Foundations of Artificial Intelligence
Joel Mattsson - Assignment 2
2024/2025

1. Problem Introduction

This report covers different types of classifiers that are trained on a dataset that serve the purpose of classifying or predicting data as one of the labels, given the data's input features. In the scope of this report, where the objective is to classify handwritten digits to their corresponding actual number, the possible classes of classification are therefore limited to the number of distinct digits in the provided dataset. Consequently, the classifier models must be able to detect the unique contours for each digit, to accurately distinguish between the possible outcomes. These contours, or implicit features associated to each class or label, must be recognized by the models so that they can map to the correct classes, and this involves two aspects:

- Dataset - The input aspect: First of all, if the images of the dataset is erroneous, faulty or blurry, or in some way hides the key contours, then the classifier model will never return accurate output, regardless how good its inherent algorithm is. In other words, the models are relying on their input: if the dataset it is fed is irrelevant to the problem to solve, the model can impossibly yield beneficial output
- Classifier model - The computational and output aspect: Now, let's assume that the input dataset is valid and provides sufficient clarity for the model to discern between the digits. At this stage, the model receives the input and is expected to make conclusions in accordance with the insights and patterns derived from the data. In this way, the dataset is crucial in the making of a great classifier.

The second aspect (the classifier model) will be addressed in detail in later sections, as multiple classifiers are implemented with respective results and evaluations. However, in regards to the dataset aspect, the fundamentals will be emphasized here as a problem introduction, as its structure and content ties into the theoretical background of the approaches. In today's day and age, there are a vast amount of databases or datasets to retrieve data from, whether the developer aims to simply observe trends, highlight business insights or predicting the future. Regardless of one's field of expertise within the broad domain of programming, it would take a lot of time if we were to put together our own set of data. Obviously, it's always possible to manually create data of a few instances on our own, but the question then becomes:

- *How accurately is the classifier model able to respond, based on the dataset in which its foundational mapping is formed, if it is strictly limited by quantity and quality?*

The natural question that emerges now is:

- *What constitutes a good dataset? What would be the ideal input to give a classifier model so that it can perform its classification algorithm with the highest accuracy?*

In this project, considering the scope of using images of handwritten-digits as elements in the dataset, the relevant key features to observe are:

- Sampling: Many samples or instances yield great benefits. Similar to interviews or polls, the more individuals or samples involved, the more reliable, accurate and representative the results are assumed to be of the overall population

- Variety: Many different examples, not just a large amount of the same type. It's important that different classes or labels are involved so that they can be compared to each other, making it easier to distinguish between them in a systematic way
- Quality: This concerns the quality of the data as in how obvious their features are. If there's too much noise the model may interpret the data incorrectly and start operating on faulty feature assumptions about a class. Data cleaning, outliers and publicly verifiable datasets are things to keep an eye on to maximize the quality

This is where the decision to use the MNIST database was made. As it more than satisfies all of the bullet points above and is well-known and verified in the sense that all instances exceed a certain threshold of quality.

1.1 MNIST Database

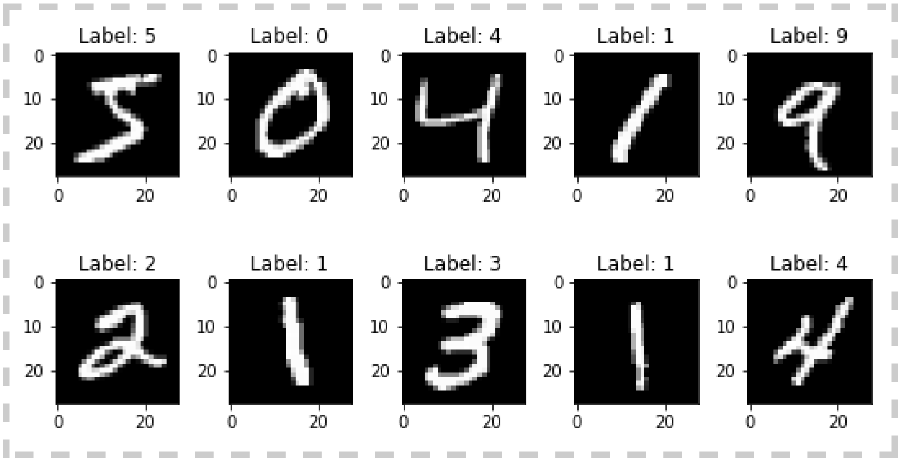
The MNIST (Modified National Institute of Standards and Technology) database contains digits that have imperfect contours that simulate handwritten ones, and it's used for training and testing models in computer vision and machine learning. Now, let's evaluate this database against the bullet points in the previous section to discover how appropriate the usage of this database is, in the context of the given classification problem of this project.

As discussed in the previous section, it's important that the dataset satisfies..

- Sampling:
  - For a database or dataset to provide reliable insights, it must contain a larger number of instances that indicate through an underlying pattern that a statement is true. If 100% of the contained data points in a small dataset of 10 rows points to the fact that the most popular politician of a nation is A, one starts questioning how representative this is for the entire national population. On the contrary, if only 60% of the data points in a larger dataset of one million rows indicates that politician B is likely to win the election, it has a higher significance. The data is taken more seriously since it's confirmed to be true for a larger percentage of the target group
  - The MNIST database does not disappoint in the field of quantity and sampling. In this project, "mnist_784" was used, which offers a training set of 60000 images and a test test of 10000 images.
- Variety:
  - All images are labeled within the range 0-9 (all possible classes involved)
  - The distribution of the database is the frequency of each of these 10 digits or classes in the dataset. As mentioned, there are 60000 instances in the training set, and due to the even distribution there are roughly 6000 occurrences of each digit. Similarly, the test set of 10000 images has about 1000 appearances of each class. This uniform distribution is great since it ensures consistency of the model's performance for all classes. If, for instance, the training set had a distribution of 59000 images for digit *4* and 100 images each for the nine other

digits, then the model wouldn't be as familiar with recognizing the features of them, ultimately resulting in incorrect output

- Quality:
  - The digits are deliberately imperfect to simulate hand-written ones. However, to maintain consistency for the model to read the features, all images are of 28x28 pixels and represented as vectors of 784 elements in the form of a grayscale. This is vital, because if the sizes and formats differ, the model interprets the features in different ways, causing inconsistencies in its way of adapting and recognizing patterns
  - Images are stored based on pixel intensity values ranging from 0 to 255, which then is mapped or normalized to the interval [0, 1] for simplicity for machine learning tasks
  - The MNIST database is also reliable and verifiable. PyTorch and Tensorflow, two of the most influential Python libraries in the field of AI and ML, include built-in utilities to fetch the MNIST database. Imagine if the database wasn't known by anyone. How would one know that all of the thousands of hand-written images aren't sufficiently interpretable for the model nor too close to the perfect digits written by a computer? If the images are too blurry it just confuses the model and no feature-patterns can be adapted. On the contrary, too "perfect" images leave no room for the model to compare imperfect contours and identify features. Conclusively, to ensure quality of a database, all instances must conform to specific standards with respect to the model that will interpret this data.



In this project, the implemented code fetched from the MNIST database with two declared variables:

- X: A structured 3-dimensional array of each 28x28 image. Note that there are 600000 images in the training set
- Y: An array of digits (0-9) connected to the corresponding image. This is also of length 60000

The MNIST database, being the broad and well-known database it is, offers datasets for both supervised and unsupervised learning. Keeping in mind that the purpose of this project is to implement handwritten digit classifiers and that it is a classification problem, not all datasets used. By virtue of the nature of classifiers and the challenges they face, where they inspect features and categorize them in a predefined class, supervised learning is more beneficial in this context. These notations along with the classification challenges and approaches are discussed further in the next section.

2. Theoretical Background

This section serves the purpose of providing a broad notion of brief introductory knowledge, so that you can understand why the specific approaches were chosen and how they systematically solve the problem addressed in the previous section. Only theoretical information will be covered here, and these are limited to the topics of machine learning models and classifiers. However, this information is applied in the next section "Approaches".

2.1 Machine Learning Models

In this project, machine learning models are needed to construct the classifiers. This is to achieve the ability of being able to classify an instance that wasn't explicitly programmed into the model. In other words, a "complete" classifier should be able to classify all instances, even if they weren't necessarily used to train the model. This makes sense because there are millions of possible combinations if one were to tweak every inherent feature to represent all possible types. Consequently, this would also require massive datasets and allocation of space, indicating significant inefficiency. By virtue of this, the implementation of machine learning models in the context of this project is crucial. In essence, their core purpose is to recognize patterns of data and make predictions or decisions. Due to their ability to make connections, the model extrapolates from patterns of the training data to carry out tasks beyond what was included in the input dataset. In other words, a valid model is able to encounter similar situations without being explicitly programmed for them. In the sections below, a few topics of machine learning models will be presented. Obviously, the topic of machine learning models is a broad one, and therefore many subtopics are excluded. For simplicity, only the most prominent and impactful ones to classification problems will be discussed.

2.1.1 Unsupervised vs Supervised Learning

All machine learning models must use data as input to learn how to process, manage and operate as intended. There are two main ways of structuring this input data, but their structure vastly differentiate and are used in separate scenarios as they force the machine learning model to adapt patterns in unique ways:

- Supervised Learning: Trained on labeled data in a pairwise manner. The input instance is coupled with its corresponding correct output. In this way, the model receives an already established blueprint for recognizing these patterns
- Unsupervised Learning: The data isn't labeled, and the objective is for the model to learn by discovering and exploring on its own rather than learning to recognize predefined classes

In summary, due to the natural structure of classification problems, where a data instance must be categorized based on inherent features, supervised learning techniques are more beneficial. Since the model must compare the features to explicit classes and their patterns, in the sense that there's a strict pattern unique to each class, a pairwise labeling helps distinguish accordingly. Either a data instance aligns or is incompatible with a certain class. And this is what separates it from unsupervised learning, where the model explores new underlying patterns. Conclusively, as all approaches or implementations in this project are classifiers and intend to solve a classification problem, they are utilizing supervised learning to refine their models, fetching from MNIST database.

2.1.2 Generalization Error

Machine learning models may generate incorrect output that has a general conjunction or relation to the expected output for the entire dataset. This general discrepancy quantifies how well a model adapts by the training data, which is tied to its performance on unseen data. Thus, the objective of machine learning models is to minimize these general errors to improve accuracy. These errors may emerge in different cases:

- Training Phase: As mentioned in the "Problem Introduction" section, the input of the machine learning model is important. If the dataset instances obscures information from the model by low quality, sampling or variety the general accuracy of the model decreases
- Overfitting: Occurs when a model is too complex for the problem, such that it absorbs noises and irrelevant patterns in addition to the actual or desired pattern. Ultimately, this results in poor and inaccurate performance
- Underfitting: Occurs when a model is too simplistic for the given problem, such that it struggles to capture underlying patterns and fails to produce accurate results

It's hard to fully avoid all of these problems at once, since there's a balance between overfitting and underfitting. Nevertheless, it's possible to evaluate the magnitude of these types of errors to gain an understanding of how well one's model is performing. To do this, there is a variety of different methods:

- Cross validation: Splits a dataset into different folds or subsets and estimates the performance of a model on independent datasets. This is implemented for all classifiers in the "Approaches" section

- Hyperparameter Tuning: Select or adjust the model parameters to better fit the problem and ultimately optimize performance. This is implemented for all SVM classifiers in the "Approaches" section

2.1.3 Cross Validation

Cross validation is a method used to assess the performance of machine learning models when faced with unseen data. This assessment helps prevent generalization errors such as overfitting and underfitting and provides an overall estimate of how it will perform on real-world data. This estimate is reliable due to the fact that the dataset is split into multiple folds to separate training- and validation- subsets of the original dataset. This process is done multiple times, allowing all data instances to be in both sets to increase the data diversity. In contrast, an arbitrary selection of data allocation for both sets only once won't yield a result as accurate, since it varies based on the inherent features of the data instanced in the selected sets. On the other hand, by mixing them in the combinatorial way of cross validation, the results become more reliable and representative of the actual performance as it represents the execution performance for multiple different sets. Of course, this requires more time and resources. Implementing cross validation is a tradeoff between time and space complexity versus a reliable measure. From one perspective, avoiding cross validation allows for quick debugging and saves the developers time. Conversely, the reliable measures that cross validation brings enables true insights, which allows one to make accurate data-driven decisions. Moreover, conclusions derived from unreliable sources cannot be ensured to be correct, since there is a chance that the source itself is incorrect. The question then becomes:

- *What is the core purpose of this particular project? Is it to quickly put forth a demo that lacks precision but has practicality? Or is it to make impactful decisions that affect the lives of other people, such that a precise measure is needed to calculated and secure conclusions?*

2.2 Classifiers

The term 'Classifier' is a broad term, and it refers to classifying something based on its displayed features. In the world, there are many different types of a specific thing, where each type has a few shared common attributes and some which are completely unique to them. To classify something is to be able to distinguish between these different types by observing their distinguishing features and then categorizing them to the corresponding class. We as humans were born with eyes that can clearly discern between multiple different types within a split second:

- *We can easily see which type of soda the person next to us is drinking: Cola or Sprite?*
- *We can easily see which type of flower is in front of us: A or B?*

However, our human capabilities are only so limited and our eyes can only detect different types and articulate their differences one at a time. This then begs the question:

- *If this process can be automated via code, such that it takes vast datasets of different types, then how efficient and how much time would be saved?*

When looking at how this code solution would be implemented from a high-level perspective, it appears that machine learning models are a must for this solution to work out and yield beneficial results. Since only using predefined and static measures to compare the data against will never be representative enough, as each instance of a type is inherently unique. There is no exact range of values that make up for a type, but rather the overall combination of features that sets it apart. And this calls for a need to read all patterns and construct a neural network to best represent and map out the differences. Below, different types of classifiers will be briefly explained:

2.2.1 Linear Classifiers

This type of classifier gets its name from its linear decision boundary that separates data by a straight line. In essence, linear classifiers predict the label of a class by determining which side of the decision boundary the point lies on. The decision boundary is a linear function and is mathematically defined as:

$$f(x) = w * x + b$$

Where..

- ***f(x)*** *is the numerical position of the data point*
- ***w*** *is the weight vector coefficient*
- ***x*** *is the vector of the data point*
- ***b*** *is the bias term*

Conditions..

- ***f(x) > 0*** *implies that data point* ***x*** *belongs to class* ***A***
- ***f(x) < 0*** *implies that data point* ***x*** *belongs to class* ***B***

As a result of the simplicity that comes with a linear hyperplane, the prominent feature of these classifiers is their computational efficiency. They are therefore applicable to more complex datasets, however, as the name suggests, linear classifiers are only ideal in the case of linearly separable data. And in many cases the data points may be distributed in a more complex way, such that the classes partially intersect with each other. For these types of problems, non-linear classifiers provide great solutions, which is covered in the next section.

2.2.2 Non-Linear Classifiers

Unlike linear classifiers that are only useful on linearly separable data, non-linear classifiers create more complex decision boundaries with curves that better fit the complexities. Non-linear classifiers can process complex relationships and can as a result produce high accuracies, but their ability to capture more complex relationships between features and labels also cause a need for more resources, time and memory. In other words, there is a tradeoff between non-linear and linear classifiers. Whereas non-linear have the potential of achieving a higher accuracy, they are also more computationally costly.

2.2.3 Probabilistic Classifiers

Probabilistic classifiers predict probabilities for all possible outcomes as opposed to just a single label. Beyond predicting which class an instance belongs to, these classifiers also predict the confidence in that prediction. The confidence value is a measure of the certainty for a prediction to be true, and thus a metric for the algorithm to prioritize which ones are likely to be true. This probabilistic approach makes it applicable to many real-world problems, as our world is influenced by many external circumstances out of our control, manifesting themselves as potential risks that affect the confidence of the probabilities. Any form of planned event in the future cannot be guaranteed to live up to its expectations. Perhaps the star employee suddenly gets severely sick, forcing the company to postpone the deadline of a project. Or perhaps symptoms indicate an upcoming medical disease but it turns out it was a hard fever. Whether it be the risk of not being able to follow a deadline, or catching a disease, or in other situations of life, there is always a risk of something happening. In the end, nothing has a 100% probability, unless we are aware of all the millions of factors influencing it. It would therefore serve as guidance or a roadmap to make successful decisions, if we could find a way to approximately quantify the certainty of a prediction.

3. Approaches

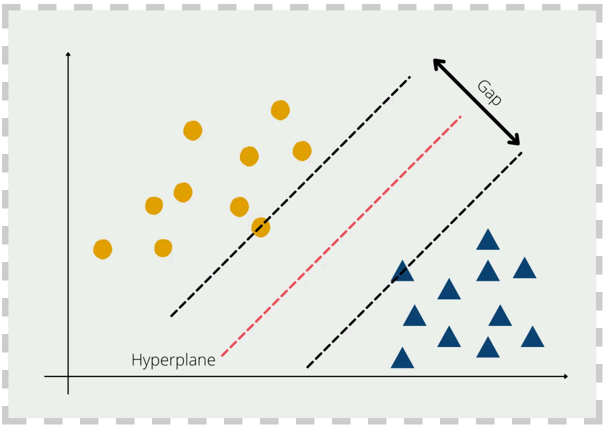
This section concerns the code implementations of a multitude of different classifiers, all of which relates to the classification problem of detecting hand-written digits from the MNIST database.

3. 1 Support Vector Machines (SVM)

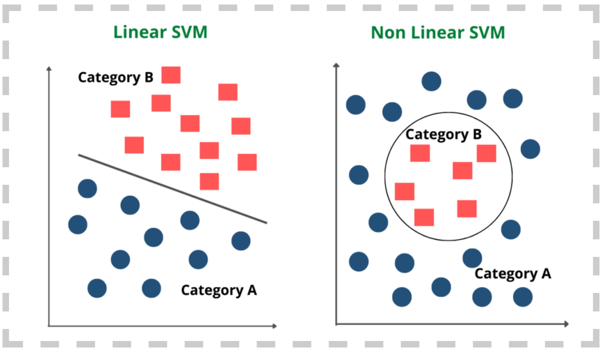
In this report, three different types of support vector machines were implemented. These are distinguished by their kernels, which in essence constitute the transformation of data into a higher-dimensional space and the computation of relationships between data points in this space. However, as all of these implementations are categorized as support vector machines, they have a few characteristics in common. In essence, the three core components of this SVMs are:

- Hyperplane: Refers to the line, or decision boundary that separates data into classes. This is useful since classification problems rely on returning a final decision on what class a particular instance belongs to. In essence, if a data point falls into one decision boundary, it is then classified as the corresponding type or class

- <u>Support Vectors:</u> Refers to the data points closest to the hyperplane. As they are the closest to the decision boundary, they also are the most important ones to observe, since they are closer to the other classes. In other words, their inherent features are "overlapping" more with similar classes, making it harder to maintain a high accuracy of the correctness of the classification
- <u>Margin:</u> Refers to the distance between the hyperplane and the support vectors (the closest data points). Margins, in this context, essentially refers to the allowed margins of errors. The bigger the margin, the more inaccurate a SVM model can afford to be while still outputting correct classification results. Thus, maximizing the margins between the hyperplane and the support vectors is a fundamental concept for improving the accuracy.



As addressed in the previous section "Theoretical Background", there exist different types of SVMs, all of which exhibit their own advantages and setbacks. The ones that were implemented will be discussed from a theoretical perspective below, and they are divided into two types: linear and non-linear.



3. 1. 1 SVM Linear Kernel

The Support Vector Machine with a linear kernel is one that falls into the category discussed in the "Linear Classifiers" section earlier. As already established, this kind of classifier uses a simplistic straight line to distinguish between classes and define decision boundaries. This type of SVM utilizes its simplicity to obtain efficient computations. In comparison to non-linear kernels, such as the polynomial one, this type of kernel does not transform data into a costly higher-dimensional space due to its simplistic linear parameters. Hence, linear kernels operate in the given or original dimensional feature space, which allows for more efficient but light-weight computations. An additional cause for this classifier's high efficiency, in comparison to the other types of kernels, lies in the dot product computation between the vectors. By virtue of the kernel's linearity, it doesn't involve exponential parameters that account for multiple dimensions, which induce detrimental effects on the time complexity. However, it's worth noting that this type of classifier only works properly on linearly separable data. Then, the next questions that naturally emerge are:

- *How impactful or efficient is SVM with linear kernel on the datasets it is supposed to be applicable to (the linearly separable datasets)?*
- *Is the linear SVM applicable to datasets that aren't linearly separable, if so, what would be an estimated approximation of the percentage of these datasets? Or to what degree of non-linearity would these datasets embody, such that the linear SVM would yield a beneficial result?*
- *On the contrary, is SVM with linear kernels even applicable to non-linear datasets? Would they yield a somewhat beneficial outcome to derive important insights from? Or would they just be incompatible and impossible to derive insights from?*

Lets first observe cases where linear SVM partially solves non-linear data to properly grasp how this is possible (it's worth noting that not ALL data for all conceivable instances in these topics are non-linear, but in general these topics tend to emulate a non-linear pattern):

- Financial Forecasting: The stock market data is volatile and is rarely manifested as a linear predictability. Due to its unpredictability, the derived data is often non-linear and spiky, making it hard to classify in a linear manner. For instance, a company in 2017 may have displayed inherent features indicative of a rapid growth in the next 5 years, but then COVID-19 resulted in the company filing for bankruptcy. In other words, an external circumstance impacted the data such that it violated its linear predictable pattern. As a result, the company would not be classified as a successful one
- Sports Analytics: The athlete gets injured one day before the big finale and thus can't perform according to his usual standards. As a result, he finished 3rd instead of 1st, even though his personal best and average ratings are significantly better than the two athletes that beat him in this particular competition

All of these domains are examples of broad industries that typically deal with non-linear data, which would challenge the linear SVM's accuracy and quality. Now, to make up for the inefficiency that would emerge by applying a linear SVM to these domains, we can make additional modifications or adjustments to the linear kernel, allowing it to perform well despite its limitations on non-linearities. A few of the possible modifications to the linear SVM are:

- **Soft margin:** The model operates with misclassifications to appropriately adjust the balance between maximizing the margin and minimizing classification errors
- **Transformations for Approximation:** Applying additional transformations to non-linear datasets to be able to linearly separate the classes:
  - Scaling and normalization of entire dataset: Performing unifying operations for the entire dataset to restructure the data points accordingly. This could simplify the overall distribution of data points, making it easier to distinguish classes by a linearity
  - Perform square-root or logarithmic operations on features: Extract mathematical relationships and simplify them for all the data points. This simplification may embody a shape closer to a straight line

Conclusively, linear SVMs can to some extent produce sufficiently accurate results on non-linear datasets. Of course, in general, the higher the degree of non-linearity of a dataset, the lower accuracy the linear kernel will produce. Nevertheless, the ideal circumstance for this type of classifier is a dataset input whose data points clearly can be separated by a straight line.

3.1.2 Non-Linear SVMs

What is unique for non-linear kernels is their kernel functions to transform data into a higher-dimensional space and adapt a hyperplane. As mentioned in the previous section "Theoretical Background", there are multiple non-linear kernels. Two of them were implemented in this project and are discussed below.

3. 1. 2.1 SVM Polynomial Kernel

As the name implies, this non-linear SVM uses a polynomial approach to solve the provided problems. In this particular approach, the polynomial of degree 2 was only implemented as demonstrated later in the "Results" section, but this will briefly compare the polynomials to grasp how the complexity generally increases as the dimensionality is incremented. What is important to note before comparing the different polynomials is the following:

- As opposed to linear kernels that completely skips the process of transforming data into a higher-dimensional space, the polynomial operates in higher-dimensional spaces, which increases the complexity and the resources required
- As the name suggests, higher numbers than 2 are allowed as the exponential parameter. Of course, this increases the complexity for each kernel evaluation and the computational resources required
- The general formula for the kernel function is:

$$K(x,y) = (x * y + c)^n$$

- *Where..*
  - ***x** and **y** are the input data points from MNIST database that the kernel uses to compute the classifications*
  - ***c** is the coefficient added to the dot product and helps shifting the decision boundaries without changing their computed shapes*
  - ***n** is the dimension or polynomial degree and dictates the model's complexity. Hence, choosing too high value for **n** results in redundant complexity and computations to the associated dataset, often leading to overfitting. Conversely, too low value for **n** prevents it from capturing enough details, instead causing underfitting*

Now, let's examine the impact of the different polynomial degrees:

- n = 2: *In essence, the difference between the first and second degree is their complexities in their decision boundaries. A lower degree yields simpler patterns and lowers the time complexity, and is therefore more suitable for larger datasets. Using high degrees to process massive amounts of data requires more time and memory*

- n >= 3: *As **n** increases, the risk of overfitting for smaller datasets also increases. Simultaneously, bigger values for **n** makes it harder to apply large datasets in an efficient manner. Although the interpretability decreases, a higher dimension makes it suitable for tasks that require flexibility. Higher degrees may also be trickier, in the sense that it's hard to discover the underlying degree of the irregular curves of the hyperplane. Thus the utilization of higher degrees usually needs to be combined with methods providing metrics of how optimal the chosen degree is to the particular dataset. For instance, as discussed in the previous section "Theoretical Background", cross validation is one approach for this*

3. 1.2.2 SVM Radial Basis Function Kernel

Just like the polynomial kernel, the RBF kernel operates in higher-dimensional feature spaces, which enables it to solve non-linear classification problems. As addressed in the previous section, the polynomial kernel is more flexible than the linear since it allows for more dimensions. Similarly, the RBF kernel is more flexible than the polynomial in the sense that no explicit specification of the degree or type of relationship is needed. This type of kernel handles a wider variety of non-linear problems by automatically adjusting to rather than assuming a specific type of relationship. The kernel function is as follows:

$$K(x, y) = \exp(-\gamma \|x - y\|^2)$$

- *Where..*

- ***x** and **y** are input data points*
- *γ or **gamma** is a hyperparameter governing the impact of training samples*
- *$||x - y||^2$ is the squared Euclidean distance between **x** and **y***

Considering the mathematical definition of the RBF kernel, there are a few conclusions to be drawn about its prominent properties:

- Similarity Measure: If x and y are close, K(x, y) is closer to its maximal value of 1, and indicates a high similarity between the two variables. Likewise, a larger distance between x and y results in a lower value of K(x, y) that approaches zero, indicating a lower similarity
- Unlimited dimensional space: The RBF kernel maps the inputs (x and y) into a dimensional space with an unlimited number of dimensions. As opposed to the polynomial kernel, where the developer has to explicitly specify the dimension prior to the execution, the RBF kernel implicitly maps the features without explicitly computing the mapping without imposing restrictions
- Gamma: This hyperparameter helps regulate the cases of overfitting and underfitting. Higher values of gamma entails a concentration on nearby local data points, which results in flexible and irregular decision boundaries, or also known as overfitting. In contrast, decreasing gamma causes an expansion of the region of focus such that the algorithm considers points further away. This often leads to the case of underfitting.

3. 2 Random Forests

This classifier is based on decision trees and their collective wisdom to reduce overfitting and enhance prediction accuracy. It constructs multiple decision trees and systematically merge their outputs to generate optimal predictions. From the input dataset, it generates several subsets in a randomized fashion, hence the denotation “random”. A decision tree is trained for each of these subsets so that they are “grown independently”, simulating the property of a forest. Random Forests are known for their ability to achieve a high accuracy, and they follow a number of sequential steps to live up to their standards:

1. Sampling: In a randomized manner, select subsets of the training dataset to train individual trees
2. Feature Randomness: For each split, a randomized subset of features is considered to constitute the highest ranked split
3. Tree Construction: Use a random subset of data and features to construct the tree used for decisions
4. Aggregation: Aggregate or summarize the results into a comprehensible and structured format. In essence, this concerns the class predictions for all trees

Due to the model’s complex architecture of trees, it inevitably struggles with solving simpler problems, and the case of overfitting is achieved. To counteract this scenario, only a random subset of features is considered within a tree. Ultimately, the classification stage relies on the trees to combine their predictions to produce more accurate results. This is done by allowing each tree to make a prediction by voting for a class, and assigning the final output as the class with the most votes. In addition, the algorithm's performance is indifferent to the size of the given dataset. Unlike non-linear SVMs that have a tendency to significantly decrease in performance due to their higher dimensionalities, Random Forests and their trees from subsets remain more consistent as each tree is splitted by feature randomness to balance the overfitting.

3. 3 Naive Bayes

This classifier is categorized as a probabilistic one, as mentioned in the "Theoretical Background". It is probabilistic in the sense that it relies on calculations of probabilities for each class, and then selects the class with the highest probability. However, it is denoted as "Naive" due to its approach of assuming an independence of all input variables. Nevertheless, this is a rare case since most classes are in one way or another influencing the opposing class' values. Although this assumption facilitates the calculations, it can in turn cause inaccuracies. The algorithm uses Bayes' Theorem to reason under uncertainty and combine prior knowledge with new evidence in a continuous fashion throughout the algorithm's runtime. In essence, it updates the current probability of a previously defined hypothesis. Considering the problem statement of this project, where classification of the MNIST database is the focus, this algorithm would analyze each feature of a data instance and continuously update the confidence or probability of it belonging to a certain class. Additionally, Bayes is generally suitable for classification problems of multiple classes, extending beyond the binary-class limitation and effective for high-dimensional datasets. As for the implementation, *Alpha* and *Beta* parameters were used to balance the properties of the classifier. Both of these parameters modifies the behavior of the model in their own unique ways:

- Alpha: Is used to handle the case of unseen data and smoothes probabilities. As this variable approaches infinity, the influence of uniform prior knowledge for all features increases
- Beta: As mentioned this classifier takes the prior knowledge and combines it with new evidence. The *Beta* parameter controls the influence of the prior knowledge. Thus, smaller values implies that more of the previous probabilities are dismissed such that the current or observed data has more impact on the final output. On the other hand, larger values makes the prior probabilities dominate the data

Conclusively, each pixel of the images fetched from the MNIST database is distributed in accordance with the *Beta* distribution, and in this way the element of prior knowledge versus new evidence is incorporated in this approach. Also, the naive assumption about independence may lead to errors in terms of the output, as the performance degrades for highly correlated features.

3. 4 k-NN

The K-Nearest Neighbors classifier is primarily used for classification tasks and processes data points based on their relational similarity. It calculates the distance between the specific point and the other instances in the dataset. This distance is equivalent to a metric derived from an algorithm or a formula such as Manhattan distance or Euclidean distance. Now, the next step is what signifies the name of this very classifier. Essentially, it filters out all data points except for the k nearest neighbors. In other words, the data points whose distance value is lowest in the metric are the ones that will be included for later computations. Keep in mind that this pattern is applied to each and every data point, such that each point is mapped to its resulting k nearest neighbors. As it appears, this approach may be inefficient due to its repetitiveness. Iterating over each data instance and applying this sequential procedure is detrimental to the time complexity. In turn, performing this classifier algorithm on large datasets, such as MNIST database, where computations for each data point pair are made, ends up being very time consuming. Hence, the choice of *k* has a substantial impact on the performance. But more importantly, *k* also determines how "complete" the classification evaluation is, in the sense that the more data points involved, the more accurate the final output is. Taking this into consideration, the implications are:

- <u>Smaller values for *K*:</u> Excluding more neighboring data points implies an elevated sensitivity and captures more details locally. As a result redundant noise may be involved in the computations, contributing to overfitting
- <u>Bigger values for *K*:</u> Extending the number of neighbors means expanding the area of data points that influence the classification. As a result the decision boundaries manifest smoother shapes. In the end, this approach may overlook the local patterns of the closest neighbors and thus suffer from the concept of underfitting

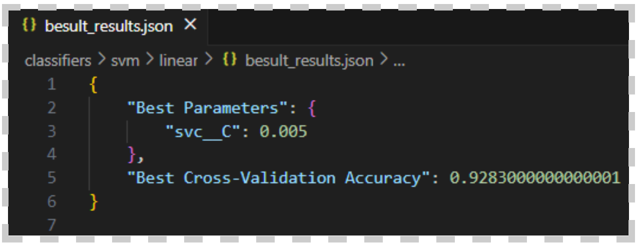
Now, the final step of the k-NN classifier is to assign the best possible class label for each data point. As opposed to the support vector machine with a linear kernel that classifies based on what side of the hyperplane an instance lies at, k-nn leverages on its distance-mapping by assigning the final class label that is most frequent for the k nearest neighbors, for a data point. Note that the two bullet points above apply to the k-NN algorithm when using uniform weighting, such that each neighbor has equal influence in the voting process. However, it is also possible to use distance-based weighting. This is done by assigning more influence to neighbors the closer to the current data point. In other words, the smaller the distance, the more voting power it has. To avoid the cases of overfitting and underfitting this concept of balancing votes by distance can be helpful. But for this project, uniform weighting was used in the implementation for simplicity.

4. Results

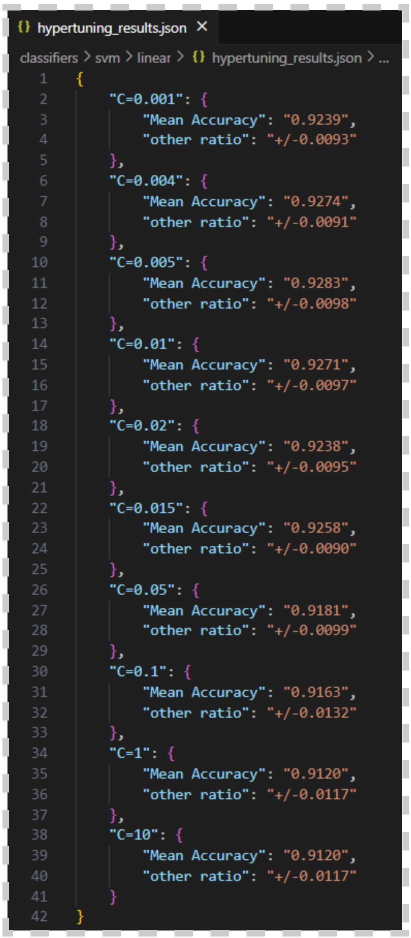
This section solely intends to present the results briefly. The results below were obtained through JSON files generated during runtime for each of the approaches described in the previous section. The rationale and conclusions derived from these insights will be presented in the next section “Conclusions”. As for this section, each subsection or header represents the respective implementation of the approaches mentioned in the previous section “Approaches”.

4.1 SVM - Linear Kernel

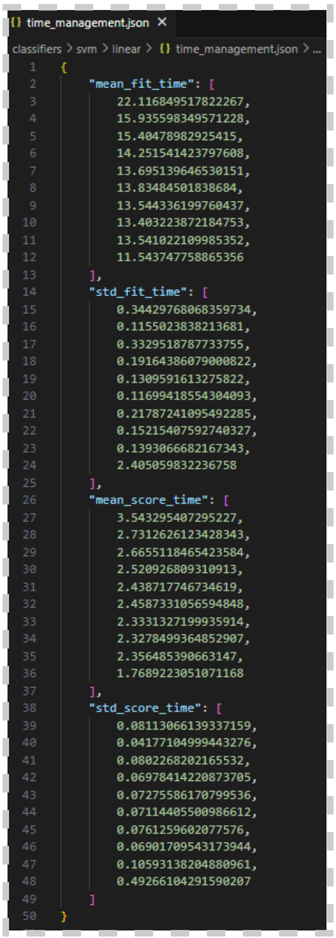
Below, the result of the linear kernel SVM is displayed in three different JSON files:



```json
{
    "Best Parameters": {
        "svc__C": 0.005
    },
    "Best Cross-Validation Accuracy": 0.9283000000000001
}
```



```json
{
    "C=0.001": {
        "Mean Accuracy": "0.9239",
        "other ratio": "+/-0.0093"
    },
    "C=0.004": {
        "Mean Accuracy": "0.9274",
        "other ratio": "+/-0.0091"
    },
    "C=0.005": {
        "Mean Accuracy": "0.9283",
        "other ratio": "+/-0.0098"
    },
    "C=0.01": {
        "Mean Accuracy": "0.9271",
        "other ratio": "+/-0.0097"
    },
    "C=0.02": {
        "Mean Accuracy": "0.9238",
        "other ratio": "+/-0.0095"
    },
    "C=0.015": {
        "Mean Accuracy": "0.9258",
        "other ratio": "+/-0.0090"
    },
    "C=0.05": {
        "Mean Accuracy": "0.9181",
        "other ratio": "+/-0.0099"
    },
    "C=0.1": {
        "Mean Accuracy": "0.9163",
        "other ratio": "+/-0.0132"
    },
    "C=1": {
        "Mean Accuracy": "0.9120",
        "other ratio": "+/-0.0117"
    },
    "C=10": {
        "Mean Accuracy": "0.9120",
        "other ratio": "+/-0.0117"
    }
}
```

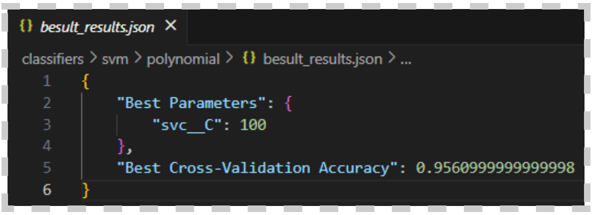


```json
{
    "mean_fit_time": [
        22.116849517822267,
        15.935598349571228,
        15.40478982925415,
        14.251541423797608,
        13.695139646530151,
        13.83484501838684,
        13.544336199760437,
        13.403223872184753,
        13.541022109985352,
        11.543747758865356
    ],
    "std_fit_time": [
        0.34429768068359734,
        0.1155023838213681,
        0.3329518787733755,
        0.19164386079000822,
        0.1309591613275822,
        0.11699418554304093,
        0.21787241095492285,
        0.15215407592740327,
        0.1393066682167343,
        2.405059832236758
    ],
    "mean_score_time": [
        3.543295407295227,
        2.7312626123428343,
        2.6655118465423584,
        2.520926809310913,
        2.438717746734619,
        2.4587331056594848,
        2.3331327199935914,
        2.3278499364852907,
        2.356485390663147,
        1.7689223051071168
    ],
    "std_score_time": [
        0.08113066139337159,
        0.04177104999443276,
        0.0802268202165532,
        0.06978414220873705,
        0.07275586170799536,
        0.07114405500986612,
        0.0761259602077576,
        0.06901709543173944,
        0.10593138204880961,
        0.49266104291590207
    ]
}
```

These results indicate that, when setting the *C* parameter to *0.005*, the highest accuracy is achieved in the 10 way cross validation, which is 0.928, approximately 93%. Also, the results of the hypertuning for different values of *C* is also shown, enabling a more comprehensive examination of the impact of *C*. Lastly, the time management aspect is also observable, breaking down the *fit* and *score* times in sequential order by corresponding 10-way-fold.

4.2 SVM - Polynomial Kernel

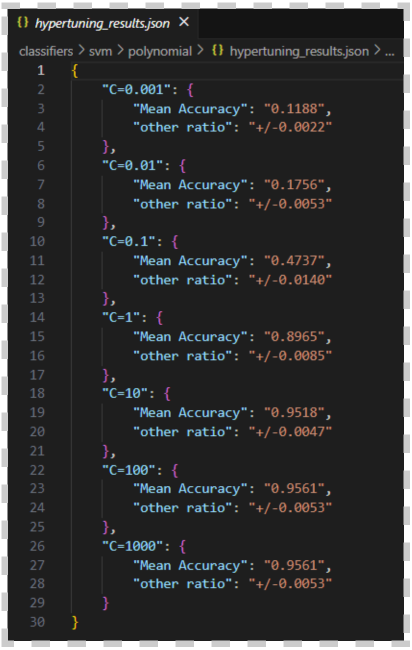
Below, the result of the polynomial kernel SVM is displayed in three different JSON files:



besult_results.json

classifiers > svm > polynomial > besult_results.json > ...

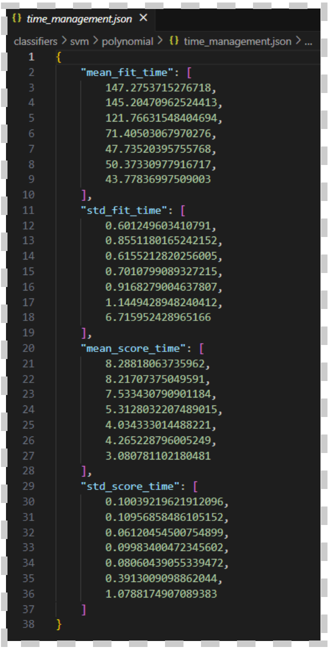
```json
{
    "Best Parameters": {
        "svc__C": 100
    },
    "Best Cross-Validation Accuracy": 0.9560999999999998
}
```



hypertuning_results.json

classifiers > svm > polynomial > hypertuning_results.json > ...

```json
{
    "C=0.001": {
        "Mean Accuracy": "0.1188",
        "other ratio": "+/-0.0022"
    },
    "C=0.01": {
        "Mean Accuracy": "0.1756",
        "other ratio": "+/-0.0053"
    },
    "C=0.1": {
        "Mean Accuracy": "0.4737",
        "other ratio": "+/-0.0140"
    },
    "C=1": {
        "Mean Accuracy": "0.8965",
        "other ratio": "+/-0.0085"
    },
    "C=10": {
        "Mean Accuracy": "0.9518",
        "other ratio": "+/-0.0047"
    },
    "C=100": {
        "Mean Accuracy": "0.9561",
        "other ratio": "+/-0.0053"
    },
    "C=1000": {
        "Mean Accuracy": "0.9561",
        "other ratio": "+/-0.0053"
    }
}
```



time_management.json

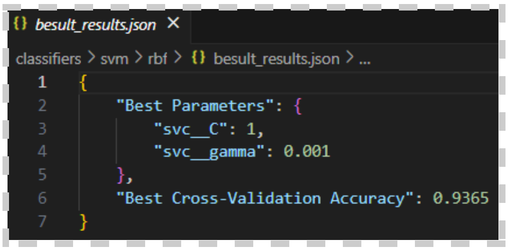
classifiers > svm > polynomial > time_management.json > ...

```json
{
    "mean_fit_time": [
        147.2753715276718,
        145.20470962524413,
        121.76631548404694,
        71.40503067970276,
        47.73520395755768,
        50.37330977916717,
        43.77836997509003
    ],
    "std_fit_time": [
        0.601249603410791,
        0.8551180165242152,
        0.6155212820256005,
        0.7010799089327215,
        0.9168279004637807,
        1.1449428948240412,
        6.715952428965166
    ],
    "mean_score_time": [
        8.28818063735962,
        8.21707375049591,
        7.533430790901184,
        5.3128032207489015,
        4.034333014488221,
        4.265228796005249,
        3.080781102180481
    ],
    "std_score_time": [
        0.10039219621912096,
        0.10956858486105152,
        0.06120454500754899,
        0.09983400472345602,
        0.08060439055339472,
        0.3913009098862044,
        1.0788174907089383
    ]
}
```

The results indicate that the best outcome is obtained when setting *C* to 100 such that the accuracy reaches almost 96%, which is an improvement of 3% in comparison to the linear SVM kernel. However, when quickly glancing over the right picture of *time_management.json*, it's apparent that every operation requires more resources and is more time consuming than that of the linear kernel. We can also derive from *hypertuning_results.json* that lower values of *C* decrease the accuracy, in contrast to the linear kernel where *C=0.005* is the best option.

4.3 SVM - Radial Basis Function Kernel

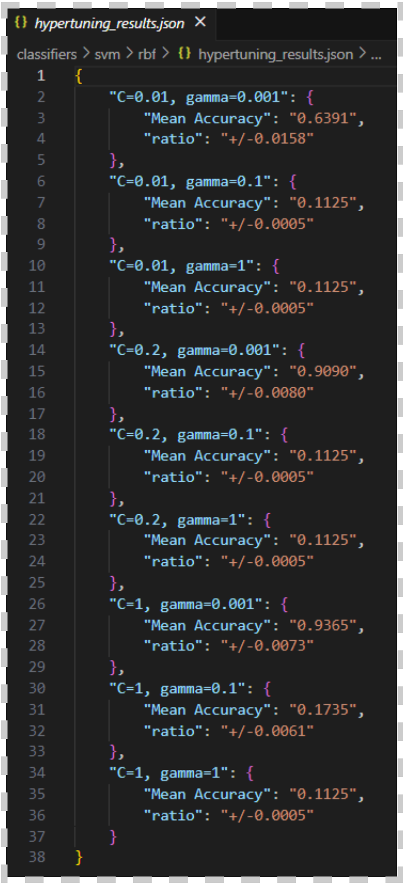
Below, the result of the RBF kernel SVM is displayed in three different JSON files:



besult_results.json

classifiers > svm > rbf > besult_results.json > ...

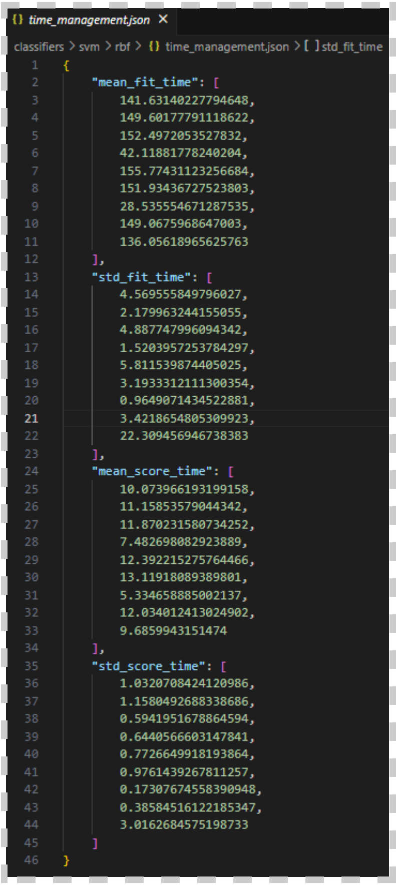
```json
{
    "Best Parameters": {
        "svc__C": 1,
        "svc__gamma": 0.001
    },
    "Best Cross-Validation Accuracy": 0.9365
}
```



hypertuning_results.json

classifiers > svm > rbf > hypertuning_results.json > ...

```
{
    "C=0.01, gamma=0.001": {
        "Mean Accuracy": "0.6391",
        "ratio": "+/-0.0158"
    },
    "C=0.01, gamma=0.1": {
        "Mean Accuracy": "0.1125",
        "ratio": "+/-0.0005"
    },
    "C=0.01, gamma=1": {
        "Mean Accuracy": "0.1125",
        "ratio": "+/-0.0005"
    },
    "C=0.2, gamma=0.001": {
        "Mean Accuracy": "0.9090",
        "ratio": "+/-0.0080"
    },
    "C=0.2, gamma=0.1": {
        "Mean Accuracy": "0.1125",
        "ratio": "+/-0.0005"
    },
    "C=0.2, gamma=1": {
        "Mean Accuracy": "0.1125",
        "ratio": "+/-0.0005"
    },
    "C=1, gamma=0.001": {
        "Mean Accuracy": "0.9365",
        "ratio": "+/-0.0073"
    },
    "C=1, gamma=0.1": {
        "Mean Accuracy": "0.1735",
        "ratio": "+/-0.0061"
    },
    "C=1, gamma=1": {
        "Mean Accuracy": "0.1125",
        "ratio": "+/-0.0005"
    }
}
```



time_management.json

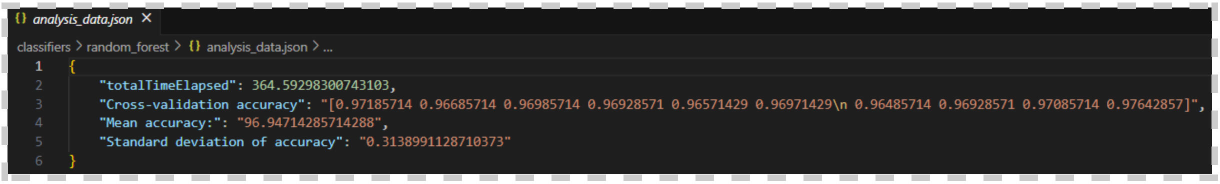
classifiers > svm > rbf > time_management.json > [ ] std_fit_time

```
{
    "mean_fit_time": [
        141.63140227794648,
        149.60177791118622,
        152.4972053527832,
        42.11881778240204,
        155.77431123256684,
        151.93436727523803,
        28.535554671287535,
        149.0675968647003,
        136.05618965625763
    ],
    "std_fit_time": [
        4.569555849796027,
        2.179963244155055,
        4.887747996094342,
        1.5203957253784297,
        5.811539874405025,
        3.1933312111300354,
        0.9649071434522881,
        3.4218654805309923,
        22.309456946738383
    ],
    "mean_score_time": [
        10.073966193199158,
        11.15853579044342,
        11.870231580734252,
        7.482698082923889,
        12.392215275764466,
        13.11918089389801,
        5.334658885002137,
        12.034012413024902,
        9.6859943151474
    ],
    "std_score_time": [
        1.0320708424120986,
        1.1580492688338686,
        0.5941951678864594,
        0.6440566603147841,
        0.7726649918193864,
        0.9761439267811257,
        0.17307674558390948,
        0.38584516122185347,
        3.0162684575198733
    ]
}
```

As the first picture alludes, the unique feature of this classifier is the *Gamma* parameter. Hence, in order to display the hypertuning, all of the possible combinations between *C* and *Gamma* are illustrated in *hypertuning_results.json*. The highest accuracy achieved is close to 94%, and similar to the polynomial kernel, the total time of runtime execution is significantly longer than the linear kernel.

4.4 Random Forests

In contrast to the SVM classifiers with 3 output JSON files, the execution details are summarized in only one file below.



analysis_data.json

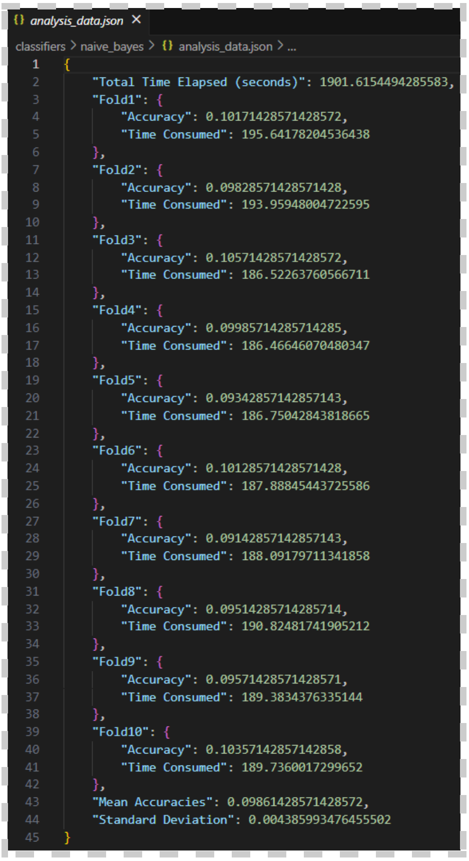
classifiers > random_forest > analysis_data.json > ...

```
{
    "totalTimeElapsed": 364.59298300743103,
    "Cross-validation accuracy": "[0.97185714 0.96685714 0.96985714 0.96928571 0.96571429 0.96971429\n 0.96485714 0.96928571 0.97085714 0.97642857]",
    "Mean accuracy:": "96.94714285714288",
    "Standard deviation of accuracy": "0.3138991128710373"
}
```

As the picture illustrates, the total execution time for processing the entire MNIST database and performing 10-way-cross validation was *364* seconds. The accuracy achieved for each fold is displayed in the shape of an array, with the highest being *0.976*, being close to 98%.

4.5 Naive Bayes

Below, the result of the Naive Bayes classifier is displayed in a JSON file:

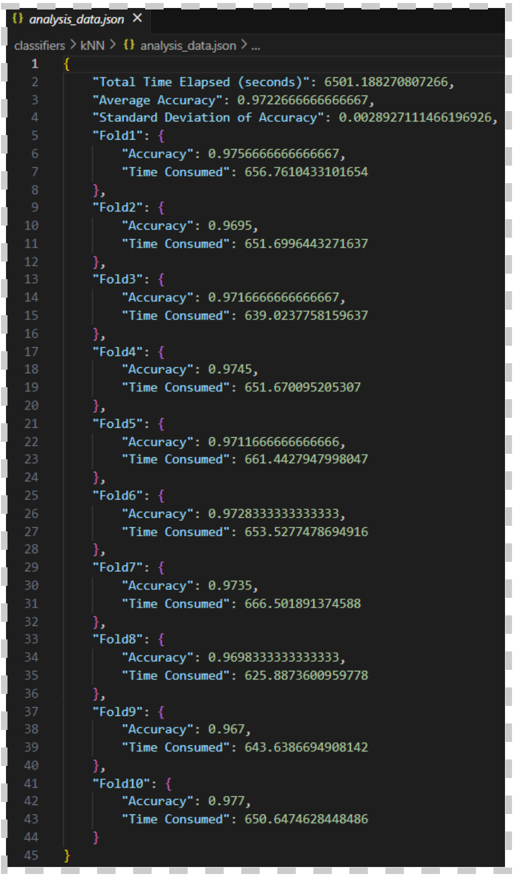


```json
{
    "Total Time Elapsed (seconds)": 1901.6154494285583,
    "Fold1": {
        "Accuracy": 0.10171428571428572,
        "Time Consumed": 195.64178204536438
    },
    "Fold2": {
        "Accuracy": 0.09828571428571428,
        "Time Consumed": 193.95948004722595
    },
    "Fold3": {
        "Accuracy": 0.10571428571428572,
        "Time Consumed": 186.52263760566711
    },
    "Fold4": {
        "Accuracy": 0.09985714285714285,
        "Time Consumed": 186.46646070480347
    },
    "Fold5": {
        "Accuracy": 0.09342857142857143,
        "Time Consumed": 186.75042843818665
    },
    "Fold6": {
        "Accuracy": 0.10128571428571428,
        "Time Consumed": 187.88845443725586
    },
    "Fold7": {
        "Accuracy": 0.09142857142857143,
        "Time Consumed": 188.09179711341858
    },
    "Fold8": {
        "Accuracy": 0.09514285714285714,
        "Time Consumed": 190.82481741905212
    },
    "Fold9": {
        "Accuracy": 0.09571428571428571,
        "Time Consumed": 189.3834376335144
    },
    "Fold10": {
        "Accuracy": 0.10357142857142858,
        "Time Consumed": 189.7360017299652
    },
    "Mean Accuracies": 0.09861428571428572,
    "Standard Deviation": 0.004385993476455502
}
```

The first thing that comes to mind when looking at the data above is the *Total Time Elapsed (seconds)* attribute. It points to as much as *1901* seconds, whereas the Random Forest algorithm only needs *365* seconds. In addition, Naive Bayes' accuracy is significantly lower. The mean accuracy is *0.098* or approximately 10%, and Random Forest's is *0.969* or 97%. As expected, the highest accuracy is only *0.1057* or 10.6%.

4.6 k-NN

Below, the result of the k-NN classifier is displayed in a JSON file:



```json
{
    "Total Time Elapsed (seconds)": 6501.188270807266,
    "Average Accuracy": 0.9722666666666667,
    "Standard Deviation of Accuracy": 0.0028927111466196926,
    "Fold1": {
        "Accuracy": 0.9756666666666667,
        "Time Consumed": 656.7610433101654
    },
    "Fold2": {
        "Accuracy": 0.9695,
        "Time Consumed": 651.6996443271637
    },
    "Fold3": {
        "Accuracy": 0.9716666666666667,
        "Time Consumed": 639.0237758159637
    },
    "Fold4": {
        "Accuracy": 0.9745,
        "Time Consumed": 651.670095205307
    },
    "Fold5": {
        "Accuracy": 0.9711666666666666,
        "Time Consumed": 661.4427947998047
    },
    "Fold6": {
        "Accuracy": 0.9728333333333333,
        "Time Consumed": 653.5277478694916
    },
    "Fold7": {
        "Accuracy": 0.9735,
        "Time Consumed": 666.501891374588
    },
    "Fold8": {
        "Accuracy": 0.9698333333333333,
        "Time Consumed": 625.8873600959778
    },
    "Fold9": {
        "Accuracy": 0.967,
        "Time Consumed": 643.6386694908142
    },
    "Fold10": {
        "Accuracy": 0.977,
        "Time Consumed": 650.6474628448486
    }
}
```

In contrast to Naive Bayes' classifier, k-NN yields more satisfying results in terms of accuracy. The highest one attained is almost 98% where the *K* value was set to 3. On the other hand, the time consumed for each single fold was longer than the total elapsed time for the Random Forest approach (364 seconds). The k-NN approach required vast resources to compute all of the pairwise mappings of the neighbors, so much that it almost took 2 hours for the classifier to execute from start to finish. In total, as much as *1.8* hours or *6501* seconds elapsed.

5. Conclusions

After exploring the six different approaches and examining what results they yielded, it's now possible to reflect on the insights and draw conclusions about their differences and similarities. First of all, it's important to acknowledge that the task or mission for this project was to classify instances in the MNIST database. Thus, the conclusions below and their associated performances demonstrated in the "Results" are based on the very problem of classification. However, attempts to generalize their quality and applicability to different scenarios will be made, so that their quality can be observed outside the limited scope of this project. This is important since each of the implemented classifiers have their own ideal circumstances affecting their performances.

5.1 Generalization Errors

As discussed earlier, generalization errors are a major part in what makes a model perform well or not. In the context of this project where classification is the main focus, it appears that all implemented classifiers have in common that they need to find a balance between underfitting and overfitting. The problem is that the model can be too simplistic or complex for the given task. In turn, this would cause it to either miss out on important details or capture additional noise, which ultimately results in incorrect or inaccurate output. Each classifier has its own distinct way of resolving this by incorporating a method to adjust or regulate the model's complexity. However, what's common for most models is their adaptation of hyperparameters that can be tuned accordingly to balance trade-offs. In this project, the linear and polynomial SVM kernels are two examples of classifiers using hyperparameters. On top of this, the RBF kernel implements automatic feature engineering to adjust by itself based on the complexity of the problem. Even though this automatic approach sounds more convenient, it doesn't matter too much in today's day and age where there's an abundance of tools and methodologies. Cross validation is one of these, and it's especially effective with refining hyperparameters. Conclusively, the notations of overfitting and underfitting are generally regarded in classifiers, forcing them to systematically find an optimal balance between the two. Taking this a step further and looking at the broader topic of classification per se, there are further general insights to conclude. As stated in the "Theoretical Background" section, supervised learning is the optimal learning approach for models seeking to classify into a finite and predefined set of categories due to the labels provided. Hence, classification is supported by supervised learning. Since relational patterns of features between the labeled data points in the dataset constitute patterns that can be simplistic or complex, the model must match this degree of complexity to avoid overfitting or underfitting. In this way, this generalization problem extends beyond the realm of classification and applies to all models using supervised learning. On the other hand, unsupervised learning models also face the same issue of underfitting and overfitting, but they differ due to the absence of labeled data. Instead of originating from the mapping on input and output labels, they emerge as a result from the general structure during the exploration. Ultimately, all models, whether it be supervised or unsupervised, and all of the algorithms or classifiers that are included in these umbrella terms, all suffer from overfitting and underfitting, and must therefore find ways to balance the model's complexity.

5.2 Classifier Performances

As can be seen in the previous section "Results", both the accuracies and times required are vastly different. All classifiers succeed in their mission to classify the MNIST database, but some are apparently more suitable for this particular task. The question is how and why they are distinguished, and this will be explored further below:

5.2.1 Support Vector Machines

These types of classifiers performed well in terms of accuracy on the MNIST database, and their best results ranged between 92-95%. Below follows a brief description of the kernels' performances:

- The linear had the lowest percentage of 92%, but it was significantly faster than the two other kernels → Due to the dimensional simplicity
- The polynomial was substantially slower than the linear kernel, but it obtained the highest accuracy of all SVMs, achieving nearly 96%
- The RBF was slightly more accurate than the linear, having almost 94%, but it was simultaneously slower than the polynomial one

It is safe to conclude that the polynomial was the most efficient in this task. But why? First of all, the linear kernel is too simple to represent the correlations in the database instances of the MNIST database. By slightly increasing the complexity by turning it into a polynomial of degree 2, the model captures some more details and the accuracy increases by 4%. What is interesting to note is that increasing the degree to, for instance 5, would only deteriorate the accuracy and make the algorithm slower, as it begins to pick up redundant noises (overfitting). This then begs the question about the RBF kernel's performance, since it automatically adjusts the degree rather than requiring an explicit specification. Sure, this kernel achieved a great accuracy (almost 94%), but how can it be lower than the polynomial kernel of 96%? Considering the advanced automatic feature engineering, it may be hard to reduce complexity for simplistic datasets. Keeping in mind that the polynomial degree of 2 almost yields a 100% accuracy, coupled with the fact that the automatic feature is more useful for finding the best degree for more complex datasets where its harder to identify the best underlying degree, we can conclude that the RBF kernel is more suitable for complex scenarios with higher degrees.

5.2.2 Other

This section concerns the classifiers of Random Forests, Naive Bayes and k-NN. Unlike the previous section, the results of these vary significantly in both accuracies and required run times. Below, the results are presented and compared. Note that the run times represents the total execution time including the cross validation:

- Random Forests: The total time required was 364 seconds and the astounding accuracy of nearly 98% was obtained
- Naive Bayes: Performed much slower than Random Forest, taking 1900 seconds while producing an insufficient result of 10% accuracy at best
- k-NN: A disappointing execution time of 6500 seconds but consistently achieved approximately 97% accuracy for all folds

The most beneficial classifier for this task was obviously Random Forest that generated the highest accuracy and was also substantially faster than the others. This is due to its decision trees successfully being able to combine their wisdom to reduce the overfitting. As mentioned in the “Approach” section, this model may be complex, but it systematically splits each tree in a randomized fashion, and uses the subsets to aggregate the data. In this way, it adapts effectively to most datasets to secure high performance. Naive Bayes, however, failed to classify with confidence. Although it used confidence as a metric for how probable or true a prediction was deemed to be, something went wrong, as it only had a 10% accuracy. The drawback of this classifier, as addressed in the “Approaches” section, is that it assumes independence of the data points. This may make it more efficient but can many times lead to lower accuracy, since the data usually possess some degree of correlation. In the case of MNIST database, the correlation isn’t necessarily the digits (0-9), but rather their inherent features, or visual contours. Lastly, the k-NN approach yields as high as 97% accuracy, but it requires a lot of time for each fold. This is because of the repetitive mapping for each data point, as in order to determine the k nearest neighbors, every instance must be traversed, processed and compared with respect to their relative distance. Therefore, performing this classifier algorithm on large datasets, such as MNIST database, where computations for each data point pair are made, ends up being very time consuming.

5.3 Classifier Applications

Always when conducting an experiment it’s important to generalize to gain a broader view of the topic’s applications to other areas. This helps us understand how it contributes to society, the field, and the industry as a whole. In turn, it also helps us uncover the more profound components and core functionalities of it.

In regards to the two non-linear SVMs, while the polynomial requires an explicit specification of the dimension, the RBF kernel sets it implicitly. The advantage with explicit specification is that the developer has more control and can adjust accordingly to experiment, but this may take time to find satisfiable degrees. Conversely, the implicit specification opens up possibilities for application on a wider range of non-linear problems. In this way, the RBF is more flexible to other types of problems with higher dimensional spaces. On the other hand, these two kernels are more sensitive to large datasets, as the degradation of their performance as the size increases is more notable than the linear kernel. In addition, as mentioned multiple times throughout this report, due to the concept of overfitting and underfitting and the problem of matching the complexity of the model, linear kernels’ ideal case is linearly separable data, whereas non-linear data is preferred for the other approaches.

As for the applicability of random forests, it’s similar to k-NN classifier since both use majority voting to determine which class will be the final output. While random forests rely on the vote for each tree, k-NN uses the k nearest neighbors. This quality of accounting for multiple subsets and their votes makes both classifiers more versatile and prepared for a wider variety of challenges or problems. As opposed to support vector machines that use a more simple approach that generally enables faster runtimes, where classifications are made based on what side of the hyperlane an instance lies at, k-NN leverages distance-mapping to prioritize the neighbors. As k-NN doesn’t rely on decision boundaries but rather finds similarity-mappings quantified in distance, it generally requires more resources than SVMs, but it maintains consistency of high output accuracies across a wider range of problems. However, there's a similarity between this classifier and the RBF kernel. The function of the *k* value resembles the Gamma parameter, as it controls the region of focus for how many data points are included in the computations, adjusting for overfitting and underfitting. Moreover, despite the Naive Bayes classifier using the *Alpha* and *Beta* parameters, it struggled to produce satisfying results. Although *Beta* controls the influence of the prior knowledge to the predictions, the absence of modelling interactions between features makes it harder to deal with complex data. Conclusively, this probabilistic classifier is more applicable to datasets with less inherent dependencies, but it's important to note that the skewness and size is less relevant to its performance.