ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

RRE
Reliability and Risk Engineering

Dominik Weiss

Kernel SVM based predictor for identifying cascading failures using power grid topology and pre-outage operating data

Bachelor Thesis

RRE – Reliability and Risk Engineering Lab
Swiss Federal Institute of Technology (ETH) Zurich

Expert: Prof. Dr. Giovanni Sansavini
Supervisors: M.Sc Anna Varbella

Zürich, July 30, 2022

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Eigenständigkeitserklärung

Die unterzeichnete Eigenständigkeitserklärung ist Bestandteil jeder während des Studiums verfassten Semester-, Bachelor- und Master-Arbeit oder anderen Abschlussarbeit (auch der jeweils elektronischen Version).

Die Dozentinnen und Dozenten können auch für andere bei ihnen verfasste schriftliche Arbeiten eine Eigenständigkeitserklärung verlangen.

Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuer und Betreuerinnen der Arbeit.

Titel der Arbeit (in Druckschrift):

Kernel SVM based predictor for identifying cascading failures using power grid topology and pre-outage operating data

Verfasst von (in Druckschrift):
Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich.

Name(n):	**Vorname(n):**
Weiss	Dominik

Ich bestätige mit meiner Unterschrift:

- Ich habe keine im Merkblatt „Zitier-Knigge“ beschriebene Form des Plagiats begangen.
- Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu dokumentiert.
- Ich habe keine Daten manipuliert.
- Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Plagiate überprüft werden kann.

Ort, Datum	**Unterschrift(en)**
Schaffhausen, 29.07.2022	

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich. Durch die Unterschriften bürgen sie gemeinsam für den gesamten Inhalt dieser schriftlichen Arbeit.

Acknowledgements

I would like to thank my supervisor Anna Varbella for giving me the chance to tackle this fascinating topic, assisting me during countless meetings and for her patience and expertise in answering all of my questions. I am also very grateful to Professor Giovanni Sansavini for allowing me to write my bachelor thesis in his group.

Abstract

The goal of this thesis is to predict cascading failures in power grids using a machine learning model. We can represent our power grids as graphs, which will be the input to our model. We want our model to learn to predict whether a given set of failed lines in a grid is going to cause a cascading failure or not. This allows us to make online predictions and enable the transmission system operator to take quick action against a cascading failure.

The main contribution of this thesis is to use specially designed graph kernels, commonly used in biochemistry, to represent power grids as graphs and use these kernels to train a kernel support vector machine. We discuss what kernel support vector machines are, and we introduce multiple different graph kernels from literature. All of the graph kernels are then evaluated and compared for their performance using balanced accuracy and a histogram of projections.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cascading failures are a major cause of blackout events. Predicting such cascades allows the transmission system operator to take immediate action against them. However, physics-based models are computationally heavy, which is why the goal of this thesis is to find a surrogate machine learning model for the Cascades module of Nexus-e [1].

1.1 Power Grids

The goal of a power grid is to supply power from power plants to end users whilst maintaining a constant voltage and frequency across the entire grid. The frequency of the grid needs to be constant in order for all of the generators to be able to be run in parallel. When there is more demand than supplied power in a grid, the frequency drops. This is because the missing energy is extracted from the spinning masses of the generators in large power plants, which causes them, and the frequency of the grid, to slow down and vice versa if there is more supplied power than demand. This means the grid frequency has to be maintained constant by exactly matching demand and supply [2]. Nowadays, power grids are still mostly traditional with centralized generation, which includes fossil fuels, nuclear and hydro power plants as stable sources of power. In the face of climate change, this means that decarbonization needs to take place, introducing more fluctuating power generation such as wind and solar power to the power grid. Additionally, the electrification of heating and transportation systems is bound to increase the world's energy demand and thus the load on the power grid even more. This means that the power grid needs to be expanded to make room for new technologies, which needs to be done in a way to ensure the new grid is as stable as possible [2]. Because of that, we need to be able to assess the stability of a given grid in the face of failed power lines. This is what the cascades module of Nexus-e is used for [1].

1.2 The Cascades Module

The Cascades module [1] is a model of the power grid which has two tasks: 1) it is used to find out whether a cascading failure occurs given a set of failed power lines (contingencies) and 2) to suggest system expansion plans to reach a certain target security. This thesis only looks at the first of these two tasks.
Firstly a model of the power grid is initialized with a set of contingencies. Then the model identifies islands in the system. Islands are part of the grid, which are completely cut off from the rest of the grid due to the given contingencies. The model then simulates grid automation. In power grids, grid automation is used to regulate the frequency and restore the power balance. This is done by deploying generation reserves and shedding loads, which means either putting more power into the grid or disconnecting consumers. Depending on how much the grid frequency deviates, load shedding is performed. All of the power flows and bus voltages are calculated using an AC power flow algorithm. Additionally, if an element of the grid exceeds its power rating, it is disconnected by the simulation [1]. At this point, it is important to note that hereafter we consider every scenario where some demands have not been served as a cascading failure irrespective of its magnitude. This means that we also consider cases where the contingencies lead to islands and thus demand not served as cascades even though there might not be an actual cascade. Still demand not served is the problem which is to be avoided, so including these scenarios is sensible.

1.3 A surrogate Machine Learning Model

This physics-based cascading failure model [1] has the advantage that it can make very accurate predictions. However, all the calculations which need to be done can be very computationally expensive. In this thesis, the goal is to use machine learning to infer if a cascading failure will occur only from the pre-outage operating conditions and topology of the grid. The advantage of machine learning is that once a model is trained, the evaluation of a new example is immediate. This means that the operator of a power system can quickly predict cascading failures and take action against them. In the past, there has been an attempt at this using support vector machines and information theory. The authors use entropy and mutual information techniques to identify critical lines in the power grid from the current state of the system and predict cascades [3]. Another possible approach is using deep learning and graph neural networks. The goal of this thesis is to train Support Vector Machines using the grid topology and pre-outage operating data.

Chapter 2

Machine Learning

2.1 Types of Machine Learning

Machine learning is the science that allows designing algorithms which are able to automatically extract valuable information from data. There are three different machine learning approaches. Supervised, unsupervised and reinforcement learning.

In **supervised learning** the algorithm builds a model of a data set which contains both inputs and desired outputs. By iteratively optimizing the loss function, the algorithm learns a model that can correctly predict the output for a new, never seen input. One type of supervised learning is classification, where the input is some n-dimensional feature vector, and the output is a discrete label which assigns a class to the input. One common example of this is image classification of cats and dogs, where the input is a 2-dimensional array (the image), and the output is a discrete label which states if the input is either an image of a cat or a dog. The goal of this thesis is to predict if a cascade has happened for a given combination of loads and contingencies, modelled as a classification task.

In **unsupervised learning** the algorithm is just given an unlabeled data set. This means, unlike with supervised learning, the algorithm has no idea what kind of output we expect from it. Its task is much rather to find structure, like groups or clusters in data [4].

Reinforcement learning deals with how intelligent agents can learn to maximize a reward. Unlike the previous two approaches, in reinforcement learning, the agent is constantly adapting to a changing environment. So there is no data but instead interaction with an environment [5].

2.2 Machine Learning Algorithms

For each machine learning approach, there are many different algorithms that can be chosen. For the task at hand, which is supervised classification, here are some of the

most popular algorithms.

K-nearest neighbour is a non-parametric learning method which classifies a new data point by looking at the labels of its k closest neighbours. Closeness can be determined by various metrics such as euclidean distance. Learning consists of simply memorizing the training data [4].

Support vector machines classify data by separating the decision space in two using a decision hyperplane. Data points are then simply classified by whichever side of the hyperplane they lie on. If we want our support vector machine to capture more complex features, we can use so-called kernel support vector machines, which will be discussed in greater depth later on [6].

Artificial neural networks are a class of learning algorithms that are structured similarly to neural networks in organisms, having neurons that are connected by weighted links into large networks. They can be used to learn functions of arbitrary complexity, which makes them perfect for all kinds of tasks, including classification [7].

Deep Neural Networks are a subclass of neural networks that have millions of neurons and hyperparameters that allows them to capture even very fine details in the input data and make very accurate predictions. One problem with deep neural networks is that their inner workings are very hard to analyze due to their large size. They can also be very hard to train as one needs to choose the objective function very carefully, so the algorithm does exactly what it is expected to [4].

2.3 Support Vector Machines

The class of machine learning methods used in this thesis are called Support Vector Machines (SVM). In this chapter, the main ideas of SVMs and the kernel trick are introduced. This chapter is in large parts based on *Support Vector Machines* by Andreas Christmann, and Ingo Steinwart [6] and the *Introduction to Machine Learning* lecture series by Andreas Krause and Fanny Yang [4].
If our task is to classify data into two classes intuitively, we can look at the data and try to come up with a line or a hyperplane which separates the two classes as well as possible, as shown in figure [2.1].
If there are two clusters in our data which are fully separable, i.e. the clusters do not overlap, there are infinitely many such dividing hyperplanes which perfectly separate the data, as in figure [2.2].
If we now want to make predictions on unseen data, not all of these dividing lines are equally good. In fact, some of them are much closer to one of the classes of our training data, and thus the chance that a new data point is very close to the decision boundary is quite high. If the measurement of this new point is only slightly off, it gets misclassified. To prevent this, we can choose to use the decision boundary, which is the furthest from both data clusters. To do this, we can specify so-called support vectors, which are the

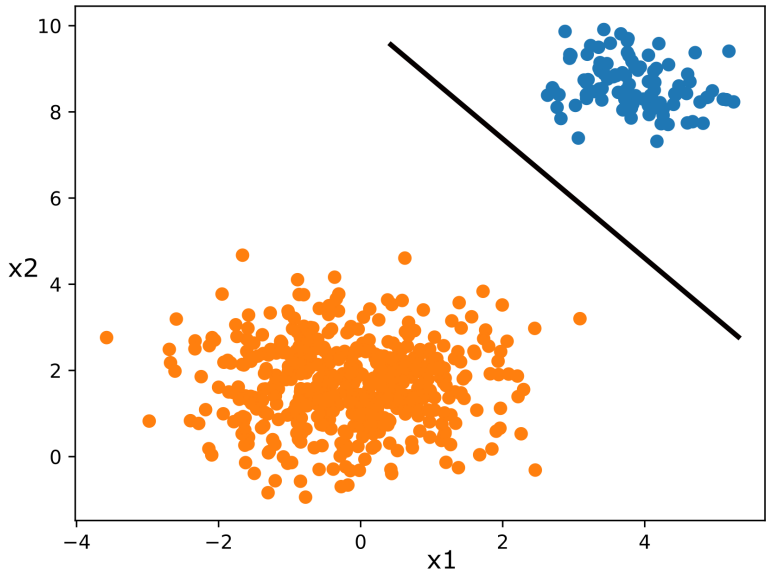


Figure 2.1: Data with two clusters and an arbitrary decision boundary.

closest points of each class to the decision boundary. By choosing these support vectors optimally, we can maximize the minimum distance of the decision boundary to the data, as seen in figure [2.3] [4].

This is where support vector machines get their name from. Now we can formalize this procedure further and introduce all the necessary tools along the way.

2.3.1 Data

First, we want to define further what a data set used in SVMs looks like. We can define a data set $D := ((\mathbf{x}_1, y_1), ...(\mathbf{x}_n, y_n))$ as a pair of features $\mathbf{x}$ and labels y. The features can be any sort of object, including graphs, but for now, we'll look at them as euclidean vectors, so $\mathbf{x} \in \mathbb{R}^n$. The labels are binary labels $y \in \{-1, 1\}$. Using SVMs we want to find a function $f : X \to Y$ such that $f(\mathbf{x})$ gives the correct label y_i to an arbitrary $\mathbf{x}_i$.

2.3.2 SVM from Geometry

As discussed at the beginning of this section, we want to find a decision plane between the two classes which is as far away as possible from the nearest point $\mathbf{x}_i$ from either class. In general, such a hyperplane can be written as an affine function, where every point $\mathbf{x}$ satisfies

$$\mathbf{w}^T\mathbf{x} - b = 0, \tag{2.1}$$

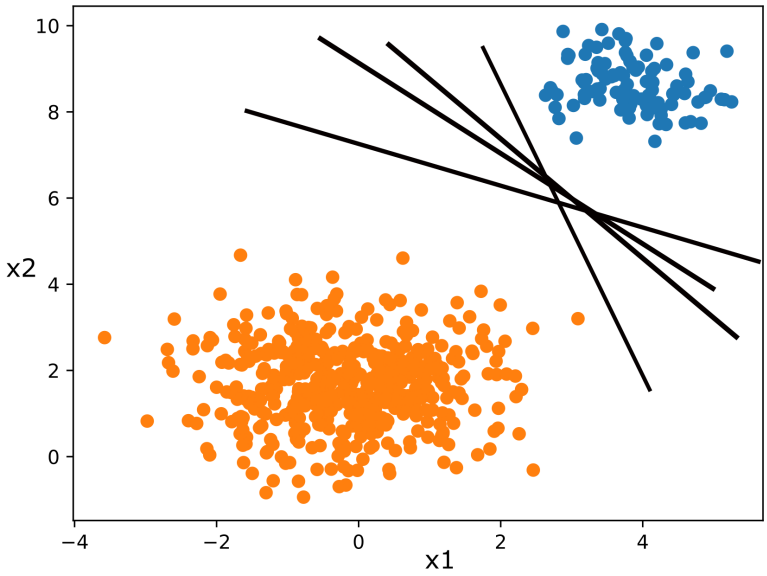


Figure 2.2: There are infinitely many possible decision hyperplanes.

where $\mathbf{w}$ is a normal vector to the hyperplane and $\frac{b}{\|\mathbf{w}\|}$ is the distance of the hyperplane to the origin in direction of $\mathbf{w}$. $\frac{margin(\mathbf{w})}{\|\mathbf{w}\|} = \frac{1}{\|\mathbf{w}\|}$, is the distance of the hyperplane to the margin, as no point in the data set is closer to the hyperplane than $\| \mathbf{w} \|$. The points which lie on the margin are called support vectors $\mathbf{x}_{support}$, hence the name support vector machines. For all of this we assume $\mathbf{w}$ is scaled accordingly, so the $margin(\mathbf{w}) = \mathbf{w}^T\mathbf{x}_{support} = 1$. Suppose now that we have found such an optimal hyperplane using an SVM, to classify a new data point $\mathbf{x}_{new}$ which we've never seen before, we can make use of the direction of $\mathbf{w}$ to find

$$y_{new} = \text{sign}(\mathbf{w}^T\mathbf{x} - b). \tag{2.2}$$

In fact if $y \in \{-1, 1\}$, and using the fact that all data points have to lie on or outside of the margins, we find:

$$\mathbf{w}^T\mathbf{x}_i - b \geq 1, \text{ if } y_i = 1 \tag{2.3}$$

and

$$\mathbf{w}^T\mathbf{x}_i - b \leq -1, \text{ if } y_i = -1 \tag{2.4}$$

We can rewrite this in one line as

$$y_i(\mathbf{w}^T\mathbf{x}_i - b) \geq 1 \text{ for all } 1 \leq i \leq n \tag{2.5}$$

This means that in the end our optimization problem comes down to minimizing $\| \mathbf{w} \|$ subject to $y_i(\mathbf{w}^T\mathbf{x}_i - b) \geq 1$ for all $1 \leq i \leq n$.

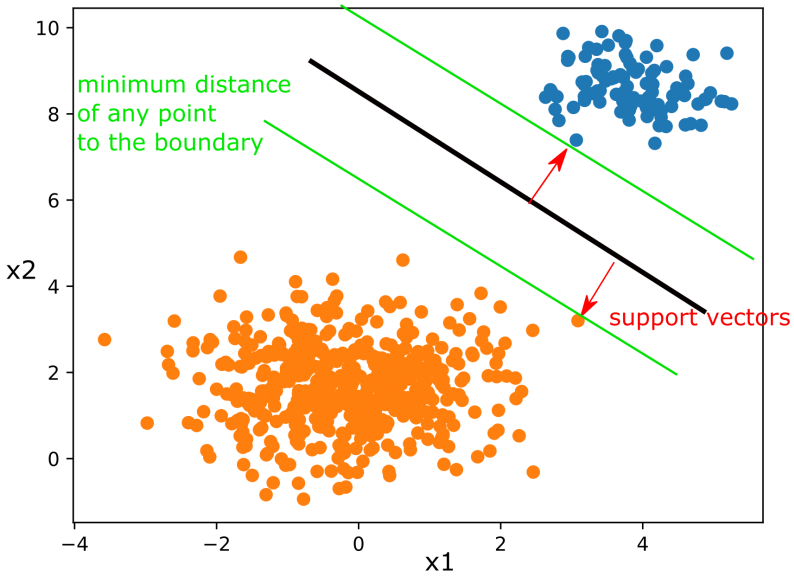


Figure 2.3: Using support vectors to maximize the minimum distance.

One problem with this definition is that no point can lie inside the margins. This is fine for a data set where both classes are separable, but it leads to problems when data clusters overlap. Additionally, as the decision hyperplane only depends on the support vectors, our classifier is not robust against outliers and noisy data. So we want to make our margins a bit "softer" to allow some data points to lie within them and maybe even on the "wrong" side of the decision plane. For this, we first need to take a look at how we can formally define a good classifier.

2.3.3 Learning risk

If we look at SVMs from the statistical learning perspective, we can define the data as being generated from a probability distribution P on $X \times Y$ for which we want to learn a function $f : X \to Y$ which gives a good approximation for the original distribution. This necessitates that the test and training data are both generated from the same distribution.
Suppose now that we already have trained a classifier, the question is how to assess its performance. We can do this by defining a so-called loss function which is higher the worse our classifier is. Or in other words, the higher the loss function, the worse our model captures the true probability distribution of X. We define the classification loss as:

$$L_{class}(x, y, f(x)) = \begin{cases} 1 & f(x) \neq y \\ 0 & f(x) = y \end{cases}$$

Where x is the input, y the output and $f(x)$ our trained classifier.

2.3.4 Universal Consistency

Using the loss function, we can assess how good the predictions our method makes are on data we already have, but of course, we want to ensure our method to also be able to make accurate predictions for data it has never seen before. For this purpose, we define the *expected risk* of f :

$$\mathcal{R}_{L,P}(f) := \int_{X\times Y} L_{class}(x, y, f(x))dP(x, y) = \int_X \int_Y L(x, y, f(x))dP(y|x)dP_X$$

where P_X is the marginal distribution of P(x,y). This definition is itself not useful since the whole point of machine learning is that we have no knowledge of the probability distribution P. For this reason, we introduce a surrogate of the expected risk, which is the *average empirical risk*

$$\frac{1}{n}\sum_{i=1}^{n} L(x_i, y_i, f(x_i)).$$

Assuming the training and test data are generated from the same probability distribution P, we can apply the law of large numbers to show that for $n \to \infty$, the average empirical loss converges towards $\mathcal{R}_{L,P}(f)$. Therefore, the learning goal is to find the function f which leads to the smallest possible risk

$$\mathcal{R}^*_{L,P} := \inf_{f:X\to Y} \mathcal{R}_{L,P}(f),$$

where the infimum is taken over all of the possible functions. This means that if both our training and test set are generated from the same probability distribution, it is possible to train an accurate enough model using the training set and then make good predictions on the test set.
The definition of the risk allows us to more closely define what a *learning method* is. The goal is to minimize the risk, but the probability distribution P is unknown. Having only labelled data $D := ((x_1, y_1), ..., (x_n, y_n))$, the learning method finds a function f_D to map each x_i to its label y_i . Generally, learning methods are required to be *universally consistent*. This means that for every possible distribution P on $X \times Y$, the functions f_D satisfy

$$\mathcal{R}_{L,P}(f_D) \to R^*_{L,P}, \text{for} \qquad n \to \infty.$$

In words, this means that we want a learning method, which for any arbitrary distribution P and training set D, finds a function with a near minimal risk without knowing anything about P. Support Vector Machines are, in general, universally consistent.

2.3.5 Overfitting and Computational Feasibility

Learning poses one major problem. The goal of learning is to find an optimal function f^* which minimizes the risk. However, the risk is generally unknown. The solution is to replace the risk with the empirical risk of our training set D

$$\mathcal{R}_{L,D}(f) := \frac{1}{n}\sum_{i=1}^{n} L(y_i, f(x_i)).$$

The law of large numbers shows that $\mathcal{R}_{L,D}(f)$ is an approximation of $\mathcal{R}_{L,P}(f)$ for a *single* f. However, this does not necessarily lead to an appropriate *minimizer* of $\mathcal{R}_{L,P}(f)$. An example where this is the case is a function, which classifies all $x_i \in D$ correctly but equals 0 otherwise. This classification function is a good approximation for the training dataset but performs poorly for the general distribution. This phenomenon is called *overfitting*, and it occurs when the learning method comes up with a function that models the training set too closely but performs poorly on unseen data. One way to prevent overfitting is to restrict the function f to a small set of functions $\mathcal{F}$, which is assumed to contain a good solution to the optimization problem. By restricting the search to a small set of functions, only approximate solutions to the original optimization problem risk to be found. Indeed, finding an optimal function space without knowing the P distribution can be complex. Another issue is the computational infeasibility of the training process. Especially for the 0/1 classification loss function, which is non-convex, solving the optimization problem is often NP-hard [8].
The first issue can be solved by scaling the size of the function set $\mathcal{F}$ up with the sample size n. The second issue can be addressed by replacing the risk $\mathcal{R}_{L,D}(f)$ with a surrogate function which is computationally more efficient and still minimizes the risk. We replace the 0/1 classification loss by a *convex surrogate*. For SVMs the most common loss function is the *hinge loss* which is defined as follows:

$$L_{hinge}(y,t) := \max\{0, 1-yt\}, \qquad y \in \{-1,+1\}, t \in \mathbb{R},$$

which is convex and so will be the corresponding empirical risk

$$\mathcal{R}_{L_{hinge},D}(f) = \frac{1}{n}\sum_{i=1}^{n} \max\left(0, 1 - y_i(\mathbf{w}^T\mathbf{x}_i -)\right) \tag{2.6}$$

$\mathcal{R}_{L_{hinge},D}(f)$, ending up with a convex optimization problem

$$\inf_{f\in\mathcal{F}} \mathcal{R}_{L_{hinge},D}(f),$$

which is our learning method.

2.3.6 Soft Margin SVM

We already introduced hard margin SVMs, where the margins are placed in a way so that within the margins, there can be no data points. Now we want to modify these

margins to be soft so that our SVM is not so dependent on the support vectors (the data points closest to the decision plane). We remember that so far, our optimization problem is

$$\min||\mathbf{w}||^2 \text{ s.t. } y_i(\mathbf{w}^T\mathbf{x}_i - b) \geq 1 \text{ for all } 1 \leq i \leq n \tag{2.7}$$

Now instead of requiring $y_i(\mathbf{w}^T\mathbf{x}_i - b) \geq 1$ for all $1 \leq i \leq n$ we can soften this statement by using the hinge loss.

$$\max(0, 1 - y_i(\mathbf{w}^T\mathbf{x}_i -)) \tag{2.8}$$

This way, if a data point $\mathbf{x}_i$ lies within the margin or even on the wrong side of the decision plane, its loss is set to 0. This softens the margins, but we still try to maximize the distance from the margin as much as possible. Using this, we can rewrite the optimization problem as

$$\min\left(||\mathbf{w}||^2 + C\left[\frac{1}{n}\sum_{i=1}^{n}\max\left(0, 1 - y_i(\mathbf{w}^T\mathbf{x}_i)\right)\right]\right) \text{ for all } 1 \leq i \leq n \tag{2.9}$$

Where C is called the *regularization parameter*. It determines how soft our margins are. We see that the hinge loss is not only needed so that we can efficiently optimize our model, in this case, SVM, but it also softens the margins. This means that both are the same, just from two different points of view.

2.3.7 Optimization Algorithm

To train an SVM we use the gradient descent method. In short, gradient descent tries to move the parameters of our model, in this case, the normal vector, to an optimum in very small steps. For every step, the loss function is calculated and then, using the gradient vector of this loss function, the model parameters are changed by a small bit to minimize the loss until we reach minimal loss. For the case of SVMs, which is a convex problem, there is only one minimum which means that we can ensure that our optimization algorithm always converges to the globally optimal solution.

2.3.8 The Kernel Trick

Our SVM can now be used to put straight lines between clusters of data. This might be useful in many cases, but in other cases, two classes can not be separated by a straight line. For such cases, we introduce the kernel trick. To do this, we first need to look at what the optimal solution of an SVM looks like. As it turns out, the optimal normal vector $\mathbf{w}^*$ is just a linear combination of all of the support vectors.

$$\mathbf{w}^* = \sum_{i=1}^{n}\alpha_i y_i \mathbf{x}_i \tag{2.10}$$

In this, we sum over all the training samples and not just the support vectors, but in the end, the coefficients for all the non-support vectors will just turn out to be zero. Now

we can use this property in our optimization problem:

$$\min \left(\sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j \mathbf{x}_i^T \mathbf{x}_j + C \left[\frac{1}{n} \sum_{i=1}^{n} \max \left(0, 1 - \sum_{j=1}^{n} \alpha_i \alpha_j \mathbf{x}_i^T \mathbf{x}_j) \right) \right] \right) \tag{2.11}$$

What see that our optimization problem now only depends on all of the inputs and a series of weights α_i, which at the centre has a dot product of these samples. A dot product itself can also be thought of as a measure of similarity of two vectors. If the dot product of two vectors is 0, then they are as dissimilar as they can possibly be (they are orthogonal). If the dot product is maximal (for two vectors of constant length but different orientation), this means they are collinear, i.e. as similar as they can be. If we are given a new input sample which we have never seen before, and we want to assign it a class, we can look at all the data we already know the class of and compare this new sample to all our training samples. The class it is more similar to is the one we assign it to.

Using a normal dot product might not be the best measure of similarity. Instead, we want to come up with a similarity measure which works better for our application and use it instead of the dot product. We call this similarity measure the kernel function of the SVM.

Let us start by considering a set of functions called *reproducing kernel Hilbert spaces* $\mathcal{H}$ (RKHS). We consider a Hilbert space $\mathcal{H}$ whose elements are functions that map from a feature space $\mathcal{E}$ to real numbers $f : \mathcal{E} \to \mathbb{R}$. A function $k : \mathcal{E} \times \mathcal{E} \to \mathbb{R}$ is defined as a reproducing kernel of the Hilbert space if it satisfies two conditions:

1. For each $x \in \mathcal{E}$, we have
$$k(x, \cdot) \in \mathcal{H}$$
.
2. Reproducing property: for each $f \in \mathcal{H}$ and $x \in \mathcal{E}$
$$f(x) = \langle f, k(x, \cdot) \rangle_{\mathcal{H}}$$
where $\langle \cdot, \cdot \rangle_{\mathcal{H}}$ is an inner product on $\mathcal{H}$.

A Hilbert space with such a reproducing kernel is called a reproducing kernel Hilbert space. [9] The kernel function can, like a dot product, be perceived as a measure of similarity. If the value of $k(x, y)$ is maximal then $x = y$ and if $k(x, y) = 0$ x and y are maximally dissimilar. Because our kernel functions are inner products of an RKHS, they have to be positive semidefinite (PSD). Now using this, our normal vector becomes a function

$$f^* = \sum_{i=1}^{n} \alpha_i k(\mathbf{x}_i, \cdot) \tag{2.12}$$

and our optimization problem can be written as

$$\min \left(\sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j k(\mathbf{x}_i, \mathbf{x}_j) + C \left[\frac{1}{n} \sum_{i=1}^{n} \max \left(0, 1 - \sum_{j=1}^{n} \alpha_i \alpha_j (\mathbf{x}_i, \mathbf{x}_j)) \right) \right] \right) \tag{2.13}$$

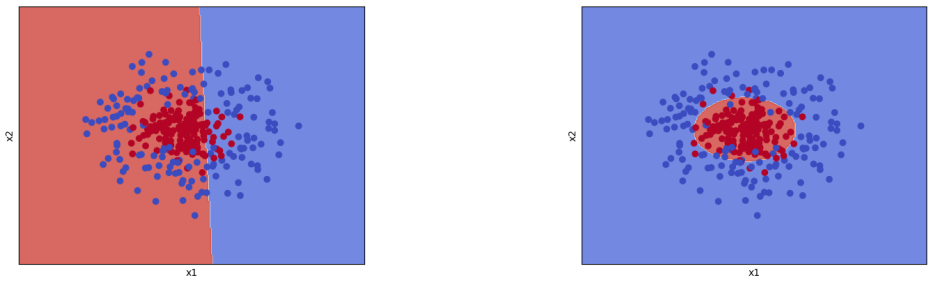
In this case, the first term can also be written as the function norm $\|f\|^2_{\mathcal{H}}$. This formulation allows us to understand the original learning goals of SVMs a bit better. Like in the original SVM this term prevents the function f or the normal vector $\mathbf{w}$ from being too large. This term acts as a regularizer which prevents the function from being too complex. The problem with a function which is too complex is that we can overfit to the training set. This means that our model learns to fit the training data incredibly well but does not generalize to new, unseen data. This means that the problem of making sure our decision plane is as far away from both classes is the same as what is called regularization in other areas of machine learning.

2.3.9 Different Kernel Functions

Now that we have the ability to use any kind of kernel function, the question at hand is which of these kernels performs best. For vector-like features, one of the most commonly used kernels is the Gaussian radial basis function (RBF) kernel due to its versatility.

$$k_{RBF}(x_1, x_2) = e^{-\frac{\|x_1 - x_2\|^2}{\tau}} \tag{2.14}$$

where τ is called the bandwidth parameter. We can see this if we take a look at two soft margin SVMs with and without such a Gaussian RBF kernel. In the following plots [2.4], the decision hyperplane is the line between the two coloured regions, which represent the two classes. The data, in this case, has the shape of two concentric circles. In the



(a) Linear SVM **(b)** Gaussian RBF kernel SVM

Figure 2.4: The effect of using a kernel function in an SVM.

linear case, the SVM does not manage to come up with a sensible decision plane. The Gaussian RBF kernel captures the shape of the data much better. This is also how we have to imagine the working mechanisms of much higher-dimensional kernels, which can not be so easily plotted. For our use case, however, we have to look a bit further. Since our samples consist of graphs, which can not be simply represented as a one-dimensional vector, we need to develop a kernel which can compare two graphs for how similar they are. For this, we introduce several such graph kernels in the next chapter.

Chapter 3

Feature Engineering and Graph Kernels

Before we train a Machine Learning model on our data, we first want to analyze the data and develop a kernel function to use in our SVM. We saw in chapter 2 that Support Vector Machines are based around so-called *kernel functions* $k : \mathcal{E} \times \mathcal{E} \to \mathbb{R}$ which essentially can be thought of as an extended dot product. Two vectors or data points x, y are more similar the larger $k(x, y)$. Choosing an appropriate kernel function allows us to use SVMs to make predictions on different types of data. In this thesis, we deal with power grids. As discussed in the introduction, power grids can mathematically be described by graphs.
One could certainly just flatten all of the features of the power system graph into a single vector. Even if we don't lose information by flattening the data, it really makes no sense to remove all the structure from it. Indeed, in our particular case, the structure of the power grid preceding a cascading failure might play a crucial role in predicting cascades. Determining the power grid stability subject to different contingencies is one of the goals of the method developed in this thesis. To this end, having a kernel function which takes whole graphs as an input is the preferred option. We will call these functions *graph kernels*.
Graph kernels have, at least to the authors' knowledge, never been used on power grids. They, however, have been used in various graph classification tasks, such as classifying molecules and proteins [10] which are also commonly modelled as graphs. Many types of graph kernels have been developed [11], of which we are going to be looking at the Weisfeiler-Lehman and Wasserstein Weisfeiler-Lehman kernels, and we will also be coming up with a kernel of our own using eigenvector centrality.

3.1 Graphs

Generally, graphs are objects that consist of nodes and connections between these nodes called edges. They are a useful mathematical construct for describing various real-world problems. One example is molecules, where the nodes represent the individual atoms,

and the edges represent the different types of bonds between the atoms. If we want to represent a power system with a graph, the nodes represent the buses, and the edges are the power lines.
Mathematically we define a graph as a tuple $G(V, E)$ where V and E denote the sets of nodes (vertices) and edges. The so-called *adjacency matrix* of a graph lists all the connections between the nodes

$$A_{ij} = \begin{cases} 1 & if\, nodes\, v_i\, and\, v_j\, are\, connected \\ 0 & otherwise \end{cases} \tag{3.1}$$

3.2 Data

The data that is being used in this thesis is generated from two different simulations of cascading power failures. The first one just has the state of every node (bus) of the grid before an outage, and some integer values that denote the contingencies, i.e. which lines have failed. The output is a label 1 for a cascade and 0 if there is no cascade. The second data set is graph-based. This means that in addition to all the node features, there is an adjacency matrix which encodes the structure of the graph. Depending on which lines have failed, this adjacency matrix looks different.
The first data set can easily be represented as a one-dimensional feature vector, which is the format our features need to have to be used in an SVM classifier. The second data set can not easily be flattened into a vector, but it is easy to see that it contains more information than the first one since the topology of the grid is encoded in the adjacency matrix. The contingencies are encoded by changing the adjacency matrix to account for the missing power lines.
To see which features are relevant, we are going to be using the first data set. To calculate the actual kernel functions, which are graph-based, we are going to use the second, graph-based data set

3.3 Correlation Matrix for Feature Selection

The data from the simulation contains active power, reactive power, apparent power, voltage magnitude and voltage angle as features. These five quantities are not necessarily linearly independent. To quantify this, we can use Pearson correlation [12].

$$\mathrm{corr}(X, Y) = \frac{\mathrm{cov}(X, Y)}{\sigma_X \sigma_Y} \tag{3.2}$$

where σ is the standard deviation of our distribution. This means that correlation is essentially a sort of normalized covariance. Correlation measures the linear dependence of two probability distributions. When the correlation is zero, the two distributions are linearly independent; when it is ± 1, the two distributions are perfectly linearly dependent with a slope of $\pm a, a \in \mathbb{R}_{>0}$. Note that a correlation of 0 does not mean that there is no dependence between the two distributions, but only that there is no linear dependence.

If we now find out that two features are highly correlated, we might think about removing one or multiple of them before training our machine learning model. If two features are correlated, removing one of them does not affect the performance of our model because even the most basic linear model can learn to calculate the second feature from the first one if need be. This can make training the model faster since we do not have that much data to look at, and it can reduce the risk of interpreting noise which might be present in the data.
So as a first step in analyzing the data, the correlation between all of the samples was calculated, which then gives a correlation matrix. This correlation matrix is then plotted as a heatmap which allows identifying all the features which are highly correlated.

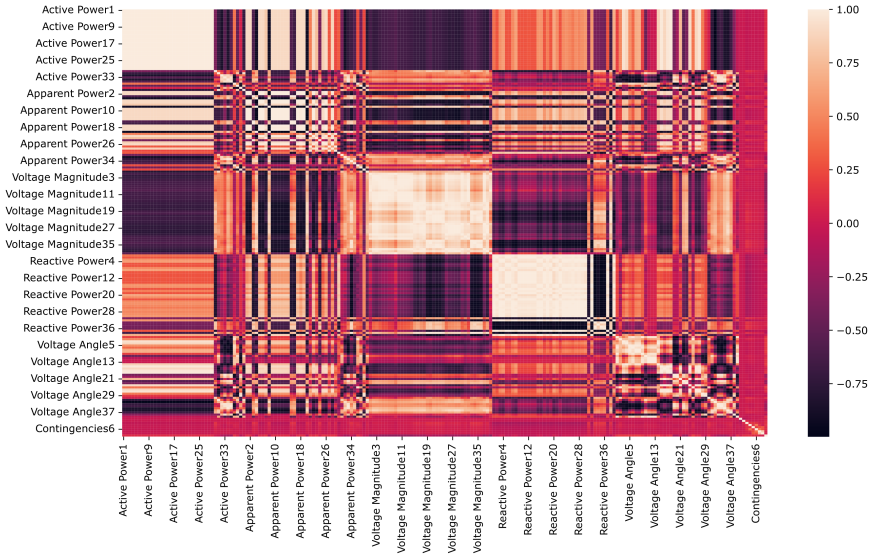


Figure 3.1: A correlation heatmap for the IEEE39 network.

In this map, the white regions are highly correlated, and the almost black regions are highly inversely correlated. The fact that large white and blackish patches are present means that there is some redundant data present which we can leave out of a further training step. Based on this, we select net active power, voltage magnitude and net apparent power. Net power is the generated power minus the demand.

3.4 Data Preparation

The data which we use to train our classifier is given as an adjacency matrix of the undisturbed power grid, the three features (Net Apparent Power, Voltage Magnitude and Voltage Angle) for each node and a set of contingencies. All of the contingencies are then removed from the adjacency matrix of the power grid to encode them in the topology of the graph.

3.5 Eigenvector Centrality Kernel

To use our data in an SVM we need a kernel as described in section [2.3.8]. The idea of this kernel is to compare two different graphs (where all of the nodes are the same but they have different edges) by how important their respective nodes are. In graph theory, the Eigenvector Centrality is the measure of the importance of a node in a network. The idea is that a node that is connected to many nodes with a high centrality also has a high centrality. We can define the centrality of a node i x_i as

$$x_i = \frac{1}{\lambda} \sum_{j=1}^{n} A_{ij} x_j, \tag{3.3}$$

where λ is a constant parameter and A_{ij} is 1 if there is a connection between nodes i and j and 0 otherwise. If now write this equation in vector form, we get

$$\lambda \mathbf{x} = \mathbf{A}\mathbf{x} \tag{3.4}$$

in which $\mathbf{x} = (x_1, x_2, ..., x_n)$ and $\mathbf{A}$ is the adjacency matrix of the vector. This means that $\mathbf{x}$ is an eigenvector of $\mathbf{A}$ with eigenvalue λ. Assuming non-negative centralities, it can be shown that λ must be the largest eigenvalue of $\mathbf{A}$ and $\mathbf{x}$ its corresponding eigenvector. For further use we will define the function $\mathbf{f}_{EV}(A) = \mathbf{x}$ where $\mathbf{x}$ is the list of eigenvector centralities. This definition means that the eigenvector centrality depends both on the number of connections a node has and the importance of its neighbouring nodes. So a node with many connections might not necessarily have a higher centrality value than a node with few connections to important nodes. [13] If we calculate the centralities for the IEEE39-network, we can visualize the graph of the network and the importance of its nodes. In the following figure [3.2], the centralities are encoded in both the size of the nodes and their colours. We see that nodes that have more connections generally have a higher centrality but also that nodes that are close to important nodes are themselves more important.
If we now have contingencies in a system, a node with a once very high centrality might now have a much lower one due to it not being as well connected anymore. If we now want to compare two graphs, we can instead just compare their EV centrality. This is our Eigenvector centrality graph kernel

$$k_{EV}(x_1, x_2) = \mathbf{f}_{EV}(x_1)^T \mathbf{f}_{EV}(x_2) \tag{3.5}$$

which is the dot product of the EV centrality vectors of two graphs x_1 and x_2. One major disadvantage of this kernel is that it can not deal with islands in a graph. As it turns out that having portions of the grid being completely isolated from the grid is not an uncommon occurrence and does not solely determine its stability. If we have such islands in our grid, the EV centrality of all the nodes in such an island is just 0, which means we lose valuable information. [14] proposes a different centrality measure called forest distance closeness centrality, which might prevent this problem. However, in this thesis, this is not further considered as we will now be taking a look at tried and tested graph kernels.

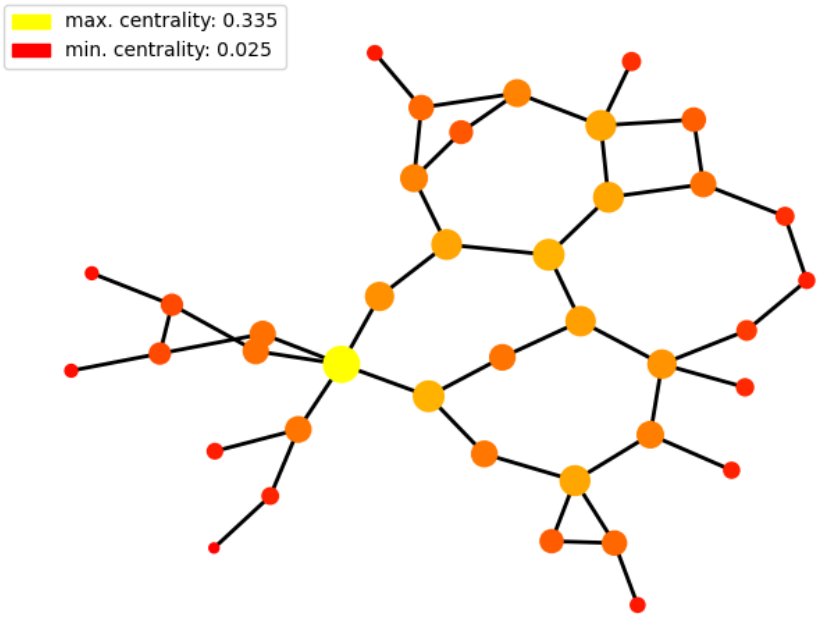


Figure 3.2: The graph of the undisturbed IEEE39 network and its eigenvector centralities.

3.6 Weisfeiler-Lehman Kernel

Weisfeiler-Lehman (WL) kernels were proposed in 2011 by Nino Shervashidze as a class of kernels which can relatively quickly be computed even on large graphs. It is based on the Weisfeiler-Lehman isomorphism test for graphs [15]. For this, we define a graph as a triplet (V, E, l), where V is the set of nodes (in our case, this corresponds to buses), E is the set of undirected edges and $l : V \to \Sigma$ is a function, which assigns a unique label from a dictionary Σ to nodes on the graph.

3.6.1 The Weisfeiler-Lehman Test of Graph Isomorphism

The Weisfeiler-Lehman kernel is based on the 1-dimensional Weisfeiler-Lehman test of graph isomorphism. We say two graphs are isomorphic if they are structurally the same. Suppose we want to find out whether two graphs G and G' are isomorphic. The idea of the WL algorithm is to add to every node label the labels of its neighbouring nodes and then compress this list of labels into a new, unique label. This step is repeated n times. In the end, if two nodes have the same label, then their n-hop neighbourhood is definitely the same. If, after n steps, the node label sets of G and G' are the same, the two graphs are considered isomorphic, or, at least, the test could not show that they are not.

Let's look at a small example of this relabeling done for a single small graph. If we

Algorithm 1 i-th iteration of the 1-dim. Weisfeiler-Lehman test of isomorphism [10]

1: Multiset-label determination

- For each vertex assign a multiset-label $M_i(v)$ to each node v in G and G' which consists of the multiset $\{l_{i-1}(u)|u \in \mathcal{N}(v)\}$ where $\mathcal{N}(v)$ is the set of all the neighboring nodes of a node v.

2: Sorting each multiset

- Sort all the elements in the multiset-label $M_i(v)$ and concatenate them into a string $s_i(v)$.
- Add the label of the node $l_{i-1}(v)$ to the front of the string $s_i(v)$.

3: Compressing each multiset into a label

- Map each string $s_i(v)$ into a new, compressed label using a function $f : \Sigma^* \to \sigma$ such that $f(s_i(v)) = f(s_i(w))$ only if $s_i(v) = s_i(w)$ which means that their i-hop neighborhoods are the same.

4: Relabeling

- Set $l_i(v) = f(s_i(v))$ for all nodes in G and G'

want to compare two graphs for isomorphism, we have to do all the steps for both graphs at the same time as described in 1.

3.6.2 The Weisfeiler-Lehman Subtree Kernel

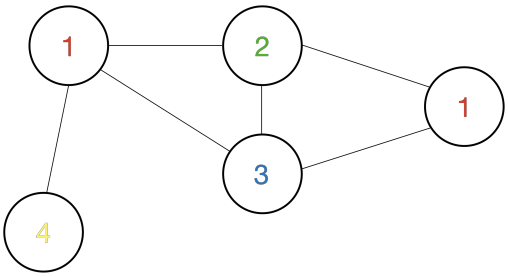
We can imagine one step of the Weisfeiler-Lehman relabeling as a function $r((V, E, l_i)) = (V, E, l_{i+1})$ where i denotes the number of times the relabeling has been performed. Note that this function depends on the set of graphs which is used. This is because, in the relabeling step, we need to know if a certain multiset label has already occurred in one of the other graphs. Using this function we can call the graph *at height i* of the graph $G = (V, E, l_0)$ as the graph $G_i = (V, E, l_i)$. We then call the sequence of Weisfeiler-Lehman graphs

$$\{G_0, G_1, ..., G_h\} = \{(V, E, l_0), (V, E, l_1), ..., (V, E, l_h)\} \tag{3.6}$$

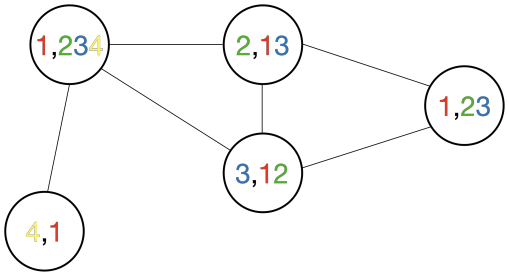
where G_0 is the initial graph, the *Weisfeiler-Lehman sequence up to height h of G*. So $G = G_0$ is the original graph and $G_1 = r(G_0)$ is the graph after one relabeling step and so on. Note that the only thing which changes is the labelling function l, whereas V and E stay the same.

Definition *Let k be any kernel for graphs that we will call the base kernel. Then the Weisfeiler-Lehman kernel with h iterations with the base kernel k is defined as*

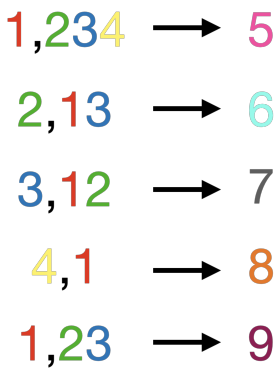
$$k_{WL}^{(h)}(G, G') = k(G_0, G'_0) + k(G_1, G'_1) + ... + k(G_h, G'_h) \tag{3.7}$$



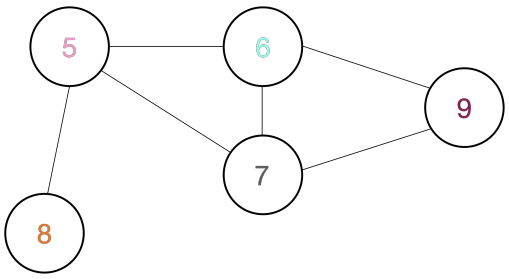
(a) The given labeled graph G_0



(b) The resulting multiset-labels after the first and second step



(c) Label compression



(d) The final graph after relabeling G_1

Figure 3.3: The Weisfeiler-Lehman test of isomorphism

where h is the number of Weisfeiler-Lehman iterations and $\{G_0, ..., G_h\}$ *and* $\{G'_0, ..., G'_h\}$ *are Weisfeiler-Lehman sequences of G and G' respectively.* [10] It can be shown that the Weisfeiler-Lehman kernel is positive semidefinite as long as the base kernel k is positive semidefinite as well.

3.6.3 Weisfeiler-Lehman Subtree Kernel

Using this general definition for Weisfeiler-Lehman kernels, we can now define one such kernel, which is the Weisfeiler-Lehman subtree kernel. First we define a *counting function* $c : \{G, G'\} \times \Sigma_i \to \mathbb{R}$ where $\Sigma_i \subseteq \Sigma$ is the subset of all the labels that occur in either G or G' at the end of the i-th iteration of the WL algorithm. Then $c_i(G, \sigma_{ij})$ is the number of occurrences of the label σ_{ij} in the Graph G. The Weisfeiler-Lehman subtree kernel with h iterations can now be defined as:

$$k^{(h)}_{WLSubtree}(G, G') = \langle \phi^{(h)}_{WLSubtree}(G), \phi^{(h)}_{WLSubtree}(G') \rangle \tag{3.8}$$

where

$$\phi^{(h)}_{WLSubtree}(G) = (c_0(G, \sigma_{01}), ..., c_0(G, \sigma_{0|\Sigma_0|}), ..., c_h(G, \sigma_{h1}), ..., c_h(G, \sigma_{h|\Sigma_h|}))$$

and

$$\phi^{(h)}_{WLSubtree}(G') = (c_0(G', \sigma_{01}), ..., c_0(G', \sigma_{0|\Sigma_0|}), ..., c_h(G', \sigma_{h1}), ..., c_h(G', \sigma_{h|\Sigma_h|}))$$

[10] This means that first, we find a ϕ vector which lists the number of occurrences of each label for each graph. The kernel on two graphs is then defined as the dot product of their respective ϕ vectors. Speaking in terms of feature engineering, we essentially reduced our entire graphs down to a vector, which does not contain any information about the original node labels but only counts how often each label occurs. With large graphs, this enables us to drastically reduce the dimensionality of our graphs while still preserving valuable information about the neighbourhoods of each node. The time complexity of the WL subtree kernel can be calculated to be $O(hm)$ where h is the number of WL iterations and m is the number of nodes of the graph. Note that if we want to calculate this kernel for many graphs, we need to do so at the same time so we can ensure that the same label always encodes the same subtree. This problem can, however, be solved by using a hash function to assign a new label to each multiset label.

3.7 Wasserstein-Weisfeiler-Lehman Kernel

A limitation of the WL kernel is that the initial node labels need to be a set of discrete values. This can make sense if we consider molecules and our initial labels just encode the atoms. But when talking about power grids, there is no obvious way how one can set initial labels that are not all just 1. In fact, we can have multiple features per node, so we call this our *node feature vector* which includes net active power, voltage magnitude and net apparent power. Additionally, we can also have *edge weights*, which in the physical

world might be the capacity of a power line connecting two nodes. We could rank all the nodes by centrality and use the rank as the initial label, but then we have a unique label for each node which might not make physical sense. We could also use the type of node, so is it a generator or consumer, as an initial label, but this tells us nothing about the size of each node. In the end, we would like to have a WL kernel but with continuous node feature vectors. This is where the Wasserstein Weisfeiler-Lehman kernel [16] comes in.
With every iteration, the WL subtree kernel essentially updates each node label to contain more information about its extended neighbourhood so that after h iterations, each node label contains information about the h-hop neighbourhood of the node. We want a continuous version of this kernel to be able to achieve the same. The way this is done in [16] is by updating each node feature vector by adding the average node feature vectors of all the neighbouring nodes. In this way, we can also easily incorporate edge weights by simply weighting the average by these edge weights. This way, we update our node feature vector $a^h(v)$ in each iteration,

$$a^{h+1}(v) = \frac{1}{2}\left(a^h(v) + \frac{1}{\deg(v)} \sum_{u \in \mathcal{N}(v)} w(v,u) a^h(u)\right) \tag{3.9}$$

where $w(v,u)$ is the weight of the edge connection v and its neighbor u. In the case where there are no edge weights we just use $w(v,u) = 1$. We use the factor $1/2$ to ensure that the new node features do not blow up. Now we have a continuous relabeling function for our new kernel function. As seen in section [3.7], we now also need some sort of kernel function which essentially compares how close two graphs are. Counting how often a label occurs, like in the WL subtree kernel, is once again not possible due to the continuous feature vectors. An obvious way of comparing how close two graphs are is to just take the distance between their updated node feature vectors. For this, the so-called Wasserstein distance is used.

3.7.1 Wasserstein Distance

The *Wasserstein distance* is a distance metric between two probability distributions. It is also called the *Earth Movers Distance*. Intuitively if each distribution is a unit amount of piled up soil, the Wasserstein distance is the minimum "cost" of turning one pile into the other. The L_p *-Wasserstein distance* between two probability distributions σ and μ on a metric space M equipped with a ground distance $d(x,y)$ for $p \in [1,\infty)$ *is defined as*

$$W_p(\sigma,\mu) := \left(\inf_{\gamma \in \Gamma(\sigma,\mu)} \int_{M \times M} d(x,y)^p \mathrm{d}\gamma(x,y)\right)^{\frac{1}{p}} \tag{3.10}$$

where $\Gamma(\sigma,\mu)$ is the set of all *transportation plans* $\gamma \in \Gamma(\sigma,\mu)$ over $M \times M$ with marginal distributions σ and μ. For our purposes, we will use the euclidean distance as our ground distance $d(x,y)$ and $p = 1$, so we will be using the L_1-Wasserstein metric. The Wasserstein distance has the advantage over the euclidean distance that if, e.g. σ and

μ are very narrow, have the same shape, and their centres are slightly shifted, they are clearly very close, and their Wasserstein distance reflects this, their euclidean distance would be quite large. So in this sense, the Wasserstein distance is much more reliable on 'spikey' probability distributions.

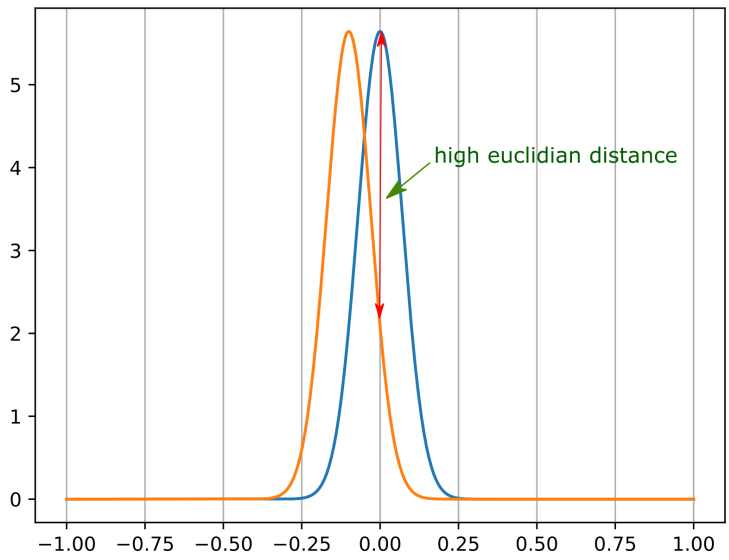


Figure 3.4: The Wasserstein distance has an advantage over the euclidean distance.

As we see, the distributions in the figure [3.4] have a large euclidean distance, but the Wasserstein distance between would actually be much smaller since moving mass from one distribution to the other would not take much work at all, i.e. we just have to move sideways by a little bit.
So to summarize, to calculate our WWL kernel, we first need to update the node labels to gather information about the neighbourhood of each node, like in the WL kernel. Then, to compare the two graphs, we take the Wasserstein distance between their updated node feature vectors.

3.8 Euclidean Weisfeiler-Lehman Kernel

The biggest problem with the Wasserstein distance is that it is computationally inefficient. This is because an optimal transport problem needs to be solved each time a distance is calculated. Even though the euclidean distance has drawbacks, we are still going to try doing the same but with a euclidean distance instead of a Wasserstein distance and see how both methods compare. If our data set is not very 'spikey', this should still work well.

Chapter 4

Experiment Setup

We compare the Eigenvector Centrality kernel (EVC), Weisfeiler-Lehman kernel (WL), Wasserstein Weisfeiler-Lehman kernel (WWL) and the Euclidean Weisfeiler-Lehman kernel (EWL) on the IEEE24 and IEEE39 grids. The data used to train and evaluate the models is generated with the Cascades module. All of the Weisfeiler-Lehman type kernels use a depth $d = 3$.
All of the computations are carried out on a single thread of an Apple M1 Pro SOC with 16 GB of RAM. The following times are for the computation of the training and testing kernels. The Eigenvector Centrality kernel takes about 20 minutes and the Weisfeiler-Lehman and Euclidean Weisfeiler-Lehman kernel are computed in roughly 2 hours for the given datasets. The Wasserstein Weisfeiler-Lehman kernel takes over 24 hours for both datasets.
As a classifier we use SVM from the scikit-learn library [17] which is based on libsvm [18]. To calculate the Wasserstein distance we use the python optimal transport library [19]. The only tunable parameter is the regularization parameter which is chosen using grid search to be $C = 1$. The algorithms are evaluated using balanced accuracy. This metric is chosen as the classes are unbalanced in the data set (due to cascades being rare events and thus much less likely than no cascades). All of the python code can be accessed via `https://github.com/doweiss/Bachelor-Thesis`.

4.1 Balanced Accuracy

The balanced accuracy is defined as $\frac{\text{sensitivity + specifity}}{2}$ where $\text{sensitivity} = \frac{\#\text{true pos.}}{\#\text{true pos. + }\#\text{false neg.}}$ and $\text{specificity} = \frac{\#\text{true neg.}}{\#\text{true neg. + }\#\text{false pos.}}$. This means even if we have a class imbalance, we can use balanced accuracy as an accurate metric.

4.2 Histogram of Projections

In section [2] we visualized SVMs using two-dimensional data sets. This way, we were able to see the decision plane and how far away all of the data points are. This is not

possible for our power grid data sets. To still get some visual understanding of the performance of our different kernels, we can use the histogram of projections proposed in [20]. The idea behind this is first to find the distance each of our data points has from the decision boundary.

$$y_j(\mathbf{w}^T\mathbf{x}_j - b) = |\mathbf{w}^T\mathbf{x}_j - b| = \left|\sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i^T \mathbf{x}_j - b\right| \tag{4.1}$$

or with kernels:

$$\left|\sum_{i=1}^{n} \alpha_i y_i k(\mathbf{x}_i, \mathbf{x}_j) - b\right| \tag{4.2}$$

which is the value of the decision function of the SVM.
The distance is normalized so that the margin equals one. We then plot a histogram of all the data points with their distance from the decision plane on the x-axis. The y-axis represents the number of samples per bar of the histogram. The bars are coloured according to the classes they belong to. We expect to see almost no data inside of the margins and then two clear clusters on either side of the decision plane.

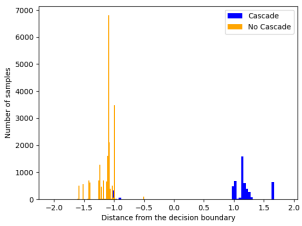
4.3 Results

4.3.1 Balanced Accuracy

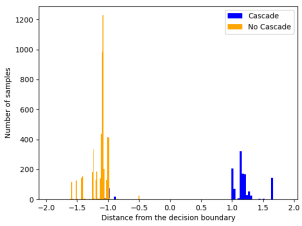
Method	IEEE24	IEEE39
EVC	0.946	0.662
WL	0.962	0.949
WWL	0.998	0.997
EWL	0.998	0.994

Table 4.1: Balanced classification accuracy on IEEE24 and IEEE39 using EVC, WL, WWL and EWL kernels.

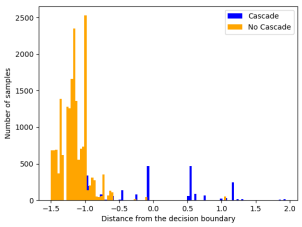
4.3.2 Histograms of Projection



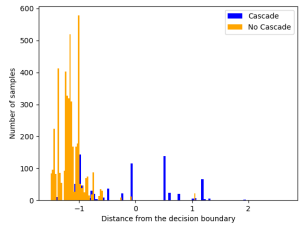
(a) IEEE24 training set



(b) IEEE24 test set

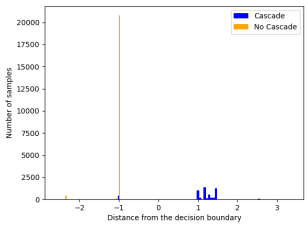


(c) IEEE39 training set

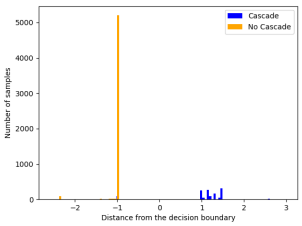


(d) IEEE39 test set

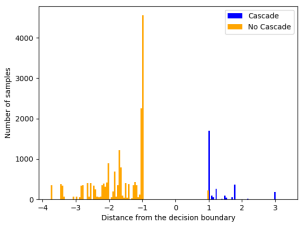
Figure 4.1: Histograms of projections for the eigenvector centrality kernel



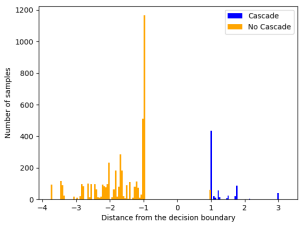
(a) IEEE24 training set



(b) IEEE24 test set

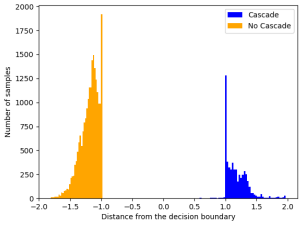


(c) IEEE39 training set

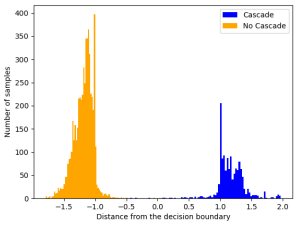


(d) IEEE39 test set

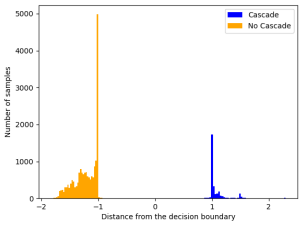
Figure 4.2: Histograms of projections for the Weisfeiler-Lehman kernel



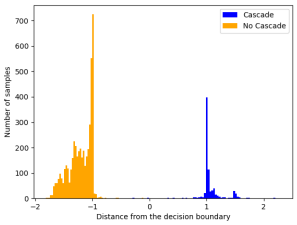
(a) IEEE24 training set



(b) IEEE24 test set

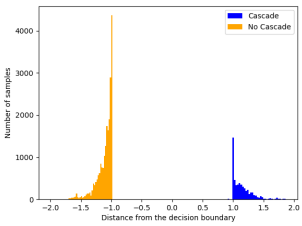


(c) IEEE39 training set

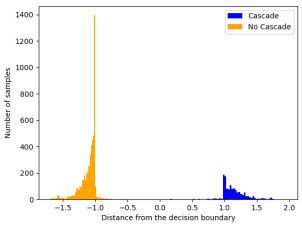


(d) IEEE39 test set

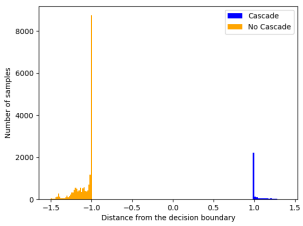
Figure 4.3: Histograms of projections for the Wasserstein Weisfeiler-Lehman kernel



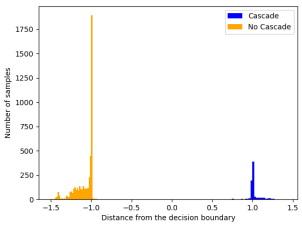
(a) IEEE24 training set



(b) IEEE24 test set



(c) IEEE39 training set



(d) IEEE39 test set

Figure 4.4: Histograms of projections for the Euclidean Weisfeiler-Lehman kernel

Chapter 5

Discussion

As we see in the results, kernel SVM can predict cascades with very high balanced accuracy. Especially the almost equal performance of the Euclidean WL kernel compared to the WWL kernel is very welcome as the EWL kernel is much quicker to compute, which enables faster predictions. Even the WL kernel, which only uses structural information, makes very accurate predictions. The Eigenvector Centrality kernel performs well on the IEEE24 data set but significantly worse on the IEEE39 data set, which also shows in the respective histogram of projections. The histograms of all of the other kernels show that they successfully manage to separate the two classes both for the training as well as the test sets.

Chapter 6

Outlook

In addition to the work done in this thesis there are several possible further research directions:

- Further research might go into further assessing the usability of the proposed methods in more real-world applications.
- Furthermore, the histogram of projections provides an exciting insight into the SVM. The question at hand is if the distance of a data point from the decision plane is in some way correlated with the severity of a cascade.
- Additionally, the proposed EV centrality kernel, although not the best performing, still did respectably on one of the data sets. The question is if the performance can be improved by using a centrality measure which can deal with islands.
- All of the methods have to be validated further on larger power grids such as the swiss transmission grid. This was unfortunately not possible in the scope of this thesis due to time and hardware constraints.

Bibliography

[1] B. Gjorgiev and G. Sansavini, "Nexus-e: Integrated energy systems modeling platform cascades module documentation," 2020.

[2] G. Hug, "Power system analysis," 2021.

[3] M. Abedi, M. R. Aghamohammadi, and M. T. Ameli, "Svm based intelligent predictor for identifying critical lines with potential for cascading failures using pre-outage operating data," *International Journal of Electrical Power and Energy Systems*, vol. 136, 2022.

[4] A. Krause and F. Yang, "Introduction to machine learning," 2022.

[5] A. Nandy and M. Biswas, *Reinforcement Learning*. 2018.

[6] A. Christmann and I. Steinwart, *Support Vector Machines*. 2008.

[7] M. Kubat, *An Introduction to Machine Learning*. 2021.

[8] K. U. Hoeffgen, H. U. Simon, and K. S. V. Horn, "Robust trainability of single neurons," *Journal of Computer and System Sciences*, vol. 50, 1995.

[9] J. Suzuki, *Kernel Methods for Machine Learning with Math and Python*. 2022.

[10] N. Shervashidze, P. Schweitzer, E. J. V. Leeuwen, K. Mehlhorn, and K. M. Borgwardt, "Weisfeiler-lehman graph kernels," *Journal of Machine Learning Research*, vol. 12, 2011.

[11] N. M. Kriege, F. D. Johansson, and C. Morris, "A survey on graph kernels," *Applied Network Science*, vol. 5, 2020.

[12] L. Meier and P. Cheridito, "Statistik und wahrscheinlichkeitsrechnung," 2021.

[13] M. Newman, *Mathematics of Networks*, pp. 1–8. 01 2008.

[14] Y. Jin, Q. Bao, and Z. Zhang, "Forest distance closeness centrality in disconnected graphs," in *2019 IEEE International Conference on Data Mining (ICDM)*, pp. 339–348, 2019.

[15] B. Y. Weisfeiler and A. A. Leman, "A reduction of a graph to a canonical form and an algebra arising from this reduction," *Nauchno-Technicheskaya Informatsia*, vol. 2, 1968.

[16] M. Togninalli, E. Ghisu, F. Llinares-Lopez, B. Rieck, and K. Borgwardt, "Wasserstein weisfeiler-lehman graph kernels," vol. 32, 2019.

[17] F. Pedregosa, 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, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[18] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[19] R. Flamary, N. Courty, A. Gramfort, M. Z. Alaya, A. Boisbunon, S. Chambon, L. Chapel, A. Corenflos, K. Fatras, N. Fournier, L. Gautheron, N. T. Gayraud, H. Janati, A. Rakotomamonjy, I. Redko, A. Rolet, A. Schutz, V. Seguy, D. J. Sutherland, R. Tavenard, A. Tong, and T. Vayer, "Pot: Python optimal transport," *Journal of Machine Learning Research*, vol. 22, no. 78, pp. 1–8, 2021.

[20] V. Cherkassky and S. Dhar, "Simple method for interpretation of high-dimensional nonlinear svm classification models," 2010.