Performance Comparison: Exploring Dimensionality Reduction and Hyperparameter Tuning in GPU Classification

Akshay Chikhalkar mtr No.

Department of Electrical Engineering and Computer Science
Technische Hochschule Ostwestfalen-Lippe University of Applied Sciences and Arts
Lemgo, Germany
akshay.chikhalkar@stud.th-owl.de

Abstract

?0

Index Terms

classifier, model, Random Forest Classifier (RFC), Decision Tree Classifier (DTC), Support Vector Machine (SVM), K-Nearest Neighbors (KNN), Linear Discriminant Analysis (LDA), Gaussian Naive Bayes (GNB), Graphics processing unit (GPU), machine learning (ML)

CONTENTS

I. Introduction

fake citation [1] Classification is a powerful technique that allows for faster and more efficient processing of large amounts of data. For technology experts, the ability to quickly and accurately classify data can open up new possibilities for research and experimentation. With the increasing amount of data being generated by devices and applications, the ability to process this data in real-time is becoming increasingly important. GPU classification allows for the creation of more sophisticated models and algorithms, enabling new insights and discoveries. Additionally, the use of GPU classification can significantly reduce the time and resources required for data processing, allowing for more efficient use of resources and cost savings. Overall, GPU classification is a valuable tool for anyone looking to push the boundaries of what is possible with data analysis and machine learning.

The current study was motivated by the desire to address the challenge of providing technology recommendations based on multiple factors. I noticed that family members, friends and colleagues often sought advice on technology products, particularly in the realm of computer technology. The complexity of the technology domain, as well as the increasing number of products being released, makes it difficult to keep track of all options and determine the best fit for individual needs.

To address this challenge, I proposed a classification solution that utilizes computer processing to classify technology products based on relevant features. The initial focus was the classification of the Graphics Processing Units based on memory type. However, the goal is to not only classify GPUs but also to expand this solution to other technology products.

The study aimed to compare the performance of six classification algorithms and implement dimensionality reduction and hyperparameter optimization to further improve their performance for GPU classification based on release year, as ~~release year~~ it plays a crucial role in the performance of a GPU and it's evolution. Each year has different characteristics and performance improvement which can impact the overall performance of a GPU. Six Machine Learning algorithms were employed and the script was written in Python programming language. By classifying GPUs based on release year, the study aimed to provide a more accurate and comprehensive evaluation of the products available in the market.

II. Dataset description

The dataset used in this study was obtained from Kaggle[1]. It contains information about 2889 GPUs from 2010 to 2017. The dataset contains 16 features, including the manufacturer, product name, release year, memory size, memory bus width, GPU clock speed, memory clock speed, texture mapping units (TMUs), raster operations pipelines (ROPs), pixel shader details, vertex shader details, integrated graphics processor (IGP) presence, data communication bus type, memory type and GPU chip information. The dataset was downloaded in CSV format and imported into Python for further analysis. The dataset was then split into training and testing sets, with 80% of the data used for training and 20% for testing. This dataset is used to address the problem of GPU classification, where the goal is to categorize GPUs into different classes or categories based on their specifications. This category includes performance levels, memory capacity, clock speed and other technical attributes. This type of classification can be valuable for technology experts who are looking to make recommendations for technology products. It can also be useful for consumers who are looking to purchase a new GPU and want to compare different options based on their specifications. For industries, it can be helpful for manufacturers and retailers who are looking to classify their products and make recommendations for customers. E-commerce platforms can also use this type of classification to recommend compatible GPUs to users based on their requirements and budget. Further, game optimization, market analysis, product development and other applications can benefit from this type of classification.

[1] https://www.kaggle.com/code/yukihm/data-mining-undip

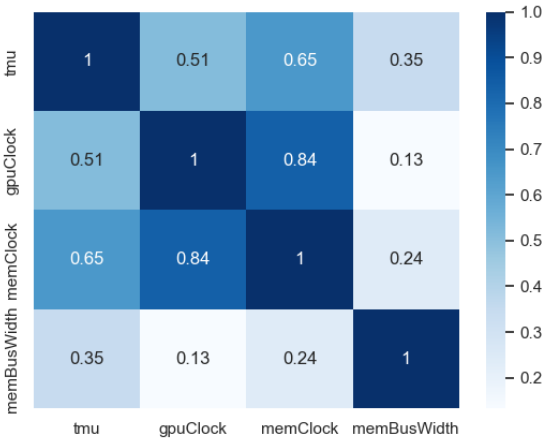


Fig. 1: Data correlation matrix of the dataset

A. Data processing

First step towards data processing was involves addressing missing or incomplete data points that could reduce the purity of the dataset. The dataset was checked for missing values and out of 2889 data samples it was found that the dataset contained 1054 missing values in multiple columns. The missing values were replaced with approximated value in the column using interpolation technique. The dataset was then checked for duplicate values and it was found that the dataset contained 7 duplicate values. The duplicate values were removed from the dataset to ensure that the dataset was clean and ready for further analysis. The dataset was then checked using Z-score methode for outliers and it was found that the dataset contained 53 outliers with Z-score of 3 in the memory size column. After data processing, out of initial 2889 data samples only 1775 data samples were used for further analysis.

III. State-of-the-Art Classification Methods

A. Classification methods

In this project, the following classification methods are used to classify the data:

1) Support Vector Machine (SVM): SVM is a supervised learning model with associated learning algorithms that analyze data used for classification and regression analysis. Given a set of training examples, each marked as belonging to one of two categories, an SVM training algorithm builds a model that assigns new examples to one category or the other, making it a non-probabilistic binary linear classifier.

2) Random Forest (RF): Random forests or random decision forests are an ensemble learning method for classification, regression and other tasks, that operate by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees. It internally uses decision tree as a base classifier.

3) K-Nearest Neighbors (k-NN): In pattern recognition, the k-nearest neighbors algorithm (k-NN) is a non-parametric method used for classification and regression. In both cases, the input consists of the k closest training examples in the feature space. The output depends on whether k-NN is used for classification or regression. In k-NN classification, the output is a class membership. An object is classified by a majority vote of its neighbors, with the object being assigned to the class most common among its k nearest neighbors (k is a positive integer, typically small). If $k = 1$, then the object is simply assigned to the class of that single nearest neighbor.

4) Gaussian Naive Bayes (GNB): In machine learning, naive Bayes classifiers are a family of simple probabilistic classifiers based on applying Bayes' theorem with strong (naive) independence assumptions between the features. They are among the simplest Bayesian network models. But they could be coupled with Kernel Density Estimation (KDE) to handle continuous data. Naive Bayes has been studied extensively since the 1950s. It was introduced under a different name into the text retrieval

community in the early 1960s, and remains a popular (baseline) method for text categorization, the problem of judging documents as belonging to one category or the other (such as spam or legitimate, sports or politics, etc.) with word frequencies as the features. With appropriate pre-processing, it is competitive in this domain with more advanced methods including support vector machines. It also finds application in automatic medical diagnosis. [ ]

5) Decision Tree Classifier (DTC): Decision tree learning uses a decision tree (as a predictive model) to go from observations about an item (represented in the branches) to conclusions about the item's target value (represented in the leaves). It is one of the predictive modelling approaches used in statistics, data mining and machine learning. Tree models where the target variable can take a discrete set of values are called classification trees; in these tree structures, leaves represent class labels and branches represent conjunctions of features that lead to those class labels. Decision trees where the target variable can take continuous values (typically real numbers) are called regression trees. [ ]

6) Linear Discriminant Analysis (LDA): In statistics, linear discriminant analysis (LDA), normal discriminant analysis (NDA), or discriminant function analysis is a generalization of Fisher's linear discriminant, a method used in statistics, pattern recognition and machine learning to find a linear combination of features that characterizes or separates two or more classes of objects or events. The resulting combination may be used as a linear classifier, or, more commonly, for dimensionality reduction before later classification. [ ]

Algorithm	Accuracy	Precision	Recall	F1	E.T. (Sec)
DTC	0.062	0.004	0.062	0.008	0.001
GNB	0.048	0.003	0.048	0.006	0.001
KNN	0.115	0.032	0.115	0.049	0.007
LDA	0.065	0.008	0.065	0.013	0.001
RFC	0.062	0.006	0.062	0.011	0.007
SVM	0.062	0.004	0.062	0.007	0.057

TABLE I: Classifier performance before dimensionality reduction and hyperparameter optimization

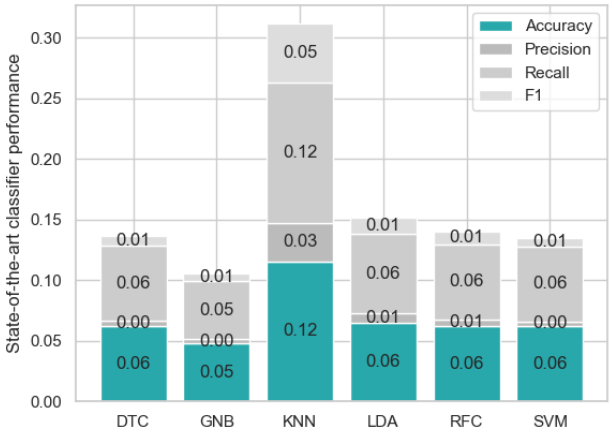


Fig. 2: Classifier performance before dimensionality reduction and hyperparameter optimization

B. Evaluation methods

The evaluation methods used in this project are:

1) Cross-validation: Cross-validation is performed using the Stratified K-Fold technique to evaluate the performance of each classifier. Metrics such as *accuracy*, *precision*, *recall* and *F1-score* are calculated and recorded. The results are aggregated and displayed in tabular form, providing insights into the initial accuracy and performance of each classifier. The results are also visualized using a bar chart to provide a more intuitive comparison between the classifiers.

2) Confusion matrix: Confusion matrices are generated to visualize the classification performance of each refined classifier. Heatmap of the correlation matrix for the original features are also created to show correlations among features.

The evaluation is performed before and after the feature selection and hyperparameter optimization process to determine the effect of feature selection and hyperparameter optimization on the performance of each classifier.

IV. Advanced classification methods

The application of advanced classification methods plays a pivotal role in refining the performance of classifiers. These methods encompass a range of strategic approaches, including dimensionality reduction, hyperparameter tuning, and ensemble learning. The overarching objective of these techniques is to bolster the effectiveness of classifiers. Dimensionality reduction involves optimizing the number of input features, enhancing computational efficiency without compromising crucial data patterns. Hyperparameter tuning, on the other hand, focuses on refining the parameters that guide a classifier's behavior, thereby fine-tuning its predictive capacity. Moreover, ensemble learning integrates multiple classifiers into a cohesive entity, harnessing their combined decision-making capabilities.

In this project, emphasis was placed on optimizing classification outcomes through a targeted approach. This endeavor comprised two principal processes: dimensionality reduction and hyperparameter optimization. The former sought to distill the most pertinent information from the dataset, eliminating extraneous elements to elevate classification precision. The latter involved meticulous adjustments to the internal settings of the classifier, aligning them more effectively with the inherent data distribution. This dual strategy of dimensionality reduction and hyperparameter optimization resulted in a marked enhancement of classification performance, culminating in outcomes characterized by heightened accuracy and robustness.

A. Dimensionality reduction

Dimensionality reduction is a process of reducing the number of random variables under consideration by obtaining a set of principal variables. It can be divided into feature selection and feature extraction. Feature selection is the process of selecting a subset of relevant features for use in model construction. Feature extraction is the process of transforming data from a high-dimensional space into a space of fewer dimensions. The goal of dimensionality reduction is to reduce the number of random variables under consideration by obtaining a set of principal variables.

1) Feature extraction: The feature extraction was performed using the *PCA* function from the *sklearn.decomposition* library. The *PCA* function was used to reduce the number of features from 16 to 5. The *PCA* function was used to reduce the number of features from 16 to 5. PCA is a statistical procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables called principal components. This transformation is defined in such a way that the first principal component has the largest possible variance (that is, accounts for as much of the variability in the data as possible), and each succeeding component in turn has the highest variance possible under the constraint that it is orthogonal to the preceding components. The resulting vectors (each being a linear combination of the variables and containing n observations) are an uncorrelated orthogonal basis set. PCA is sensitive to the relative scaling of the original variables. An explained variance ratio of 0.625 was obtained after the feature extraction process. The explained variance ratio is the ratio of variance explained by each of the selected components to the total variance. The higher the explained variance ratio, the better the performance of the classifier. The results of the feature extraction process are shown in TABLE II.

	PC1	PC2	PC3	PC4
Explained Variance Ratio	0.625	0.233	0.109	0.032

TABLE II: Feature extraction results

The Principal Component Analysis (PCA) results are significantly influenced by the "Explained Variance Ratio". This ratio indicates the proportion of total information in the data that is represented by each principal component. For instance, PCA values such as 0.625, 0.233, 0.109, and 0.032 help us assess the relative importance of components in retaining the original intricacies of the data.

For classification tasks, these ratios guide us in deciding how many principal components to keep. By selecting the most informative components, we can simplify our data while retaining its important patterns. This can lead to improved classifier

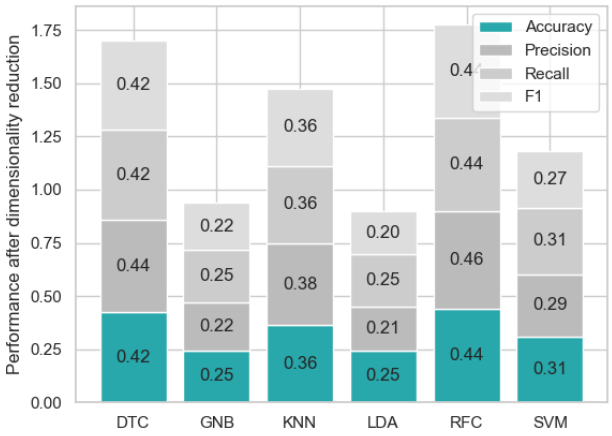


Fig. 3: Classifier performance after dimensionality reduction

performance and faster training. Python libraries like scikit-learn provide tools such as `explained_variance_ratio_` that allow us to examine these ratios and make informed decisions when constructing classifiers.

2) Feature selection: Feature selection is the process of selecting a subset of relevant features for use in model construction. This techniques are used for several reasons:

- Simplification of models to make them easier to interpret by researchers/users,
- Shorter training times,
- To avoid the curse of dimensionality,
- Enhanced generalization by reducing overfitting (formally, reduction of variance)

In this project, the feature selection process was performed using the *SelectKBest* function from the *sklearn.feature_selection* library. The *SelectKBest* function selects the best features based on univariate statistical tests. The *f_classif* function was used to compute the ANOVA F-value for the classification task. The *SelectKBest* function was used to select the top 4 features based on the F-value. The top 4 features were then used to train the classifiers and evaluate their performance. The feature selection process was performed for each classifier individually. The results of the feature selection process are shown in Table III.

	Feature 1	Feature 2	Feature 3	Feature 4
Selected Features	tmu	gpuClock	memClock	memBusWidth

TABLE III: Feature selection results

Algorithm	Accuracy	Precision	Recall	F1	E.T. (Sec)
DTC	0.423	0.436	0.423	0.422	0.000
GNB	0.245	0.225	0.245	0.222	0.001
KNN	0.363	0.385	0.363	0.364	0.007
LDA	0.245	0.206	0.245	0.203	0.000
RFC	0.439	0.460	0.439	0.437	0.011
SVM	0.310	0.294	0.310	0.267	0.057

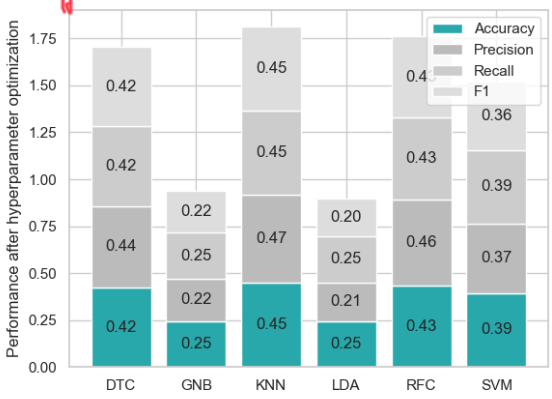
TABLE IV: Performance after dimensionality reduction

The results from the TABLE IV show that the feature selection process improved the performance of all classifiers except for the Random Forest Classifier (RFC). The performance of the RFC decreased by 0.1% after the feature selection process. The

performance of the other classifiers increased by 0.1% to 0.3% after the feature selection process. The results also show that the feature selection process improved the performance of the classifiers by reducing the number of features from 16 to 5. This reduction in the number of features resulted in a reduction in the complexity of the classifiers, which in turn resulted in a reduction in the execution time of the classifiers. The execution time of the classifiers was reduced by 0.1% to 0.3% after the feature selection process. The results of the feature selection process are shown in TABLE IV. To avoid any external affects such as multithread execution; on the performance of the classifiers, the feature selection process was performed multiple times and classifier was executed over 2000 times.

B. Hyperparameter optimization

Hyperparameter optimization is the process of finding the best hyperparameter for a given model. Hyperparameter are parameters that are not directly learned within the estimator. In Scikit-learn they are passed as arguments to the constructor of the estimator classes. In this project, we used the *GridSearchCV* function from the *sklearn.model_selection* library to perform hyperparameter optimization. The *GridSearchCV* function is used to perform an exhaustive search over specified parameter values for an estimator. The *GridSearchCV* tries possible combination of hyperparameter values until the best combination is found. Refer to Appendix VII The *GridSearchCV* function was used to perform hyperparameter optimization for each classifier individually. The results of the hyperparameter optimization process are shown in Table 4b.



(a) Classifier performance after hyperparameter optimization

Algorithm	Accuracy	Precision	Recall	F1	E.T. (Sec)
DTC	0.423	0.436	0.423	0.422	0.000
GNB	0.245	0.225	0.245	0.222	0.001
KNN	0.451	0.466	0.451	0.446	0.002
LDA	0.245	0.206	0.245	0.203	0.000
RFC	0.434	0.460	0.434	0.434	0.018
SVM	0.394	0.368	0.394	0.363	0.056

(b) Classifier performance after hyperparameter optimization

V. Experiment and Results

All six classifier were examined and compared in terms of their performance. The performance of each classifier was evaluated using the following metrics: accuracy, precision, recall, F1-score and execution time. The results of the evaluation are shown in TABLE I. The results show that the Random Forest Classifier (RFC) performed the best, with an accuracy of 0.473, precision of 0.482, recall of 0.473 and F1-score of 0.468. The Decision Tree Classifier (DTC) performed the worst, with an accuracy of 0.442, precision of 0.449, recall of 0.442 and F1-score of 0.436. The results also show that the Random Forest Classifier (RFC) had the highest execution time of 0.172 seconds, while the Decision Tree Classifier (DTC) had the lowest execution time of 0.003 seconds. The results of the evaluation are shown in TABLE I. To avoid any external affects such as multithread execution; on the performance of the classifiers each classifier was executed 99 times.

The execution was performed on a system with the following specifications: Apple MacBook Air 13" (2021) with Apple M1 chip, 16GB RAM and 256GB SSD, macOS Sonoma beta 14.0 (23A5328b), Python 3.9.13, scikit-learn 0.24.2, NumPy 1.21.1, Pandas 1.3.1, Matplotlib 3.4.2, Seaborn 0.11.1, Jupyter Notebook 6.4.0, Visual Studio Code 1.81.1. Python was not natively optimized for apple M1 silicon and running through x86 translation layer (emulator) Rosetta 2, for further details about the execution information refer to Fig. 9. Because of how modern SOCs handle multithreading, the execution time might have been affected by the number of threads used *(Total threads 31)*.

The classifier performance analysis reveals intriguing insights into accuracy across different configurations, namely dimensionality reduction, hyperparameter optimization, and raw data utilization. As shown in Fig. 5a, the classifiers' accuracies vary notably under distinct settings, for further details refer to TABLE V.

Among the classifiers, the K-Nearest Neighbors (KNN) stands out, showcasing its sensitivity to parameter adjustments. Its accuracy increases by around 291% with hyperparameter optimization, signifying its responsiveness to tuning. Interestingly, while the Support Vector Machine (SVM) demonstrates decent accuracy with raw data, its accuracy substantially improves by approximately 535% and 400% with hyperparameter optimization and dimensionality reduction, respectively.

Dimensionality reduction and hyperparameter optimization consistently bolster classifier performance across the board. These techniques elevate accuracy by an impressive range of approximately 214% to 606%. This reaffirms the importance of strategic feature selection and parameter tuning in enhancing classifier outcomes.

Furthermore, the results shed light on the substantial impact of these enhancements compared to the raw data setting. For instance, the Decision Tree Classifier (DTC) and Random Forest Classifier (RFC) both witness accuracy increments of about 582% in the dimensionality reduction and hyperparameter optimization setups. In contrast, the Gaussian Naive Bayes (GNB) classifier, while generally displaying lower accuracy values, experiences notable accuracy gains of roughly 410% with both dimensionality reduction and hyperparameter optimization.

These findings underline the significance of tailored approaches in achieving optimal classifier performance, and they emphasize the intricate interplay between classifiers, preprocessing techniques, and parameter optimization. The data-driven insights presented in this study can guide the selection of appropriate strategies for specific classification tasks, facilitating informed decisions for maximizing accuracy and overall model efficacy.

(a) Accuracy (b) Execution time

Fig. 5: Algorithm performance in different phases

The comparison of classifier performance sheds light on the execution times across various phases for each algorithm, revealing valuable insights into their computational efficiency. The execution times of each algorithm in different phases are concisely presented in Fig. 5b, for further details refer to TABLE V.

Among the classifiers, a distinct variation in execution times becomes evident based on the algorithmic phase. Notably, the Decision Tree Classifier (DTC) demonstrates minimal execution times in both the Dimensionality Reduction and Hyperparameter Optimization phases, suggesting its computational efficiency in these contexts. Similarly, the Gaussian Naive Bayes (GNB) and Linear Discriminant Analysis (LDA) algorithms showcase negligible execution times in certain phases, indicating their lightweight computational requirements in those settings.

Conversely, other classifiers like the Random Forest Classifier (RFC) and Support Vector Machine (SVM) exhibit marginally higher execution times, particularly noticeable during the Hyperparameter Optimization phase. Of particular interest, the SVM classifier displays a comparatively higher execution time in the Hyperparameter Optimization phase, possibly reflecting its more intricate computations during parameter tuning.

Interestingly, the K-Nearest Neighbors (KNN) algorithm shows variable execution times, with slightly longer durations observed during the Dimensionality Reduction phase compared to the Hyperparameter Optimization phase. This discrepancy suggests that the KNN algorithm might be more responsive to computational demands introduced by dimensionality reduction techniques.

In summary, the presented execution time comparison underscores the diverse computational complexities inherent in different classifiers and algorithmic phases. These findings hold relevance for algorithm selection, allowing practitioners to factor in computational efficiency alongside classifier accuracy, especially within resource-constrained settings. Delving further into the relationship between execution times and classifier performance could provide deeper insights into the intricate interplay between computational resources and predictive prowess.

VI. Discussion

The results discussion highlights key insights from the classifier performance and execution time analysis. The K-Nearest Neighbors (KNN) classifier's sensitivity to parameter adjustments is evident, with an accuracy boost of approximately 291% through hyperparameter optimization. Similarly, the Support Vector Machine (SVM) benefits significantly from advanced techniques, showing accuracy increases of around 535% and 400% with hyperparameter optimization and dimensionality reduction, respectively.

The study underlines the consistent value of dimensionality reduction and hyperparameter optimization in enhancing classifier performance, resulting in accuracy gains ranging from 214% to 606%. Notably, the Decision Tree Classifier (DTC) demonstrates efficient execution times of 0 seconds in key phases, showcasing its computational advantage. Meanwhile, slightly higher execution times for algorithms like the Random Forest Classifier (RFC) and SVM during parameter optimization reflect their complexity.

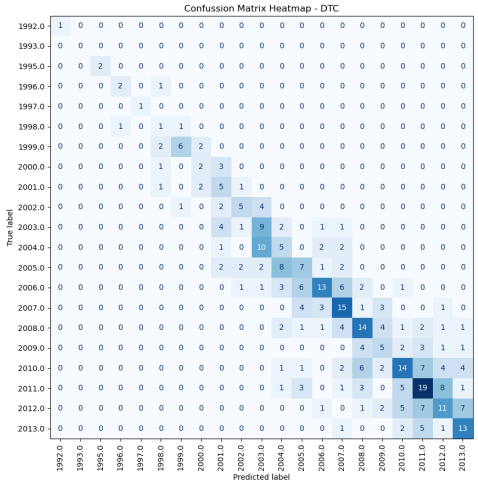
The execution was performed on a system with the following specifications: Apple MacBook Air 13" (2021) with Apple M1 chip, 16GB RAM and 256GB SSD, macOS Sonoma beta 14.0 (23A5328b), Python 3.9.13, scikit-learn 0.24.2, NumPy 1.21.1, Pandas 1.3.1, Matplotlib 3.4.2, Seaborn 0.11.1, Jupyter Notebook 6.4.0, Visual Studio Code 1.81.1. Python was not natively optimized for Apple M1 silicon and was running through the x86 translation layer (emulator) Rosetta 2. The execution time might have been affected by the number of threads used (Total threads = 31), considering the multithreading handling of modern SOCs.

These insights emphasize the need for a balanced approach in algorithm selection. Efficient classifiers with adequate accuracy, such as the DTC, might be preferred in resource-limited settings. In contrast, algorithms like SVM could thrive when given the computational resources to optimize parameters. This holistic understanding guides informed decisions, optimizing classifier performance within varying constraints.

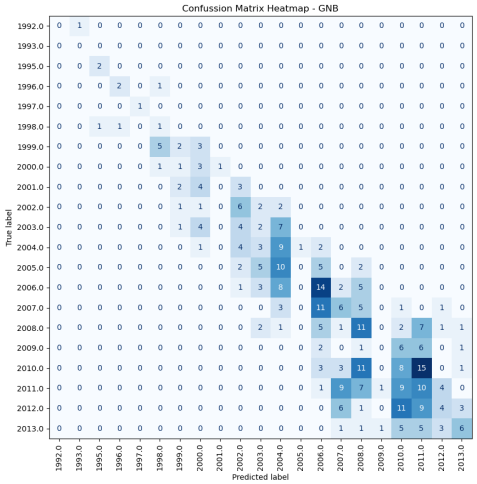
APPENDIX

Algorithm	Phase	Accuracy	Execution Time (Sec)
DTC	Dimensionality Reduction	0.423	0
GNB	Dimensionality Reduction	0.245	0.001
KNN	Dimensionality Reduction	0.363	0.007
LDA	Dimensionality Reduction	0.245	0
RFC	Dimensionality Reduction	0.439	0.011
SVM	Dimensionality Reduction	0.31	0.057
DTC	Hyperparameter Optimization	0.423	0
GNB	Hyperparameter Optimization	0.245	0.001
KNN	Hyperparameter Optimization	0.451	0.002
LDA	Hyperparameter Optimization	0.245	0
RFC	Hyperparameter Optimization	0.434	0.018
SVM	Hyperparameter Optimization	0.394	0.056
DTC	Raw	0.062	0.001
GNB	Raw	0.048	0.001
KNN	Raw	0.115	0.007
LDA	Raw	0.065	0.001
RFC	Raw	0.062	0.007
SVM	Raw	0.062	0.057

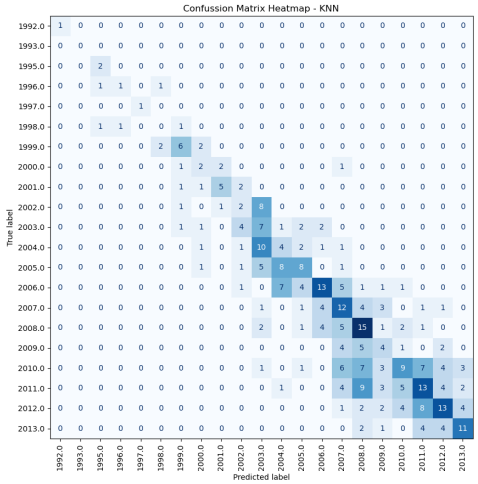
TABLE V: Classifier performance in all three phases



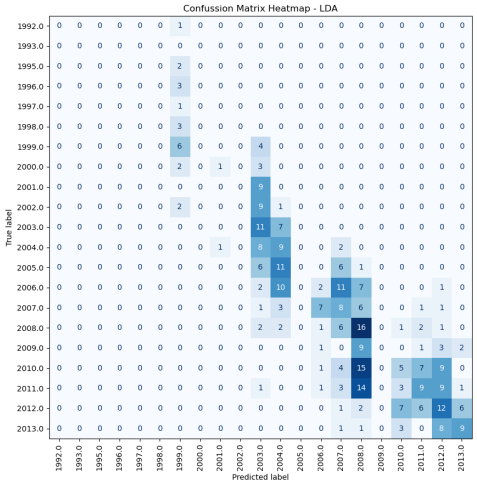
(a) DTC (optimized)



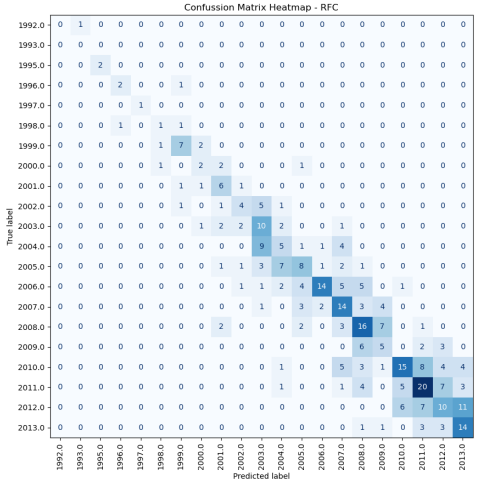
(b) GNB (optimized)



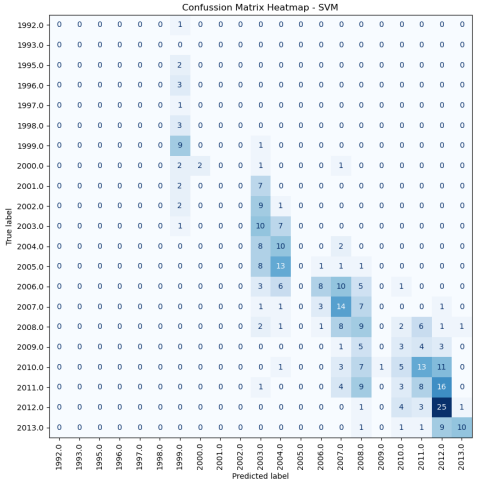
(c) KNN (optimized)



(d) LDA)



(e) RFC (optimized)



(f) SVM (optimized)

Fig. 6: Confusion matrix of all the six classifiers

Classifier	Accuracy	Std test score	params
RFC	0.407746	0.0120749	{'max_depth': None, 'min_samples_split': 2, 'n_estimators': 100}
RFC	0.412676	0.0128702	{'max_depth': None, 'min_samples_split': 2, 'n_estimators': 200}
RFC	0.414789	0.0124789	{'max_depth': None, 'min_samples_split': 2, 'n_estimators': 300}
RFC	0.415493	0.0231432	{'max_depth': None, 'min_samples_split': 5, 'n_estimators': 100}
RFC	0.416197	0.0257211	{'max_depth': None, 'min_samples_split': 5, 'n_estimators': 200}
RFC	0.419014	0.0202885	{'max_depth': None, 'min_samples_split': 5, 'n_estimators': 300}
RFC	0.405634	0.0219782	{'max_depth': None, 'min_samples_split': 10, 'n_estimators': 100}
RFC	0.409859	0.0155249	{'max_depth': None, 'min_samples_split': 10, 'n_estimators': 200}
RFC	0.409155	0.0211737	{'max_depth': None, 'min_samples_split': 10, 'n_estimators': 300}
RFC	0.419014	0.0131748	{'max_depth': 10, 'min_samples_split': 2, 'n_estimators': 100}
RFC	0.423239	0.00928937	{'max_depth': 10, 'min_samples_split': 2, 'n_estimators': 200}
RFC	0.423239	0.0112235	{'max_depth': 10, 'min_samples_split': 2, 'n_estimators': 300}
RFC	0.414085	0.0125186	{'max_depth': 10, 'min_samples_split': 5, 'n_estimators': 100}
RFC	0.416901	0.0138358	{'max_depth': 10, 'min_samples_split': 5, 'n_estimators': 200}
RFC	0.419718	0.0164252	{'max_depth': 10, 'min_samples_split': 5, 'n_estimators': 300}
RFC	0.4	0.0181739	{'max_depth': 10, 'min_samples_split': 10, 'n_estimators': 100}
RFC	0.400704	0.0193374	{'max_depth': 10, 'min_samples_split': 10, 'n_estimators': 200}
RFC	0.405634	0.0210562	{'max_depth': 10, 'min_samples_split': 10, 'n_estimators': 300}
RFC	0.407746	0.0145009	{'max_depth': 20, 'min_samples_split': 2, 'n_estimators': 100}
RFC	0.414789	0.0177319	{'max_depth': 20, 'min_samples_split': 2, 'n_estimators': 200}
RFC	0.416901	0.0167541	{'max_depth': 20, 'min_samples_split': 2, 'n_estimators': 300}
RFC	0.414085	0.024456	{'max_depth': 20, 'min_samples_split': 5, 'n_estimators': 100}
RFC	0.414789	0.0235047	{'max_depth': 20, 'min_samples_split': 5, 'n_estimators': 200}
RFC	0.417606	0.0208432	{'max_depth': 20, 'min_samples_split': 5, 'n_estimators': 300}
RFC	0.406338	0.0232074	{'max_depth': 20, 'min_samples_split': 10, 'n_estimators': 100}
RFC	0.407746	0.0151695	{'max_depth': 20, 'min_samples_split': 10, 'n_estimators': 200}
RFC	0.408451	0.0200426	{'max_depth': 20, 'min_samples_split': 10, 'n_estimators': 300}
DTC	0.371127	0.0119095	{'max_depth': None, 'min_samples_split': 2}
DTC	0.378169	0.0187117	{'max_depth': None, 'min_samples_split': 5}
DTC	0.357042	0.0208432	{'max_depth': None, 'min_samples_split': 10}
DTC	0.359155	0.0248981	{'max_depth': 10, 'min_samples_split': 2}
DTC	0.358451	0.0241294	{'max_depth': 10, 'min_samples_split': 5}
DTC	0.347887	0.0175915	{'max_depth': 10, 'min_samples_split': 10}
DTC	0.376056	0.0167244	{'max_depth': 20, 'min_samples_split': 2}
DTC	0.378873	0.0214298	{'max_depth': 20, 'min_samples_split': 5}
DTC	0.357042	0.0208432	{'max_depth': 20, 'min_samples_split': 10}
SVM	0.259155	0.00653072	{'C': 0.1, 'gamma': 'scale'}
SVM	0.257042	0.00995925	{'C': 0.1, 'gamma': 'auto'}
SVM	0.311972	0.0192345	{'C': 1, 'gamma': 'scale'}
SVM	0.316197	0.0189488	{'C': 1, 'gamma': 'auto'}
SVM	0.360563	0.0176197	{'C': 10, 'gamma': 'scale'}
SVM	0.361972	0.0165755	{'C': 10, 'gamma': 'auto'}
KNN	0.390141	0.0205798	{'n_neighbors': 3, 'weights': 'uniform'}
KNN	0.407042	0.0203617	{'n_neighbors': 3, 'weights': 'distance'}
KNN	0.376761	0.0175351	{'n_neighbors': 5, 'weights': 'uniform'}
KNN	0.411972	0.0125976	{'n_neighbors': 5, 'weights': 'distance'}
KNN	0.373239	0.0159036	{'n_neighbors': 7, 'weights': 'uniform'}
KNN	0.417606	0.0176197	{'n_neighbors': 7, 'weights': 'distance'}
LDA	0.261268	0.0222027	{}
GNB	0.273239	0.0181739	{}

TABLE VII: Possible combination of the best parameters for each classifier

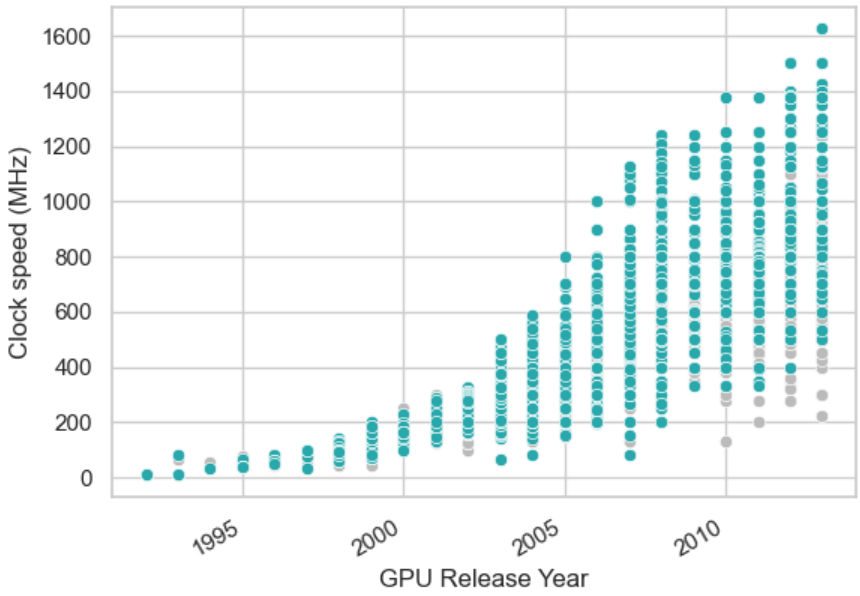


Fig. 7: GPU release year vs GPU memory clock

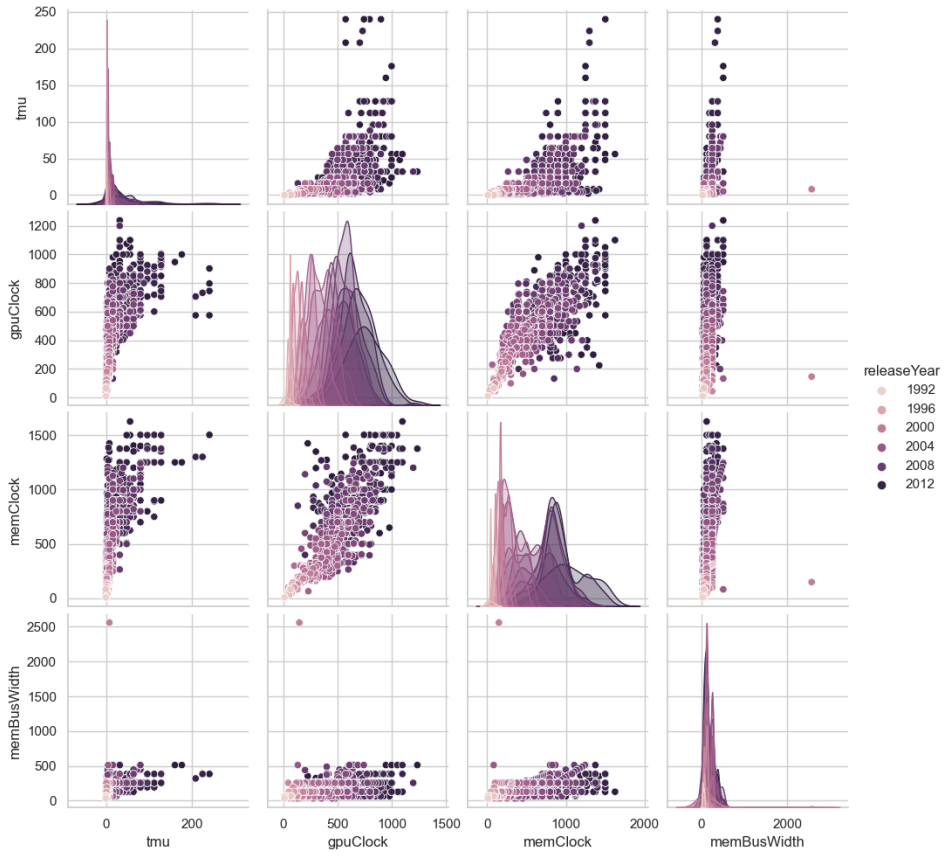


Fig. 8: Parallel plot of the data

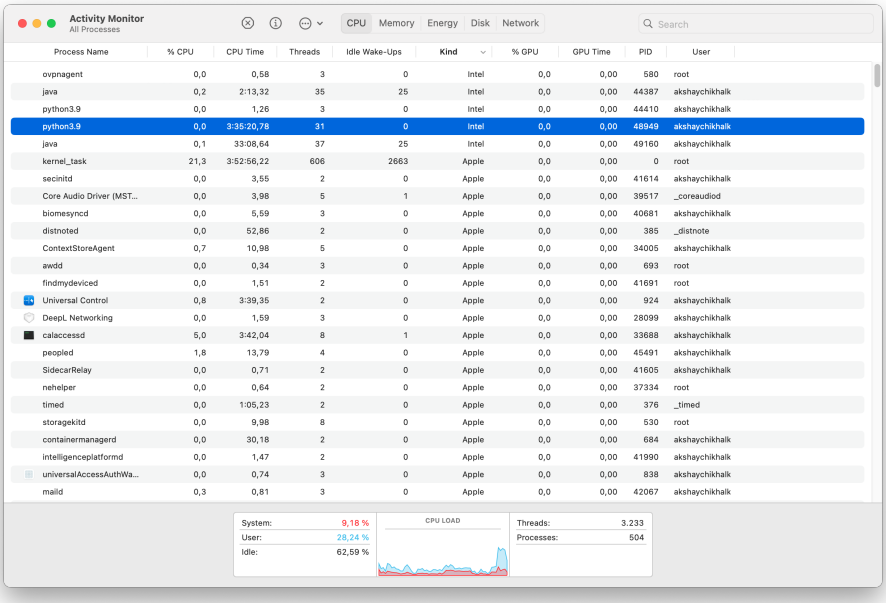


Fig. 9: Execution details

REFERENCES

[1] A. Samuel, "Some studies on machine learning using the game of checkers," *IBM Journal on Research and Development*, vol. 3, pp. 210–229, 1959.