Bachelor Thesis

Exploration of the Potential of Graph Edit Distance-Based Classifiers

Simon E. Schumacher
(matriculation number 1923651)

December 9, 2025

Submitted to
Data and Web Science Group
Prof. Gemulla
University of Mannheim

Abstract

This thesis analyzes the use of Graph Edit Distance (GED) in K-Nearest Neighbors (KNN) and Support Vector Machine (SVM) classifiers for graph classification of discrete labeled and unlabeled graphs. While only a few studies have explored the use of GED in graph classifiers, they all focus on attributed graphs with vectorial node and edge labels. It provides a comprehensive evaluation of multiple GED-based classifiers on a range of benchmark datasets, comparing their performance to state-of-the-art graph kernels. Specifically, three different GED-based kernels and a GED-based KNN classifier are evaluated on their classification performance and set in relation to their computational cost. The results show that GED-based classifiers can achieve competitive classification performance compared to state-of-the-art graph kernels. However, in relation to their huge computational cost stemming from the computation of GED, they are not a viable choice for graph classification in practice, in contrast to other graph kernels. Further analysis of driving factors influencing the performance of GED-based classifiers is done, suggesting an advantage of GED-based classifiers that are of rather simple implementations, in contrast to more complex GED-based kernels.

Contents

Abstract ii

1. Introduction 1
1.1. Problem statement . . . 1
1.2. Thesis Structure . . . 2

2. Preliminaries 3
2.1. Graph Data . . . 3
2.2. Graph Edit Distance . . . 4
2.3. K-Nearest Neighbors . . . 6
2.4. Support Vector Machines . . . 7
2.4.1. The Kernel Trick . . . 9
2.4.2. Complexity of SVMs . . . 11

3. Literature Review 12
3.1. Non-GED Graph Kernels . . . 12
3.1.1. Histogram based Kernels . . . 12
3.1.2. Weisfeiler-Lehman kernel . . . 13
3.1.3. Random Walk Kernels . . . 15
3.1.4. Other Graph Kernels . . . 16
3.2. GED based Kernels . . . 17
3.2.1. Trivial GED Graph Kernel . . . 17
3.2.2. Diffusion Kernel from Edit Distance . . . 18
3.2.3. Random Walk Edit Kernel . . . 19
3.2.4. Other GED based Graph Kernels . . . 20
3.3. GED based KNN . . . 20
3.4. Other Graph Classifiers . . . 21
3.5. Model Complexities . . . 21
3.6. Related Experimental Studies . . . 22

4. Experimental Design 24
4.1. Datasets . . . 24
4.2. Preprocessing . . . 25
4.3. Nested Cross-Validation Setup . . . 26
4.4. Classifier Specifications . . . 26
4.4.1. Trivial GED-Based Kernel with Bandwidth λ . . . 26
4.4.2. Random Walk Edit Kernel for Discretely Labeled and Unlabeled Graphs . . . 27

4.4.3. Hyperparameter Search Spaces 28
4.5. Evaluation Methodology 28

5. Experimental Results **30**
5.1. Classification performance 30
5.1.1. Labeled Datasets 30
5.1.2. Unlabeled Datasets 31
5.1.3. Answer to RQ 1 32
5.2. Computational efficiency analysis 33
5.2.1. Model Fitting runtimes 33
5.2.2. Model Prediction runtimes 34
5.2.3. Analysis and answer to RQ 2 35
5.3. Hyperparameter Influence Analysis 36
5.3.1. GED-based KNN 36
5.3.2. Trivial GED Kernel SVM 36
5.3.3. Diffusion GED Kernel SVM 38
5.3.4. Random Walk Edit Kernel SVM 39
5.3.5. Discussion 40

6. Conclusion **42**
6.1. Research Contributions 42
6.2. Summary of Findings 42
6.3. Limitations 43
6.4. Future Work 43

Bibliography **44**

A. Implementation Details **51**
A.1. Dataset Descriptions 51
A.2. IPFP Description 52
A.3. GED Computation Details 52
A.4. Libraries Used 52
A.4.1. Nested-Cross Validation and Hyperparameter Tuning 53
A.4.2. Libraries Used for Classifier Implementations 53
A.4.3. Additional Libraries 54
A.5. Computational Resources 54
A.6. Metrics Used 54

B. Additional Experimental Results **56**
B.1. Results for Additional Metrics 56
B.2. Support vectors and GED computations 60
B.3. Additional Plots 61

C. Proof Details **69**
C.1. Proof for optimal GED in Example 69

D. Ehrenwörtliche Erklärung **71**

1. Introduction

Many modern data analysis tasks rely not only on attributes of individual objects but on the relationships between them. For such tasks, graphs are a natural way to describe objects and their relationships, where vector representations are insufficient. Graphs are used in a wide range of applications, whether comparing atoms or molecules and their bonds in chemistry, people and their relationships in social networks, or analyzing transportation systems, making building machine learning models for such data an essential task (Das and Soylu 2023). However, most established machine learning and data mining techniques are designed for vectorial data, making it challenging to apply them directly to graph-structured data. Finding effective ways to address this problem is, therefore, a challenging task and an active field of research (Fuchs and Riesen 2021). Among the most popular and effective are Support Vector Machines (SVMs) with Graph Kernels, using a wide range of approaches (Kriege et al. 2020). A relatively underexplored approach is to use Graph Edit Distance within kernels, or in other graph classifiers such as K-NN, as a distance measure (Neuhaus and Bunke 2007). Graph Edit Distance, or GED, is a well-known and widely studied measure for quantifying the difference between graphs (Gao et al. 2010). It is a potent, intuitively interpretable measure, as it quantifies the dissimilarity between two graphs by the minimum number of edit operations needed to transform one into the other. However, using it for graph classification has not been widely explored in the literature, leaving its potential unclear.

1.1. Problem statement

While the GED has been widely studied as a distance measure between graphs (Gao et al. 2010), its use in graph classification has been limited to a few studies (Neuhaus and Bunke 2006b). Moreover, these studies focused only on graphs with vectorial attributes in their vertices and edges, rather than on graphs with discrete or no labels, limiting their applicability. Furthermore, another gap in the literature is the lack of comparative studies that compare GED-based classifiers with other state-of-the-art graph classifiers that do not use GED (Neuhaus and Bunke 2007). This makes it difficult to assess the effectiveness of GED-based classifiers in general, and to understand their strengths and weaknesses compared to other approaches. A potential reason for this limited investigation is the high computational complexity of computing GED, which is known to be NP-hard (Gao et al. 2010). The goal of this thesis is therefore to explore the potential of GED-based K-NN and kernels for SVMs for graph classification, and to compare their performance to other state-of-the-art graph classifiers on a range of benchmark datasets.

From this we derive the following research questions.

1. **RQ** Are GED based classifiers competitive with other state-of-the-art graph classifiers in terms of classification performance?
2. **RQ** Are GED-based classifiers also competitive regarding the cost-to-performance ratio?
3. **RQ** What are the main factors influencing the performance of GED-based classifiers?

To address these research questions, this thesis will evaluate and analyze the classification performance and computational cost of several GED-based classifiers and baseline kernels on multiple common Benchmark datasets.

1.2. Thesis Structure

The second chapter 2 introduces the foundational concepts and notations used throughout this thesis. It provides the necessary background on graph data and Graph edit distance, as well as graph classification, specifically KNN and SVMs. Building on these foundations, chapter 3 provides a comprehensive literature review of graph classification, introducing both baseline and GED-based models that are later used in the experimental study. Additionally, an overview of other related graph classification studies and their results is given. Chapter 4 then outlines the experimental design and the evaluation methodology. This includes the procedures for comparing GED-based classifiers with baseline approaches, evaluating computational cost, and analyzing factors influencing classifier performance.

This evaluation is then carried out in chapter 5. For each research question, the results from the practical experiments are presented and analyzed. Concluding the thesis, chapter 6 summarizes the main takeaways of this study, discusses its limitations, and suggests possible directions for future research.

2. Preliminaries

In this chapter, we will introduce foundational concepts and notations used throughout this thesis.

2.1. Graph Data

Graphs are mathematical structures consisting of vertices (sometimes called nodes) and edges that connect pairs of vertices. They are widely used to model relationships and interactions among entities across various domains, such as social, biological, and transportation networks. Their central advantage over other data structures is their ability to represent complex relationships and object structures. Figure 2.1 illustrates examples with molecules in chemistry, where atoms and their bonds are represented as graphs, and social networks, where people and their relationships to others are represented as graphs.

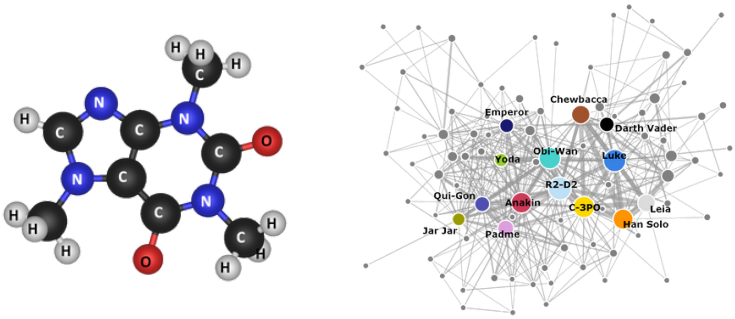


Figure 2.1.: Graph representations of a molecule[1] and a social network of Star Wars characters[2]

Formally, a graph is defined as a tuple $G = (V, E, \alpha, \beta)$. The set V denotes the finite set of vertices and $E \subseteq \{\{u, v\} \mid u, v \in V,\ u \neq v\}$ denotes the set of edges connecting unordered pairs of distinct vertices. Throughout this thesis, we restrict attention to undirected graphs without self-loops, meaning every edge connects two distinct nodes, neither of which is an origin or destination. For convenience, edges of the form $\{u, v\}$ are written as (u, v), and the number of vertices and edges of a graph G is denoted by

[2]Source: https://courses.lumenlearning.com/introchem/chapter/molecules/
[2]Source: https://evelinag.com/blog/2015/12-15-star-wars-social-network/index.html

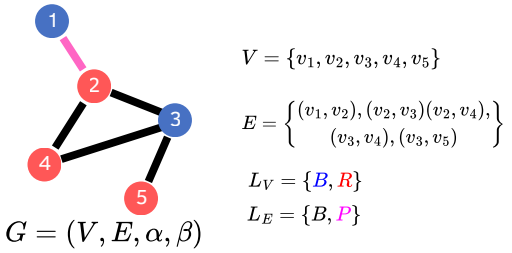


Figure 2.2.: Example of a labeled graph with discrete node and edge labels

$n_G = |V|$ and $m_G = |E|$, respectively. The functions $\alpha : V \to L_V$ and $\beta : E \to L_E$ are labeling functions that assign labels to the nodes and edges, respectively. L_V and L_E are alphabet sets that can contain categorical or numerical values. These labels can represent various properties or characteristics of the nodes and edges, such as types, weights, or other relevant information. Whether the labels are categorical or numerical can significantly impact the choice of algorithms and methods used to analyze and process the graph. Graphs whose nodes or edges carry continuous numerical vectors are typically referred to as *(continuously) attributed graphs*, while graphs with discrete labels are called *(discretely) labeled graphs* (Nikolentzos et al. 2021; Kriege et al. 2020). This thesis considers only graphs with discrete labels (or no labels). In figure 2.2, a simple graph with discretely labeled nodes and edges is illustrated as an example.

For Graph classification, each graph G is associated with a class label $c \in C$, where C is a finite set of possible classes. Given a dataset $\mathcal{G} = \{G_1, \ldots, G_n\}$, the goal of a classifier is to learn patterns correlating graph structure and labels from a training subset $\mathcal{G}_{\text{train}}$ and to predict the class of unseen graphs in a test set $\mathcal{G}_{\text{test}}$. The number of graphs in a given dataset will be denoted as $\mathcal{N}$ throughout this thesis.

2.2. Graph Edit Distance

Graph Edit Distance (GED) is one of the most popular methods for measuring the dissimilarity between two graphs (Neuhaus and Bunke 2005). The idea of edit distances in general first originated in the field of string matching, where the Levenshtein distance (Levenshtein 1965) was introduced to measure the difference between two strings. This concept was later adapted to many other data structures, including graphs for which the Graph Edit Distance was formally introduced in Sanfeliu and Fu (1983).

The core idea of GED is to define the dissimilarity of two graphs G, G' by quantifying the minimum number of edit operations e needed to transform G into (a graph isomorphic to) G'. The standard set of edit operations typically consists of deletions, insertions and substitutions of nodes and edges (Gao et al. 2010). These operations are associated with costs $c(e) \geq 0$, which can be defined based on the specific application and the relevant node and edge labels. The simplest costs are constant uniform costs, where the price for

each operation is typically set to 1.

A general definition of the Graph Edit Distance is given by equation 2.1.

$$GED(G, G') = \min_{\pi \in \rho(G,G')} \sum_{e \in \pi} c(e) \tag{2.1}$$

Here $\rho(G, G')$ is the set of all possible edit paths that transform G into G', and $\pi = (e_1, e_2, \ldots, e_k)$ is a specific edit path, consisting of a sequence of edit operations. To illustrate this in an example, consider the two graphs G and G' in figure 2.3. The graphs consist of nodes v_i with categorical labels R and B, indicated in red and blue, respectively. To simplify the example, the graphs do not contain any edge labels.

Assuming uniform costs of 1 for all edit operations, the shortest edit path consists of

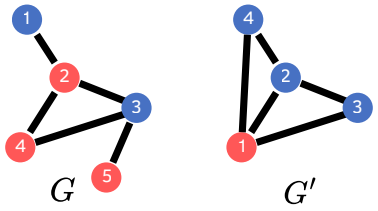


Figure 2.3.: Example graphs G and G' for Graph Edit Distance

the following steps:

- Substitute the label of node v_2 from 'R' to 'B'
- Insert an edge between nodes v_1 and v_4
- Delete the edge between nodes v_3 and v_5
- Delete node v_5

The total cost of this edit path is 4, which means that $GED(G, G') = 4$. Figure 2.4 illustrates the edit operations transforming graph G into graph G'. This path can be

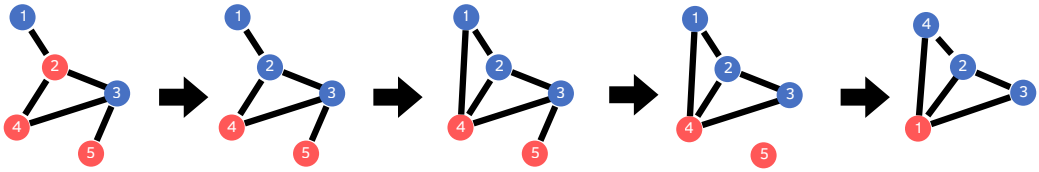
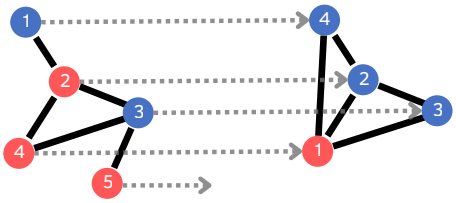


Figure 2.4.: Example of edit operations transforming graph G into graph G'

proven to be optimal, as shown in the appendix in section C.1. In the process of finding the optimal edit path and computing the GED, a central task is identifying the optimal node mapping $M : V(G_1) \cup \{\epsilon\} \rightarrow V(G_2) \cup \{\epsilon\}$ between the individual nodes of the two graphs. This mapping defines which nodes are substituted or matched $M(v) = u$, inserted $M(\epsilon) = v$ or deleted $M(v) = \epsilon$, creating the basis for the edit operations. For



$$M = \{(v_1, v'_4), (v_2, v'_2), (v_3, v'_3), (v_4, v'_1), (v_5, \epsilon)\}$$

Figure 2.5.: Node mapping derived from the edit path between graphs G and G'

our previous example, the node mapping derived from the edit path is noted in figure 2.5.

Finding the Graph Edit Distance and the optimal edit path between two graphs is a complex combinatorial optimization problem. It has been shown that computing the exact GED is NP-hard, meaning that there is no known polynomial-time algorithm to solve it for all cases (Gao et al. 2010).

This can intuitively be explained by looking at finding the optimal node mapping between two graphs (Abu-Aisheh et al. 2015). The number of possible node mappings between two graphs grows exponentially with the number of nodes, resulting in a computational complexity of $O(n^{n'})$ to compute the exact Graph Edit Distance between two graphs G and G' with n and n' number of nodes respectively (Neuhaus et al. 2006).

As a result, many approximation algorithms have been proposed to compute GED more efficiently, trading off some accuracy for speed (Bougleux et al. 2017; Gao et al. 2010).

2.3. K-Nearest Neighbors

K-Nearest Neighbors (KNN) is a simple and intuitive algorithm that can be used for both classification and regression tasks. Initially introduced in Fix and Hodges (1989) and later formalized in Cover and Hart (1967), it has since become a popular choice for many machine learning applications due to its simplicity and effectiveness. KNN is often referred to as a lazy or instance-based learner because it does not learn an explicit model during training. Instead, it stores the training data and uses it directly for predictions (Cunningham and Delany 2021). KNN classifiers work by finding the K nearest data points in the training set to a given data point G in the test set according to a defined distance metric $d(G, G_i)$, as illustrated in figure 2.6. The classifier assigns the class label based on the majority class of these neighbors, making it a very flexible approach (Syriopoulos et al. 2025), as it can be used with different types of data, as long as a suitable distance metric is defined. In figure 2.6 the test point will be classified to class A as the majority of its 3 nearest neighbors belong to class A.

The hyperparameter K, which defines the number of neighbors to consider, is critically

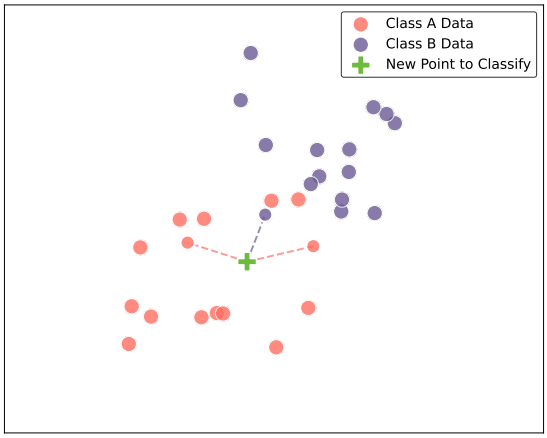


Figure 2.6.: Illustration of a (3)-NN classifier in a 2D feature space

important for prediction.In its simplest form, where $K = 1$, the class of the nearest neighbor is assigned to the test point (Syriopoulos et al. 2025), which can be very sensitive to noise in the data. Increasing K can smooth the decision boundary, making the classifier more robust to noise but potentially less sensitive to local patterns in the data. It can lead to a bias towards the majority class in imbalanced datasets (Syriopoulos et al. 2025). Additionally, a popular approach is to weight neighbors' votes to increase the influence of closer neighbors relative to more distant ones (Dudani 1976).
KNNs have the advantage of being very widely applicable, as only a distance measure is needed, making them very simple classifiers that can model very complex data.

2.4. Support Vector Machines

Support Vector Machines (SVMs) were introduced in (Cortes and Vapnik 1995) and are another simple yet powerful supervised learning algorithm that can be used for both classification and regression tasks.

The core concept of SVMs is to find an optimal separating hyperplane $w^T x + b = 0$ with maximum margin between the data points of different classes (Burges 1998). With that the model can classify new data points with a signum function $sgn(w^T x + b)$, which assigns a label based on which side of the hyperplane a data point lies. In its simplest form, a set of linearly separable data points in the training data can be perfectly separated by two parallel hyperplanes (hard margin), with a maximum margin between them, as shown in figure 2.7, which run through the closest data points of each class, x^+ and x^-, called the support vectors (Burges 1998).

The margin between the hyperplane and either support vector is maximized for both support vectors equally, resulting in both of them having the same distance to the hyperplane. In the following notation, this distance will be set to 1 for ease of notation, which can always be achieved by scaling w and b accordingly, so the conditions $(w^T x^+ +$

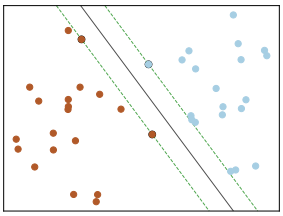


Figure 2.7.: hard margin

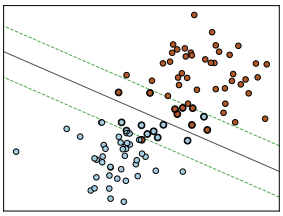


Figure 2.8.: soft margin

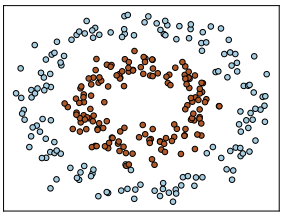


Figure 2.9.: non-linear

$b) = 1$ and $(w^T x^- + b) = -1$ hold (Smola and Schaulkopf 2004). When multiplied by the class labels y_i, the condition can then be generalized to all training data points as $y_i(w^T x_i + b) \geq 1$, with the class labels being 1 or -1 (Burges 1998). The distance between the two margin boundaries is $\frac{2}{\|w\|}$, which means that maximizing the margin is equivalent to minimizing $\|w\|$. So to find the optimal hyperplane we need to solve the optimization problem of finding the best combinations of values for w and b, to minimize $\|w\|$, while satisfying the margin constraints for all training data points formulated in 2.2. To have a more convenient optimization problem the objective function is reformulated to minimize $\frac{1}{2}\|w\|^2$ instead. This is mathematically equivalent to minimizing $\|w\|$ but is easier differentiable and therefore easier to solve (Burges 1998).

$$\min_{w,b} \frac{1}{2}\|w\|^2 \quad s.t. \quad y_i(w^T x_i + b) \geq 1, \quad i = 1, \ldots, n \tag{2.2}$$

However in practice, this has three significant limitations. Firstly, real-world data is often not perfectly (linearly) separable, meaning that no hyperplane can perfectly separate the data points of different classes, as illustrated in figure 2.8. Another limitation is that computing the optimal hyperplane as the optimization problem in equation 2.2 can be computationally expensive for high-dimensional data. And lastly, many datasets have class relations that are rather complex and non-linear, as illustrated in figure 2.9.

The first limitation can be addressed by introducing slack variables $\xi_i \geq 0$ for each data point, allowing some data points to be misclassified or to lie within the margin (soft margin). Simultaneously, a penalty term is added to the objective function in equation 2.3, which can be controlled by a regularization parameter C, to balance the trade-off between maximizing the margin and minimizing the classification error (Djemai et al. 2016).

$$\min_{w,\xi} \left(\frac{1}{2}\|w\|^2 + C \sum_{i=1}^{n} \xi_i \right) \quad s.t. \quad y_i(w^T x_i + b) \geq 1 - \xi_i, \quad \xi_i \geq 0, \quad i = 1, \ldots, n \tag{2.3}$$

Setting C to a high value will result in a smaller margin and fewer misclassifications in the training data. In comparison, a low value of C will result in a larger margin with more misclassifications, as illustrated in figure 2.10, but potentially better generalization to unseen data (Smola and Schaulkopf 2004).

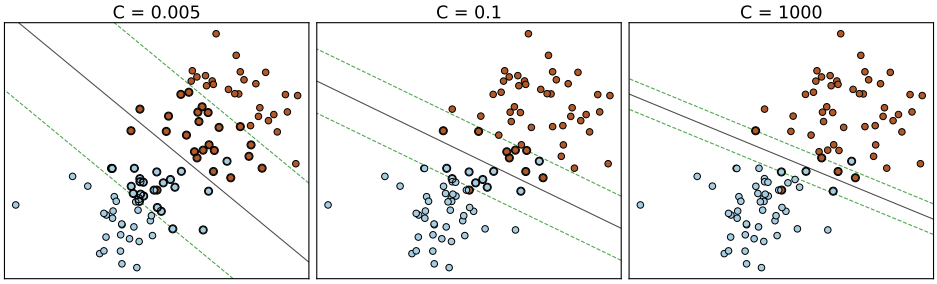


Figure 2.10.: Effect of the regularization parameter C on the margin and misclassifications

The second limitation is addressed by the conversion of the primal optimization problem for finding the optimal hyperplane in equation 2.2 into a dual problem in Lagrange formulation in equation 2.4 (Djemai et al. 2016). To achieve that, the parameter w of the optimal separation hyperplane is reformulated as its linear combination of the training data $w = \sum_{i=1}^{n} \alpha_i y_i x_i$, where α_i denote the Lagrange multipliers, which correspond to the importance of each training data point in defining the optimal hyperplane. The detailed explanation and conversion from the primal to the dual form is beyond the scope of this thesis and can be found in Bennett and Bredensteiner (2000).

$$\max_{\alpha} \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \langle x_i, x_j \rangle \quad s.t. \quad 0 \leq \alpha_i \leq C, \sum_{i=1}^{n} \alpha_i y_i = 0 \tag{2.4}$$

Computing the optimal hyperplane as an optimal set of Lagrange multipliers α_i is a quadratic optimization problem, which can be solved more efficiently (Smola and Schaulkopf 2004). The Lagrange multipliers α_i can be interpreted as weights for each training data point, indicating their importance in defining the optimal hyperplane. The data points with non-zero Lagrange multipliers are the support vectors used to classify new data points while the other data points can be ignored. Another property of the dual form is that the data points appear only as dot products between pairs of data points $\langle x_i, x_j \rangle$ which enables us to address the third limitation using the kernel trick.

2.4.1. The Kernel Trick

Since the SVM is solved with dot products, between all pairs of data points, we supply the SVM with a matrix of these dot products, called the Gram matrix or kernel matrix $\mathcal{S}_{ij} = \langle x_i, x_j \rangle$, instead of the actual data points (Burges 1998). To compute the kernel matrix, we can use kernel functions $\mathcal{K}(x_i, x_j)$, which have the property that they are equivalent to mapping the data points into a high dimensional feature space $\phi(x)$ and computing the dot product in that space, $\mathcal{K}(x_i, x_j) = \langle \phi(x_i), \phi(x_j) \rangle$ (Schölkopf 2000). To illustrate this, consider the following distribution of data points in a two-dimensional space, belonging to two different classes, as shown in figure 2.9. In the original two-dimensional space, these data points are not linearly separable. Attempting to do so

would use a simple dot product, $\langle x_i, x_j \rangle$, to compute the kernel matrix. However by using a different kernel function, we can implicitly map the data points into a different (often higher-dimensional) vector space, where they are linearly separable. In this case, a visually intuitive mapping is to map the data points with a function $\phi(x) = \exp(-(x^2+y^2))$, visualized in figure 2.11. In the new space, the data points are now linearly separable by a hyperplane.

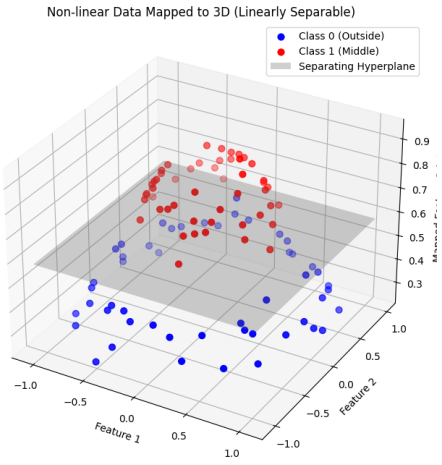


Figure 2.11.: 2-dimensional data points mapped into a new dimension

Instead of explicitly computing this mapping $\phi(x)$, we can use a kernel function to compute the dot products in this high-dimensional space implicitly. The kernel function corresponding to this mapping is $\mathcal{K}(x_i, x_j) = \exp(-(x_1^2 + x_2^2 + y_1^2 + y_2^2))$. Plugging this kernel function into the dual form of the SVM in equation 2.4 allows us to find an optimal hyperplane in this high-dimensional space, without having to compute the mapping $\phi(x)$ explicitly. This can generally be done for many different kernel functions, allowing SVMs to find non-linear decision boundaries in the original input space (Smola and Schaulkopf 2004).

The kernel function can be chosen based on the specific data and application, making SVMs very flexible and powerful (Smola and Schaulkopf 2004), especially for complex, non-vectorial data structures such as graphs. Simultaneously, the kernel function implicitly maps the data points into a high-dimensional space, where they are more likely to be linearly separable.

For a similarity measure to be a valid kernel function, it must satisfy Mercer's condition (Cervantes et al. 2020). This ensures that the implicit mapping into the high-dimensional space follows the rules of Euclidean geometry so that the kernel function behaves like a dot product in that space. Specifically, the kernel function must be symmetric $\mathcal{K}(x_i, x_j) = \mathcal{K}(x_j, x_i)$ and positive semi-definite (PSD) (Mercer 1909). For that to be the case, the kernel function must satisfy the condition in equation 2.5 for any finite set of data points $\{x_1, x_2, \ldots, x_n\}$ and any set of real coefficients $\{c_1, c_2, \ldots, c_n\}$

(Mercer 1909).

$$\sum_{i=1}^{n}\sum_{j=1}^{n} c_i c_j \mathcal{K}(x_i, x_j) \geq 0 \tag{2.5}$$

This also ensures that the double sum in equation 2.4 is always non-negative, making the optimization problem convex and ensuring that it has a unique solution (Smola and Schaulkopf 2004). Additionally, this property also ensures that all eigenvalues of the Gram matrix are non-negative (Cervantes et al. 2020). In practice, many kernel functions used in SVMs are not strictly PSD but still work well in practice (Neuhaus and Bunke 2007). It is, however, important to keep this in mind when defining new kernel functions. Using a non-Mercer kernel can lead the SVM to not converge to an optimal solution but instead get stuck in local minima (Burges 1998). On vectorial data, the most common kernel functions are the linear kernel $\mathcal{K}(x_i, x_j) = \langle x_i, x_j \rangle$, the polynomial kernel $\mathcal{K}(x_i, x_j) = (\langle x_i, x_j \rangle + c)^d$ and the radial basis function (RBF) kernel $\mathcal{K}(x_i, x_j) = \exp\left(-\frac{\|x_i - x_j\|^2}{2\sigma^2}\right)$. For more complex data structures such as graphs, kernel functions are convenient because the data is not in vector form. Explicitly mapping the graphs into a vectorial space can be very complex and computationally expensive. Instead, the kernel trick allows defining a similarity measure between graphs that can operate directly on the graph structure and implicitly map the graphs into a high-dimensional space. Therefore, defining suitable graph kernel functions is an active area of research.

Additionally, it should be noted that support vector machines are not the only machine learning algorithms that use kernel functions. Kernels can also be used in other algorithms such as Logistic Regression or Principal Component Analysis (PCA) (Lee et al. 2006; Marukatat 2023).

2.4.2. Complexity of SVMs

The complexity of SVMs depends on the specific implementation and the optimization algorithm used. In general, computing the kernel matrix for n data points is $O(n^2 \cdot C_k)$, with C_k denoting the complexity for computing the kernel value between two data points (Chang and Lin 2011). Training the SVM with the dual-form optimization problem in equation 2.4 has a worst-case complexity of $O(n^3)$ (Burges 1998) However, depending on the sparsity of the kernel, modern optimization is faster in practice (Chang and Lin 2011). In summary, the overall computational complexity of training an SVM is $O(n^2 \cdot C_k + n^3)$(Smola and Schaulkopf 2004). For graph kernels, it is common that computing the kernel value dominates the overall complexity, making kernel construction significantly more expensive than training. When predicting the class of a new data point, the SVM computes the decision function based on the support vectors and their corresponding Lagrange multipliers. This involves computing the kernel function between the new data point and each support vector and combining these values with the Lagrange multipliers to compute the final decision value. The computational complexity of predicting the class of a new data point is therefore $O(s \cdot C_k)$ (Smola and Schaulkopf 2004).

3. Literature Review

Graph classification has been approached with various methods over the years. The two historically dominant ones are graph kernels, which enable the use of kernel machines such as SVMs (Cervantes et al. 2020) and graph neural networks (GNNs) (Kriege et al. 2020; Wu et al. 2020). Among kernel-based approaches, methods differ mainly in how they compare graphs—ranging from neighborhood aggregation and random walks to edit-distance-based similarity measures.

Because this thesis focuses on Graph Edit Distance (GED)-based classifiers, particularly GED kernels and GED-KNN, the review emphasizes these methods while also summarizing widely used non-GED graph kernels and selected non-kernel approaches for context. The chapter concludes with an overview of model complexities and relevant comparative studies from the literature, providing reference points for the experimental evaluation in Chapter 5.

3.1. Non-GED Graph Kernels

Graph kernels have been widely studied for many years as a method for classifying graphs (Kriege et al. 2020). Their core challenge is to capture structural information that is discriminative for classification while remaining computationally tractable. Existing kernels vary significantly in their emphasis on local patterns (e.g. neighborhoods), global structures (e.g., walks), or combinatorial substructures. The most influential families include neighborhood aggregation kernels, random-walk kernels, assignment kernels, and subgraph-based kernels (Kriege et al. 2020). This section reviews representative methods from these families that do not rely on graph edit distance, specifically 3 types of graph kernels are presented in detail.

3.1.1. Histogram based Kernels

One of the most straightforward and naive approaches to define a graph kernel is to use histograms of node labels or edge labels. These kernels construct a feature vector for each graph, where each entry corresponds to a unique node or edge label and contains the number of nodes or edges with that label. The kernel function $k_{hist}(G, G')$ is defined as the dot product or another kernel function, such as RBF, between the feature vectors of two graphs (Kriege et al. 2020).

For example, consider the graph G in figure 3.1, which consists of nodes with 3 categorical labels for nodes and 2 categorical labels for edges, visualized by different colors. The number of occurrences of each node or edge label in the graph is summarized in the vectors next to the graph.

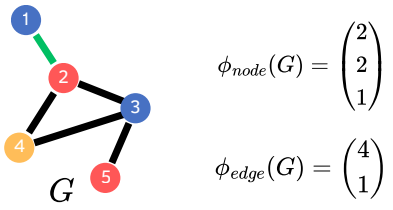


Figure 3.1.: Example graph for histogram-based kernel

To compute the whole kernel matrix for a set of $\mathcal{N}$ graphs with n nodes and m edges, the computational complexity is $O(\mathcal{N} \cdot (n + |L_V|) + \mathcal{N}^2 \cdot (|L_V|))$ for node histograms and $O(\mathcal{N} \cdot (m + |L_E|) + \mathcal{N}^2 \cdot (|L_E|))$ for edge histograms (Nikolentzos et al. 2021; Shervashidze et al. 2011). To store the histograms, $O(|L_V|)$ memory is needed for node histograms and $O(|L_E|)$ for edge histograms per graph. Prediction with an SVM using histogram features requires computing the histogram of the new graph in $O(n_p + |L_V|)$ and evaluating the kernel against all support vectors in $O(s \cdot |L_V|)$ (Nikolentzos et al. 2021).

Histogram kernels are, in general, valid Mercer kernels, however, they entirely ignore the graph structure (Kriege et al. 2020). In this study, a simple node histogram is used as a baseline, since it is a very standardly used baseline method in many studies and is very computationally efficient (Nikolentzos et al. 2021).

3.1.2. Weisfeiler-Lehman kernel

While histogram kernels ignore structural information entirely, neighborhood aggregation approaches aim to capture it by comparing local graph substructures based on similarly labeled neighborhoods (Shervashidze et al. 2011). These methods capture structural similarity more effectively by iteratively updating node labels based on each node's neighborhood. Most neighborhood-aggregation kernels are based on the Weisfeiler-Lehman (WL) framework (Shervashidze et al. 2011), which serves as the basis for the concrete kernels that define rigorous kernel functions. The WL framework is based on and named after the Weisfeiler-Lehman test of graph isomorphism (Lehman and Weisfeiler 1968), which is a method for determining whether two graphs are isomorphic, i.e. have the same structure. It updates node labels iteratively by combining each node's current label with the multiset of its neighbor's labels (Schulz et al. 2022).

Starting from an initial labeling function $\alpha : V \to L_V^0$, the WL update at iteration i is defined by equation 3.1.

$$\alpha^i(v) = hash\left(\alpha^{i-1}(v), sort(\{\alpha^{i-1}(u) : u \in N(v)\})\right) \qquad \forall v \in V \tag{3.1}$$

Here, $N(v)$ denotes the set of neighbors of node v and $hash$ is a function that maps the combined label and sorted multiset of neighbor labels to a new unique label. With increasing iterations, the labels capture increasingly larger neighborhood structures. Figure 3.2 illustrates the first relabeling step from the initial set of nodes $L_V^0 = \{B, R\}$. This process continues until a stopping criterion is met such as a fixed number of iterations h.

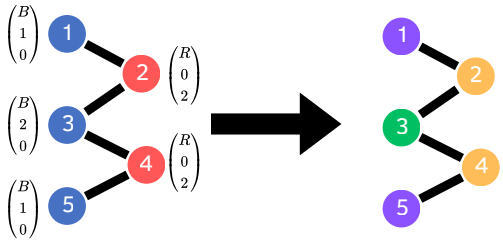


Figure 3.2.: WL relabeling from iteration 0 to iteration 1

WL-based graph kernels prove to be very effective and efficient in practice (Zhang et al. 2018; Kriege et al. 2020), and therefore often considered the state of the art (Nikolentzos et al. 2021). Its computational complexity is $O(h \cdot m)$ for the framework computation of one graph to h iterations (Shervashidze et al. 2011). Among the most widely used WL-based kernels is the WL subtree kernel (Shervashidze et al. 2011), which counts the number of standard node labels between two graphs after applying the WL relabeling process for a fixed number of iterations h. Practically, this is done by constructing a feature vector $\phi(G)$ for each graph G, where each entry in the vector corresponds to a unique node label and the value of the entry is the number of nodes in the graph with that label (Shervashidze et al. 2011). Based on these feature vectors, the kernel function is defined as the dot product or another kernel function between the feature vectors of two graphs. This method is powerful because it creates a huge implicit feature space of all possible combinations of neighborhoods and their labels.

Computing the kernel matrix for a train set of $\mathcal{N}_{\text{train}}$ with n nodes and m edges can be done in $O(\mathcal{N}_{\text{train}}(h \cdot m) + {\mathcal{N}_{\text{train}}}^2(h \cdot n))$. This is achieved by first computing the WL relabeling and vector representations for all graphs in $O(\mathcal{N}_{\text{train}}(h\ m))$, and then computing the dot products between all pairs of graphs in $O({\mathcal{N}_{\text{train}}}^2(h\ n))$ (Shervashidze et al. 2011). When predicting the class of a new graph, the computational complexity is $O(s \cdot (h \cdot n_{max}) + h \cdot m_p)$.

The WL subtree kernel is a valid Mercer kernel (Shervashidze et al. 2011).

In general, most WL kernels work by applying the WL relabeling process and then performing a base kernel on the relabeled graphs. The WL subtree kernel can, in that regard, be seen as performing a simple node histogram as a base kernel.

Beyond the subtree kernel, WL-based kernels include the WL shortest-path kernel (Shervashidze et al. 2011), the WL optimal assignment kernel and WL pyramid kernels (Nikolentzos et al. 2021). These variants differ in the base kernel applied after WL relabeling. WL-based kernels serve as a strong benchmark for evaluating GED-based methods due to their favorable computational properties and strong empirical performance. In this study, the WL subtree kernel will serve as a baseline method as it is efficient, popular and widely regarded as a strong performer among graph kernels (Nikolentzos et al. 2021).

3.1.3. Random Walk Kernels

Another popular group of graph kernels are random walk kernels (Vishwanathan et al. 2010), which count the number of walks (label sequences) that two graphs share. A walk is a sequence of nodes and edges in a graph, where each node is connected to the next by an edge and the walk is characterized by the sequence of node and edge labels it traverses. In the example graph in figure 3.3, a walk of length 3 is illustrated from node v_1 to node v_5, traversing the edges (v_1, v_2), (v_2, v_3) and (v_3, v_5) with the label sequence $W_{labels} = (B, R, B, R)$.

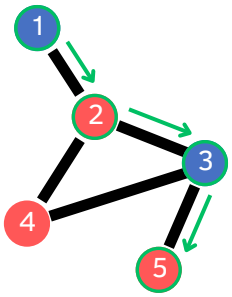


Figure 3.3.: Example of a random walk of length 3 $W = (v_1, v_2, v_3, v_5)$

Early kernels by Gärtner et al. (2003) and Kashima et al. (2003) formalized this idea: two graphs are similar if they share many walks with identical label sequences. Since walks may have arbitrary length, the corresponding feature space becomes infinite-dimensional.

To compute the common random walks, the direct product graph of two graphs $G_\times = G_1 \times G_2$ is constructed based on the definition in equation 3.2, on which every walk corresponds to a common walk in G_1 and G_2 (Kriege et al. 2020).

$$\begin{aligned} V_\times &= \{(v_1, v_2) \in V(G_1) \times V(G_2) \mid \alpha(v_1) = \alpha(v_2)\}, \\ E_\times &= \big\{((u_1, u_2), (v_1, v_2)) \in V_\times \;\big|\; \\ &\qquad (u_1, v_1) \in E(g_1)\ (u_2, v_2) \in E(g_2),\ \beta((u_1, v_1)) = \beta((u_2, v_2))\big\}. \end{aligned} \tag{3.2}$$

To compute the number of walks in the product graph, the product graph's adjacency matrix $A_\times$, defined in equation 3.3.

$$[A_\times]_{(u_1,u_2),(v_1,v_2)} = \begin{cases} 1 & \text{if } \{(u_1, u_2), (v_1, v_2)\} \in E_\times \\ 0 & \text{otherwise} \end{cases} \tag{3.3}$$

To illustrate the construction of the product graph and its adjacency matrix, figure 3.4 shows an example of a product graph between two graphs G and G'.

To compute the final kernel value, the random walk kernel is then defined as in equation 3.4, often referred to as the direct product kernel. The adjacency matrix $A_\times$ is raised to

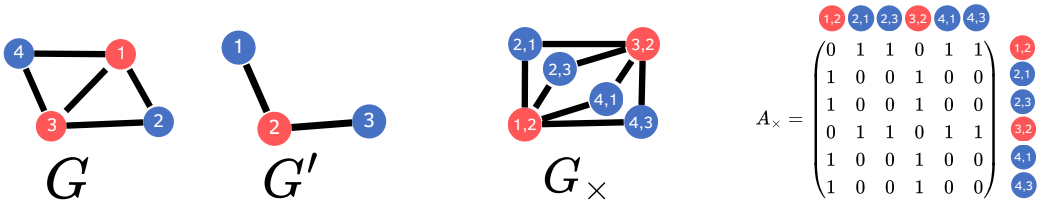


Figure 3.4.: Product graph of G and G' and its adjacency matrix

the power of l, to compute the number of walks of length l between all pairs of nodes in the product graph, with the weighting factor γ_l determining the influence of particular lengths.

$$K_{RW}(g_1, g_2) = \sum_{i,j=1}^{|V_\times|} [\sum_{l=0}^{\infty} \gamma_l A_\times^l]_{ij} \tag{3.4}$$

The sequence of weights $\Gamma = \{\gamma_0, \gamma_1, \ldots\}$ must be a sequence of weights such that the above sum in equation 3.4 converges. A popular variant of the direct product kernel is the geometric random walk kernel where the weights are defined as $\gamma_i = \lambda^i$, with $0 < \lambda \leq 1$, gradually weighting walks by their length. This allows the kernel to avoid restricting itself to walks of fixed length, setting l to infinity while still ensuring convergence of the sum. This resulting sum can be reformulated using matrix inversion as in equation 3.5 (Neuhaus and Bunke 2007).

$$K_{RW}(g_1, g_2) = \sum_{i,j=1}^{|V_\times|} [(I - \lambda A_\times)^{-1}]_{ij} \tag{3.5}$$

I denotes the identity matrix of appropriate size. Random walk kernels have the disadvantage of being computationally expensive as the size of the product graph can be immense, because it either requires computing the powers of the adjacency matrix or a matrix inversion (Kriege et al. 2020). Using optimizations, the complexity of computing the random walk kernel function between two graphs is $O(n^3)$ (Borgwardt et al. 2006).

3.1.4. Other Graph Kernels

Beyond the neighborhood aggregation based kernels and random walk based kernels, the assignment and matching based kernels, shortest path kernels and graphlet kernels are other popular groups of graph kernels from the literature.

The **Assignment and Matching based Kernels** (Kriege et al. 2020) work by finding an optimal matching between the nodes of two graphs, based on their labels and local structures. Its most prominent representative is the optimal assignment kernel (Kriege et al. 2016). It finds an optimal matching between the nodes of two graphs, with a node based kernel function comparing local structures and labels.

Shortest path Kernels are another rather popular approach, which is based on the similarities of the shortest paths between all pairs of nodes in two graphs. How similar the paths are is determined by a base kernel function that compares the lengths of the two paths as well as the labels of the nodes and edges along the paths.

The idea of **Graphlet kernels** (Pržulj 2007) is to decompose the graphs into small subgraphs, called graphlets, and then compare the graphlets between two graphs (Nikolentzos et al. 2021). This is being done by constructing a feature vector for each graph, where each entry is the number of occurrences of a specific graphlet in the graph. A major advantage of graphlet kernels is that they are easily scalable by means of simple sampling schemes (Nikolentzos et al. 2021). Graphlet kernels as well as histogram kernels can be counted to the family of subgraph kernels.

3.2. GED based Kernels

GED-based kernels transform the graph edit distance into a similarity measure, aiming to preserve its sensitivity to structural discrepancies at the cost of significant computational overhead. The work on GED-based kernels is not as extensive as that on non-GED-based kernels and only a few approaches have been proposed in the literature, mainly by Horst Bunke, Michel Neuhaus and Kaspar Riesen (Neuhaus and Bunke 2007, 2004; Neuhaus et al. 2009; Bunke and Riesen 2007; Riesen et al. 2015).

3.2.1. Trivial GED Graph Kernel

Similar to using GED as a distance metric in KNN, a fundamental approach is to transform the GED distance into a similarity measure and use that as a kernel function. In Neuhaus and Bunke (2007), four such functions are proposed as the trivial GED kernel.

- $k_1(G_1, G_2) = -GED(G_1, G_2)^2$
- $k_2(G_1, G_2) = -GED(G_1, G_2)$
- $k_3(G_1, G_2) = \tanh(-GED(G_1, G_2))$
- $k_4(G_1, G_2) = \exp(-GED(G_1, G_2))$

These transformations mainly differ in how the similarities are scaled illustrated in figure 3.5.

Only the exponential function guarantees positive similarity values, while the others can or will produce negative similarity values for large GEDs. Even though this yields unintuitive similarity values, it is not required for kernel functions to produce only positive similarity values.

The authors note that these kernels are not positive semi-definite (PSD) and therefore not valid kernels in the strict sense (Neuhaus and Bunke 2007).

Beyond the GED computation, these kernels are very simple to compute. Therefore, the computational complexity of these kernels matches that of GED (Neuhaus and Bunke

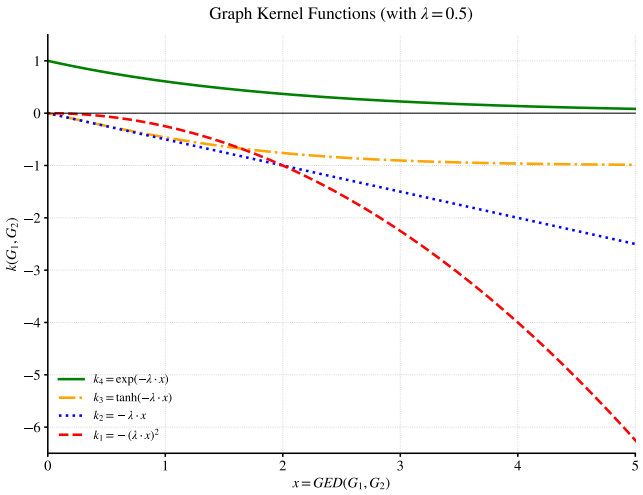


Figure 3.5.: Visualization of the trivial GED-based kernels

2007). In the Experimental evaluation in this study, this kernel will be adapted with a bandwidth parameter λ as specified in section 4.4.1.

3.2.2. Diffusion Kernel from Edit Distance

Diffusion kernels are a type of node kernel on graph structures that compute a kernel matrix of similarities between individual nodes. Their idea comes from differential heat equations and is adapted to graph structures (Kondor and Lafferty 2002). They take the negative graph Laplacian matrix as in equation 3.6 as a base generator structure and transform it using a diffusion process on the graph structure. The authors note that the base similarity matrix is not limited to the one defined in equation 3.6 but can be any symmetric similarity measure.

$$B_{ij} = \begin{cases} -\deg(v_i) & \text{if } i = j \\ 1 & \text{if } (v_i, v_j) \in E \\ 0 & \text{otherwise} \end{cases} \tag{3.6}$$

The diffusion process works by using the exponential function in equation 3.7 or the von Neumann diffusion kernel in equation 3.8. These equations transform the base generator structure matrix B into a diffusion kernel matrix K, simulating a diffusion process on the graph structure analogous to heat diffusion in a medium.

$$K = \sum_{k=0}^{\infty} \frac{\lambda^k B^k}{k!} = exp(\lambda B) \tag{3.7}$$

$$K = \sum_{k=0}^{\infty} \lambda^k \cdot B^k \tag{3.8}$$

The role of the decay factor $\lambda \in (0, 1)$ is to control that for increasing k the influence of B^k decreases, to ensure convergence of the series. To practically compute these kernels, the series are truncated after k iterations. This guarantees a positive semi-definite (PSD) kernel matrix D which can then be used as a kernel function in an SVM. In Neuhaus et al. (2009), GED is used to modify the diffusion kernel approach, yielding a kernel for graph classification. The idea is to use a GED-based similarity matrix in the diffusion step instead of the graph Laplacian. For that, the matrix B, which is defined in equation 3.9, is proposed.

$$B_{ij} = \max_{1 \leq s,t \leq n} (GED(g_s, g_t)) - GED(g_i, g_j) \quad \forall 1 \leq i, j \leq n \tag{3.9}$$

This base similarity matrix B is transformed using the diffusion process from the orginal node kernel with equations 3.7 or 3.8, to obtain the final kernel matrix K (Neuhaus et al. 2009).

This GED-based diffusion kernel has the advantage of being a valid graph kernel as it is guaranteed to be positive semi-definite (PSD) by the diffusion process. The Computation Complexity of the diffusion process is $O(k \cdot \mathcal{N}^3)$, which with the complexity of computing GED, adds up to a total complexity of $O(\mathcal{N}^2 \cdot (n^n) + k \cdot \mathcal{N}^3)$, where k is the number of iterations before the series is truncated. Since the value of the diffusion kernel between two graphs depends on the entire dataset, it is not sufficient to store only the support vectors as in standard SVMs. For the computational complexity of assigning a label to a new graph, this also means that the GED between the new graph and the whole training set must be computed rather than just the support vectors. Therefore, this requires a computational complexity of $O(\mathcal{N} \cdot (n^n) + k \cdot \mathcal{N}^2)$ (Neuhaus and Bunke 2007).

3.2.3. Random Walk Edit Kernel

The random walk edit kernel from Neuhaus et al. (2009) is an approach to enhance a standard random walk kernel by incorporating the node mappings from GED. It is based on the attributed random walk kernel graphs from Borgwardt et al. (2005). In that variant the node- and edge-label equality constraints from the product graph were removed. Instead, the adjacency matrix was redefined in equation 3.10 to reflect the similarity between node attributes K_V and edge attributes K_E. This allows the kernel to consider attribute similarity rather than requiring exact matches.

$$[A_\times]_{(u_1,u_2),(v_1,v_2)} = K_V(u_1, u_2) \cdot K_E(\{u_1, v_1\}, \{u_2, v_2\}) \tag{3.10}$$

Neuhaus et al. (2009) restricts the adjacency matrix to contain only nodes that appear in the node mapping M of GED in equation 3.11.

$$A_\times = \begin{cases} \|((u_1, u_2), (v_1, v_2)) & \text{if } ((u_1, u_2), (v_1, v_2)) \in E_\times \\ \text{and } (u_1 \to u_2) \in M & \\ \text{and } (v_1 \to v_2) \in M & \\ 0 & \text{otherwise} \end{cases} \tag{3.11}$$

This modified adjacency matrix will be less dense than the original adjacency matrix of the product graph, since it contains only edges between nodes in the optimal node mapping from GED. As it is based on the modified random walk kernel for attributed graphs from Borgwardt et al. (2005), the resulting kernel is also only applicable to attributed graphs, which was the focus in Neuhaus and Bunke (2007). The final kernel value is computed using the same method as in the standard random walk kernel as defined in equation 3.4 or equation 3.5.

Because it depends on the GED node mapping, the random walk edit kernel is not positive semi-definite (PSD). Nevertheless, since it is closely related to the standard random walk kernel, which is PSD, the authors argue that the kernel can be successfully applied in practice (Neuhaus and Bunke 2007). The computation of GED dominates its computational complexity and therefore is equal to that of the trivial GED kernel. To make this kernel applicable to discretely labeled graphs and unlabeled graphs, the kernel will be adapted in the experimental evaluation in this study in section 4.4.2.

3.2.4. Other GED based Graph Kernels

Beyond the GED-based kernels discussed above, several additional approaches have been proposed in the literature, many of which originated with the work of Bunke and Neuhaus.

The zero graph kernel (Neuhaus and Bunke 2006b) selects one or more zero graphs from the dataset and computes similarities by comparing how far each input graph lies from these reference graphs. The kernel value between two graphs is derived from the difference in their distances to the zero graph and their direct edit distance. When multiple zero graphs are used the resulting kernel values are aggregated by summation or multiplication.

Using GED approximations Neuhaus and Bunke (2007) also introduces the convolution edit kernel (Neuhaus and Bunke 2006a) and the local matching kernel. Additionally, the maximum similarity kernel uses a GED-inspired and related similarity measure.

Two further GED-based kernels are the mapping distance kernels MDKS and MDKV proposed by Kataoka et al. (2018). They are built on the mapping distance (MD), which approximates GED by representing each node as a star structure consisting of the node and its immediate neighbors. Edit distances between all star pairs of two graphs are computed and an optimal assignment of these stars yields an approximate graph distance. The MDKS kernel converts this distance into a similarity via an exponential function while MDKV extends the representation to higher-order subtree feature vectors.

3.3. GED based KNN

A direct and conceptually simple approach to graph classification is to use the Graph Edit Distance as the distance metric in a K-Nearest Neighbors classifier. This approach is commonly used as a baseline for comparison to more complex GED-based graph classifiers (Neuhaus and Bunke 2007; Riesen and Bunke 2009b). Additionally, many studies on GED approximations evaluate the quality of GED approximation methods

by comparing the classification performance of the exact GED or other approximations of GED with the KNN classification performance of their proposed method (Riesen and Bunke 2009a; Riesen et al. 2015; Blumenthal and Gamper 2018). Riesen et al. (2015) proposed an ensemble version of GED-KNN using the Greedy GED approximation, which is subject to an arbitrary ordering choice, where multiple runs with randomized orderings vote on the final class label.
One disadvantage of using GED-based KNN is that while fitting runs in constant time, prediction requires computing the GED between the new graph and all training graphs, resulting in a computational complexity of $O(\mathcal{N} \cdot (n^n))$. Finding approaches to reduce this complexity for GED-based KNN is an active area of research (Abu-Aisheh et al. 2020).

3.4. Other Graph Classifiers

Beyond graph kernels and GED-based KNN, several additional approaches to graph classification exist. This section briefly highlights two representative methods relevant to context but outside the primary scope of this study.

In (Riesen and Bunke 2009b), an idea is proposed to construct **GED-based vector Representations** of graphs by selecting a set of prototype graphs from the dataset and computing the GED between each graph and the prototypes. This allows the use of standard vector space classifiers such as Linear Regression, SVMs, or Decision Trees on the graphs. The selection of prototypes is a crucial step in this approach as they must be representative of the dataset to ensure good performance. For that the authors propose 6 different prototype selection strategies including random selection, k-medoids clustering and spanning tree based selection. Based on this idea the authors further proposed different classifiers and approaches in Riesen and Bunke (2010) and Bunke and Riesen (2007).

Another approach recently gaining popularity is **Graph Neural Networks**. These are neural networks with specialized layers for capturing relationships and dependencies between nodes in a graph. The first layers are the message-passing layers, which aggregate information from neighboring nodes to update node representations similar to neighborhood aggregation approaches such as WL Kernels (Xu et al. 2019; Bronstein et al. 2021). These aggregations are then transformed into a fixed size by local and global pooling layers. In recent years these models have shown state-of-the-art performance on many graph classification tasks (Wu et al. 2020; Nikolentzos et al. 2021).

3.5. Model Complexities

Computational cost is an essential factor when comparing graph classifiers, as it determines both how long model training takes and how efficiently new graphs can be classified. While the literature review sections above introduce the theoretical complexities of each kernel in context, this section consolidates the relevant observations and

contrasts the models on a higher level. We distinguish between fitting and predicting complexity.

The fitting complexity of KNNs is constant since the model only stores the training graphs. For SVMs the complexity of fitting needs to be further distinguished in computing the kernel matrix and training the model on it. While training on the kernel matrix runs in cubic time, building the kernel matrix depends on the kernel and is often the dominant part of the computation.

For predicting new samples, GED-based KNNs need to compute the GED between the new sample and the whole training set, resulting in complexity of $O(\mathcal{N} \cdot n^n)$. For SVMs the detailed complexity is again kernel-dependent. However, most kernels only need to consider the support vectors in the training data, resulting in fewer computations and reducing the complexity. Table 3.1 summarizes the kernel matrix computation and prediction complexities of the different kernels.

Table 3.1.: Comparison of computational complexities of graph kernels

Kernel	Kernel computation	Prediction
Vertex Histogram Kernel	$O(\mathcal{N}^2 \cdot n)$	$O(s \cdot n)$
Weisfeiler-Lehman Subtree Kernel	$O(\mathcal{N} \cdot m + \mathcal{N}^2 \cdot n)$	$O(s \cdot n)$
Random walk Kernel	$O(\mathcal{N}^2 \cdot n^3)$	$O(s \cdot n^3)$
Trivial GED Kernel	$O(\mathcal{N}^2 \cdot n^n)$	$O(s \cdot n^n)$
Diffusion GED Kernel	$O(\mathcal{N}^3 + \mathcal{N}^2 \cdot n^n)$	$O(\mathcal{N} \cdot n^n + \mathcal{N}^2)$
Random walk Edit Kernel	$O(\mathcal{N}^2 \cdot n^n)$	$O(s \cdot n^n)$

To simplify it is approximately assumed that all graphs have n nodes and m edges.

How these complexities translate into practical runtimes is evaluated in the experimental study in chapter 4.

3.6. Related Experimental Studies

With the previous sections providing an overview of different graph classifiers, both GED-based and non-GED-based, this section now discusses related experimental studies from the literature.

In Zhang et al. (2018) a comprehensive survey of a wide range of graph kernels is presented on a large number of unlabeled and label-removed benchmark datasets from the TUDataset repository (Morris et al. 2020). The authors evaluate model accuracy using a 10-fold cross-validation setup, repeating the experiment 10 times with different random seeds to ensure robust results. Building on that, Kriege et al. (2020) uses the same setup for discretely labeled datasets. Neither study includes GED-based classifiers. Both studies find that the best-performing classifiers vary widely across datasets and that WL-Kernels are among the best-performing kernels on most datasets.

In Nikolentzos et al. (2021) a third study that used a similar setup was conducted. It also included GNNs alongside graph kernels on unlabeled, discretely labeled and contin-

uously attributed graph data, and additionally compared runtimes. They find that for unlabeled, continuously attributed data, GNNs achieve better average performance than most graph kernels. Especially the GIN architecture, which is a fusion of graph kernels and GNNs (Du et al. 2019), performs with the best average rank for both unlabeled and attributed graphs. At the same time, for labeled data, GNNs seem to perform at mid-field ranks, compared to the other graph kernels. Among the kernel methods they obtain results similar to those of the previously mentioned studies. In their runtime analysis the vertex histogram is the fastest model for computing the kernel matrix but the WL subtree kernel is also relatively quick compared to other methods. While none of these studies test GED-based classifiers Neuhaus and Bunke (2007) performs an experimental evaluation on multiple classifiers introduced in previous sections of this study. They test on multiple datasets using a simple methodology with continuously attributed graphs.

Classifiers are trained on a training set, hyperparameters are tuned on a validation set, and final performance is evaluated on a test set. In addition to their graph kernels GED-KNN is used as a baseline classifier. They observe that the GED-based KNN outperforms or matches the performance of the trivial GED-based kernels on almost all datasets. For the random walk edit and diffusion kernels they find that, while they manage to outperform their baselines on 2 datasets, they drastically underperform on other datasets. The best-performing model in their evaluation is the convolution edit kernel, which is omitted from testing in this study due to the lack of available implementations.

While these studies provide valuable insights into the performance of various graph classifiers none of them specifically focus on GED-based classifiers for discretely labeled and unlabeled graphs. This gap is addressed in the experimental study presented in the following chapter.

4. Experimental Design

To practically evaluate the GED-based classifiers introduced in the previous chapter, an experimental study is designed. The overall goal of this study is to evaluate the performance of the different GED-based graph classifiers and answer the research questions posed in section 1.1. This chapter therefore describes the experimental setup in detail in the following sections. The experimental study of the classifiers was conducted using python3 using different libraries for the implementation of the classifiers and the GED computations. The Concrete libraries used and some technical details are described in appendix A. The source code used to conduct the experiments is made publicly available on GitHub[1].

4.1. Datasets

The datasets used in this study are widely used benchmark datasets for graph classification. To cover a wide range of graph types and characteristics, datasets from various domains with different properties are selected. This allows us to evaluate the performance of the different GED-based classifiers across various types of graph data, and to see whether their performance varies with dataset characteristics. The datasets used in this study were obtained from the TUDataset repository[2] (Morris et al. 2020), which provides a wide range of benchmark datasets for graph classification tasks. Table 4.1 summarizes the properties and statistics of the selected datasets. All datasets are used exactly as provided in the TUDataset repository, only additionally using the datasets with removed labels. A short description of each dataset is provided in the Appendix A.1.

Table 4.1.: Statistics of the datasets used in the experimental study

Dataset	$\mathcal{N}$	Avg. n	Avg. m	num. Classes	Max Class Imbalance	Node Label	Edge Labels
MUTAG	188	17.93	19.79	2	1:98	✓	✓
PTC_FR	351	14.56	15.00	2	1:1.9	✓	✓
KKI	83	29.96	48.42	2	1:1.24	✓	-
BZR_MD	306	21.30	225.06	2	1:1.05	✓	✓
MSRC_9	221	40.58	97.94	8	1:1.58	✓	-
IMDB-MULTI	1500	13.00	65.94	3	1:1	-	-

[1] https://github.com/SimonESchumacher/Exploration-of-the-Potential-of-GED-based-Classifiers
[2] http://www.graphlearning.io/

4.2. Preprocessing

The preprocessing of the data mainly involved precomputing and storing the Graph Edit distances between all pairs of graphs in the datasets to save time during classifier training and evaluation, and to allow more computational effort to be devoted to accurate GED computation. This is done once per dataset, so that all classifiers using GED can then look up precomputed distances in shared memory rather than recomputing them for each iteration and each model. Due to the high computational complexity and the additional high number of graph pairs in some datasets, it is still not feasible to compute the exact GED distances between all graph pairs, even if precomputed. To address this issue, a two-step approach was used: compute the exact GED distances for as many graph pairs as possible, and use an approximate GED computation method for the remaining graph pairs.

First, approximate GED distances and node mappings were computed for all pairs of graphs in the datasets. This was done using the IPFP approximation algorithm (Leordeanu et al. 2009) from the gedlibpy library[3], which is a Python wrapper for the GEDLIB C++ library (Blumenthal et al. 2019, 2020). A short description of the IPFP algorithm is provided in the appendix in A.2. Second, a modified version of the exact GED implementation from the graph-edit-distance library[4] was used (Chang et al. 2020, 2022). This specific exact GED implementation uses A* search with a number of optimizations to compute exact GED distances (Chang et al. 2020). The modification made to the original implementation allows it to output not only the exact GED distance between two graphs, but also the optimal node mapping between the two graphs, which is required for the random walk edit kernel from section 3.2.3. Using this modified implementation, the exact GED distances and node mappings were computed for as many graph pairs as possible within a given time limit. If that time limit was exceeded, the computation was aborted, and the approximate GED distance and node mapping from the first step were used instead. Statistics on computation time, timeouts, and the mean absolute error (MAE) between approximate and exact values (for pairs where the latter was obtained) were recorded to monitor the quality of the resulting distance matrices. Because for some datasets a non-negligible portion of pairwise distances relied on the approximation fallback, the resulting distance matrices may introduce a small bias into the downstream classifiers. The time limit was usually 7 minutes, 30 seconds (450 seconds) for the GED computation per graph pair, with dataset-specific adjustments described in the appendix A.3 along with the complete results of these statistics. This pipeline provides a robust, efficient foundation for generating GED-based similarity measures for subsequent classifier exploration. To speed up the overall computation, the computations were parallelized using joblib (joblib developers 2025), allowing to utilize multiple CPU cores.

[3] https://github.com/pygraph/gedlibpy
[4] https://github.com/LijunChang/Graph_Edit_Distance

4.3. Nested Cross-Validation Setup

To obtain unbiased and accurate performance estimates, a 5 fold nested cross-validation scheme was employed. This involves an outer loop for final performance evaluation, an inner loop for hyperparameter tuning and 3 repetitions with different random seeds. The random seeds are predetermined to guarantee the same conditions for every model at every step. All splits in the inner and outer loops are stratified to preserve class structure.

In the outer loop, 4 folds are first used for hyperparameter tuning in the inner loop. The remaining fold is held out for the final evaluation of the best model from the inner loop. The inner loop also uses 5-fold cross-validation with a random search hyperparameter tuner to find the best hyperparameter configuration for each classifier. The hyperparameter tuner samples 100 hyperparameter configurations from the defined search spaces and trains and evaluates them on each of the 5 splits. The specific hyperparameters and the distributions from which they are sampled are detailed in section 4.4. The configuration with the highest average score across the 5 folds is selected for the outer loop.

In the outer loop, that configuration is tested on the 5th fold. The final score for the model will be the average score of all 5 evaluations in the outer loop and all 3 repetitions. A detailed description of the scoring metrics is found in section 4.5 and the libraries used to implement the setup is described in the appendix in section A.4.1.

4.4. Classifier Specifications

In the experimental study, the classifiers specified in the previous chapters were implemented.

As baseline classifiers, without using GED, SVMs using the vertex histogram kernel (VH) from 3.1.1, the Weisfeiler-Lehman subtree kernel (WL-ST) from 3.1.2, and a random walk kernel (RW) from 3.1.3 were implemented. For the random walk kernel, due to runtime constraints, only three datasets could be evaluated because of its high computational complexity, and less possibility of precomputation.

As GED-based classifiers, a GED-based KNN classifier (GED-KNN) was implemented, described in 2.3, as well as SVMs using the trivial GED-based kernels from 3.2.1 (Triv-GED), the diffusion kernel from 3.2.2 (Diff-GED), and the random walk edit kernel from 3.2.3 (RWE). The trivial GED-based SVM and random walk edit kernel implementations were additionally extended, as described in the following subsections. A detailed description of the libraries used to implement the classifiers can be found in the appendix in A.4.2.

4.4.1. Trivial GED-Based Kernel with Bandwidth λ

In the original theoretical description of the trivial GED-based kernels in section 3.2.1, the functions k_1, k_2, k_3, k_4 are presented without any adjustable parameters (Neuhaus

and Bunke 2007). In this study however, these kernels are extended by adding a bandwidth parameter, λ, to control the sensitivity of the kernels to GED distances. Especially for the nonlinear kernel functions k_3 and k_4, this bandwidth parameter could increase the kernels' expressiveness by allowing us to adjust the influence of GED distances on kernel values. The modified kernel functions with the bandwidth parameter λ are defined as follows in equation 4.1:

$$\begin{aligned} k_1(G_1, G_2) &= -(\lambda \cdot GED(G_1, G_2))^2 \\ k_2(G_1, G_2) &= -\lambda \cdot GED(G_1, G_2) \\ k_3(G_1, G_2) &= tanh(-\lambda \cdot GED(G_1, G_2)) \\ k_4(G_1, G_2) &= exp(-\lambda \cdot GED(G_1, G_2)) \end{aligned} \tag{4.1}$$

For the kernel functions k_1 and k_2, the bandwidth parameters are not expected to have a significant influence on the performance, since they are linear transformations of the GED distances, just linearly scaling the distances by λ and λ^2 respectively. They are included for completeness and to allow a fair comparison between all four trivial GED-based kernels.

4.4.2. Random Walk Edit Kernel for Discretely Labeled and Unlabeled Graphs

The random walk edit kernel from section 3.2.3 is by default only defined for continuously attributed graphs (Neuhaus and Bunke 2007). To use it on discretely labeled and unlabeled graphs, the kernel is adapted by returning to the original definition of the product graph 3.2 and adding the node mapping constraint to the adjacency matrix definition. The resulting definition for the adjacency matrix for discretely labeled graphs is shown in 4.2.

$$A_\times = \begin{cases} 1 & \text{if } \{(u_1, u_2), (v_1, v_2)\} \in E_\times \\ & \quad \text{and } (u_1 \to u_2) \in M \\ & \quad \text{and } (v_1 \to v_2) \in M \\ 0 & \text{otherwise} \end{cases} \tag{4.2}$$

How this modified adjacency matrix behaves is illustrated in figure 4.1, for two graphs G and G'. In the figure, the product graph $G_\times$ and the GED node mapping M of G and G' are used to define the modified adjacency matrix.

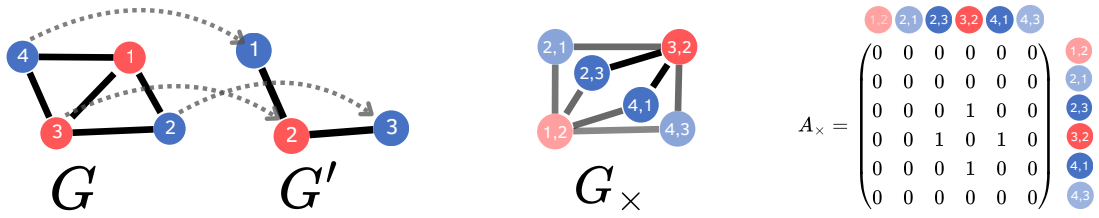


Figure 4.1.: Adjacency matrix for the RWE kernel between two graphs

The GED node mapping of the two graphs in the figure is denoted in equation 4.3

$$M = \{v_1 \to \epsilon, v_2 \to v_3', v_3 \to v_2', v_4 \to v_1'\} \tag{4.3}$$

4.4.3. Hyperparameter Search Spaces

For each classifier used in the experimental study, a set of hyperparameters was defined, which were tuned using random search in the inner loop of the nested cross-validation setup. The specific hyperparameters and their search spaces for each classifier are detailed in table 4.2.

Table 4.2.: Consolidated Hyperparameter Search Spaces for Graph Classifiers

Classifier / Kernel	**Hyperparameter**	**Search Space**
SVM Vertex Histogram Kernel (VH)	C	$[0.0005, 10]^{\ddagger}$
SVM WL Subtree Kernel (WL-ST)	Iterations h C	$[1, 7]^{\dagger}$ $[0.0005, 10]^{\ddagger}$
SVM Random Walk Kernel (RW)	Decay factor λ Kernel Type Walk Length l C	$[0.005, 0.95]^{\ddagger}$ $\{geometric, exponential\}$ $[2, 6]^{\dagger} \cup [\infty]$ $[0.0005, 10]^{\ddagger}$
KNN (GED)	K weighting	$[1, 7]^{\dagger}$ $\{uniform, distance\}$
SVM Trivial GED Kernel (Triv-GED)	Kernel Type Bandwidth λ C	$\{k1, k2, k3, k4\}$ $[0.01, 100]^{\ddagger}$ $[0.0005, 10]^{\ddagger}$
SVM Diffusion GED Kernel (Diff-GED)	Diffusion Type Decay factor λ Iterations k C	$\{exponential, laplacian\}$ $[0.005, 0.95]^{\ddagger}$ $[2, 6]^{\dagger}$ $[0.0005, 10]^{\ddagger}$
SVM Random Walk Edit Kernel (RWE)	Decay factor λ Walk Length l C	$[0.005, 0.95]^{\ddagger}$ $[2, 6]^{\dagger} \cup [\infty]$ $[0.0005, 10]^{\ddagger}$

† Search space was sampled uniformly on a log scale (log-uniform distribution).
‡ Search space was sampled uniformly on a linear scale (uniform distribution).

4.5. Evaluation Methodology

This section describes in detail how the experiment was evaluated to answer the three research questions posed in section 1.1. This involves explicit metrics and data recorded during the experiments, and how they are used to answer the research questions.

The first research question (**RQ 1**) focuses on the classification performance of the different GED-based classifiers. This is of obvious interest, since the primary purpose of these classifiers is to perform well on graph classification tasks.
As a primary metric, F1-macro was chosen, as it ensures a balanced evaluation across all classes in the dataset. The hyperparameter tuner therefore selected models in the inner loop based on F1-macro, and the final reported performance is also evaluated using F1-macro. Since many other studies use accuracy as their primary metric (Kriege et al. 2020; Nikolentzos et al. 2021; Zhang et al. 2018), an additional run with accuracy as the main metric was conducted to allow better comparison with these studies.
In addition, precision and recall are recorded to provide a more comprehensive view of the classifiers' performance. The concrete definitions of the used metrics are provided in the appendix in section A.6. Since the performance of the classifiers can vary across folds and repetitions, as well as across other potential splits, the accuracy and F1-macro scores are reported along with their 95% confidence intervals. This provides context on how much the performance varies, which helps especially in comparing different classifiers.
The second research question (**RQ 2**) focuses on the cost-to-performance ratio of the different GED-based classifiers. This is an important aspect, since GED computations are known to be computationally expensive. Therefore, analyzing the classification performance in relation to computational cost is another critical aspect in understanding the competitiveness and practicality of GED-based classifiers.
Because the GED values are precomputed, the runtimes measured in classification experiments do not include GED computation runtimes for the GED-based classifiers. As an alternative, estimated runtimes are reconstructed that include the GED computation times. This is done by measuring the average time for computing the GED between two graphs in a dataset during precomputation. This is used to estimate the total time required for GED computations during both fitting and prediction, based on the number of GED computations a classifier has to perform. The fitting runtime is estimated in equation 4.4, while the prediction runtime is estimated in equation 4.5, with $\bar{t}_{GED}$ being the average GED computation time. The number of support vectors $||S||$ were averaged in evaluation and are listed in the appendix in table B.4.

$$T_{fit} = R_{fit} + \left(\frac{N \cdot (N-1)}{2} \cdot \bar{t}_{GED} \right) \quad (4.4) \qquad T_{pred} = R_{pred} + \left(||S|| \cdot \bar{t}_{GED} \right) \quad (4.5)$$

The third research question (**RQ 3**) focuses on the main factors influencing the performance of GED-based classifiers. Analyzing this is interesting because it can provide insights into how the different hyperparameters of the classifiers influence performance and what patterns emerge.
Here, the data retrieved from the hyperparameter tuners from the nested cross-validation setup is analyzed, to see which hyperparameters have the most significant influence on the performance of the GED-based classifiers. Another point of analysis is looking at the kernel matrices produced by the different classifiers.

5. Experimental Results

In this chapter the results of the experimental study are presented and analyzed, and the research questions from section 1.1 are answered.

5.1. Classification performance

This section presents the core classification performance, measured by the F1-macro score, to address the fundamental question of model competitiveness (**RQ 1**). The results are split into the datasets with discretely labeled graphs in table 5.1 and unlabeled datasets in table 5.2, to analyze both types separately. Scores for accuracy as well as the precision and recall can be found in the appendix B.1.

5.1.1. Labeled Datasets

Table 5.1 shows the F1-macro scores for the different classifiers on the discretely labeled datasets.

Table 5.1.: F1-Macro Scores on Labeled Datasets

Model	VH	WL-ST	RW	GED-KNN	Triv-GED	Diff-GED	RWE
MUTAG	**0.831** ± 0.025	0.770 ± 0.026	0.699 ± 0.028	**0.846** ± 0.023	**0.838** ± 0.027	**0.842** ± 0.038	**0.805** ± 0.030
PTC_FR	0.491 ± 0.021	0.501 ± 0.020	0.396 ± 0.001	**0.584** ± 0.026	0.462 ± 0.025	0.448 ± 0.018	0.401 ± 0.005
KKI	0.381 ± 0.029	0.360 ± 0.013	–	**0.477** ± 0.050	**0.441** ± 0.038	**0.429** ± 0.051	**0.397** ± 0.038
BZR_MD	**0.716** ± 0.019	0.574 ± 0.021	0.638 ± 0.022	0.656 ± 0.020	0.656 ± 0.050	0.655 ± 0.022	0.621 ± 0.022
MSRC_9	**0.850** ± 0.019	0.813 ± 0.017	–	0.820 ± 0.018	**0.854** ± 0.015	**0.845** ± 0.020	0.113 ± 0.018
AVG.	0.654	0.604	-	0.677	0.650	0.644	0.467

Best performing models are highlighted bold and underlined, while models performing within the confidence interval of the best model are only bold.

To visualize the comparative performance, Figure 5.1 illustrates the F1-Macro scores across all labeled datasets.

Analyzing the results in table 5.1 on the discretely labeled graph datasets, a few observations can be made. For most datasets, multiple models perform similarly well, often within each other's confidence intervals, indicating that their performance is not

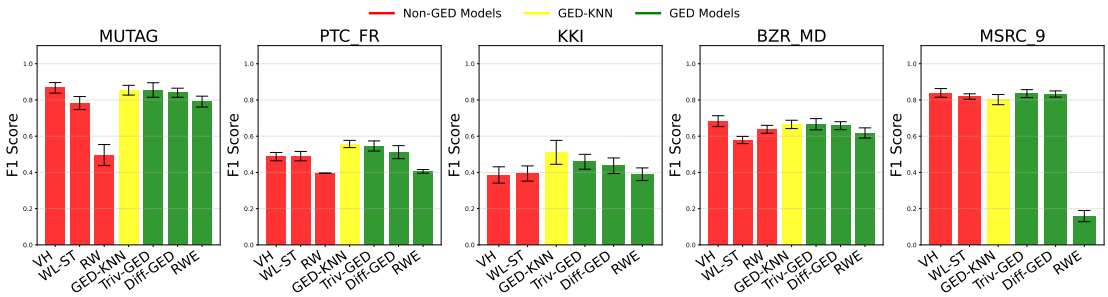


Figure 5.1.: F1-Macro Scores on Labeled Datasets

significantly different. The model with the best average scores is the GED KNN. The vertex histogram kernel, trivial GED SVM and the diffusion GED SVM also perform well on average with only slightly lower average scores. Particularly, the trivial and diffusion GED SVMs perform have very similar results on the datasets.

The worst performing models are the random walk kernel and the random walk edit kernel. On the 3 Datasets where the Random Walk kernel could be evaluated, it has significantly lower average scores than the other models. The Random Walk Edit kernel performs worse on almost all datasets compared to the other models, especially on the MSRC_9 dataset, where it has a drastically lower score than all other models.

5.1.2. Unlabeled Datasets

The results on the datasets with removed labels or naturally unlabeled graphs, are shown in table 5.2.

Table 5.2.: F1_Macro unlabeled Datasets

Model	VH	WL-ST	RW	GED-KNN	Triv-GED	Diff-GED	RWE
MUTAG	**0.817** ± 0.021	**0.843** ± 0.027	0.393 ± 0.001	**0.833** ± 0.017	**0.830** ± 0.025	**0.812** ± 0.018	**0.812** ± 0.020
PTC_FR	0.396 ± 0.001	0.411 ± 0.009	0.395 ± 0.002	**0.551** ± 0.025	0.417 ± 0.015	0.406 ± 0.011	0.400 ± 0.005
KKI	0.350 ± 0.020	**0.453** ± 0.043	–	**0.462** ± 0.051	**0.416** ± 0.035	**0.422** ± 0.036	0.388 ± 0.026
BZR_MD	**0.622** ± 0.028	**0.575** ± 0.023	0.353 ± 0.037	0.552 ± 0.026	**0.620** ± 0.029	**0.608** ± 0.031	**0.622** ± 0.019
MSRC_9	0.143 ± 0.014	0.101 ± 0.023	–	**0.174** ± 0.019	0.132 ± 0.017	0.131 ± 0.016	0.121 ± 0.024
IMDB-MULTI	0.245 ± 0.019	**0.498** ± 0.012	–	0.407 ± 0.025	0.472 ± 0.012	0.436 ± 0.012	0.363 ± 0.023
AVG.	0.429	0.480	-	0.497	0.481	0.469	0.451

Best performing models are highlighted bold and underlined.
Models whose confidence intervals overlap with the best performing model are only bold.

To visualize the comparative performance, Figure 5.2 illustrates the F1-Macro scores across all unlabeled datasets. When removing the labels, table 5.2 shows the performance

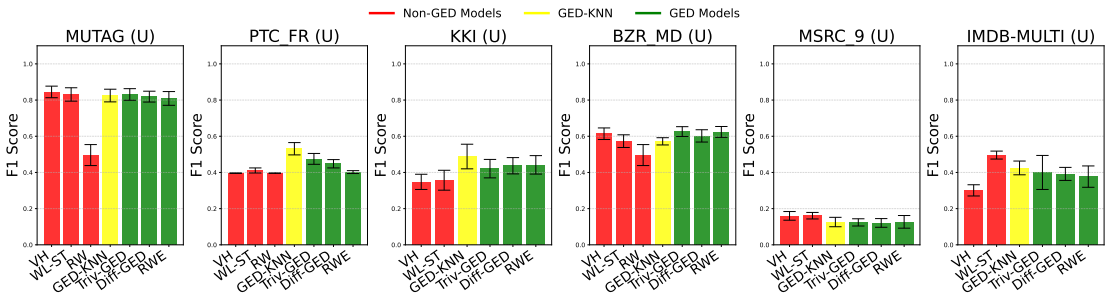


Figure 5.2.: F1-Macro Scores on Unlabeled Datasets

on most datasets decreases for all models, as expected. The gap between the best and worst models also becomes smaller, resulting in more models performing within each other's confidence intervals. Despite this compression of differences, GED-KNN maintains the highest average ranking across unlabeled datasets, followed closely by the trivial GED kernel SVM and the Weisfeiler-Lehman subtree kernel. The relative ranking of the strongest models therefore remains broadly similar to the labeled case, although the Weisfeiler-Lehman subtree kernel becomes more competitive once label information is absent. On Dataset level, the Weisfeiler-Lehman subtree kernel is also performing best on The IMDB-MULTI dataset, which was naturally unlabeled, with a significant margin with no model overlapping its confidence interval. Comparing model scores to their performance on labeled datasets, the vertex histogram kernel sees the largest drop in performance when labels are removed. The random walk edit kernel, does not seem to be effected by the removal of labels, as its performance remains roughly the same on all datasets.
On the MSRC_9 Dataset, the models experience the largest drops in performance, with all models achieving very low F1-macro scores around 0.13-0.16. This indicates the labels were particularly important for classification on this dataset. Additionally the dataset has 8 different classes, which makes the classification task more difficult in general.

5.1.3. Answer to RQ 1

Comparing the results from the GED based models to the baseline models, it is evident that the GED-KNN and the trivial GED kernel are competitive with the baseline models. On all datasets, they receive similar scores to the baseline models on both labeled and unlabeled datasets. The only exception is the IMDB-MULTI dataset, where both models perform worse than the Weisfeiler-Lehman subtree kernel. Overall, this supports the notion that GED serves as an effective structure-based approach for discretely labeled graph classification tasks.

A critical finding is the consistent underperformance of the random walk edit kernel across all labeled datasets and still shows rather poor performance on unlabeled datasets.

This suggests that the approach of leveraging node mapping information for random walk kernels may not be as effective as directly utilizing GED values. It may therefore not be regarded as a competitive method for graph classification tasks. Furthermore, the slight advantage of the trivial GED kernel and GED-KNN over the diffusion GED kernel indicates that more complex GED-based kernels do not necessarily yield better performance. The best approach for using GED for graph classification, therefore seems to be to directly use the GED values either in a KNN classifier or in a trivial GED kernel SVM.

5.2. Computational efficiency analysis

This section presents the runtimes for both fitting and prediction of the classifiers,and analyzes the models from a computational efficiency perspective, to answer RQ 2. An important note here is, that these are reconstructed estimates of the runtimes, as described in section 4.5, since GEDs where precomputed. Important context to these runtimes is the average time it took to compute the exact GED between two graphs. For the prediction times the number of support vectors are relevant the models uses after training. These are shown in the appendix in table B.4, to not distract too much from the main results.

5.2.1. Model Fitting runtimes

Table 5.3 shows the estimated fitting times for all classifiers on all datasets.

Table 5.3.: Model Fitting Times

Model	WL-SP	VH	RW	GED-KNN	Triv-GED	Diff-GED	RWE
MUTAG	0.034s	0.01s	3m 4s	0.001s	1h 23m 8s	1h 23m 8s	1h 23m 8s
PTC_FR	0.171s	0.078s	8m 40s	0.005s	153h 12m 24s	153h 12m 24s	153h 12m 25s
KKI	0.029s	0.008s	–	0.001s	12h 28m 45s	12h 28m 45s	12h 28m 45s
BZR_MD	0.171s	0.078s	–	0.001s	43h 58m 7s	43h 58m 7s	43h 58m 8s
MSRC_9	0.256s	0.096s	–	0.003s	>27000h*	>27000h*	>27000h*
MUTAG (U)	0.088s	0.03s	54.771s	0.003s	21h 12m 22s	21h 12m 22s	21h 12m 22s
PTC_FR (U)	0.148s	0.048s	9m 43s	0.005s	39h 44m	39h 44m	39h 44m 1s
KKI (U)	0.057s	0.023s	–	0.002s	12h 48m 46s	12h 48m 46s	12h 48m 46s
BZR_MD (U)	0.171s	0.078s	–	0.001s	13.316s	13.242s	14.302s
MSRC_9 (U)	0.256s	0.096s	–	0.003s	>27000h*	>27000h*	>27000h*
IMDB-MULTI (U)	1.141s	0.566s	–	0.003s	200h	200h	200h

*GED computations timed out after 100 min per graph pair, more in the appendix A.3
(U) indicates unlabeled datasets.

The fitting times observed in table 5.3 align well with the theoretical computational

complexities discussed in section 3.5. They show that for the GED based Kernel SVMs, the fitting times can amount to multiple days, even for semi-small datasets like PTC_FR or BZR_MD. Even for the small MUTAG dataset, the models need over an hour to be fit. This also aligns well with its $O(\mathcal{N}^2 \cdot n^n)$ kernel matrix computation complexity. The GED-KNN is the fastest model, since it does not need to fit a model, but only store the training samples. Additionally, the random walk kernel also has higher fitting times compared to the other non GED kernels. This is also expected, due to its cubic complexity. The Weisfeiler-Lehman subtree kernel and the Vertex Histogram kernel have negligible fitting times in comparison.

5.2.2. Model Prediction runtimes

Table 5.4 shows the estimated prediction times for all classifiers on all datasets. The

Table 5.4.: Model Prediction Times

Model	WL-SP	VH	RW	GED-KNN	Triv-GED	Diff-GED	RWE
MUTAG	0s	0s	2.406s	1m 6s	3.459s	1m 6s	4.793s
PTC_FR	0.001s	0s	3.918s	1h 5m 39s	26m 43s	1h 5m 39s	26m 5s
KKI	0.001s	0s	–	22m 41s	4m 32s	22m 41s	3m 38s
BZR_MD	0.001s	0s	–	35m 10s	10m 55s	35m 10s	6m
MSRC_9	0.002s	0.001s	–	>300h*	>138h 30m*	>300h*	>138h 40m*
MUTAG (U)	0.001s	0s	0.727s	16m 57s	4m 58s	16m 57s	3m
PTC_FR (U)	0.148s	0.048s	4m 52s	17m 1s	6m 46s	17m 1s	6m 40s
KKI (U)	0.001s	0s	–	23m 17s	9m 29s	23m 17s	9m 4s
BZR_MD (U)	0.001s	0s	–	0.109s	0.021s	0.109s	0.02s
MSRC_9 (U)	0.002s	0.001s	–	>293h 20m*	>146h 40m*	>293h 20m*	>145h 20m*
IMDB-MULTI (U)	0.001s	0s	–	20m	1m 28s	20m	1m 27s

*GED computations timed out after 100 min per graph pair, more in the appendix A.3.
(U) indicates unlabeled datasets.

observed patterns for prediction runtimes in table 5.4 are also as expected.

The GED-KNN and diffusion GED kernel SVM need minutes to hours to predict a single new sample on most datasets, since they both need to compute $\mathcal{N}$ GED values for each new prediction. The trivial GED kernel and random walk edit kernel have significantly lower, however still very high prediction times, with roughly between half and a tenth of the time needed by the GED-KNN and diffusion GED kernel. Between them, the kernels alternate in which one is faster, depending on the dataset, however the differences are not very large. At the same time, the WL kernel and histogram kernel seem to have negligible, almost instant prediction times.

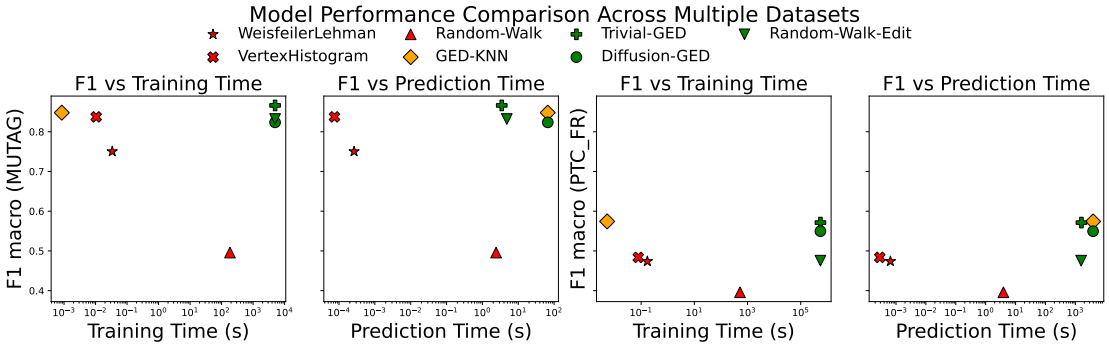


Figure 5.3.: F1 Macro and Estimated Runtimes on log Scale for MUTAG and PTC_FR Datasets

5.2.3. Analysis and answer to RQ 2

Overall the observed runtimes make it evident that the GED based classifiers have a infeasibly high computational cost. To quantify that, the GED based classifiers need more than one million times longer for fitting and prediction than the Vertex Histogram kernel, on almost all datasets. To further provide a perspective on that tradeoff, figure 5.3 visualizes the f1-macro scores against the estimated runtimes for the models for training and predicting new samples. Here, only the results for the MUTAG and PTC_FR datasets are shown, to keep the graphs readable. The other datasets show similar patterns and are shown in the appendix B.4.

In response to **RQ 2**, while the GED based classifiers can achieve competitive classification performance, their high computational cost due to the GED computations makes them unpractical for many applications, especially with larger datasets or graphs. The runtimes observed are, just simply put, too high for practical use, as they need several hours to train or predict on relatively small datasets. Especially, when there are other classifiers available, that can achieve similar performance with in comparison negligible computational cost, there is little reason to use GED based classifiers in practice. In Neuhaus and Bunke (2007), the graph datasets used were significantly smaller, with an average of about 5 nodes per graph, making the GED computations more manageable. However even in their study, GED computations posed a limitation.

For applications where GED based classification is still required and the computational cost for training a model is less of a concern than the prediction time, the trivial GED kernel SVM seems to be the best choice. It achieves better performance as the RWE Kernel SVM, while benefiting from lower runtimes compared to the GED KNN. For applications where the prediction time is less of a concern, but rather the training time, the GED KNN seems to be a good choice. It achieves good performance, while having no training time at all. The Diffusion GED kernel SVM seems to be the least favorable choice, since it has the combination of high training and prediction times, while achieving slightly worse performance. In reality however, these scenarios are quite rare.

5.3. Hyperparameter Influence Analysis

Our third research question focuses on understanding how different hyperparameters influence the performance of the GED based classifiers.

Generally, hyperparameters can significantly influence the performance of machine learning models. The effects of the different hyperparameters are analyzed in this section to understand their behavior and strengths. In the following, we will review the different GED-based classifiers and analyze the impact of their hyperparameters on performance.

5.3.1. GED-based KNN

The GED-based KNN only has the number of neighbors K and the weighting scheme as hyperparameters, however, the effect they have on the performance seems quite small for the datasets analyzed. For some datasets, a slight advantage of some number of neighbors can be observed, but overall, the performance seems quite robust to the choice of these hyperparameters. The effect the number of neighbors k has on the performance of the datasets can be seen in figure 5.4.

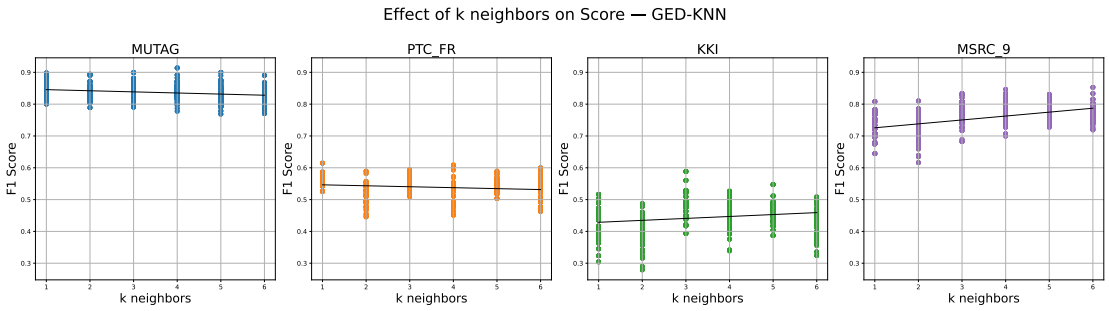


Figure 5.4.: GED-based KNN Hyperparameter Influence

The differences are quite small Nevertheless, it can be observed that for the KKI dataset, 3 neighbors seem to be a sweet spot, and the MSRC_9 dataset seems to benefit from higher numbers of neighbors.

5.3.2. Trivial GED Kernel SVM

The trivial GED kernel-based SVM has the regularization parameter C, as well as the bandwidth parameter λ, and the choice of the kernel function as hyperparameters. The regularization parameter C seems to have practically no influence on the f1-macro scores on most datasets. Here, especially the choice of the kernel function seems to have significant influence on performance and behavior. This can be seen in figure 5.5, the distributions of the performance scores for the 4 kernel functions. For the remaining datasets similar observations can be made, as seen in the appendix in figures B.5 and B.6. Overall the performance of the k1 and k2 kernel functions seem to be more robust,

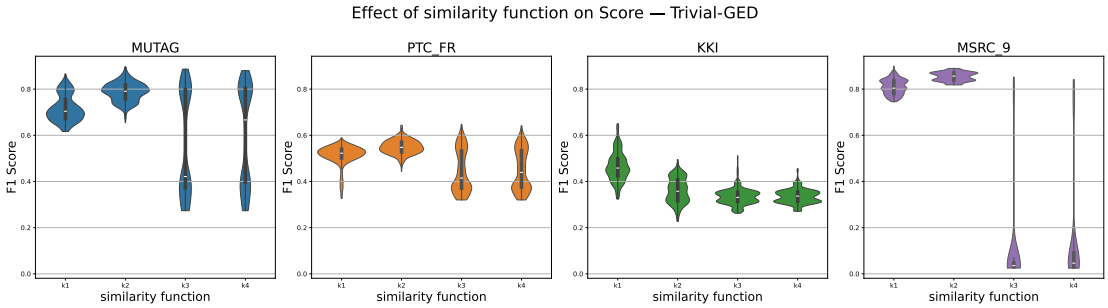


Figure 5.5.: Violin plots showing the performance of the different kernel functions of the trivial GED kernel SVM on MUTAG

while the k3 and k4 kernel functions show a high variance in performance. The hypothesis of the bandwidth parameter λ having a significant influence on the performance, especially for the k3 and k4 kernel functions, can be confirmed when analyzing deeper. For low values of λ, for k3 and k4 the performance is significantly better, while for higher values of λ, the performance decreases significantly as seen in figure 5.6. Interestingly, the bandwidth parameter seems to also have an effect on the k1 and k2 kernels for some datasets, even though they are linear functions. Figure 5.6 shows the effect of the bandwidth parameter λ on the performance on the PTC_FR dataset

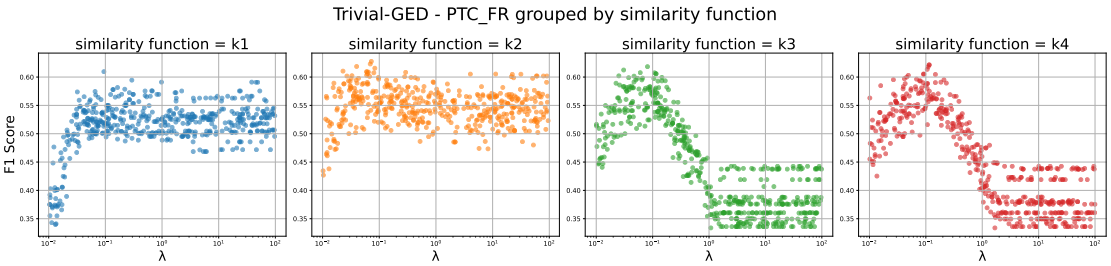


Figure 5.6.: Influence of bandwidth parameter λ on trivial GED kernel performance on PTC_FR on log Scale

For all kernel functions, very low values of λ seem to result in a rather poor performance on the PTC_FR dataset. For k1 and k2 the performance than seems to settle at $\lambda = 0.1$, while for k3 and k4 the performance peaks there, and falls off again for higher values of λ. On the MUTAG dataset in figure 5.7, something similar can be observed.

On the MUTAG dataset, very low values of λ do not seem to have a negative impact on performance, at least not in the tested range. The expected effect for k3 and k4 however is observed here again. For k1 and k2 there also seems to be a slight decrease in performance for higher values of λ. On other datasets, the behavior of k3 and k4 can also be observed, while the behavior of k1 and k2 seems to vary more between datasets.

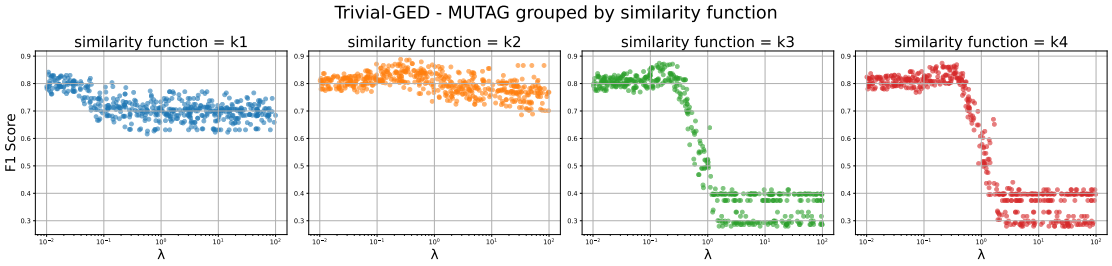


Figure 5.7.: Influence of bandwidth parameter λ on trivial GED kernel performance on MUTAG on log Scale

Their figures are shown in the appendix B.7 and B.8.
To visualize why the bandwidth parameter has such a dramatic effect on k3 and k4, kernel matrices for different values of λ are analyzed. In figure 5.8, the kernel matrices for the k4 kernel function with different values of λ on the MSRC_9 dataset are shown, along with one matrix for the k1 kernel for comparison. The graphs are ordered by class, to better visualize the class separability in the kernel matrices.

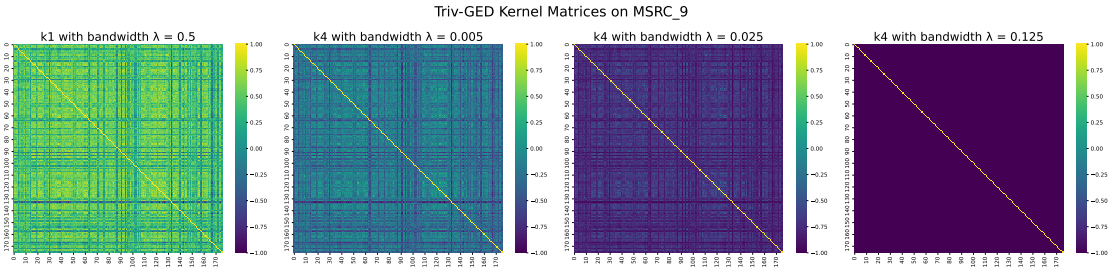


Figure 5.8.: Trivial GED Kernel k4 Kernel Matrices for different values of λ on MSRC_9

λ is increased by a factor of 5 for each matrix from left to right. For the lowest value of $\lambda = 0.005$, the kernel matrix looks similar to the k1 kernel matrix, with clear class structure visible. With the increasing values of λ the structure fades, and pretty quickly, the kernel values become indistinguishable.

5.3.3. Diffusion GED Kernel SVM

The diffusion GED kernel SVM has similar hyperparameters as the trivial GED kernel SVM. Analyzing the performance scores here, only slight influences in the combinations of the iterations and the decay factor λ can be observed on the datasets. Interestingly, the effect seems to vary between datasets. For most datasets, the variance between the results with increasing iterations t seems to increase, which could be explained by overfitting. For some datasets like the MSRC_9 dataset, the performance seems to decline with increasing decay factor λ, with high numbers of iterations t, as shown in figure 5.9.

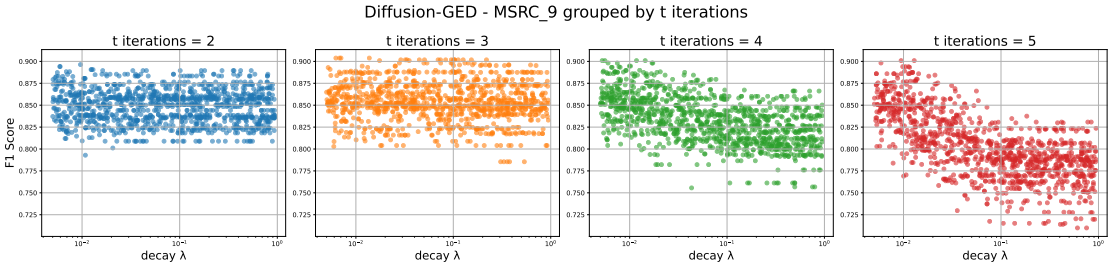


Figure 5.9.: effect of iterations and decay factor on log scale for MSRC_9

For increasing values of the depth parameter t, the performance seems to decline with higher values of λ, when plotted on a logarithmic scale. This hints that the information in the lower iterations is more useful for classification, while the higher iterations add less useful information. When the decay factor λ is higher and the depth parameter t is increased, that information from the later iterations removes some of the expressiveness of the earlier iterations, leading to worse performance. The fact that the earlier iterations seem much more expressive hints, that the effect of the diffusion process might rather remove information from the GED values instead of adding useful information. A kernel matrix, which is more similar to the one of the trivial GED kernel, might be beneficial instead. The strength of this effect, however, seems to vary between datasets, not being very pronounced on other datasets like MUTAG or PTC_FR as shown in the appendix in figures B.9 and B.10.

5.3.4. Random Walk Edit Kernel SVM

For the random walk edit kernel, similar effects can be observed as for the diffusion GED kernel. Alone the hyperparameters do not seem to have a significant influence on the performance of the classifier. However, the combination of the decay factor λ and the maximum path walk length seems to have some influence on the performance. On the MUTAG dataset this behavior is shown in figure 5.10, where for the higher path lengths the performance seems to decline with higher values of λ.

The kernel matrices of the different combinations in figure 5.11 show that with infinite path length and a high decay factor, the kernel matrix becomes almost uniform, losing all class structure.

For the IMDB-MULTI dataset, a similar behavior can be observed, and also PTC_FR and the BZR_MD datasets show similar, however, much slighter effects. On the MSRC_9 dataset, the effect seems to be reversed, with higher values of λ resulting in better performance for higher path lengths, however with huge variance. The plots for the other datasets are shown in the appendix in figures B.11 and B.12.

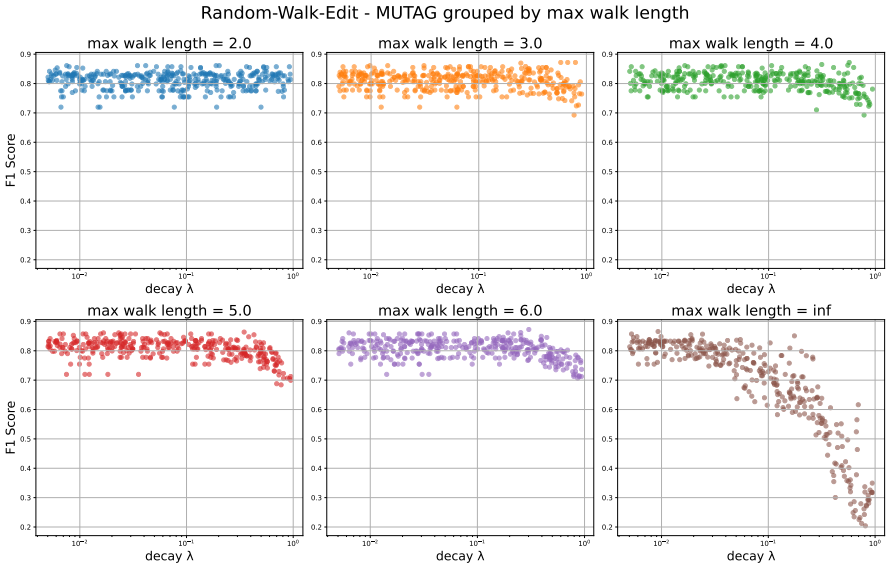


Figure 5.10.: Effect of max path length and decay factor on log scale for MUTAG

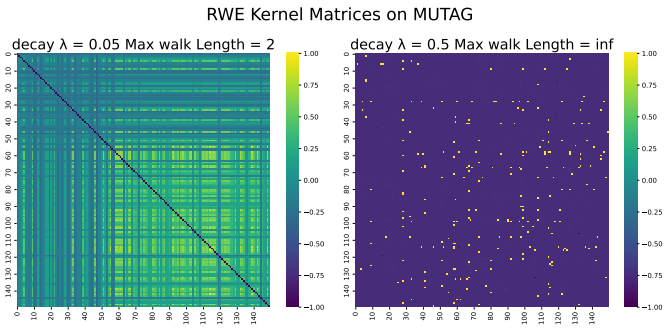


Figure 5.11.: Random Walk Edit Kernel k4 Kernel Matrices for different values of λ on MUTAG

5.3.5. Discussion

Summarizing the findings and answering **RQ 3**, overall the influence of many hyperparameters is limited. The GED based KNN seems to be quite robust to its hyperparameters, while for the trivial GED kernel SVM, the choice of the kernel function and the bandwidth parameter λ can have a significant influence on the performance. For the diffusion kernel and the random walk edit kernels, the rather rougher approximations of their kernel functions seem to perform better.

Given the slightly superior performance of the trivial GED kernels and GED-based KNN over the diffusion and random walk edit kernels, and the observation that simpler configurations of the latter tend to perform better on most datasets, we hypothesize that more complex ways of using GED values do not necessarily improve performance.

The kernel matrices produced by the simpler configurations are closer to those of the trivial GED kernel, which performed best overall. This suggests that the additional information extracted from GED values may not be useful for classification and can even remove important information. However, this is only a hypothesis that would need further investigation to be confirmed.

6. Conclusion

This chapter concludes the present thesis by summarizing the research contributions made and discussing the limitations of the study. It also proposes potential directions for future work.

The core research of this thesis is focused on filling the gap with of a missing comprehensive analysis of different GED-based classifiers on both discretely labeled and unlabeled graph datasets. Additionally, the lack of a direct comparison of different GED based classifiers with non GED based classifiers, as well as the lack of an analysis of the computational cost of GED based classifiers in relation to their performance, is addressed.

6.1. Research Contributions

The present thesis provides both a theoretical and practical contribution, answering the research questions posed in section 1.1. A comprehensive overview of different GED based graph kernels and K-NN classifiers is given. Additionally, the trivial GED kernel is extended by the bandwidth parameter, to allow for better control of the influence of different GED values on the kernel values and the random walk edit kernel is extended, to work with discretely labeled graphs. The practical evaluation introduces a novelty experimental evaluation, directly comparing GED based classifiers on a variety of discretely labeled and unlabeled graph datasets. Previous studies on GED based classifiers were only conducted on continuously attributed graphs. This fills a significant gap in the literature, as many real world graph datasets are either discretely labeled or unlabeled. The experimental evaluation also provides a direct comparison of GED based classifiers, with non GED baseline classifiers, both by performance and computational cost, thus evaluating the practicality of the classifiers as well. Lastly, a deeper analysis of the influences of different hyperparameters on the performance of GED based classifiers is conducted, to better understand the factors influencing their performance. These contributions together provide a comprehensive understanding of GED based classifiers both theoretically and practically from multiple perspectives.

6.2. Summary of Findings

Across the datasets analyzed, the GED based classifiers were able to match and in some cases exceed the performance of the non GED baseline classifiers. Especially the GED based KNN showed a strong performance, while the random walk edit kernel SVM proved to be rather weak in comparison. However, these performances are overshadowed

by the extremely high computational cost of the GED based classifiers. The GED based classifiers would need several hours to days to be trained in a practical setting, and are also very slow at predicting new samples. This makes them impractical for real applications, especially compared to non GED based classifiers, which can achieve similar performance with in comparison negligible computational cost. Analyzing the influence of different hyperparameters on the performance of the GED based classifiers, it was found that many hyperparameters have only a limited influence on performance. It showed that the introduction of the bandwidth parameter was beneficial for the trivial GED kernel SVM, while for the diffusion GED kernel and random walk edit kernel, simpler configurations tended to perform better.

6.3. Limitations

The analysis conducted in this study has several limitations that should be acknowledged. These primarily stem from the issue of computational cost and resource constraints as well as a limited scope of the study. Firstly the choice of datasets is limited both by individual graph sizes and number of graphs in a dataset, due to the high computational cost of GED computations and kernel matrix computations. Additionally, for the pre-computations a timeout to prevent excessive runtimes on some graph pairs was needed, leading to some GED values being approximated instead of exact. Consequently, while robust and conclusive, the study suffers some losses due to computational effort. While more extensive setups and hardware could strengthen these results partially, they are expected to run into similar problems fast.

6.4. Future Work

Based on the findings and limitations of this study, several approaches for future work can be proposed. The issue of computational effort was a central limitation and concern for practical applications of GED based classifiers. Future work could conduct similar studies as the present one, but using different approximation methods of GED instead of exact values. For future work on developing new GED based classifiers, rather simple approaches of directly using the GED values should be prioritized, since they seem to perform better than more complex approaches. Overall, future work on GED based classifiers for improved graph classification seems to be of limited practical use when computational cost issues cannot be solved, when compared to other graph classification methods, like non GED based graph kernels or graph neural networks.

Bibliography

Abu-Aisheh, Z., R. Raveaux, and J.-Y. Ramel (2020). Efficient k-nearest neighbors search in graph space. *Pattern Recognition Letters 134*, 77–86. Applications of Graph-based Techniques to Pattern Recognition.

Abu-Aisheh, Z., R. Raveaux, J.-Y. Ramel, and P. Martineau (2015, Jan). An exact graph edit distance algorithm for solving pattern recognition problems. In *4th International Conference on Pattern Recognition Applications and Methods 2015*, Lisbon, Portugal.

Bennett, K. P. and E. J. Bredensteiner (2000). Duality and geometry in svm classifiers. In *ICML*, Volume 2000, pp. 57–64.

Blumenthal, D. B., N. Boria, J. Gamper, S. Bougleux, and L. Brun (2020). Comparing heuristics for graph edit distance computation. *The VLDB Journal 29*(1), 419–458.

Blumenthal, D. B., S. Bougleux, J. Gamper, and L. Brun (2019). Gedlib: A c++ library for graph edit distance computation. In D. Conte, J.-Y. Ramel, and P. Foggia (Eds.), *Graph-Based Representations in Pattern Recognition*, Cham, pp. 14–24. Springer International Publishing.

Blumenthal, D. B. and J. Gamper (2018). Improved lower bounds for graph edit distance. *IEEE Transactions on Knowledge and Data Engineering 30*(3), 503–516.

Borgwardt, K., N. Schraudolph, and S. Vishwanathan (2006). Fast computation of graph kernels. In B. Schölkopf, J. Platt, and T. Hoffman (Eds.), *Advances in Neural Information Processing Systems*, Volume 19, pp. 1449–1456. MIT Press.

Borgwardt, K. M., C. S. Ong, S. Schoenauer, S. Vishwanathan, A. J. Smola, and H.-P. Kriegel (2005). Protein function prediction via graph kernels. *Bioinformatics 21*(suppl_1), i47–i56.

Bougleux, S., L. Brun, V. Carletti, P. Foggia, B. Gaüzère, and M. Vento (2017). Graph edit distance as a quadratic assignment problem. *Pattern Recognition Letters 87*, 38–46. Advances in Graph-based Pattern Recognition.

Bronstein, M. M., J. Bruna, T. Cohen, and P. Velickovic (2021). Geometric deep learning: Grids, groups, graphs, geodesics, and gauges. *CoRR abs/2104.13478*.

Bunke, H. and K. Riesen (2007). A family of novel graph kernels for structural pattern recognition. In L. Rueda, D. Mery, and J. Kittler (Eds.), *Progress in Pattern Recognition, Image Analysis and Applications*, Volume 4746 of *Lecture Notes in Computer Science*, Berlin, Heidelberg, pp. 20–31. Springer Berlin Heidelberg.

Burges, C. J. (1998). A tutorial on support vector machines for pattern recognition. *Data mining and knowledge discovery 2*(2), 121–167.

Cervantes, J., F. Garcia-Lamont, L. Rodríguez-Mazahua, and A. Lopez (2020). A comprehensive survey on support vector machine classification: Applications, challenges and trends. *Neurocomputing 408*, 189–215.

Chang, C.-C. and C.-J. Lin (2011). Libsvm: A library for support vector machines. *ACM transactions on intelligent systems and technology (TIST) 2*(3), 1–27.

Chang, L., X. Feng, X. Lin, L. Qin, W. Zhang, and D. Ouyang (2020). Speeding up ged verification for graph similarity search. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pp. 793–804. IEEE.

Chang, L., X. Feng, K. Yao, L. Qin, and W. Zhang (2022). Accelerating graph similarity search via efficient ged computation. *IEEE Transactions on Knowledge and Data Engineering 35*(5), 4485–4498.

Cortes, C. and V. N. Vapnik (1995). Support-vector networks. *Machine Learning 20*, 273–297.

Cover, T. and P. Hart (1967). Nearest neighbor pattern classification. *IEEE transactions on information theory 13*(1), 21–27.

Craddock, R. C., G. James, P. E. Holtzheimer III, X. P. Hu, and H. S. Mayberg (2012). A whole brain fmri atlas generated via spatially constrained spectral clustering. *Human Brain Mapping 33*(8), 1914–1928.

Cunningham, P. and S. J. Delany (2021). K-nearest neighbour classifiers-a tutorial. *ACM computing surveys (CSUR) 54*(6), 1–25.

da Costa-Luis, C. O. (2019). 'tqdm': A fast, extensible progress meter for python and cli. *Journal of Open Source Software 4*(37), 1277.

Das, R. and M. Soylu (2023). A key review on graph data science: The power of graphs in scientific studies. *Chemometrics and Intelligent Laboratory Systems 240*, 104896.

Debnath, A. K., R. L. Lopez de Compadre, G. Debnath, A. J. Shusterman, and C. Hansch (1991, feb). Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. correlation with molecular orbital energies and hydrophobicity. *Journal of Medicinal Chemistry 34*(2), 786–797.

Djemai, S., B. Brahmi, and M. O. Bibi (2016). A primal-dual method for svm training. *Neurocomputing 211*, 34–40. SI: Recent Advances in SVM.

Du, S. S., K. Hou, B. Póczos, R. Salakhutdinov, R. Wang, and K. Xu (2019). Graph neural tangent kernel: Fusing graph neural networks with graph kernels. In *Neural Information Processing Systems*.

Dudani, S. A. (1976). The distance-weighted k-nearest-neighbor rule. *IEEE Transactions on Systems, Man, and Cybernetics* (4), 325–327.

Fix, E. and J. L. Hodges (1989). Discriminatory analysis. nonparametric discrimination: Consistency properties. *International Statistical Review / Revue Internationale de Statistique 57*(3), 238–247.

Fuchs, M. and K. Riesen (2021). Matching of matching-graphs - a novel approach for graph classification. In *2020 25th International Conference on Pattern Recognition (ICPR)*, pp. 6570–6576.

Gao, X., B. Xiao, D. Tao, and X. Li (2010). A survey of graph edit distance. *Pattern Analysis and Applications 13*(1), 113–129.

Gärtner, T., P. Flach, and S. Wrobel (2003). On graph kernels: Hardness results and efficient alternatives. In *Learning theory and kernel machines: 16th annual conference on learning theory and 7th kernel workshop, COLT/kernel 2003, washington, DC, USA, August 24-27, 2003. proceedings*, pp. 129–143. Springer.

Hagberg, A., P. J. Swart, and D. A. Schult (2007, 12). Exploring network structure, dynamics, and function using networkx. In *Proceedings of the 7th Python in Science Conference (SciPy)*. Los Alamos National Laboratory (LANL).

Harris, C. R., K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant (2020, sep). Array programming with NumPy. *Nature 585*(7825), 357–362.

Helma, C., R. D. King, S. Kramer, and A. Srinivasan (2001). The predictive toxicology challenge 2000–2001. *Bioinformatics 17*(1), 107–108.

Hunter, J. D. (2007). Matplotlib: A 2d graphics environment. *Computing in Science & Engineering 9*(3), 90–95.

joblib developers, T. (2025, may). joblib.

Kashima, H., K. Tsuda, and A. Inokuchi (2003). Marginalized kernels between labeled graphs. In *Proceedings of the 20th international conference on machine learning (ICML-03)*, pp. 321–328.

Kataoka, T., E. Shiotsuki, and A. Inokuchi (2018). Graph classification with mapping distance graph kernels. In M. De Marsico, G. S. di Baja, and A. Fred (Eds.), *Pattern Recognition Applications and Methods*, Cham, pp. 21–44. Springer International Publishing.

Kondor, R. I. and J. Lafferty (2002). Diffusion kernels on graphs and other discrete structures. In *Proceedings of the 19th international conference on machine learning*, Volume 2002, pp. 315–322.

Kriege, N. M., P.-L. Giscard, and R. Wilson (2016). On valid optimal assignment kernels and applications to graph classification. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett (Eds.), *Advances in Neural Information Processing Systems*, Volume 29, pp. 1623–1631. Curran Associates, Inc.

Kriege, N. M., F. D. Johansson, and C. Morris (2020). A survey on graph kernels. *Applied Network Science 5*(1), 6.

Lee, H., Z. Tu, M. Deng, F. Sun, and T. Chen (2006). Diffusion kernel-based logistic regression models for protein function prediction. *Omics: a journal of integrative biology 10*(1), 40–55.

Lehman, A. A. and B. Y. Weisfeiler (1968). Reduction of a graph to a canonical form and an algebra which appears in the process. *NTI Ser 2*(9), 12–16.

Leordeanu, M., M. Hebert, and R. Sukthankar (2009). An integer projected fixed point method for graph matching and map inference. In Y. Bengio, D. Schuurmans, J. Lafferty, C. Williams, and A. Culotta (Eds.), *Advances in Neural Information Processing Systems*, Volume 22, pp. 1114–1122. Curran Associates, Inc.

Levenshtein, V. I. (1965). Binary codes capable of correcting deletions, insertions, and reversals. In *Doklady Akademii Nauk*, Volume 163, pp. 845–848. Russian Academy of Sciences.

Marukatat, S. (2023). Tutorial on pca and approximate pca and approximate kernel pca. *Artif Intell Rev 56*, 5445–5477.

Mercer, J. (1909). Xvi. functions of positive and negative type, and their connection the theory of integral equations. *Philosophical transactions of the royal society of London. Series A, containing papers of a mathematical or physical character 209*(441-458), 415–446.

Morris, C., N. M. Kriege, F. Bause, K. Kersting, P. Mutzel, and M. Neumann (2020). Tudataset: A collection of benchmark datasets for learning with graphs. In *ICML 2020 Workshop on Graph Representation Learning and Beyond (GRL+ 2020)*.

Neuhaus, M. and H. Bunke (2004). An error-tolerant approximate matching algorithm for attributed planar graphs and its application to fingerprint classification. In A. Fred, T. M. Caelli, R. P. W. Duin, A. C. Campilho, and D. de Ridder (Eds.), *Structural, Syntactic, and Statistical Pattern Recognition*, Volume 3138 of *Lecture Notes in Computer Science*, Berlin, Heidelberg, pp. 180–189. Springer Berlin Heidelberg.

Neuhaus, M. and H. Bunke (2005). Graph-based multiple classifier systems a data level fusion approach. In F. Roli and S. Vitulano (Eds.), *Image Analysis and Processing – ICIAP 2005*, Berlin, Heidelberg, pp. 479–486. Springer Berlin Heidelberg.

Neuhaus, M. and H. Bunke (2006a). A convolution edit kernel for error-tolerant graph matching. In *18th International Conference on Pattern Recognition (ICPR'06)*, Volume 4, pp. 220–223.

Neuhaus, M. and H. Bunke (2006b). Edit distance-based kernel functions for structural pattern classification. *Pattern Recognition 39*(10), 1852–1863. Similarity-based Pattern Recognition.

Neuhaus, M. and H. Bunke (2007). *Bridging the Gap between Graph Edit Distance and Kernel Machines*. WORLD SCIENTIFIC.

Neuhaus, M., K. Riesen, and H. Bunke (2006). Fast suboptimal algorithms for the computation of graph edit distance. In D.-Y. Yeung, J. T. Kwok, A. Fred, F. Roli, and D. de Ridder (Eds.), *Structural, Syntactic, and Statistical Pattern Recognition*, Berlin, Heidelberg, pp. 163–172. Springer Berlin Heidelberg.

Neuhaus, M., K. Riesen, and H. Bunke (2009). Novel kernels for error-tolerant graph classification. *Spatial Vision 22*(5), 425–441.

Neumann, M., R. Garnett, C. Bauckhage, and K. Kersting (2016). Propagation kernels: efficient graph kernels from propagated information. *Machine learning 102*(2), 209–245.

Nikolentzos, G., G. Siglidis, and M. Vazirgiannis (2021). Graph kernels: A survey. *Journal of Artificial Intelligence Research 72*, 943–1027.

Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research 12*, 2825–2830.

Pržulj, N. (2007, 01). Biological network comparison using graphlet degree distribution. *Bioinformatics 23*(2), e177–e183.

Riesen, K. and H. Bunke (2009a). Approximate graph edit distance computation by means of bipartite graph matching. *Image and Vision Computing 27*(7), 950–959. 7th IAPR-TC15 Workshop on Graph-based Representations (GbR 2007).

Riesen, K. and H. Bunke (2009b). Graph classification based on vector space embedding. *International Journal of Pattern Recognition and Artificial Intelligence 23*(06), 1053–1081.

Riesen, K. and H. Bunke (2010). *Graph Classification and Clustering Based on Vector Space Embedding*. WORLD SCIENTIFIC.

Riesen, K., M. Ferrer, R. Dornberger, and H. Bunke (2015). Greedy graph edit distance. In P. Perner (Ed.), *Machine Learning and Data Mining in Pattern Recognition. MLDM 2015*, Volume 9166 of *Lecture Notes in Computer Science*, pp. 3–16. Springer, Cham.

Riesen, K., M. Ferrer, and A. Fischer (2015). Building classifier ensembles using greedy graph edit distance. In F. Schwenker, F. Roli, and J. Kittler (Eds.), *Multiple Classifier Systems*, Cham, pp. 125–134. Springer International Publishing.

Sanfeliu, A. and K.-S. Fu (1983). A distance measure between attributed relational graphs for pattern recognition. *IEEE Transactions on Systems, Man, and Cybernetics SMC-13*(3), 353–362.

Schölkopf, B. (2000). The kernel trick for distances. *Advances in neural information processing systems 13*, 301–307.

Schulz, T. H., T. Horváth, P. Welke, and S. Wrobel (2022). A generalized weisfeiler-lehman graph kernel. *Machine Learning 111*(7), 2601–2629.

Shervashidze, N., P. Schweitzer, E. J. Van Leeuwen, K. Mehlhorn, and K. M. Borgwardt (2011). Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research 12*(9), 2539–2561.

Siglidis, G., G. Nikolentzos, S. Limnios, C. Giatsidis, K. Skianis, and M. Vazirgiannis (2020). Grakel: A graph kernel library in python. *Journal of Machine Learning Research 21*(54), 1–5.

Smola, A. J. and B. Schaulkopf (2004). A tutorial on support vector regression. *Statistics and computing 14*(3), 199–222.

Sutherland, J. J., L. A. O'brien, and D. F. Weaver (2003). Spline-fitting with a genetic algorithm: A method for developing classification structure- activity relationships. *Journal of chemical information and computer sciences 43*(6), 1906–1915.

Syriopoulos, P. K., N. G. Kalampalikis, S. B. Kotsiantis, and M. N. Vrahatis (2025). k nn classification: a review. *Annals of mathematics and artificial intelligence 93*(1), 43–75.

Virtanen, P., R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors (2020). SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods 17*, 261–272.

Vishwanathan, S. V. N., N. N. Schraudolph, R. Kondor, and K. M. Borgwardt (2010). Graph kernels. *The Journal of Machine Learning Research 11*, 1201–1242.

Waskom, M. L. (2021). seaborn: statistical data visualization. *Journal of Open Source Software 6*(60), 3021.

Wes McKinney (2010). Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman (Eds.), *Proceedings of the 9th Python in Science Conference*, pp. 56 – 61.

Wu, Z., S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu (2020). A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems 32*(1), 4–24.

Xu, K., W. Hu, J. Leskovec, and S. Jegelka (2019). How powerful are graph neural networks? In *International Conference on Learning Representations*.

Yanardag, P. and S. Vishwanathan (2015). Deep graph kernels. In *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*, pp. 1365–1374.

Yoo, A., M. Jette, and F. Grondona (2009). Slurm: Simple Linux Utility for Resource Management. *Parallel and Distributed Computing Practices (PDCP) 10*(4), 44–60.

Zhang, Y., L. Wang, and L. Wang (2018). A comprehensive evaluation of graph kernels for unattributed graphs. *Entropy 20*(12), 984.

A. Implementation Details

In this Appendix, some implementation details of the GED based classifiers are provided, which were not discussed in the main text.

A.1. Dataset Descriptions

The following datasets were used in the experimental evaluation of this study.
The **MUTAG** dataset (Debnath et al. 1991) is relatively small, consisting of 188 mutagenic graphs with an average of 17.93 nodes and 19.79 edges per graph and belonging to two classes: mutagenic and non-mutagenic. The graphs in this dataset are discretely labeled, with node labels indicating different atom types (e.g. Carbon, Nitrogen, Oxygen) and edge labels indicating different bond types (e.g. single, double, triple).

The **PTC_FR** dataset (Helma et al. 2001) contains 344 organic molecules, which are classified by their rodent carcinogenicity. It was published as part of the Predictive Toxicology Challenge (PTC) in 2000, for obtaining models that predict the outcome of biological tests for the carcinogenicity of chemicals using information related to chemical structure only. The graphs in this dataset have discrete node and edge labels. **KKI** It is the smallest dataset in terms of the number of graphs, with only 83. It is part of the BrainNet dataset collection (Craddock et al. 2012), which contains graphs representing brain networks. The graphs in the KKI dataset are of brain structures of people with or without ADHD[1]. It has only discretely labeled nodes representing different brain regions, but no edge labels.

BZR_MD contains 306 chemical compounds (ligands for the benzodiazepine receptor), classified by their binding affinity to the receptor (Sutherland et al. 2003). It has discretely labeled nodes and edges that represent different atom and bond types. An interesting note about this dataset is that it has very dense graphs, with an average of 225.06 edges per graph, despite having only 21.30 nodes per graph.

MSRC_9 is a state-of-the-art real-world dataset in semantic image processing, containing images that are represented by conditional Markov random fields (Neumann et al. 2016). A more in-depth description of the dataset can be found in the MSRC-9 dataset documentation. The graphs in this dataset have discrete node labels, but no edge labels. The 221 graphs in the dataset belong to 8 classes, each representing a different object.

IMDB-MULTI is a popular benchmark dataset in the field of Social Networks. It contains 1500 graphs with unlabeled nodes and edges, representing ego networks of actors/actresses, with each graph belonging to one of three genres: Sci-Fi, Comedy or

[1] https://github.com/TrustAGI-Lab/graph_datasets

Romance (Yanardag and Vishwanathan 2015). It is a larger dataset, and the only one that, by standard, is an entirely unlabeled graph dataset.

A.2. IPFP Description

The Integer Projected Fixed Point (IPFP) algorithm (Leordeanu et al. 2009) is an optimization-based approximation algorithm for the graph edit distance. These work by iteratively improving the node mapping between two graphs to minimize the overall edit cost. Starting from an initial mapping (based on heuristics or random assignment), the algorithm continuously improves the mapping by solving a relaxed version of the assignment problem, and then projecting the solution back to a valid integer assignment (Bougleux et al. 2017). The process is repeated until convergence, i.e. when no further improvements can be made to the mapping. The quality of the approximation depends on the initial mapping and whether it converges to a good solution. An in-depth description is out of scope for this thesis, but details of the algorithm can be found in Leordeanu et al. (2009) and Bougleux et al. (2017). The IPFP algorithm was chosen for this study due to its good balance between accuracy and computational efficiency.

A.3. GED Computation Details

To ensure the quality of the GED computations and measuring accurate runtime estimates, while not exceeding available computational resources, several measures were taken during the GED computation process. As discussed in section 4.2, a timeout was set for each GED computation, to prevent excessive runtimes on graph pairs, where the exact GED computation would take too long. This timeout was set to 7.5 minutes per GED computation. To monitor the accuracy of the approximate GED computation and the overall kernel matrix, the number of pairs hitting the timeout and the average deviation of the approximate GED calculations from the exact GED values for the pairs that did not hit the timeout was measured. However, for the MSRC_9 dataset, both with and without labels, within the 7.5-minute timeout, no exact GED computations could be completed, leading to all graph pairs requiring the approximate GED computation. Even with an extended timeout of 100 minutes, no exact GED computations could be completed for this dataset. This is due to the size of the graphs in this dataset with an average of about 40 nodes. For the IMDB-MULTI dataset the timeout had to be reduced to 60 seconds due to the high number of graph pairs, leading to an overall excessive computation time. The results of the resulting statistics can be found in table A.1.

A.4. Libraries Used

The experimental evaluation in this study was implemented using Python 3.12.11. For the implementations of different modules many libraries were used. The detailed imple-

Table A.1.: GED Computation Statistics per Dataset

Dataset	# Pairs	Timeouts	MAE of approx. GED	avg. Time (s)	avg. GED
MUTAG	17578	1	4.2152	0.443	18.06
PTC_FR	61425	74	3.8421	4.454	26.81
KKI	3403	788	17.6092	16.225	77.11
BZR_MD	46665	1209	4.7496	14.070	110.99
MSRC_9	24310	24310	?	>6000.000	107.18
MUTAG (U)	17578	450	5.6952	6.786	14.6
PTC_FR (U)	61425	1498	4.5678	3.649	21.57
KKI (U)	3403	1268	18.1808	21.178	91.51
BZR_MD (U)	46665	0	0	0.00	100.2
MSRC_9 (U)	24310	24310	?	>6000.000	78.43
IMDB-MULTI (U)	1124250	74200	7.4026	0.480	

(U) indicates unlabeled datasets.

mentations can be found in the provided code repository[2]. Version infos of the libraries used can be found in the requirements.txt file in the code repository.

A.4.1. Nested-Cross Validation and Hyperparameter Tuning

For hyperparameter tuning and model training, the scikit-learn library (Pedregosa et al. 2011) was used, providing efficient implementations of classifiers and tools for model selection and evaluation. Additionally, for processing scores and other logic as well as logging and storing results, standard Python libraries such as numpy (Harris et al. 2020), pandas (Wes McKinney 2010), SciPy (Virtanen et al. 2020) and joblib (joblib developers 2025) were used.

A.4.2. Libraries Used for Classifier Implementations

To implement the classifiers, the scikit-learn library (Pedregosa et al. 2011) was used, which provides efficient implementations of SVMs and KNN classifiers. For the baseline graph kernels, the GraKel library (Siglidis et al. 2020) was used, which provides efficient implementations of many popular graph kernels for use with SVMs from scikit-learn. Specifically, the node histogram kernel, the Weisfeiler-Lehman subtree kernel, and the standard random walk kernel implementations from GraKel were used. For the GED-based kernels, custom implementations were done using numpy (Harris et al. 2020) and NetworkX (Hagberg et al. 2007) for graph handling and numerical computations. In some cases involving computationally intensive functions, the SciPy library (Virtanen et al. 2020) was used for efficient implementations.

[2] https://github.com/SimonESchumacher/Exploration-of-the-Potential-of-GED-based-Classifiers

A.4.3. Additional Libraries

In addition to the libraries used in the main modules of the practical evaluation, some additional libraries were used for specific tasks. For plotting and visualizations, the Matplotlib (Hunter 2007) and Seaborn (Waskom 2021) libraries were used. Other libraries used include tqdm (da Costa-Luis 2019) for progress bars.

A.5. Computational Resources

The computational workload was distributed between a dedicated university workstation, utilized primarily for code development and initial small-scale trials, and a high-performance computing (HPC) cluster managed by the DWS Research Group.

The university workstation provided a stable environment for iterative development and debugging. It was equipped with an Intel® Core™ i7-9700K CPU operating at 3.60 GHz and 64 GiB of RAM, running Ubuntu Linux. The limit to 8 cores made it unsuitable for large-scale experiments such as exact GED computations and hyperparameter tuning. These experiments were therefore executed on the DWS research cluster[3], which provided access to multiple nodes with high core counts and large memory capacities. For resource allocation, Slurm (Yoo et al. 2009) was used, allowing for efficient distribution of tasks across available nodes; however, this resulted in a range of different CPU architectures being used. Here, the GED precomputations were executed, utilizing up to 80 parallel CPU cores and 200 GiB of RAM for memory-intensive tasks. For these computations, the typical CPU used were AMD EPYC 7713P or AMD EPYC 9474F processors. The model training was parallelized across 15 CPU cores, corresponding to the outer cross-validation folds. Here the nodes typically featured dual-socket Intel® Xeon® E5-2640 v2, E5-2640 v3 or Silver 4114 processors, running at 2.20 to 2.60 GHz.

A.6. Metrics Used

For evaluating the classification performance, the f1_macro score from scikit-learn (Pedregosa et al. 2011) was used, which computes the F1 score for each class and averages them, treating all classes equally regardless of their support. Along with the f1_macro score results, the corresponding scores for precision and recall where also recorded and reported. In a second run, accuracy was also used as an optimization metric for hyperparameter tuning, to allow comparisons with other studies that report accuracy as the main metric. The purpose of this section is to define these metrics. All metrics are based on the definition of true positives (TP), false positives (FP), true negatives (TN) and false negatives (FN) for each class C_i.

- True positives (TP): Number of instances correctly predicted as class C_i.
- False positives (FP): Number of instances incorrectly predicted as class C_i.

[3] https://www.uni-mannheim.de/dws/

- True negatives (TN): Number of instances correctly predicted as not class C_i.
- False negatives (FN): Number of instances incorrectly predicted as not class C_i.

Based on these definitions, precision and recall are defined in equations A.1 and A.2 respectively.

$$\mathcal{P} = \frac{TP_i}{TP_i + FP_i} \quad \text{(A.1)} \qquad \mathcal{R} = \frac{TP_i}{TP_i + FN_i} \quad \text{(A.2)}$$

the F1 score for each class C_i is defined as the harmonic mean of precision and recall, defined in equation A.3. Accuracy is simply describing the overall proportion of correctly classified instances, defined in equation A.4.

$$F1\_macro = \frac{1}{C} \sum_{i=1}^{C} \frac{2 \cdot \mathcal{P}_i \cdot \mathcal{R}_i}{\mathcal{P}_i + \mathcal{R}_i} \quad \text{(A.3)} \qquad Acc = \frac{TP + TN}{TP + TN + FP + FN} \quad \text{(A.4)}$$

B. Additional Experimental Results

B.1. Results for Additional Metrics

In addition to the f1_macro scores presented in the main results, here are the accuracy scores for all models on all datasets are presented for completeness. The table B.1 shows the accuracy scores for the labeled and unlabeled datasets. The results were obtained using the same nested cross validation setup as described in section 4.3, with accuracy set as the optimization metric for the hyperparameter tuner. Primarily, these results are provided to allow comparisons to other studies, which often report accuracy as the main metric.

Table B.1.: Accuracy Scores

Dataset	VH	WL-ST	RW	GED-KNN	Triv-GED	Diff-GED	RWE
MUTAG	**0.857** ± 0.030	0.793 ± 0.030	0.739 ± 0.026	**0.841** ± 0.021	**0.858** ± 0.023	**0.860** ± 0.033	**0.832** ± 0.023
PTC_FR	**0.683** ± 0.010	**0.671** ± 0.014	0.655 ± 0.002	0.654 ± 0.014	0.644 ± 0.011	0.641 ± 0.015	0.656 ± 0.003
KKI	**0.482** ± 0.045	**0.486** ± 0.042	–	0.449 ± 0.044	**0.550** ± 0.025	**0.543** ± 0.041	**0.538** ± 0.036
BZR_MD	**0.700** ± 0.029	0.608 ± 0.019	**0.688** ± 0.023	**0.656** ± 0.022	**0.669** ± 0.033	**0.697** ± 0.024	0.624 ± 0.039
MSRC_9	**0.891** ± 0.015	**0.876** ± 0.014	–	0.829 ± 0.020	**0.881** ± 0.019	**0.884** ± 0.017	0.170 ± 0.019
MUTAG (U)	**0.846** ± 0.031	**0.850** ± 0.035	0.655 ± 0.002	**0.835** ± 0.021	**0.849** ± 0.026	**0.853** ± 0.031	**0.826** ± 0.034
PTC_FR (U)	**0.655** ± 0.002	**0.653** ± 0.006	**0.655** ± 0.021	**0.662** ± 0.015	**0.656** ± 0.003	**0.650** ± 0.006	**0.654** ± 0.002
KKI (U)	**0.509** ± 0.029	**0.504** ± 0.048	–	**0.475** ± 0.051	**0.542** ± 0.039	**0.526** ± 0.032	**0.547** ± 0.042
BZR_MD (U)	**0.626** ± 0.021	**0.607** ± 0.026	0.466 ± 0.029	0.565 ± 0.021	**0.625** ± 0.019	**0.630** ± 0.022	**0.617** ± 0.018
MSRC_9 (U)	**0.171** ± 0.017	0.137 ± 0.016	–	**0.184** ± 0.023	0.144 ± 0.014	**0.153** ± 0.013	**0.163** ± 0.020
IMDB-MULTI (U)	0.335 ± 0.012	**0.508** ± 0.010	–	0.377 ± 0.018	0.483 ± 0.007	0.460 ± 0.010	0.366 ± 0.023
Average	0.614	0.609	–	0.593	0.627	0.627	0.545

(U) indicates unlabeled datasets. Models tuned for accuracy.

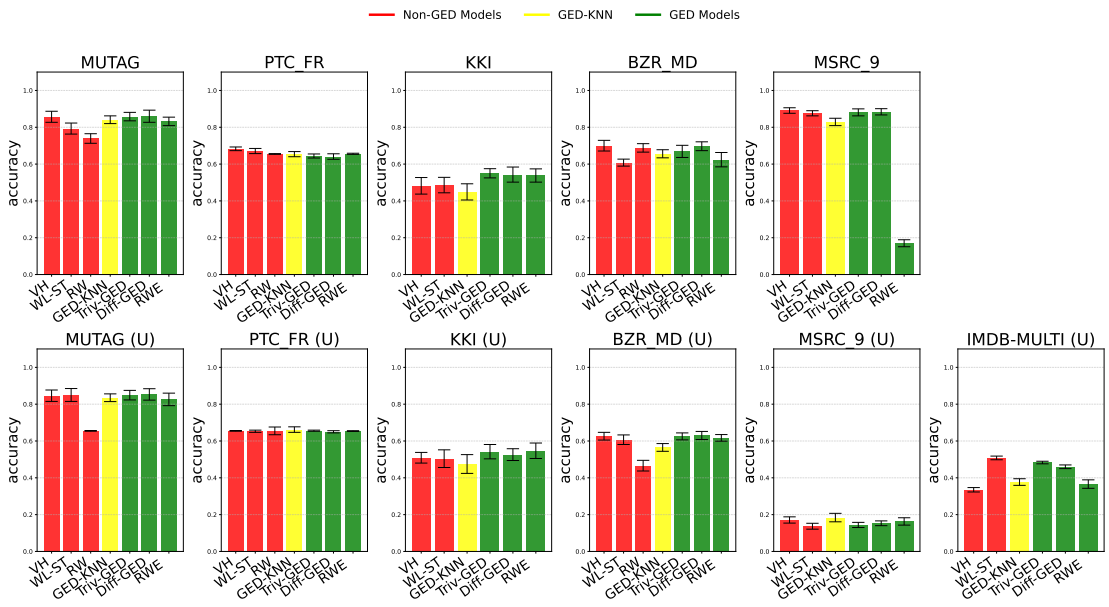


Figure B.1.: Bar Chart for Accuracy Scores of GED-based Classifiers

Additionally, in table B.2 and B.3 the precision and recall scores for all models on all datasets are presented for completeness. Table for precision on labeled and unlabeled datasets

Table B.2.: Precision on labeled and unlabeled datasets:

Dataset	VH	WL-ST	RW	GED-KNN	Triv-GED	Diff-GED	RWE
MUTAG	0.849	0.779	0.723	0.849	0.844	**0.856**	0.818
PTC_FR	0.714	**0.720**	0.328	0.600	0.514	0.549	0.381
KKI	0.365	0.309	–	**0.487**	0.425	0.409	0.376
BZR_MD	**0.722**	0.623	0.652	0.665	0.633	0.687	0.638
MSRC_9	0.856	0.809	–	0.853	0.869	**0.879**	0.106
MUTAG (U)	0.834	**0.844**	0.535	0.841	0.819	0.816	0.824
PTC_FR (U)	0.328	0.466	0.328	**0.563**	0.394	0.381	0.364
KKI (U)	0.276	**0.508**	–	0.472	0.385	0.415	0.353
BZR_MD (U)	0.633	0.601	0.335	0.569	0.628	0.630	**0.647**
MSRC_9 (U)	0.074	0.042	–	**0.183**	0.138	0.135	0.130
IMDB-MULTI (U)	0.264	**0.503**	–	0.456	0.487	0.463	0.381

(U) indicates unlabeled datasets. Models tuned for F1-macro score.
Best scores per dataset are bold and underlined.

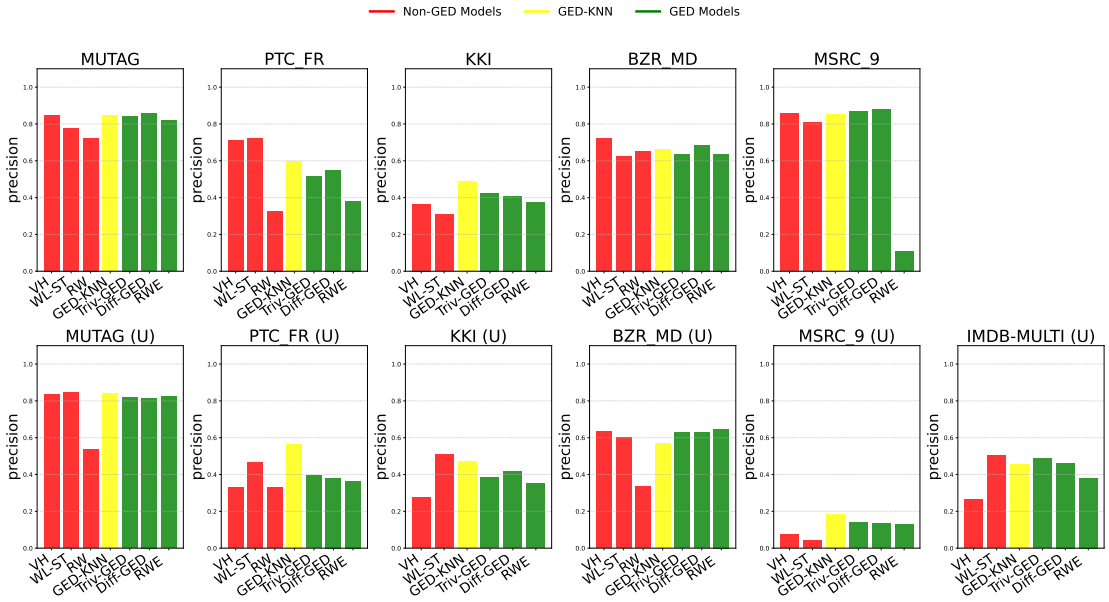


Figure B.2.: Bar Chart for Precision Scores of GED-based Classifiers

Table B.3.: Recall on labeled and unlabeled datasets:

Dataset	VH	WL-ST	RW	GED-KNN	Triv-GED	Diff-GED	RWE
MUTAG	0.824	0.773	0.699	0.854	0.833	**0.855**	0.810
PTC_FR	0.545	0.548	0.500	**0.585**	0.513	0.510	0.499
KKI	0.464	0.459	–	0.502	**0.513**	0.505	0.481
BZR_MD	**0.718**	0.595	0.642	0.659	0.626	0.679	0.627
MSRC_9	0.865	0.840	–	0.826	0.859	**0.871**	0.157
MUTAG (U)	0.811	**0.853**	0.550	0.834	0.813	0.817	0.816
PTC_FR (U)	0.500	0.504	0.500	**0.555**	0.501	0.500	0.501
KKI (U)	0.501	**0.518**	–	0.481	0.485	0.479	0.476
BZR_MD (U)	0.626	0.586	0.474	0.561	0.623	0.617	**0.630**
MSRC_9 (U)	0.152	0.125	–	**0.191**	0.149	0.153	0.158
IMDB-MULTI (U)	0.337	**0.505**	–	0.424	0.485	0.463	0.381

(U) indicates unlabeled datasets. Models tuned for F1-macro score.
Best scores per dataset are bold and underlined.

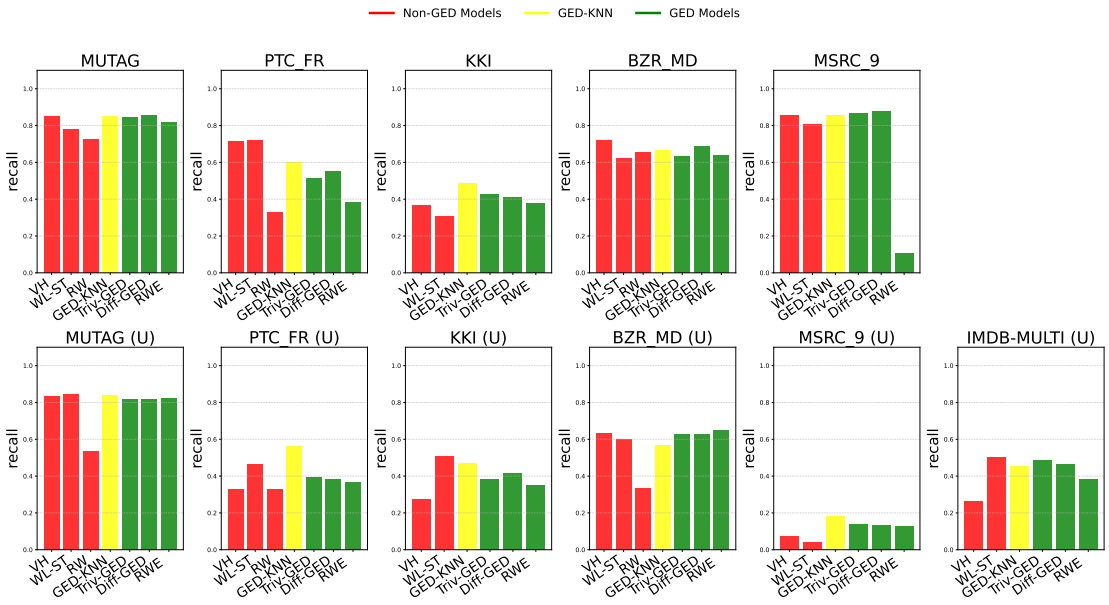


Figure B.3.: Bar Chart for Recall Scores of GED-based Classifiers

B.2. Support vectors and GED computations

For the Random Walk Edit Kernel and Trivial GED Kernel, the average number of support vectors, for each dataset is presented in table B.4, along with the average time taken for GED computations on each dataset. For the other classifiers, these numbers are not relevant, as their runtime calculation did not depend on the number of support vectors for predictions. For the GED KNN and Diffusion GED kernel, GED needs to be computed with all training graphs, so the number of support vectors is not applicable. Their prediction runtime therefore depends on the number of training graphs $(0.8 \cdot \mathcal{N})$ and the average time taken for GED computations. For the baseline kernels, no GED computations are needed at all, so their runtime is purely the measured runtime.

Labeled datasets

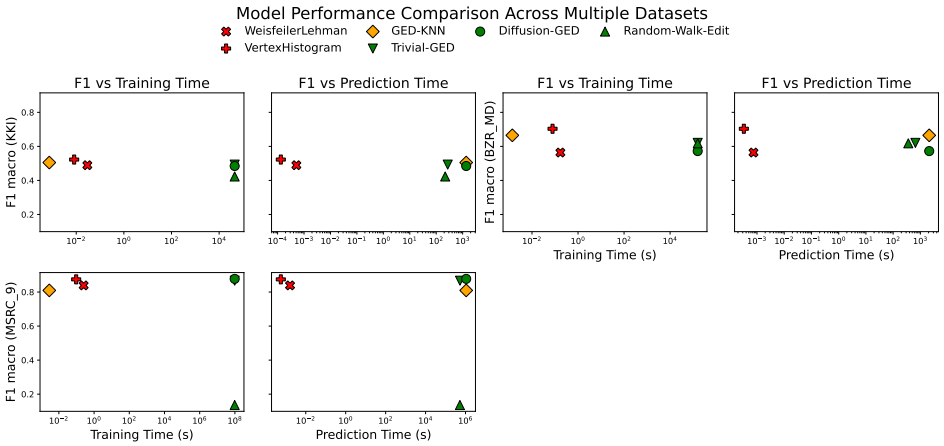
Model	MUTAG	PTC_FR	KKI	BZR_MD	MSRC_9
Triv-GED	7.8	114.0	13.2	46.6	83.1
RWE	10.8	111.3	10.6	25.6	87.2
avg. GED time	0.443s	14.070s	20.627s	14.070s	6000.000s

Unlabeled datasets

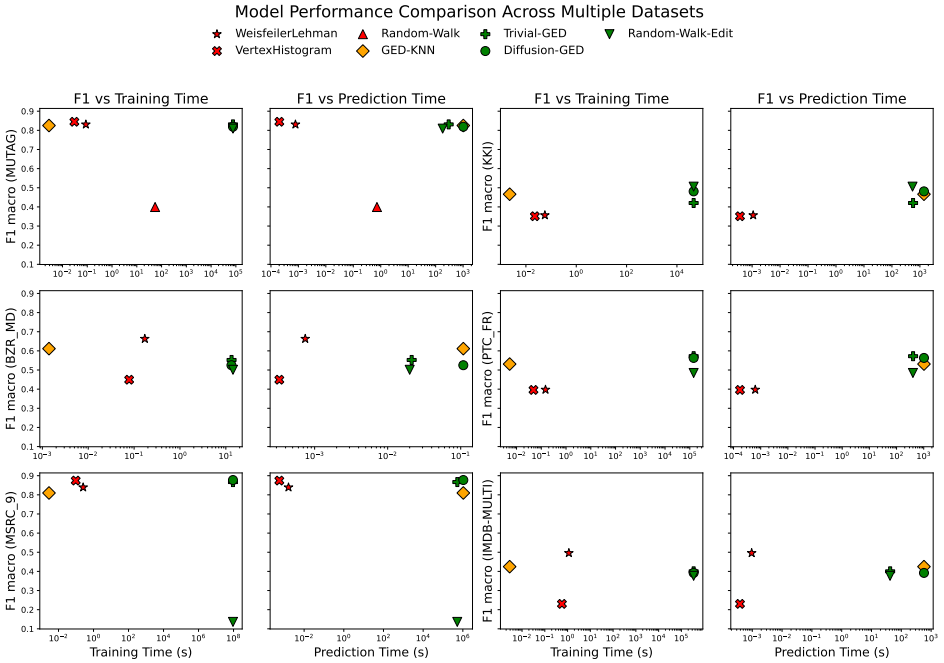
Model	MUTAG	PTC_FR	KKI	BZR_MD	MSRC_9	IMDB-MULTI
Triv-GED	43.9	111.3	26.9	50.1	86.3	83.1
RWE	26.6	109.7	25.7	27.8	87.8	87.2
avg. GED time	6.786s	3.649s	21.178s	0.000s	6000.000s	0.480s

Table B.4.: Average Number of Support Vectors and GED times

B.3. Additional Plots



(a) labeled datasets



(b) unlabeled datasets

Figure B.4.: Additional Plots for Classifiers Score and runtime relationship, on log scale.

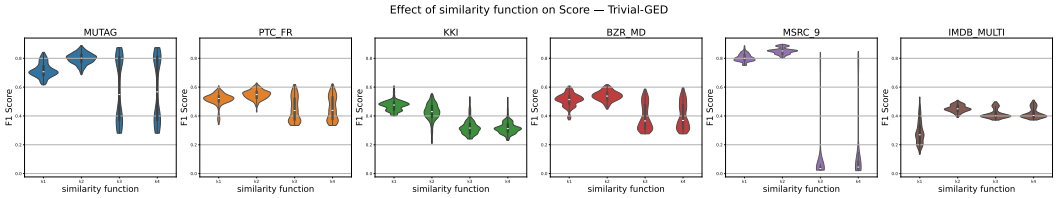


Figure B.5.: Comparions of the effect of different similarity functions on f1 score of the Trivial-GED kernel on labeled datasets.

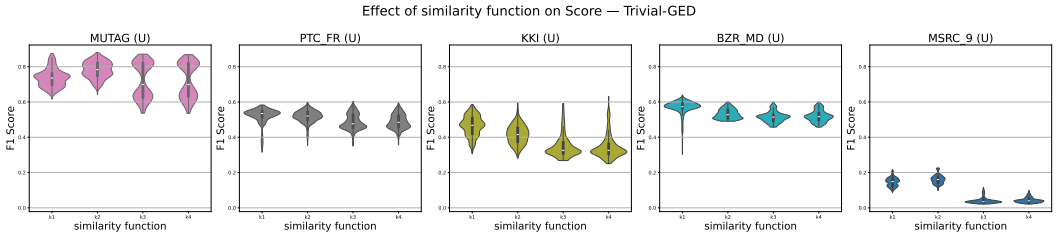


Figure B.6.: Comparions of the effect of different similarity functions on f1 score of the Trivial-GED kernel on unlabeled datasets.

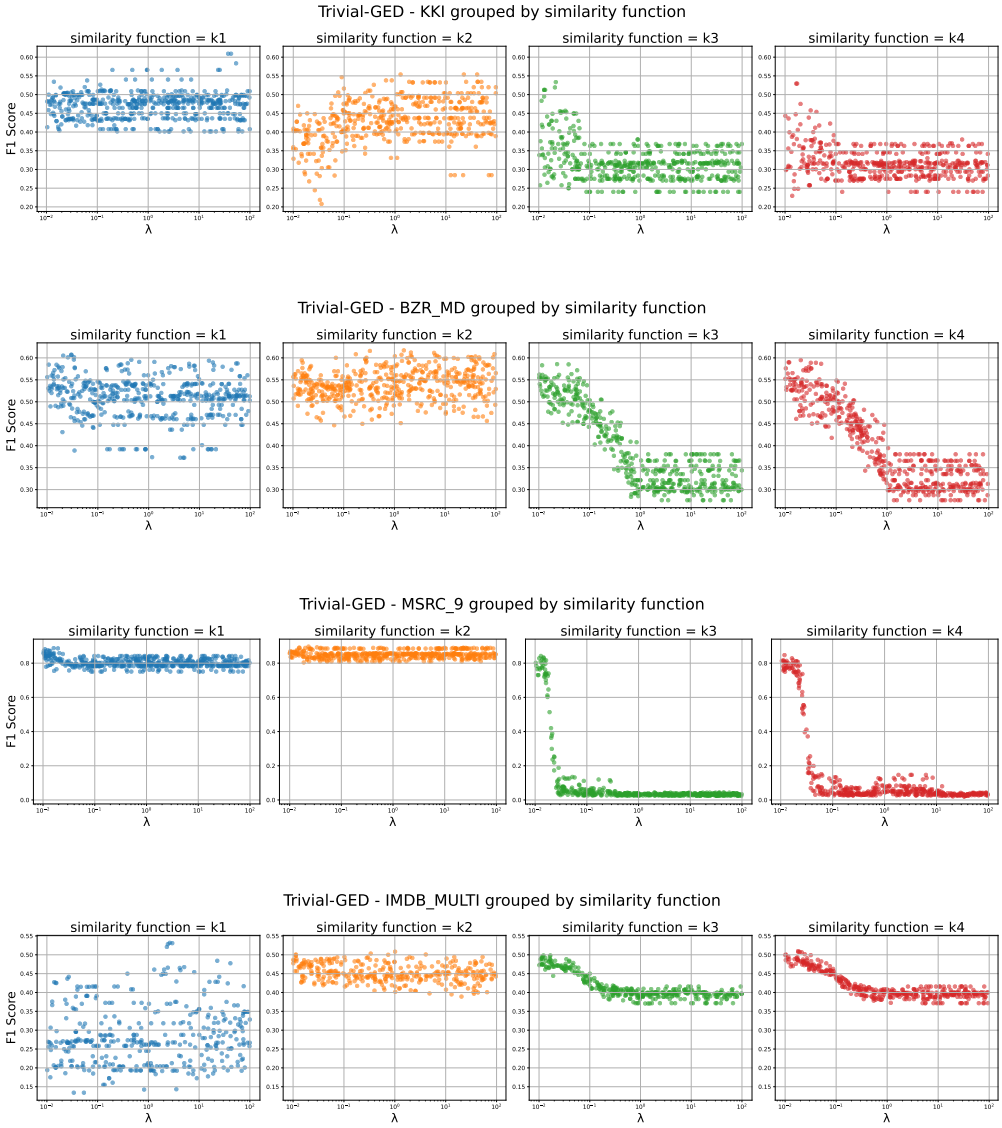


Figure B.7.: Additional plots for the relationship of bandwidth parameter λ and similarity function on different datasets for the trivial GED Kernel.

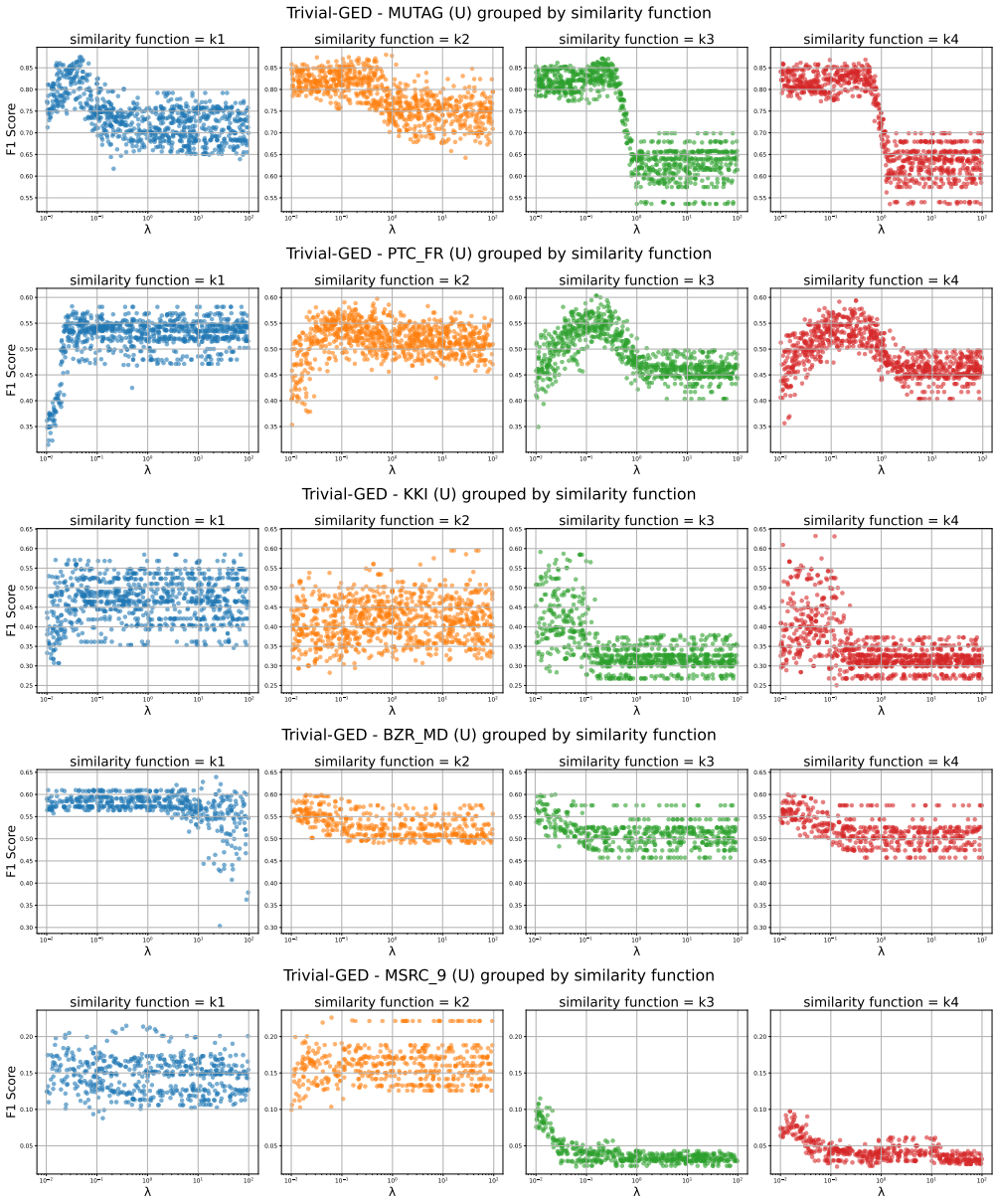


Figure B.8.: Additional plots for the relationship of bandwidth parameter λ and similarity function on different datasets for the trivial GED Kernel.

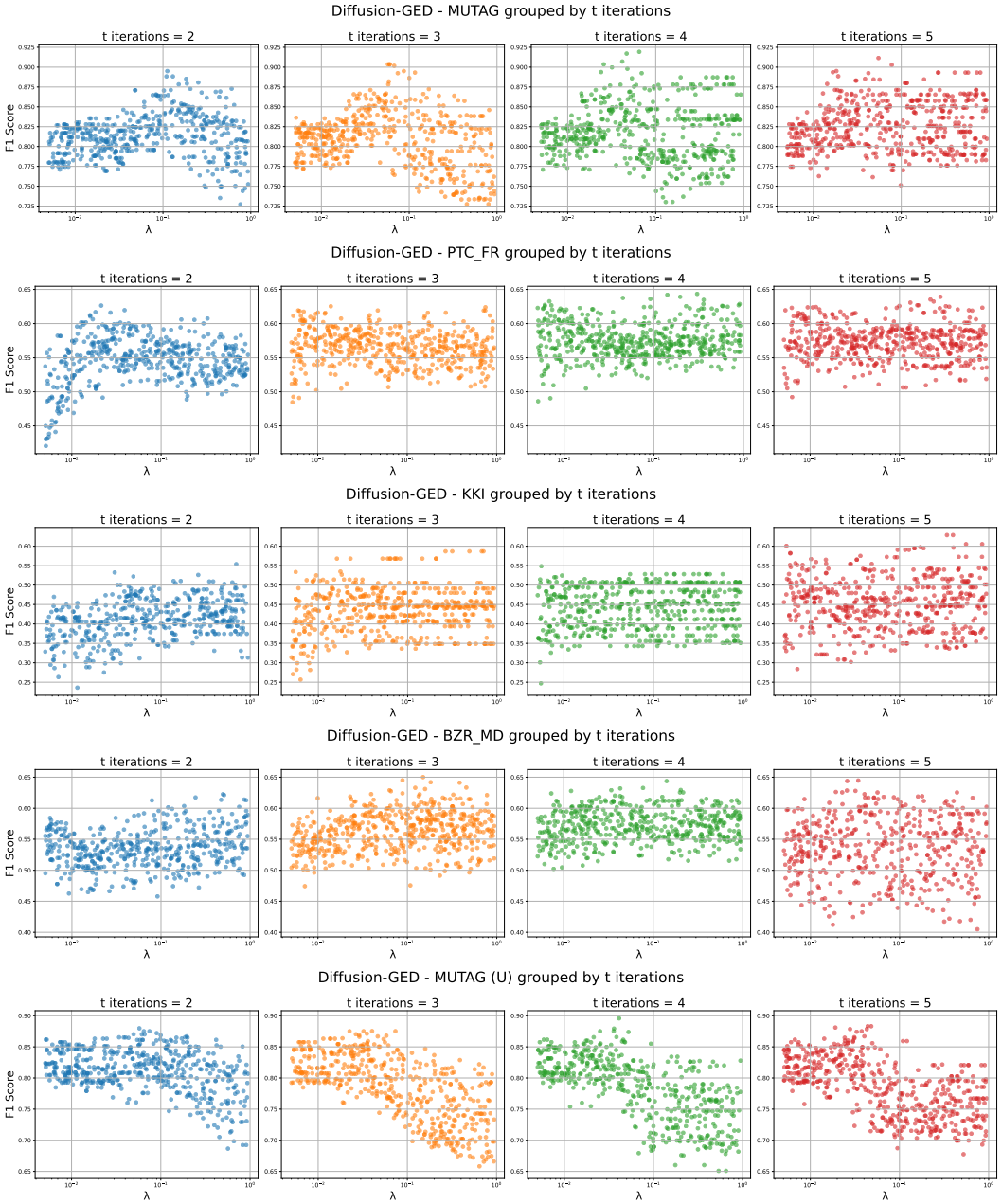


Figure B.9.: Remaining plots for the relationship of the decay parameter λ and number of diffusion iterations on different datasets for the Diffusion GED Kernel.

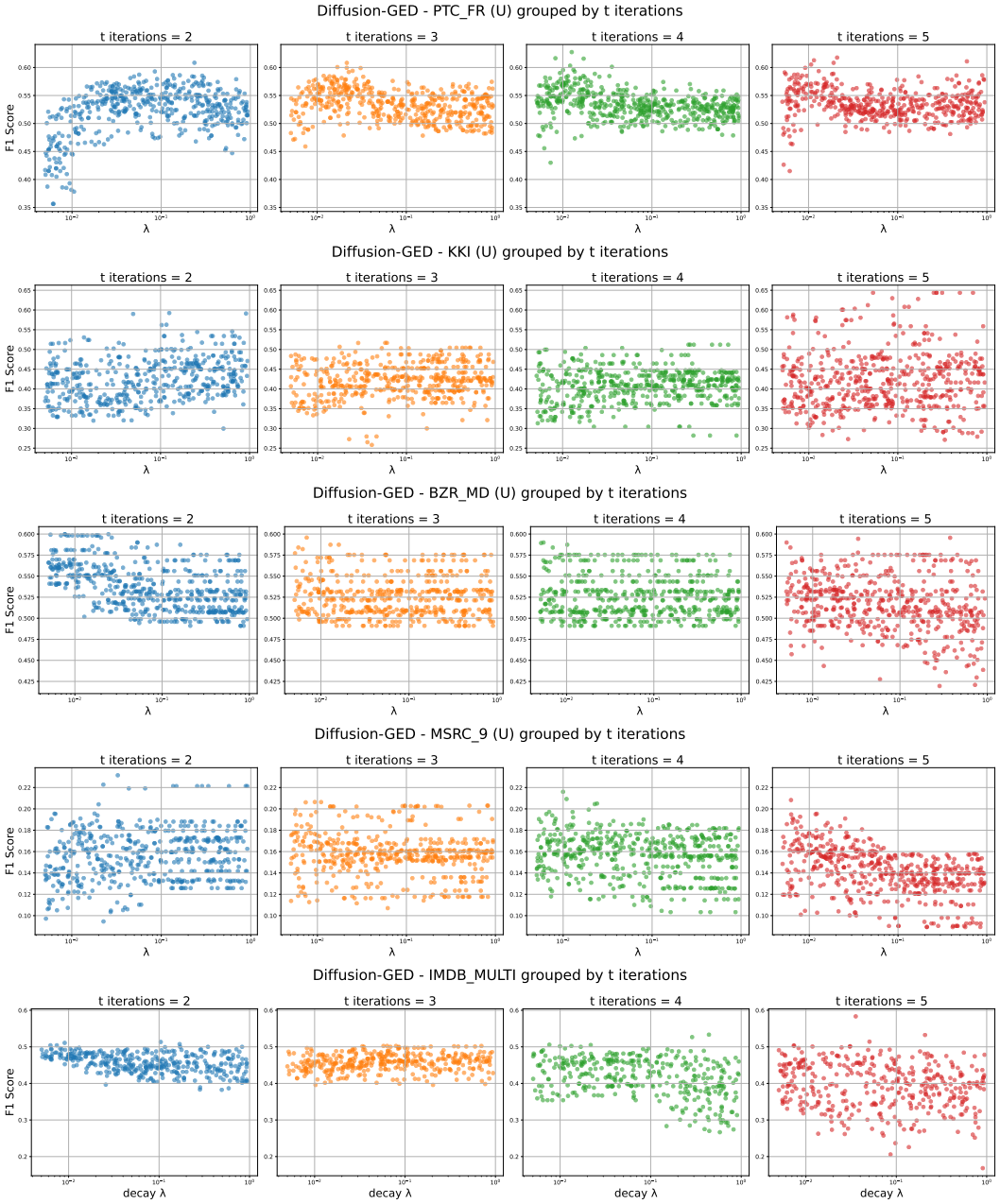
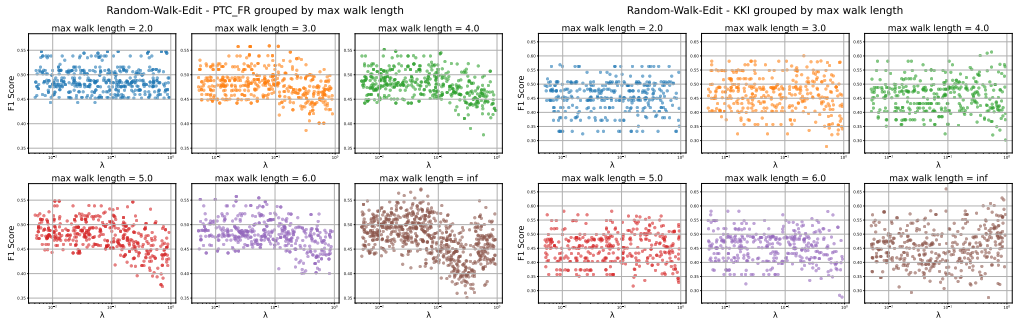
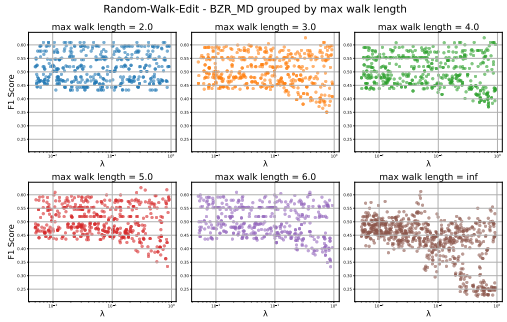


Figure B.10.: Remaining plots for the relationship of the decay parameter λ and number of diffusion iterations on different datasets for the Diffusion GED Kernel.

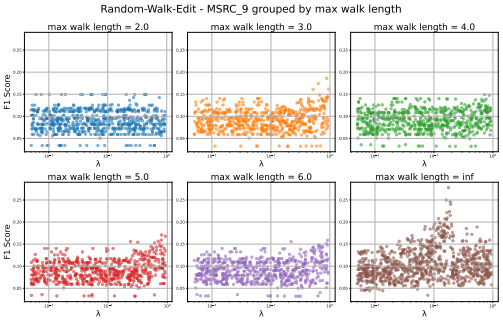


(a) PTC_FR

(b) KKI

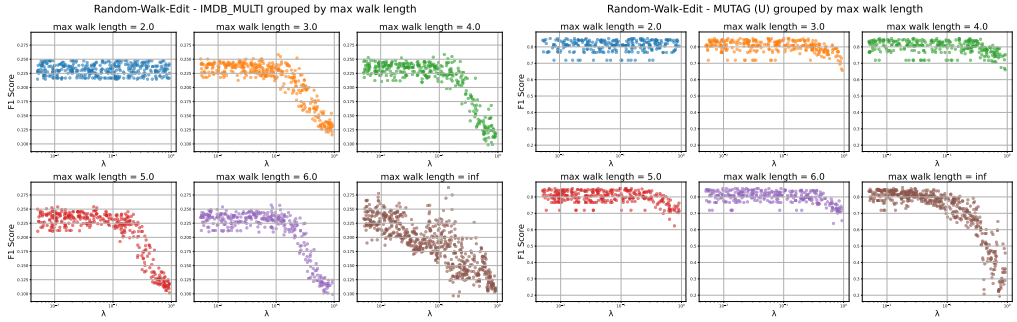


(c) BZR_MD



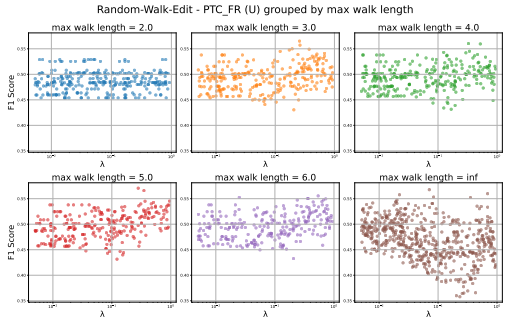
(d) MSRC_9

Figure B.11.: Remaining plots of the relationship of the decay parameter λ and maximum walk length on different datasets for the Random Walk Edit Kernel.

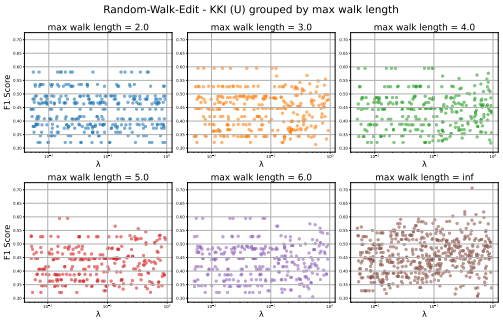


(a) IMDB-MULTI

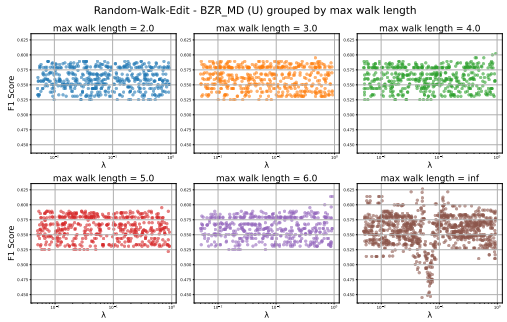
(b) MUTAG (U)



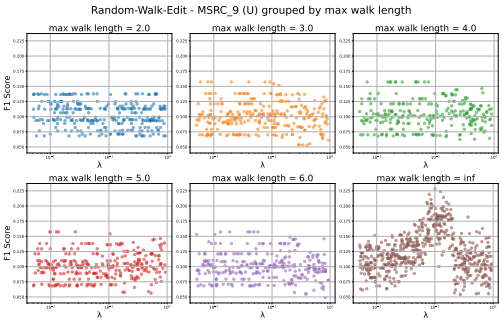
(c) PTC_FR (U)



(d) KKI (U)



(e) BZR_MD (U)



(f) MSRC_9 (U)

Figure B.12.: Remaining plots for unlabeled datasets of the relationship of the decay parameter λ and maximum walk length on different datasets for the Random Walk Edit Kernel.

C. Proof Details

C.1. Proof for optimal GED in Example

The graphics in figure 2.3 show that the edit path is a valid edit path from G to G', and that it is also minimal. We begin by restating the two graphs G_1 and G' in mathematical notation.

Definition 1 (Graph G). *Let $G = (V, E, \alpha, \beta)$ be a graph with*

- $V = \{v_1, v_2, v_3, v_4, v_5\}$.
- $E = \{(v_1, v_2), (v_2, v_3), (v_2, v_4), (v_2, v_4), (v_3, v_5)\}$.
- $\alpha(v_1) = \alpha(v_3) = B$ *(Blue).*
- $\alpha(v_2) = \alpha(v_4) = \alpha(v_5) = R$ *(Red).*
- $\beta(e) = NILL \quad \forall e \in E$ *(no edge labels).*

Definition 2 (Graph G'). *Let $G' = (V', E', \alpha', \beta')$ be a graph with*

- $V' = \{u_1, u_2, u_3, u_4\}$.
- $E' = \{(u_1, u_2), (u_1, u_3), (u_1, u_4), (u_2, u_3), (u_2, u_4)\}$.
- $\alpha'(u_1) = R$ *(Red).*
- $\alpha'(u_2) = \alpha'(u_3) = \alpha'(u_4) = B$ *(Blue).*
- $\beta'(e) = NILL \quad \forall e \in E'$ *(no edge labels).*

To simplify notions, we denote the number of nodes per node label in the two graphs as follows: $B_G = |\{v \in V | \alpha(v) = B\}|$, $R_G = |\{v \in V | \alpha(v) = R\}|$, $B_{G'} = |\{v' \in V' | \alpha(v') = B\}|$ and $R_{G'} = |\{v' \in V' | \alpha(v') = R\}|$.

The path $\pi = (sub_v(v_2, B), del_e((v_3, v_5)), del_v(v_5), ins_e(v_1, v_4))$ illustrated in the example in section 2.2 has a cost of 4 for unified costs of 1. We prove a lower bound of 4 for any edit path transforming G into an isomorphic graph of G'.

Proof for π being an optimal edit path from G to G'. For every two graphs to be isomorphic, the following conditions must trivially hold:

1. $|\{v \in V | \alpha(v) = c\}| = |\{v' \in V' | \alpha(v') = c\}| \quad \forall c \in \alpha$

2. $|E| = |E'|$

From the definitions 1 and 2, the following properties of the graphs G and G' can be observed:

- $|V| = 5$, $|E| = 5$.
- $B_G = 2$, $R_G = 3$.
- $deg(v) \geq 1 \quad \forall v \in V$.
- $|V'| = 4$, $|E'| = 5$.
- $B_{G'} = 3$, $R_{G'} = 1$.
- $deg(u) \geq 1 \quad \forall u \in V'$.

The obvious observation is that the first condition cannot be met, as $|V| \neq |V'|$ and the number of nodes per color differ. To transform G into a graph with the same number of nodes per label as G', node edit operations are required.

1. Necessity of at least one node-deletion $\mathbf{del_v(v)}$.
Since $|V(G)| - |V(G')| = 1$, any sequence that transforms G into a graph with $|V(G')|$ vertices must contain at least one node deletion. Hence the sequence contains at least one node-deletion operation.

2. Consequence for edge counts. Deleting a node v removes every incident edge for the edit operations to result in a valid graph structure $\mathbf{del_e(e)}$.
In G every node has $deg(v) \geq 1$, so the chosen deleted node has $deg(v) \geq 1$ and therefore the sequence must include at least one edge-deletion operation in addition to the node deletion.

3. Matching edge counts requires at least one edge-insertion $\mathbf{ins_e(e)}$.
After one node-deletion and the necessary deletions of incident edges the resulting graph has at most $|E(G)|-1 = 4$ edges. Since $|E(G')| = 5$, at least one edge-insertion operation is required to reach the edge count of G'.

4. Matching label counts requires at least one node-substitution $\mathbf{sub_v(v)}$.
Consider the node label counts: $(B_G, R_G) = (2, 3)$ and $(B_{G'}, R_{G'}) = (3, 1)$. Deleting one node from G yields two possibilities for the pair of color counts of the intermediate graph. 1. If the deleted node is Red: counts become $(2, 2)$. To obtain $(3, 1)$ from $(2, 2)$ requires at least one substitution $R \to B$. 2. If the deleted node is Blue: counts become $(1, 3)$. To obtain $(3, 1)$ from $(1, 3)$ requires at least two substitutions, which is worse. Hence, in any minimal-cost path a node with label R must be deleted and at least one node-substitution must be performed.

Combining these minimally required edit operations, any edit sequence must therefore include at least 4 edit operations. Thus, a lower bound of 4 edit operations to transform G into an isomorphic graph of G' is established.

The edit path π shown in the example contains exactly 4 edit operations, and is therefore an optimal edit path from G to G'. □

Additionally to the edit path in the example, 2 other optimal edit paths with a cost of 4 exist. These are:

- $\pi_2 = (sub_v(v_4, B)\ , del_e(v_3, v_5)\ , del_v(v_5)\ , ins_e(v_1, v_4))$
- $\pi_3 = (sub_v(v_4, B)\ , del_e(v_3, v_3)\ , del_v(v_4)\ , ins_e(v_1, v_3))$

D. Ehrenwörtliche Erklärung

Ich versichere, dass ich die beiliegende Bachelor-, Master-, Seminar-, oder Projektarbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Quellen und in der untenstehenden Tabelle angegebenen Hilfsmittel angefertigt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Declaration of Used AI Tools

Tool	Purpose	Where?	Useful?
Anara	Summarization of papers	Literature Review	-
Anara and Scispace	Finding relevant literature	Literature Review and SVMs	+
Google Scholar Labs	Finding relevant literature	Literature Review	+++
Grammarly	Grammar checking and Rephrasing	Throughout	++
GitHub Copilot	Code autocompletion	Source code	+++
Google Gemini	Latex syntax assistance	Throughout	++
Google Gemini	Code for plots	Results and Appendix	++

Unterschrift
Mannheim, den 09. Dezember 2025