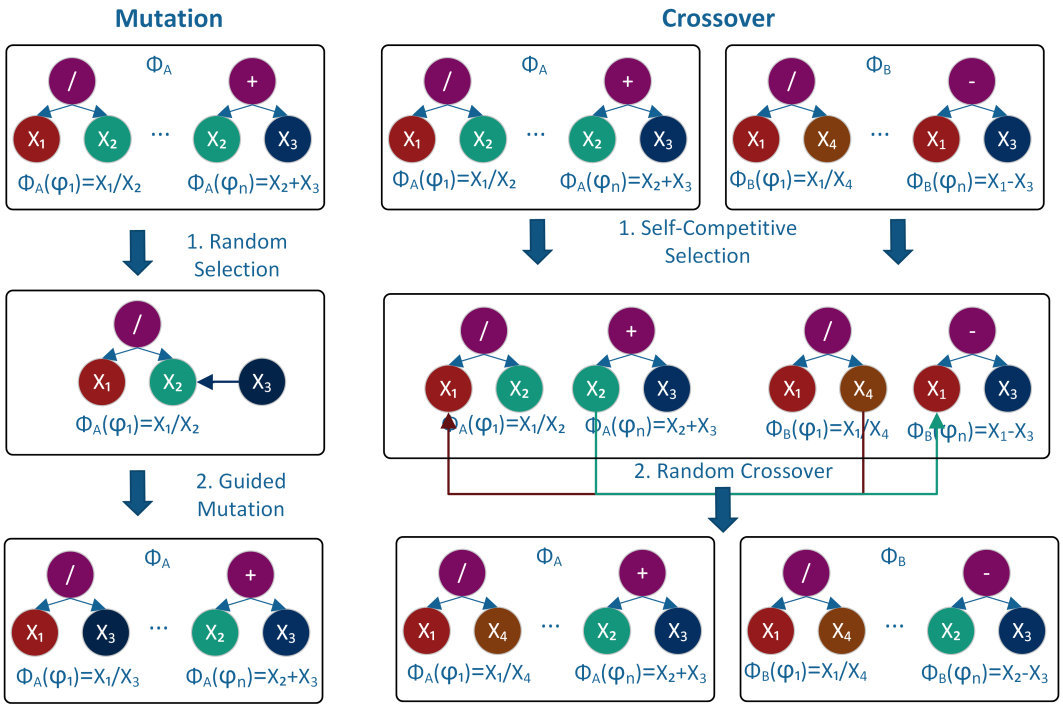

Figure 9: An example of the crossover and mutation operators.

D Further Explanation of Mutation and Crossover

In EvoFeat, the guided mutation operator and the self-competitive crossover operator are used for new individual generation. To make these operators more clear, we present an example in Figure 9 to show the workflow of the guided mutation and the self-competitive crossover operator. First, for the guided selection operator, it randomly selects a symbolic tree from all symbolic trees $\{\phi_1 \dots \phi_n\}$, and then it is applied to that symbolic tree to randomly replace an old subtree with a newly generated tree. The non-terminal nodes of the new tree are randomly generated, but the value of terminal nodes are generated based on the feature importance values. In Figure 9, $\Phi_A(\phi_1)$ is selected for mutation, and then a subtree x_3 is generated to replace the subtree x_2 in $\Phi_A(\phi_1)$. Following the substitution of the subtree, the new individual Φ_A is composed of the new tree and $n-1$ old trees with no variation. In terms of the crossover operator, each of two parent individuals selects one acceptor and donor based on the feature importance values, and then a subtree of the donor replaces the subtree of the acceptor. For example, $\Phi_B(\phi_1)$ is chosen as the donor and $\Phi_A(\phi_1)$ is chosen as the acceptor. Then, $\Phi_B(\phi_1)$ substitutes the subtree x_1 of $\Phi_A(\phi_1)$ with x_4. After the replacement, the new acceptor tree replaces the old tree in the corresponding individual. Because the selection probability of each acceptor tree is inversely proportional to the feature importance value, it is highly likely that replacing the new acceptor tree will not degrade the model performance and thus strike a balance between exploration and exploitation.

E Further Analysis on Key Components

E.1 Selection Operators

This paper proposes a two-layer selection mechanism using cross-validation losses and feature importance values. To test the effectiveness of the two-layers design, we test 12 groups of selection operators in this section. For the upper-level selection operator for individuals, we test random selection, tournament selection (TN), and automatic lexicase selection (AL). Both TR and AL use cross-validation losses to select parent individuals. TR only uses the sum of cross-validation losses for parent selection, while AL uses the vector of cross-validation losses in the selection process to increase the selection diversity. For the lower-level selection operator in the feature crossover operator, we test random crossover and self-competitive crossover (SC) operators. The self-competitive crossover operator selects two features based on feature importance values, while the random crossover operator

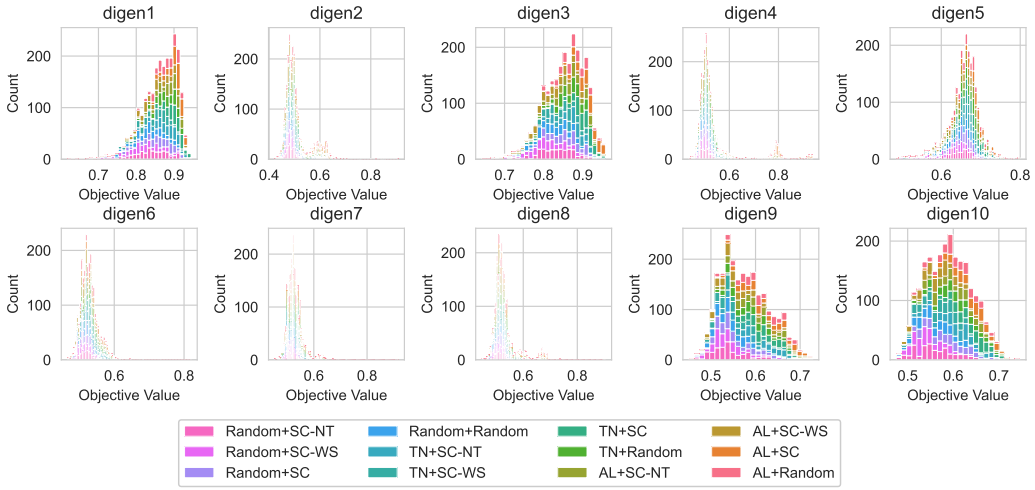

Figure 10: Distribution of objective values for different combinations of selection operators.

does not. Remarkably, the SC operator in EvoFEAT differs from the SC operator in job scheduling in that we use a softmax function with a low temperature ($T = 1/20$) to strengthen the distinction between important features and useless features. In order to demonstrate that this is an appropriate setting, we compare SC with two variants, i.e., SC with normal temperature (SC-NT, $T = 1$) and SC without softmax (SC-WS).For the evaluation protocol, we randomly initialize 200 individuals and generate 200 offspring using different combinations of selection operators. We plot the distribution of offspring objective values and uses the Wilcoxon rank-sum test to test the significance of the results between operators.

Figure 10 presents the distribution of objective values of offspring for different selection operators. Combining with statistical comparison in Table 5, it is clear that integrating AL with SC is better than other combination of operators. For the selection operator of features, Table 5 shows that AL+SC outperforms AL+Random on all ten datasets, showing that it is effective to take feature importance values into consideration. In terms of selection operators for individuals, AL+SC is better than TN+SC and Random+SC on 5 and 10 datasets respectively, showing the importance of using the vector of cross-validation losses to select parent individuals. As for whether a softmax function with low temperature should be used, Table 5 demonstrates that AL+SC is better than AL+SC-NT and AL+SC-WS on all 10 datasets, confirming that strengthening important features and weakening useless features is important when applying using feature importance values for selection in evolutionary feature construction scenario.

E.2 Multi-task Evolution

Evolving features with machine learning models is an intuitive idea because different ML models may perform well on different sets of features. However, one question is whether we should evolve features on different ML models independently or within a multi-task optimization framework. In this section, we conduct experiments to investigate whether it is beneficial to evolve features simultaneously. Two groups of strategies are compared. One group is to generate offspring based on 100 individuals using DT or LR as base learners. Another group is to cross features between good individuals for DT and LR, where 100 offspring are generated by using good individuals for DT as acceptors and grafting subtrees from good individuals for LR or in an opposite way. In order to mitigate adverse impact caused by the negative transfer, we only allow building blocks to transfer if the fitness value individual of the donor individual is better than the acceptor individual. Figure 11 shows the distribution plots of top-50% individuals and Table 6 presents the statistical results of top-50% individuals using the Wilcoxon rank-sum test. Table 6 shows that transferring features from good individuals on LR/DT to good individuals on DT/LR is better than crossing two good individuals either on DT or LR on 3 out of 20 cases. These experiments show that good individuals on different base learners can benefit from each other by exchanging building blocks. It worth to note that negative transfer is possible, and

Table 5: Statistical results of objective values over different selection operators. ("+", "=" or "-" represents AL+SC is significant better than, similar to and worse than other selection operators.)

	AL+SC	AL+Random	AL+SC-WS	AL+SC-NT
digen1	0.9065	0.879 (+)	0.8535 (+)	0.888 (+)
digen2	0.5775	0.502 (+)	0.4845 (+)	0.505 (+)
digen3	0.908	0.873 (+)	0.8285 (+)	0.8675 (+)
digen4	0.791	0.523 (+)	0.5 (+)	0.5225 (+)
digen5	0.6835	0.663 (+)	0.649 (+)	0.671 (+)
digen6	0.539	0.529 (+)	0.519 (+)	0.527 (+)
digen7	0.54	0.5275 (+)	0.518 (+)	0.527 (+)
digen8	0.536	0.523 (+)	0.5145 (+)	0.524 (+)
digen9	0.608	0.6045 (+)	0.552 (+)	0.5895 (+)
digen10	0.6345	0.602 (+)	0.5655 (+)	0.5985 (+)
+/=/-	-	10/0/0	10/0/0	10/0/0
	TN+Random	TN+SC	TN+SC-WS	TN+SC-NT
digen1	0.881 (+)	0.8995 (+)	0.851 (+)	0.88 (+)
digen2	0.498 (+)	0.503 (+)	0.489 (+)	0.499 (+)
digen3	0.8685 (+)	0.8995 (+)	0.843 (+)	0.873 (+)
digen4	0.517 (+)	0.524 (+)	0.506 (+)	0.5185 (+)
digen5	0.67 (+)	0.681 (=)	0.653 (+)	0.667 (+)
digen6	0.532 (+)	0.539 (=)	0.525 (+)	0.533 (=)
digen7	0.527 (+)	0.538 (=)	0.522 (+)	0.532 (+)
digen8	0.523 (+)	0.5335 (+)	0.515 (+)	0.525 (+)
digen9	0.582 (+)	0.609 (=)	0.548 (+)	0.591 (+)
digen10	0.6025 (+)	0.636 (=)	0.573 (+)	0.605 (+)
+/=/-	10/0/0	5/5/0	10/0/0	9/1/0
	Random+Random	Random+SC	Random+SC-WS	Random+SC-NT
digen1	0.836 (+)	0.86 (+)	0.817 (+)	0.8395 (+)
digen2	0.482 (+)	0.492 (+)	0.48 (+)	0.483 (+)
digen3	0.823 (+)	0.856 (+)	0.809 (+)	0.834 (+)
digen4	0.501 (+)	0.505 (+)	0.5 (+)	0.5 (+)
digen5	0.6485 (+)	0.659 (+)	0.6355 (+)	0.6435 (+)
digen6	0.52 (+)	0.521 (+)	0.514 (+)	0.518 (+)
digen7	0.518 (+)	0.522 (+)	0.5155 (+)	0.5175 (+)
digen8	0.5115 (+)	0.515 (+)	0.5085 (+)	0.514 (+)
digen9	0.5415 (+)	0.544 (+)	0.533 (+)	0.536 (+)
digen10	0.561 (+)	0.584 (+)	0.538 (+)	0.554 (+)
+/=/-	10/0/0	10/0/0	10/0/0	10/0/0

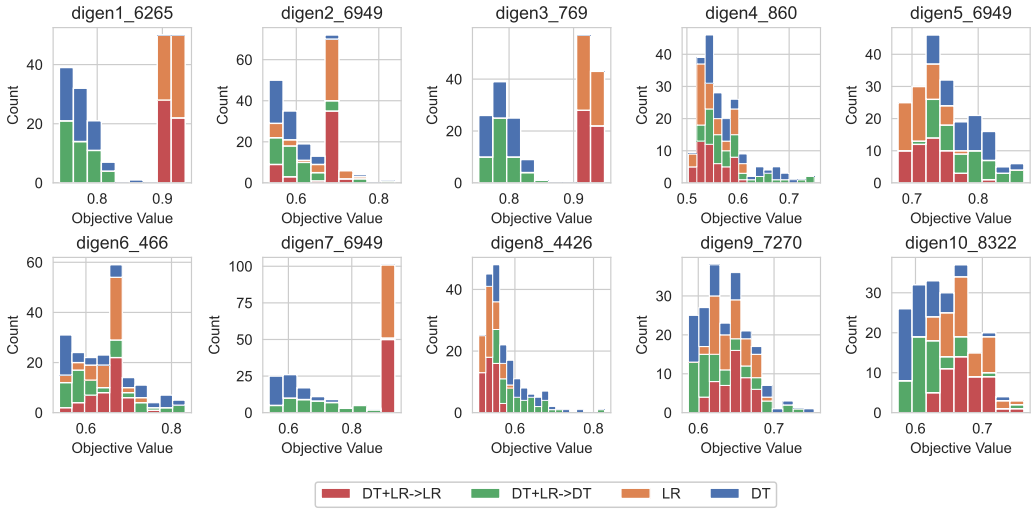

Figure 11: Distribution of objective values for using transfer optimization or not.

Table 6: Statistical results of objective values using transfer optimization or not. ("+", "=" or "-" represents transferring features from another kind of base learner is significant better than, similar to and worse than not transferring features for generating offsprings.)

	DT	LR	DT+LR->DT	DT+LR->LR
digen1_6265	0.775	0.9164	0.771(=)	0.9139(=)
digen2_6949	0.5864	0.6627	0.6111(+)	0.657(=)
digen3_769	0.7878	0.9251	0.7911(=)	0.9265(=)
digen4_860	0.5877	0.5513	0.5858(=)	0.551(=)
digen5_6949	0.7829	0.7159	0.7776(=)	0.727(+)
digen6_466	0.6478	0.6524	0.6393(=)	0.6502(=)
digen7_6949	0.6055	0.9221	0.6913(+)	0.9227(=)
digen8_4426	0.5976	0.5384	0.6066(=)	0.5381(=)
digen9_7270	0.6306	0.6426	0.6292(=)	0.6461(=)
digen10_8322	0.6122	0.6732	0.6207(=)	0.6732(=)
+/=/-	-	-	2/8/0	1/9/0

thus investigating how to avoid negative transfer (Wei et al., 2021) is an important future research direction.

E.3 Feature Importance Calculation Methods

EvoFeat includes two different base learners: DT and LR. Based on these two base learners, EvoFeat uses the built-in feature importance calculation methods in these base learners to obtain the feature importance values. For DT, EvoFeat uses the reduction of Gini impurity to calculate the importance of each input feature. For LR, EvoFeat uses the normalized absolute value of the coefficient to determine the feature importance value of each feature. Nonetheless, there are other ways to calculate feature importance values in the machine learning community. Here, we compare our built-in method with two feature importance calculation methods: permutation importance and Shapley additive explanations (SHAP) values Lundberg & Lee (2017). Thanks to EvoFeat using a 5-fold cross-validation method to evaluate each individual, we can use the test set in each fold to calculate permutation importance and SHAP values. For the evaluation protocol, we randomly generate 200 individuals and evaluate them with different feature importance calculation methods. Then, we generate 200 offspring based on calculated feature importance values and compare the objective values of 200 offspring generated according to different importance calculation methods. Figure 12 presents the distribution of objective values of offspring for different feature importance calculation methods. The statistical comparison results for objective values are presented in Table 7, and the

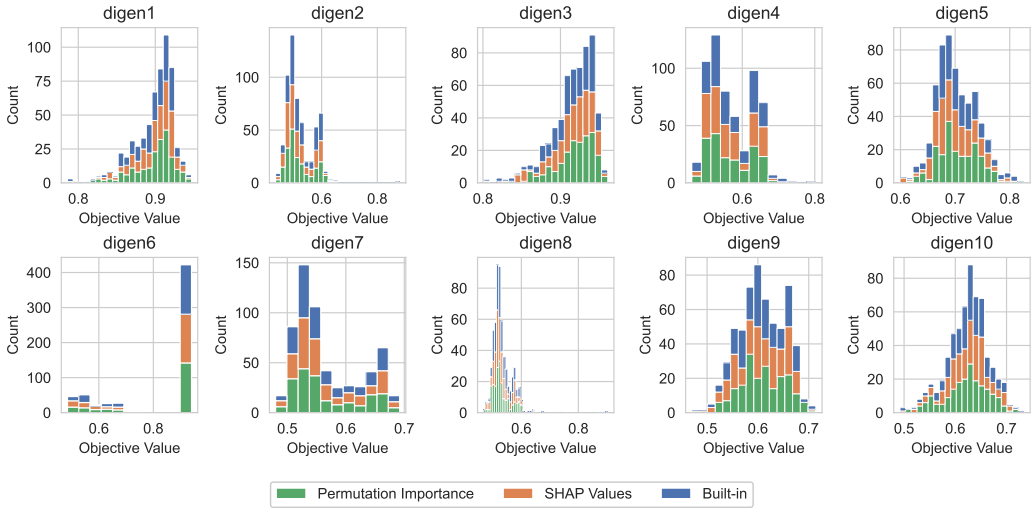

Figure 12: Distribution of objective values for different feature importance calculation methods.

Table 7: Statistical results of objective values with respect to different feature importance calculation methods. ("+", "=" or "-" represents using the built-in feature importance calculation method is significant better than, similar to and worse than using other methods for calculating importance values.)

	Built-in	SHAP Values	Permutation Importance
digen1_6265	0.904	0.907 (=)	0.906 (=)
digen2_6949	0.51	0.5025 (=)	0.503 (=)
digen3_769	0.917	0.9195 (=)	0.921 (=)
digen4_860	0.557	0.553 (=)	0.5495 (=)
digen5_6949	0.6965	0.694 (=)	0.699 (=)
digen6_466	0.911	0.91 (=)	0.911 (=)
digen7_6949	0.545	0.5435 (=)	0.544 (=)
digen8_4426	0.5255	0.523 (=)	0.523 (=)
digen9_7270	0.6025	0.607 (=)	0.608 (=)
digen10_8322	0.6285	0.628 (=)	0.626 (=)
+/=/-	-	0/10/0	0/10/0

comparison results for evaluation time are presented in Table 8. Based on these two tables, we can find that there is a minor difference between different feature importance calculation methods in terms of objective values for offspring. As for the evaluation time, the built-in method is significantly faster than SHAP and permutation importance on all 10 datasets. According to these experimental results, we can conclude that using built-in feature importance calculation methods in LR and DT is an appropriate choice in EvoFeat to calculate feature importance values, as it is faster than two other methods while it has similar effectiveness.

F Impact of Ensemble Feature Construction

In contrast to many traditional feature construction methods, which construct a set of features to enhance the performance of a specific classifier, such as XGBoost or RF. EvoFeat evolves multiple sets of features to improve the performance of multiple base classifiers. In this section, we test whether the ensemble feature construction paradigm outperforms the traditional wrapper-based one on the RDT and LR models, which are two types of classifiers used in EvoFeat. There are four variants to be studied in total, and a brief introduction to these algorithms is shown below:

- LR (Ensemble): An ensemble of LR with distinct feature sets for each LR.
- RDT (Ensemble): An ensemble of RDT with distinct feature sets for each RDT.

Table 8: Statistical results of evaluation time (seconds) with respect to different feature importance calculation methods. ("-", "=" or "+" represents using the built-in feature importance calculation method is significant better than, similar to and worse than using other methods for calculating importance values.)

	Built-in	SHAP Values	Permutation Importance
digen1_6265	0.03	0.0365 (-)	0.1294 (-)
digen2_6949	0.0265	0.032 (-)	0.1166 (-)
digen3_769	0.029	0.036 (-)	0.1115 (-)
digen4_860	0.0248	0.03 (-)	0.1056 (-)
digen5_6949	0.025	0.0315 (-)	0.1066 (-)
digen6_466	0.027	0.033 (-)	0.1155 (-)
digen7_6949	0.028	0.032 (-)	0.1175 (-)
digen8_4426	0.027	0.036 (-)	0.1259 (-)
digen9_7270	0.026	0.0345 (-)	0.125 (-)
digen10_8322	0.0288	0.032 (-)	0.1138 (-)
+/=/-	-	0/0/10	0/0/10

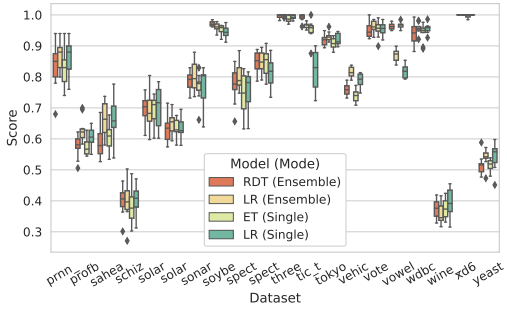

Figure 13: The balanced testing accuracy of our algorithm with respect to different models and modes.

Model (Mode)	Mean	Imp. ↑
LR (Ensemble)	0.849455	-
RDT (Ensemble)	0.848104	-
ET (Single)	0.840809	0.87%
LR (Single)	0.833936	1.86%

Table 9: Comparison of balanced testing accuracy over different models and modes, in which the relative improvement is calculated based on the corresponding ensemble version of the specific mode.

- LR (Single): Using LR to evolve a feature set and evaluating the best feature set on a LR model. A single LR model is used instead of an ensemble of LR models because LR returns the same result for the same dataset.
- ET (Single): Using an RDT as a base learner to evolve feature sets and evaluating the best feature set on an ET with 100 RDTs.

The experiment is conducted on 129 datasets, same as used in the ablation studies. Table 9 shows the experimental results of various models and modes. First, when compared to using only the best set of features to enhance an ensemble of RDTs, an ensemble of feature sets with RDTs performs 0.8% better on average testing accuracy. The performance improvement for the LR model is more significant, with a 1.8 % improvement in average testing accuracy. Based on these experimental results, we can conclude that the best feature set on the training set may not always work well on the testing set, and thus an ensemble of feature sets is beneficial for enhancing the predictive performance of the final model.

G Large-scale Experiment on DIGEN

DIGEN is a synthetic dataset developed specifically to differentiate the performance of ML algorithms. In this section, we conduct an experiment on 40 DIGEN datasets to compare EvoFeat to six ML algorithms. Figure 14 depicts the distribution of testing accuracy for all algorithms across 40 DIGEN datasets. The difference between different algorithms is more noticeable on DIGEN datasets than on PMLB datasets. Experimental results show that, while XGBoost and LightGBM perform quite well, EvoFeat remains the best algorithm. The numerical results shown in Table 1 confirm this, as our method achieves an average accuracy of 96%, which is a more than 4% improvement over the

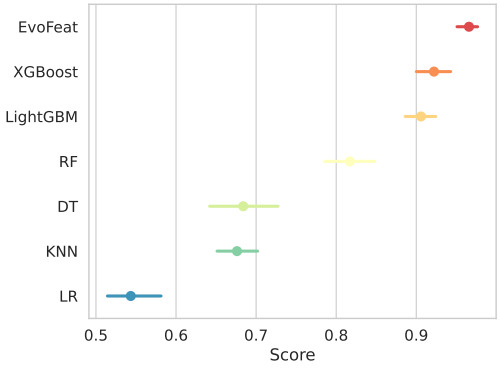

Figure 14: Comparison of balanced testing accuracy over different algorithms on 40 DIGEN datasets.

Algorithm	Mean	Imp. ↑
EvoFeat*	0.965800	-
XGBoost†	0.922100	4.74%
LightGBM†	0.905862	6.62%
RF†	0.816987	18.21%
DT†	0.683950	41.21%
KNN†	0.676150	42.84%
LR†	0.543488	77.70%

Table 10: Statistical results of balanced testing accuracy for different algorithms on 40 DIGEN datasets. ($\star$: Our algorithm, †: ML algorithms, Imp. ↑: Relative improvement)

average accuracies of XGBoost and LightGBM. Based on these results, we further confirm that the evolution-based feature construction method is a effective representation learning mechanism for improving existing machine learning algorithms.

H Hyperparameter Settings

In this section, the hyperparameters of all experimental algorithms are presented. Table 11 presents hyperparameter settings of our method. We follow the empirical experience in genetic programming domain to use a large population size and use a crossover rate that is higher than the mutation rate Zhang et al. (2007). The hyperparameter grids tuned by HEBO are presented from Table 16 to Table 25.

Hyperparameter Tuning for Traditional Machine Learning Methods

The hyperparameters of all methods are tuned using HEBO with 100 iterations. To avoid excessive long waiting time, we also set 4 hours as a time budget constraint. HEBO is chosen because it is the state-of-the-art Bayesian optimization (BO) algorithm for hyperparameter tuning tasks (Cowen-Rivers et al., 2020). We use the five-fold cross-validation method to evaluate each set of hyperparameters, with the average balanced classification accuracy across five-folds set as the optimization objective.

Hyperparameter Tuning for Deep Learning Methods

All hyperparameters are tuned in the same manner as in the previous section. However, because of the high computing cost of the neural network and the enormous number of experimental datasets, we use a single-fold training-validation scheme rather than a cross-validation scheme to optimize the neural network. This is helpful for searching for more hyperparameters within a limited amount of time.

It is worth noting that the self-attention mechanism in the FT-Transformer has a quadratic complexity with respect to the input features. Therefore, for six PMLB datasets with more than 100 features, we reduce the dimensionality to 100 using the principal component analysis (PCA) method before training an FT-Transformer.

Hyperparameter Tuning for Feature Construction Methods

In view of the high complexity of feature construction, we directly use the hyperparameter recommended in existing literature (La Cava & Moore, 2017a) directly. In order to allow the feature construction methods to maximize their potential, the constructed features are validated on RF, ET, XGBoost and LightGBM with a five-fold cross-validation, and the best performing algorithm is chosen as the final algorithm to construct a prediction model. Features generated by AutoFeat are additionally validated on LR because LR is used to evaluate the quality of features in AutoFeat. Due to the high time-complexity of AutoFeat, we reduce the dimensionality to 20 using the PCA method before using it.

Table 11: Hyperparameter settings for EvoFeat.

Parameter	**Value**
Population Size	200
Maximal Number of Generations	30
Crossover and Mutation Rates	0.8 and 0.05
Maximum Tree Depth	8
Initial Tree Depth	2-6
Maximum Forest Size	100
Number of Trees	30
Function Set	+, -, *, Analytical Quotient, Sqaure Root, Sin, Cos, Maximum, Minimum, Negative

Population Size	Mean	Imp. ↑
200	0.860044	-
100	0.860427	-0.04%
300	0.859963	0.01%

Table 12: Comparison of balanced testing accuracy over different population sizes.

Number of Trees	Mean	Imp. ↑
30	0.860044	-
20	0.859476	0.07%
40	0.861319	-0.15%

Table 13: Comparison of balanced testing accuracy over different number of feature trees.

I Hyperparameter Analysis

In this section, we conduct two experiments to investigate the effects of different population sizes and the number of trees in an individual. The hyperparameter analysis experiments, like the ablation studies in the main body of this paper, are carried out on 40 DIGEN datasets and 89 PMLB datasets.

Population Size

Population size is a hyperparameter that determines the number of individuals produced for each generation. A larger population size is advantageous for avoiding premature convergence, but it incurs additional computational costs. Even worse, for some selection operators, such as the roulette wheel selection or the tournament selection operator, a larger population size reduces the likelihood of selecting the top-1 individual, which may impede the exploitation. Therefore, it is worthwhile to study the impact of different population sizes. Table 12 presents the experimental results of EvoFeat with different population sizes. These results show that there is only a 0.01% difference in the average test accuracy of EvoFeat for population sizes of 200 and 300. Thus, the population size is not a critical hyperparameter for EvoFeat, and we can reduce the population size to reduce the computational cost in practice.

Number of Trees

The number of trees in each GP individual is a hyperparameter used to determine the number of features for each base learner in the final model. Since a base learner may not work well if there are not enough features, it might be beneficial to have numerous features in a GP individual. Table 13 presents the relationship between the number of features and the testing accuracy. The results indicate that testing accuracy improves as the number of features increases. For example, using 30 trees improves performance by 0.07% over using 20 trees. It is worth noting that, even though experimental results show that including more trees improves the prediction performance, it also increases the complexity of the final model. Therefore, in real practice, the optimal number of trees should be determined based on the preference for predictive accuracy versus interpretability.

J DETAILED RESULTS

In the main body of our paper, we simply give aggregated results due to the page limit. To further demonstrate the benefits of the proposed method, we present the relative rankings of five top-performing algorithms in Table 14. Besides that, the relative rankings of five top-performing algorithms on DIGEN datasets are given in Table 15. When accuracy scores are equal, we assign a lower rank to them. In other words, the first-rank algorithm should be strictly better than other algorithms. The results show that EvoFeat ranks first on nearly double the number of datasets that LightGBM ranks first on, which further confirms the effectiveness of EvoFeat.

Furthermore, to facilitate reproducibility, the detailed results of PMlb are present in Table 26, and the detailed results of DIGEN are presented in Table 27. Due to space constraints, we only present the details of the top-performing algorithms. The names of datasets are trimmed by retaining only the first 20 characters.

Table 14: The number of occurrences of relative ranking for the top five algorithms on 119 PMLB datasets.

Rank	EvoFeat	XGBoost	LightGBM	RF	FEW
1	51	8	26	20	3
2	14	28	33	18	19
3	10	39	27	18	20
4	25	29	20	20	29
5	19	15	13	43	48

Table 15: The number of occurrences of relative ranking for the top five algorithms on 40 DIGEN datasets.

Rank	EvoFeat	XGBoost	LightGBM	RF	DT
1	32	3	3	1	1
2	4	21	7	6	0
3	1	12	24	5	0
4	2	4	4	28	2
5	1	0	2	0	37

Table 16: XGBoost hyperparameter space.

Parameter	Distribution
Booster	['gbtree','dart']
Number of Estimators	UniformInt[10,1000]
Lambda	UniformLog[1e-5,1e2]
Alpha	UniformLog[1e-5,1e2]
Gamma	Uniform[0,0.5]
Learning Rate	UniformLog[1e-8,1]
Max Depth	UniformInt[1,10]

Table 17: RF hyperparameter space.

Parameter	Distribution
Number of Estimators	UniformInt[10,100]
Criterion	['gini','entropy']
Max Depth	UniformInt[1,10]
Max Features	[None,'auto','log2']
Min Samples Split	UniformInt[2,20]
Min Samples Leaf	UniformInt[1,20]

Table 18: LightGBM hyperparameter space.

Parameter	Distribution
Booster	['gbdt','dart','goss']
Number of Leaves	UniformInt[2,256]
Max Depth	UniformInt[1,10]
Number of Estimators	UniformInt[10,1000]

Table 19: DT hyperparameter space.

Parameter	Distribution
Criterion	['gini','entropy']
Max Depth	UniformInt[1,10]
Min Samples Split	UniformInt[2,20]
Min Samples Leaf	UniformInt[1,20]
Max Features	[None,'auto','log2']

Table 20: KNN hyperparameter space.

Parameter	Distribution
Neighbors	UniformInt[1,50]
Weights	['uniform','distance']
Power	UniformInt[1,5]
Metric	['euclidean','minkowski']

Table 21: LR hyperparameter space.

Parameter	Distribution
Penalty	['l1','l2']
Weights	UniformLog[1e-4,1e4]

Table 22: DNN hyperparameter space.

Parameter	Distribution
Layers	UniformInt[1,8]
Layer Size	UniformInt[1,512]
Embedding Size	UniformInt[64,512]
Dropout	Uniform[0,0.5]
Learning Rate	UniformLog[1e-5,1e-2]
Weight Decay	UniformLog[1e-6,1e-3]

Table 23: DCN V2 hyperparameter space.

Parameter	Distribution
Cross Layers	UniformInt[1,8]
Hidden Layers	UniformInt[1,8]
Layer Size	UniformInt[64,512]
Hidden Dropout	Uniform[0,0.5]
Cross Dropout	Uniform[0,0.5]
Embedding Size	UniformInt[64,512]
Learning Rate	UniformLog[1e-5,1e-2]
Weight Decay	UniformLog[1e-6,1e-3]

Table 24: ResNet hyperparameter space.

Parameter	Distribution
Hidden Layers	UniformInt[1,8]
Layer Size	UniformInt[64,512]
Hidden Factor	UniformInt[1,4]
Hidden Dropout	Uniform[0,0.5]
Residual Dropout	Uniform[0,0.5]
Embedding Size	UniformInt[64,512]
Learning Rate	UniformLog[1e-5,1e-2]
Weight Decay	UniformLog[1e-6,1e-3]

Table 25: FTTransformer hyperparameter space.

Parameter	Distribution
Attention Dropout	Uniform[0,0.5]
Residual Dropout	Uniform[0,0.2]
FFN Dropout	Uniform[0,0.5]
FFN Factor	Uniform[$\frac{2}{3}$,$\frac{8}{3}$]
Token Dimension	UniformInt[64,512]
Layers	UniformInt[1,4]
Learning Rate	UniformLog[1e-5,1e-3]
Weight Decay	UniformLog[1e-6,1e-3]

Table 26: Detailed results of top-performing algorithms on all 119 PMLB datasets.

Dataset	EvoFeat	XGBoost	LightGBM	RF	FEW
GAMETES_Epistasis_2_	0.631875	0.500625	0.508750	0.492813	0.495937
GAMETES_Epistasis_2_	0.669687	0.635000	0.637500	0.575313	0.598750
GAMETES_Epistasis_2_	0.788437	0.770625	0.751250	0.707813	0.735625
GAMETES_Epistasis_3_	0.682500	0.585938	0.564688	0.551875	0.596562
GAMETES_Heterogeneit	0.728125	0.704063	0.685937	0.661875	0.668125
GAMETES_Heterogeneit	0.732187	0.705000	0.720000	0.675000	0.672500
Hill_Valley_with_noi	0.843612	0.560937	0.617718	0.540814	0.774939
Hill_Valley_without_	0.999167	0.649624	0.681433	0.570772	0.994207
agaricus_lepiota	1.000000	1.000000	1.000000	1.000000	1.000000
allbp	0.782345	0.855343	0.916451	0.898575	0.768858
allhyper	0.792185	0.840366	0.866854	0.870332	0.775066
allhypo	0.844344	0.912637	0.920731	0.892881	0.915077
allrep	0.801274	0.824360	0.852013	0.837907	0.803246
analcatdata_authorsh	0.985142	0.987394	0.987071	0.983913	0.990408
analcatdata_dmft	0.227366	0.213644	0.205909	0.211565	0.183660
analcatdata_germangs	0.388750	0.387500	0.382500	0.427500	0.375000
analcatdata_lawsuit	0.915816	0.965816	0.962755	0.968878	0.927296
ann_thyroid	0.990571	0.994478	0.997364	0.996923	0.995587
australian	0.866958	0.860230	0.850426	0.851725	0.844347
auto	0.845623	0.799182	0.828805	0.798675	0.838026
balance_scale	0.979554	0.944874	0.924521	0.597385	0.663192
biomed	0.992963	0.900000	0.922963	0.899259	0.905926
breast	0.962545	0.957201	0.961504	0.964130	0.950453
breast_cancer	0.649928	0.659469	0.674677	0.700430	0.611119
breast_cancer_wiscon	0.958730	0.949504	0.953571	0.941369	0.942659
breast_w	0.955661	0.960915	0.956295	0.962409	0.944112
buggyCrx	0.886257	0.879977	0.872908	0.889036	0.881552
bupa	0.608487	0.580672	0.593067	0.604160	0.552395
calendarDOW	0.578242	0.559130	0.573893	0.574665	0.556432
car	0.992724	0.991234	0.989256	0.972226	0.979733
car_evaluation	0.970392	0.987825	0.993846	0.966908	0.991204
cars	0.984906	0.974702	0.953614	0.911139	0.976743
chess	0.990798	0.993996	0.994609	0.992145	0.993329
churn	0.887691	0.901099	0.903459	0.895098	0.892534
clean1	1.000000	1.000000	1.000000	1.000000	1.000000
clean2	1.000000	1.000000	1.000000	1.000000	1.000000
cleve	0.804275	0.772944	0.810877	0.810335	0.767695
cleveland	0.312208	0.342251	0.304935	0.351602	0.286580
cleveland_nominal	0.270303	0.304069	0.323030	0.335844	0.296277
cmc	0.563572	0.564362	0.558684	0.551561	0.506953
coil2000	0.703093	0.682611	0.694250	0.686071	0.542678
colic	0.808629	0.815997	0.820804	0.835106	0.797203
collins	1.000000	0.978767	0.998901	0.993333	0.998718
contraceptive	0.563572	0.558591	0.554198	0.551711	0.494645
credit_a	0.875357	0.868629	0.865968	0.878955	0.871748
credit_g	0.694881	0.698452	0.704762	0.710357	0.694048
crx	0.875016	0.865627	0.864328	0.863168	0.859889
dermatology	0.963056	0.968889	0.970000	0.968056	0.960386
diabetes	0.769833	0.757204	0.762630	0.758648	0.712722
dis	0.905137	0.855473	0.860571	0.857621	0.793680
dna	0.951622	0.959522	0.961782	0.947150	0.961074
ecoli	0.844922	0.791416	0.815847	0.836494	0.808378
flare	0.719492	0.711925	0.720744	0.701542	0.656070
german	0.710357	0.716786	0.711071	0.709643	0.680238
glass	0.742190	0.691524	0.672000	0.682667	0.664857

haberman	0.607745	0.617391	0.611549	0.612636	0.576223
heart_c	0.809632	0.813636	0.818452	0.812392	0.811310
heart_h	0.804887	0.793922	0.756078	0.762594	0.786153
heart_statlog	0.788333	0.827500	0.816667	0.824167	0.794167
horse_colic	0.818913	0.821355	0.820292	0.827266	0.829905
house_votes_84	0.958352	0.951526	0.953940	0.950472	0.955189
hungarian	0.774060	0.770614	0.759085	0.780827	0.770802
hypothyroid	0.941642	0.947090	0.946277	0.935365	0.916028
ionosphere	0.922783	0.914261	0.922217	0.912826	0.905348
irish	1.000000	1.000000	1.000000	0.997727	1.000000
kr_vs_kp	0.988837	0.993983	0.994445	0.993207	0.994187
led24	0.723979	0.717716	0.722558	0.723022	0.708586
led7	0.732964	0.735512	0.734311	0.737601	0.732142
mfeat_factors	0.974500	0.961500	0.975500	0.966250	0.968250
mfeat_fourier	0.856000	0.828500	0.842750	0.828500	0.834250
mfeat_karhunen	0.952750	0.946250	0.967750	0.956000	0.966250
mfeat_morphological	0.746750	0.723750	0.724000	0.721500	0.697250
mfeat_pixel	0.951750	0.958000	0.971000	0.972250	0.964000
mfeat_zernike	0.825750	0.780750	0.799250	0.771500	0.782750
mofn_3_7_10	1.000000	1.000000	1.000000	0.998034	1.000000
monk1	1.000000	0.999107	1.000000	0.995536	1.000000
monk2	1.000000	0.998750	0.992561	0.988902	0.968079
monk3	0.988842	0.987118	0.986256	0.987980	0.988842
movement_libras	0.880000	0.773667	0.801667	0.813333	0.835667
mushroom	1.000000	1.000000	1.000000	1.000000	1.000000
new_thyroid	0.941270	0.905556	0.921746	0.923492	0.914762
optdigits	0.964103	0.975267	0.986466	0.982049	0.985331
page_blocks	0.919222	0.929403	0.937245	0.941061	0.871124
parity5+5	1.000000	1.000000	1.000000	0.615138	1.000000
penguins	0.997619	0.990640	0.990476	0.989409	0.995238
phoneme	0.894170	0.894812	0.896019	0.878479	0.886315
pima	0.768481	0.754074	0.747519	0.756222	0.706889
prnn_crabs	0.990000	0.940000	0.960000	0.940000	0.987500
prnn_fglass	0.789333	0.724190	0.739333	0.734762	0.727524
prnn_synth	0.854000	0.836000	0.822000	0.848000	0.826000
profb	0.612778	0.627778	0.633333	0.638333	0.568333
ring	0.969834	0.974377	0.974149	0.931072	0.969462
saheart	0.653381	0.672157	0.678279	0.672234	0.582249
satimage	0.884280	0.898247	0.903237	0.886872	0.897283
schizo	0.389861	0.430000	0.384167	0.428889	0.378472
segmentation	0.984199	0.983766	0.986147	0.978139	0.981169
solar_flare_1	0.690778	0.699778	0.707000	0.663667	0.695368
solar_flare_2	0.658073	0.658406	0.657407	0.650708	0.647044
sonar	0.794091	0.855909	0.878864	0.824318	0.824773
soybean	0.970833	0.959105	0.961111	0.968519	0.961111
spambase	0.944675	0.953517	0.954179	0.939200	0.954177
spect	0.782770	0.763002	0.799789	0.808774	0.743235
spectf	0.847265	0.869505	0.872807	0.834056	0.864809
splice	0.956363	0.964333	0.968326	0.954284	0.965821
texture	0.993000	0.979182	0.990909	0.976364	0.987909
threeOf9	0.998182	1.000000	1.000000	0.994545	0.998182
tic_tac_toe	0.979743	0.999254	1.000000	0.930090	0.999600
tokyo1	0.925680	0.921969	0.920095	0.925698	0.908554
twonorm	0.976078	0.973240	0.973918	0.969388	0.969320
vehicle	0.809732	0.758412	0.760054	0.733704	0.753409
vote	0.962014	0.957519	0.950694	0.953635	0.950888
vowel	0.971717	0.901515	0.945455	0.952525	0.976263
waveform_21	0.865730	0.855471	0.860156	0.849969	0.855443

waveform_40	0.867075	0.849184	0.853248	0.846454	0.853220
wdbc	0.958433	0.944345	0.955556	0.945635	0.933929
wine_quality_red	0.413217	0.380638	0.375489	0.431208	0.369281
wine_quality_white	0.427080	0.412426	0.401853	0.390069	0.402562
xd6	1.000000	1.000000	0.999231	1.000000	0.999231
yeast	0.550036	0.535756	0.505806	0.541290	0.524369

Table 27: Detailed results of top-performing algorithms on all 40 DIGEN datasets.

Dataset	EvoFeat	XGBoost	LightGBM	RF	DT
digen1_6265	0.9770	0.9265	0.9275	0.9170	0.8390
digen2_6949	0.9785	0.9780	0.9545	0.7540	0.5925
digen3_769	0.9785	0.9170	0.9255	0.9045	0.8280
digen4_860	0.9865	0.9070	0.8775	0.6670	0.5315
digen5_6949	0.9525	0.9555	0.9560	0.9560	0.9565
digen6_466	0.9745	0.9770	0.9600	0.7910	0.6500
digen7_6949	0.9790	0.9740	0.9770	0.7325	0.5465
digen8_4426	0.9860	0.9555	0.9125	0.8435	0.7020
digen9_7270	0.9080	0.8845	0.8540	0.8910	0.8645
digen10_8322	0.9845	0.8050	0.7990	0.7750	0.5950
digen11_7270	0.9805	0.9830	0.9775	0.9570	0.6520
digen12_8322	0.9885	0.9800	0.9525	0.8215	0.6575
digen13_769	0.8935	0.8440	0.8365	0.8160	0.7085
digen14_769	0.9770	0.9670	0.9175	0.6910	0.4965
digen15_5311	0.9165	0.9300	0.9345	0.9375	0.9075
digen16_5390	0.9790	0.8220	0.8275	0.7625	0.5715
digen17_6949	0.9690	0.9605	0.9440	0.7445	0.6105
digen18_5578	0.9745	0.7575	0.7185	0.6465	0.5585
digen19_7270	0.9825	0.8550	0.8485	0.8490	0.7270
digen20_5191	0.9945	0.9930	0.9945	0.9945	0.9930
digen21_6265	0.9710	0.9590	0.9485	0.9620	0.9570
digen22_2433	0.9860	0.9920	0.9925	0.9885	0.7520
digen23_5191	0.9800	0.9770	0.9580	0.7575	0.5490
digen24_2433	0.9755	0.9590	0.9440	0.8295	0.6300
digen25_2433	0.9820	0.9810	0.9625	0.7775	0.5830
digen26_7270	0.9760	0.9525	0.9495	0.9540	0.9160
digen27_860	0.9690	0.9530	0.8685	0.8670	0.7945
digen28_769	0.9835	0.9705	0.9375	0.8075	0.5720
digen29_8322	0.7390	0.7340	0.7410	0.7365	0.7030
digen30_4426	0.9810	0.9460	0.9525	0.6895	0.5335
digen31_2433	0.9770	0.8195	0.8205	0.8265	0.6770
digen32_5191	0.9810	0.9735	0.9420	0.7855	0.6000
digen33_769	0.9730	0.8690	0.8575	0.8515	0.8045
digen34_769	0.9870	0.9735	0.9200	0.6580	0.5035
digen35_4426	0.9630	0.9635	0.9310	0.8915	0.6950
digen36_466	0.9200	0.7800	0.7775	0.7865	0.6465
digen37_769	0.9775	0.8645	0.8635	0.8815	0.7835
digen38_4426	0.9835	0.9815	0.9240	0.6450	0.5105
digen39_5578	0.9915	0.9015	0.9125	0.8370	0.6370
digen40_5390	0.9750	0.9615	0.9365	0.6955	0.5225