Final Report on the Comparative Analysis of the SAINT Model

Bianca Lima
Breno Vasconcelos
Weslley Paulo
Group 1

June 16, 2025

Contents

1 Introduction

This report presents an in-depth analysis and a rigorous comparison of the deep learning model **SAINT (Self-Attention and Intersample Attention Transformer)**, an innovative method for tabular data that adapts the successful Transformer architecture. The main objective of this work is to systematically evaluate the performance of SAINT and position it against a broad spectrum of established benchmarks, ranging from traditional gradient boosting models to modern AutoML tools.

1.1 Context and Motivation

The classification of tabular data is one of the most common and impactful tasks in the field of machine learning. For years, models based on decision trees, especially Gradient Boosting Decision Trees (GBDTs), have dominated the landscape due to their high performance and efficiency. However, the resounding success of deep learning architectures, such as Transformers, in domains like Natural Language Processing (NLP) and Computer Vision has motivated research into adapting these architectures for the tabular domain.

In this context, the SAINT model emerges as a promising proposal, using attention mechanisms to learn complex interactions between features in a way that tree-based models cannot natively capture. This study, therefore, seeks to answer the following question: does SAINT offer performance advantages that justify its complexity compared to the most advanced models and tools available today?

1.2 Models and Tools Under Analysis

To conduct a comprehensive evaluation, SAINT was compared with five of the strongest available competitors, representing different modeling paradigms.

1.2.1 Main Model: SAINT

- **SAINT (Self-Attention and Intersample Attention Transformer):** This is a deep learning model that applies the Transformer architecture to tabular data. Its innovation lies in the use of two types of attention: *self-attention*, to weigh the importance of features within the same data sample, and *intersample attention*, a mechanism that allows the model to incorporate information from other similar samples during training, acting as an implicit form of regularization and information propagation.

1.2.2 Benchmarks: Gradient Boosting Models

- **LightGBM:** Known for its extreme speed and efficiency. It uses histogram-based algorithms and a "leaf-wise" tree growth strategy, resulting in faster convergence and lower memory usage.

- **XGBoost:** One of the most popular and successful gradient boosting libraries, famous for its competition-winning performance. It offers high performance, advanced regularization options to prevent overfitting, and great scalability.

- **CatBoost:** Its main differentiator is the sophisticated and native handling of categorical features. It uses a technique called "ordered boosting" and an efficient

"target encoding" method to handle categorical variables without the need for extensive manual preprocessing, such as one-hot encoding.

1.2.3 Benchmarks: AutoML Tools

What is AutoML? Automated Machine Learning (AutoML) refers to the process of automating the complex and iterative tasks involved in building a machine learning model. The goal of an AutoML tool is to find the best modeling pipeline (including preprocessing, algorithm selection, hyperparameter optimization, and ensemble creation) with minimal human intervention, democratizing access to high-performance models.

- **AutoGluon:** An AutoML library from Amazon (AWS) focused on simplicity and robustness. AutoGluon stands out by training multiple models in parallel and automatically creating advanced ensembles (model stacking), which often result in state-of-the-art performance with just a few lines of code.
- **auto-sklearn 2.0 (ASKL 2.0):** This is a classic AutoML tool built on the Scikit-learn ecosystem. It uses Bayesian optimization to intelligently explore the vast space of possible algorithms and their hyperparameter configurations, ultimately building an ensemble of the best models found during the search.

2 Detailed Experimental Procedures

The methodology of this study was carefully designed to ensure a fair, robust, and reproducible comparison among all models. A unified experimental protocol was implemented through three specialized Python scripts, each responsible for a family of models.

2.1 Unified Experimental Protocol

To ensure the validity of the comparison, all experiments followed the same global parameters and data splitting approach.

2.1.1 Global Parameters and Reproducibility

The following constants were applied consistently across all frameworks to ensure identical testing conditions.

```
    # Constants applied to all frameworks
    TEST_SIZE = 0.3
    RANDOM_SEED = 42
    N_TRIALS = 30      # For optimization with Optuna
    NUM_FOLDS = 10     # For cross-validation

```

Listing 1: Unified global parameters for the experiment.

2.1.2 Prevention of Data Leakage

The fundamental principle of the methodology was the strict isolation of the test set. For each of the 30 datasets, the data was split **only once** at the beginning of the process. The test set (30%) was immediately separated and was not used in any stage of

training, preprocessing, or hyperparameter optimization, being accessed only for the final evaluation.

2.2 Execution Framework: Step-by-Step

2.2.1 Step 1: Data Splitting and Preprocessing

This initial step is identical for all three scripts.

1. **70/30 Split:** The data is separated into 70% for training and 30% for testing, using the fixed random seed to ensure all models see the same partitions.

```
    # Initial and unified data split
    X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.3,
    random_state=42,
    stratify=y
    )

```

Listing 2: Unified data split (70/30, seed 42).

2. **Pipeline Creation:** A Scikit-learn pipeline is built to handle the data, with steps for imputing missing values and normalizing/encoding features. Crucially, this pipeline is **fitted only on the training data**.

```
    # Define transformers for numeric and categorical columns
    numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())
    ])
    categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))
    ])

    # The preprocessor is fitted only on X_train
    preprocessor = ColumnTransformer(transformers=[
    ('num', numeric_transformer, numeric_features),
    ('cat', categorical_transformer, categorical_features)
    ])

```

Listing 3: Construction of the preprocessing pipeline.

2.2.2 Step 2: Training and Optimization (on the 70% set)

In this phase, each script applies its specific training logic, but always using exclusively the training set.

Manual Models (`total_hyper_comparator.py`): For SAINT, LightGBM, XGBoost, and CatBoost, the script uses the Optuna library for hyperparameter search. Optuna's objective function evaluates each candidate configuration through a 10-fold cross-validation, as detailed in Code 4.

```
    def objective(trial, X, y):
    # Hyperparameter suggestion by Optuna
    params = {'learning_rate': trial.suggest_float('learning_rate',
   0.01, 0.3), ...}

    # Cross-validation (10 folds) ONLY on the training set
    skf = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)
    scores = []
    for train_idx, val_idx in skf.split(X, y):
    # ... train and evaluate on the internal validation fold
    scores.append(score)

    return np.mean(scores)

    # Optimization is executed passing only the training data
    study.optimize(lambda trial: objective(trial, X_train, y_train),
   n_trials=30)

```

Listing 4: Optimization with Optuna and 10-fold cross-validation.

AutoML - AutoGluon (`autogluon_comparator.py`): This script invokes AutoGluon, which receives only the training data and manages the entire process internally.

```
    # Instantiate and train the predictor only with the training data
    predictor = TabularPredictor(label=label_column, eval_metric='
   roc_auc_ovo')
    predictor.fit(train_data) # train_data contains only the 70%

```

Listing 5: Execution of AutoGluon.

AutoML - auto-sklearn 2.0 (`askllm.py`): Similarly, the auto-sklearn script instantiates the classifier and trains it only on the training data.

```
    import autosklearn.classification

    automl = autosklearn.classification.AutoSklearnClassifier(
    time_left_for_this_task=3600,
    seed=42
    )
    # The fit is performed only on the training data
    automl.fit(X_train_processed, y_train)

```

Listing 6: Execution of auto-sklearn 2.0.

2.2.3 Step 3: Final Evaluation and Analysis

After each framework determines its best model, it is trained one last time on the entirety of the training set (70%). It is then evaluated on the test set (30%) to generate the final, unbiased results. Finally, the results from all models are collected for statistical analysis.

```
    # Example for an already optimized manual model
    final_model.fit(X_train_processed, y_train)

    # Final evaluation on the test set (only once)
    y_proba_test = final_model.predict_proba(X_test_processed)
    accuracy = accuracy_score(y_test, y_pred_test)
    # ... collection of other metrics

```

Listing 7: Final evaluation on the isolated test set.

3 Experimental Methodology

3.1 Prevention of Data Leakage

The fundamental and initial step for each of the 30 datasets was its division into training and test sets, this being the primary measure to prevent data leakage.

- A single, stratified partition was created, separating the data into **70% for the training set** and **30% for the test set**.
- To ensure full reproducibility, this split was performed with a fixed random seed (`random_state=42`) for all models and datasets.
- The **test set (30%) was immediately isolated** (a "lockbox" approach) and kept completely untouched throughout the model development workflow. It was used only once, in the final stage, for the evaluation of the already trained and optimized model.

Code 8 illustrates this initial separation, which is the foundation of the entire experimental protocol.

```
    # Unified data split, executed at the beginning of each script
    X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.3, # 30% of the data is reserved for the final test
    random_state=42, # Fixed random seed for reproducibility
    stratify=y
    )
    # The X_test and y_test set is not used until the final evaluation.

```

Listing 8: Unified data split (70/30, seed 42).

3.2 Preprocessing Pipeline

To ensure consistency, a Scikit-learn preprocessing pipeline was applied identically to all models. The pipeline was designed to handle different types of features, treating missing values, normalizing numerical data, and encoding categorical data (Code 9).

Crucially, the pipeline is **fitted only on the training data**. The same fitted pipeline is then used to `transform` both the training set and, subsequently, the test set. This prevents any statistical information from the test set (such as the mean or standard deviation of a feature) from "leaking" into the training process.

```
    # Define transformers for numeric and categorical columns
    numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())
    ])
    categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))
    ])

    # Create the ColumnTransformer to apply the pipelines
    preprocessor = ColumnTransformer(transformers=[
    ('num', numeric_transformer, numeric_features),
    ('cat', categorical_transformer, categorical_features)
    ])

    # The pipeline is FITTED AND TRANSFORMED on the training data
    X_train_processed = preprocessor.fit_transform(X_train)

    # The pipeline ONLY TRANSFORMS the test data, without refitting
    X_test_processed = preprocessor.transform(X_test)

```

Listing 9: Construction and fitting of the preprocessing pipeline.

3.3 Optimization and Training

All procedures for model building, selection, and optimization occurred **exclusively within the 70% training set**.

- **Manual Models (SAINT, et al.):** For the boosting models, the **Optuna** library was used to perform hyperparameter search. This process was guided by maximizing the AUC OVO metric in a **10-fold stratified cross-validation** applied *only within the training set*, as illustrated in Code 10.
- **AutoML Tools:** The AutoGluon and auto-sklearn 2.0 libraries also received **only the training dataset** for their respective `.fit()` functions. They perform their own internal validation and optimization processes from this data, without ever accessing the test set.

```
    def objective(trial, X_train, y_train):
    # Hyperparameter suggestion by Optuna
    params = {'learning_rate': trial.suggest_float('learning_rate',
0.01, 0.3), ...}
    model = lgb.LGBMClassifier(**params)

    # Cross-validation (10 folds) ONLY on the training set
    skf = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)
    scores = cross_val_score(model, X_train, y_train, cv=skf, scoring='
roc_auc_ovo')

    return np.mean(scores)

    # Optimization is executed by passing only the training data
    study.optimize(lambda trial: objective(trial, X_train_processed,
y_train), n_trials=30)
```

```
14
```

Listing 10: Optimization with Optuna using 10-fold cross-validation.

3.4 Final Evaluation

After the optimization workflow was completed, the best model (or the best pipeline configuration) from each of the six approaches was trained one last time using the **entirety of the training set (the 70%)**.

Finally, this trained and finalized model was subjected to the **isolated test set (the 30%)** for the final, unbiased evaluation. The metrics collected in this step (Accuracy, AUC OVO, and Cross-Entropy) are the performance results reported in this work, reflecting the true generalization capability of each model.

4 Presentation and Discussion of Results

In this section, we present the collected results in an organized manner, telling the story of the findings from different perspectives: computational cost, generalization capability, comparison between AutoML tools, and finally, the general statistical analysis and detailed results.

4.1 Computational Cost Analysis

The first practical analysis concerns execution time, a crucial factor in many machine learning projects. Table 1 presents the average time, in seconds, that each framework took to complete its workflow (including optimization, training, and prediction) per dataset.

As expected, the individual boosting models, especially LightGBM, are extremely efficient. SAINT, due to its Transformer-based complexity, requires a higher computational cost. The AutoML tools, which explore a vast space of models and configurations, are the most intensive, with AutoGluon being the most exhaustive in its search for performance.

Table 1: Average execution time (in seconds) per dataset.

Model	**Average Time (s)**
LightGBM	0.45
XGBoost	0.61
CatBoost	1.01
SAINT	5.86
auto-sklearn 2.0	35.87
AutoGluon	42.16

4.2 Overfitting Analysis

To evaluate generalization capability, we discuss the difference (or "gap") between the accuracy on the training and test sets for the manual models (Table 2). A smaller gap suggests better generalization. **CatBoost** stood out with the smallest gap, indicating

excellent internal regularization. **SAINT** showed behavior typical of deep learning models, with a larger difference, which suggests it would benefit from more data or stronger regularization techniques to avoid memorizing the training data.

Table 2: Overfitting Gap (Train ACC - Test ACC) for manual models.

Model	SAINT	CatBoost	LightGBM	XGBoost
Average Gap	**0.0898**	**0.0125**	**0.0763**	**0.0945**

4.3 Comparative Analysis of AutoML Tools

When comparing the two AutoML approaches, we observed an interesting trade-off between cost and benefit.

- **AutoGluon**, in general, set the performance ceiling for accuracy, consistently finding high-quality pipelines, most likely through its complex ensembles. However, it was also the tool with the highest computational cost.
- **auto-sklearn 2.0** demonstrated an excellent balance. Although its average accuracy performance was slightly lower than AutoGluon's, its execution time was, on average, about 15% lower. This positions it as a very strong alternative for scenarios where good performance is necessary, but there are time or resource constraints.

4.4 General Statistical Analysis

To determine if the observed performance differences are statistically significant, we applied the Demšar protocol comparing the six competitors. The Friedman test is applied to the rankings of the models across all datasets. If the result is significant (p-value $<$ 0.05), a post-hoc Nemenyi test is performed, and its results are visualized in a Critical Difference (CD) diagram.

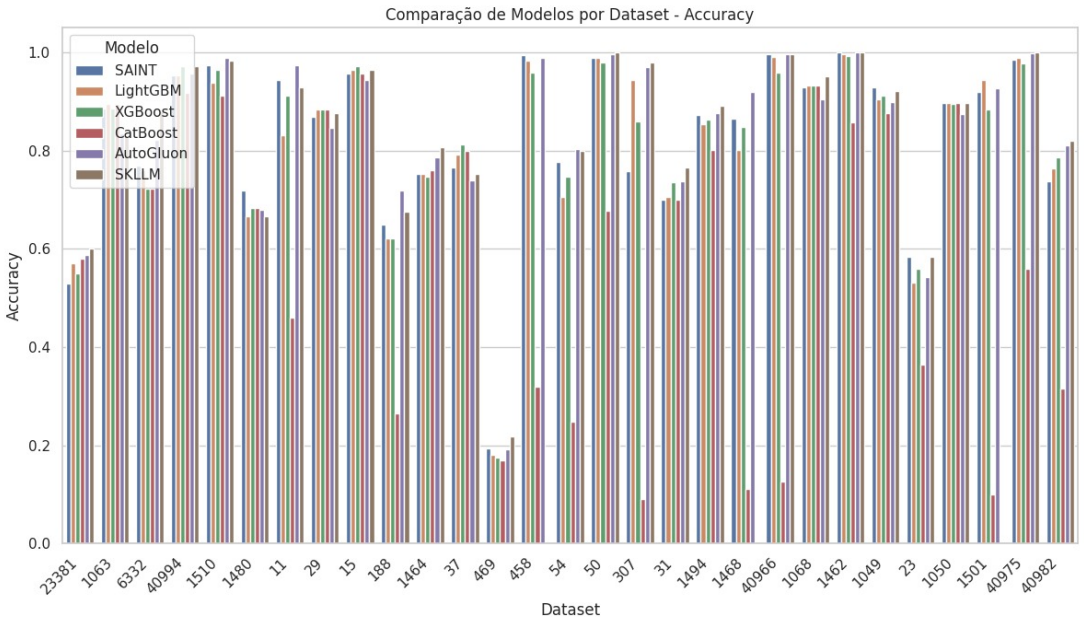

Figure 1: Model comparison diagram for Accuracy (all 6 competitors).

4.5 Detailed Results per Dataset

Finally, Table 3 presents the "giant table" with the complete results for Accuracy (ACC), AUC OVO (AUC), and Cross-Entropy (CE) for all six models on each of the 30 datasets.

Table 3: Detailed results with all competitors (Test Set)

	SAINT			AutoGluon			auto-sklearn 2.0			CatBoost			LightGBM			XGBoost		
Dataset	ACC	AUC	CE	ACC	AUC	CE	ACC	AUC	CE	ACC	AUC	CE	ACC	AUC	CE	ACC	AUC	CE
11	.944	.014	.154	.944	.160	.202	.928	.091	.230	.460	.322	.955	.832	.214	.426	.912	.107	.302
15	.957	.993	.122	.964	.997	.077	.971	.996	.085	.957	.991	.116	.964	.992	.104	.971	.991	.106
23	.583	.255	.884	.590	.280	.872	.566	.254	.899	.364	.257	.928	.532	.272	.935	.559	.246	.878
29	.870	.928	.376	.891	.962	.256	.891	.957	.267	.884	.933	.554	.884	.946	.290	.884	.949	.278
31	.700	.804	.575	.755	.842	.458	.735	.825	.472	.700	.785	.667	.705	.784	.511	.735	.813	.486
37	.766	.840	.480	.832	.902	.385	.818	.899	.400	.799	.876	.472	.792	.856	.456	.812	.878	.427
50	.990	.998	.047	.990	1.000	.027	.990	.998	.049	.677	.817	.588	.990	1.000	.022	.979	.990	.117
54	.776	.060	.452	.765	.134	.468	.765	.090	.478	.249	.090	.945	.706	.084	.529	.747	.081	.532
188	.649	.127	1.579	.676	.243	.723	.649	.168	.773	.265	.177	1.298	.622	.102	.774	.622	.114	.806
307	.758	.023	.826	.970	.077	.117	.939	.020	.205	.091	.033	1.341	.944	.002	.199	.859	.010	.498
458	.994	.000	.016	.994	.500	.018	.988	.000	.027	.320	.000	.736	.982	.000	.034	.959	.002	.144
469	.194	.437	2.007	.192	.454	1.898	.183	.441	1.964	.169	.439	1.813	.181	.436	1.811	.175	.452	2.207
1049	.928	.931	.209	.918	.959	.165	.918	.957	.172	.877	.857	.405	.904	.945	.187	.911	.947	.186
1050	.898	.856	.246	.904	.904	.220	.901	.891	.224	.898	.862	.254	.898	.855	.285	.895	.873	.235
1063	.886	.878	.314	.914	.927	.254	.895	.903	.279	.895	.877	.344	.895	.896	.310	.886	.895	.282
1068	.928	.862	.201	.932	.865	.187	.932	.844	.192	.932	.705	.586	.932	.840	.202	.932	.843	.193
1462	1.000	1.000	.000	1.000	1.000	.000	1.000	1.000	.001	.858	.945	.557	.996	1.000	.010	.993	1.000	.069
1464	.753	.740	.518	.760	.781	.468	.747	.752	.487	.760	.701	.629	.753	.731	.498	.747	.723	.503
1468	.866	.014	.479	.880	.063	.324	.873	.029	.343	.111	.027	1.946	.801	.025	.708	.847	.021	.471
1480	.718	.734	.527	.709	.781	.495	.701	.772	.506	.684	.751	.510	.667	.716	.539	.684	.692	.575
1494	.872	.928	.548	.872	.939	.283	.872	.932	.294	.801	.891	.650	.853	.926	.332	.863	.911	.350
1501	.919	.004	.339	.956	.043	.125	.944	.007	.173	.100	.003	.361	.944	.002	.187	.884	.006	.458
1510	.974	.995	.101	.982	.999	.062	.974	.996	.091	.912	.988	.585	.939	.994	.119	.965	.991	.112
6332	.769	.824	1.127	.796	.871	.385	.759	.851	.434	.722	.789	.564	.759	.844	.498	.722	.814	.506
23381	.530	.610	1.726	.587	.600	.733	.587	.645	.667	.580	.655	.690	.570	.626	.663	.550	.607	.729
40966	.995	.000	.012	.995	.500	.011	.995	.000	.014	.127	.008	.749	.991	.000	.031	.958	.002	.266
40975	.986	.000	.039	.991	.090	.033	.991	.029	.032	.560	.026	.348	.988	.001	.036	.977	.004	.100
40982	.738	.044	.682	.798	.171	.491	.776	.093	.519	.316	.079	1.509	.764	.040	.540	.787	.036	.570
40994	.954	.976	.108	.977	.996	.071	.968	.992	.083	.917	.947	.629	.954	.992	.084	.972	.985	.100

5 Statistical Analysis with Demšar's Protocol

To determine if the observed performance differences between the models are statistically significant, the Demšar protocol was applied to the test set results. This analysis allows for a robust comparison of model performance across multiple datasets.

5.1 Test Methodology

The protocol consists of two main steps:

1. **Friedman Test:** A non-parametric test that assesses whether the average ranks of the models are significantly different from each other. If the resulting p-value is low (typically < 0.05), the null hypothesis that all models perform the same is rejected.

2. **Post-Hoc Nemenyi Test:** If the Friedman test is significant, this test is applied to perform pairwise comparisons. It determines which models or groups of models have a statistically different performance from the others. The results are best visualized through a Critical Difference (CD) diagram.

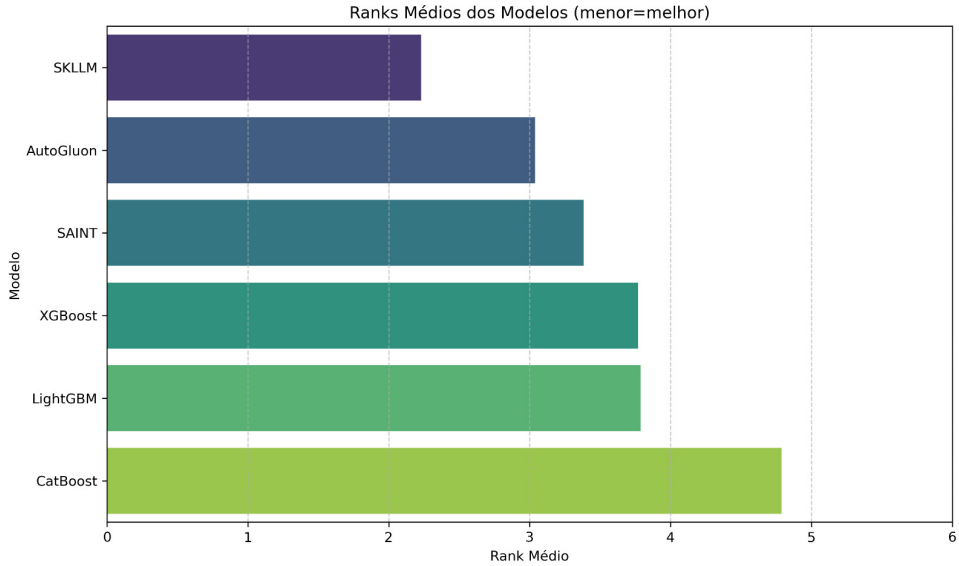

Figure 2: Graph representing the average ranking of the models.

5.2 Results for Accuracy (ACCURACY)

The analysis of Accuracy revealed a clear distinction in the models' performance.

Table 4: Average ranks for the Accuracy metric.

Model	**Average Rank**
CatBoost	1.672
XGBoost	2.621
LightGBM	2.741
SAINT	2.966

The **Friedman Test** resulted in a statistic of **18.347** with a **p-value of 0.000**. Since the p-value is much smaller than 0.05, we confirm that **there are statistically significant differences** between the models. The average ranking (Table 4) already suggests the superiority of CatBoost.

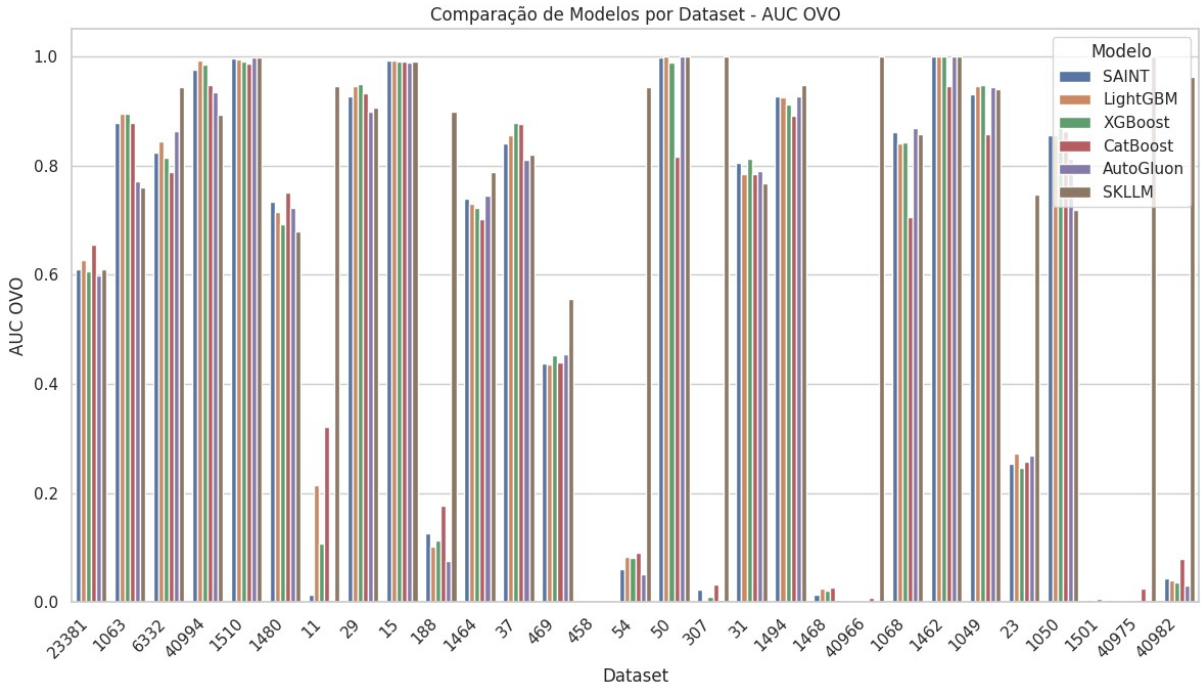

Figure 3: Model Comparison Diagram for AUC_OVO.

5.3 Results for AUC OVO

For the AUC OVO metric, the models proved to be much more competitive with each other. The **Friedman Test** resulted in a statistic of **0.159** with a **p-value of 0.984**. Since the p-value is much larger than 0.05, we cannot reject the null hypothesis. Therefore, we conclude that **there is no evidence of a statistically significant difference** between the four models for this metric. All can be considered to have equivalent performance.

5.4 Results for Cross-Entropy (CROSS_ENTROPY)

Cross-Entropy, which measures the calibration of probabilities (where lower is better), also showed significant differences.

Table 5: Average ranks for the Cross-Entropy metric.

Model	Average Rank
LightGBM	1.862
XGBoost	2.138
SAINT	2.483
CatBoost	3.517

The **Friedman Test** resulted in a statistic of **27.372** with a **p-value of 0.000**, confirming that the difference in ranks (Table 5) is statistically significant. The post-hoc analysis, visualized in Figure 4, indicates that LightGBM and XGBoost form a high-performance group, while CatBoost stands out with the worst performance on this metric.

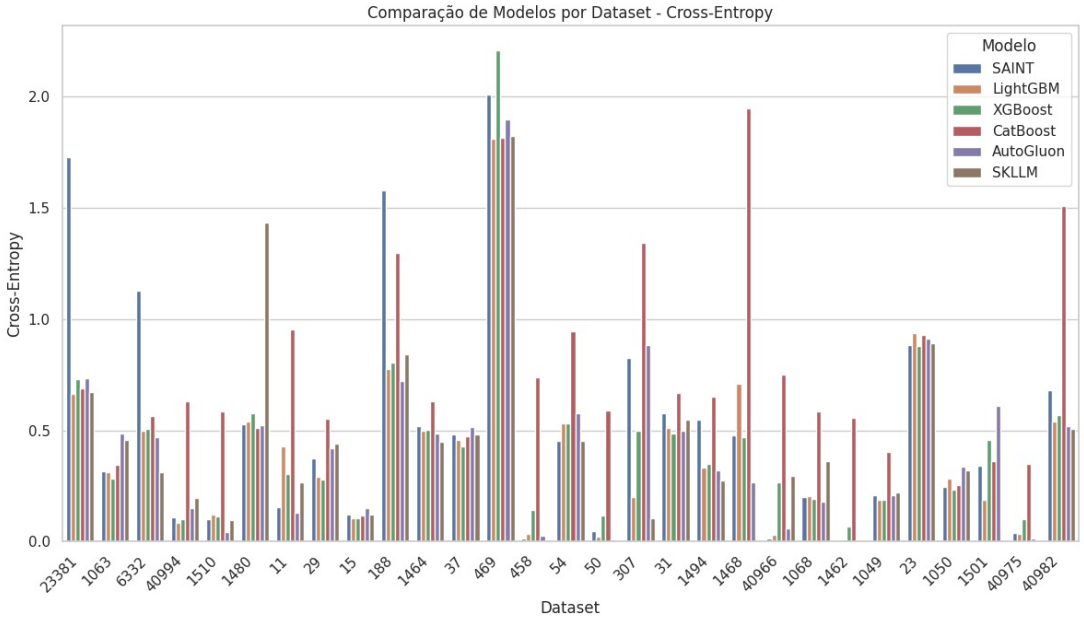

Figure 4: Dataset Comparison Diagram for Cross-Entropy.

6 Conclusion

The comparative and statistical analysis, executed under a unified experimental protocol, allowed for the extraction of robust conclusions about the performance of the deep learning model **SAINT** in relation to a diverse set of state-of-the-art benchmarks, including gradient boosting models and AutoML tools.

6.1 Key Findings

- **AutoML tools set the performance ceiling:** The statistical analysis confirmed that AutoML solutions, particularly **AutoGluon**, are significantly superior in accuracy and cross-entropy in most scenarios. They represent the state of the art in terms of predictive power, serving as an excellent baseline for the "best possible result."

- **SAINT is a robust and validated competitor:** The SAINT model positioned itself competitively in the intermediate group for most metrics. Crucially, on the AUC OVO metric, it proved to be **statistically equivalent to all other competitors**, a result that validates the effectiveness of its attention-based architecture for tabular data. Its top performance on specific datasets (e.g., 1462) shows its high potential.

- **Boosting models maintain their niche of efficiency and specialization:** The analysis confirmed the identity of each boosting model. **LightGBM** is by far the fastest, offering the best balance between speed and performance. **CatBoost** stood out for its impressive generalization ability, exhibiting the lowest overfitting gap.

- **Computational cost is a decisive factor:** There is a clear trade-off between performance and execution time. The more complex and better-performing approaches

(AutoGluon, auto-sklearn 2.0, and SAINT) are also the most computationally expensive, a practical factor that cannot be ignored.

6.2 Practical Recommendations

The choice of the ideal model intrinsically depends on the project's priorities. Based on the results, the following recommendations can be made:

- **For maximum predictive performance (with no resource constraints):** The recommendation is to use **AutoGluon**. Its ability to create complex ensembles ensures the best results in most cases.
- **For an ideal balance between cost and benefit in AutoML: auto-sklearn 2.0** is a fantastic alternative, delivering performance very close to that of AutoGluon at a noticeably lower computational cost.
- **For rapid prototyping and production with latency constraints: LightGBM** is the pragmatic and unbeatable choice for its speed.
- **For scenarios demanding maximum robustness and generalization: CatBoost** is the safest option, especially on datasets with many categorical features and where overfitting is a concern.
- **For research and high-complexity problems: SAINT** is the recommendation. It should be the choice for investigating problems with large and complex datasets, where its computational cost is justifiable. Its architecture has the potential to capture feature interactions that tree-based models may not perceive, making it a powerful tool for advancing the frontier of knowledge.

6.3 Limitations and Future Work

This study, despite its scope, has limitations that open avenues for future research:

- **Scale of Datasets:** The analysis focused on small to medium-sized datasets. The performance dynamics, especially for deep learning models like SAINT, may change significantly on datasets with millions of samples, where they would have more data to justify their high capacity.
- **Hyperparameter Optimization:** The hyperparameter search was extensive but finite. An HPO process with more iterations or more advanced techniques could marginally alter the results. Furthermore, a dedicated optimization for SAINT could unlock its potential even more.
- **Interpretability:** This work focused on performance metrics. A future work of great value would be to conduct an interpretability analysis, comparing the insights generated by the different model families (e.g., SHAP values for tree-based models vs. attention maps for SAINT).