ChatGPT-Driven Machine Learning Code Generation for Android Malware Detection

Jordan Nelson

School of Architecture, Technology and Engineering, University of Brighton, Brighton, BN2 4GJ, United Kingdom.

Michalis Pavlidis

School of Architecture, Technology and Engineering, University of Brighton, Brighton, BN2 4GJ, United Kingdom.

Andrew Fish

Department of Computer Science, University of Liverpool, Liverpool, L69 3BX, United Kingdom.

Stelios Kapetanakis

Distributed Analytics Solutions, London, E14 6FD, United Kingdom.

Nikolaos Polatidis*

School of Architecture, Technology and Engineering, University of Brighton, Brighton, BN2 4GJ, United Kingdom.

***Correspondence to:** Dr Nikolaos Polatidis, School of Architecture, Technology and Engineering,
University of Brighton, Brighton, BN2 4GJ, United Kingdom.
E-mail: N.Polatidis@Brighton.ac.uk; ORCID: 0000-0003-4249-4953

Abstract

Android is a widely used operating system, primarily found on mobile phones and tablets. Applications (commonly known as "apps") for Android can be easily installed from Google Play, third-party stores, or manually using Android Package Kit (APK) files. Due to its growing popularity, Android has attracted significant attention from malicious actors deploying various forms of malware. To address this challenge, artificial intelligence-based approaches are increasingly used to protect systems from cyber-attacks. This research paper focuses on the application of ChatGPT, a powerful large language model, in cybersecurity, specifically for malware detection. It evaluates ChatGPT's potential as an innovative tool in fighting cyber threats, exploring the process of fine-tuning ChatGPT, its performance and its limitations in malware detection tasks. The objective is to reduce the effort and time required to generate AI-based malware detection systems, simplifying their development process. This research shows how ChatGPT can be utilized to generate code for detecting malware in structured datasets with high accuracy. The focus is not on introducing any new algorithms but on allow individuals without programming expertise to create and apply these models effectively.

Keywords: ChatGPT, Machine Learning, Neural Networks, Malware Detection

1. INTRODUCTION

Android is a prevalent operating system employed on mobile phones and tablets. Applications (referred to as "apps") for Android can be conveniently installed from Google Play, third-party stores, or manually from Android Package Kit (APK) files. Given the increasing prevalence of Android usage, it has garnered considerable attention from malicious actors who deploy various forms of malware. Consequently, substantial efforts have been directed towards Android malware detection, employing diverse techniques encompassing static, dynamic, or hybrid methodologies. Static approaches hinge on static information such as permissions or signatures, while dynamic methods rely on data collected during the execution of the app. Hybrid techniques constitute a fusion of both static and dynamic strategies [1, 2, 3].

In contemporary times, Machine Learning (ML), particularly Deep Learning (DL) methods, have gained widespread adoption in malware detection. Various algorithms and multiple datasets are readily accessible in the literature, yielding highly accurate prediction results. Nonetheless, the process of procuring new data and developing novel algorithms is time intensive. Furthermore, recent research has indicated that greater computational efficiency and accuracy can be achieved when distinct datasets and algorithms are fashioned for individual malware categories. However, it is important to underscore that the training of DL architectures to achieve optimality remains a time-consuming endeavour.

This paper introduces a novel approach, leveraging Transfer Learning, rooted in 1-Dimensional deep Convolutional Neural Networks (CNN), to address the challenge of computational complexity in the detection of closely related Android malware categories, such as fake anti-malware, Trojans, malicious VPN clients, and others. The proposed methodology entails training a model on a dataset comprising malicious and benign antimalware apps [1], followed by the transfer of the model to analogous datasets containing malicious Trojans and benign apps [2], as well as datasets encompassing both malicious and benign VPN clients. The specifics of these datasets are indicated in the evaluation section.

The key contributions of this work are:

- To demonstrate how ChatGPT can be used to generate code for detecting malware in structured datasets with high accuracy. Although the algorithms themselves are not new, the focus is on enabling individuals without programming expertise to create and apply such models, while also helping experienced programmers save time by leveraging a large language model to generate code for these tasks.
- Comprehensive evaluation of the proposed methodology, substantiating its practicality and effectiveness, with results demonstrating a high level of detection accuracy in comparison to algorithms developed by humans.

The remainder of this paper is organised as follows: Section 2 presents an overview of related work, Section 3 outlines the problem statement, Section 4 describes the proposed methodology, Section 5 details the experimental evaluation process and reports the results, and Section 6 concludes the paper and discusses future research directions.

2. **RELATED WORK**

2.1. Static

Static solutions are primarily geared towards the detection of malicious applications in Android without necessitating their execution on mobile devices. These solutions make use of static features for the purpose of malware detection. In static analysis, the features derived from the APK file are gathered prior to its deployment on the device. This approach to Android malware detection places a strong emphasis on resource conservation, as it circumvents the necessity of installing the application on the user's device. Several pertinent works based on static analysis are elucidated below.

DeepDetect [4] is a machine learning-based model that operates on static features, facilitating on-device malware detection. DeepDetect exhibits effectiveness when paired with adept feature engineering, rendering it suitable for deployment on mobile devices. Another research endeavour postulates a Bayesian classification-driven system for the detection of Android malware, grounded in permission features and employing static analysis for the extraction of these permission features [5]. This is motivated by the pursuit of gauging the efficacy of static analysis in Android malware detection by focusing on permission-based attributes. The research posits the application of machine learning in conjunction with diverse sets of classifiers to assess Android malware detection. In this context, the feature selection method is adopted to ascertain which attributes are most proficient in distinguishing malicious software [6]. The efficacy of supervised machine-learning algorithms with static analysis data, drawn from the Drebin dataset, has been explored [48] and a brief overview of related studies in this domain was presented in [7].

Another study [8] proposes a novel Android malware detection system predicated on filter-based feature selection techniques. The methodology is rooted in machine learning and is based on static attributes extracted from application files, specifically permissions. To enhance the efficiency and execution speed of machine learning algorithms, dimension reduction is carried out by employing eight distinct feature selection methods. Four of these methods have been previously integrated into Android malware detection systems, while the remaining four have been adapted from research in text classification. In a parallel vein, an alternate machine learning-based malware detection system is presented in [9], aimed at distinguishing Android malware from benign applications. The feature selection stage of this malware detection system endeavours to eliminate redundant attributes using a linear regression-based feature selection approach. Consequently, this process reduces the feature vector's dimension, minimises training time, and permits the classification model to be utilised in real-time malware detection systems. In this context, research in [10] applies and evaluates machine learning approaches that are founded on static features for the identification of malware in the Android OS. Correlation-based feature selection techniques are employed to train each classifier on the training set through hyperparameter tuning, followed by an evaluation of their performance on an unseen test set. Additionally, in [11], 'Information Gain' is employed to rank permissions and intents with the objective of identifying the optimal set of permissions and intents to achieving high accuracy in Android malware detection. The study introduces a novel algorithm that combines machine learning algorithms, including Random Forest, SVM, and Naive Bayes, to ascertain the most suitable set. Furthermore, a lightweight Android malware detection system is proposed in [12], leveraging machine learning techniques that rely on fewer static attributes to differentiate between malicious and benign applications. The research adopts a feature engineering approach to streamline feature dimensions, employing a multi-level feature reduction and elimination process to establish a lightweight detection model. Subsequently, the research crafts a machine learning-based detection system using the refined feature set, outperforming models founded on the original feature set.

Another study introduces an effective framework grounded in the fusion of static attributes and machine learning classifiers to identify malware applications. Three categories of static attributes are extracted, namely API calls, permissions, and intents. API calls are extracted from Classes.dex, permissions from AndroidManifest.xml, and intents from the same manifest file. These features are harnessed for the training and testing of application classification [13]. A separate research endeavour seeks to expand the repertoire of malware detection methodologies by unveiling a static-based classification approach for malware detection that relies on Android permissions and API calls. This approach is underpinned by three prominent Machine Learning algorithms, namely Support Vector Machines (SVM), K-nearest neighbours (KNN), and Naive Bayes (NB), with the aim of achieving robust malware detection rates and contributing to efforts and studies aimed at safeguarding mobile information development [14].

In this study, a model is devised, drawing from a combination of four static features: permissions, API calls, monitoring system events, and permission rates. The dataset encompasses 2,820 samples of both malware and benign applications. This research introduces a pioneering Recurrent Neural Network (RNN) architecture that surpasses traditional machine learning algorithms in the context of malware detection [15]. In a related work, an innovative approach for detecting malware in Android applications is introduced, making use of a Gated Recurrent Unit (GRU), a subtype of Recurrent Neural Network (RNN). The research extracts two static attributes from Android applications, namely Application Programming Interface (API) calls and Permissions [16]. Consequently, this research aspires to develop a contemporary, effective, and dependable malware detection system employing deep learning algorithms. The study evaluates RNN-based LSTM, BiLSTM, and GRU algorithms across 8,115 static attributes in the proposed system for malware detection [17].

In the domain of deep learning-based static detection, some researchers directly extract bytecode from Android APK files, converting it into a two-dimensional bytecode matrix. Subsequently, a detection model is trained and applied for malware classification, harnessing the deep learning algorithm, Convolutional Neural Network (CNN). CNN autonomously learns the characteristics of bytecode files, enabling the identification of malware [18]. Furthermore, as an alternative solution for malware detection founded on deep learning, a novel anti-malware system is proposed, utilising customised deep learning models that are sufficiently deep, characterised as 'End to End deep learning architectures for detecting and attributing Android malware via opcodes extracted from application bytecode' [19]. In yet another research effort, a method is suggested that employs static analysis in conjunction with the natural language processing (NLP) technique of document embeddings to generate feature vectors representing information within Android manifests and Dalvik executables contained in an APK. These embeddings are subsequently deployed to train binary classifiers capable of distinguishing between benign and malicious Android applications [20]. Lastly, this research introduces a static Android app analysis method grounded in an app similarity graph (ASG). In contrast to expert-based attributes, the study posits that the core of app behaviour classification resides in their shared, reusable building blocks, such as functions [21].

2.2. Dynamic

Static solutions do not execute apps; hence, such solutions may not detect apps that download malicious components at update time. Hence, dynamic solutions came into existence. In this approach, the features of an APK are collected by running them in a sandbox environment. This is a more resource-consuming approach than static malware detection. Summaries of some papers based on dynamic analysis are given below.

EnDroid [22] introduces a powerful dynamic analysis framework for implementing highly precise malware detection based on multiple types of dynamic behaviour features. These features cover system-level behaviour tracing as well as common application-level malicious behaviours such as data theft, premium service subscription, and malicious service communication. EnDroid also employs a feature selection algorithm to eliminate noisy or irrelevant features and extracts critical behaviour features via a runtime monitor and uses an ensemble learning algorithm to distinguish between malicious and benign applications. In another study, they use malware and the benign app it infects to test the effectiveness of mining sandboxes in detecting malicious behaviour. They create a sandbox based on sensitive APIs used by the benign app and test it to see if it can detect malicious behaviour in the corresponding malware [23]. They developed a system that detects the behaviours of Android applications and identifies known and unknown malware. By loading a kernel module, their system can monitor specific applications. Following the detection process, the associated documents are uploaded to the server, and the dynamic behaviours are rebuilt [24]. DroidCat [25] is a novel dynamic app classification technique to supplement existing approaches. DroidCat achieves superior robustness over static and dynamic approaches that rely on system calls by utilising a diverse set of dynamic features based on method calls and inter-component communication (ICC) Intents, but without involving permission, app resources, or system calls while fully handling reflection. The characteristics were derived from a study of benign versus malicious apps' behavioural characterisation.

De-LADY [26] (Deep Learning-based Android Malware Detection Using Dynamic Features) is proposed as an obfuscation-resistant approach. It makes use of behavioural characteristics derived from the dynamic analysis of an application running in an emulated environment. A similar paper proposes DL-Droid [27], a deep learning system for detecting malicious Android applications using dynamic analysis and stateful input generation. EntropLyzer [28] proposes an entropy-based behavioural analysis technique, as a technique for classifying the behaviour of 12 prominent Android malware categories and 147 malware families. To classify and characterise Android malware, this study employs six classes of dynamic characteristics: memory, API, network, logcat, battery, and process. PICAndro [29] uses packet inspection of captured network traffic to improve the accuracy and depth of malware detection and categorisation. The network interactions identified are represented as images that are fed into the CNN engine.

In another work, they use pseudo-label, a semi-supervised learning technique for deep neural networks that they train with a set of labelled and unlabelled instances. They employ dynamic analysis to create dynamic behaviour profiles in the form of feature vectors. They evaluate and compare their proposed model to Label Propagation (LP), a well-known semi-supervised machine learning technique, and other common machine learning algorithms [30]. Finally, the impact of all dynamic analysis categories and features on Android malware detection is examined using various filter and wrapper methods [31].

2.3. Hybrid

According to [32], the use of a single approach, whether dynamic or static, falls short in accurately classifying malware due to challenges posed by obfuscation and execution stalling. Consequently, researchers have started to embrace hybrid analysis techniques. This section provides an overview of hybrid malware analysis which revolves around the detection and classification of Android malware.

The MFF-AMD mechanism [33] aims to enhance the accuracy of Android malware detection using machine learning techniques. This system begins by extracting diverse features through a combination of static and dynamic analyses, resulting in a comprehensive multiscale feature set. The Relief algorithm is introduced to fuse these features, and four weight distribution algorithms are designed to merge base classifiers, thereby achieving superior classification performance. MFF-AMD also defines a threshold that facilitates the selection of either static or hybrid analysis for malware samples. Subsequently, AmandaSystem [34] presents a novel bottom-up static analysis methodology for the creation of PerApTool, an efficient and comprehensive tool dedicated to mapping relationships between Android permissions and API calls. Sttatic and dynamic analysis of Android malware is explored, comparing the outcomes of pattern identification in datasets and the utilisation of a range of classifiers to identify the most effective approach for malware analysis in [35].

In another study, benign and malware data from various sources are consolidated, resulting in an expanded dataset comprising 489 static and dynamic features. The primary outcome is a novel, labelled, and hybrid-featured Android dataset equipped with timestamps for each data sample, encompassing the entirety of Android history from 2008 to 2020, while taking into account distinct sources of dynamic data [36]. Another study introduces an effective image-based Android malware detection system, extracting six different features from Android applications using both static and dynamic analyses. These features include intent, opcode, and permission from static analysis, as well as unigram, bigram, and trigram from the system call log derived from dynamic analysis [37]. A comprehensive benchmarking exercise is conducted, comparing the detection performance of six distinct timestamping approaches for static and dynamic feature sets in [38].

'Chimera,' a novel multimodal deep learning (DL) Android malware detection approach that amalgamates both manual and automatic feature engineering is proposed, with the amalgamation leveraging DL architectures such as Convolutional Neural Networks (CNN), Deep Neural Networks (DNN), and Transformer Networks (TN) for feature learning from raw data (Dalvik Executable (DEX) grayscale images), static analysis data (Android Intents & Permissions), and dynamic analysis data (system call sequences), respectively [39]. In a different study, a novel Tree Augmented Naive Bayes (TAN)-based hybrid malware detection mechanism is suggested, capitalising on conditional dependencies between pertinent static and dynamic features, encompassing API calls, permissions, and system calls required for an application's functionality. This approach involves training three regularised logistic regression classifiers, each aligning with an application's API calls, permissions, and system calls. The output relationships of these classifiers are modelled using a TAN to determine the malignancy of the application [40]. Another paper introduces a hybrid analysis approach for detecting Android malware and categorising malware families, with partial optimisation for multi-feature data. This employs permissions and intent as static features in the context of static analysis. Dynamic analysis is focused on sessions, maintaining all protocol layers, and network traffic is harnessed. The Res7 LSTM model is subsequently utilised to further classify malicious and partially benign samples detected during static analysis [41]. Similarly, CoDroid [42] is a sequence-based hybrid Android malware detection approach that utilises static opcode and dynamic system call sequences. In a natural language processing (NLP) context, a sequence is treated as a sentence, and a CNN-BiLSTM-Attention classifier is constructed from Convolutional Neural Networks (CNNs) and Bidirectional Long Short-Term Memory (BiLSTM) with an attention language model.

Another recent paper [43] suggests a system for classifying Android applications that combines static permissions and dynamic packet analysis. The system gathers static information about Android applications through static analysis, employing machine learning to classify them as benign or malicious, while filtering out benign applications to minimise dynamic data collection time. The malware's network traffic is then employed to extract multiple types of features in the dynamic analysis phase, with machine learning facilitating malware family classification. Moreover, this paper employs a hybrid approach to malware detection based on static, dynamic, and intrinsic features, utilising k-nearest neighbours (k-NN) and logistic regression machine learning algorithms. The intrinsic feature contribution is also evaluated, and a linear discriminant analysis technique is deployed to assess its impact on the detection rate [44]. Furthermore, the authors propose a malware detection algorithm for Android that relies on a hybrid deep learning model, combining a deep belief network (DBN) and a gate recurrent unit (GRU). This research begins by examining Android malware, extracting both static and dynamic behavioural features that possess robust anti-obfuscation capabilities. Subsequently, a hybrid deep-learning model for Android malware detection is created [45]. Two datasets for binary and multiclass (family) classification are generated, harnessing a robust set of features extracted from static and dynamic malware analysis. And various machine-learning algorithms deployed to detect and classify malware using the features extracted from static and dynamic malware analysis in [46]. Yet, another study proposes an efficient and accurate machine learning and deep learning model to tackle this challenge. For static analysis, the researchers draw upon the malware genome dataset and the Drebin project [47], while the CICMalDroid2020 dataset [48] serves as the source of dynamic analysis data. Hybrid analysis is then performed, combining features extracted from these two datasets [49].

In the literature and in the current paper, it is shown that code generated from ChatGPT doesn't always work. One publication [52] looking at correctness of synthetic code found several weaknesses and limited evaluation power of the original test inputs from HumanEval [53]. They stated that they have found a way to improve on several inconsistencies, highlighting the improvements' ability to identify significant amounts of previously undetected code errors. Other publications [54, 55] find that ChatGPT has several flaws and security issues with the code it generates. Of 21 reported use-cases only 5 appeared initially secure, with a further 7 made more secure when explicitly told to do so, and how, by the user. Other publications [56] also found that the synthetic code generated, appeared to be vulnerable in more than a third of their use-cases with another at only around 12% [57].

The aim of this paper is to show that ChatGPT can be utilized to generate code that can be applied on a structured dataset to detect malware with very high accuracy. While the algorithms are not novel, the goal is for people who do not have a programming background to be able to generate and apply such models and for experienced programmers to save time by asking a large language model to write code for such problems. In the context of malware detection, we have used ChatGPT to generate machine learning algorithms that can be applied to a dataset with features and the experiments do not involve any dynamically executed code in a controlled environment or otherwise.

3. PROBLEM STATEMENT

While LLM-based code generation provides promising advantages in accelerating productivity and automating certain tasks for business, the synthetic code generated by modern LLM's is facing some criticism. In this paper we are interested in systematically evaluating the code generated by ChatGPT [51] for correctness, compatibility and usability in malware detection using machine learning. The generated models from ChatGPT are not fundamentally different from traditional machine learning models. The key distinction here is not in the models themselves but in the process of how they were created. ChatGPT was used to automate the code generation process, which provides a new approach to developing these algorithms and novelty lies in the code generation process facilitated by ChatGPT.

The paper aims to demonstrate that while ChatGPT can automate the initial code generation, the resulting model still requires training and validation to function effectively. ChatGPT's role is limited to generating the initial code, with the actual model training being a subsequent and essential step. The process of using ChatGPT to generate code does differ from manually developing a model. While ChatGPT can quickly generate code, ensuring the accuracy and performance of this code still requires further steps, including validation, debugging, and optimization. This process can be more efficient than manual coding, but it also comes with its own challenges, such as verifying the correctness of the generated code.

4. PROPOSED METHODOLOGY

While the features and models used in this study are commonly known in the machine learning community, the innovative aspect of this research lies in the use of ChatGPT to automate the code generation process. This approach can significantly reduce the time and effort required to develop models, particularly in the context of Android malware detection. The advantage of using ChatGPT to generate model code lies in its ability to produce customized code tailored to specific datasets and needs. Unlike open-source libraries, which may require significant adaptation to fit specific requirements, ChatGPT can generate code that is more directly aligned with the task at hand. This approach can save time and reduce the likelihood of introducing errors during the customization process.

Moreover, because of recent publications showing that generated code using ChatGPT doesn't always work, we began to question how ChatGPT would fare in the domain of machine learning. We devised a series of prompts that would question ChatGPT's ability to discuss and build ML models and architectures. We proposed a total 63 prompts ranging in topic and complexity from simple queries and common tasks to more complex challenges, all within the domain of machine learning. These prompts can be found in Appendix A. From this we examined the code, ran it and evaluated its outputs, where applicable. To evaluate the effectiveness of ChatGPT's synthetic code, we created three distinct classifications to which all tasks in question could be classified, as discussed in Table 1.

Table 1: The classifications and descriptions for the code prompted from ChatGPT

Classification	Description
Green	All code under green classification compiles and runs as expected. To achieve this classification the code must not be altered, added too, or changed in anyway from how ChatGPT originally provided it
Yellow	All code under yellow classification satisfies at least one of the following: 1. Code does not compile or run without minor modification, addition or change to some degree from how ChatGPT originally provided it. 2. Code will compile or run but not without warnings or errors 3. Code will not compile or run due to deprecated code 4. Code will compile or run but uses deprecated code. Deprecated code: using an older version of an API, using a dataset that is unavailable at the time of testing or using deprecated names for parameters, settings and other such matters. Minor modifications include: no dataset (as a direct result of the prompt), code snippets that require more code to run due to not prompting for a whole model e.g. a snippet depicting a new evaluation metric, that isn't attached to a ML model, so a ML model must be provided.
Red	All code under red classification does not compile or run as expected.

	To achieve this classification instead of yellow classification means the code has either significant errors, such as calling undefined variables, GPT was unable to provide code when prompted, code doesn't run due to all errors that result in it not compiling e.g. calling an index out of range of an array, attempting to use an API which hasn't previously been called and defined, passing more or less parameters into functions than what is required etc.

To put this into layman's terms, Green means the code runs first time when copied and pasted into an IDE. Yellow means that there is some error with the code that may be a result of the period ChatGPT was trained and as such is using older libraries etc, or the prompt could have been more explicit in the case of not attaching the snippet to a larger model for context. Red means the code has some major problems that don't satisfy the yellow requirements.

Our findings from these prompts concluded that 58.26% were categorised in green, 22.41% in yellow and 18.97% in red. Thus, we may state that the code ChatGPT provided us only worked 58% of the time using the raw, unadulterated original code that it generated. For the purposes of this paper, we took these findings and decided that we should apply ChatGPT's synthetic code to an area we know it has displayed the ability to meet the green category and as such, compiles. Our last few prompts depict applications of classifiers to a malware dataset. You may find further discussion of the classifiers in Section 4.1. From this, we may test and critically evaluate the synthetic code provided by ChatGPT against our own unique modifications, as discussed in the following sections.

Datasets

In this study, we are using a balanced dataset called spy.csv. The dataset was built for the purpose of identifying spyware on the Android operating system. It is a subset of a pre-existing one, called CIC-AND-MAL 2020 [60, 61]. To create the dataset, we asked ChatGPT to write Python code to select random spyware and random benign apps to create a balanced dataset. This code worked as expected, and a new spy.csv dataset was developed. The dataset contains 10,021 spyware and 10,021 benign Android apps with 9,503 features.

Each feature represents a specific permission or characteristic of the app, given a binary value: 1 if the app requests the permission, and 0 if it does not. This binary representation allows for efficient processing and analysis of the dataset. A snapshot of the dataset's attributes is presented in Table 2. The balanced nature of the dataset ensures that machine learning models trained on it are not biased towards benign or malicious apps, thus providing a robust foundation for developing effective spyware detection mechanisms. The comprehensive feature set captures a wide range of app behaviours and permissions, crucial for accurate malware detection. However, it should be noted that the original dataset doesn't provide any details about the attributes, and we refer to them in Table 2 as characteristics.

Table 2: The dataset attributes

Heading	*Characteristic 1*	*Characteristic 2*	…	*label*
Value	Binary value	Binary value	Binary value	Binary value

4.1 Generated Classifiers

To begin with, we require both the synthetic models created by ChatGPT and the modified versions of these models that we have created. This section will cover which models are used, how we acquired them, and how they were built. For convenience, we restrict to use of the following models: CNN, MLP, Decision Tree, Random Forest, KNN and SVM. We chose to limit our selection of algorithms due to time constraints on the project. As such, a handful of random but common algorithms were chosen. To acquire our models from ChatGPT we formatted a prompt which, by design, would only have one discernible difference: the name of the model. A template of the prompt is as follows:

> *Implement a <model name> model for malware classification. The dataset is called Spy.csv and it has 9503 columns in total where the last one is the label. Skip the first row.*

After obtaining the code that we require from ChatGPT, there were some minor modifications required. Firstly, in the process of reading the dataset, we removed the variable that held the file path and altered the code to be cleaner, as shown in Figure 1.

```
# Load the dataset
df = pd.read_csv('Spy.csv', skiprows=[0])
```

Figure 1: A cleaner reading of the dataset, having removed the unnecessary variable holding the file paths value.

Secondly, we altered the scoring metrics and visual output to better represent the models' performance, for the purpose of comparing the findings against our own modifications. Ergo for the purpose of a fair representation of ChatGPT's generated models, none of these modifications altered the models' performance, hyperparameters or settings of any kind. The following sections will now be split into two parts. We will first discuss in detail, the models ChatGPT generated. We will then discuss our own modified models in detail, in instances where our models differ from that of the ChatGPT generated models only, to reduce repetition.

4.1.1 Convolutional Neural Network

Previous works [1, 2, 3], including that discussed in Sections 1 and 2, have demonstrated that machine learning models, specifically neural networks, work extremely well for our problem task. In this paper, we only explore the use of these methods in a supervised learning classification context. The first model we will look at is the Convolutional Neural Network (CNN). The architecture of the ChatGPT generated CNN is shown in Figure 2.

CNNs are like a standard neural network, consisting of neurons that have learnable weights and biases. Every neuron in the network receives some input to which it performs a dot product and can optionally follow that up with a non-linearity. Likewise, they still encompass a score function, loss function and many other similar attributes that one may associate with neural networks. The main difference is that the CNN architecture can make the presumption that the inputs are images. This is very explicit and allows for specific properties we may want to be encoded into said architectures. The by-product of this, is that it allows for a greater efficiency in the forward function, and so fewer parameters are required. There are a few main components required in the building of this model. These are: the convolutional layer, the pooling layer and the fully connected layer. We will begin with the first layer, the convolutional layer.

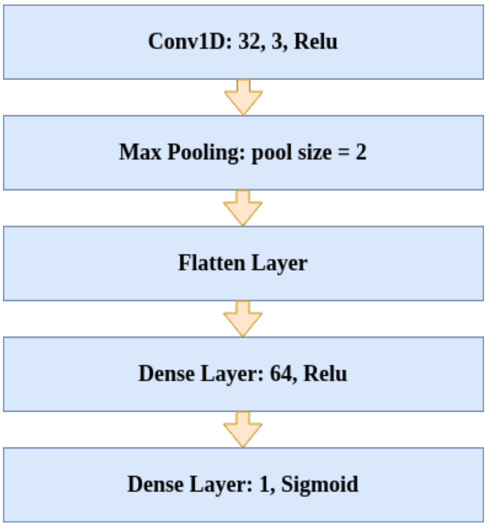


Figure 2: The CNN model ChatGPT generated for us using the prompt format discussed.

The convolutional layer is the key building block of any CNN. This layer performs a dot product between two matrices (*W* and *X*) as shown in Equation (1); one of these matrices refers to the restricted part of the receptive field, while the other is the kernel which is a set of learnable parameters. We can calculate the spatial size of the volume of the output (W_{out}) as a function. If we declare the input volume size as W, the convolution layers' neurons' receptive field size as F, the stride as S, and the degree of padding as P, then we may calculate the number of neurons that will fit, as seen in Equation (2).

$$W \cdot X = w_1x_1 + w_2x_2 + \ldots + w_nx_n = \sum_{i=1}^{n} w_ix_i \tag{1}$$

$$W_{out} = \frac{W - F + 2P}{S} + 1 \tag{2}$$

Looking more closely at the convolutional layer, let the output be denoted by y, the length of the input to the convolutional layer of n be denoted by x, the previously mentioned kernel k and the number of strides by s, which may succeed each convolution. We obtain the formulation in Equation (3).

$$y(n) = \begin{cases} \sum_{i=0}^{k} x(n+i)h(i), if no = 0 \\ \sum_{i=0}^{k} x\big(n+i+(s-1)\big)h(i), otherwise \end{cases} \tag{3}$$

Following this, the convolutional layer uses an activation function, which is taken to be the rectified linear unit (ReLU), as shown in Figure 3. The purpose of this is to introduce non-linearities to the CNN, as we see in the formulation in Equation (4). ChatGPT chose to use ReLU as opposed to another metric. The reasoning for this is unknown.

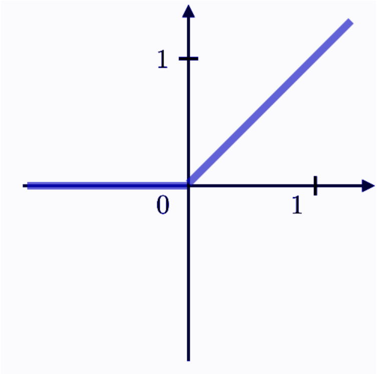


Figure 3: The ReLU activation function

$$y(x) = max(0, x) \tag{4}$$

Next, we have the pooling layer. In our GPT generated CNN, it provided a size of two. Pooling is required typically in the aid of overfitting. It consists of reducing the dimensionality of the mapping while giving prominence to certain features. This is typically applied to the output of the convolutional layer. In this instance max pooling is used, wherein the max value is selected of a window size W, which is slid over the input using stride size S after each pooling execution. After this, we may find that the output has the predisposition attributed to the layer's depth, of being greater than one. The subsequent flatten layer rectifies this, concatenating the output into a flat-like structure, which can then be distributed as input into a multi-layer perceptron (MLP). This is indicated in Equation (5), where F_{out} is the function output, the function name f followed by with the function inputs (x and w), biases b, and while the Greek letter sigma represents the sum of the inputs.

$$F_{out} = f(x_n, w_{mn}) = b + \sum_{m} x_n w_{mn} \tag{5}$$

Now considering we are relating to a classification problem, there are numerous ways in which we may express the output. In the instance of this model, it is via one-hot encoding. This signifies that each element of the output vector may only obtain a value of one or zero. This is conjoined with the sigmoid function. The sigmoid function, as shown in Equation (6), is a non-linear operation that ensures the real-valued output is in the range between 0 and 1.

$$\sigma(net) = \frac{1}{1 + e^{-net}} \tag{6}$$

4.1.2 Multilayer Perceptron

The second model we consider is the Multilayer Perceptron (MLP) model. The architecture of the GPT generated MLP is shown in Figure 4. MLP is a supervised learning algorithm often used throughout machine learning and has three types of layers: the input and output layers, and the hidden layers.

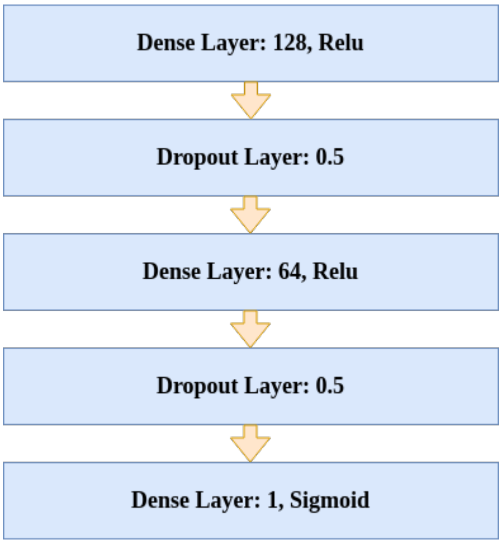


Figure 4: The ChatGPT generated MLP model architecture

MLP's consist of neurons (perceptron's) that have learnable weights and biases. Every neuron in the network receives some input to which it performs a dot product and can optionally follow that up with a non-linearity. They encompass a score function, loss function and all the common attributes you may associate with other neural network architectures. We consider the first layer of the GPT generated MLP, the input layer. An example of what this may look like is shown in Figure 5.

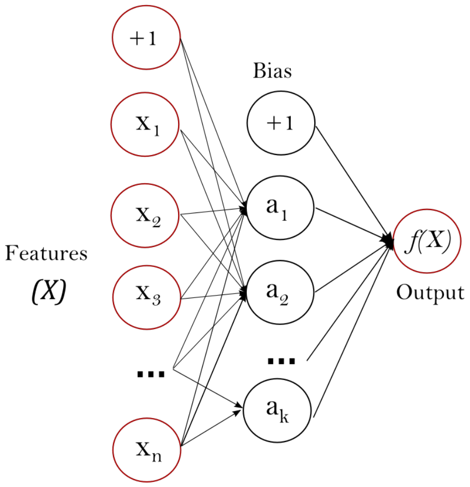


Figure 5: A One Hidden Layer MLP model [58]

The input layer consists of a predetermined number of neurons that represent the input features. As we move from layer to layer, each neuron transforms the values provided by the previous layer with a weighted linear summation, followed by an activation function, as shown in Equation (7), where w is the weight, and x is the feature.

$$w_1x_1 + w_2x_2 + \ldots + w_mx_m \tag{7}$$

The final layer is the output layer, which upon receiving the values from the previous hidden layer, transforms them into an output. In between our layers, we use dropout regularisation. Dropout is a simple technique for reducing overfitting. Dropout refers to randomly selecting neurons to be essentially ignored during the training phase. The effect of this is that the model becomes less sensitive to the individual weights of the neurons inside it, thus increasing its generalisation ability.

4.1.3 Decision Tree

The third model we consider is the Decision Tree (DT) model. We can see the architecture of the GPT generated DT as having no parameters or settings defined, other than the random state being set at 42. Decision trees are a supervised learning method quite common in ML classification problems. They can be seen as a decision support tool that has a tree-like structure, as shown in Figure 6. This informs decisions and their possible consequences. They work by making these decisions using something called entropy, as seen in Equation (8), where S is a subset of the training, +p the probability of positive -p and negative classes, respectively.

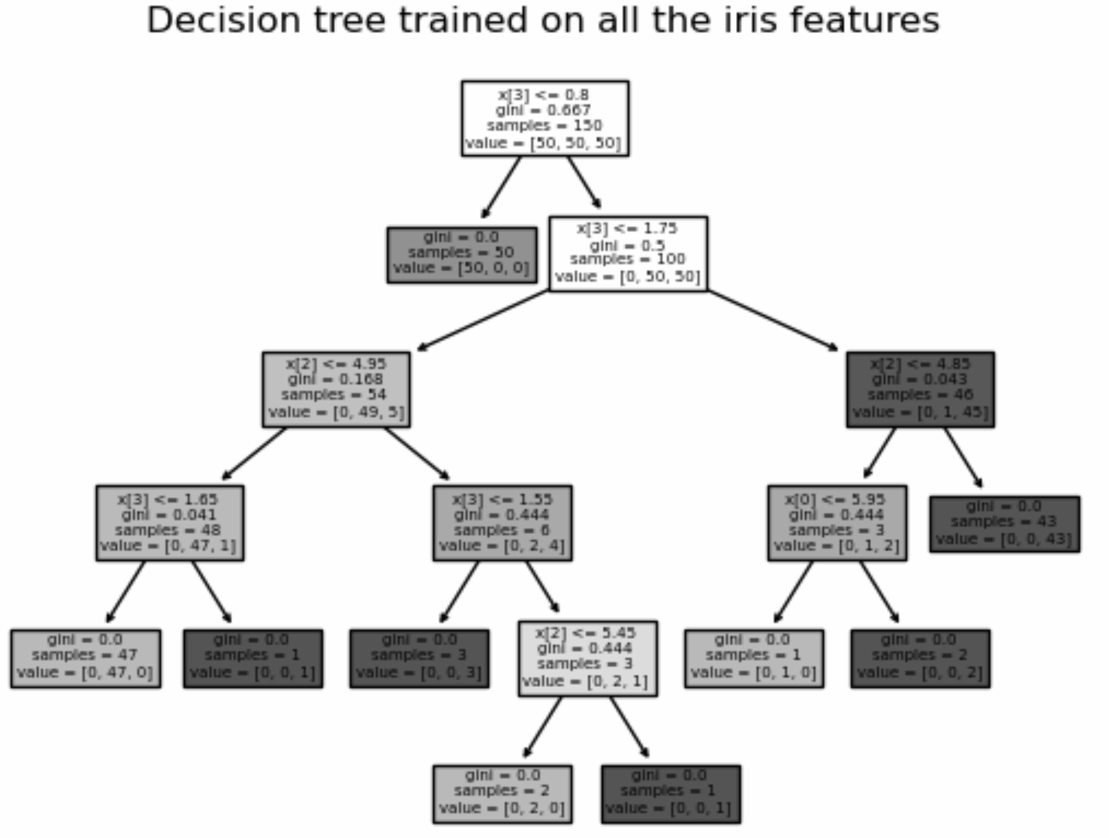


Figure 6: A decision tree trained on the iris dataset [59]

$$E(S) = -p + log2(p+) - p - log2(p-) \tag{8}$$

Entropy allows the impurity of a node to be measured, where impurity refers to the randomness of the data provided. Therefore, in the building of the tree, we may measure the impurity of a given node in the tree, and we can choose those at 100% impurity to be the leaf nodes. We can also consider information gain (IG), when considering what node plays what role. IG is a measurement on the reduction of uncertainty, provided by some features. In the context of deciding whether a node shall be assigned to be a standard node or a root node, we can calculate its IG, as shown in Equation (9), where E is the entropy, A is the full dataset and X is the feature.

$$IG = E(A) - E(A \vee X) \tag{9}$$

4.1.4 Random Forest

The fourth model we consider is the Random Forest (RF) model. The architecture of the GPT generated RF is akin to the decision tree, except for the number of estimators parameters is set to 100.

A RF can be seen as an ensemble of decision trees. It is a meta-estimator that fits a predetermined number of trees over varying samples of a given dataset. The benefit of this over a single DT is that it implements a control of averaging over the trees, thus reducing overfitting and improving accuracy. A key difference is the handling of feature selection. Whereas a DT performs feature selection within a single tree, an RF will average this out across all the DTs within the forest. Feature selection is based on feature importance, where we consider a set of n features $f_1, f_2 \dots f_n$ with a Gini index GI and we denote the importance of the variable by Vim. Then we can state the average change in the impurity of node splitting of the f feature across all decisions is $Vim_f^{(Gini)}$, where the formula for the Gini index GI is shown in Equation (10), with c being the number of categories, and p the proportion of c categories in n nodes.

$$GI_n = \sum_{c=1}^{c\vee\sum_{c'\neq c} p_{nc}p_{nc'}=1-\sum_{c=1}^{c\vee p_{nc}}{}^{2}} \tag{10}$$

3.1.5 K-Nearest Neighbour

The next model we consider is the K-Nearest Neighbour (KNN) model. We can find the architecture of the GPT generated KNN as having no parameters or settings defined, other than setting the number of neighbours to be 3. KNN is a simplistic model which is common in machine learning. Essentially it builds on the idea that data points that share some similarity will have similar data values or labels. It works by initialising K in a given number of neighbours, then for each example in the data it will calculate the distance between the queried example, and the current example at hand. This can be seen in Equations (11) and (12), which indicate the Euclidean and Manhattan distances respectively. Next, the distance and index are recorded in a collection which is then sorted into ascending order, using the distance value. The first k entries are then selected, and the labels are returned. The most common class among these labels can be seen as the predicted label.

$$\sqrt{\sum_{i=1}^{k}(x_i - y_i)^2} \tag{11}$$

$$\sqrt{\sum_{i=1}^{k} x_i - y_i \vee} \tag{12}$$

4.1.6 Support Vector Machine

The next model we consider is the Support Vector Machine (SVM) model. The architecture of the GPT generated SVM has no parameters or settings defined, other than setting the kernel type to 'linear'. SVM is a simplistic model quite common in machine learning. Essentially it builds on the idea that data points can be separated by a hyperplane. It works by initialising the kernel function, and then for each example in the data, it will find the optimal hyperplane that maximises the margin between different classes. This can be seen in Equations (13) and (14), which depict the linear and polynomial kernels respectively. Next, the support vectors are recorded in a collection which is used to form the decision boundary. The classification decision for any queried example is based on which side of the hyperplane it falls. The side determines the predicted label.

$$f(x) = w \cdot x + b \tag{13}$$

$$K(x_i, x_j) = \left(x_i \cdot x_j + c\right)^d \tag{14}$$

4.2 Phase 1 Modifications

Our first set of changes to the models GPT generated were minor modifications to bring the models up to a baseline that would set the foundations for more substantial changes later. Namely, all models underwent the following changes, as shown in Table 3.

Table 3: The phase one modifications with accompanying description

Modification	Description

Reading the dataset	All models had the format of reading the dataset altered, to properly reflect the parameters of the dataset for clarity. Please see code snippets below:
	Original:
	`x = df.iloc[:, :-1].values`
	`y = df.iloc[:, -1].values`
	Phase 1:
	`x = dataset[:, 0:9503]`
	`y = dataset[:, 9503]`

Cross-field validation	All models, where applicable, had cross-field validation applied. This took different formats from scikit-learn's cross-val function to its stratified k-fold respectively. For the purposes of this project, all models were initialised to 10-folds.
Random State	ChatGPT states that a random state value of 42 is common practice in machine learning for the purposes of reproducibility. For the purposes of this stage of modifications this parameter was removed entirely, so all models revert to their respective defaults for this parameter. The purposes of this exclusion were that it was seemingly a pointless addition at this stage of testing. Considerations may be made in Phase 2, as to whether this shall be initialised in our experimental evaluation.

The purpose of Phase 1 is to implement simple changes to alter the original models into a cleaner and more appropriate state for further customisations. Because of these minor alterations, we could observe any changes in the original models' performances without making sizable changes to parameters or model structures. Phase 2, in which we explore the best model architectures and parameters is discussed in the next sub-section.

4.3 Phase 2 Modifications

The second phase of our modifications looked at more substantial changes to each of the models. All models were tuned and modified using scikit-Learn's grid search and the Keras tuner respectively. All modifications were built on top of what was discussed in Phase 1. A description of the architectures and changes is discussed in the following sub-section.

4.3.1 Convolutional Neural Network

Using the prior discussed dataset, the CNN underwent some significant changes, notably its hyperparameters, as shown in Figure 7.

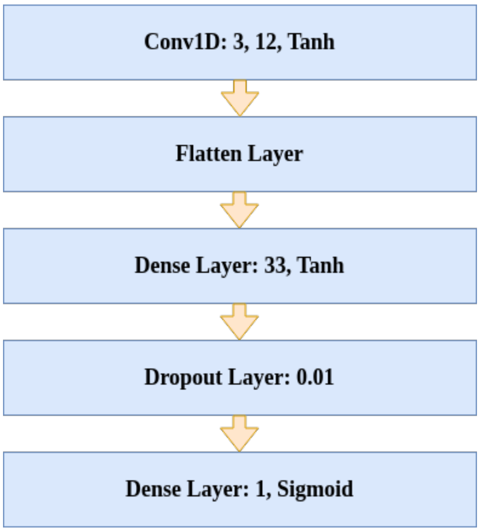


Figure 7: The Phase 2 CNN model architecture

As we can see, the filter, kernel and activation function, taken to be the tanh function shown in Equation (15), have all changed in the convolutional layer and respectively for the other layer's parameters. Other key differences include the use of bias in the convolutional layer that was set to false, and the learning rate altered to 0.006784, all of which were found to be most optimal using the Keras tuner. The epochs were also tuned the same way against the learning rate to determine after how many epochs the model would become saturated. This is shown in Figure 8.

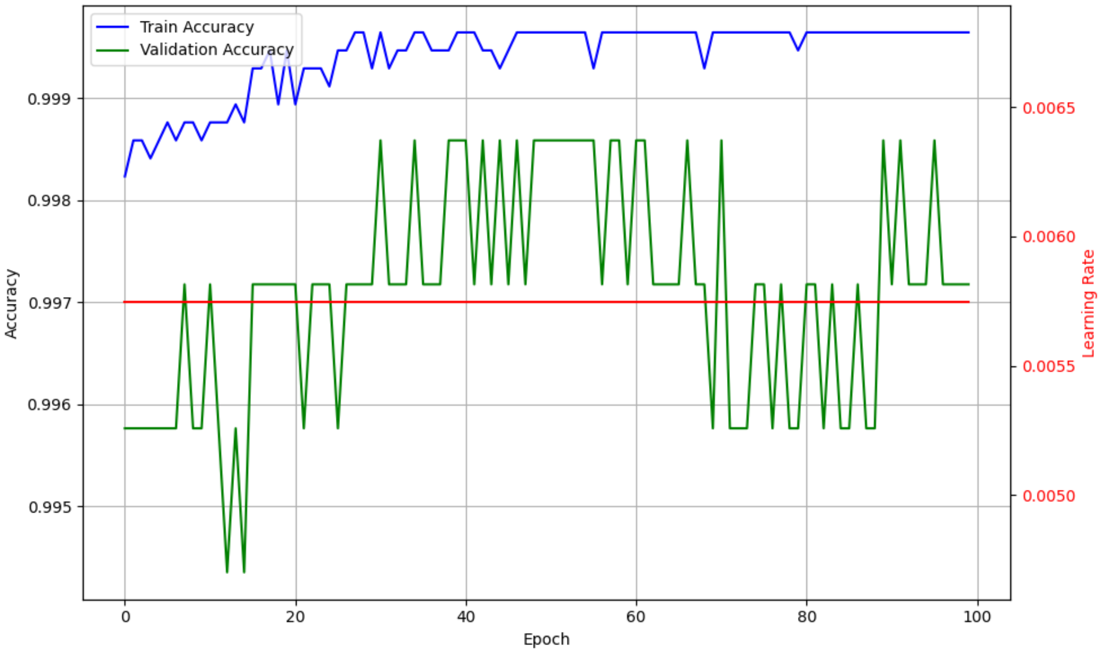


Figure 8: Epochs vs Learning Rate for the CNN model

$$f(x) = tanh(x) = \frac{2}{1 + e^{-2x}} - 1 \tag{15}$$

4.3.2 Multilayer Perceptron

The MLP model also underwent modifications as shown in Figure 9.

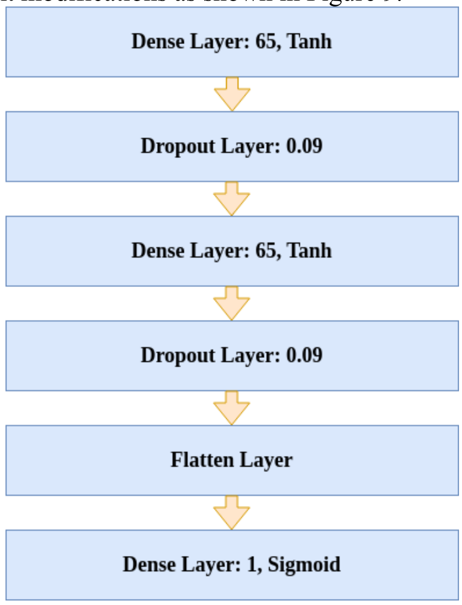


Figure 9: The Phase 2 MLP model architecture

Notable changes included the addition of a flatten layer amongst the various parameters, and the setting of the learning rate to 0.00080391, which was found to be most optimal using the Keras tuner.

4.3.3 Decision Tree and Random Forest

The DT and RF models underwent minor modifications. Due to our experimental evaluation, there was little room for improvement in the contexts of accuracy, as further discussed in Section 5. However, grid search enables us to fill in some of the parameters, notably the criterion set as Gini and the max depth of 4 for the DT and criterion set as entropy, max depth of nineteen and the number of estimators as 15 for the

RF.

4.3.5 K-Nearest Neighbour

The k-nearest neighbour classifier also sat in a similar situation as the DT and RF classifiers. Using grid search, the following parameters were obtained: number of neighbours was set to 1, and the weights assigned as uniform and the metric as Manhattan.

4.3.6 Support Vector Machine

The SVM classifier underwent minor modifications. Due to our experimental evaluation, there was little room for improvement in the contexts of accuracy, as further discussed in Section 5.

5. EXPERIMENTAL EVALUATION

In our experimentation we used the python programming language exclusively, along with the machine learning libraries Keras, Keras tuner, AutoKeras and Scikit-learn. All experiments were executed on an Intel® Core™ i7-9750H CPU @ 2.60GHz × 12 using 16GB of DDR4 memory on the Linux Ubuntu 22.04.3 LTS OS. Reruns and some tests were performed on an 13th Gen Intel® Core™ i9 24 Core Processor 13900HX (5.4GHz Turbo) using 32GB Corsair 4800MHz SODIMM DDR5 memory as well as CUDA GPU acceleration using NVIDIA® GeForce® RTX 4060 - 8.0GB GDDR6 Video RAM - DirectX® 12.1 on the Linux Ubuntu 22.04.4 LTS OS. As previously discussed, all experiments used 10-fold cross validation, where validation referred to a binary type of classification. The metrics used to critically evaluate our models were accuracy, precision, recall and F1-Score, shown in Equations (16) - (19), respectively. The contexts of the equations all relate to true positive (TP), true negative (TN), false positive (FP) and false negative (FN).

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN} \tag{16}$$

$$Precision = \frac{TP}{TP + FP} \tag{17}$$

$$Recall = \frac{TP}{TP + FN} \tag{18}$$

$$F1Score = \frac{2 * Precision * Recall}{Precision + Recall} \tag{19}$$

We conducted ten tests to evaluate the performance of our models before implementing any changes. Running multiple tests is crucial in ensuring the reliability of the results. It helps to account for variability and anomalies inherent in the data and model training processes. By averaging the outcomes across several tests, we can obtain a more accurate and robust measure of the model's performance, which reduces the impact of outliers and provides a solid baseline for comparison when assessing the effectiveness of subsequent modifications. We determined that ten tests would be sufficient for our test environment. This decision balances the need for reliable performance evaluation with practical considerations, such as time constraints. The rest of our evaluation will be split into three sub-sections to explore each stage of our methodology's findings.

5.1 The Original ChatGPT Generated Models

Following the generation of our original models generated by GPT, we ran ten tests per metric resulting in 40 results observed per model. The most common metric: accuracy, is shown in Table 4. Looking at the table, we can see that the DT, RF and SVM classifiers performed exceptionally well in being able to classify android permission-based malware, undoubtedly due to the cleanliness of the given dataset. Meanwhile the other classifiers also scored highly with zero, or close to zero, standard deviation.

Table 4: ChatGPT generated model results using the accuracy metric (Score%/Standard Deviation)

%	CNN	MLP	Decision Tree	Random Forest	KNN	SVM

Run 1	99 / -	99 / -	100 / .00	100 / .00	98 / .00	100 / .00
Run 2	99 / -	99 / -	100 / .00	100 / .00	98 / .00	100 / .00
Run 3	99 / -	99 / -	100 / .00	100 / .00	98 / .00	100 / .00
Run 4	99 / -	99 / -	100 / .00	100 / .00	98 / .00	100 / .00
Run 5	99 / -	99 / -	100 / .00	100 / .00	98 / .00	100 / .00
Run 6	99 / -	99 / -	100 / .00	100 / .00	98 / .00	100 / .00
Run 7	99 / -	99 / -	100 / .00	100 / .00	98 / .00	100 / .00
Run 8	99 / -	99 / -	100 / .00	100 / .00	98 / .00	100 / .00
Run 9	99 / -	99 / -	100 / .00	100 / .00	98 / .00	100 / .00
Run 10	99 / -	99 / -	100 / .00	100 / .00	98 / .00	100 / .00
Average	**99 / -**	**99 / -**	**100 / .00**	**100 / .00**	**98 / .00**	**100 / .00**

5.2 The initial modified Models

Next, we conducted minor changes, as discussed in Section 4. When looking at the accuracy metric, shown in Table 5, we see that the CNN and MLP models suffered a slight dip in their overall scores and/or variance in standard deviation respectively. Our findings showing that the ability to successfully classify android permission-based malware was largely unaffected when compared to the GPT generated models. This was also echoed in the KNN model to a smaller degree. We reiterate that the only changes were minor and didn't affect the models' architectures in any way.

Table 5: Phase 1 model results using the accuracy metric (Score%/Standard Deviation)

%	**CNN**	**MLP**	**Decision Tree**	**Random Forest**	**KNN**	**SVM**
Run 1	99 / .43	99 / .86	100 / .00	100 / .00	98 / .01	100 / .00
Run 2	98 / 1.14	99 / .51	100 / .00	100 / .00	98 / .01	100 / .00
Run 3	99 / .25	99 / .27	100 / .00	100 / .00	98 / .01	100 / .00
Run 4	98 / .49	99 / .83	100 / .00	100 / .00	98 / .01	100 / .00
Run 5	98 / .63	99 / .49	100 / .00	100 / .00	98 / .01	100 / .00
Run 6	98 / .82	99 / .15	100 / .00	100 / .00	98 / .01	100 / .00
Run 7	98 / .65	99 / .62	100 / .00	100 / .00	98 / .01	100 / .00
Run 8	99 / .75	98 / .91	100 / .00	100 / .00	98 / .01	100 / .00
Run 9	98 / .28	99 / .43	100 / .00	100 / .00	98 / .01	100 / .00
Run 10	98 / .52	99 / .28	100 / .00	100 / .00	98 / .01	100 / .00
Average	**98 / .60**	**99 / .54**	**100 / .00**	**100 / .00**	**98 / .01**	**100 / .00**

5.3 The Final Models

Finally, we present findings in relation to the Phase 2 modifications. When looking at the accuracy metric, as shown in Table 5, we can see an improvement in the CNN, MLP and KNN models, although it is minor.

One observation from these results, in contrast to the others, is that the standard deviation was reduced quite substantially in some cases. We can see from this that the ability to successfully classify android permission-based malware has improved.

Table 5: Our final model results using the accuracy metric (Score%/Standard Deviation)

%	**CNN**	**MLP**	**Decision Tree**	**Random Forest**	**KNN**	**SVM**
Run 1	99 / .00	99 / .11	100 / .00	100 / .00	99 / .00	100 /.00
Run 2	100 / .00	99 / .06	100 / .00	100 / .00	99 / .00	100 / .00
Run 3	99 / .08	99 / .10	100 / .00	100 / .00	99 / .00	100 /.00
Run 4	99 / .12	99 / .06	100 / .00	100 / .00	99 / .00	100 / .00
Run 5	99 / .08	99 / .09	100 / .00	100 / .00	99 / .00	100 /.00
Run 6	99 / .03	99 / .02	100 / .00	100 / .00	99 / .00	100 / .00
Run 7	99 / .14	99 / .11	100 / .00	100 / .00	99 / .00	100 /.00
Run 8	99 / .01	99 / .07	100 / .00	100 / .00	99 / .00	100 / .00
Run 9	99 / .10	99 / .13	100 / .00	100 / .00	99 / .00	100 / .00
Run 10	99 / .01	99 / .05	100 / .00	100 / .00	99 / .00	100 / .00
Average	**99 / .05**	**99 / .08**	**100 / .00**	**100 / .00**	**99 / .00**	**100 / .00**

5.4 Training/Testing Loss

Training loss in machine learning measures the model's error on the training dataset, reflecting how well the model is learning. It is calculated as the difference between the model's predictions and the actual values during training.

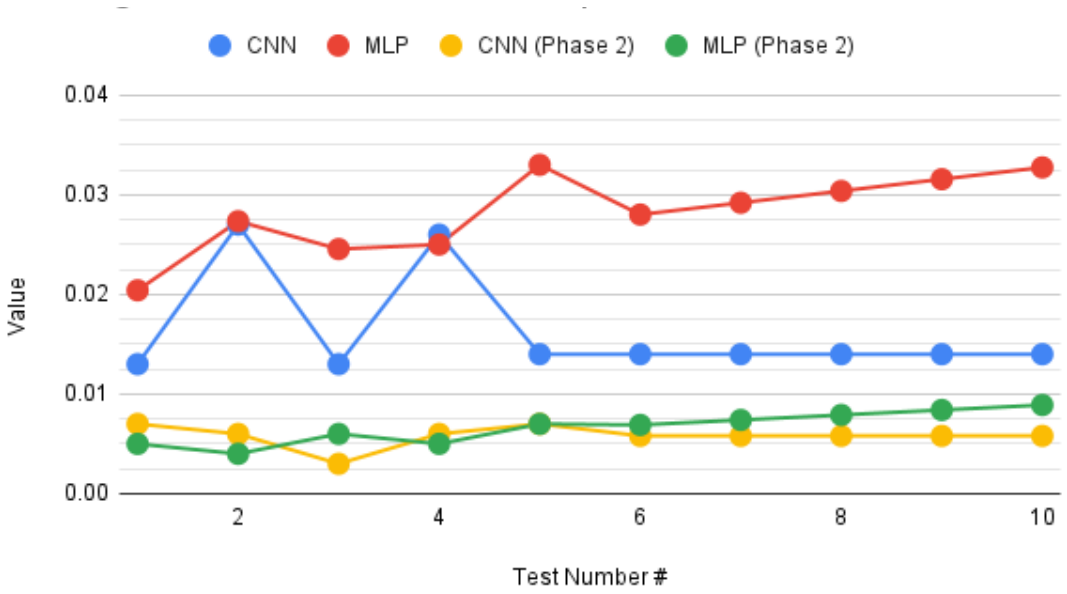


Figure 10: Training loss figures for each model per test

As we can see from Figure 10, the Phase 2 models performed at a more consistent rate with an overall lower trend in training loss than both GPT generated models. The GPT generated models themselves appear to have more variance and they appear to be more prone to anomalies.

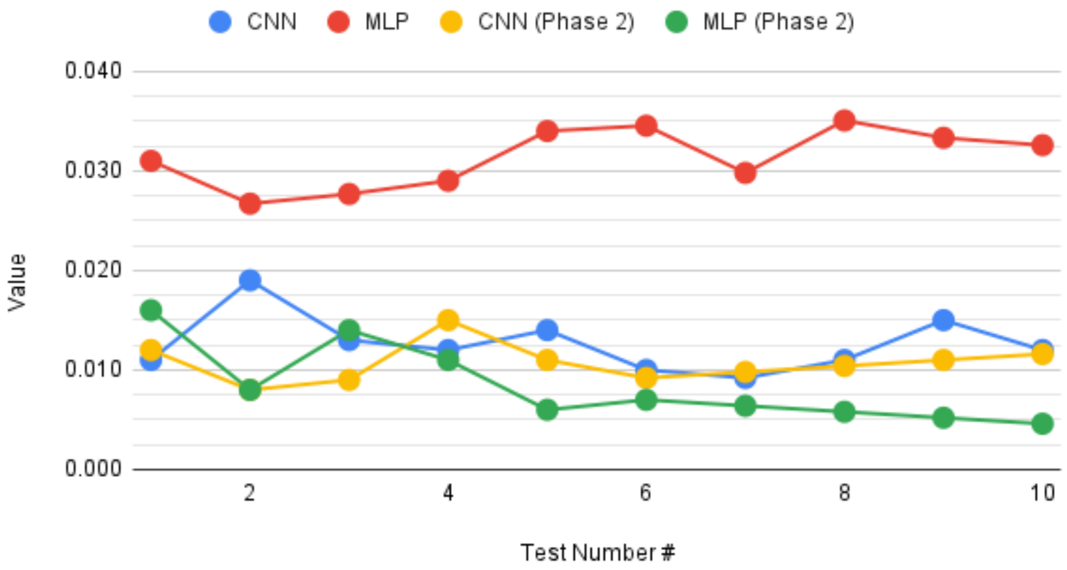


Figure 11: Testing loss figures for each model per test

This is similar to the testing loss values for each model, as shown in Figure 11. We can see that the MLP GPT generated model performed notably worse than the Phase 2 models. However, the GPT generated CNN performed at a similar rate, but it scored the highest overall value when only comparing it against the two custom models. As in training, the custom models performed relatively consistently when compared to the GPT generated versions.

5.5 Training/Testing Accuracy

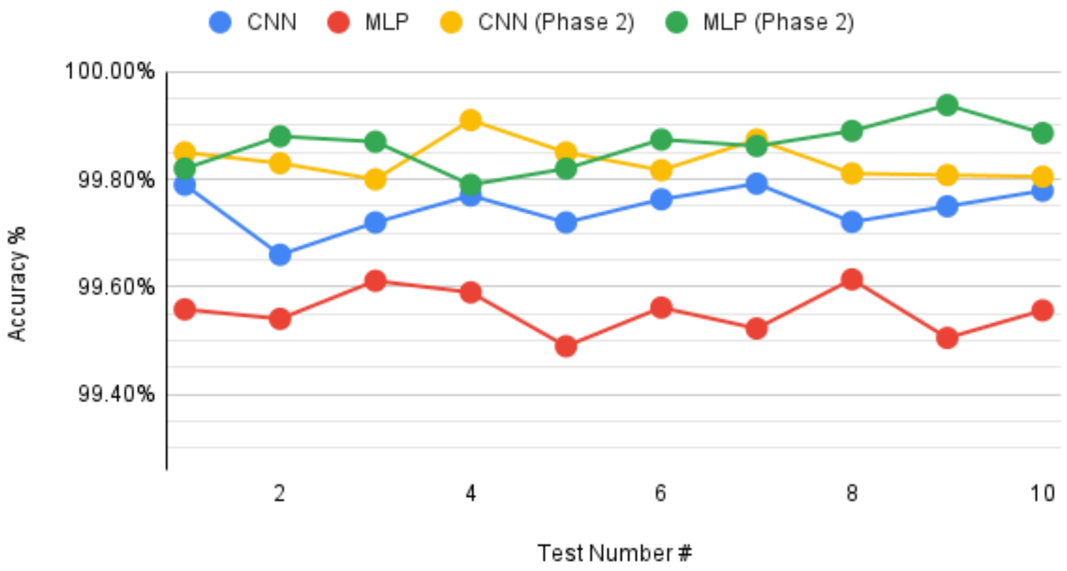


Figure 12: Training accuracy figures for each model per test

The models training accuracy are shown in Figure 12. We see that the two GPT generated models performed at a worse rate than the custom models, with MLP being the more significant. The custom models also appeared to perform at a more consistent level overall than the GPT generated versions, but it is notable that test 4 for the custom MLP and test 5 did perform below average, while test 4 for the custom CNN performed at an above average rate.

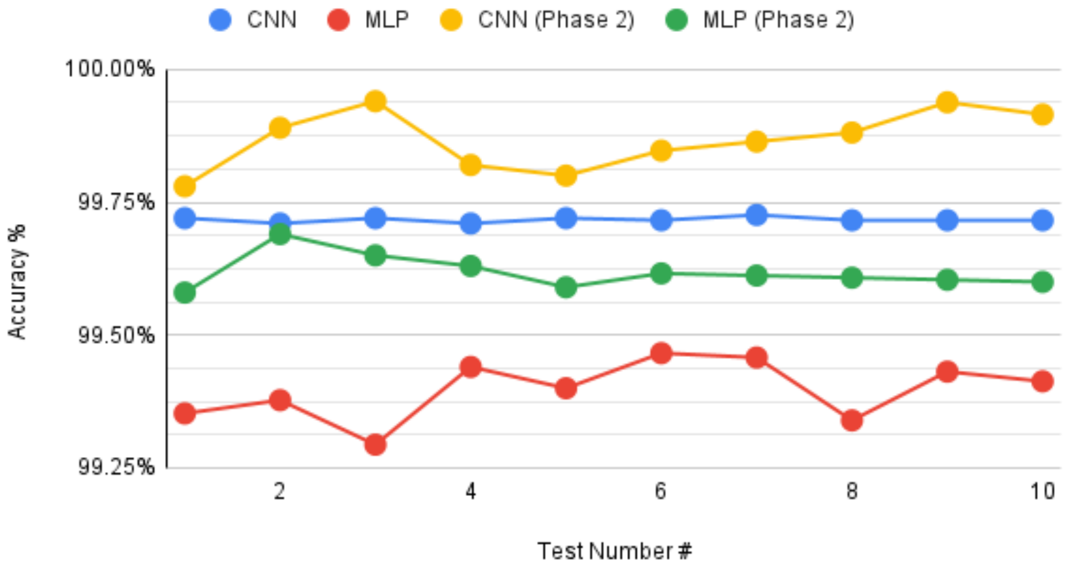


Figure 13: Testing accuracy figures for each model per test

The overall theme was also evident in the testing accuracy too, shown in Figure 13. The GPT generated MLP performed at a notably lower accuracy than the other models. A key difference was that in these tests, the GPT generated CNN performed consistently better than the custom MLP model. While they are different models, and the custom CNN outperformed the GPT generated CNN. This was the first instance of the GPT generated models consistently scoring better than any custom model.

6. DISCUSSION

Overall, we see that our findings made some improvements over the GPT generated models although they are very similar. When looking at our findings in Section 5 we noted that our modifications did have some advantages over that of the original models GPT generated. We observed that the original models, especially the Decision Tree (DT), Random Forest (RF), and Support Vector Machine (SVM) classifiers, performed at a very high standard due to the quality and cleanliness of our dataset. These models required no initial tuning, which is uncommon. As a result, any improvements from our modifications were minor.

Table 6: ChatGPT generated model results: overall averages for each metric (Score%/Standard Deviation)

%	CNN	MLP	Decision Tree	Random Forest	KNN	SVM
Accuracy	99 / -	99 / -	100 / .00	100 / .00	98 / .00	100 / .00
Precision	99 / -	99 / -	100 / .00	100 / .00	98 / .00	100 / .00
Recall	99 / -	98 / -	100 / .00	100 / .00	98 / .00	100 / .00
F1	98 / -	98 / -	100 / .00	100 / .00	98 / .00	100 / .00

Table 6 and Table 7 present the results of our evaluation. The original models exhibited high accuracy, precision, recall, and F1 scores, particularly for DT, RF, and SVM, which all achieved 100% across these metrics with minimal standard deviation. Our Phase 2 modifications yielded more reliable results, slightly improving the metrics' consistency.

Table 7: Our final model results: overall averages for each metric (Score%/Standard Deviation)

%	CNN	MLP	Decision Tree	Random Forest	KNN	SVM
Accuracy	99 / .05	99 / .08	100 / .00	100 / .00	99 / .00	100 / .00
Precision	99 / .06	99 / .09	100 / .00	100 / .00	99 / .00	100 / .00
Recall	99 / .17	99 / .03	100 / .00	100 / .00	99 / .00	100 / .00
F1	99 / .06	99 / .12	100 / .00	100 / .00	99 / .00	100 / .00

The literature emphasises the challenges and shortcomings of LLM-generated code, particularly in terms of security and correctness. Our findings, however, demonstrate that with a high-quality dataset, some generated code can achieve near-perfect performance in malware detection tasks. While prior studies identified significant vulnerabilities and correctness issues, our research indicates that these issues can be mitigated through careful dataset preparation and slight modifications to the generated code. The discrepancy between the literature and our findings could be attributed to the specific context of malware detection, where binary classification tasks on a well-constructed dataset may be inherently more straightforward than other coding tasks assessed in previous studies.

Our comparative analysis highlights a divergence between the general findings in the literature and our specific results in the domain of malware detection. While acknowledging the broader concerns about LLM-generated code, our study provides evidence that, under the right conditions, ChatGPT can produce highly accurate and reliable results using the model code it generates for detecting malware. This opens avenues for further research to test these findings across different datasets and more complex domains, potentially offering deeper insights into the strengths and limitations of LLM-based code generation.

6.1 Interpretation of the results

The results of this study indicate that ChatGPT can be effectively utilised to generate machine learning classifiers for Android malware detection. This has significant implications for other researchers, as it demonstrates a novel approach to leveraging advanced language models for cybersecurity applications. By automating the creation of classifiers, researchers can save substantial time and resources that would otherwise be spent on manual feature engineering and model training. This accessibility is particularly beneficial for researchers who may lack extensive expertise in machine learning or programming, thereby democratising the field and enabling a broader range of contributions. By integrating ChatGPT in the development of machine learning classifiers, this study contributes to the growing body of knowledge on the intersection of natural language processing (NLP) and cybersecurity. Other researchers can build upon these results to further refine the models, explore different datasets, or apply similar techniques to other areas of cybersecurity. This study also underscores the potential for interdisciplinary collaboration. By showcasing the application of an NLP model in a cybersecurity context, it encourages researchers from different fields to explore cross-disciplinary approaches. This can lead to innovative solutions that leverage the strengths of various domains, ultimately advancing the state of the art in both AI and cybersecurity research.

6.2 Practical implications

The practical implications of using ChatGPT to generate machine learning classifiers for Android malware detection are substantial. By leveraging the capabilities of an advanced language model, we can significantly streamline the process of developing and deploying robust malware detection systems. Below, we outline several key practical implications and indicate how we ensured the realism and representativeness of the classifiers generated. The classifiers generated by ChatGPT demonstrated high accuracy and reliability when tested on publicly available datasets. This indicates that they can be effectively used to detect and classify Android malware in real-world scenarios. By integrating these classifiers into existing security solutions, organizations can improve their malware detection capabilities, leading to better protection of user data and enhanced overall security.

Moreover, developing machine learning classifiers traditionally involves extensive feature engineering, model training, and validation processes. Using ChatGPT to automate these steps can significantly reduce the time and costs associated with developing new classifiers. This is particularly beneficial for small to medium-sized enterprises (SMEs) and research institutions with limited resources. It enables them to implement advanced malware detection systems without extensive investment, and the approach of using ChatGPT for classifier generation is highly adaptable and scalable. It also allows for the rapid creation of new classifiers as new malware variants emerge, ensuring that malware detection systems can keep pace with evolving threats. This adaptability is crucial for maintaining robust security in Android malware.

7. CONCLUSIONS

The experimental results showed that ChatGPT can generate fully functional machine learning models for malware detection with high accuracy, without requiring any modifications when these models are applied to a structured dataset. This suggests that ChatGPT could significantly reduce the time and effort needed to develop malware detection systems, thereby promoting greater AI adoption in the cybersecurity industry. The findings indicate that GPT can effectively create simple machine learning models that achieve high accuracy in detecting malware, provided the dataset is clean, balanced, and has sufficient features. However, there are instances where the generated code does not work, as noted in the appendix. Nonetheless, the code that does function can still serve as a valuable starting point for companies and individuals, allowing experts to refine and enhance it as needed.

Several inconsistencies have been highlighted in literature and in part, demonstrated in our findings. While LLM generated code can indeed be useful in many domains, it can suffer from code errors and poor programming choices. It is evident that while the models performed well, they do not perform as consistently, nor as accurately, as expertly crafted models. Thus, this leaves room for improvement on the part of LLM based code generation. In terms of generating machine learning code, an already niche sub-section of programming, one could suggest that LLM's should be trained on more ML code, and better code, to further the LLM's capability of what a good model looks like.

Another direction would be to cut out the training middleman entirely. Instead, using existing tools and technologies such as autoML to create models for users, as these are less prone to providing the user with a model that doesn't work due to a code error, or doesn't perform as well as a user may like, due to built-in hyper parameter tuning. Both directions diverge into typical methodologies in machine learning for improving on existing systems. Both have merit and raise some interesting research questions and issues. For instance, acquiring more training data on machine learning models may be more problematic than worthwhile, since questions of ethics and data protection arise. Also, an autoML solution would likely cause a reasonable increase in the time it takes for an LLM to generate a ML model, and so the usability of such an integration would be in question.

DATA AVAILABILITY STATEMENT

The datasets were derived from sources in the public domain: https://www.unb.ca/cic/datasets/index.html

CONFLICTS OF INTERESTS

All authors declared that there are no conflicts of interest.

REFERENCES

1. Seraj, S., Khodambashi, S., Pavlidis, M. and Polatidis, N., 2022. HamDroid: permission-based harmful android anti-malware detection using neural networks. *Neural Computing and Applications*, 34(18), pp.15165-15174.
2. Seraj, S., Pavlidis, M. and Polatidis, N., 2022. TrojanDroid: Android malware detection for Trojan discovery using convolutional neural networks. In: *International Conference on Engineering Applications of Neural Networks*. Springer, Cham, pp. 203-212.
3. Seraj, S., Khodambashi, S., Pavlidis, M. and Polatidis, N., 2023. MVDroid: an android malicious VPN detector using neural networks. *Neural Computing and Applications*, 35(29), pp.21555-21565.
4. Kumar, S., Mishra, D., Panda, B. and Shukla, S.K., 2021. DeepDetect: A practical on-device Android malware detector. In: *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, pp. 40-51.

5. Tuan Mat, S.R., Ab Razak, M.F., Mohmad Kahar, M.N., Arif, J.M. and Firdaus, A., 2022. A Bayesian probability model for Android malware detection. *ICT Express*, 8(3), pp.424-431.
6. Mohamad Arif, J., Ab Razak, M.F., Awang, S., Tuan Mat, S.R., Ismail, N.S.N. and Firdaus, A., 2021. A static analysis approach for Android permission-based malware detection systems. *PloS one*, 16(9), pp.1-23.
7. Syrris, V. and Geneiatakis, D., 2021. On machine learning effectiveness for malware detection in Android OS using static analysis data. *Journal of Information Security and Applications*, 59, p.102794.
8. Şahin, D.Ö., Kural, O.E., Akleylek, S. and Kılıç, E., 2023. A novel Android malware detection system: adaption of filter-based feature selection methods. *Journal of Ambient Intelligence and Humanized Computing*, pp.1-15.
9. Şahin, D.Ö., Kural, O.E., Akleylek, S. and Kılıç, E., 2023. A novel permission-based Android malware detection system using feature selection based on linear regression. *Neural Computing and Applications*, pp.1-16.
10. Islam, F.Z., Jamil, A. and Momen, S., 2021. Evaluation of machine learning methods for Android malware detection using static features. In: *2021 IEEE International Conference on Artificial Intelligence in Engineering and Technology (IICAIET)*. IEEE, pp. 1-6.
11. Khariwal, K., Singh, J. and Arora, A., 2020. IPDroid: Android malware detection using intents and permissions. In: *2020 Fourth World Conference on Smart Trends in Systems, Security and Sustainability (WorldS4)*. IEEE, pp. 197-202.
12. Jain, K. and Dave, M., 2021. Machine learning-based lightweight Android malware detection system with static features. In: *Evolving Technologies for Computing, Communication and Smart World*. Springer, Singapore, pp. 345-359.
13. Dhalaria, M. and Gandotra, E., 2020. A framework for detection of Android malware using static features. In: *2020 IEEE 17th India Council International Conference (INDICON)*. IEEE, pp. 1-7.
14. Shatnawi, A.S., Yassen, Q. and Yateem, A., 2022. An Android malware detection approach based on static feature analysis using machine learning algorithms. *Procedia Computer Science*, 201, pp.653-658.
15. Almahmoud, M., Alzu'bi, D. and Yaseen, Q., 2021. ReDroidDet: Android malware detection based on recurrent neural network. *Procedia Computer Science*, 184, pp.841-846.
16. Elayan, O.N. and Mustafa, A.M., 2021. Android malware detection using deep learning. *Procedia Computer Science*, 184, pp.847-852.
17. Bayazit, E.C., Sahingoz, O.K. and Dogan, B., 2022. A deep learning based Android malware detection system with static analysis. In: *2022 International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA)*. IEEE, pp. 1-6.
18. Ding, Y., Zhang, X., Hu, J. and Xu, W., 2023. Android malware detection method based on bytecode image. *Journal of Ambient Intelligence and Humanized Computing*, 14(5), pp.6401-6410.
19. Amin, M., Tanveer, T.A., Tehseen, M., Khan, M., Khan, F.A. and Anwar, S., 2020. Static malware detection and attribution in Android byte-code through an end-to-end deep system. *Future Generation Computer Systems*, 102, pp.112-126.
20. Raghav, U., Martinez-Marroquin, E. and Ma, W., 2021. Static analysis for Android malware detection with document vectors. In: *2021 International Conference on Data Mining Workshops (ICDMW)*. IEEE, pp. 805-812.
21. Frenklach, T., Cohen, D., Shabtai, A. and Puzis, R., 2021. Android malware detection via an app similarity graph. *Computers & Security*, 109, p.102386.
22. Feng, P., Ma, J., Sun, C., Xu, X. and Ma, Y., 2018. A novel dynamic Android malware detection system with ensemble learning. *IEEE Access*, 6, pp.30996-31011.
23. Bao, L., Le, T.-D.B. and Lo, D., 2018. Mining sandboxes: Are we there yet? In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, pp. 445-455.
24. Sun, S., Fu, X., Ruan, H., Du, X., Luo, B. and Guizani, M., 2018. Real-time behavior analysis and identification for Android application. *IEEE Access*, 6, pp.38041-38051.
25. Cai, H., Meng, N., Ryder, B. and Yao, D., 2018. Droidcat: Effective Android malware detection and categorization via app-level profiling. *IEEE Transactions on Information Forensics and Security*, 14(6), pp.1455-1470.
26. Sihag, V., Vardhan, M., Singh, P., Choudhary, G. and Son, S., 2021. De-LADY: Deep learning based Android malware detection using dynamic features. *J. Internet Serv. Inf. Secur.*, 11(2), pp.34-45.
27. Alzaylaee, M.K., Yerima, S.Y. and Sezer, S., 2020. DL-Droid: Deep learning based Android malware detection using real devices. *Computers & Security*, 89, p.101663.

28. Keyes, D.S., Li, B., Kaur, G., Lashkari, A.H., Gagnon, F. and Massicotte, F., 2021. EntropLyzer: Android malware classification and characterization using entropy analysis of dynamic characteristics. In: *2021 Reconciling Data Analytics, Automation, Privacy, and Security: A Big Data Challenge (RDAAPS)*. IEEE, pp. 1-12.
29. Sihag, V., Choudhary, G., Vardhan, M., Singh, P. and Seo, J.T., 2021. PICAndro: Packet inspection-based Android malware detection. *Security and Communication Networks*, 2021.
30. Mahdavifar, S., Abdul Kadir, A.F., Fatemi, R., Alhadidi, D. and Ghorbani, A.A., 2020. Dynamic Android malware category classification using semi-supervised deep learning. In: *2020 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*. IEEE, pp. 515-522.
31. Khalid, S. and Hussain, F.B., 2022. Evaluating dynamic analysis features for Android malware categorization. In: *2022 International Wireless Communications and Mobile Computing (IWCMC)*. IEEE, pp. 401-406.
32. Gandotra, E., Bansal, D. and Sofat, S., 2014. Malware analysis and classification: A survey. *Journal of Information Security*, 5, pp.56-64. doi: 10.4236/jis.2014.52006.
33. Xu, G., Feng, M., Jiao, L., Liu, J., Dai, H.-N., Wang, D., Panaousis, E. and Zheng, X., 2021. MFF-AMD: Multivariate feature fusion for Android malware detection. In: *International Conference on Collaborative Computing: Networking, Applications and Worksharing*. Springer, Cham, pp. 368-385.
34. Tang, J. and Zhao, H., 2022. AmandaSystem: A new framework for static and dynamic Android malware analysis. *Journal of Intelligent & Fuzzy Systems*, 43(5), pp.6575-6589.
35. Raghuraman, C., Suresh, S., Shivshankar, S. and Chapaneri, R., 2020. Static and dynamic malware analysis using machine learning. In: *First International Conference on Sustainable Technologies for Computational Intelligence*. Springer, Singapore, pp. 793-806.
36. Guerra-Manzanares, A., Bahsi, H. and Nõmm, S., 2021. KronoDroid: Time-based hybrid-featured dataset for effective Android malware detection and characterization. *Computers & Security*, 110, p.102399.
37. Rathore, H., Narasimhan, B.R., Sahay, S.K. and Sewak, M., 2022. Image-based Android malware detection models using static and dynamic features. In: *International Conference on Intelligent Systems Design and Applications*. Springer, Cham, pp. 1292-1305.
38. Guerra-Manzanares, A. and Bahsi, H., 2022. On the relativity of time: Implications and challenges of data drift on long-term effective Android malware detection. *Computers & Security*, 122, p.102835.
39. de Oliveira, A.S. and Sassi, R.J., 2020. Chimera: An Android malware detection method based on multimodal deep learning and hybrid analysis. *TechRxiv*.
40. Surendran, R., Thomas, T. and Emmanuel, S., 2020. A TAN based hybrid model for Android malware detection. *Journal of Information Security and Applications*, 54, p.102483.
41. Ding, C., Luktarhan, N., Lu, B. and Zhang, W., 2021. A hybrid analysis-based approach to Android malware family classification. *Entropy*, 23(8), p.1009.
42. Zhang, N., Xue, J., Ma, Y., Zhang, R., Liang, T. and Tan, Y.-a., 2021. Hybrid sequence-based Android malware detection using natural language processing. *International Journal of Intelligent Systems*, 36(10), pp.5770-5784.
43. Shyong, Y.-C., Jeng, T.-H. and Chen, Y.-M., 2020. Combining static permissions and dynamic packet analysis to improve Android malware detection. In: *2020 2nd International Conference on Computer Communication and the Internet (ICCCI)*. IEEE, pp. 75-81.
44. Mantoo, B.A. and Khurana, S.S., 2020. Static, dynamic and intrinsic features based Android malware detection using machine learning. In: *Proceedings of ICRIC 2019*. Springer, Cham, pp. 31-45.
45. Lu, T., Du, Y., Ouyang, L., Chen, Q. and Wang, X., 2020. Android malware detection based on a hybrid deep learning model. *Security and Communication Networks*, pp.1-11.
46. Dhalaria, M. and Gandotra, E., 2020. A hybrid approach for Android malware detection and family classification. Available at: http://ir.juit.ac.in:8080/jspui/jspui/handle/123456789/9331
47. Wei, F., Li, Y., Roy, S., Ou, X. and Zhou, W., 2017. Deep ground truth analysis of current Android malware. In: *International conference on detection of intrusions and malware, and vulnerability assessment*. Springer, Cham, pp. 252-276.
48. Mahdavifar, S., Abdul Kadir, A.F., Fatemi, R., Alhadidi, D. and Ghorbani, A.A., 2020. Dynamic Android malware category classification using semi-supervised deep learning. In: *2020 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*. IEEE, pp. 515-522.

49. Hadiprakoso, R.B., Kabetta, H. and Buana, I.K.S., 2020. Hybrid-based malware analysis for effective and efficiency Android malware detection. In: *2020 International Conference on Informatics, Multimedia, Cyber and Information System (ICIMCIS)*. IEEE, pp. 8-12.
50. Tian, H., Lu, W., Li, T.O., Tang, X., Cheung, S.C., Klein, J. and Bissyandé, T.F., 2023. Is ChatGPT the ultimate programming assistant--How far is it? *arXiv preprint arXiv:2304.11938*.
51. Chatgpt, 2023. OpenAI. Available at: https://chat.openai.com
52. Liu, J., Xia, C.S., Wang, Y. and Zhang, L., 2024. Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36.
53. Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H.P.D.O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G. and Ray, A., 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.
54. Khoury, R., Avila, A.R., Brunelle, J. and Camara, B.M., 2023. How secure is code generated by ChatGPT? In: *2023 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. IEEE, pp. 2445-2451.
55. Kocoń, J., Cichecki, I., Kaszyca, O., Kochanek, M., Szydło, D., Baran, J., ... & Kazienko, P., 2023. ChatGPT: Jack of all trades, master of none. *Information Fusion*, 101861.
56. Liu, Z., Tang, Y., Luo, X., Zhou, Y. and Zhang, L.F., 2023. No need to lift a finger anymore? Assessing the quality of code generation by ChatGPT. *arXiv preprint arXiv:2308.04838*.
57. Yetiştiren, B., Özsoy, I., Ayerdem, M. and Tüzün, E., 2023. Evaluating the code quality of AI-assisted code generation tools: An empirical study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT. *arXiv preprint arXiv:2304.10778*.
58. 1.17. Neural network models (supervised) scikitlearn. Available at: https://scikit-learn.org/stable/modules/neural_networks_supervised.html (Accessed: November 2023).
59. 1.10. Decision trees scikitlearn. Available at: https://scikit-learn.org/stable/modules/tree.html (Accessed: November 2023).
60. Keyes, D.S., Li, B., Kaur, G., Lashkari, A.H., Gagnon, F. and Massicotte, F., 2021. EntropLyzer: Android malware classification and characterization using entropy analysis of dynamic characteristics. In: *Reconciling Data Analytics, Automation, Privacy, and Security: A Big Data Challenge (RDAAPS)*. IEEE, Canada, ON, McMaster University.
61. Rahali, A., Lashkari, A.H., Kaur, G., Taheri, L., Gagnon, F. and Massicotte, F., 2020. DIDroid: Android malware classification and characterization using deep image learning. In: *10th International Conference on Communication and Network Security (ICCNS2020)*. Pages 70–82, Tokyo, Japan, November 2020.

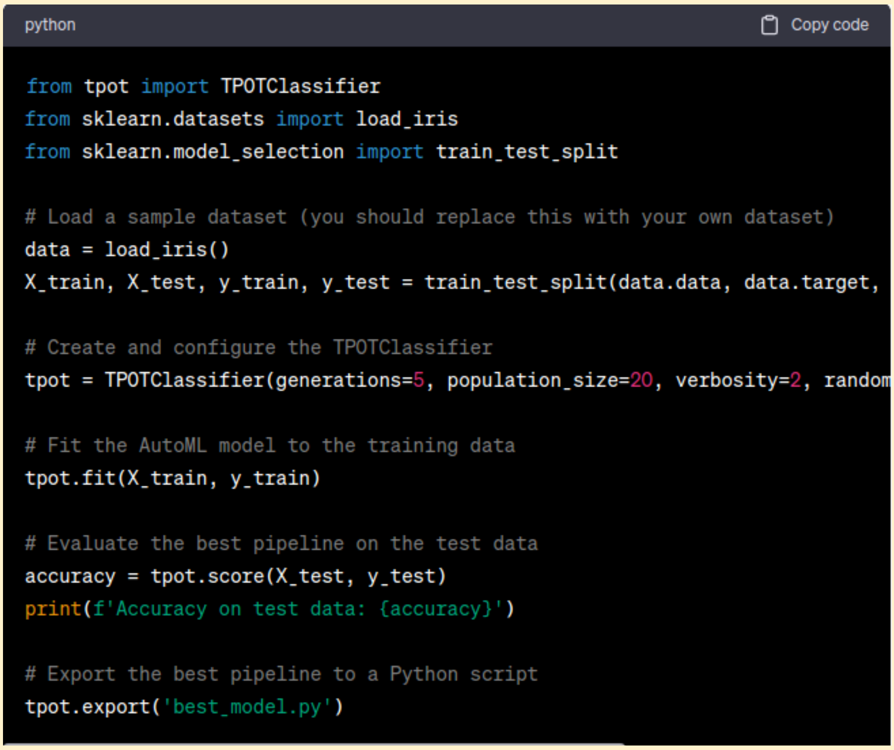
Appendix

Appendix A:

GPT Prompt Log

Classification:	*Green*	*Yellow*	*Red*
Score:	***58.62%***	***22.41%***	***18.97%***

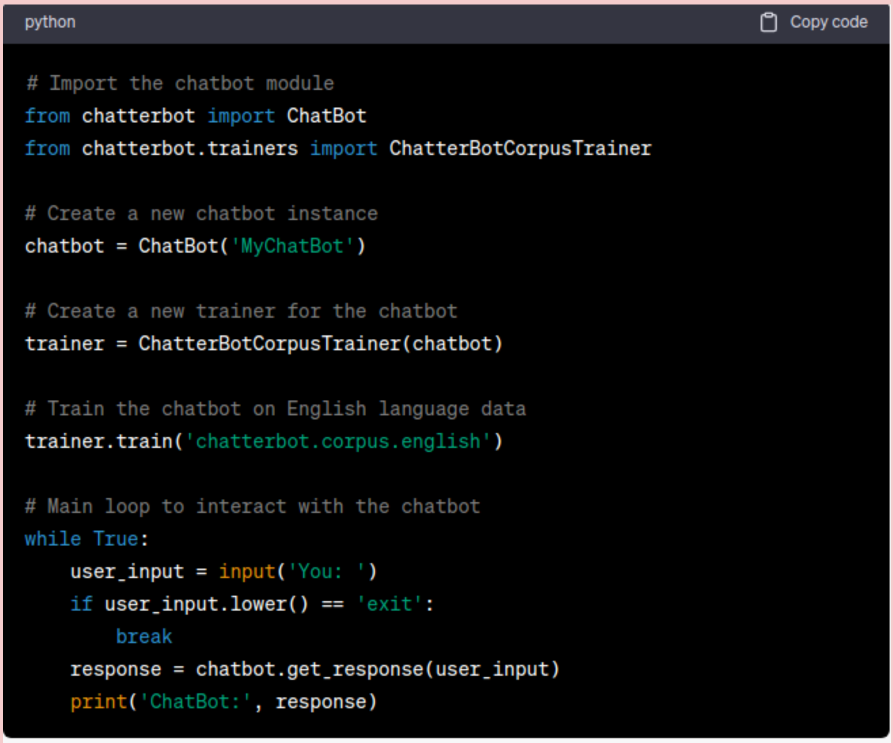
Prompt	**Response**
Generate an AutoML model for me	I can provide you with a basic example of how you might use an AutoML library, such as TPOT, to automate the model selection and hyperparameter tuning process



```python
from tpot import TPOTClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

# Load a sample dataset (you should replace this with your own dataset)
data = load_iris()
X_train, X_test, y_train, y_test = train_test_split(data.data, data.target,

# Create and configure the TPOTClassifier
tpot = TPOTClassifier(generations=5, population_size=20, verbosity=2, random

# Fit the AutoML model to the training data
tpot.fit(X_train, y_train)

# Evaluate the best pipeline on the test data
accuracy = tpot.score(X_test, y_test)
print(f'Accuracy on test data: {accuracy}')

# Export the best pipeline to a Python script
tpot.export('best_model.py')
```

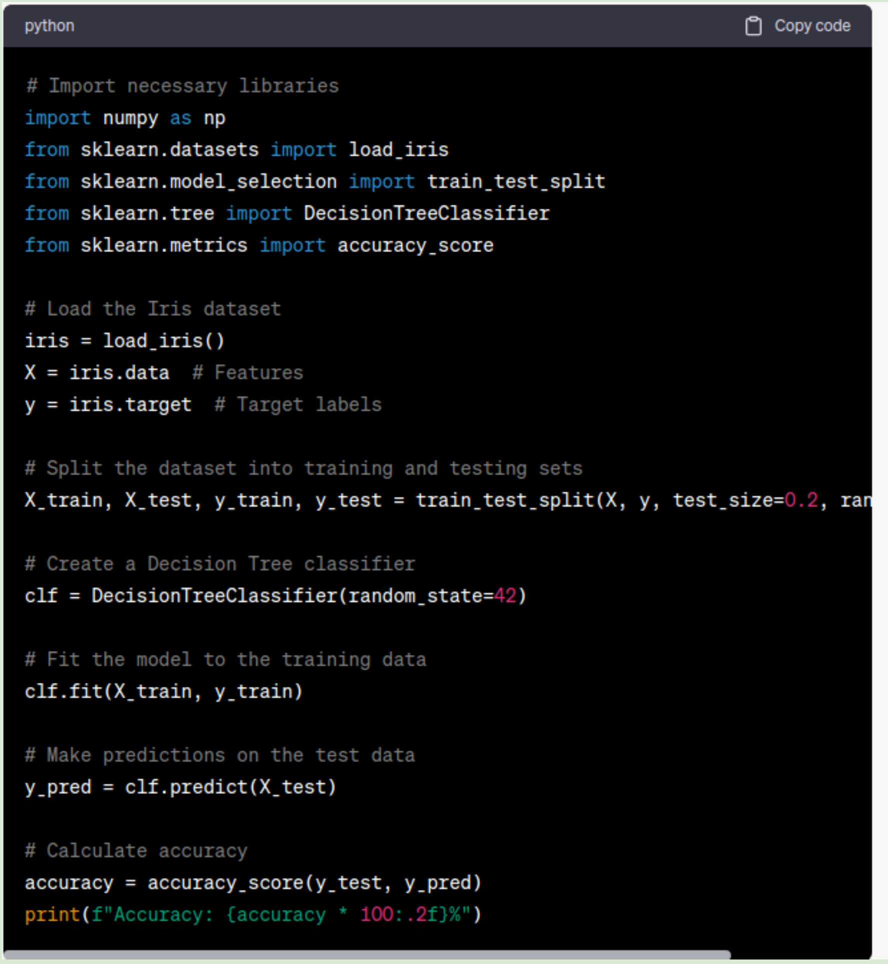
Generate a bidirectional transformer model for natural language processing	Creating a bidirectional transformer model for natural language processing (NLP) from scratch is a complex task that typically requires significant computational resources and expertise in deep learning. However, I can provide you with a high-level overview of the steps involved

Generate Chatbot code for me



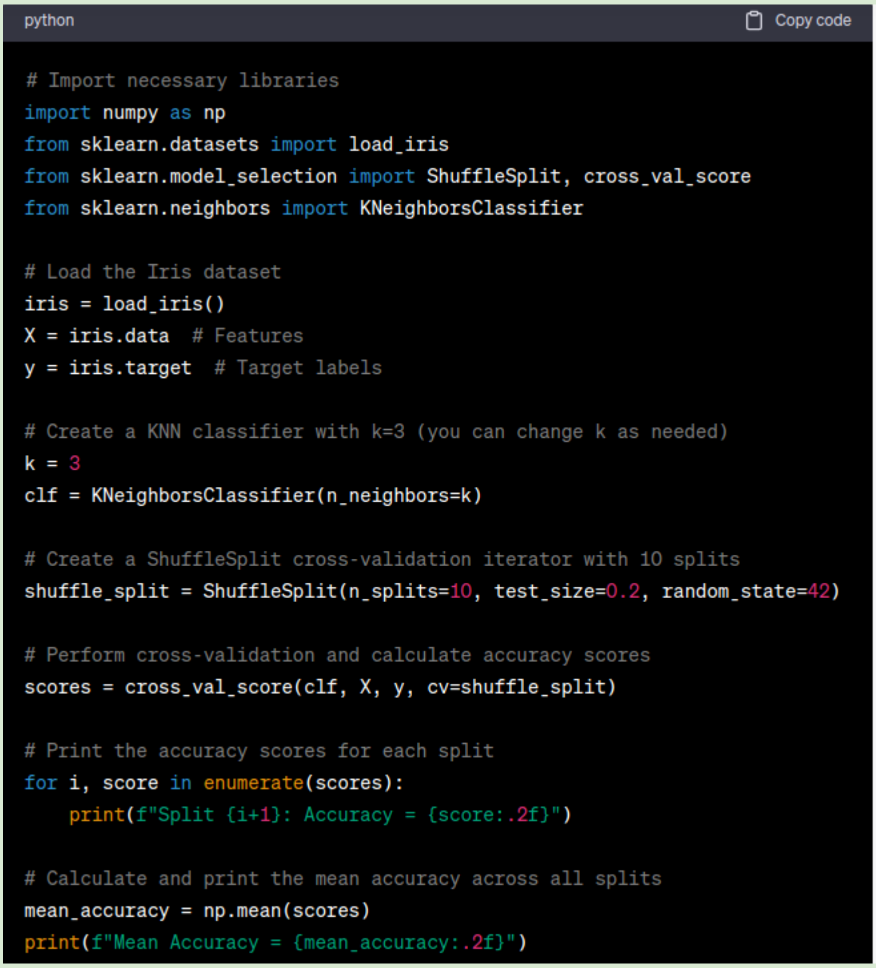
```python
# Import the chatbot module
from chatterbot import ChatBot
from chatterbot.trainers import ChatterBotCorpusTrainer

# Create a new chatbot instance
chatbot = ChatBot('MyChatBot')

# Create a new trainer for the chatbot
trainer = ChatterBotCorpusTrainer(chatbot)

# Train the chatbot on English language data
trainer.train('chatterbot.corpus.english')

# Main loop to interact with the chatbot
while True:
    user_input = input('You: ')
    if user_input.lower() == 'exit':
        break
    response = chatbot.get_response(user_input)
    print('ChatBot:', response)
```

Generate a GPT API model for me	I'm sorry for any confusion, but as of my last knowledge update in September 2021, OpenAI had not released a public API for GPT-3.5 or any later versions. However, OpenAI had previously provided access to GPT-3 through a limited beta program.

	If OpenAI has released an API for GPT-3.5 or any newer models since then, you should visit the OpenAI website or developer portal to get the most up-to-date information on how to access and use their API. Please note that the availability and usage policies may have changed since my last update, so it's essential to refer to the official OpenAI resources for accurate and current information.
Generate a GPT3.5 model for me	I apologise for any confusion, but I cannot generate a GPT-3 model for you. GPT-3 is a proprietary model developed by OpenAI, and access to it typically requires a partnership or API access provided by OpenAI directly.
Can you generate code, or is it searched for	I can certainly help generate code for various programming languages and tasks based on the input and requirements you provide. I can also explain code, provide code samples, and assist with debugging and optimization. However, please keep in mind that while I can generate code, it's important to review and test the code for correctness, security, and efficiency, as I cannot guarantee that the code I generate is free of errors or follows best practices for every specific use case. If you have a specific coding task or question, feel free to describe it, and I'll do my best to assist you in generating code or providing guidance.
Generate an SKLearn model for the iris dataset	

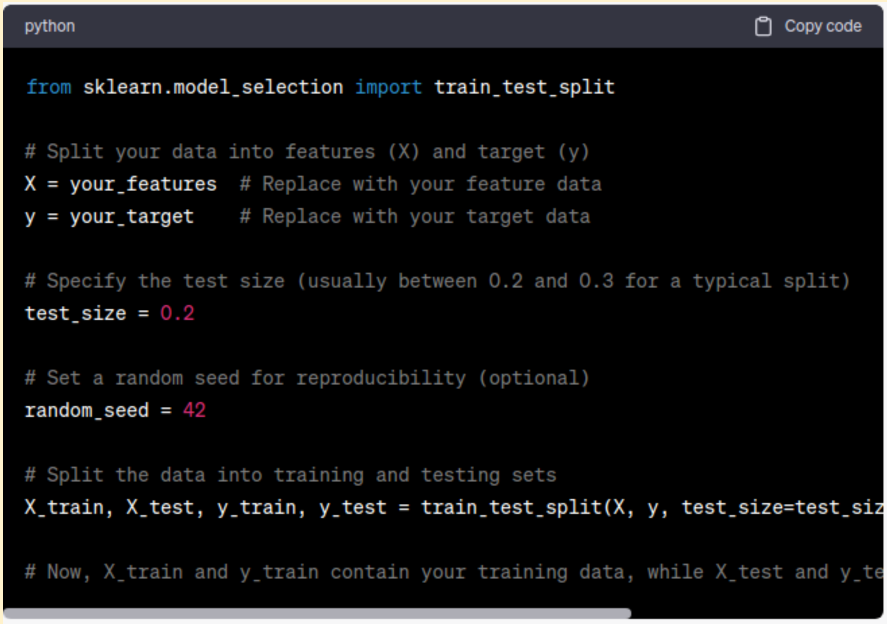
```python
# Import necessary libraries
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

# Load the Iris dataset
iris = load_iris()
X = iris.data  # Features
y = iris.target  # Target labels

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, ran

# Create a Decision Tree classifier
clf = DecisionTreeClassifier(random_state=42)

# Fit the model to the training data
clf.fit(X_train, y_train)

# Make predictions on the test data
y_pred = clf.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy * 100:.2f}%")
```

Where do you get this code from	The code I provided to create a scikit-learn model for the Iris dataset is generated based on my training data and knowledge of Python programming and scikit-learn. I don't retrieve code from external sources; instead, I generate code based on commonly used programming patterns and best practices for the given task.

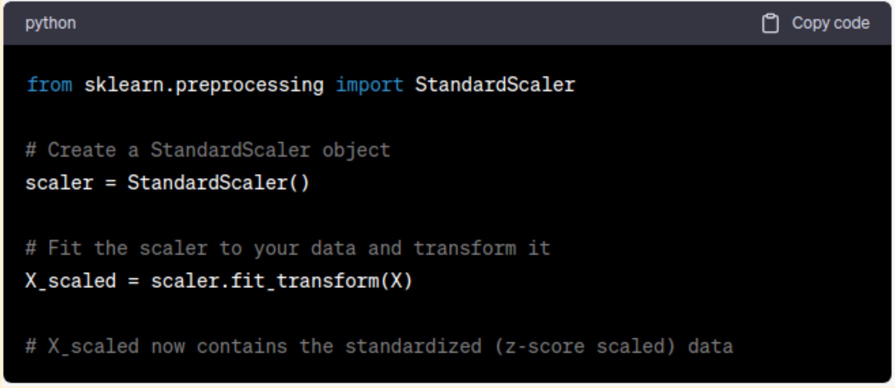
Generate a KNN model for the iris dataset with a shufflesplit of ten and cross validation	

```python
# Import necessary libraries
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import ShuffleSplit, cross_val_score
from sklearn.neighbors import KNeighborsClassifier

# Load the Iris dataset
iris = load_iris()
X = iris.data  # Features
y = iris.target  # Target labels

# Create a KNN classifier with k=3 (you can change k as needed)
k = 3
clf = KNeighborsClassifier(n_neighbors=k)

# Create a ShuffleSplit cross-validation iterator with 10 splits
shuffle_split = ShuffleSplit(n_splits=10, test_size=0.2, random_state=42)

# Perform cross-validation and calculate accuracy scores
scores = cross_val_score(clf, X, y, cv=shuffle_split)

# Print the accuracy scores for each split
for i, score in enumerate(scores):
    print(f"Split {i+1}: Accuracy = {score:.2f}")

# Calculate and print the mean accuracy across all splits
mean_accuracy = np.mean(scores)
print(f"Mean Accuracy = {mean_accuracy:.2f}")
```

Why do all of your machine learning models use random state of 42	The use of a random_state of 42 (or any other fixed number) in machine learning code is a common practice for reproducibility. It ensures that the random processes, such as data splitting, initialization of models, or random shuffling, produce consistent results every time you run the code. Here are a few reasons why a specific value like 42 might be used: ● Reproducibility: Machine learning experiments often involve randomness (e.g., random data splitting, initialization of model weights, or random sampling). Setting a fixed random_state allows researchers, developers, or data scientists to obtain the same results every time they run the code, making it easier to verify and share their work. ● Debugging and Testing: Using a fixed random seed is helpful when debugging or testing machine learning models. It allows you to track down and reproduce specific issues or evaluate the model's behaviour consistently. ● Consistency: When working on a team or collaborating with others, using a common random_state value ensures that everyone gets the same results, making it easier to compare and discuss model performance. While 42 is just a commonly chosen value and not inherently special, any fixed integer can be used as a random_state. The specific value doesn't matter as long as it remains consistent across runs of the code. You could choose any number that's meaningful to you or your team, or simply stick to conventions like 42 for simplicity.
How do you split a dataset into training and testing sets using scikit-learn?	In scikit-learn, you can split a dataset into training and testing sets using the train_test_split function. This function shuffles the data and divides it into two subsets: one for training your machine learning model and the other for testing its performance. Here's how you can use it:

```python
from sklearn.model_selection import train_test_split

# Split your data into features (X) and target (y)
X = your_features  # Replace with your feature data
y = your_target    # Replace with your target data

# Specify the test size (usually between 0.2 and 0.3 for a typical split)
test_size = 0.2

# Set a random seed for reproducibility (optional)
random_seed = 42

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_siz

# Now, X_train and y_train contain your training data, while X_test and y_te
```

Write code to perform standardisation (z-score scaling) on a dataset.	To perform standardisation (z-score scaling) on a dataset in scikit-learn, you can use the StandardScaler class. This scaler will transform your data so that it has a mean of 0 and a standard deviation of 1. Here's how you can use it: 

```python
from sklearn.preprocessing import StandardScaler

# Create a StandardScaler object
scaler = StandardScaler()

# Fit the scaler to your data and transform it
X_scaled = scaler.fit_transform(X)

# X_scaled now contains the standardized (z-score scaled) data
```

Implement a logistic regression classifier for binary classification.	To implement a logistic regression classifier for binary classification in Python using scikit-learn, you can follow these steps. In this example, we'll use a sample dataset for demonstration purposes:

```
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix,
classification_report
# Create a sample dataset for binary classification
X, y = make_classification(n_samples=1000, n_features=2, n_classes=2,
n_clusters_per_class=1, random_state=42)
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
# Create a Logistic Regression classifier
classifier = LogisticRegression()
# Train the classifier on the training data
classifier.fit(X_train, y_train)
# Make predictions on the testing data
y_pred = classifier.predict(X_test)
# Evaluate the classifier
```

```
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
class_report = classification_report(y_test, y_pred)
# Display the results
print("Accuracy:", accuracy)
print("Confusion Matrix:\n", conf_matrix)
print("Classification Report:\n", class_report)
```

How can you visualise the decision boundary of a 2D classifier?

You can visualise the decision boundary of a 2D classifier by creating a contour plot or scatter plot that shows the decision regions or boundaries in the feature space. Here's how you can do it using Python and matplotlib:

Assuming you have a trained classifier classifier and the feature data X (which should be a 2D array with two features), you can follow these steps to visualise the decision boundary:

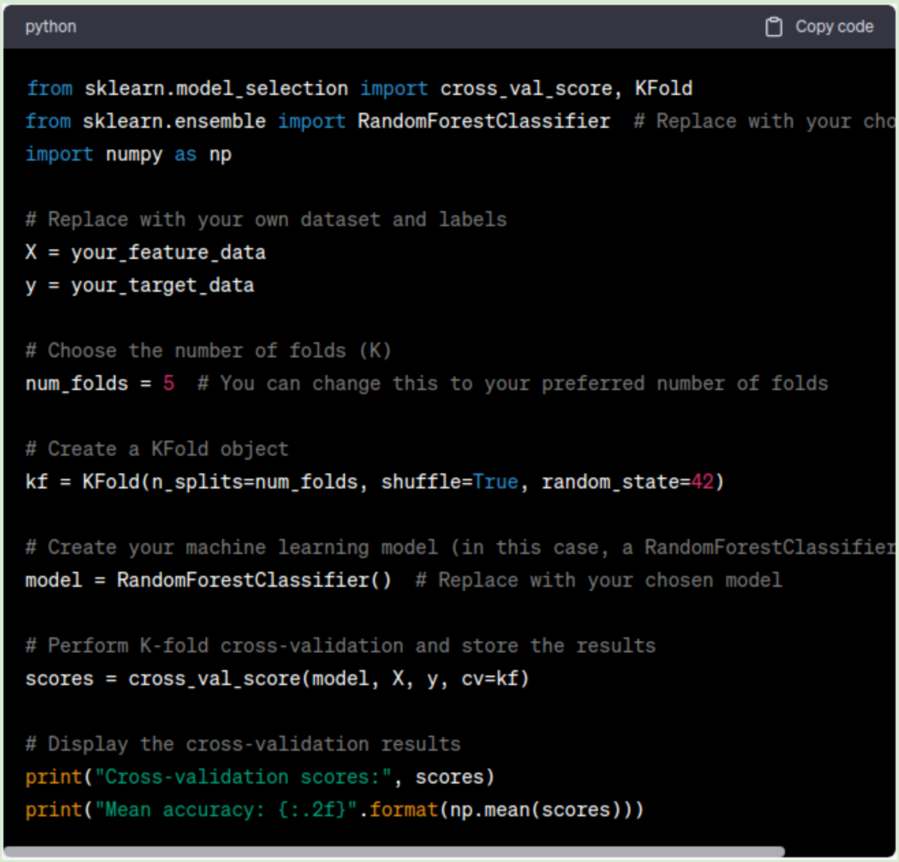
1. Define a grid of points in the feature space to create a mesh for prediction.
2. Use the classifier to predict the class labels for each point in the grid.
3. Create a contour plot or a scatter plot to visualise the decision boundary.

Here's an example:

```
import numpy as np
import matplotlib.pyplot as plt

# Generate a grid of points to create a mesh
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.01), np.arange(y_min, y_max, 0.01))

# Use the classifier to predict the class labels for each point in the grid
Z = classifier.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

# Create a contour plot to visualise the decision boundary
plt.contourf(xx, yy, Z, alpha=0.8, cmap=plt.cm.RdBu)
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.RdBu, edgecolor='k')  # Scatter plot of the data points
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('Decision Boundary of 2D Classifier')
plt.show()
```

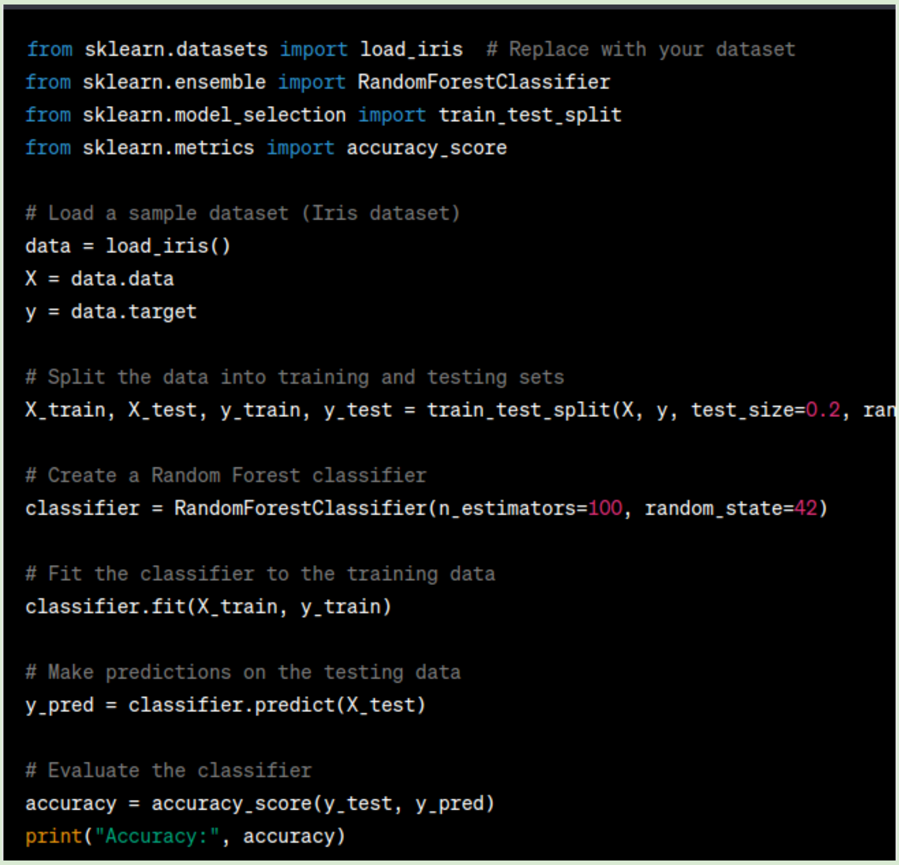
Write code to perform k-fold cross-validation for a machine learning model.

K-fold cross-validation is a technique for assessing a machine learning model's performance by splitting the dataset into K subsets (folds). It involves training and evaluating the model K times, using different subsets as the test set and the remaining data as the training set in each iteration. Here's an example of how to perform K-fold cross-validation using scikit-learn:



```python
from sklearn.model_selection import cross_val_score, KFold
from sklearn.ensemble import RandomForestClassifier  # Replace with your cho
import numpy as np

# Replace with your own dataset and labels
X = your_feature_data
y = your_target_data

# Choose the number of folds (K)
num_folds = 5  # You can change this to your preferred number of folds

# Create a KFold object
kf = KFold(n_splits=num_folds, shuffle=True, random_state=42)

# Create your machine learning model (in this case, a RandomForestClassifier
model = RandomForestClassifier()  # Replace with your chosen model

# Perform K-fold cross-validation and store the results
scores = cross_val_score(model, X, y, cv=kf)

# Display the cross-validation results
print("Cross-validation scores:", scores)
print("Mean accuracy: {:.2f}".format(np.mean(scores)))
```

Create a random forest classifier and fit it to a dataset.



```python
from sklearn.datasets import load_iris  # Replace with your dataset
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Load a sample dataset (Iris dataset)
data = load_iris()
X = data.data
y = data.target

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, ran

# Create a Random Forest classifier
classifier = RandomForestClassifier(n_estimators=100, random_state=42)

# Fit the classifier to the training data
classifier.fit(X_train, y_train)

# Make predictions on the testing data
y_pred = classifier.predict(X_test)

# Evaluate the classifier
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

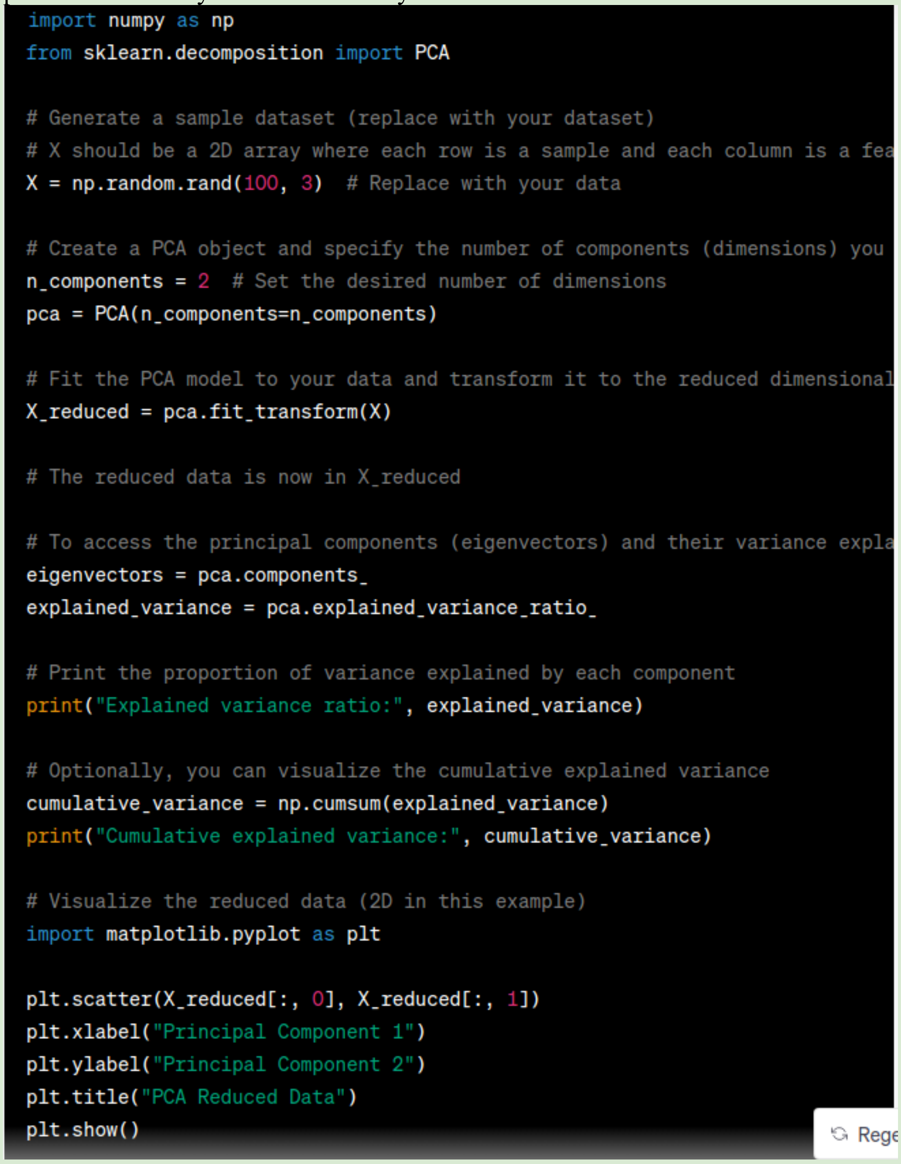
How do you perform hyperparameter tuning using GridSearchCV?	Hyperparameter tuning is a critical step in optimising the performance of a machine learning model. You can use GridSearchCV in scikit-learn to systematically search for the best combination of hyperparameters by evaluating the model's performance using cross-validation. Here's a step-by-step guide on how to perform hyperparameter tuning using GridSearchCV:

```
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier  # Replace with your chosen model
X, y = your_features, your_target  # Replace with your data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
param_grid = {
    'n_estimators': [50, 100, 150],
    'max_depth': [None, 10, 20],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}
model = RandomForestClassifier(random_state=42)
grid_search = GridSearchCV(model, param_grid, cv=5, scoring='accuracy')
grid_search.fit(X_train, y_train)
best_params = grid_search.best_params_
best_model = grid_search.best_estimator_
y_pred = best_model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print("Best Model Accuracy:", accuracy)
```

Implement k-means clustering on a dataset and visualise the clusters.	To implement K-means clustering on a dataset and visualise the clusters, you can use Python with libraries like NumPy, scikit-learn, and Matplotlib. Here's a step-by-step guide:

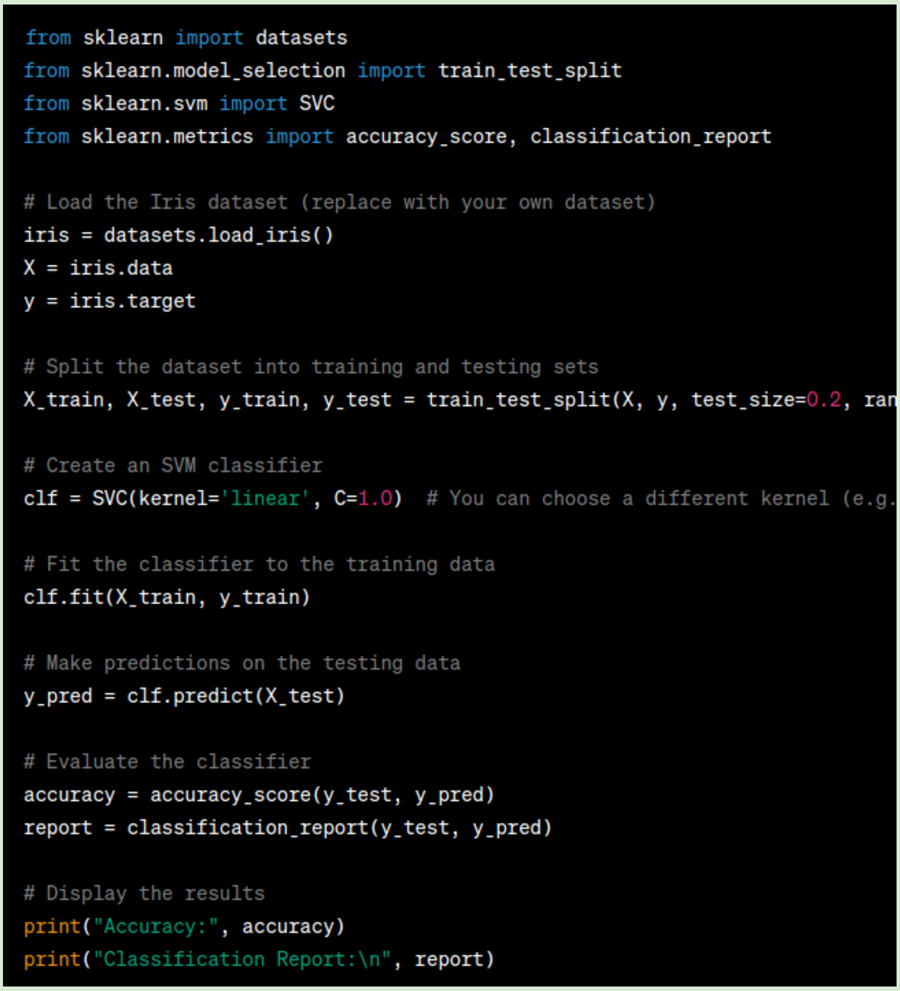
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs  # Replace with your dataset

# Generate a sample dataset (replace with your own data)
X, y = make_blobs(n_samples=300, centers=3, random_state=42)

# Create and fit the K-means model
kmeans = KMeans(n_clusters=3, random_state=42)
kmeans.fit(X)

# Get the cluster labels for each data point
labels = kmeans.labels_

# Get the cluster centers
cluster_centers = kmeans.cluster_centers_

# Visualise the data points and cluster centres
plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis')
plt.scatter(cluster_centers[:, 0], cluster_centers[:, 1,], marker='x', s=200, c='red')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('K-means Clustering')
plt.show()
```

Write code to perform principal component analysis (PCA) for	Principal Component Analysis (PCA) is a technique for dimensionality reduction. It aims to project data into a lower-dimensional space while preserving the most significant variance in the original data. You can use the scikit-learn library to

dimensionality reduction.	perform PCA in Python. Here's how you can do it: 

```
import numpy as np
from sklearn.decomposition import PCA

# Generate a sample dataset (replace with your dataset)
# X should be a 2D array where each row is a sample and each column is a fea
X = np.random.rand(100, 3)  # Replace with your data

# Create a PCA object and specify the number of components (dimensions) you
n_components = 2  # Set the desired number of dimensions
pca = PCA(n_components=n_components)

# Fit the PCA model to your data and transform it to the reduced dimensional
X_reduced = pca.fit_transform(X)

# The reduced data is now in X_reduced

# To access the principal components (eigenvectors) and their variance expla
eigenvectors = pca.components_
explained_variance = pca.explained_variance_ratio_

# Print the proportion of variance explained by each component
print("Explained variance ratio:", explained_variance)

# Optionally, you can visualize the cumulative explained variance
cumulative_variance = np.cumsum(explained_variance)
print("Cumulative explained variance:", cumulative_variance)

# Visualize the reduced data (2D in this example)
import matplotlib.pyplot as plt

plt.scatter(X_reduced[:, 0], X_reduced[:, 1])
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.title("PCA Reduced Data")
plt.show()
```

Build a support vector machine (SVM) classifier for a classification task.	To build a Support Vector Machine (SVM) classifier for a classification task using scikit-learn in Python, you can follow these steps. In this example, I'll use the popular Iris dataset for demonstration:



```
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, classification_report

# Load the Iris dataset (replace with your own dataset)
iris = datasets.load_iris()
X = iris.data
y = iris.target

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, ran

# Create an SVM classifier
clf = SVC(kernel='linear', C=1.0)  # You can choose a different kernel (e.g.

# Fit the classifier to the training data
clf.fit(X_train, y_train)

# Make predictions on the testing data
y_pred = clf.predict(X_test)

# Evaluate the classifier
accuracy = accuracy_score(y_test, y_pred)
report = classification_report(y_test, y_pred)

# Display the results
print("Accuracy:", accuracy)
print("Classification Report:\n", report)
```

How can you calculate the accuracy, precision, recall, and F1-score of a classification model?

You can calculate the accuracy, precision, recall, and F1-score of a classification model using scikit-learn in Python. Here's how to do it:

```
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score, classification_report, confusion_matrix

# Example true and predicted labels
y_true = [1, 0, 1, 1, 0, 0, 1, 0, 1, 1]
y_pred = [1, 0, 1, 0, 1, 0, 1, 1, 0, 1]

# Accuracy
accuracy = accuracy_score(y_true, y_pred)
print("Accuracy:", accuracy)

# Precision
precision = precision_score(y_true, y_pred)
print("Precision:", precision)

# Recall
recall = recall_score(y_true, y_pred)
print("Recall:", recall)

# F1-score
f1 = f1_score(y_true, y_pred)
print("F1-score:", f1)

# Classification report
```

```
report = classification_report(y_true, y_pred)
print("Classification Report:\n", report)

# Confusion matrix
conf_matrix = confusion_matrix(y_true, y_pred)
print("Confusion Matrix:\n", conf_matrix)
```

Implement a simple linear regression model using scikit-learn.

A simple linear regression model can be implemented using scikit-learn for performing linear regression in Python. Here's a step-by-step guide:

```
import numpy as np
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt

# Generate sample data
np.random.seed(0)
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.rand(100, 1)

# Create a LinearRegression model
model = LinearRegression()

# Fit the model to the data
model.fit(X, y)

# Make predictions
X_new = np.array([[0], [2]])
y_pred = model.predict(X_new)

# Visualise the data and the regression line
plt.scatter(X, y, alpha=0.6)
plt.plot(X_new, y_pred, "r-", linewidth=2)
plt.xlabel("X")
plt.ylabel("y")
plt.title("Linear Regression")
plt.show()
```

Create a confusion matrix for evaluating a classifier's performance.

A confusion matrix is a crucial tool for evaluating a classifier's performance, especially in binary and multiclass classification problems. You can create a confusion matrix in Python using scikit-learn. Here's how:

```
from sklearn.metrics import confusion_matrix

# Example true and predicted labels
y_true = [1, 0, 1, 1, 0, 0, 1, 0, 1, 1]
y_pred = [1, 0, 1, 0, 1, 0, 1, 1, 0, 1]

# Create the confusion matrix
confusion = confusion_matrix(y_true, y_pred)

# Display the confusion matrix
print("Confusion Matrix:")
print(confusion)
```

Write code to handle missing data in a dataset using imputation.

Handling missing data is an essential step in data preprocessing. You can use imputation techniques to fill in missing values in a dataset. One common approach is to use the mean, median, or mode of the available data to replace the missing values. Here's an example of how to handle missing data using scikit-learn and the SimpleImputer class:

```
import numpy as np
from sklearn.impute import SimpleImputer
```

```
# Example dataset with missing values
data = np.array([[1, 2, np.nan],
                 [4, np.nan, 6],
                 [7, 8, 9]])

# Create a SimpleImputer object to impute missing values with the mean
imputer = SimpleImputer(strategy='mean')

# Fit the imputer to the dataset and transform it to impute missing values
data_imputed = imputer.fit_transform(data)

# Print the imputed dataset
print("Imputed Data:")
print(data_imputed)
```

Perform feature selection using the SelectKBest method.

Feature selection is an essential step in machine learning to choose the most relevant features to improve model performance and reduce overfitting. The SelectKBest method in scikit-learn is a way to select the top K features based on statistical tests. Here's how to perform feature selection using the SelectKBest method:

```
from sklearn.datasets import load_iris
from sklearn.feature_selection import SelectKBest, chi2

# Load the Iris dataset (replace with your dataset)
data = load_iris()
X = data.data
y = data.target

# Create a SelectKBest object and specify the scoring function
k_best = SelectKBest(score_func=chi2, k=2)  # Select the top 2 features

# Fit and transform the data to select the top K features
X_new = k_best.fit_transform(X, y)

# Visualise the selected features
selected_features = k_best.get_support()
print("Selected features:", selected_features)
```

Implement a Naive Bayes classifier for text classification.

A Naive Bayes classifier is commonly used for text classification tasks, such as spam detection and sentiment analysis. Here's how to implement a Naive Bayes classifier for text classification using scikit-learn in Python. In this example, I'll use the 20 Newsgroups dataset as a sample text classification task:

```
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report
# Load the 20 Newsgroups dataset (you can use your own dataset)
newsgroups = fetch_20newsgroups(subset='all', remove=('headers', 'footers',
'quotes'))

X = newsgroups.data
y = newsgroups.target

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
tfidf_vectorizer = TfidfVectorizer(max_features=5000)
X_train_tfidf = tfidf_vectorizer.fit_transform(X_train)
X_test_tfidf = tfidf_vectorizer.transform(X_test)
```

```
naive_bayes_classifier = MultinomialNB()
naive_bayes_classifier.fit(X_train_tfidf, y_train)
naive_bayes_classifier = MultinomialNB()
naive_bayes_classifier.fit(X_train_tfidf, y_train)
y_pred = naive_bayes_classifier.predict(X_test_tfidf)
accuracy = accuracy_score(y_test, y_pred)
report = classification_report(y_test, y_pred,
target_names=newsgroups.target_names)

print("Accuracy:", accuracy)
print("Classification Report:\n", report)
```

How do you save and load a scikit-learn model to/from a file?

You can save and load a scikit-learn model to/from a file using Python's pickle module, which allows you to serialise and deserialize Python objects. Here's how to do it:

```
import pickle
from sklearn.datasets import load_iris
from sklearn.ensemble import RandomForestClassifier

# Load or create your scikit-learn model
data = load_iris()
X, y = data.data, data.target
model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X, y)

# Save the model to a file using pickle
with open('model.pkl', 'wb') as model_file:
    pickle.dump(model, model_file)
```

Loading a scikit-learn model from a file:

```
import pickle

# Load the saved model from a file using pickle
with open('model.pkl', 'rb') as model_file:
    loaded_model = pickle.load(model_file)
```

Write code to apply Min-Max scaling to a dataset.

Min-Max scaling (also known as feature scaling or normalisation) is a technique used to scale the features of a dataset to a specific range, typically between 0 and 1. You can apply Min-Max scaling to a dataset using scikit-learn. Here's how to do it:

```
from sklearn.preprocessing import MinMaxScaler
import numpy as np

# Create a sample dataset (replace with your data)
data = np.array([[1.0, 2.0, 3.0],
                 [4.0, 5.0, 6.0],
                 [7.0, 8.0, 9.0]])

# Create a MinMaxScaler object
scaler = MinMaxScaler()

# Fit the scaler to your data and transform it to apply Min-Max scaling
scaled_data = scaler.fit_transform(data)

# Display the scaled data
print("Original Data:\n", data)
print("Scaled Data:\n", scaled_data)
```

Create a pipeline to preprocess data and train a machine

Creating a pipeline in scikit-learn is a powerful way to streamline the process of preprocessing data and training a machine learning model. Pipelines allow you to combine multiple steps, such as data preprocessing, feature engineering, and model

learning model.	training, into a single, easy-to-use entity. Here's an example of how to create a simple pipeline for a classification task:

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.ensemble import RandomForestClassifier
from sklearn.pipeline import Pipeline
from sklearn.metrics import accuracy_score

# Load the dataset (Iris dataset as an example)
data = load_iris()
X = data.data
y = data.target

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Create a pipeline with multiple steps
pipeline = Pipeline([
    ('scaler', StandardScaler()),  # Step 1: Standardise the features
    ('pca', PCA(n_components=2)),  # Step 2: Reduce dimensionality with PCA
    ('classifier', RandomForestClassifier(n_estimators=100, random_state=42))  #
Step 3: Train a RandomForestClassifier
])

# Fit the entire pipeline to the training data
pipeline.fit(X_train, y_train)

# Make predictions on the testing data
y_pred = pipeline.predict(X_test)

# Evaluate the classifier's performance
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

Implement a decision tree classifier for a classification task.	A Decision Tree classifier is a popular machine learning algorithm used for both classification and regression tasks. Here's how to implement a Decision Tree classifier for a classification task using scikit-learn in Python:

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, classification_report

# Load the dataset (Iris dataset as an example)
data = load_iris()
X = data.data
y = data.target

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Create a Decision Tree classifier
classifier = DecisionTreeClassifier(random_state=42)

# Fit the classifier to the training data
classifier.fit(X_train, y_train)
```

```
# Make predictions on the testing data
y_pred = classifier.predict(X_test)

# Evaluate the classifier's performance
accuracy = accuracy_score(y_test, y_pred)
report = classification_report(y_test, y_pred, target_names=data.target_names)

print("Accuracy:", accuracy)
print("Classification Report:\n", report)
```

How can you perform one-hot encoding of categorical variables?

One-hot encoding is a technique used to convert categorical variables into a numerical format that can be used for machine learning models. Each category is transformed into a binary (0 or 1) representation, creating new binary columns for each category. You can perform one-hot encoding in Python using libraries like scikit-learn or pandas. Here's how to do it using both methods:

```
from sklearn.preprocessing import OneHotEncoder
import numpy as np

# Sample data with categorical variables
data = np.array([['Red'], ['Green'], ['Blue'], ['Green'], ['Red']])

# Create a OneHotEncoder object
encoder = OneHotEncoder(sparse=False)

# Fit and transform the data to one-hot encoding
one_hot_encoded = encoder.fit_transform(data)

# Display the one-hot encoded data
print("One-Hot Encoded Data:\n", one_hot_encoded)
```

Using pandas (get_dummies):

```
import pandas as pd

# Sample data with categorical variables
data = pd.DataFrame({'Color': ['Red', 'Green', 'Blue', 'Green', 'Red']})

# Perform one-hot encoding using pandas' get_dummies
one_hot_encoded = pd.get_dummies(data, columns=['Color'])

# Display the one-hot encoded data
print("One-Hot Encoded Data:\n", one_hot_encoded)
```

Write code to plot learning curves for a machine learning model.

Plotting learning curves is a useful way to visualise how the performance of a machine learning model evolves as the size of the training dataset increases. Learning curves can help you identify issues like overfitting or underfitting. Here's an example of how to plot learning curves in scikit-learn:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.model_selection import learning_curve
from sklearn.ensemble import RandomForestClassifier

# Load the dataset (Iris dataset as an example)
data = load_iris()
X = data.data
y = data.target

# Create a RandomForestClassifier (replace with your preferred model)
```

```
model = RandomForestClassifier(n_estimators=100, random_state=42)

# Specify the training set sizes
train_sizes, train_scores, test_scores = learning_curve(
    model, X, y, cv=5, train_sizes=np.linspace(0.1, 1.0, 10), n_jobs=-1
)

# Calculate the mean and standard deviation of training and testing scores
train_scores_mean = np.mean(train_scores, axis=1)
train_scores_std = np.std(train_scores, axis=1)
test_scores_mean = np.mean(test_scores, axis=1)
test_scores_std = np.std(test_scores, axis=1)

# Create a learning curve plot
plt.figure(figsize=(8, 6))
plt.title("Learning Curve")
plt.xlabel("Training Examples")
plt.ylabel("Score")

plt.fill_between(
    train_sizes, train_scores_mean - train_scores_std, train_scores_mean +
train_scores_std, alpha=0.1, color="r"
)
plt.fill_between(
    train_sizes, test_scores_mean - test_scores_std, test_scores_mean +
test_scores_std, alpha=0.1, color="g"
)

plt.plot(train_sizes, train_scores_mean, 'o-', color="r", label="Training Score")
plt.plot(train_sizes, test_scores_mean, 'o-', color="g", label="Cross-Validation
Score")

plt.legend(loc="best")
plt.grid()
plt.show()
```

Build a gradient boosting classifier and fit it to a dataset.

Gradient Boosting is a powerful ensemble method used for classification tasks. You can build and fit a Gradient Boosting Classifier to a dataset using scikit-learn in Python. Here's an example:

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import accuracy_score, classification_report

# Load the dataset (Iris dataset as an example)
data = load_iris()
X = data.data
y = data.target

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Create a Gradient Boosting Classifier
classifier = GradientBoostingClassifier(n_estimators=100, random_state=42)

# Fit the classifier to the training data
classifier.fit(X_train, y_train)
```

```
# Make predictions on the testing data
y_pred = classifier.predict(X_test)

# Evaluate the classifier's performance
accuracy = accuracy_score(y_test, y_pred)
report = classification_report(y_test, y_pred, target_names=data.target_names)

print("Accuracy:", accuracy)
print("Classification Report:\n", report)
```

Implement a K-Nearest Neighbors (KNN) classifier

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, classification_report

# Load the dataset (Iris dataset as an example)
data = load_iris()
X = data.data
y = data.target

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Create a KNN classifier
k = 3  # Number of neighbours
classifier = KNeighborsClassifier(n_neighbors=k)

# Fit the classifier to the training data
classifier.fit(X_train, y_train)

# Make predictions on the testing data
y_pred = classifier.predict(X_test)

# Evaluate the classifier's performance
accuracy = accuracy_score(y_test, y_pred)
report = classification_report(y_test, y_pred, target_names=data.target_names)

print("Accuracy:", accuracy)
print("Classification Report:\n", report)
```

How do you perform feature engineering to create new features?

Feature engineering involves creating new features from the existing data to improve a machine learning model's performance. It's a crucial step in the data preprocessing process. Here are some common techniques for performing feature engineering:

Polynomial Features: You can create new features by raising existing features to a power, which is useful when dealing with non-linear relationships:

```
from sklearn.preprocessing import PolynomialFeatures
poly = PolynomialFeatures(degree=2)
X_poly = poly.fit_transform(X)
```

Interaction Features: Create new features as the product of two or more existing features. This can capture relationships between variables.

Binning or Discretization: Convert continuous numerical features into categorical features by creating bins or intervals.

```
from sklearn.preprocessing import KBinsDiscretizer
discretizer = KBinsDiscretizer(n_bins=3, encode='onehot', strategy='uniform')
X_binned = discretizer.fit_transform(X)
```

One-Hot Encoding: Convert categorical variables into a binary vector to represent the categories.

```
from sklearn.preprocessing import StandardScaler
```

```
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

Feature Extraction: Use dimensionality reduction techniques like Principal Component Analysis (PCA) or Linear Discriminant Analysis (LDA) to create a new set of features that capture most of the variance in the original data.

```
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)
```

Time Series Features: Extract time-based features such as day of the week, month, or year from timestamps in time series data.

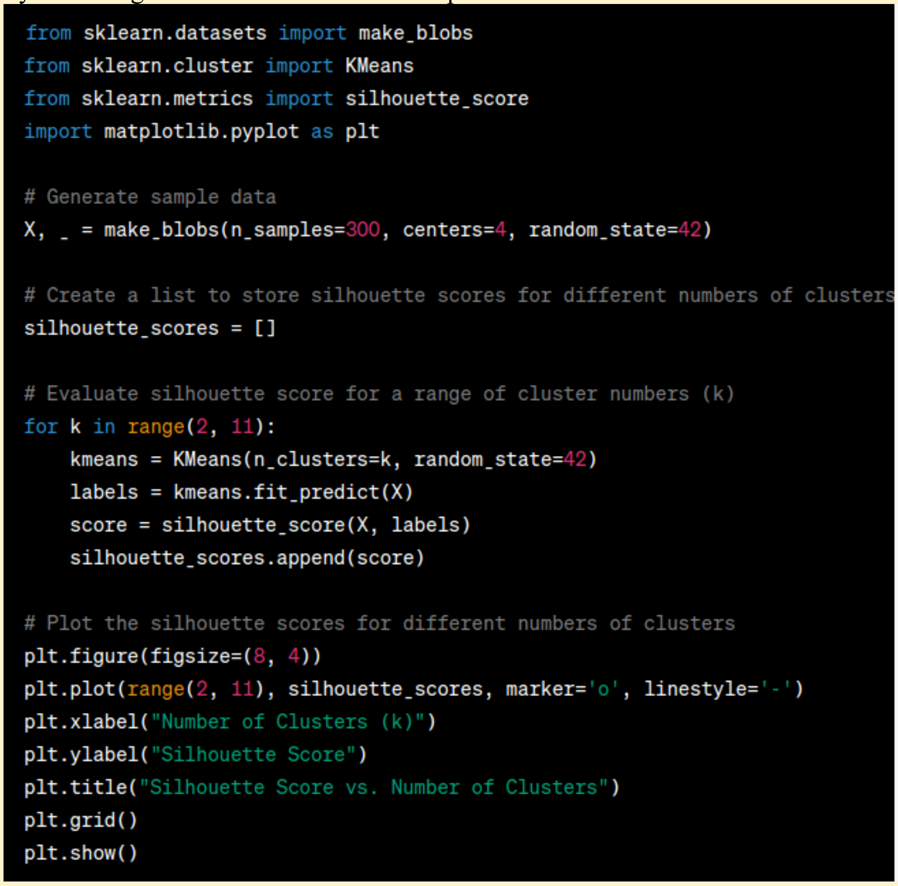
Text Data Features: For text data, you can perform techniques like TF-IDF, word embeddings (Word2Vec, GloVe), or topic modelling to create features from text.

Domain-Specific Features: Incorporate domain knowledge to engineer features that are relevant to the problem you're solving.

Feature Crosses: Combine two or more features to create new ones that capture interactions between them.

Write code to calculate the silhouette score for clustering evaluation.

The silhouette score is a metric used to evaluate the quality of clusters in a clustering algorithm. A higher silhouette score indicates that the data points are well clustered and have good separation between clusters. You can calculate the silhouette score in Python using scikit-learn. Here's an example:



```
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
import matplotlib.pyplot as plt

# Generate sample data
X, _ = make_blobs(n_samples=300, centers=4, random_state=42)

# Create a list to store silhouette scores for different numbers of clusters
silhouette_scores = []

# Evaluate silhouette score for a range of cluster numbers (k)
for k in range(2, 11):
    kmeans = KMeans(n_clusters=k, random_state=42)
    labels = kmeans.fit_predict(X)
    score = silhouette_score(X, labels)
    silhouette_scores.append(score)

# Plot the silhouette scores for different numbers of clusters
plt.figure(figsize=(8, 4))
plt.plot(range(2, 11), silhouette_scores, marker='o', linestyle='-')
plt.xlabel("Number of Clusters (k)")
plt.ylabel("Silhouette Score")
plt.title("Silhouette Score vs. Number of Clusters")
plt.grid()
plt.show()
```

Create a ROC curve for a binary classification model.

A Receiver Operating Characteristic (ROC) curve is a useful tool for visualising the performance of a binary classification model, specifically for varying thresholds. You can create an ROC curve in Python using scikit-learn. Here's an example:

```
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import roc_curve, roc_auc_score, auc
import matplotlib.pyplot as plt
```

```
# Generate synthetic binary classification data
X, y = make_classification(n_samples=1000, n_features=20, random_state=42)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Create a binary classification model (Random Forest Classifier as an example)
model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X_train, y_train)

# Get predicted probabilities for the positive class
y_scores = model.predict_proba(X_test)[:, 1]

# Calculate ROC curve
fpr, tpr, thresholds = roc_curve(y_test, y_scores)

# Calculate AUC (Area Under the ROC Curve)
roc_auc = auc(fpr, tpr)

# Plot the ROC curve
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (AUC =
{roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.grid()
plt.show()
```

Implement a random search for hyperparameter tuning.	Randomised Search is a technique for hyperparameter tuning that randomly samples a set of hyperparameter combinations from predefined ranges. It's more efficient than an exhaustive grid search and is available in scikit-learn through the RandomizedSearchCV class. Here's an example of how to perform hyperparameter tuning using randomised search:

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split, RandomizedSearchCV
from sklearn.ensemble import RandomForestClassifier
import numpy as np

# Load the dataset (Iris dataset as an example)
data = load_iris()
X = data.data
y = data.target

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Define hyperparameter ranges to search
param_dist = {
    'n_estimators': [50, 100, 150, 200],
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': ['auto', 'sqrt', 'log2', None],
}

# Create a RandomizedSearchCV object
clf = RandomForestClassifier(random_state=42)
random_search = RandomizedSearchCV(clf, param_distributions=param_dist,
n_iter=20, cv=5, random_state=42)

# Fit the randomised search to the training data
random_search.fit(X_train, y_train)

# Get the best hyperparameters
best_params = random_search.best_params_
print("Best Hyperparameters:", best_params)

# Make predictions using the best model
best_model = random_search.best_estimator_
y_pred = best_model.predict(X_test)

# Evaluate the model's performance
from sklearn.metrics import accuracy_score
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

How can you perform time series forecasting using scikit-learn?	Scikit-learn is primarily designed for traditional machine learning tasks, and while it may not be the best choice for time series forecasting, you can still use it for some aspects of time series analysis, especially when time series data is transformed into a supervised learning problem. Below are the general steps to perform time series forecasting using scikit-learn: ● Data Preparation: You need to prepare your time series data for use in a supervised learning context. This typically involves creating lag features or using a rolling window approach to convert time series data into a tabular format. Each row in your dataset represents a specific time point with features derived from past time steps.

- Feature Engineering: Define lag features as input features and the future values you want to predict as the target variable.
- Train-Test Split: Split your dataset into a training set and a testing (validation) set, keeping the temporal order of the data intact. You might use a time-based split or cross-validation if the order matters.
- Model Selection: Choose a scikit-learn model suitable for regression or time series forecasting, such as linear regression, support vector regression, decision tree regression, or ensemble methods.
- Model Training: Fit the selected model to the training data.
- Model Evaluation: Evaluate the model's performance using appropriate regression metrics (e.g., Mean Absolute Error, Mean Squared Error, etc.).
- Hyperparameter Tuning: Perform hyperparameter tuning if needed to optimise the model's performance.
- Forecasting: Use the trained model to make predictions on the test data or future data points.

Here's a simplified example of how to perform time series forecasting using linear regression in scikit-learn:

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
import numpy as np

# Simulated time series data
# Replace this with your own time series data
time_series = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
target = np.array([11, 13, 15, 17, 19, 21, 23, 25, 27, 29])

# Create lag features
lags = 3
X, y = [], []
for i in range(lags, len(time_series)):
    X.append(time_series[i-lags:i])
    y.append(target[i])

# Train-test split
split = int(0.7 * len(X))
X_train, y_train = X[:split], y[:split]
X_test, y_test = X[split:], y[split:]

# Create and train a linear regression model
model = LinearRegression()
model.fit(X_train, y_train)

# Make predictions
y_pred = model.predict(X_test)

# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error: {mse}")
```

Write code to handle class imbalance in a classification problem.

Handling class imbalance is crucial in a classification problem, as imbalanced datasets can lead to biassed models that perform poorly on underrepresented classes. There are various techniques to address class imbalance. Here's an example of how to do it in Python using scikit-learn:

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix
from imblearn.over_sampling import RandomOverSampler
```

```
from imblearn.under_sampling import RandomUnderSampler
from imblearn.combine import SMOTEENN

# Load the dataset (Iris dataset as an example)
data = load_iris()
X = data.data
y = data.target

# Introduce class imbalance (undersample one class)
X, y = X[y != 0], y[y != 0]

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# RandomOverSampler to oversample minority class
# ros = RandomOverSampler(sampling_strategy='auto', random_state=42)
# X_train, y_train = ros.fit_resample(X_train, y_train)

# RandomUnderSampler to undersample majority class
# rus = RandomUnderSampler(sampling_strategy='auto', random_state=42)
# X_train, y_train = rus.fit_resample(X_train, y_train)

# SMOTE-ENN to oversample and clean
# smote_enn = SMOTEENN(sampling_strategy='auto', random_state=42)
# X_train, y_train = smote_enn.fit_resample(X_train, y_train)

# Create a Random Forest Classifier
clf = RandomForestClassifier(n_estimators=100, random_state=42)

# Fit the classifier to the training data
clf.fit(X_train, y_train)

# Make predictions on the testing data
y_pred = clf.predict(X_test)

# Evaluate the classifier's performance
accuracy = accuracy_score(y_test, y_pred)
report = classification_report(y_test, y_pred, target_names=data.target_names)
conf_matrix = confusion_matrix(y_test, y_pred)

print("Accuracy:", accuracy)
print("Classification Report:\n", report)
print("Confusion Matrix:\n", conf_matrix)
```

Build a multi-layer perceptron (MLP) neural network classifier.

A Multi-Layer Perceptron (MLP) is a type of feedforward neural network commonly used for classification tasks. You can create an MLP classifier in Python using libraries like scikit-learn and Keras. Below is an example using scikit-learn:

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score, classification_report

# Load the dataset (Iris dataset as an example)
data = load_iris()
X = data.data
y = data.target

# Split the data into training and testing sets
```


```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Create an MLP classifier
# You can customise the architecture by adjusting the hidden_layer_sizes,
activation, and other hyperparameters.
mlp = MLPClassifier(hidden_layer_sizes=(10, 10), activation='relu',
max_iter=1000, random_state=42)

# Fit the classifier to the training data
mlp.fit(X_train, y_train)

# Make predictions on the testing data
y_pred = mlp.predict(X_test)

# Evaluate the classifier's performance
accuracy = accuracy_score(y_test, y_pred)
report = classification_report(y_test, y_pred, target_names=data.target_names)

print("Accuracy:", accuracy)
print("Classification Report:\n", report)
```

Implement a Ridge regression model for regression tasks.	Ridge regression is a linear regression technique used for regression tasks, especially when dealing with multicollinearity in the data. Here's how to implement Ridge regression using scikit-learn in Python:

```
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.linear_model import Ridge
from sklearn.metrics import mean_squared_error, r2_score

# Load the dataset (Boston Housing dataset as an example)
data = load_boston()
X = data.data
y = data.target

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Create a Ridge regression model
alpha = 1.0  # Regularisation strength (adjust as needed)
ridge = Ridge(alpha=alpha)

# Fit the model to the training data
ridge.fit(X_train, y_train)

# Make predictions on the testing data
y_pred = ridge.predict(X_test)

# Evaluate the model's performance
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print("Mean Squared Error:", mse)
print("R-squared (R2) Score:", r2)
```

How do you perform model stacking using scikit-learn?	Model stacking, also known as stacked generalisation, is an ensemble machine learning technique that combines the predictions of multiple models to improve overall predictive performance. Scikit-learn doesn't have native support for model

stacking, but you can implement it manually. Here's a general outline of how to perform model stacking using scikit-learn and Python:

1. Create a Diverse Set of Base Models:
    a. Select a set of diverse base models, which can be classifiers, regressors, or any other machine learning algorithms.
    b. Train each base model on your training data.
2. Generate Predictions from Base Models:
    a. Use the trained base models to make predictions on the validation and/or test datasets.
3. Combine Base Models' Predictions:
    a. Create a new dataset that consists of the predictions made by each base model for the validation or test data.
4. Train a Meta-Model:
    a. Choose a meta-model (a model that takes the predictions from the base models as input) and train it using the new dataset created in the previous step.
    b. This meta-model learns to make the final predictions based on the outputs of the base models.
5. Stacking in Practice:
    a. You can implement the stacking process using NumPy or pandas to handle the data manipulation.
    b. You can also use scikit-learn to create a meta-model and stack the predictions.
    c. Ensure that the validation process avoids data leakage. For example, use k-fold cross-validation to prevent the meta-model from seeing the same data used in training the base models.

Here's a simplified example of model stacking using scikit-learn for a binary classification task:

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# Load the dataset (Iris dataset as an example)
data = load_iris()
X = data.data
y = (data.target == 2).astype(int)  # Binary classification, 1 if class 2, 0 otherwise

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Create two base models
base_model_1 = RandomForestClassifier(n_estimators=100, random_state=42)
base_model_2 = LogisticRegression(random_state=42)

# Train the base models
base_model_1.fit(X_train, y_train)
base_model_2.fit(X_train, y_train)

# Generate predictions from the base models
preds_base_1 = base_model_1.predict(X_test)
preds_base_2 = base_model_2.predict(X_test)

# Combine base models' predictions
stacked_X = np.column_stack((preds_base_1, preds_base_2))
```

```
# Train a meta-model (e.g., Logistic Regression) on the stacked data
meta_model = LogisticRegression()
meta_model.fit(stacked_X, y_test)

# Make predictions with the meta-model
stacked_preds = meta_model.predict(stacked_X)

# Evaluate the stacked model's performance
stacked_accuracy = accuracy_score(y_test, stacked_preds)
print("Stacked Model Accuracy:", stacked_accuracy)
```

Write code to perform text classification using TF-IDF vectors.

Text classification using TF-IDF (Term Frequency-Inverse Document Frequency) vectors is a common technique in natural language processing. Scikit-learn provides a straightforward way to perform text classification using TF-IDF vectors. Here's an example for binary text classification:

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score, classification_report

# Sample text data for binary classification
corpus = [
    "This is a positive document",
    "Another positive example",
    "Negative sentiment in this text",
    "Not a good review",
    "This is negative",
]

labels = [1, 1, 0, 0, 0]  # 1 for positive, 0 for negative

# Create a TF-IDF vectorizer
tfidf_vectorizer = TfidfVectorizer()

# Transform the text data into TF-IDF vectors
X = tfidf_vectorizer.fit_transform(corpus)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, labels, test_size=0.2,
random_state=42)

# Create a classifier (e.g., Multinomial Naive Bayes)
classifier = MultinomialNB()

# Fit the classifier to the training data
classifier.fit(X_train, y_train)

# Make predictions on the testing data
y_pred = classifier.predict(X_test)

# Evaluate the classifier's performance
accuracy = accuracy_score(y_test, y_pred)
report = classification_report(y_test, y_pred, target_names=["Negative",
"Positive"])

print("Accuracy:", accuracy)
print("Classification Report:\n", report)
```

Create a custom transformer for data preprocessing.

Creating a custom transformer for data preprocessing in scikit-learn is a powerful way to encapsulate data transformations and make your data preprocessing pipeline more modular and maintainable. You can create custom transformers by subclassing the TransformerMixin class from scikit-learn and implementing the fit and transform methods. Here's an example of creating a custom transformer to perform z-score scaling (standardisation) on a dataset:

```
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.preprocessing import StandardScaler

class ZScoreScaler(BaseEstimator, TransformerMixin):
    def __init__(self):
        self.scaler = StandardScaler()

    def fit(self, X, y=None):
        # Fit the scaler on the input data
        self.scaler.fit(X)
        return self

    def transform(self, X):
        # Transform the input data using the fitted scaler
        return self.scaler.transform(X)
```

Now you can use this custom transformer as part of your scikit-learn pipeline for data preprocessing. For example:

```
from sklearn.pipeline import Pipeline

# Sample data
X = [[1, 2], [3, 4], [5, 6]]

# Create a data preprocessing pipeline with the custom transformer
preprocessing_pipeline = Pipeline([
    ('z_score_scaler', ZScoreScaler())
])

# Fit and transform the data
X_transformed = preprocessing_pipeline.fit_transform(X)
```

Implement a Gaussian Mixture Model (GMM) for clustering.

A Gaussian Mixture Model (GMM) is a probabilistic model that represents a mixture of Gaussian distributions. GMMs are often used for clustering, where each Gaussian component represents one cluster. You can implement a GMM for clustering using scikit-learn. Here's an example:

```
from sklearn.mixture import GaussianMixture
import numpy as np
import matplotlib.pyplot as plt

# Create synthetic data
np.random.seed(0)
n_samples = 300
X = np.concatenate((np.random.randn(n_samples, 2), 5 +
np.random.randn(n_samples, 2)))

# Create and fit a GMM model
n_components = 2
gmm = GaussianMixture(n_components=n_components, random_state=0)
gmm.fit(X)

# Predict cluster labels for each data point
labels = gmm.predict(X)

# Get the means and covariances of the Gaussian components
```

```
means = gmm.means_
covariances = gmm.covariances_

# Visualise the data and clustering results
plt.scatter(X[:, 0], X[:, 1], c=labels, s=40, cmap='viridis')
plt.scatter(means[:, 0], means[:, 1], s=200, c='red', marker='X', label='Cluster
Centers')
plt.legend()
plt.title('GMM Clustering')
plt.show()
```

How can you handle imbalanced classes in a multi-class classification problem?

Handling imbalanced classes in a multi-class classification problem is important to ensure that the model doesn't become biassed towards the majority class. Several techniques can be applied to address class imbalance in multi-class classification. Here are some strategies:

1. Resampling Techniques:
    a. Oversampling: Increase the number of instances in the minority classes by duplicating or generating synthetic examples. Methods like SMOTE (Synthetic Minority Over-sampling Technique) can be applied to create synthetic examples.
    b. Undersampling: Reduce the number of instances in the majority class by randomly removing examples. Be cautious not to remove too much data, as this may lead to a loss of information.
2. Cost-Sensitive Learning:
    a. Assign different misclassification costs to different classes. This can be done by setting class weights in the learning algorithm. Many classifiers in scikit-learn support setting class weights.
3. Ensemble Methods:
    a. Use ensemble techniques such as Balanced Random Forest or EasyEnsemble that combine multiple models to balance the class distribution while making predictions.
4. Anomaly Detection:
    a. Treat the minority class as an anomaly detection problem, where you model the majority class and classify instances from the minority class as anomalies.
5. Data Augmentation:
    a. Augment the data in the minority classes by applying techniques like text augmentation for natural language processing tasks. This can help generate additional training data.
6. Transfer Learning:
    a. Utilise pre-trained models or features from related tasks or domains. Transfer learning can help when there is limited data in minority classes.
7. Re-sampling with Clustering:
    a. Apply clustering techniques to identify clusters within the majority class and then perform resampling techniques within each cluster. This can help preserve diversity within the majority class.
8. Evaluation Metrics:
    a. Choose appropriate evaluation metrics that are not biassed towards the majority class. Metrics like F1-score, precision-recall curves, or area under the precision-recall curve (AUC-PR) are often better for imbalanced multiclass problems.
9. Modify Decision Thresholds:
    a. Adjust the decision threshold of the classifier to improve the balance between precision and recall. This can help find a more suitable trade-off.
10. Algorithm Selection:
    a. Consider using algorithms specifically designed to handle class

	imbalance, like cost-sensitive algorithms or ensemble methods designed for imbalanced data. The choice of which strategy to apply depends on the specific characteristics of your multi-class classification problem. It's often necessary to experiment with different techniques and evaluate their performance using appropriate metrics to determine the most effective approach.
Write code to perform stratified sampling for dataset splitting.	Stratified sampling is a technique used to ensure that the class distribution in the training and testing sets remains representative of the original dataset. In scikit-learn, you can perform stratified sampling using the train_test_split function with the stratify parameter set to the target variable. Here's an example:

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

# Load the dataset (Iris dataset as an example)
data = load_iris()
X = data.data
y = data.target

# Perform stratified sampling
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y,
random_state=42)

# Print the class distribution in the training and testing sets
from collections import Counter

train_class_distribution = Counter(y_train)
test_class_distribution = Counter(y_test)

print("Class distribution in the training set:", train_class_distribution)
print("Class distribution in the testing set:", test_class_distribution)
```

Build a Recurrent Neural Network (RNN) using scikit-learn.	Scikit-learn is primarily focused on traditional machine learning models and doesn't provide built-in support for recurrent neural networks (RNNs). For building RNNs and more complex neural networks, you should consider using deep learning frameworks like TensorFlow or PyTorch. Here's a basic example of building a simple RNN using TensorFlow:

```
import tensorflow as tf
from tensorflow.keras.layers import SimpleRNN, Dense
from tensorflow.keras.models import Sequential
import numpy as np

# Sample data
X = np.random.random((100, 10, 1))  # 100 sequences of length 10, with 1 feature
y = np.random.randint(0, 2, size=(100,))

# Create a Sequential model
model = Sequential()

# Add an RNN layer with 32 units
model.add(SimpleRNN(32, input_shape=(10, 1)))

# Add a Dense layer for binary classification
model.add(Dense(1, activation='sigmoid'))

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Fit the model to the data
```

	`model.fit(X, y, epochs=10, batch_size=32)`
Implement a Lasso regression model for feature selection.	Lasso regression is a linear regression technique that includes L1 regularisation, which can be used for feature selection by encouraging some feature coefficients to be exactly zero. You can implement Lasso regression using scikit-learn to perform feature selection. Here's an example:

```
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.linear_model import Lasso
from sklearn.metrics import mean_squared_error, r2_score

# Load the dataset (Boston Housing dataset as an example)
data = load_boston()
X = data.data
y = data.target

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Create a Lasso regression model
alpha = 1.0  # Regularisation strength (adjust as needed)
lasso = Lasso(alpha=alpha)

# Fit the model to the training data
lasso.fit(X_train, y_train)

# Make predictions on the testing data
y_pred = lasso.predict(X_test)

# Evaluate the model's performance
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print("Mean Squared Error:", mse)
print("R-squared (R2) Score:", r2)
```

How do you handle time-dependent features in machine learning?	Handling time-dependent features in machine learning is essential when working with data where time plays a crucial role, such as time series data or event data. Here are some strategies for dealing with time-dependent features: 1. Lag Features: a. Create lag features by shifting the values of the feature in time. This can be particularly useful for time series forecasting or prediction tasks where past values of a feature are important. 2. Rolling Statistics: a. Compute rolling statistics like moving averages, standard deviations, or other aggregation functions over a window of time. This can help capture trends and seasonality in the data. 3. Time-Based Features: a. Extract features from the timestamp, such as day of the week, month, year, or time of day. These features can capture periodic behaviour and seasonality. 4. Time Since Events: a. Calculate the time elapsed since specific events or occurrences. This can be valuable in event-driven analysis. 5. Time Decay: a. Apply time decay to features, giving more weight to recent data

	and less to older data. This is useful when older data becomes less relevant over time. 6. Resampling: a. Resample time-dependent data to different time frequencies, aggregating or interpolating values as needed. This is often used when merging data from different sources with varying time resolutions. 7. Feature Engineering: a. Engineer domain-specific features that account for the temporal nature of the data. For example, in finance, you might create features like moving averages, volatility, or momentum. 8. Windowed Features: a. Create features based on windows or time intervals. For example, you could compute statistics for the last 7 days or the next 3 hours. 9. Cross-Validation: a. Use time-based cross-validation techniques, like time series cross-validation, to ensure that your model doesn't train on future data when predicting past events. 10. Feature Selection: a. Use feature selection techniques to identify the most relevant time-dependent features for your specific task. Features with low importance can be pruned to reduce model complexity. 11. Model Selection: a. Choose models that are well-suited for time-dependent data, such as autoregressive models, recurrent neural networks (RNNs), or gradient boosting models. These models can capture temporal dependencies. 12. Ensemble Learning: a. Combine predictions from different models or different time intervals to improve overall performance. This can be useful for tasks like time series forecasting. 13. Regularisation: a. Apply regularisation techniques to control model complexity, especially when working with high-dimensional time-dependent data. L1 regularisation can help with feature selection. 14. Data Preprocessing: a. Ensure your data is properly preprocessed for time-dependent tasks, such as handling missing values, scaling, and encoding categorical features. Handling time-dependent features effectively requires a deep understanding of the data, the problem domain, and the specific requirements of the machine learning task. It often involves a combination of feature engineering, model selection, and domain expertise to make informed decisions.
Write code to apply feature scaling to a subset of columns in a DataFrame.	You can apply feature scaling to a subset of columns in a DataFrame using libraries like pandas and scikit-learn. Here's an example of how to do it:

```
import pandas as pd
from sklearn.preprocessing import StandardScaler

# Sample DataFrame
data = {
    'Feature1': [10, 20, 30, 40, 50],
    'Feature2': [0.1, 0.2, 0.3, 0.4, 0.5],
    'Feature3': [5, 10, 15, 20, 25]
}

df = pd.DataFrame(data)
```

```
# Columns to be scaled
columns_to_scale = ['Feature1', 'Feature2']

# Create a StandardScaler
scaler = StandardScaler()

# Fit and transform the selected columns
df[columns_to_scale] = scaler.fit_transform(df[columns_to_scale])

# Print the scaled DataFrame
print(df)
```

Create a custom scoring metric for model evaluation.

Creating a custom scoring metric for model evaluation in scikit-learn can be done by defining a Python function that computes the metric you want to use. You can then use this custom metric when cross-validating or evaluating your models. Here's an example of how to create a custom scoring metric:

Suppose you want to create a custom scoring metric called "custom_metric" for a classification problem that takes the true labels y_true and predicted labels y_pred as input. This metric calculates the F1-score for a specific class (e.g., class 1) while ignoring other classes.

```
from sklearn.metrics import make_scorer
from sklearn.metrics import f1_score

def custom_metric(y_true, y_pred):
    # Calculate F1-score for class 1 (positive class) while ignoring other classes
    f1 = f1_score(y_true, y_pred, labels=[1], average='micro')
    return f1

# Create a custom scorer based on the custom metric
custom_scorer = make_scorer(custom_metric)

# Now you can use custom_scorer in model evaluation
```

Implement an Isolation Forest for anomaly detection.

Isolation Forest is an anomaly detection algorithm that's effective for identifying outliers or anomalies in a dataset. Scikit-learn provides an implementation of Isolation Forest that you can use for anomaly detection. Here's an example of how to implement and use Isolation Forest:

```
from sklearn.ensemble import IsolationForest
import numpy as np

# Create a sample dataset (2D data for simplicity)
np.random.seed(0)
X = 0.3 * np.random.randn(100, 2)
X = np.r_[X, 2 + 0.3 * np.random.randn(10, 2)]  # Add some anomalies

# Create and fit an Isolation Forest model
clf = IsolationForest(contamination=0.1, random_state=42)
clf.fit(X)

# Predict outliers (anomalies)
y_pred = clf.predict(X)

# Anomalies are labelled as -1, while inliers are labelled as 1
# You can convert the labels to boolean values (True for inliers, False for
anomalies)
is_inlier = y_pred == 1

# Print the results
print("Anomaly predictions (1 for inliers, -1 for anomalies):")
```

```
print(y_pred)
print("Is inlier (True for inliers, False for anomalies):")
print(is_inlier)
```

How can you perform oversampling of minority classes in imbalanced datasets?	Oversampling is a technique used to address class imbalance in datasets by increasing the number of instances in the minority class. This helps the model learn the minority class more effectively. There are various oversampling methods available. Here's how you can perform oversampling of the minority class in imbalanced datasets: 1. Random Oversampling: a. Randomly select instances from the minority class with replacement to increase their frequency. 2. SMOTE (Synthetic Minority Over-sampling Technique): a. Generate synthetic examples for the minority class by interpolating between existing instances. SMOTE creates new samples by selecting a minority class instance and its k-nearest neighbours and then creating synthetic samples along the line segments connecting them. 3. ADASYN (Adaptive Synthetic Sampling): a. ADASYN is an extension of SMOTE that adaptively generates synthetic samples based on the density distribution of minority class instances. It generates more synthetic samples in regions of lower density. 4. Borderline-SMOTE: a. A variant of SMOTE that focuses on borderline instances—those minority class instances that are near the decision boundary. It creates synthetic samples for these borderline cases. 5. SMOTE-NC (SMOTE for Nominal and Continuous Features): a. An extension of SMOTE that works with datasets containing both categorical and numerical features. It adapts the synthetic sample generation process for both types of features. 6. Random Oversampling with Replacement: a. This method randomly selects instances from the minority class and duplicates them to balance the class distribution. Here's an example of using the RandomOverSampler from the imbalanced-learn library, which is an extension of scikit-learn specifically designed for addressing class imbalance:

How can you perform oversampling of minority classes in imbalanced datasets?

```
from imblearn.over_sampling import RandomOverSampler
from collections import Counter

# Sample data
X, y = X_train, y_train  # Replace with your own data

# Initial class distribution
print("Class distribution before oversampling:", Counter(y))

# Apply random oversampling
oversampler = RandomOverSampler(sampling_strategy='auto', random_state=42)
X_resampled, y_resampled = oversampler.fit_resample(X, y)

# New class distribution after oversampling
print("Class distribution after oversampling:", Counter(y_resampled))
```

Write code to calculate the Kullback-Leibler divergence between two probability distributions.	You can calculate the Kullback-Leibler (KL) divergence between two probability distributions in Python using various libraries, such as NumPy. Here's an example of how to compute the KL divergence:

Write code to calculate the Kullback-Leibler divergence between two probability distributions.

```
import numpy as np

# Define two probability distributions (as NumPy arrays)
p = np.array([0.2, 0.4, 0.4])
```

```
q = np.array([0.3, 0.3, 0.4])

# Ensure that the distributions sum to 1
p = p / p.sum()
q = q / q.sum()

# Calculate the KL divergence
kl_divergence = np.sum(p * np.log(p / q))

# Alternatively, you can use scipy's entropy function to calculate the KL divergence
from scipy.stats import entropy
kl_divergence_scipy = entropy(p, q)

print("KL Divergence (numpy):", kl_divergence)
print("KL Divergence (scipy):", kl_divergence_scipy)
```

Build a stacked ensemble model using multiple base classifiers.

Building a stacked ensemble model involves training multiple base classifiers and then combining their predictions using a meta-learner. This ensemble technique can improve predictive performance. Here's an example of how to build a stacked ensemble model using scikit-learn:

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.ensemble import StackingClassifier, RandomForestClassifier,
GradientBoostingClassifier
from sklearn.linear_model import LogisticRegression

# Load a dataset (Iris dataset as an example)
data = load_iris()
X = data.data
y = data.target

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Define base classifiers
base_classifiers = [
    ('rf', RandomForestClassifier(n_estimators=100, random_state=42)),
    ('gb', GradientBoostingClassifier(n_estimators=100, random_state=42))
]

# Define the meta-learner
meta_learner = LogisticRegression()

# Create the stacked ensemble model
stacked_model = StackingClassifier(estimators=base_classifiers,
final_estimator=meta_learner)

# Train the stacked model on the training data
stacked_model.fit(X_train, y_train)

# Make predictions on the testing data
y_pred = stacked_model.predict(X_test)

# Evaluate the stacked ensemble model
from sklearn.metrics import accuracy_score

accuracy = accuracy_score(y_test, y_pred)
print("Accuracy of the stacked ensemble model:", accuracy)
```

Implement a t-SNE visualisation of high-dimensional data.	t-Distributed Stochastic Neighbour Embedding (t-SNE) is a dimensionality reduction technique commonly used for visualising high-dimensional data in lower dimensions. Here's an example of how to implement a t-SNE visualisation of high-dimensional data using scikit-learn and matplotlib:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.manifold import TSNE
from sklearn.datasets import load_iris

# Load a high-dimensional dataset (Iris dataset as an example)
data = load_iris()
X = data.data
y = data.target

# Apply t-SNE for dimensionality reduction
tsne = TSNE(n_components=2, random_state=42)
X_tsne = tsne.fit_transform(X)

# Create a scatter plot to visualise the reduced data
plt.figure(figsize=(8, 6))
plt.scatter(X_tsne[:, 0], X_tsne[:, 1], c=y, cmap='viridis', s=50)
plt.title("t-SNE Visualization")
plt.xlabel("Dimension 1")
plt.ylabel("Dimension 2")
plt.show()
```

How do you perform automatic feature selection using Recursive Feature Elimination (RFE)?	Recursive Feature Elimination (RFE) is a technique used for automatic feature selection in machine learning. It recursively fits the model with different subsets of features and selects the best-performing subset. scikit-learn provides the RFE class to implement this feature selection method. Here's how to perform automatic feature selection using RFE:

```
from sklearn.datasets import load_iris
from sklearn.feature_selection import RFE
from sklearn.linear_model import LogisticRegression

# Load a dataset (Iris dataset as an example)
data = load_iris()
X = data.data
y = data.target

# Create a base model (e.g., Logistic Regression)
model = LogisticRegression(solver='lbfgs')

# Create an RFE model
n_features_to_select = 2  # Number of features to select
rfe = RFE(model, n_features_to_select)

# Fit the RFE model to the data
rfe.fit(X, y)

# Get the selected features
selected_features = rfe.support_
ranking = rfe.ranking_

# Print the results
print("Selected Features:", selected_features)
print("Feature Ranking:", ranking)
```

Write code to apply quantile transformation to a dataset.	Quantile transformation is a technique used to map the data to a uniform or Gaussian distribution. Scikit-learn provides a QuantileTransformer class that you can use to apply quantile transformation to a dataset. Here's how to do it:

```
from sklearn.datasets import load_iris
from sklearn.preprocessing import QuantileTransformer

# Load a dataset (Iris dataset as an example)
data = load_iris()
X = data.data

# Create a QuantileTransformer
quantile_transformer = QuantileTransformer(output_distribution='normal',
random_state=42)

# Fit and transform the data
X_transformed = quantile_transformer.fit_transform(X)

# Print the transformed data
print("Original Data:")
print(X[:5])
print("\nTransformed Data:")
print(X_transformed[:5])
```

Implement a deep learning model CNN for malware classification. The dataset is called Spy.csv and it has 9503 columns in total where the last one is the label. Skip the first row	

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv1D, MaxPooling1D, Flatten, Dense
from tensorflow.keras.utils import to_categorical

# Load the dataset and skip the first row
data = pd.read_csv('Spy.csv', skiprows=[0])
X = data.iloc[:, :-1].values
y = data.iloc[:, -1].values

# Encode the labels
label_encoder = LabelEncoder()
y = label_encoder.fit_transform(y)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Reshape the data for Conv1D input (assuming each row represents a sequence)
input_shape = (X.shape[1], 1)
X_train = X_train.reshape(X_train.shape[0], X.shape[1], 1)
X_test = X_test.reshape(X_test.shape[0], X.shape[1], 1)

# Create a CNN model
model = Sequential()
model.add(Conv1D(filters=64, kernel_size=3, activation='relu',
input_shape=input_shape))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(64, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

```
# Compile the model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

# Train the model
model.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_test,
y_test))

# Evaluate the model
loss, accuracy = model.evaluate(X_test, y_test)
print(f"Test Loss: {loss:.4f}")
print(f"Test Accuracy: {accuracy * 100:.2f}%")
```

Implement a deep learning model LSTM for malware classification. The dataset is called Spy.csv and it has 9503 columns in total where the last one is the label. Skip the first row

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Embedding
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.utils import to_categorical

# Load the dataset and skip the first row
data = pd.read_csv('Spy.csv', skiprows=[0])
X = data.iloc[:, :-1].values
y = data.iloc[:, -1].values

# Encode the labels
label_encoder = LabelEncoder()
y = label_encoder.fit_transform(y)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Preprocess the data for LSTM input
max_sequence_length = 100  # Set an appropriate sequence length
X_train = pad_sequences(X_train, maxlen=max_sequence_length)
X_test = pad_sequences(X_test, maxlen=max_sequence_length)

# Create an LSTM model
model = Sequential()
model.add(Embedding(input_dim=X_train.max() + 1, output_dim=128,
input_length=max_sequence_length))
model.add(LSTM(64, return_sequences=True))
model.add(LSTM(32))
model.add(Dense(1, activation='sigmoid'))

# Compile the model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

# Train the model
model.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_test,
y_test))

# Evaluate the model
loss, accuracy = model.evaluate(X_test, y_test)
print(f"Test Loss: {loss:.4f}")
print(f"Test Accuracy: {accuracy * 100:.2f}%")
```

Build a multi-layer perceptron (MLP) neural network for malware classification. The dataset is called Spy.csv and it has 9503 columns in total where the last one is the label. Skip the first row

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Load the dataset and skip the first row
data = pd.read_csv('Spy.csv', skiprows=[0])
X = data.iloc[:, :-1].values
y = data.iloc[:, -1].values

# Encode the labels
label_encoder = LabelEncoder()
y = label_encoder.fit_transform(y)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Create an MLP model
model = Sequential()
model.add(Dense(128, input_dim=X_train.shape[1], activation='relu'))
model.add(Dense(64, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

# Compile the model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

# Train the model
model.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_test,
y_test))

# Evaluate the model
loss, accuracy = model.evaluate(X_test, y_test)
print(f"Test Loss: {loss:.4f}")
print(f"Test Accuracy: {accuracy * 100:.2f}%")
```