The Australian National University
School of Computing
Convenor: Lexing Xie & Yuan-Sen Ting
Other Contributors: Dillon Chen & Evan Markou & Chamin Hewa K

Semester 1, 2023
Assignment 2
Weight: 18%

COMP4670/8600: Statistical Machine Learning

Release Date. 22nd April 2023

Due Date. Monday 15th May 2023 at 23:59 AEST.

Maximum credit. 100 Marks for COMP4670 and COMP8600 (scaled from 100 or 120 as explained below).

For submission, we are using Ed. Check under the submission tab and make sure to follow instructions on what to submit.

Although using Ed for submission, we recommend that you work on the assignment locally on your computer.

If you want to submit in LaTeX, we have a general template which you can use: https://quicklink.anu.edu.au/a4me.

Grading is different for COMP4670 and COMP8600 students. Grades will be calculated as a percentage out of 100 or 120. Questions which are required for COMP8600 students are marked in their titles. These are optional for COMP4670 students. For COMP4670, their final grade will be taken as the maximum percentage of only COMP4670 questions vs all questions.

Installation can be done by using `requirements.txt` provided. For a fresh install, create a new Python environment (*e.g.*, via Conda) and then run `pip install -r requirements.txt` in said environment:

```
$ conda create -n sml
$ conda activate sml
$ pip install -r requirements.txt
```

Collaboration, you are free to discuss the material in the assignment with your classmates. However, every proof and code submitted must be written by yourself without assistance. You should be able to explain your answers if requested.

Extension requests will be processed via the Extension app online form. Details of this are on Ed.

Other notes:

- When writing proofs, use the equation numbers when referring the equations in the assignment, *i.e.*, "Through Equation (3.1) we can show ..."
- You are not required to complete the assignment linearly, you may want to solve which-ever questions you can regardless of order of appearance
- If you have trouble with proving a statement, try reducing dimensionality of the problem first
- Unless stated otherwise, code is only graded on correctness of functions implemented, not on performance or code quality. Our main requirement on performance/code quality is that we can run your code in reasonable time when marking it.

Note: For the coding questions, you may use the packages that are already imported into your solutions but do not import any other package. There will be a penalty if you import extra packages.

Section 1: Kernels (50 or 60 Total Points)

One of the challenges in machine learning is dealing with vast and intricate data sets. Kernels are a useful tool in dealing with such challenges, as they allow our data to operate in a higher dimensional space that may have nice properties, such as linear separation of the class distributions. They can also be more intuitive to apply to difficult domains, where good basis functions to apply to the data are hard to think of, but similarity functions between data instances from the domain are easier to define.

In this section, we explore how kernels can be used in two such difficult domains: images and graphs. We will use support vector machines (SVMs) to solve classification problems in both domains, which will require us to define good kernel functions on the domains to achieve good performance.

Figure 1: Corn kernels, from `https://en.wikipedia.org/wiki/Corn_kernel`

Kernels on non-Euclidean data. In lectures, we were introduced to kernel functions $k : X \times X \to \mathbb{R}$ as inner products of feature maps

$$k(x, x') = \langle \phi(x), \phi(x') \rangle \tag{1.1}$$

where $\phi : X \to \mathbb{R}^m$ is a nonlinear mapping. We have mostly encountered the case when $X = \mathbb{R}^d$. One limitation of this definition is that we must specify the feature mapping ϕ. However, kernel functions provide a significant advantage, both theoretically and practically, since they do not require knowledge of ϕ's structure or computation. Thus, we provide a more general definition of the kernel function as follows.

Definition 1 (Positive definite kernel function)**.** Let X be a non-empty set. A function $k : X \times X \to \mathbb{R}$ is called a *positive definite kernel function* if

- k is symmetric, i.e. $k(x, x') = k(x', x), \forall x, x' \in X$, and
- for any set of objects $x_1, \ldots, x_n \in X$, the *kernel matrix* $K \in \mathbb{R}^{n \times n}$ defined by $K_{ij} = k(x_i, x_j)$ is positive semidefinite, i.e. $\mathbf{u}^T K \mathbf{u} \geq 0$ for all $\mathbf{u} \in \mathbb{R}^n$.

One may wonder how this new definition of a kernel function relates to the initial definition we learned. The answer is provided by Mercer's condition.

Theorem 1 (Mercer's condition)**.** Let X be a non-empty set. Then $k : X \times X \to \mathbb{R}$ is a positive definite kernel function if and only if there exists a Hilbert space[1] H and a feature map $\phi : X \to H$ such that

$$k(x, x') = \langle \phi(x), \phi(x') \rangle_H \tag{1.2}$$

where $\langle \cdot, \cdot \rangle_H$ denotes the inner product of H.

Equations (6.13)-(6.22) in Bishop (rewritten as Lemma 1 in the Appendix) can further be used to construct new kernel functions from existing ones.

[1] A vector space equipped with an inner product and is complete under the norm induced by the inner product. One can informally think of Hilbert spaces as a class of spaces that is slightly more abstract than Euclidean space: they have enough of the properties of Euclidean space that most of the theory we have for Euclidean space (linear algebra, calculus, etc.) can work. The inner product allows for distances and "angles", while the completeness allows for limits to be taken.

We now examine some examples of kernels that can be created for arbitrary objects.

Question 1.1: Kernel functions for arbitrary objects (12 Points)

Show that the following functions are valid kernel functions:

(a) Let $\sigma \in \mathbb{R}$. The Gaussian kernel defined by $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$

$$k(\mathbf{x}, \mathbf{x}') = \exp(-\|\mathbf{x} - \mathbf{x}'\|^2/2\sigma^2). \tag{1.3}$$

(b) Let D be a non-empty set of finite cardinality and denote $\mathcal{P}(D)$ to be the set containing all subsets of D. The intersection kernel defined by $k : \mathcal{P}(D) \times \mathcal{P}(D) \to \mathbb{R}$

$$k(A_1, A_2) = 2^{|A_1 \cap A_2|} \tag{1.4}$$

where $|A|$ denotes the size.

(c) Let $p(\mathbf{x} \mid \theta)$ be a parametric generative model. The Fisher kernel defined by $k : X \times X \to \mathbb{R}$,

$$k(\mathbf{x}, \mathbf{x}') = g_\theta(\mathbf{x})^T \mathbf{F}^{-1} g_\theta(\mathbf{x}') \tag{1.5}$$

where $g_\theta(\mathbf{x})$ is the Fisher score defined by

$$g_\theta(\mathbf{x}) = \nabla_\theta \log p(\mathbf{x} \mid \theta). \tag{1.6}$$

and $\mathbf{F}$ is the Fisher information matrix defined by

$$\mathbf{F} = \mathbb{E}_{p(\mathbf{x}|\theta)}[g_\theta(\mathbf{x}) g_\theta(\mathbf{x})^T]. \tag{1.7}$$

For all the above kernel functions, you must show that they are valid kernels using the definition or a theorem. You cannot state that they are kernels because the textbook says so.

Hint: Use (6.13)-(6.22) in the textbook (rewritten as Lemma 1 in the Appendix) to construct the kernels, thus showing that they are valid. For (b), use (6.19) and figure out a suitable mapping function $\phi(\cdot)$.

For more explanation about these kernels and what type of situations they are used in, see Section A.1 in the Appendix.

The Fisher Kernel for Image Classification

Before convolution neural networks became the standard choice for extracting features from neural networks, logistic regression and kernel methods utilizing local feature descriptions were widely used. In this section, we will explore one such method, based on [2], which uses multiple fundamental topics covered in lectures.

We wish to classify images using an SVM. To do this, we first need to determine a kernel function for describing how similar two images are. However, a good similarity function on images is hard to define. If we can specify or learn a generative model on the space of images, then we can use the Fisher kernel described earlier.

So now we have shifted the difficulty to describing a probability distribution over images. If we think of images as having a set of objects within them, it can make sense to model images as i.i.d. sampling of various image objects/features. Then we only need to describe how to get image features and a distribution over such features.

We will use SIFT features, which is a standard image feature extraction method that finds local features within the image while being invariant to the scale of the feature in the image. We will assume that it will extract a set of local features from an image X for us, i.e. it will generate the set $\phi(X) = \{\boldsymbol{x}_t \mid t = 1, ..., T\}$, $\boldsymbol{x}_t \in \mathbb{R}^d$. We then model features as a Gaussian Mixture Model (GMM), i.e. there is a

GMM over the space $\mathbb{R}^d$ from which features in images are sampled from. Let the parameters of the GMM be $\theta = \{\pi_i, \boldsymbol{\mu}_i, \Sigma_i \mid i = 1, ..., K\}$, where K is the number of Gaussians. We will simplify the model and assume that the covariance matrices are diagonal, i.e., $\Sigma_i = \text{diag}(\boldsymbol{\sigma}_i^2)$ with $\boldsymbol{\sigma}_i \in \mathbb{R}^d$, so $\theta = \{\pi_i, \boldsymbol{\mu}_i, \boldsymbol{\sigma}_i \mid i = 1, ..., K\}$.

As we assumed an image is a set of features drawn from the GMM, we now have a generative model for images specifying probabilities $p(X \mid \theta)$. Thus applying Fisher Kernels to this generative model, we get that

$$g_\theta(X) = \nabla_\theta \ln p(X \mid \theta) \tag{1.8}$$

$$\boldsymbol{F}_\theta = \mathbb{E}_{p(X|\theta)}[g_\theta(X) g_\theta(X)^T] \tag{1.9}$$

$$k(X, X') = g_\theta(X)^T \mathbf{F}_\theta^{-1} g_\theta(X'). \tag{1.10}$$

Given that the Fisher kernel is a valid kernel from Question 1.1 (Eq. 1.5), Mercer's Theorem implies that $k(X, Y) = G_\theta(X)^T G_\theta(X')$, where $G_\theta(X)$ is called the Fisher vector in the literature. Then using the Cholesky decomposition, as $\boldsymbol{F}_\theta^{-1}$ can be shown to be positive semi-definite, there exists a matrix $\boldsymbol{F}^{-\frac{1}{2}}$ such that $\boldsymbol{F} = \boldsymbol{F}^{\frac{1}{2}}(\boldsymbol{F}^{\frac{1}{2}})^T$. Thus the Fisher vectors are given by $G_\theta(X) = \boldsymbol{F}_\theta^{-\frac{1}{2}} g_\theta(X)$.

We simplify the model by only considering $\theta = \{\boldsymbol{\mu}_i, \boldsymbol{\sigma}_i \mid i = 1, ..., K\}$ in the Fisher kernel. Thus $g_\theta(X), G_\theta(X)$ are vectors of size $2Kd$ and $\boldsymbol{F}_\theta$ is a matrix of size $2Kd \times 2Kd$. Furthermore, we will use the result from [2] that under some assumptions, $\boldsymbol{F}$ is a diagonal matrix with diagonal elements

$$f_{\boldsymbol{\mu}_i} = \mathbb{E}_{p(X|\theta)}\left[(\nabla_{\boldsymbol{\mu}_i} \ln p(X \mid \theta))^2\right] = \frac{T\pi_i}{\boldsymbol{\sigma}_i^2} \in \mathbb{R}^d \tag{1.11}$$

$$f_{\boldsymbol{\sigma}_i} = \mathbb{E}_{p(X|\theta)}\left[(\nabla_{\boldsymbol{\sigma}_i} \ln p(X \mid \theta))^2\right] = \frac{2T\pi_i}{\boldsymbol{\sigma}_i^2} \in \mathbb{R}^d \tag{1.12}$$

where both terms are vectors, and the vector division is done componentwise.

Note then that the $2Kd$-dimensional vector $G_\theta(X)$ is made up of K terms $G_{\boldsymbol{\mu}_i}(X) \in \mathbb{R}^d$ and K terms $G_{\boldsymbol{\sigma}_i}(X) \in \mathbb{R}^d$.

Question 1.2: Deriving the Fisher vectors (14 Points)

Show that

$$G_{\boldsymbol{\mu}_i}(X) = \frac{1}{\sqrt{T\pi_i}} \sum_{t=1}^{T} \gamma_t(i) \left(\frac{\mathbf{x}_t - \boldsymbol{\mu}_i}{\boldsymbol{\sigma}_i}\right) \tag{1.13}$$

$$G_{\boldsymbol{\sigma}_i}(X) = \frac{1}{\sqrt{2T\pi_i}} \sum_{t=1}^{T} \gamma_t(i) \left(\frac{(\mathbf{x}_t - \boldsymbol{\mu}_i)^2}{\boldsymbol{\sigma}_i^2} - \mathbf{1}\right). \tag{1.14}$$

(Again, note that these terms are vectors, and all operations on vectors (division, squaring, etc.) are performed elementwise.)

Here, we consider $\gamma_t(i)$ to be the soft assignment of descriptor x_t to Gaussian i in our GMM, given by:

$$\gamma_t(i) = \frac{\pi_i \mathcal{N}(\mathbf{x}_t \mid \boldsymbol{\mu}_i, \boldsymbol{\sigma}_i)}{\sum_{k=1}^{K} \pi_k \mathcal{N}(x_t \mid \boldsymbol{\mu}_k, \boldsymbol{\sigma}_k)}. \tag{1.15}$$

Hint: If you are stuck on understanding the model described above, see Section B in the Appendix for a little more depth.

We will now implement and perform some experiments on the canonical MNIST dataset, which involves classifying images of the ten numerical digits. The necessary groundwork for this experiment has been provided to you in the form of `image_kernels.py`. However, it remains for you to implement the Fisher vectors and a simple performance improvement technique. Note that the SVM algorithm utilized in this experiment has been adapted to the multiclass scenario by creating ten one-vs-rest classifiers.

Question 1.3: Implement the Fisher kernel (10 Points)

In `kernelsframework/kernel_definitions.py` complete the `compute_fisher_vector` (Eqn. (1.13) and (1.14)) function.
Note that for GMMs we use the `GaussianMixture` class from sklearn which has suitable documentation online. Use the `predict_proba` function from that class rather than calculating probabilities yourself (as the implementation is done in log space and thus has much less floating point error).
To test your code, run

```
python3 kernelsframework/image_kernels.py
```

in the root directory. Use the help argument `-h` to find out what parameters you can tune. Use the `--small` argument to run the training process faster to debug your code. Note that the command

```
python3 kernelsframework/image_kernels.py -k 16 -f 16 --small
```

takes under a minute to train, ignoring the data loading and Fisher vector construction. A correct implementation may achieve accuracy of at least 60% of these parameters. Training on the whole dataset may take up to one hour.

Hint: We will not be marking you for the performance of your code and only the correctness of your Fisher vector and power normalisation implementation. However, performance is a good indicator of whether your code is correct. You may perform a parameter search on the testset[a] in order to test your code. It is possible to get over 80% accuracy with the correct choice of parameters.

[a] You shouldn't do this in practice and instead use a validation set which is a subset of your training set.

Question 1.4 [COMP8600]: Why linear SVM and not logistic regression? (2 Points)

As we computed the feature mapping that is equivalent to Fisher kernels, the Fisher vectors, we could have used logistic regression with them. Logistic regression and SVM both have the potential to perform well in image classification tasks, but they approach the problem in different ways. Hypothesise a reason why we may prefer using linear SVM and not simply logistic regression for this task.

Hint: Try implementing logistic regression and observing the results, although doing this is not necessary to get full marks for the question.

The WL Kernel for Graph Classification

Prior to the emergence of graph neural networks as a widely used technique for generating powerful graph representations, kernel methods, such as support vector machines, were widely used to address graph prediction problems. However, devising an informative similarity measure between pairs of graphs is a more complex challenge than in the case of images, where robust methods like SIFT can be used to extract features and measure their similarity. In the context of graph kernels, this challenge is even more pronounced, as a simplistic approach, such as aggregating all the node features of a graph to create a vector representation and taking the dot product, neglects the critical structural characteristics of the graph that are necessary to distinguish between graphs for classification.

Graph kernels such as the Weisfeiler-Lehman (WL) graph kernel you will implement below are able to generate features based on the structure of input graphs for use of graph classification problems such as predicting properties of molecules represented as graphs. Let us begin with some notation and define the WL graph kernel.

Let $G = (V, E)$ be a graph where V is a (finite) set of nodes, and $E \subseteq V \times V$ is a set of edges. Define the neighbourhood of a node v to be $N(v) = \{u \mid (u, v) \in E\}$. Graph colouring $c : V \to \mathbb{N}$ is a function

that assigns colours to each node of a graph (really a natural number label, but this is called colouring due to the similarity to the graph colouring problem).

The WL kernel is based on the WL algorithm, which is a test for graph isomorphism. For each graph, the algorithm generates a colouring and outputs a representation of each graph as a multiset [2] of colours of neighbours of each node and using an injective function to construct a new colour for the node as described in Line 4 of their nodes' colours. If the multisets of two graphs are different, then the graphs are not isomorphic, but if they are the same, then they could be isomorphic but we cannot conclude this definitively: one can construct two non-isomorphic graphs that give the same multiset. For more background on the WL algorithm, see Section C in the appendix.

Algorithm 1: The WL algorithm for N graphs.

Data: Graphs $G_1, \ldots, G_n$, number of WL iterations h, injective function f.
Result: Graph colourings $l_i : V_i \to \mathbb{N}$ for $i = 1, \ldots, N$.

1 **for** $i = 1, \ldots, N$ **do**
2 $l_i(v) \leftarrow |N(v)|, \quad \forall v \in G_i$
3 **for** $k = 1, \ldots, h$ **do**
4 $l_i'(v) \leftarrow f(l_i(v), \{\!\{l_i(u) \mid u \in N(v)\}\!\}), \quad \forall v \in G_i$
5 $l_i \leftarrow l_i'$
6 **return** $l_1, \ldots, l_N$

We now explain the WL algorithm, which is shown in Alg. 1, and also demonstrated in Figure 2. Given a set of N graphs $\{G_1 = (V_1, E_1), \ldots, G_N = (V_N, E_N)\}$ the algorithm returns a colouring l_i for each graph G_i where it is possible the intersection of the ranges of two different colourings to be non-empty, i.e. there may exist i, j with $i \neq j$ such that $l_i(V_i) \cap l_j(V_j) \neq \emptyset$.

The algorithm initially sets the colour of each node to be its degree (Line 2). Then the algorithm iteratively updates node colours (Line 5) by combining local information: given a node v, a multiset of the colours of its neighbours is created, and then this is compressed into a single colour using an injective function f (Line 4).

To compute the similarity of two graphs, the WL kernel first runs this algorithm for a set number of iterations. The representation/feature mapping of a graph, $\phi(G)$, is then the histogram vector of the graph's colouring as output by the WL algorithm, and the WL kernel is the inner product of these features. More explicitly, let $\Sigma = [\sigma_1, \ldots, \sigma_m]$ be an ordering of all the colours in the final colouring of all graphs $G_1, ..., G_n$. Then

$$\phi(G) = [c(G, \sigma_1), \ldots, c(G, \sigma_m)] \tag{1.16}$$

where $c(G, \sigma_j)$ counts the number of nodes in G coloured σ_j, and

$$k(G, G') = \phi(G)^T \phi(G'). \tag{1.17}$$

Fig. 2 illustrates the WL algorithm, which uses string sorting and compression to define f. Step a of the figure refers to the initialisation step in line 2. Step b represents the input to f of the form $l_i(v), \{\!\{l_i(u) \mid u \in N(v)\}\!\}$. Steps c and d represent the updating of colours corresponding to lines 4 and 5. Step e represents Eq. 1.16. Note that for this assignment, we *ignore* the counts of the original node labels and labels of each iteration except the final iteration.

[2] A set in which elements can appear more than once. An example of a multiset: $\{\!\{1, 1, 1, 5, 3, 6, 6\}\!\}$.

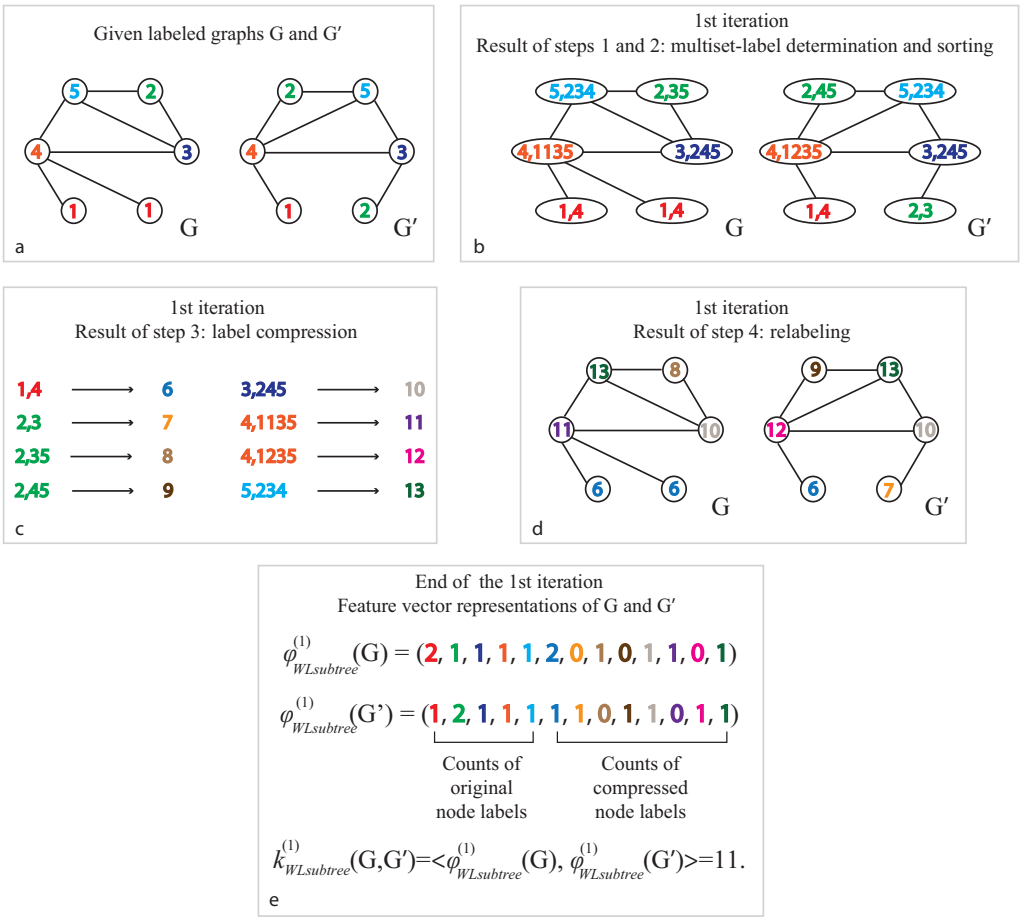

Figure 2: The WL algorithm and kernel, which are detailed in [3] and the text, have been implemented with a small modification. Unlike the original paper and schematic, where the compressed node labels of each iteration and the original node labels are considered, we have simplified the process and will only consider the compressed node labels from the final iteration. This modification has been made for the sake of simplicity. The WL algorithm and kernel are crucial in generating features for individual graphs. The algorithm produces a colour for each graph, which is based on the refinement of node labels in each graph iteration. During the refinement process, each node's label is updated to a multiset of its own label and the labels of its neighbours. This process is repeated for a fixed number of iterations or until the labels stabilize. The resulting label sets are then compared between the two graphs using a hash function. The hash function used in the WL graph kernel generates hash values that are unique for each label set of a graph. The kernel function then computes the number of common hash values between the two sets of labels as a measure of similarity. By using hash values, the WL graph kernel can compare the structural characteristics of graphs without explicitly comparing their actual node labels. This is particularly useful when dealing with noisy or incomplete data, where the exact node labels may not be reliable. Instead, the hash values provide a more robust and efficient method for measuring the similarity between graphs.

In the upcoming task, your objective is to implement the Weisfeiler-Lehman (WL) kernel to conduct binary and multiclass classification on a range of molecular datasets represented as graphs. The datasets that will be utilized include MUTAG, NCI1, and NCI109.

The MUTAG, NCI1, and NCI109 datasets are collections of chemical molecules that have been represented as graphs for use in graph classification tasks. The MUTAG dataset consists of 188 mutagenic aromatic and heteroaromatic nitro compounds and their nonmutagenic counterparts. Each molecule in the dataset is represented as a labelled graph, where the nodes represent atoms, and the edges represent chemical bonds between the atoms. The NCI1 and NCI109 datasets are subsets of the National Cancer Institute (NCI) database and contain molecular graphs representing chemical compounds screened for anticancer activity. The NCI1 dataset contains 4,110 compounds, and the NCI109 dataset contains 4,127 compounds.

The nodes in each graph represent atoms, and the edges represent chemical bonds. The task in these datasets is to classify the molecules as active or inactive against cancer cells based on their molecular graphs.

It is important to note that the primary framework, which comprises the training and testing pipeline, has already been established. As a result, your responsibility will be to focus solely on implementing the aforementioned algorithm.

Question 1.5: Implement the WL kernel (14 Points)

Complete the `compute_wl_features` function in the `kernelsframework/kernel_definitions.py` file for computing the features described in Eqn. 1.16 in Alg. 1.
Note that for graphs we use the `networkx` package which has suitable documentation online.
To execute your code, run

```
python3 kernelsframework/graph_kernels.py
```

in the root repository.

Hint: The SVM algorithm and training pipeline has been implemented for you. If you implement the `wl_features` function correctly, the test accuracy on the MUTAG, NCI1, and NCI109 datasets should be at least 70%. To improve the scores, you can perform a parameter search using a validation set. However, we will only evaluate the correctness of your implementation, not its performance.
Also, note that you will have to come up with an injective function f. Note that this can be more algorithmic than mathematical: you can decide on the fly where new inputs map to.

Question 1.6 [COMP8600]: Complexity (5 Points)

Find the computational complexity of calculating the kernel matrix produced by Eqn. 1.17 for N graphs with the method described above for computing Eqn. 1.16. Give your answer in big-O notation in terms of the number of graphs N, the maximum number of nodes n, the maximum node degree d, the number of iterations h in Alg. 1 and the size of the computed feature map in Eqn. 1.16 L.

Hint: You may assume the cost of computing f in Alg. 1 is negligible.

Within the scope of this course, a primary objective is to develop the skills necessary to select the optimal learning model for a given task. One approach towards achieving this objective is to gain a thorough understanding of the limitations inherent in certain algorithms, as well as to gain a deep understanding of the datasets with which we are working. In the following question, we will endeavour to explore the limitations of the WL kernel.

Question 1.7 [COMP8600]: Where the WL kernel fails. (3 Points)

You may notice that the train and test accuracy for the WL kernel on the EXP dataset is 50%, no better than random guessing. Hypothesise why the WL kernel may perform poorly on this dataset.

Hint: The EXP dataset consists of pairs of non-isomorphic[a] graph encodings of propositional formulas. Take a look at what the graph isomorphism problem is and how the WL kernel relates to it in Section C. You can also find a comprehensive description of the construction of the EXP dataset in [1].

[a] `https://en.wikipedia.org/wiki/Graph_isomorphism_problem`

Section 2: Gaussian Processes and Bayesian Optimisation (50 or 60 Total Points)

In this question, we consider Gaussian Processes (GPs) as a framework for regression tasks. GPs offer a flexible and powerful approach to modelling complex systems with a potentially infinite number of unknown parameters. For reference, we have defined the key concepts and notations of a Gaussian Process in Appendix D.

Given a set of input-output pairs $(X, \mathbf{y})$, where X denotes the input and $\mathbf{y}$ denotes the corresponding output, we want to predict the output of a new input X_*. To accomplish this, we must first define the prior and joint distributions over the output space, which we have discussed in detail in the Appendix D. With these distributions established, we can define the predictive distribution of the Gaussian Process as follows:

$$\mathbf{y}_* \mid X, \mathbf{y}, X_* \sim \mathcal{N}(\bar{\mathbf{y}}_*, \mathrm{cov}(\mathbf{y}_*)) \tag{2.1}$$

where $\mathbf{y}_*$ represents the predicted output, $\bar{\mathbf{y}}_*$ is the mean of the predictive distribution, and $\mathrm{cov}(\mathbf{y}_*)$ is the covariance matrix. To compute these quantities, we have derived equations (2.2a) and (2.2b) that express the mean and covariance of the predictive distribution in terms of the training data and the hyperparameters of the GP. Depending on our choice of the covariance function, we must carefully select its hyperparameters to ensure that our predictions are accurate and confident

$$\bar{\mathbf{y}}_* \overset{\text{def}}{=} \mathbb{E}[\mathbf{y}_* \mid X, \mathbf{y}, X_*] = K_*^\top K_\mathbf{y}^{-1} \mathbf{y} \tag{2.2a}$$

$$\mathrm{cov}(\mathbf{y}_*) = K_{**} - K_*^\top K_\mathbf{y}^{-1} K_*. \tag{2.2b}$$

If we have only a single point to predict, the above equations simplify to,

$$\bar{y}_* = \mathbf{k}_*^\top K_\mathbf{y}^{-1} \mathbf{y} \tag{2.3a}$$

$$\mathrm{V}(y_*) = k_{**} - \mathbf{k}_*^\top K_\mathbf{y}^{-1} \mathbf{k}_*. \tag{2.3b}$$

However, finding the optimal hyperparameters is a non-trivial task that requires optimisation algorithms. In this work, we focus on maximizing the log marginal likelihood of the observed data with respect to the hyperparameters $\boldsymbol{\theta}$. The log marginal likelihood, as defined in the following equation, quantifies how well our model fits the data and how complex our model is.

$$\log p(\mathbf{y} \mid \mathbf{X}, \boldsymbol{\theta}) = -\frac{1}{2} \mathbf{y}^\top K_\mathbf{y}^{-1} \mathbf{y} - \frac{1}{2} \log |K_\mathbf{y}| - \frac{n}{2} \log 2\pi. \tag{2.4}$$

One challenge that arises when computing the log marginal likelihood is the need to compute the inverse of the training covariance matrix $K_\mathbf{y}^{-1}$ multiple times. Directly computing the inverse is computationally expensive and numerically unstable, especially when the size of the training dataset is large. Instead, we use the Cholesky factorisation method to decompose the covariance matrix, as outlined in Algorithm 2, which is a more efficient and stable approach. Please note that the backslash notation $A \backslash \mathbf{b}$ represents the vector $\mathbf{x}$ that solves $A\mathbf{x} = \mathbf{b}$. Also, we will use the original definition of the Cholesky decomposition where the produced outcome is a lower triangular matrix.

Algorithm 2: Efficient Gaussian Process Regression

$L \leftarrow \mathrm{cholesky}(K_\mathbf{y})$ ▷ Cholesky factorisation for predictions and log marginal likelihood
$\boldsymbol{\alpha} \leftarrow L^\top \backslash (L \backslash \mathbf{y})$
$\bar{y}_* \leftarrow \mathbf{k}_*^\top \boldsymbol{\alpha}$ ▷ predictive mean Eq. 2.3
$\mathbf{v} \leftarrow L \backslash \mathbf{k}_*$
$\mathrm{V}(y_*) \leftarrow k_{**} - \mathbf{v}^\top \mathbf{v}$ ▷ predictive variance Eq. 2.3
$\log p(\mathbf{y}|\mathbf{X}) \leftarrow -\frac{1}{2} \mathbf{y}^\top \boldsymbol{\alpha} - \sum_i \log L_{ii} - \frac{n}{2} \log 2\pi$ ▷ log marginal likelihood Eq. 2.4

Choice of Covariance

As we previously discussed, the choice of covariance plays a crucial factor in the success of a GP regression. In this section, we will focus on stationary covariance functions that are invariant to translations in the input space. Specifically, we will consider a general class of covariance functions called Matérn.

The Matérn covariance function is a popular choice in the machine learning community due to its flexibility and ability to generalise to other well-known kernels such as the Radial Basis Function (RBF) class (i.e. Gaussian kernel). In fact, as ν approaches infinity, the Matérn covariance function converges to the RBF covariance function.

The Matérn covariance function is characterised by two positive parameters: ν and ℓ. The parameter ν determines the smoothness of the function, while ℓ controls the length-scale over which the covariance function varies. Γ denotes the Gamma function, and the modified Bessel function K_ν is used to ensure that the covariance function is non-negative. Here, we present the mathematical definition of the Matérn covariance function, which is given by equation 2.5:

$$k_{\text{Matern}}(x, y) = \sigma_f^2 \cdot \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\frac{\sqrt{2\nu} d}{\ell} \right)^{\nu} K_\nu \left(\frac{\sqrt{2\nu} d}{\ell} \right) \tag{2.5}$$

with positive parameters ν and ℓ. K_ν is a modified Bessel function, $d = \text{dist}(x, y)$ denotes the pairwise Euclidean distance, and σ_f^2 being the signal variance.

Question 2.1: Matérn Class Covariance Decomposition (5 Points)

For machine learning applications, it is customary to opt for half-integer values for the smoothness parameter ν in order to achieve further simplification of the Matérn covariance function. Show that for $\nu = p + 1/2$, where $p \in \mathbb{N}$, the Matérn covariance function can be expressed as a product of an exponential and a p-polynomial term:

$$k_{\text{Matern}}(x, y) = \sigma_f^2 \exp\left(-\frac{\sqrt{2p+1} d}{\ell} \right) \frac{\Gamma(p+1)}{\Gamma(2p+1)} \sum_{i=0}^{p} \frac{(p+i)!}{i!(p-i)!} \left(\frac{2\sqrt{2p+1} d}{\ell} \right)^{p-i} \tag{2.6}$$

Hint: You may utilise any Gamma function identities without proof from here (although you should include the identities you have used in each step of your proof). Additionally, you may use without proof the following formula:

$$\sqrt{\frac{1}{2}\pi/z} K_{p+1/2}(z) = \left(\frac{1}{2}\pi/z \right) \exp(-z) \sum_{i=0}^{p} \left(p + \frac{1}{2}, i \right) (2z)^{-i}$$

where $\left(n + \frac{1}{2}, i\right) = \frac{(n+i)!}{i!\Gamma(n-i+1)}$, and $K_*(\cdot)$ is the modified Bessel function.

As discussed, in the asymptotic regime of the Matérn class of covariance functions, it converges to the RBF kernel. Although this convergence entails a highly smooth covariance, it may not be the most suitable choice for machine learning tasks. In real-world settings, the smoothness argument cannot be taken for granted, and thus, we require more appropriate values of the smoothness parameter ν that are better suited to practical applications.

Question 2.2: Matérn Class Covariance Functions (10 Points)

a) The term "smoothness" in the Matérn covariance function is defined as such because it relates to the level of smoothness of the resulting stochastic process since processes produced by kernel functions tend to inherit the smoothness of the kernel. It is also related to the degree of mean square (MS) differentiability of a stochastic process (a detailed explanation of MS properties of stochastic processes can be found in Appendix F). For the Matérn class, the process $f(\mathbf{x})$ is m-times MS differentiable if and only if $\nu > m$.

Therefore, by choosing an appropriate value for the smoothness parameter ν for the Matérn covariance function, we can ensure that the approximated function, .i.e, the stochastic process is suitably smooth and MS differentiable for the desired application.

Given that we restrict to solutions that will satisfy Equation 2.6, an appropriate threshold of the smoothness value ν for the Matérn class that will produce a nice balance between the level of smoothness and the degree of MS differentiability for real-world machine learning problems is $n = 7/2$, where we accept choices of ν such that $\nu < 7/2$. Comment on the reasoning behind this choice of the threshold with respect to the smoothness and the MS differentiability of the underlying stochastic process.

b) Using the predefined threshold, implement the Matérn class covariance function (`__call__()`) under the `FIXME` sections in `kernels.py`, based on all accepted levels of smoothness ν. In your implementation, please also include the asymptotic case that induces a squared exponential (RBF) covariance function for Matérn class.

Hint: To solve the coding part of the question efficiently, plug the accepted values of ν in the Eqn. 2.6, simplify the resulting Matérn class expressions, and directly implement them as a function of ν.

Question 2.3: Gaussian Process Regression (10 Points)

Implement the `optimisation()`, `update()`, `predict()`, and `log_marginal_likelihood()` functions under the respective `FIXME` sections in `gp.py`. To assist you in this task, you should refer to Algorithm 2, which provides a helpful framework for implementing these functions.

By completing these functions, you will have a fully functioning Gaussian process regression algorithm that can be used for a variety of applications. So, take your time and ensure that your implementation is accurate and efficient. Note that the code will be also marked based on its efficiency.

Bayesian Optimisation

We have seen that Gaussian process regression is an influential tool for modelling data, but at the same time, it can also be employed to solve optimization problems. In this section, we will introduce the concept of Bayesian optimization that integrates Gaussian process regression to tackle optimization problems.

Bayesian optimization, also known as BayesOpt, is a zeroth-order optimization method designed to find the maximum of expensive cost functions. Zeroth-order methods do not use derivatives of any order to solve the optimisation problem. This approach employs Bayesian techniques by setting a prior distribution over the objective function and combining it with evidence to derive a posterior distribution. This permits a utility-based selection of the next observation to make on the objective function, which takes into account both exploration (sampling from areas of high uncertainty) and exploitation (sampling areas that are likely to offer an improvement over the current best observation).

We want to solve the optimisation problem

$$\max_{x \in \mathcal{X}} f(x) \tag{2.7}$$

where f:

- is *L-Lipschitz-continuous*, i.e. cannot change too rapidly.
- is *expensive to evaluate*, i.e. given an input x, computing $f(x)$ is expensive to do either in the sense of time or money (or other factors that will result in hurdles along the way), so we want to limit the number of evaluations.
- lacks a known special structure, such as convexity or quasi-convexity (similarly with concavity) or perhaps even differentiability, that would allow us to use known gradient-based optimisation algorithms. We say that f is a black box in those cases. As such, we want to use zeroth-order optimisation methods.

Note that our focus is on finding global optima rather than local optima.

In Bayesian Optimisation, we choose points to evaluate f sequentially and update our model afterwards. Thus at step t we have a dataset $\mathcal{D}_{1:t-1} = \{(\mathbf{x}_i, y_i) \mid i = 1, ..., t-1\}$ and make a decision of the next point $\mathbf{x}_t$ to evaluate at. This decision has an exploration vs exploitation tradeoff: it can try to explore the domain to get a better idea of f, or it can exploit where f is mostly likely to be high to try and improve the current estimate of the maximum: $\mathbf{x}^+_{t-1} = \text{argmax}_{i=1}^{t-1} f(\mathbf{x}_i)$.

The optimisation consists of two main components, a surrogate function f' that approximates our function f and an acquisition function u that tells us how to choose the next point. Thus at time step t, we update our surrogate function f' based on the previous evaluations to get an updated surrogate function $f'(\mathbf{x} \mid \mathcal{D}_{1:t-1})$, and then use our acquisition function $u(\mathbf{x} \mid \mathcal{D}_{1:t-1})$ to determine out next point by $\mathbf{x}_t = \text{argmax}_{\mathbf{x}}\, u(\mathbf{x} \mid \mathcal{D}_{1:t-1})$. The acquisition function $u(\mathbf{x} \mid \mathcal{D}_{1:t-1})$ uses information from our current estimate $f'(\mathbf{x} \mid \mathcal{D}_{1:t-1})$, such as the mean and the variance at certain points.

$$\mu(\mathbf{x} \mid \mathcal{D}_{1:t-1}) \approxeq \mathbb{E}_{p(f|\mathcal{D}_{1:t-1})}[f(\mathbf{x})] \tag{2.8}$$

$$\sigma^2(\mathbf{x} \mid \mathcal{D}_{1:t-1}) \approxeq \mathrm{V}_{p(f|\mathcal{D}_{1:t-1})}[f(\mathbf{x})]. \tag{2.9}$$

To summarise, the BayesOpt algorithm can be described in Algorithm 3.

Algorithm 3: Bayesian Optimisation

for $t = 1, 2, \ldots$ **do**
- Choose $\mathbf{x}_t$ by optimising the acquisition function u over the Gaussian Process such that:
- $\mathbf{x}_t = \text{argmax}_{\mathbf{x}}\, u(\mathbf{x} \mid \mathcal{D}_{1:t-1})$.
- Sample the objective function: $y_t = f(\mathbf{x}_t) + \epsilon$.
- Augment the data $\mathcal{D}_{1:t} = \mathcal{D}_{1:t-1} \cup \{(\mathbf{x}_t, y_t)\}$ and update the GP.

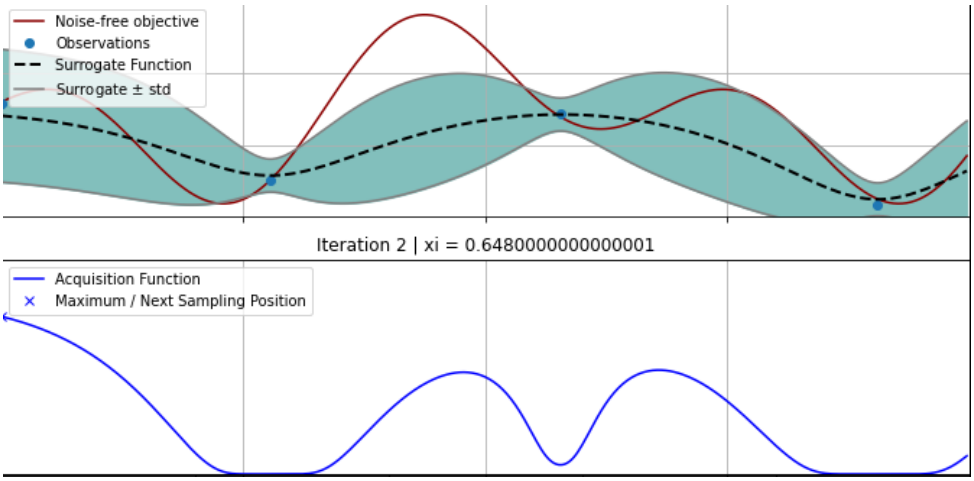

Figure 3: A visualisation of the objective function f, statistics of the surrogate model f', and the acquisition function u. At this step, the only thing we know about the f is the three points we have evaluated, which make up $\mathcal{D}_{1:3}$. From this, our surrogate model f' estimates what f is likely to be, and in this case, keeps track of the expected position of f, $\mu(\mathbf{x} \mid \mathcal{D}_{1:t-1})$ (the dashed line), and the standard deviation in this estimate, $\sigma(\mathbf{x} \mid \mathcal{D}_{1:t-1})$ (the coloured region). Finally, on the bottom, we have a plot of the acquisition function, which takes $\mu(\mathbf{x} \mid \mathcal{D}_{1:t-1})$ and $\sigma(\mathbf{x} \mid \mathcal{D}_{1:t-1})$ into consideration, and the maximum indicating where we should evaluate next is shown. This image is taken from the framework you will be coding in, so your output will look like this (at some step of the optimisation).

Surrogate Model. We will use the Gaussian process regression model that we have implemented before. Since our evaluation functions will be continuous, a GPR is an ideal surrogate model, and it can give us the mean $\mu(\mathbf{x} \mid \mathcal{D}_{1:t-1})$ and standard deviation $\sigma(\mathbf{x} \mid \mathcal{D}_{1:t-1})$ at input point. In general, you can read about surrogate models here.

Improvement-based Acquisition Functions. Now, let's turn over focus on the acquisition functions. The role of the acquisition function is to guide the search for the optimum. Typically, acquisition functions are defined such that high acquisition corresponds to potentially high values of the objective function, whether because the prediction is high, the uncertainty is great, or both. Maximising the acquisition function is used to select the next point at which to evaluate the function. That is, we wish to sample f at $\text{argmax}_x\, u(x \mid D)$, where $u(\cdot)$ is the generic symbol for an acquisition function.

The concept of determining the "next best point" is inherently ambiguous, resulting in several acquisition functions that capture different aspects of this concept. Nevertheless, for the purposes of this investigation, we shall concentrate solely on the improvement-based acquisition functions. This class of functions is based on the random variable *Improvement*:

$$\mathrm{I}(\mathbf{x}) = \max\{0, f(\mathbf{x}) - f(\mathbf{x}^+_{t-1}) - \xi\}, \tag{2.10}$$

where $f(\mathbf{x}^+_{t-1})$ is the incumbent solution. Note that $\xi \geq 0$ is the exploration-exploitation trade-off *hyper-parameter*, and it is employed to balance the exploration and exploitation of the objective function. The larger the value of ξ, the more substantial the barrier for sampling a point that does not vastly improve upon the incumbent point, limiting "exploitation" while encouraging "exploration".

Intuitively, $\mathrm{I}(\mathbf{x})$ assigns a reward of $\{f(\mathbf{x}) - f(\mathbf{x}^+_{t-1}) - \xi\}$ if $f(\mathbf{x}) > f(\mathbf{x}^+_{t-1}) + \xi$, and zero otherwise. Therefore, we should select the point that has the highest probability of improving upon the incumbent function value $f(\mathbf{x}^+_{t-1})$. This effectively means that we should maximise the probability of the event $\{\Pr(\mathrm{I}(\mathbf{x}) > 0)\}$ or equivalently maximise the probability of the event $\{f(\mathbf{x}) > f(\mathbf{x}^+_{t-1}) + \xi\}$. Another way of seeing this is by taking the expectation of the utility function $\mathrm{u}(\mathbf{x} \mid \mathcal{D}_{1:t-1}) = 1_{\{f(\mathbf{x}) > f(\mathbf{x}^+_{t-1}) + \xi\}}$, which is $\mathrm{u}(\mathbf{x} \mid \mathcal{D}_{1:t-1}) = 1$ when $f(\mathbf{x}) > f(\mathbf{x}^+_{t-1}) + \xi$ and zero otherwise.

A well-known acquisition function is the maximisation of this *probability of improvement* (PI) over the

$f(\mathbf{x}^+_{t-1})$, and is defined as such,

$$\begin{aligned}\mathrm{PI}(\mathbf{x}) &= \Pr\left(\mathrm{I}(\mathbf{x}) > 0 \mid \mathcal{D}_{1:t-1}\right) \\ &= 1 - \Pr\left(\frac{f(\mathbf{x}) - \mu(\mathbf{x} \mid \mathcal{D}_{1:t-1})}{\sigma(\mathbf{x} \mid \mathcal{D}_{1:t-1})} \leq \frac{f(\mathbf{x}^+_{t-1}) + \xi - \mu(\mathbf{x} \mid \mathcal{D}_{1:t-1})}{\sigma(\mathbf{x} \mid \mathcal{D}_{1:t-1})} \mid \mathcal{D}_{1:t-1}\right) \Rightarrow \text{reparameterisation trick } u \\ &= \Phi(Z),\end{aligned} \tag{2.11}$$

where

$$Z = \frac{\mu(\mathbf{x} \mid \mathcal{D}_{1:t-1}) - f(\mathbf{x}^+_{t-1}) - \xi}{\sigma(\mathbf{x} \mid \mathcal{D}_{1:t-1})}, \tag{2.12}$$

$\phi(\cdot)$ and $\Phi(\cdot)$ are the PDF and CDF of the standard normal distribution $\mathcal{N}(0, 1)$, and ξ is the exploration-exploitation trade-off *hyperparameter*, as before. The full derivation can be found in Appendix E.

However, the solution of the PI acquisition function is *greedy* as it selects the point most likely to offer an improvement of at least ξ. A somewhat more satisfying alternative acquisition function would be one that takes into account not only the probability of improvement but also the magnitude of the improvement a point can potentially yield. Ideally, we would minimise the expected deviation from the true maximum $f(\mathbf{x}^\star)$ when choosing a new trial point.

$$\begin{aligned}\mathbf{x}_t &= \operatorname*{argmin}_{\mathbf{x}} \mathop{\mathbb{E}}_{\Pr(f_t \mid \mathcal{D}_{1:t-1})} \left[\|f_t(\mathbf{x}) - f(\mathbf{x}^\star)\| \mid \mathcal{D}_{1:t-1}\right] \\ &= \operatorname*{argmin}_{\mathbf{x}} \int \|f_t(\mathbf{x}) - f(\mathbf{x}^\star)\| \Pr(f_t \mid \mathcal{D}_{1:t-1}) \, \mathrm{d}f_t.\end{aligned} \tag{2.13}$$

However, this is not feasible, so we can instead maximise the expected improvement with respect to $f(\mathbf{x}^+)$

$$\mathbf{x}_t = \operatorname*{argmax}_{\mathbf{x}} \mathop{\mathbb{E}}_{p(f \mid \mathcal{D}_{1:t-1})} \left[\mathrm{I}(\mathbf{x}) \mid \mathcal{D}_{1:t-1}\right]. \tag{2.14}$$

This is the *expected improvement* (EI) acquisition function, which can be simplified to

$$\mathrm{EI}(\mathbf{x}) = \begin{cases} (\mu(\mathbf{x} \mid \mathcal{D}_{1:t-1}) - f(\mathbf{x}^+_{t-1}) - \xi)\Phi(Z) + \sigma(\mathbf{x} \mid \mathcal{D}_{1:t-1})\phi(Z) & \text{if } \sigma(\mathbf{x} \mid \mathcal{D}_{1:t-1}) > 0, \\ 0 & \text{if } \sigma(\mathbf{x} \mid \mathcal{D}_{1:t-1}) = 0. \end{cases} \tag{2.15}$$

where

$$Z = \begin{cases} \frac{\mu(\mathbf{x} \mid \mathcal{D}_{1:t-1}) - f(\mathbf{x}^+_{t-1}) - \xi}{\sigma(\mathbf{x} \mid \mathcal{D}_{1:t-1})} & \text{if } \sigma(\mathbf{x} \mid \mathcal{D}_{1:t-1}) > 0, \\ 0 & \text{if } \sigma(\mathbf{x} \mid \mathcal{D}_{1:t-1}) = 0. \end{cases} \tag{2.16}$$

Lastly, up until this point, we have assumed that our dataset of evaluations $\mathcal{D}$ has been free from noise. However, in practice, and as we discussed in Appendix D, the data can often be noisy, so we need to adapt. Rather than using the best observation $\mathbf{x}^+_{t-1} = \operatorname{argmax}_{i=1}^{t-1} f(\mathbf{x}_i)$, we use the whole distribution at the sample points and define the current best solution as the sample point with the highest expectation.

$$\boldsymbol{\mu}^+_{t-1} = \operatorname*{argmax}_{i=1,\ldots,t-1} \mathop{\mathbb{E}}_{p(f \mid \mathcal{D}_{1:t-1})} [f(\mathbf{x}_i)]. \tag{2.17}$$

Question 2.4 [COMP8600]: Expected Improvement (10 Points)

Derive the Expected Improvement (EI) acquisition function in Equation 2.15. That is, show that,

$$\mathop{\mathbb{E}}_{p(f \mid \mathcal{D}_{1:t-1})} [\mathrm{I}(\mathbf{x})] = \begin{cases} (\mu(\mathbf{x} \mid \mathcal{D}_{1:t-1}) - f(\mathbf{x}^+_{t-1}) - \xi)\Phi(Z) + \sigma(\mathbf{x} \mid \mathcal{D}_{1:t-1})\phi(Z) & \text{if } \sigma(\mathbf{x} \mid \mathcal{D}_{1:t-1}) > 0, \\ 0 & \text{if } \sigma(\mathbf{x} \mid \mathcal{D}_{1:t-1}) = 0. \end{cases}$$

where Z is defined in Equation 2.16.

Question 2.5: Acquisition Function (5 Points)

Implement the Probability of Improvement (PI) and Expected Improvement (EI) acquisitions functions under the respective `FIXME` sections in `acquisitions.py`. You should use the Equations 2.11 (the final result) and 2.15 for your implementation.

Hint: : Since we have introduced noisy observations in the GP, we need to use the definition of the incumbent solution as per Eq 2.17.

Question 2.6: Sampling via Acquisition Maximisation (4 Points)

Implement the sampling process in the `sample_next_point()` function, which is located in `bayesopt.py`, for the next point of the surrogate model by optimising the given acquisition function, i.e., solve the optimisation problem:

$$\mathbf{x}_t = \underset{\mathbf{x}}{\operatorname{argmax}}\, u(\mathbf{x} \mid \mathcal{D}_{1:t-1})$$

using a Limited-Memory BFGS solver.

Hint: : Optimise the equivalent minimisation problem to align with the signature of the built-in optimisation solver.

Question 2.7: Bayesian Optimisation (12 Points)

a) Implement the Bayesian Optimisation algorithm function (`__call__()`) under the respective `FIXME` section in `bayesopt.py`. You should use Algorithm 3 for your implementation.

Hint: : Following Algorithm 3, in your implementation, you do not need to implement the for-loop operation (line 1) explicitly, as this is taken care of implicitly by the animation code.

Note: The default choices for the exploitation-exploration trade-off hyperparameter ξ for the two acquisition functions are set to 0.2 for PI and 0.05 for EI, respectively. However, given the choice of the initial points, you will see that the algorithm doesn't always converge. If fact, since the function is multi-modal, it will require scheduling of this parameter ξ to achieve the global maximum in the given iterations. A simple scheduler would at first start with a higher ξ to account for more exploration and, as the iterations proceed, decay the parameter for more exploitation. However, this is a suggestion, and feel free to come up with a different scheduling strategy.

As long as you can justify your strategy and the algorithm converges, you will receive full marks. To check for convergence, run the `bayesopt_implementation_viewer.ipynb` for the single-dimensional case. We expect two things at your final iteration: 1) The surrogate model with *closely* approximates the expensive black-box function, and 2) as a consequence of 1) the surrogate model can find the global optimum of the black-box function.

b) Finally, run Bayesian optimisation with different Matérn class covariance functions, the ones you have defined in the Matérn class based on the smoothness ν. Discuss the results and compare the effect of the different Matérn kernels in the BayesOpt process.

Question 2.8: Intuition on Higher Dimensions (4 Points)

a) After you run the code on `bayesopt_implementation_viewer.ipynb` for the higher dimensional case, comment on the benefit of including a hyperparameter (meaning the parameters of the covariance function $\boldsymbol{\theta}$ of the surrogate model) optimisation approach on your Bayesian Optimisation algorithm. (3 Points)

b) Try running the code on `bayesopt_implementation_viewer.ipynb` for the higher dimensional case with different Matérn kernels and comment on the effect of such choice. (1 Point)

A Q1.1: Kernel Functions

Here we link the definition of a kernel function with the definition and content given in lectures / the textbook.

The textbook introduces kernel functions as a similarity function between data points x, x' that can be constructed by $k(x, x') = \phi(x)^T \phi(x')$ for some feature space mapping $\phi(\cdot)$ (p.g. 292). It then goes on to note that kernels are inner products in a feature space of arbitrary objects, so we can define kernel functions (similar functions) between arbitrary objects by showing it is equivalent to an inner product in some mapping of the objects.

Definition 1 formalises this idea while Theorem 1 shows why that definition works.

The following techniques for constructing new kernels (given in p.g. 296 in Bishop) then following

Lemma 1. Let $x, x' \in X$. If $k_1(x, x')$ and $k_2(x, x')$ are valid kernel functions, then the following kernels are also valid

$$k(x, x') = ck_1(x, x') \tag{6.13}$$
$$k(x, x') = f(x)k_1(x, x')f(x') \tag{6.14}$$
$$k(x, x') = q(k_1(x, x')) \tag{6.15}$$
$$k(x, x') = \exp(k_1(x, x')) \tag{6.16}$$
$$k(x, x') = k_1(x, x') + k_2(x, x') \tag{6.17}$$
$$k(x, x') = k_1(x, x')k_2(x, x') \tag{6.18}$$
$$k(x, x') = k_3(\phi(x), \phi(x')) \tag{6.19}$$
$$k(x, x') = x^T \mathbf{A} x', \qquad x, x' \in \mathbb{R}^n \tag{6.20}$$
$$k(x, x') = k_a(x_a, x'_a) + k_b(x_b, x'_b), \qquad x, x' \in X^n \tag{6.21}$$
$$k(x, x') = k_a(x_a, x'_a)k_b(x_b, x'_b), \qquad x, x' \in X^n \tag{6.22}$$

where $c > 0$ is a constant, $f(\cdot)$ is any function, $q(\cdot)$ is a polynomial with nonnegative coefficients, $\phi(x)$ is a function from $X \to \mathbb{R}^n$, $k_3(\cdot, \cdot)$ is a valid kernel in $\mathbb{R}^n$, $\mathbf{A}$ is a symmetric positive semidefinite matrix, x_a and x_b are variables (not neccessarily disjoint) with $x = (x_a, x_b)$, and k_a and k_b are valid kernel functions over their respective spaces.

A.1 Uses of the kernel functions in Question 1.1

The Gaussian kernel (Eq. 1.3) is a popular choice for kernel methods. It assigns weights to data points based on their distance from a query point, with closer points receiving higher weights.

The intersection kernel (Eq. 1.4) operates on sets, and its value is determined by the size of the intersection between two sets. The intersection kernel is useful for tasks such as document classification, where each document can be represented as a set of words, and the similarity between documents can be computed using the intersection kernel.

The Fisher kernel (Eq. 1.5) uses the gradient of the log-likelihood of the data with respect to the model parameters to compute the similarity measure. This is based on the intuition that the gradient of the log-likelihood contains information about how the parameters should be adjusted to improve the model's fit to the data. The idea behind using the gradient of the log-likelihood as a similarity measure is that if two objects are generated by similar models, then their gradients with respect to the model parameters should be similar as well. The Fisher information matrix represents the local geometry of the probability distribution that generates the observed data.

B Q1.2: Image Classification using SVMs, Fisher Kernels and SIFT features

We formalise the model used in Q1.2 from the ground up here. We first describe some of the components in more detail.

SIFT Features. Finding good features for images is a major research area. SIFT (Scale-invariant feature transform) is one of the most famous and widely feature extraction methods. It extracts a set of local features (features that describe a small area of the image) that are invariant to rotation and scale.

Fisher Score, Kernel and Vector The Fisher score $g_\theta(\mathbf{x})$ of a data point $\mathbf{x}$, which is the derivative of the log-likelihood w.r.t. the parameters

$$g_\theta(\mathbf{x}) = \nabla_\theta \log p(\mathbf{x} \mid \theta),$$

tells how sensitive the log-likelihood of the data point (or dataset) is to changes in the parameters and is what we maximise in maximum likelihood (when taken over a whole dataset). The Fisher information matrix $\boldsymbol{F}$ or $\boldsymbol{F}_\theta$ is the variance of the score

$$\mathbf{F} = \mathbb{E}_{p(\mathbf{x}|\theta)}[g_\theta(\mathbf{x}) g_\theta(\mathbf{x})^T]$$

(as the expectation of the score is 0) and measures how sensitive the log-likelihood is for a given parameter in general. Note that the Fisher score is a function of the data point $\mathbf{x}$ (or dataset) while the Fisher information matrix is not; it is constant w.r.t. the data as it takes the expectation over the data distribution.

In Fisher Kernels, where $k(\mathbf{x}, \mathbf{x}')$ is defined as

$$k(\mathbf{x}, \mathbf{x}') = g_\theta(\mathbf{x})^T \mathbf{F}^{-1} g_\theta(\mathbf{x}')$$

we consider the feature mapping as the score as it shows useful information about the data points w.r.t. the generative model and normalise it by the Fisher information matrix. As the Fisher Kernel is a valid kernel, it is equivalent to an inner product for some other feature mapping $\phi(\mathbf{x})$, and what exactly that is can be easily derived from the Fisher Kernel by using the Cholesky decomposition, as $\boldsymbol{F}$ can be shown to be positive semi-definite, there exists a matrix $\boldsymbol{F}^{-\frac{1}{2}}$ such that $\boldsymbol{F} = \boldsymbol{F}^{\frac{1}{2}}(\boldsymbol{F}^{\frac{1}{2}})^T$. Thus the feature mapping is $\phi(\mathbf{x}) = F^{-\frac{1}{2}} g_\theta(\mathbf{x})$, which we call the Fisher Vector. This can be thought of as a normalised Fisher score and is a commonly used feature mapping in machine learning as it allows for working linearly in the space the Fisher Kernel operates in.

B.1 Full model in depth

For a given image X, let $\phi(X) = \{\boldsymbol{x}_t \mid t = 1, ..., T\}$, $\boldsymbol{x}_t \in \mathbb{R}^d$, be the SIFT features extracted from the image. We will assume T is the same for all images, i.e. we extract an equal number of SIFT features for each image.

We will model SIFT features using a GMM with K components, with parameters $\theta = \{\pi_i, \boldsymbol{\mu}_i, \Sigma_i \mid i = 1, ..., K\}$. We will simplify the model and assume that the covariance matrices are diagonal, i.e., $\Sigma_i = \text{diag}(\boldsymbol{\sigma}_i^2)$ with $\boldsymbol{\sigma}_i \in \mathbb{R}^d$, so $\theta = \{\pi_i, \boldsymbol{\mu}_i, \boldsymbol{\sigma}_i \mid i = 1, ..., K\}$. Thus the probability of a SIFT feature $\boldsymbol{x}_t$ is

$$p(\boldsymbol{x}_t \mid \theta) = \sum_{i=1}^{K} \pi_i \mathcal{N}(\boldsymbol{x}_t \mid \boldsymbol{\mu}_i, \boldsymbol{\sigma}_i).$$

Assuming that the SIFT features for an image are i.i.d. sampled from this GMM, we can then calculate the probability of an image as

$$\begin{aligned} p(X \mid \theta) &= p(\phi(X) \mid \theta) \\ &= \prod_{t=1}^{T} p(\boldsymbol{x}_t \mid \theta), \end{aligned}$$

thus giving us a generative model for our task.

We then use the Fisher kernel on this generative model. Our Fisher score is then

$$\begin{aligned} g_\theta(X) &= \nabla_\theta \ln p(X \mid \theta) \\ &= \nabla_\theta \log \prod_{t=1}^{T} p(\boldsymbol{x}_t \mid \theta) \\ &= \sum_{t=1}^{T} \nabla_\theta \log p(\boldsymbol{x}_t \mid \theta). \end{aligned}$$

and our Fisher information matrix is

$$\begin{aligned} \boldsymbol{F} &= \mathbb{E}_{p(X|\theta)}[g_\theta(X) g_\theta(X)^T] \\ &= \mathbb{E}_{p(X|\theta)} \left[\begin{bmatrix} \sum_{t=1}^{T} \nabla_{\boldsymbol{\pi}} \log p(\boldsymbol{x}_t \mid \theta) \\ \sum_{t=1}^{T} \nabla_{\boldsymbol{\mu}} \log p(\boldsymbol{x}_t \mid \theta) \\ \sum_{t=1}^{T} \nabla_{\boldsymbol{\sigma}} \log p(\boldsymbol{x}_t \mid \theta) \end{bmatrix} \begin{bmatrix} \sum_{t=1}^{T} \nabla_{\boldsymbol{\pi}} \log p(\boldsymbol{x}_t \mid \theta) \\ \sum_{t=1}^{T} \nabla_{\boldsymbol{\mu}} \log p(\boldsymbol{x}_t \mid \theta) \\ \sum_{t=1}^{T} \nabla_{\boldsymbol{\sigma}} \log p(\boldsymbol{x}_t \mid \theta) \end{bmatrix}^T \right] \end{aligned}$$

which is of size $(K + 2Kd, K + 2Kd)$.

We simplify the model to only take the derivative w.r.t. μ and σ in the Fisher kernel. Furthermore, we will use the result from [2] that under some assumptions, $\boldsymbol{F}$ is a diagonal matrix with diagonal elements

$$f_{\boldsymbol{\mu}_i} = \mathbb{E}_{p(X|\theta)} \left[\left(\nabla_{\boldsymbol{\mu}_i} \ln p(X \mid \theta) \right)^2 \right] = \frac{T \pi_i}{\sigma_i^2} \in \mathbb{R}^d \tag{B.1}$$

$$f_{\boldsymbol{\sigma}_i} = \mathbb{E}_{p(X|\theta)} \left[\left(\nabla_{\boldsymbol{\sigma}_i} \ln p(X \mid \theta) \right)^2 \right] = \frac{2T \pi_i}{\sigma_i^2} \in \mathbb{R}^d. \tag{B.2}$$

The Fisher vector is then

$$G_\theta(X) = F_\theta^{-\frac{1}{2}} g_\theta(X).$$

C Q1.3: Graph Classification using SVMs and the WL Kernel

We formalise the model used in Q1.3 from the ground up here. We first describe some of the components in more detail.

Graph Isomorphism Problem Two graphs are considered isomorphic if they have the same structure, i.e. there is a mapping f from the nodes of one graph to the nodes of the other such that any property in one graph implies the same property in the other graph under the mapping. For example, if nodes u and v in the first graph have an edge between them, nodes $f(u)$ and $f(v)$ in the second graph must have an edge between them.

The graph isomorphism problem, determining if two graphs are isomorphic, is an important problem in graph theory. No polynomial time algorithm is known for it, but it has not been shown to be NP-complete either. The problem appears in many fields, especially computer science and chemistry.

WL Algorithm The Weisfeiler-Lehman (WL) algorithm/test for isomorphism computes a representation for each graph and uses that to determine if two graphs are isomorphic. However, it is not a conclusive test. If the representations are different, then the two graphs are not isomorphic, but if they are the same, then they are not necessarily isomorphic: one can construct two non-isomorphic graphs that give the same WL representation.

The idea of the algorithm is to store the local structure of the graph at each node by compressing information about the adjacent nodes. This is repeated until convergence or for a fixed number of iterations, where after each iteration, the information at each node depends on a larger subset of the graph. The final representation is the multiset of node structures with compressed information from each node, which is often referred to as a multiset or distribution of colours.

WL Kernel As two graphs being isomorphic means they are very similar (in fact, identical), it makes sense to define a similarity function for graphs based on graph isomorphism theory. The WL kernel computes multiple iterations of the WL algorithm to get a representation for each graph and then computes the similarity as the inner product of the histogram vectors of the two graphs (the histograms keep track of the number of nodes assigned to each colour).

D Q2: Gaussian Processes

Definition 2. *A Gaussian process (GP) is a collection of random variables, any finite number of which have a joint Gaussian distribution.*

Setup. We specify a GP by its mean function and covariance function[3] as such:

$$m(\mathbf{x}) = \mathbb{E}[f(\mathbf{x})], \tag{D.1a}$$
$$k(\mathbf{x}, \mathbf{x}') = \mathbb{E}[(f(\mathbf{x}) - m(\mathbf{x}))(f(\mathbf{x}') - m(\mathbf{x}'))] \tag{D.1b}$$

and mathematically define a Gaussian process as,

$$f(\mathbf{x}) \sim \mathcal{GP}(m(\mathbf{x}), k(\mathbf{x}, \mathbf{x}')) \tag{D.2}$$

In this assignment, we will perform Gaussian Process Regression (GPR) using noisy observations to account for more realistic modelling situations. Instead of directly observing $f(\mathbf{x})$, we observe a noisy transformation of $f(\mathbf{x})$. Thus, our evaluation functions will have the form of

$$\mathbf{y} = f(\mathbf{x}) + \epsilon \tag{D.3}$$

where ϵ is some random (Gaussian) noise.

Without loss of generality, we can assume the prior mean is zero (we can always add the mean vector later), and incorporating the noise into the covariance function, we can define the prior with respect to the covariance (using matrix notation) as,

$$K_{\mathbf{y}} = K(X, X) + \sigma_n^2 I \tag{D.4}$$

where σ_n^2 is the variance of the Gaussian noise, assuming it's derived from *i.i.d.* data.

In addition, for the noise training outputs $\mathbf{y}$ and the testing outputs $\mathbf{y}_*$, we can define their joint distribution by carefully partitioning the covariance matrix. This will produce the following result,

$$\begin{bmatrix} \mathbf{y} \\ \mathbf{y}_* \end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} K_{\mathbf{y}} & K_* \\ K_*^\top & K_{**} \end{bmatrix}\right) \tag{D.5}$$

where for simplicity, we have defined each block component of the covariance matrix as,

$K_{\mathbf{y}}$: defined as the noisy training covariance matrix in Eq. D.4
K_* : defined as the training-testing covariance matrix, i.e., $K(X, X_*)$
$K_*^\top$: defined as the testing-training covariance matrix, i.e., $K(X_*, X)$
K_{**} : defined as the testing covariance matrix, i.e., $K(X_*, X_*)$

[3] We use the terms *covariance function* and *kernel* interchangeably in this assignment.

E Q2: Bayesian Optimisation

Derivation of Eq 2.11

$$\begin{aligned}
\mathrm{PI}(\mathbf{x}) &= \Pr\left(\mathrm{I}(\mathbf{x}) > 0\right)^{4} \\
&= \mathbb{E}[1_{\{f(\mathbf{x}) > f(\mathbf{x}^{+}_{t-1}) + \xi\}}] \\
&= \Pr\left(f(\mathbf{x}) > f(\mathbf{x}^{+}_{t-1}) + \xi\right) \\
&= \Pr\left(\frac{f(\mathbf{x}) - \mu(\mathbf{x})}{\sigma(\mathbf{x})} > \frac{f(\mathbf{x}^{+}_{t-1}) + \xi - \mu(\mathbf{x})}{\sigma(\mathbf{x})}\right) \\
&= 1 - \Pr\left(\frac{f(\mathbf{x}) - \mu(\mathbf{x})}{\sigma(\mathbf{x})} \leq \frac{f(\mathbf{x}^{+}_{t-1}) + \xi - \mu(\mathbf{x})}{\sigma(\mathbf{x})}\right) \Rightarrow \text{reparameterisation trick } u \\
&= 1 - \int_{-\infty}^{-Z} \phi(u) du \\
&= 1 - \Phi(-Z) \\
&= \Phi(Z),
\end{aligned} \tag{E.1}$$

where

$$Z = \frac{\mu(\mathbf{x}) - f(\mathbf{x}^{+}_{t-1}) - \xi}{\sigma(\mathbf{x})}. \tag{E.2}$$

F Q2: Mean Square Continuity and Differentiability

It's difficult to find a simple relationship between the covariance function of a stochastic process and the smoothness of its realisations. In this section, however, we will relate the covariance function to what is known as the mean square properties of a stochastic process. Let's take a sequence of points, $\mathbf{x}_1, \mathbf{x}_2, ...,$ and a fixed point in $\mathbb{R}^d$, $\mathbf{x}_*$. We will define $\mathbf{x}_n \xrightarrow{L^2} \mathbf{x}_*$ to mean $|\mathbf{x}_n - \mathbf{x}_*| \to 0$ as $n \to \infty$. We say that $\{\mathbf{x}_n\}$ converges in L^2 (mean square convergence) if there exists fixed point $\mathbf{x}_*$ such that $\mathbf{x}_n \xrightarrow{L^2} \mathbf{x}_*$.
Suppose $f(\mathbf{x})$ is a stochastic process in $\mathbb{R}^d$. Then $f(\mathbf{x})$ is mean square continuous at $\mathbf{x}_*$, if:

$$\lim_{n \to \infty} \mathbb{E}[|f(\mathbf{x}_n) - f(\mathbf{x}_*)|^2] = 0 \tag{F.1}$$

If this holds for all $\mathbf{x}_* \in \mathcal{A}$ where $\mathcal{A} \subseteq \mathbb{R}^d$ then $f(\mathbf{x})$ is said to be continuous in the mean square (MS) over $\mathcal{A}$. If $f(\mathbf{x})$ is a weakly stationary process with a covariance function k, then it can be shown that $f(\mathbf{x})$ is MS continuous at $\mathbf{x}_*$ if and only if k is continuous at the origin. Since a weakly stationary stochastic process is either MS continuous everywhere or nowhere, it suffices to only check at $k(\mathbf{0})$.

Mean square (MS) differentiability has a similar definition as an L^2 limit. The MS derivative of a process $f(\mathbf{x})$ in the i-th direction is defined as:

$$\nabla_{\hat{\mathbf{e}_i}} f(\mathbf{x}) = \operatorname*{l.i.m}_{h \to 0} \frac{f(\mathbf{x} + h\hat{\mathbf{e}_i}) - f(\mathbf{x})}{h} \tag{F.2}$$

when the limit exists, where $\hat{\mathbf{e}_i}$ is the unit vector in the i-th direction. Here, the notation l. i. m denotes the limit in the mean square. It means that the expression in F.2 converges in L^2, that is:

$$\lim_{h \to 0} \mathbb{E}\left[\left(\frac{f(\mathbf{x} + h\hat{\mathbf{e}_i}) - f(\mathbf{x})}{h} - \nabla_{\hat{\mathbf{e}_i}} f(\mathbf{x})\right)^2\right] = 0 \tag{F.3}$$

For weakly stationary processes, we say that they are m-times MS differentiable if and only if the $2m$-th partial derivative of their matching covariance function exists and is finite at $\mathbf{x} = \mathbf{0}$. Thus, to check whether a stochastic process is m-times MS differentiable, it suffices to prove that $k^{(2m)}(\mathbf{0})$ exists and is finite. Notice that ultimately it is the properties of the covariance k around the origin that determine the smoothness properties (MS continuity and differentiability) of a stationary process.

[4] For simplicity, we have dropped the condition on the sample dataset, although note that this is implied.

References

[1] Ralph Abboud, İsmail İlkan Ceylan, Martin Grohe, and Thomas Lukasiewicz. The surprising power of graph neural networks with random node initialization. In *Proc. of the 30th International Joint Conference on Artificial Intelligence (IJCAI)*, 2021.

[2] Florent Perronnin and Christopher R. Dance. Fisher kernels on visual vocabularies for image categorization. In *Proc. of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2007.

[3] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler-lehman graph kernels. *J. Mach. Learn. Res.*, 12:2539–2561, 2011.