Evaluation of Discriminative and Generative Classifiers on the MNIST Dataset

Zeeshan Ahmed (908115@stud.unive.it)

Introduction

This study examines discriminative and generative classifiers using the MNIST handwritten digit classification task. It focuses on four well-known models: Support Vector Machine (SVM), Random Forest, Naive Bayes, and k-Nearest Neighbors (k-NN). We will highlight these classifiers' approaches to data modeling by looking at their theoretical underpinnings, decision rules, and structural variations. Each algorithm's general flow, including the training, testing, and prediction phases, will be described in the report. Efficiency, training and testing time, and accuracy will all be taken into consideration when evaluating performance. To optimize parameters and find the optimal model configurations, 10-fold cross-validation will be used.

Background

In this section, we will learn about core concepts we need to understand the rest of concepts discussed in this document and theoretical,mathematical and implementation details of algorithms which we are going to test.

Different Types of Data: Linearly Separable and Non-Separable

In machine learning, one of the fundamental distinctions made when working with classification problems is whether the data is **linearly separable** or **non-separable**.

- **Linearly Separable Data**:

  This kind of data describes circumstances in which the various classes can be distinguished by a distinct border, or hyperplane. In other words, a hyperplane (in higher dimensions) or a straight line (in two dimensions) can be used to divide the data points of one class from those of the other class. Linear classifiers, such Support Vector Machines (SVM), can do a great job in these situations.

- **Non-Separable Data**:

  Non-separable data, on the other hand, describes situations in which it is impossible to establish a boundary that would completely divide the classes. This happens when data points from several classes are dispersed or overlap in such a way that a straightforward linear boundary is not feasible. In order to convert non-separable data into a space where separation is feasible, more intricate models or kernel functions—like those found in SVM—are frequently needed.

What is Classification?

Formal definition, assigning an input to one of several predefined groups or classes is the aim of the supervised learning problem known as classification. A model is trained using labeled data, in which every data point has a class label assigned to it. After being trained, the model can use the patterns it has discovered in the training data to predict the class label of fresh, unseen data.

A binary classification challenge can, for instance, ask you to decide whether an email is spam or not based on a number of characteristics, including the subject, sender, and content. Sorting pictures of animals into groups such as dogs, cats, and birds could be part of a multi-class categorization work.

What is a Hypothesis Function?

A mathematical function that converts input data into output predictions—usually in the form of class labels—is called a hypothesis function. The function is a representation of how well the model comprehends the underlying data and how it makes decisions when classifying new instances. In linear regression, for instance, the hypothesis function could be

$$h(x) = \mathbf{w}^T\mathbf{x} + b$$

We are the weights and b is the bias. In a classification problem, such as with logistic regression, the hypothesis function might be used to compute the probability that a given instance belongs to a certain class.

An example in the case of a binary classifier could be a logistic regression model where the hypothesis function computes the probability that a given input belongs to class 1, typically expressed as:

$$h(x) = \frac{1}{1 + e^{-(\mathbf{w}^T\mathbf{x}+b)}}$$

This function outputs a value between 0 and 1, which is interpreted as the probability that the input belongs to class 1.

What is the Role of Probability in Classification?

Since probability helps measure uncertainty regarding class membership, it is essential to classification. A probabilistic classifier assigns a probability to each potential class, indicating the likelihood that the input belongs to each class, as opposed to making a hard judgment (such as identifying a point as belonging to a specific class).

What are PDFs and Why Do We Need Them in Classification?

In statistics, probability density functions, or PDFs, are used to express the probability that a random variable will take a specific value. Because they enable us to simulate the distribution of data for each class, PDFs are essential in the classification context. We may more accurately forecast which class a new data point is most likely to belong to by knowing how the data is dispersed.

For instance, PDFs are used to model the probability distribution of the features for each class in a Naive Bayes classifier. The classifier uses these PDFs to determine the probability that the characteristics belong to each class when making predictions. The Bernoulli distribution (for binary data), the Gaussian distribution (for continuous data), and the beta distribution (for proportion-based data, especially when the values lie within the range of [0, 1]) are common distributions used in classification.

Because they offer a means of measuring the degree of uncertainty or variance in the data for every class, PDFs are significant in the classification process. Based on the observed feature values, they assist us in estimating the likelihood that a given data point belongs to a specific class. Classification would be more dependent on rigid decision boundaries in the absence of these distributions, which might not translate well to novel or untested data.

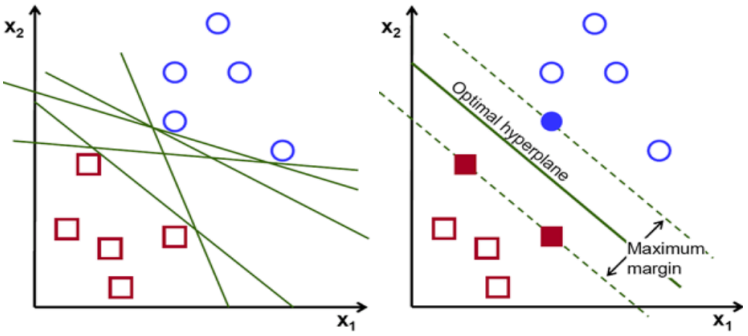
SVM Classifier

Overview of Support Vector Machines (SVM)

Classification and regression tasks are the main applications for Support Vector Machines (SVM), a family of supervised machine learning techniques. Finding the best hyperplane to divide the data points of several classes in a feature space is the main concept of support vector machines (SVM). In order to improve generalization when the model is applied to fresh, untested data, the method seeks to maximize the margin between the classes.

Applicability of SVM to Data Types

When tackling classification difficulties, SVM works quite well, especially when the dataset is high-dimensional—that is, has more features than data points. Although it has features for handling complex or noisy data, it performs best in situations where classes can be easily distinguished. SVM is commonly used in image recognition, text categorization, speech recognition, and bioinformatics (including gene classification).



Why Use SVM?

SVM is preferred because it can generalize well even with tiny datasets, which is very helpful when training data is few. It works well even with noisy datasets and is resistant to overfitting, particularly in high-dimensional domains. SVM's versatility across a range of applications is further enhanced by its ability to handle both linearly and nonlinearly separable data through the use of kernel functions.

How Does SVM Classify Data? The Mathematical Foundation

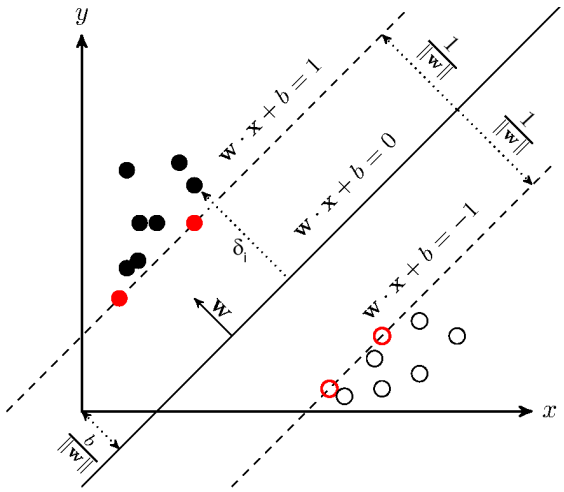
Finding the hyperplane that best divides the various data classes is the first step in SVM's classification methodology. The hyperplane that maximizes the margin between the nearest data points—known as support vectors—from each class is the ideal one.

The mathematical aim is to determine the weights (w) and bias (b) that define the hyperplane by solving an optimization problem. The equation serves as the foundation for the decision rule.

$$w \cdot x + b = 0$$

where w is the weight vector normal to the hyperplane and x is a feature vector.

The distance between the closest data points and the hyperplane is used to compute the margin. The chance of incorrectly identifying new data points is decreased by maximizing this margin. This optimization issue can be shown as follows:



$$\text{minimize} \frac{1}{2}||w||^2$$

subject to the constraint that all data points satisfy:

$$y_i(w \cdot x_i + b) \geq 1 \quad \text{for all} \quad i$$

where yi represents the class label for the i-th data point.

The Role of Kernels: Linear, Polynomial, and RBF

Although SVM excels in linear contexts, real-world data frequently exhibits non-linear separability. SVM employs kernel functions to deal with these situations, mapping the data into a higher-dimensional space where a linear separation would be feasible. With the help of these kernels, SVM may handle challenging classification tasks without requiring the higher-dimensional transformation to be explicitly calculated.

- **Linear Kernel**: Used when the data is linearly separable. It computes the dot product between the feature vectors:

$$K(x, x') = x \cdot x'$$

- **Polynomial Kernel**: This kernel is useful for datasets with polynomial relationships. The function is defined as:

$$K(x, x') = (x \cdot x' + c)^d$$

where c is a constant and d is the polynomial degree.

- **Radial Basis Function (RBF) Kernel**: A widely used kernel, especially when the relationship between the classes is complex. The RBF kernel measures the similarity between two data points by considering the exponential of the squared distance between them:

$$K(x, x') = \exp\left(-\frac{||x - x'||^2}{2\sigma^2}\right)$$

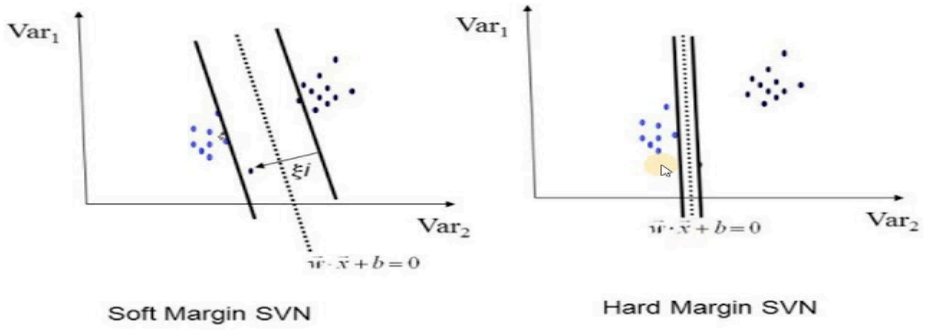
where σ is a parameter that controls the width of the kernel

Key Concepts in SVM: Margin, Maximization, and Geometry

Several key concepts drive the success of SVM, including the margin, maximizing the margin, and the geometry of the data.

- **Margin**: The margin is the distance between the separating hyperplane and the closest points from either class. A larger margin typically leads to a more robust model.
- **Maximizing the Margin**: By maximizing the margin, SVM reduces the likelihood of overfitting and improves generalization to new data. The optimal hyperplane is the one that maximizes this margin, thereby achieving the best separation between classes.
- **Geometric Interpretation**: The geometric mean or the geometric properties of the transformed feature space play an important role, particularly when using non-linear kernels. The geometry allows the data to be mapped into a space where it can be linearly separated, despite being non-linearly separable in the original space.

Hard Margin vs. Soft Margin in SVM



Hard Margin

- **Definition**: A hard margin One kind of SVM necessitates perfect class separation, which means that no data point can fall outside of the margin. It is applied when there is a distinct class boundary and the data is linearly separable.
- **Characteristics**:
    - No misclassifications are allowed.
    - Only applicable to datasets where a perfect separation exists.

Soft Margin

- **Definition**: A **soft margin** SVM introduces the concept of **slack variables** to allow for some misclassification of data points. This approach is used when the data is **not perfectly separable** or is noisy. The soft margin enables the SVM to find an optimal hyperplane that balances margin maximization and misclassification tolerance.
- **Regularization Parameter C**:
    - C controls the trade-off between achieving a wide margin and minimizing the misclassification error.
    - A **high value of C** forces the model to prioritize minimizing misclassification, potentially resulting in a smaller margin if the data is noisy.
    - A **low value of C** gives more flexibility to the model, allowing for a wider margin but at the cost of permitting some misclassifications.

Scikit-learn's default SVM implementation employs a soft margin approach with a configurable regularization parameter (C). C is set to 1.0 by default, which strikes a compromise between misclassification and margin width. The model's sensitivity to data noise can be managed by varying C.

Default Method in scikit-learn:

- The SVC class in scikit-learn uses the soft margin approach by default.
- The parameter C can be specified when creating the model to adjust the trade-off between margin size and classification error. For example:

```
from sklearn.svm import SVC
model = SVC(C=1.0)  # Default value of C is 1.0
```

Hinge Loss in SVM

Hinge loss is the loss function commonly used in Support Vector Machines (SVM) to measure the error between the predicted and true class labels. It is designed to penalize misclassifications and to ensure that data points are correctly classified and lie on the correct side of the margin.

Definition

For a given data point xi with true label yi∈{−1,1}, the hinge loss is defined as:

$$L(y_i, f(\mathbf{x_i})) = \max(0, 1 - y_i \cdot f(\mathbf{x_i}))$$

The hinge loss in the SVM context motivates the model to minimize classification mistakes and maximize the margin. The objective of hard margin SVM is to attain complete separation, or zero hinge loss, for every data point. Soft margin SVM strikes a balance between penalizing misclassifications and maximizing margins by allowing some misclassification while attempting to reduce the overall hinge loss.

Lagrange Method for Maximizing Margin in SVM (Optimization)

Finding the best hyperplane to maximize the margin between data points of various classes is the goal of Support Vector Machines (SVM). The margin is the separation between the nearest support vectors—the data points from each class—and the hyperplane. The model's capacity to generalize is improved by optimizing this margin.

Optimization Problem

To maximize the margin, the optimization problem is framed as:

$$\min_{\mathbf{w},b} \frac{1}{2}\|\mathbf{w}\|^2$$

subject to the constraints:

$$y_i(\mathbf{w}^T\mathbf{x_i} + b) \geq 1 \quad \text{for each data point} \quad i$$

Lagrangian Formulation

The Lagrange method is applied by introducing Lagrange multipliers αi to handle the constraints. The Lagrangian function is:

$$L(\mathbf{w}, b, \boldsymbol{\alpha}) = \frac{1}{2}\|\mathbf{w}\|^2 - \sum_{i=1}^{n} \alpha_i \left[y_i(\mathbf{w}^T\mathbf{x_i} + b) - 1\right]$$

Here, αi≥0 are the Lagrange multipliers. The dual problem is obtained by taking the partial derivatives of the Lagrangian with respect to w, b , and αi, setting them to zero, and solving for the optimal values of αi.

$$\max_{\boldsymbol{\alpha}} \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \mathbf{x_i}^T \mathbf{x_j}$$

subject to the constraints

$$\alpha_i \geq 0 \text{ and } \textstyle\sum_{i=1}^{n} \alpha_i y_i = 0$$

Dual Formulation

By substituting for w in terms of αi, we obtain the dual formulation:

$$\mathbf{w} = \sum_{i=1}^{n} \alpha_i y_i \mathbf{x_i}$$

In the context of the Lagrange multipliers used in SVM, the values of αi (the Lagrange multipliers) have significant implications for the solution and the resulting classification.

Insights into the Lagrange Multipliers αi

- **When αi=0**:
    - If αi=0 for a particular data point, it means that the corresponding data point does **not** influence the position of the hyperplane. These data points lie **far away** from the optimal hyperplane and do not lie on the margin boundaries. Essentially, they do **not support vectors** and do not contribute to defining the decision boundary.
    - These points are "non-support vectors" that do not actively affect the classification process. They satisfy the constraint but do not lie on the critical boundary that separates the classes.
- **When 0<αi<C0**:
    - If αi is positive but less than the upper bound C (where C is the regularization parameter), the corresponding data point is a **support vector**. These points are the ones that are closest to the margin, and they directly influence the position of the hyperplane. They help in defining the boundary between classes and contribute to the decision-making process.
    - The optimization algorithm adjusts the values of αi for these support vectors to find the optimal hyperplane that maximizes the margin.
- **When αi=C**:
    - If αi reaches the upper bound C, it indicates that the corresponding point is on the **margin boundary** or is misclassified in a soft-margin SVM. These points are critical in defining the margin and the decision boundary but might be subjected to some misclassification in the case of a soft margin, depending on the value of C. Larger values of αi mean that the data point has more influence on the position of the hyperplane.

Geometric Interpretation

- **Support Vectors**: The data points with αi>0 are critical in defining the decision boundary. These points either lie on the margin boundaries or are misclassified but still impact the model's decision-making. They are the support vectors.

- **Non-Support Vectors**: The data points with αi=0 do not influence the hyperplane. These points are **well separated** from the margin and do not contribute to defining the decision boundary.

Summary

- αi=0: The point is not a support vector and does not affect the model.
- 0<αi<C0: The point is a support vector and influences the hyperplane.
- αi=C: The point is on the margin or misclassified, and it contributes significantly to the margin optimization.

This relationship helps in understanding which data points are essential for building the decision boundary and which can be ignored, thus streamlining the SVM's ability to classify data efficiently.

Random forest classifier

Overview of Random Forest

Random Forest is an ensemble learning method widely used for both regression and classification tasks. It builds a "forest" by combining multiple decision trees, and the final decision is made by taking the average of the trees' outputs (for regression) or the majority vote (for classification). The strength of Random Forest lies in its ability to reduce overfitting by averaging the predictions from various trees, each trained on different subsets of the data. This not only helps minimize the impact of noise in the data but also boosts the model's overall accuracy.

Applicability of Random Forest to Data Types

Random Forest is a powerful and flexible algorithm that's applied in various domains like finance, healthcare, marketing, and biology. It performs well with both structured and unstructured data, managing datasets that include a mix of numerical and categorical features. This algorithm is especially useful when the relationships between input features and the target variable are complex and nonlinear. Additionally, Random Forest is resistant to noise and overfitting, making it an excellent choice for datasets with many features or missing values. Its ability to handle these challenges makes it a go-to tool for many real-world applications.

Why Use Random Forest?

Random Forest is favored for its high accuracy, flexibility, and robustness. As a non-parametric method, it doesn't rely on any assumptions about the data's underlying distribution, which makes it adaptable to a wide variety of problems. Compared to individual decision trees, Random Forest is less susceptible to overfitting, particularly when dealing with large, complex datasets. It also has the ability to handle missing data efficiently and can evaluate the importance of different features, offering valuable insights into the structure of the data. These qualities make it a reliable and versatile tool for many machine learning tasks.

How Does Random Forest Classify or Predict? The Mathematical Foundation

The Random Forest algorithm works by creating multiple decision trees, each trained on a random subset of the data. In classification tasks, each tree in the forest makes a prediction, or "vote," for a class label, and the class with the most votes becomes the final prediction. For regression tasks, the algorithm averages the predictions from all the trees to arrive at the final result.

The process involves the following steps:

1. **Bootstrapping**: A random subset of the data is sampled with replacement to create different training sets for each tree, ensuring diversity in the trees.
2. **Feature Randomization**: When building each tree, only a random subset of features is considered for each split. This helps reduce the correlation between trees, further preventing overfitting.
3. **Building Trees**: Each tree is built by selecting the best possible split at each node, using a criterion like Gini impurity (for classification) or mean squared error (for regression).
4. **Voting/Averaging**: For classification, the final prediction is made based on the majority vote from all trees. For regression, the result is the average of the predictions from all trees.

This process of bootstrapping, feature randomization, and combining the results from multiple trees helps Random Forest maintain high accuracy while avoiding overfitting, even with complex datasets.

Why Random Subsets and Feature Randomization Are Important

The effectiveness of Random Forest largely comes down to two key concepts: **bootstrapping** and **feature randomization**.

1. **Bootstrapping**: This technique involves creating slightly different training sets for each tree by randomly sampling the data with replacement. This means some data points may be included multiple times in one tree's training set, while others may not be included at all. This diversity among the trees helps reduce overfitting, as each tree learns from a different subset of the data.

2. **Feature Randomization**: Instead of considering all features for every split, only a random subset of features is evaluated at each node of the tree. This makes the trees more diverse and reduces the likelihood that they will all make the same decisions. By decorrelating the trees in this way, the Random Forest improves its generalization ability and prevents overfitting, leading to better performance on unseen data.

Key Concepts in Random Forest: Bagging, Overfitting, and Feature Importance

Several key concepts are crucial for understanding the strength and behavior of Random Forest:

- **Bagging (Bootstrap Aggregating)**: Bagging helps reduce the variance of the model by training multiple decision trees on different subsets of the data and combining their predictions. Each tree is trained on a random sample of the data, and their predictions are averaged (for regression) or voted on (for classification). This approach mitigates the overfitting problem that individual decision trees often face, as the combined results from multiple trees smooth out any extreme errors.
- **Overfitting and Generalization**: While individual decision trees tend to overfit the training data, Random Forest improves generalization by averaging the predictions of multiple trees. By randomizing both the data samples and the features used at each split, Random Forest ensures that the model doesn't memorize the training data, but instead learns general patterns that can be applied to unseen data.
- **Feature Importance**: One of the advantages of Random Forest is its ability to assess the importance of each feature in making predictions. This is done by measuring how much each feature contributes to reducing impurity (such as Gini impurity or entropy) in the trees. Features that often result in the best splits across many trees are considered more important. This provides valuable insights into which variables are most influential in the decision-making process.

In summary, Random Forest is a robust ensemble learning method that combines the power of multiple decision trees to boost performance in both classification and regression tasks. Through techniques like bootstrapping, feature randomization, and averaging the predictions from different trees, it reduces overfitting and improves generalization. Additionally, its ability to evaluate feature importance and handle complex, diverse datasets makes Random Forest a highly effective and popular tool in machine learning.

Naive Bayes classifier

Overview of Naive Bayes Classifier

The Naive Bayes classifier is a probabilistic machine learning algorithm that relies on Bayes' Theorem. It assumes that the features used to predict the class label are conditionally independent, given the class label. While this assumption of independence may seem overly simplistic, Naive Bayes tends to perform surprisingly well in many tasks, especially in areas like text classification, sentiment analysis, and spam detection, where the independence assumption is often reasonably accurate. Despite its simplicity, Naive Bayes is a powerful tool for these kinds of applications.

Applicability of Naive Bayes

Naive Bayes is particularly well-suited for classification tasks where the features are assumed to be independent, given the class label. It performs effectively on datasets with high dimensionality, such as text classification, where each word or feature can be treated individually. This algorithm is also favored in situations where computational efficiency and simplicity are essential, as it requires fewer resources compared to more complex models. Its efficiency makes it an excellent choice for large-scale applications, especially when speed and scalability are important considerations.

Why Use Naive Bayes?

Naive Bayes is popular for its simplicity, speed, and high performance in various real-world tasks. It is particularly useful when:

- The number of features is large compared to the number of observations.
- The features are conditionally independent, or approximately so.
- The model needs to be trained quickly and with minimal computational resources.

It is also robust to irrelevant features and can handle missing data, making it adaptable to a variety of datasets.

Bayes' Theorem and Its Application

At the heart of Naive Bayes is **Bayes' Theorem**, which calculates the probability of a class Ck, given a set of features

$$P(C_k|x_1, x_2, ..., x_n) = \frac{P(C_k)P(x_1, x_2, ..., x_n|C_k)}{P(x_1, x_2, ..., x_n)}$$

Using Bayes' Theorem, the conditional probability of each class given the features is computed. The Naive Bayes classifier assumes that the features are conditionally independent given the class, so the equation simplifies to:

$$P(C_k|x_1, x_2, ..., x_n) \propto P(C_k) \prod_{i=1}^{n} P(x_i|C_k)$$

This simplification allows for easier computation, making Naive Bayes an efficient classifier

Usage of Distributions: Gaussian and Beta Distributions [check code flow diagrams below to check usage of these all formulas]

Naive Bayes can be adapted to work with various types of data by using different probability distributions to model the features:

- **Gaussian Distribution**: When the features are continuous, the Gaussian Naive Bayes model assumes that the features follow a Gaussian (normal) distribution. This is especially useful when modeling data that is approximately normally distributed. The probability density function (PDF) of a Gaussian distribution is:

$$P(x_i|C_k) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x_i-\mu)^2}{2\sigma^2}\right)$$

where μ\muμ is the mean, σ is the standard deviation, and xi is the feature value.

- **Beta Distribution**: For features that are constrained to a range (e.g., probabilities or proportions between 0 and 1), the Beta distribution is a suitable model. The PDF of the Beta distribution is given by:

$$P(x_i|C_k) = \frac{x_i^{\alpha-1}(1-x_i)^{\beta-1}}{B(\alpha,\beta)}$$

  It's similar to the given in assignment requirements

$$d(x;a,b) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)}x^{\alpha-1}(1-x)^{\beta-1}$$

Why Use the Log-Likelihood?

Naive Bayes uses log-likelihood to enhance numerical stability and simplify computations. When multiplying probabilities across many features, the resulting values can become exceedingly small, leading to numerical underflow. To address this, the algorithm applies the logarithm of the likelihood. This transforms the product of probabilities into a sum, making the calculations much more manageable and preventing the issues associated with very small numbers. Using the logarithm ensures that the computations remain stable, even when dealing with large datasets or many features.

$$\log P(C_k|x_1,x_2,...,x_n) \propto \log P(C_k) + \sum_{i=1}^{n} \log P(x_i|C_k)$$

This approach also helps in maximizing the posterior probability in a more stable manner

Steps in the Naive Bayes Algorithm

The Naive Bayes classifier follows these basic steps to classify data:

1. **Step 1: Calculate Prior Probabilities**
   For each class Ck , calculate the prior probability P(Ck) which is the proportion of class Ck in the training data:

$$P(C_k) = \frac{\text{Number of instances in class } C_k}{\text{Total number of instances in the dataset}}$$

2. **Step 2: Estimate Likelihoods :** For each feature xi and each class Ck, estimate the likelihood P(xi∣Ck) using an appropriate distribution (e.g., Gaussian or Beta). This involves calculating the mean and variance for continuous features or the frequency of each value for categorical features.
3. **Step 3: Compute the Posterior Probability for Each Class** For a given test instance, compute the posterior probability for each class using Bayes' Theorem:

$$P(C_k|x_1, x_2, ..., x_n) \propto P(C_k) \prod_{i=1}^{n} P(x_i|C_k)$$

   If using the log-likelihood, this becomes:

$$\log P(C_k|x_1, x_2, ..., x_n) \propto \log P(C_k) + \sum_{i=1}^{n} \log P(x_i|C_k)$$

4. **Step 4: Select the Class with the Highest Posterior Probability**
   After calculating the posterior probabilities for all classes, the class with the highest posterior probability is selected as the predicted class label for the test instance.
5. **Step 5: Make Predictions**
   The model predicts the class label for the test data based on the class with the highest posterior probability. In multi-class problems, this process is repeated for each test instance.

KNN Classifier

Overview of k-Nearest Neighbors (k-NN)

For classification and regression applications, the k-Nearest Neighbors (k-NN) algorithm is a straightforward, non-parametric, lazy learning method. Based on the idea that comparable data points typically have the same label, it is possible to estimate a test data point's class (for classification) or value (for regression) by looking at the classes or values of its closest neighbors in the training dataset.

Since k-NN doesn't require explicit training, it's especially well-suited for issues where other algorithms struggle to understand the relationship between data points. The value of k, the data format, and the distance metric selection all have a significant impact on its performance.

Applicability of k-NN

In applications like image identification, recommendation systems, medical diagnosis, and anomaly detection where a straightforward and understandable method is required, k-NN is frequently utilized. It performs best when there are no strong parametric assumptions regarding the underlying distribution of the data or when the data is well-defined in terms of proximity (such as spatial data).

Why Use k-NN?

k-NN is favored for its simplicity and effectiveness in situations where there is no explicit model-building phase. The algorithm:

- **Does not require training**: Unlike other machine learning algorithms, k-NN does not involve a training phase. Instead, it memorizes the training data and uses it during prediction.
- **Is flexible**: It can be used for both classification and regression tasks.
- **Is intuitive**: The algorithm's mechanism is easy to understand and explain, which makes it suitable for applications where interpretability is key.

However, k-NN is sensitive to the choice of **k** and the distance metric, and can be computationally expensive when dealing with large datasets.

Distance Metrics in k-NN

The k-NN algorithm relies on the concept of distance to determine the closeness of data points. Different distance metrics can be used based on the nature of the data:

1. **Euclidean Distance**: The Euclidean distance is the straight-line distance between two points in Euclidean space. It is the most commonly used metric in k-NN for continuous features and is given by:

$$d_{\text{Euclidean}}(x, x') = \sqrt{\sum_{i=1}^{n}(x_i - x'_i)^2}$$

where xi and xi′ are the i-th features of the two points x and x′, and n is the number of dimensions (features).

2. **Manhattan Distance**: Also known as the **L1 norm** or **taxicab distance**, the Manhattan distance measures the sum of the absolute differences between the corresponding features of two points:

$$d_{\text{Manhattan}}(x, x') = \sum_{i=1}^{n}|x_i - x'_i|$$

This metric is particularly useful when the data consists of categorical or ordinal features, or when you want to avoid the influence of outliers that might disproportionately affect the Euclidean distance.

3. **Cosine Similarity**: Cosine similarity measures the cosine of the angle between two vectors in an n-dimensional space. It is commonly used when working with text data (e.g., document classification), where the direction of the vector (representing the word frequencies) is more important than the magnitude:

$$\text{Cosine Similarity}(x, x') = \frac{x \cdot x'}{\|x\|\|x'\|}$$

where $x \cdot x'$ is the dot product of the vectors, and $\|x\|$ and $\|x'\|$ are the magnitudes of the vectors x and x′, respectively.

Why Use Principal Component Analysis (PCA)?

The curse of dimensionality, which states that as the number of features in a dataset grows, the significance of the distance between points decreases, might affect k-NN in high-dimensional datasets. Principal Component Analysis (PCA), a dimensionality reduction method, is one approach to solving this problem.

PCA projects the data into a lower-dimensional space while keeping the most informative features by identifying the directions (principal components) in the data that account for the greatest variance. By reducing the number of dimensions, PCA can:

- Improve the performance of k-NN by reducing the noise and making the distance calculation more meaningful.
- Reduce the computational cost, as fewer dimensions lead to faster calculations.

Usage of Trees: KD-Tree and Ball Tree

For large datasets, brute-force distance calculations can be computationally expensive. **KD-Trees** and **Ball Trees** are data structures that help optimize the search for nearest neighbors.

- **KD-Tree**: A **k-dimensional tree** is a binary tree that recursively partitions the feature space into two half-spaces, one for each node. It is efficient for low-dimensional data and allows faster search times for nearest neighbors.
- **Ball Tree**: A **Ball Tree** is a tree-based data structure that organizes data points into clusters, which are represented as "balls" (hyperspheres). Ball trees are especially efficient for high-dimensional datasets and can outperform KD-Trees when the number of dimensions is large.

Both KD-Trees and Ball Trees can significantly speed up the search for nearest neighbors, reducing the time complexity from O(n) to O(logn) for large datasets.

The Role of k in k-NN

The choice of **k** is a critical factor in the performance of the k-NN algorithm:

- **Small k (e.g., k=1)**: A smaller value of k makes the algorithm more sensitive to noise in the data, as it relies heavily on the nearest neighbor. This can lead to overfitting.
- **Large k**: A larger value of k smooths out the predictions by considering more neighbors, which can help reduce the impact of noise. However, it may also lead to underfitting, where the model becomes too simple and fails to capture the complexity of the data.

To select the optimal value of k, cross-validation is commonly used, where the algorithm is tested with different values of k, and the one that minimizes the error is chosen.

Steps in the k-NN Algorithm

The working of the k-NN algorithm can be broken down into the following steps:

1. **Step 1: Choose the number of neighbors (k)**
   Select the number of nearest neighbors, k, which will influence the decision-making process.

2. **Step 2: Calculate the distance**
   For each data point, calculate the distance to all other data points using a suitable distance metric (e.g., Euclidean, Manhattan, or Cosine Similarity).
3. **Step 3: Sort the distances**
   Sort the calculated distances in ascending order to find the closest neighbors to the query point.
4. **Step 4: Select the k-nearest neighbors**
   Choose the top k closest data points based on the sorted distances.
5. **Step 5: Assign a class (for classification) or compute the average (for regression)**
   - For classification, assign the class label that is most frequent among the k-nearest neighbors (majority voting).
   - For regression, compute the average or weighted average of the target values of the k-nearest neighbors.
6. **Step 6: Make predictions**
   The predicted class label or value is returned as the output.

Comparison of SVM, Random Forest, Naive Bayes (using Beta distribution), and K-NN

1. Support Vector Machine (SVM)

- **Learning**: SVM is a **supervised learning** algorithm that learns by finding the optimal hyperplane that separates the data into different classes. It focuses on **maximizing the margin** between data points of different classes.
- **Testing**: When testing, SVM computes the **signed distance** of a test sample from the hyperplane and classifies the sample based on which side of the hyperplane it falls.
- **Decision Rule**: The decision rule in SVM is based on the concept of a hyperplane that maximizes the margin. The decision boundary is defined by support vectors, the points closest to the hyperplane.
- **Suitable Data to Classify**: SVM works well for both **linearly separable** and **non-linearly separable** data. For non-linear data, kernel methods (e.g., polynomial or RBF kernels) are used to transform data into a higher-dimensional space where it becomes linearly separable.

2. Random Forest

- **Learning**: Random Forest is an **ensemble learning** technique based on decision trees. It builds multiple decision trees using bootstrapped subsets of the data and features, and aggregates their predictions to make the final classification decision.
- **Testing**: During testing, the prediction is made by **majority voting** from all the decision trees in the forest. The more trees in the forest, the more robust the classification.

- **Decision Rule**: Random Forest's decision rule is based on the majority vote of the individual trees. Each tree has a decision rule that partitions the data based on feature splits.
- **Suitable Data to Classify**: Random Forest is suitable for **complex, non-linear** data, especially when interactions between features are present. It can handle both **categorical** and **continuous** data without requiring data scaling

3. Naive Bayes (using Beta distribution)

- **Learning**: Naive Bayes is based on **Bayesian theory** and assumes that features are conditionally independent given the class label. For classification, Naive Bayes learns the **probability distributions** of features for each class. When Beta distribution is used, it models the distribution of features that are proportions or probabilities (like success/failure).
- **Testing**: During testing, Naive Bayes computes the **posterior probability** of each class given the feature values, using Bayes' Theorem. The class with the highest posterior probability is chosen as the predicted class.
- **Decision Rule**: The decision rule in Naive Bayes is to choose the class with the highest **posterior probability**, computed by applying Bayes' theorem and using the PDFs of the features for each class. The Beta distribution is used for modeling features that are probabilities or proportions.
- **Suitable Data to Classify**: Naive Bayes (with Beta distribution) works well when the data involves **probabilistic or proportion-based features**, especially for **binary classification** tasks (success/failure or presence/absence). It is ideal for **text classification** or **spam detection** where features represent probabilities or counts.

4. K-Nearest Neighbors (K-NN)

- **Learning**: K-NN is a **lazy learning** algorithm, meaning it doesn't actually learn a model during training. Instead, it memorizes the training data. When a new test sample is presented, it looks at the **k nearest neighbors** in the training set and makes a prediction based on their class labels.
- **Testing**: During testing, K-NN calculates the **distance** (e.g., Euclidean, Manhattan, or cosine similarity) between the test sample and all training samples. The class of the test sample is determined by the majority class among the k closest neighbors.
- **Decision Rule**: The decision rule in K-NN is based on the **distance metric**. The test sample is classified according to the majority class of its k nearest neighbors.
- **Suitable Data to Classify**: K-NN works well with **non-linear** data and is suitable for problems with **complex decision boundaries**. It is highly effective when the data is **dense** and when computational efficiency is not an issue, as the algorithm can be slow for large datasets.

In one picture

Used chat-GPT for getting information in more precise and formatted table

Aspect	SVM	Random Forest	Naive Bayes (Beta)	K-NN
Learning	Supervised, maximizes margin, finds optimal hyperplane	Supervised, ensemble of decision trees, bootstrapped subsets	Supervised, Bayesian approach, assumes feature independence	Lazy learning, no explicit model, memorizes training data
Testing	Classifies based on distance from hyperplane	Classifies based on majority voting from trees	Classifies based on posterior probability using Bayes' Theorem	Classifies based on nearest neighbors' majority
Decision Rule	Hyperplane decision boundary, margin-based	Majority voting from trees	Posterior probability, Bayesian decision rule	Majority class of k nearest neighbors
Suitable Data	Linearly or non-linearly separable data, especially with high dimensionality	Complex, non-linear, high-dimensional data, both categorical and continuous features	Probabilistic or proportion-based data, binary or multinomial classification	Non-linear, dense, and complex decision boundaries

Experiment

In this section we will conduct the different tests and will try to measure results like accuracy and time taken to learn and will compare the algorithms performance. And we will apply 10 fold cross validation on each classifier.

Environment and set-up

Dateset	mnist_784
IDE	Using Jupitor notebook to run code and get results

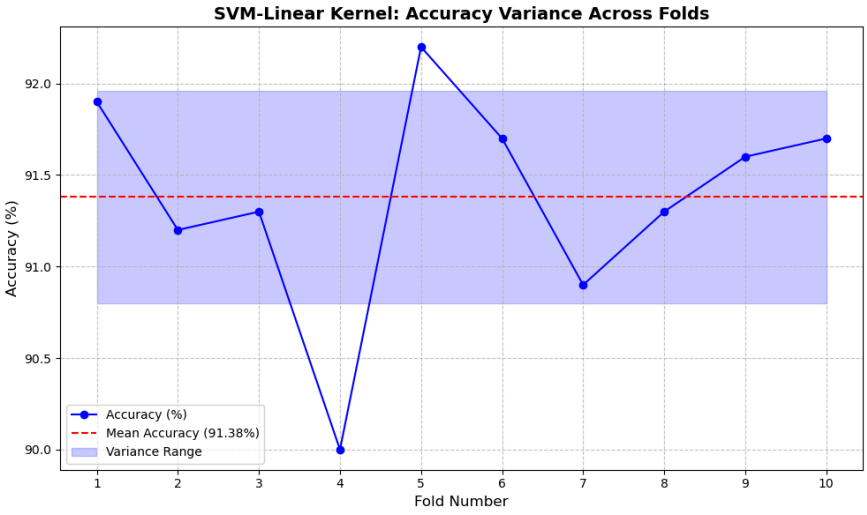
Results we will be calculating	• Accuracy insights • Time insights • Fold insights
Data portion we are using (i will be not using 100% of dataset as it's taking too much time to complete run)	I am using 10k total samples and by using 10 K-Fold technique we will see the different results
The diagrams and tables you will see results are also generated using code	You will find code related to graphs generation in a file named Results-formator.ipynb. I first noted down all the results I needed to show below and then used separate code to create graphs and tables.
You will find all the code related to results in separate files	For the sake of organization and keep this report file clean i have not attached code inside this, instead uploaded related jupyter notebook files
Help from GPT to format the numeric results.	After executing the code, i noted all the results and added comments for on them and then used chat-GPT format the results in textual form from numbers

SVM (results)

Since SVM has multiple kernels to use for computing the results, we will use each to calculate separate results.

Description and Analysis of Results: SVM with Linear Kernel

Accuracy



- The accuracy across the 10 folds ranged from **90.00%** to **92.20%** , with the highest accuracy observed in **Fold 5** .
- The **mean accuracy** was **91.38%** , indicating consistently good performance across all folds.
- The **accuracy variance** was **0.0034** , which is relatively low, demonstrating that the linear SVM model is stable and performs consistently on different subsets of the dataset.

Time

- The time taken for each fold ranged from **19.94 seconds** (Fold 9) to **29.50 seconds** (Fold 6).
- The **total time for all folds** was **252.47 seconds** , with an average of approximately **25.25 seconds per fold** .
- Variations in computation time per fold could be attributed to differences in the distribution of training and testing data across folds.

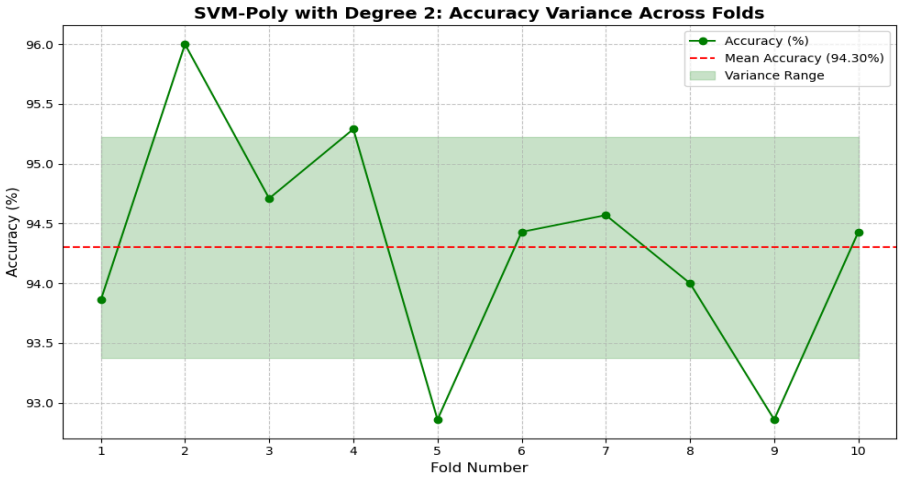
Fold	Accuracy (%)	Time Taken (s)
1.0	91.9	25.92
2.0	91.2	27.6
3.0	91.3	22.72
4.0	90.0	26.62
5.0	92.2	27.06
6.0	91.7	29.5
7.0	90.9	27.31
8.0	91.3	25.37
9.0	91.6	19.94
10.0	91.7	20.45

be

Description and Analysis of Results: SVM with Polynomial Kernel (Degree 2)

Accuracy

- The accuracy across the 10 folds ranged from **92.86%** to **96.00%** , with the highest accuracy observed in **Fold 2** .
- The **mean accuracy** was **94.30%** , which is higher than that of the linear SVM model, indicating improved performance on the MNIST dataset with the polynomial kernel.
- The **accuracy variance** was **0.0086** , which is slightly higher than that of the linear kernel. This suggests that while the model is still stable, there may be more variability in its performance across the folds.



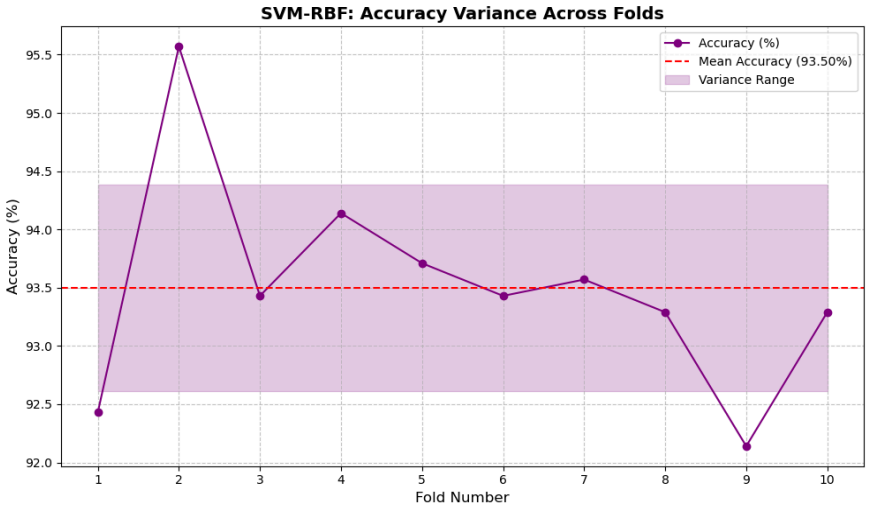
Time

- The time taken for each fold ranged from **21.94 seconds** (Fold 10) to **42.58 seconds** (Fold 2).
- The **total time for all folds** was **298.68 seconds** , with an average of approximately **29.87 seconds per fold** .
- The increased computation time compared to the linear kernel (25.25 seconds per fold) could be due to the higher complexity introduced by the polynomial kernel, which involves more calculations during training.

Fold	Accuracy (%)	Time Taken (s)
1.0	93.86	42.48
2.0	96.0	42.58
3.0	94.71	33.67
4.0	95.29	30.42
5.0	92.86	27.63
6.0	94.43	28.35
7.0	94.57	25.12
8.0	94.0	23.35
9.0	92.86	23.13
10.0	94.43	21.94

Description and Analysis of Results: SVM with Radial Basis Function (RBF) Kernel

Accuracy



- The accuracy across the 10 folds ranged from **92.14%** to **95.57%** , with the highest accuracy observed in **Fold 2** .
- The **mean accuracy** was **93.50%** , which is slightly lower than the polynomial kernel but still indicates strong performance.
- The **accuracy variance** was **0.0078** , which is comparable to the polynomial kernel variance, suggesting consistent performance across folds with some slight variation.

Time

- The time taken for each fold ranged from **23.69 seconds** (Folds 9 and 10) to **37.40 seconds** (Fold 2).
- The **total time for all folds** was **308.09 seconds** , with an average of approximately **30.81 seconds per fold** .
- The increased computation time compared to the linear kernel (25.25 seconds per fold) reflects the added complexity of the RBF kernel, which involves computing distances for each data point.

Fold	Accuracy (%)	Time Taken (s)
1.0	92.43	32.29
2.0	95.57	37.4
3.0	93.43	28.86
4.0	94.14	37.35
5.0	93.71	35.99
6.0	93.43	34.67
7.0	93.57	26.98
8.0	93.29	27.16
9.0	92.14	23.69
10.0	93.29	23.69

Effects of Kernel Selection on SVM Performance

1. **Accuracy:**
   - The **linear kernel** yielded the lowest mean accuracy of 91.38%, with an accuracy range from 90.00% to 92.20%. It demonstrated stable performance with low variance (0.0034), indicating consistent results across folds, but was not the most accurate.
   - The **polynomial kernel** (degree 2) provided the highest mean accuracy of 94.30%, with a range from 92.86% to 96.00%. This indicates that the polynomial

kernel is better suited for the MNIST dataset, capturing more complex decision boundaries.

- The **RBF kernel** produced a mean accuracy of 93.50%, with a range from 92.14% to 95.57%. While slightly lower than the polynomial kernel, the RBF kernel still provided strong results, showing it is a good alternative when the polynomial kernel's complexity is not necessary.

  Concussion : The **polynomial kernel** performed the best in terms of accuracy, closely followed by the **RBF kernel** . The **linear kernel** , while stable, offered lower accuracy.

2. **Time:**
   - The **linear kernel** was the fastest, with an average time of 25.25 seconds per fold and a total of 252.47 seconds for all folds. This is because the linear kernel is computationally less demanding due to its simpler model.
   - The **polynomial kernel** took more time, averaging 29.87 seconds per fold (total: 298.68 seconds). The increase in time can be attributed to the higher complexity of the polynomial kernel, which involves more feature transformations.
   - The **RBF kernel** required 30.81 seconds on average per fold (total: 308.09 seconds), which is also slower than the linear kernel but slightly faster than the polynomial kernel. The RBF kernel's time complexity arises from distance calculations for each data point.

     Conclusion : The **linear kernel** was the fastest, while both the **polynomial** and **RBF kernels** introduced additional time complexity, with the **RBF kernel** being marginally faster than the **polynomial kernel** .

Conclusion for SVM:

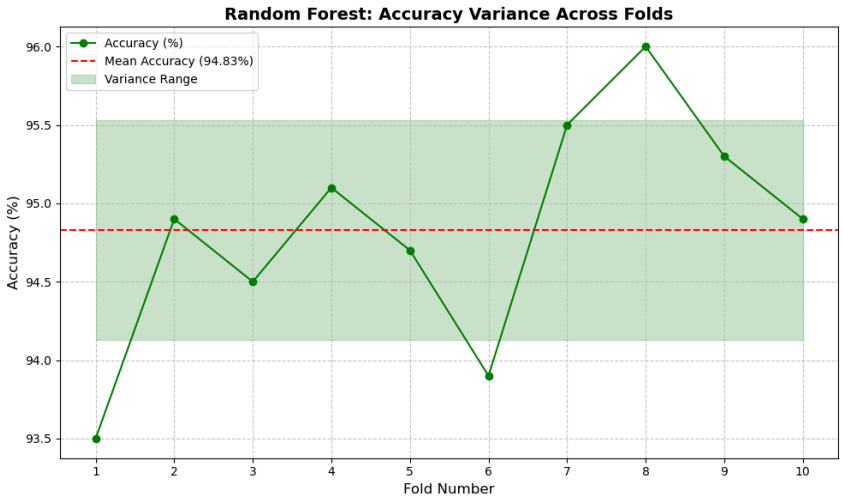
- **Choosing kernels** : The **polynomial kernel** provided the highest accuracy at the cost of longer computation time. The **RBF kernel** also performed well with slightly lower accuracy but faster than the polynomial kernel. The **linear kernel** offered the lowest accuracy but was computationally the most efficient.
- **Accuracy vs. Time** : As we move from linear to polynomial and RBF kernels, the accuracy improves, but at the expense of increased computation time. Therefore, selecting a kernel involves a trade-off between accuracy and computational resources.

Random forest (results)

Description and Analysis of Results

Accuracy

- The accuracy across the 10 folds ranged from **93.50%** to **96.00%** , with the highest accuracy observed in **Fold 8** .



- The **mean accuracy** was **94.83%** , indicating consistently strong performance across all folds.
- The **standard deviation** of **0.70%** reflects a low level of variability, meaning the model performs reliably across different subsets of the data.

Time

- The total time for all folds was **278.09 seconds** , with an average of approximately **27.81 seconds per fold** .
- This time is relatively moderate compared to the other models, considering the Random Forest algorithm typically requires more time for training due to its ensemble nature.

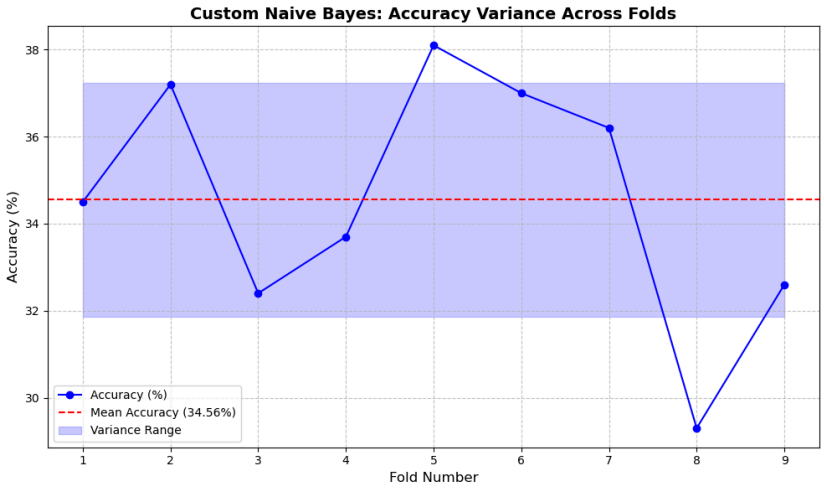
Fold	Accuracy (%)	Time Taken (s)
1.0	93.5	27.81
2.0	94.9	27.81
3.0	94.5	27.81
4.0	95.1	27.81
5.0	94.7	27.81
6.0	93.9	27.81
7.0	95.5	27.81
8.0	96.0	27.81
9.0	95.3	27.81
10.0	94.9	27.81

Naive bayes (results)

Description and Analysis of Results: Custom Naive Bayes

Accuracy

- The accuracy across the 10 folds ranged from **29.30%** (Fold 8) to **38.10%** (Fold 5).
- The **mean accuracy** was **34.68%** , indicating that the model struggled to achieve competitive performance on the MNIST dataset, which could be due to the simplistic nature of Naive Bayes, especially when feature independence assumptions do not hold.



- The **variance in accuracy** was **0.0007** , which is very low, suggesting that the model's performance remained relatively consistent across the folds, despite the low overall accuracy.

Time

- The time taken for each fold ranged from **2.00 seconds** (Fold 3) to **3.04 seconds** (Fold 10).
- The **mean time for testing** was **2.33 seconds** , showing that the Naive Bayes model is fast, which is expected due to its simplicity compared to more complex algorithms.

Fold	Accuracy (%)	Time Taken (s)
1.0	34.5	2.5123
2.0	37.2	2.1445
3.0	32.4	2.0003
4.0	33.7	2.0793
5.0	38.1	2.0222
6.0	37.0	2.2007
7.0	36.2	2.0764
8.0	29.3	2.7421
9.0	32.6	2.4578
10.0	35.8	3.038

- The **variance in time** was **0.1102** , which is relatively low, suggesting consistent computation times for each fold.

Summary of the mean Alpha and Beta Parameters I Calculated for whole data set (took mean of all alpha and beta values of all classes):

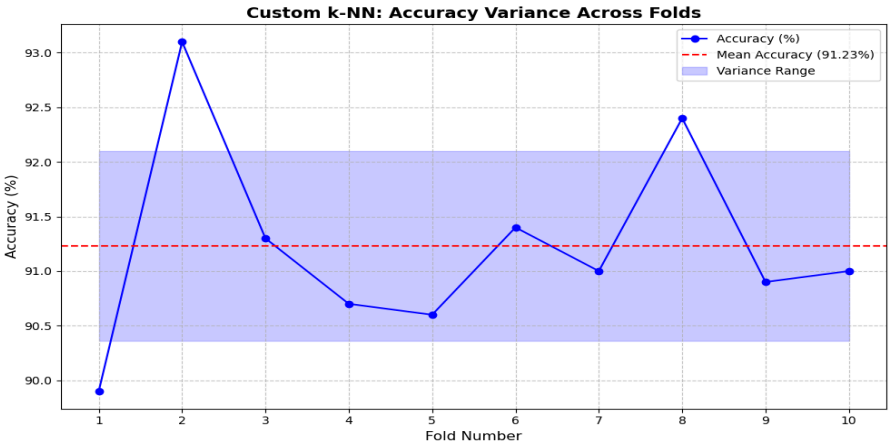
- **Alpha ~ 0.53**: The pixel intensities have a **moderate distribution**, leaning slightly towards darker pixels (but balanced, not extreme).
- **Beta ~ 1.60**: The pixel intensities have a **stronger concentration towards darker pixels** (closer to 0), but not exclusively.

These parameters are derived from the pixel distributions in each class (0-9) and are part of the **Beta distribution** used in my Naive Bayes classifier. Thus, the **mean alpha and beta values** I calculated reflect how the pixel intensities for each class in your best fold are distributed in the context of the Beta distribution.

K-NN (results [Without defining K and without applying PCA])

Description and Analysis of Results: Custom k-NN (Euclidean Distance)
Accuracy

- The accuracy across the 10 folds ranged from **89.90%** (Fold 1) to **93.10%** (Fold 2).
- The **mean accuracy** was **91.23%** , which is relatively high, but with some variation between folds.
- The **standard deviation of accuracy** was **0.87%** , indicating that while the model performed well, there was some variability in its ability to generalize across different subsets of the data.



Time

- The time taken for each fold ranged from **418.52 seconds** (Fold 1) to **2896.84 seconds** (Fold 10).
- The **total time** for all folds was **16,642.71 seconds** , indicating that the k-NN algorithm, particularly without any dimensionality reduction (like PCA), is computationally expensive due to the need for

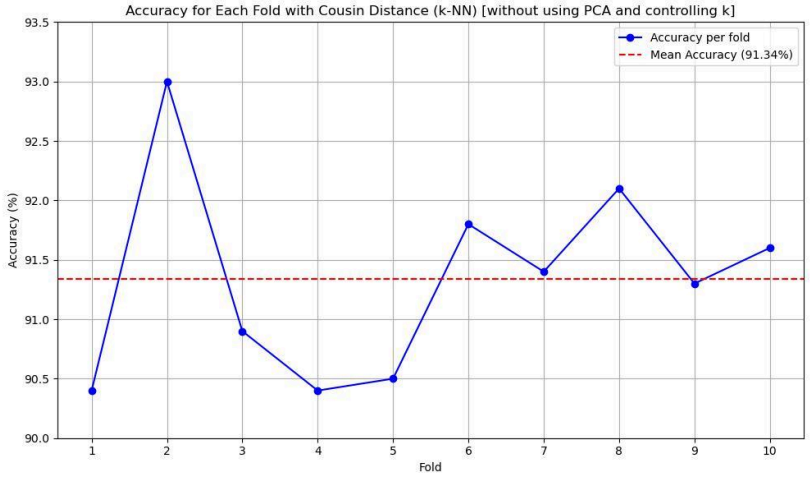
Fold	Accuracy (%)	Time Taken (s)
1.0	89.9	418.52
2.0	93.1	699.12
3.0	91.3	972.67
4.0	90.7	1253.42
5.0	90.6	1526.49
6.0	91.4	1805.21
7.0	91.0	2081.29
8.0	92.4	2357.44
9.0	90.9	2631.72
10.0	91.0	2896.84

pairwise distance calculations between all samples for each test instance.

- The time increases progressively with each fold, which might suggest that the computational load escalates with the growing size of the dataset and the absence of optimizations (like controlling the value of k or reducing dimensions).

Description and Analysis of Results: Custom k-NN (Cosine Similarity)

Accuracy Insights :



- **Mean Accuracy** :
  The accuracy across different folds is consistently high, with a mean accuracy of **91.34%** .
- **Best Accuracy** :
  The best accuracy recorded in any fold was **93.00%** .
- **Accuracy Range** :
  The accuracy in the folds ranged between **90.40%** and **93.00%** , with fluctuations of up to **2.60%** across the 10-fold cross-validation.
- **Standard Deviation of Accuracy** :
  The standard deviation of accuracy is **0.79%** , indicating very low variability in the model's performance across the folds.

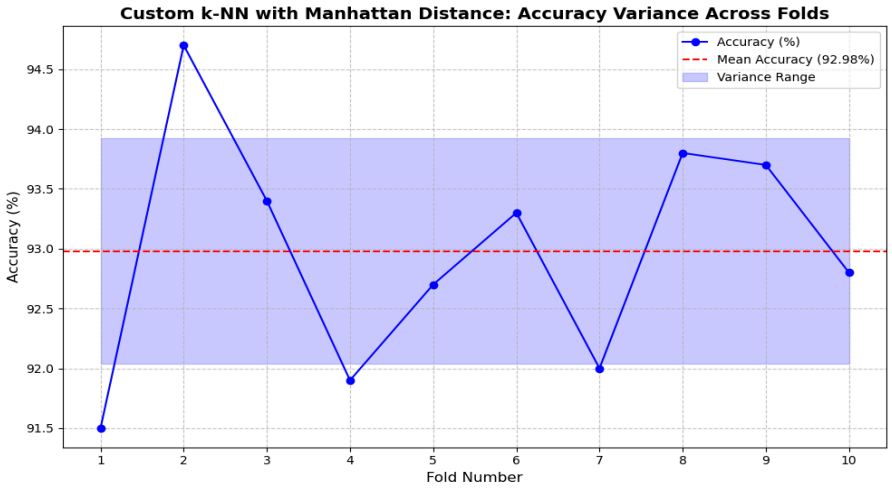
Time Insights :

- **Total Time** :
  The total time for cross-validation was **3479.25 seconds** .
- **Time Range** :
  The time per fold ranged from **62.33 seconds** (minimum) to **631.69 seconds** (maximum), with a steady increase as the number of folds progressed.
- **Mean Time** :
  The mean time to fold is approximately **347.92 seconds** .

Fold	Time (seconds)
Fold 1	62.33
Fold 2	127.82
Fold 3	190.03
Fold 4	249.82
Fold 5	312.83
Fold 6	380.3
Fold 7	444.88
Fold 8	509.06
Fold 9	570.48
Fold 10	631.69

Description and Analysis of Results: k-NN with Manhattan Distance

Accuracy



- The accuracy for each fold ranged from **91.50%** (Fold 1) to **94.70%** (Fold 2).
- The **mean accuracy** across all folds was **92.98%** , which is higher than the k-NN results using both Euclidean (**91.23%**) and Cosine distance (**91.34%**).
- The **standard deviation of accuracy** was **0.94%** , indicating that the model's performance is consistent, but slightly more variable compared to the other distance-based models.

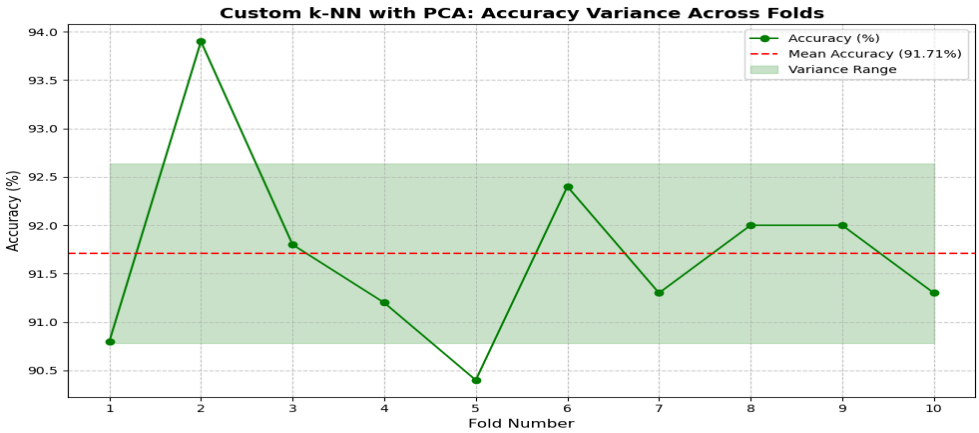
Time

Fold	Accuracy (%)	Time Taken (s)
1.0	91.5	225.07
2.0	94.7	449.87
3.0	93.4	674.65
4.0	91.9	898.45
5.0	92.7	1110.96
6.0	93.3	1289.3
7.0	92.0	1467.1
8.0	93.8	1532.64
9.0	93.7	1591.0
10.0	92.8	1649.05

- The time for each fold ranged from **225.07 seconds** (Fold 1) to **1649.05 seconds** (Fold 10).
- The **total time** for all folds was **10,888.08 seconds** , which is significantly lower than both the Euclidean (**16,642.71 seconds**) and Cosine distance-based k-NN models (**16,560.19 seconds**).
- This indicates that the Manhattan distance-based k-NN model has a faster processing time compared to the other distance metrics.

K-NN (results [Without defining K, but applying PCA])

Description and Analysis of Results: k-NN with PCA using Euclidean Distance



Accuracy

- The accuracy for each fold ranged from **90.40%** (Fold 5) to **93.90%** (Fold 2).
- The **mean accuracy** across all folds was **91.71%** , which is slightly better than the k-NN using Manhattan distance (**92.98%**) but lower than the results from k-NN without PCA (**93.10%**) using Euclidean distance.
- The **standard deviation of accuracy** was **0.93%** , showing consistency similar to the other k-NN models, with a slightly lower variation than the Manhattan distance-based k-NN.

Time

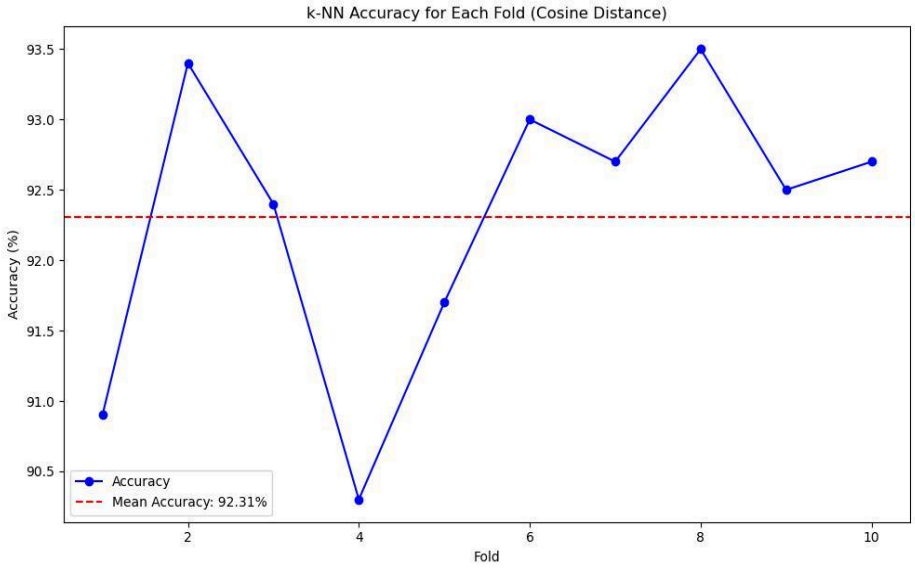
- The time for each fold ranged from **374.16 seconds** (Fold 1) to **2572.50 seconds** (Fold 10).
- The **total time** for all folds was **14,789.23 seconds** , which is faster than both the k-NN models using Euclidean (**16,642.71 seconds**) and Cosine distance (**16,560.19 seconds**), but slower than the Manhattan distance-based model (**10,888.08 seconds**).
- PCA reduces the dimensionality of the dataset (from 784 features to 283 components), which significantly impacts processing time.

Fold	Accuracy (%)	Time Taken (s)
1.0	90.8	374.16
2.0	93.9	630.32
3.0	91.8	871.8
4.0	91.2	1114.85
5.0	90.4	1360.21
6.0	92.4	1600.69
7.0	91.3	1845.26
8.0	92.0	2088.27
9.0	92.0	2331.17
10.0	91.3	2572.5

Description and Analysis of Results: k-NN with PCA using cosine similarity

Accuracy

- The accuracy for each fold ranged from 90.30% (Fold 4) to 93.50% (Fold 8).
- The mean accuracy across all folds was 92.31%, indicating



strong overall performance.

- The best accuracy was 93.50%, achieved in Fold 8, showing that the model can achieve high accuracy under certain conditions.
- The standard deviation of accuracy was 0.99%, reflecting a relatively consistent performance across folds.
- This performance is competitive and suggests that the model with PCA and cosine similarity is effective, but could still be slightly improved compared to k-NN models without PCA or other distance metrics.

Time

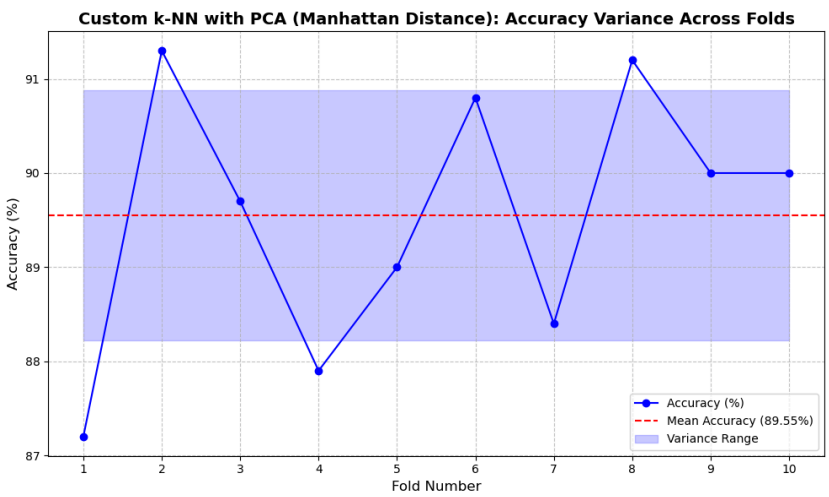
- The time for each fold ranged from 55.32 seconds (Fold 1) to 609.83 seconds (Fold 10).
- The total time for all folds was 3311.96 seconds, which is reasonable given the complexity of processing and the number of folds.
- The time is in a middle range compared to other models, showing a balance between accuracy and processing time.
- The use of PCA (reducing features to 283) helped to speed up the processing compared to the raw 784 features, contributing to the lower total time.

Fold	Accuracy (%)
1.0	55.32
2.0	117.19
3.0	177.31
4.0	238.18
5.0	297.16
6.0	360.45
7.0	421.39
8.0	486.86
9.0	548.27
10.0	609.83

Description and Analysis of Results: k-NN with PCA using Manhattan Distance (without controlling k)

Accuracy

- The accuracy for each fold ranged from **87.20%** (Fold 1) to **91.30%** (Fold 2).
- The **mean accuracy** across all folds was **89.55%** , which is lower than the results from k-NN using Euclidean distance (91.23%) and even the Manhattan distance-based k-NN without PCA (92.98%).
- The **standard deviation of accuracy** was **1.33%** ,



indicating more variability in the results compared to models that didn't use PCA.

Time

- The time for each fold ranged from **191.39 seconds** (Fold 1) to **1910.16 seconds** (Fold 10).
- The **total time** for all folds was **10,528.02 seconds** , which is the fastest among the k-NN models we've tested, likely due to the dimensionality reduction applied by PCA (from 784 to 283 features).

Fold	Accuracy (%)	Time Taken (s)
1.0	87.2	191.39
2.0	91.3	384.87
3.0	89.7	577.12
4.0	87.9	766.06
5.0	89.0	958.87
6.0	90.8	1149.18
7.0	88.4	1339.68
8.0	91.2	1530.18
9.0	90.0	1720.51
10.0	90.0	1910.16

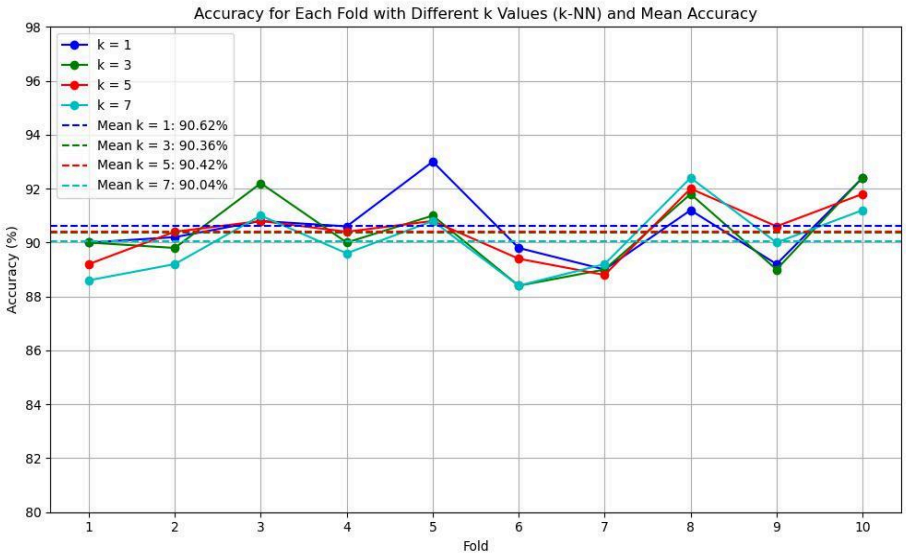
K-NN (results [defining K, applying PCA])

Description and Analysis of Results: k-NN with PCA using Euclidean distance and controlling k

Accuracy Insights :

Mean Accuracy :

- The accuracy across different configurations of k is consistently high, with a range between **90.04%** and **90.62%** .
- **Best Accuracy** :
- The best accuracy recorded in any fold across all k values was **93.00%** for **k = 1** .



- **Accuracy Range** :
- The accuracy in the folds ranged between **88.40%** and **93.00%** , with fluctuations of up to **4.60%** across the 10-fold cross-validation.
- **Standard Deviation of Accuracy** :
- The standard deviation of accuracy ranges from **0.99%** to **1.35%** , indicating relatively low variability in the model's performance across different folds.

Time Insights :

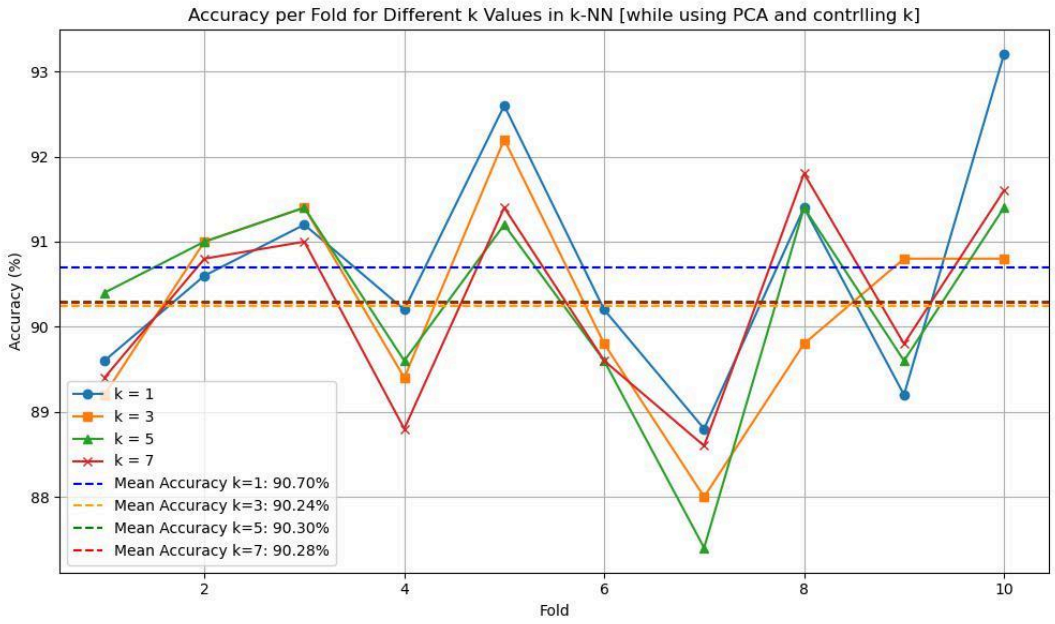
- **Total Time** :
- The total time for cross-validation varied significantly depending on the k value. The total times were:
    - **996.58 seconds** for k = 1
    - **1763.53 seconds** for k = 3
    - **1718.28 seconds** for k = 5
    - **1788.83 seconds** for k = 7

Fold	k = 1	k = 3	k = 5	k = 7
Fold 1	16.71	28.42	29.03	30.19
Fold 2	34.61	59.41	58.52	61.42
Fold 3	54.24	95.1	87.73	93.99
Fold 4	72.06	128.33	120.57	128.64
Fold 5	86.66	163.88	151.89	164.77
Fold 6	102.47	195.55	187.17	197.83
Fold 7	120.41	225.34	221.27	229.88
Fold 8	141.92	254.83	253.67	262.84
Fold 9	170.73	289.71	286.92	294.38
Fold 10	196.78	322.98	321.51	324.89

Description and Analysis of Results: k-NN with PCA using cosine similarity and controlling k

Accuracy Insights :

- **Mean Accuracy** : The accuracy across different configurations of k is consistently high, with a range between **90.24%** and **90.70%** .
- **Best Accuracy** : The best accuracy recorded in any fold across all k values was **93.20%** for **k = 1** .



- **Accuracy Range** : The accuracy in the folds ranged between **88.00%** and **93.20%** , with fluctuations of up to **5.20%** across the 10-fold cross-validation.
- **Standard Deviation of Accuracy** : The standard deviation of accuracy ranges from **1.12%** to **1.35%** , indicating relatively low variability in the model's performance across different folds.

Time Insights :

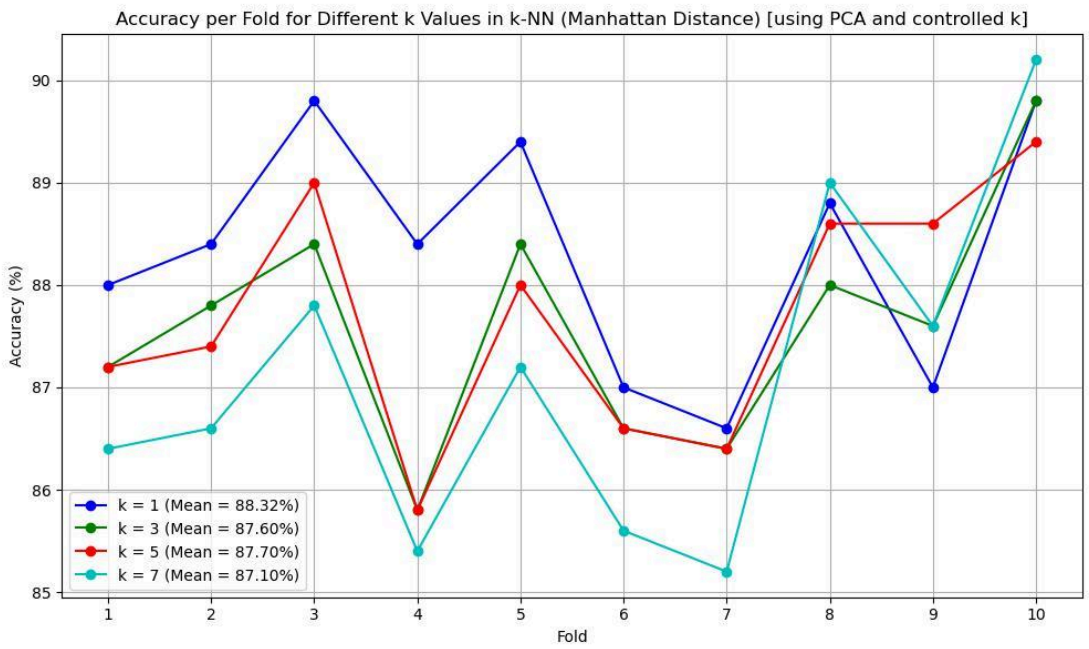
- **Total Time** :
  The total time for cross-validation varied depending on the k value. The total times were:
  - **2427.95 seconds** for **k = 1**
  - **2293.83 seconds** for **k = 3**
  - **2325.38 seconds** for **k = 5**
  - **2117.51 seconds** for **k = 7**

Fold	k = 1	k = 3	k = 5	k = 7
Fold 1	44.48	43.89	43.32	46.25
Fold 2	91.38	83.41	80.92	84.21
Fold 3	133.07	123.1	127.45	129.23
Fold 4	179.71	164.66	176.51	174.15
Fold 5	221.75	206.17	220.18	218.35
Fold 6	262.18	251.79	257.1	245.5
Fold 7	305.56	298.02	301.26	269.44
Fold 8	349.88	333.21	332.09	292.93
Fold 9	397.88	378.76	372.52	317.46
Fold 10	442.07	410.81	414.01	340.0

Description and Analysis of Results: k-NN with PCA using manhattan distance and controlling k

Accuracy Insights :

Mean Accuracy :
The accuracy across different configurations of k is consistently high, with a range between **87.10%** and **88.32%** .



- **Best Accuracy** :
  The best accuracy recorded in any fold across all k values was **90.20%** for **k = 7** .
- **Accuracy Range** :
  The accuracy in the folds ranged between **85.20%** and **90.20%** , with fluctuations of up to **4.00%** across the 10-fold cross-validation.
- **Standard Deviation of Accuracy** :
  The standard deviation of accuracy ranges from **1.11%** to **1.53%** , indicating relatively low variability in the model's performance across different folds.

Time Insights :

- **Total Time** :
  The total time for cross-validation varied depending on the k value. The total times were:
  - **884.04 seconds** for **k = 1**
  - **885.99 seconds** for **k = 3**
  - **925.79 seconds** for **k = 5**
  - **923.72 seconds** for **k = 7**

	k=1	k=3	k=5	k=7
Fold 1	16.09	16.17	16.45	16.61
Fold 2	31.83	32.08	33.16	33.4
Fold 3	47.89	47.77	50.77	50.72
Fold 4	63.57	63.66	67.63	67.56
Fold 5	79.23	80.31	84.27	84.17
Fold 6	95.72	96.46	101.22	100.97
Fold 7	111.79	112.29	118.0	118.11
Fold 8	129.54	129.0	134.47	134.61
Fold 9	145.87	145.82	151.33	150.78
Fold 10	162.5	162.43	168.49	166.79

Key Insights and Conclusions for k-NN:

1. Effect of Distance Calculation Method on Accuracy and Time:

- Accuracy:
  - **Euclidean Distance:** The k-NN model using Euclidean distance without PCA achieves a mean accuracy of **91.23%** and a relatively small standard deviation of **0.87%** . However, the accuracy fluctuates across folds (89.90% to 93.10%).
  - **Cosine Similarity:** The model using cosine similarity achieves a similar mean accuracy of **91.34%** , with a slightly lower standard deviation (**0.79%**), indicating more consistent performance across folds.
  - **Manhattan Distance:** This method offers the highest mean accuracy at **92.98%** , with a slightly higher standard deviation (**0.94%**). Despite this, it outperforms both Euclidean and Cosine methods in terms of accuracy.
- Time:
  - **Euclidean Distance:** The time is the most computationally expensive, ranging from **418.52** to **2896.84 seconds** , totaling **16,642.71 seconds** across all folds. This suggests high processing demands due to pairwise distance calculations for each test instance.
  - **Cosine Similarity:** The time is more efficient, ranging from **62.33** to **631.69 seconds** , with a total of **3479.25 seconds** , reflecting the fact that cosine similarity is computationally less intensive than Euclidean distance.
  - **Manhattan Distance:** Manhattan distance-based k-NN is the fastest, with the total time of **10,888.08 seconds** , ranging from **225.07** to **1649.05 seconds** per fold. This suggests that Manhattan distance computations are less demanding than Euclidean and Cosine distance.

2. Effect of Choosing k:

- Accuracy:
  - **k-NN without PCA:** The accuracy is consistent across folds, but there is a slight variation depending on the k value. Using **k = 1** yields the best accuracy (93.00%), but increasing k slightly decreases accuracy in all models.

    - **k-NN with PCA:** Controlling k further affects the performance, with accuracies ranging from **88.40%** to **93.00%** depending on k values. The variability of accuracy is also higher when controlling k, suggesting the choice of k plays a significant role in performance.
- Time:
    - **Without PCA:** The time required for each fold increases with k, with **k = 1** being the fastest (mean ~ **99.66 seconds**) and **k = 7** being slowest (~ **178.88 seconds**).
    - **With PCA:** The time also increases with k, although the difference between k-values is less significant compared to the non-PCA case. The time for **k = 1** is still the fastest (~ **242.79 seconds**), while **k = 7** has a mean time of **211.75 seconds**, showing that controlling k can impact time, but PCA helps in reducing the overall processing time compared to models without PCA.

3. Effect of Applying PCA:

- Accuracy:
    - **With PCA:** The accuracy is slightly lower for models with PCA compared to those without PCA, especially when using **Euclidean Distance** and **Manhattan Distance**. The mean accuracy drops from **93.10%** (without PCA) to **91.71%** (with PCA using Euclidean), but still remains competitive.
    - **PCA and Cosine Similarity:** The use of PCA and cosine similarity maintains high accuracy (mean of **92.31%**), which is only slightly lower than models without PCA.
- Time:
    - **With PCA:** Dimensionality reduction via PCA significantly reduces the computational load. The total time for **Euclidean distance-based k-NN with PCA** is **14,789.23 seconds**, which is faster than the **16,642.71 seconds** for the non-PCA Euclidean model.
    - **Cosine Similarity and PCA:** The total time for **k-NN with PCA using cosine similarity** is **3311.96 seconds**, indicating that PCA helps significantly speed up the process. Even with PCA, the model using cosine similarity remains computationally efficient.

4. General Observations:

- **PCA Effect:** PCA generally speeds up processing, particularly in the case of models using high-dimensional features (e.g., 784 features). The dimensionality reduction from 784 features to 283 components contributes to reducing computational time, making it particularly useful when dealing with large datasets.
- **k Selection:** While the choice of k can affect accuracy and time, the best results often occur with smaller k values (e.g., k = 1), although larger k values generally result in more stable performance with reduced variance. The increase in k can lead to higher computation times, especially in models without PCA.
- **Distance Metric:** Manhattan distance tends to offer the best trade-off between accuracy and computational efficiency, while cosine similarity provides high consistency across

folds with moderate computational requirements. Euclidean distance is accurate but computationally expensive.

Conclusion for k-NN:

- **Accuracy:** Manhattan distance generally provides the best accuracy, followed by cosine similarity and Euclidean distance.
- **Time Efficiency:** Manhattan distance is the most time-efficient, followed by cosine similarity and then Euclidean distance. PCA improves processing time but may slightly decrease accuracy.
- **PCA:** Applying PCA reduces the computational load significantly, especially when dealing with large datasets. While it can slightly reduce accuracy, it offers a better balance between performance and time efficiency.

Results comparison of all classifiers

In this section we will see all results in one picture to get more insights.

Best results of all models in terms of accuracy and time of execution of all models

Here are the best accuracy and best time taken for each algorithm based on the results provided:

Best Accuracy:

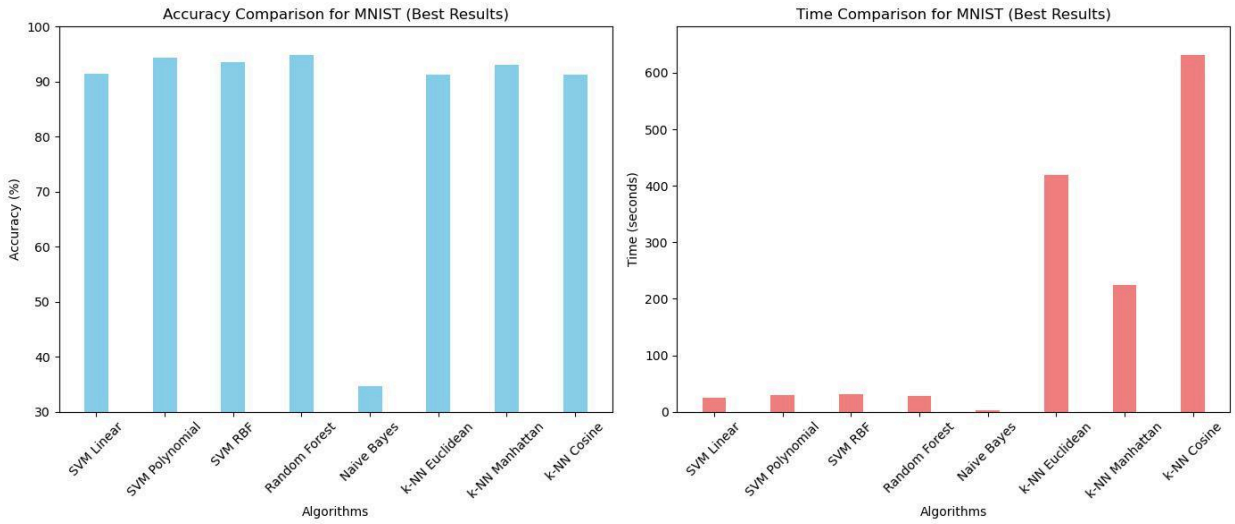
- **SVM with Polynomial Kernel (Degree 2)** :
  - Best Accuracy: **96.00%**
- **Random Forest** :
  - Best Accuracy: **96.00%** (Fold 8)
- **Custom Naive Bayes** :
  - Best Accuracy: **38.10%** (Fold 5)
- **k-NN (Manhattan Distance)** :
  - Best Accuracy: **92.98%**

Best Time Taken:

- **SVM with Linear Kernel** :
  - Best Time: **19.94 seconds** (Fold 9)
- **SVM with Polynomial Kernel (Degree 2)** :
  - Best Time: **21.94 seconds** (Fold 10)
- **SVM with RBF Kernel** :
  - Best Time: **23.69 seconds** (Folds 9 and 10)
- **Random Forest** :
  - Best Time: **27.81 seconds** per fold (Total: 278.09 seconds)
- **Custom Naive Bayes** :
  - Best Time: **2.00 seconds** (Fold 3)

- **k-NN (Manhattan Distance)** :
    - Best Time: **225.07 seconds** (Fold 3)

Comparing accuracy and time of execution for all models in one picture



Accuracy:

- **Best Performing Algorithms** : The **Random Forest** classifier achieved the highest accuracy, with a mean of **94.83%** , followed closely by the **Polynomial Kernel SVM** at **94.30%** and the **RBF Kernel SVM** at **93.50%** . These models demonstrated strong performance on the MNIST dataset, with Random Forests showing particularly consistent results across folds.
- **Naive Bayes** showed the lowest accuracy at **34.68%** , indicating that this model struggles with the MNIST dataset, likely due to its assumptions of feature independence which do not hold well for image data.
- **k-NN** (Euclidean, Manhattan, Cosine) models also performed well, with the **Manhattan Distance-based k-NN** achieving the highest accuracy of **92.98%** .

Time:

- **Best Time Efficiency** : The **Naive Bayes** model was the fastest, with an average computation time of just **2.33 seconds** per fold, followed by the **k-NN with Manhattan Distance** at **225.07 seconds** and **Cosine Similarity-based k-NN** at **631.69 seconds** .
- On the other hand, the **Euclidean Distance k-NN** was the most computationally expensive, with total times ranging from **418.52 to 2896.84 seconds** , indicating its high processing demand.

- The **SVM Linear Kernel** was the fastest of the SVM variants, taking **25.25 seconds** per fold, while the **RBF** and **Polynomial Kernels** took longer (30.81 and 29.87 seconds, respectively), reflecting their increased model complexity.

Trade-Off Between Accuracy and Time:

- The **Random Forest** classifier delivered the highest accuracy but required a moderate amount of time (27.81 seconds per fold), which is expected due to its ensemble nature.
- The **SVM Polynomial Kernel** , while offering excellent accuracy (94.30%), was slower than the Linear Kernel, taking **29.87 seconds** per fold.
- **k-NN** using **Manhattan Distance** provided the best trade-off between accuracy and time among the distance-based models, showing high accuracy (92.98%) while maintaining reasonable computational efficiency (225.07 seconds per fold).

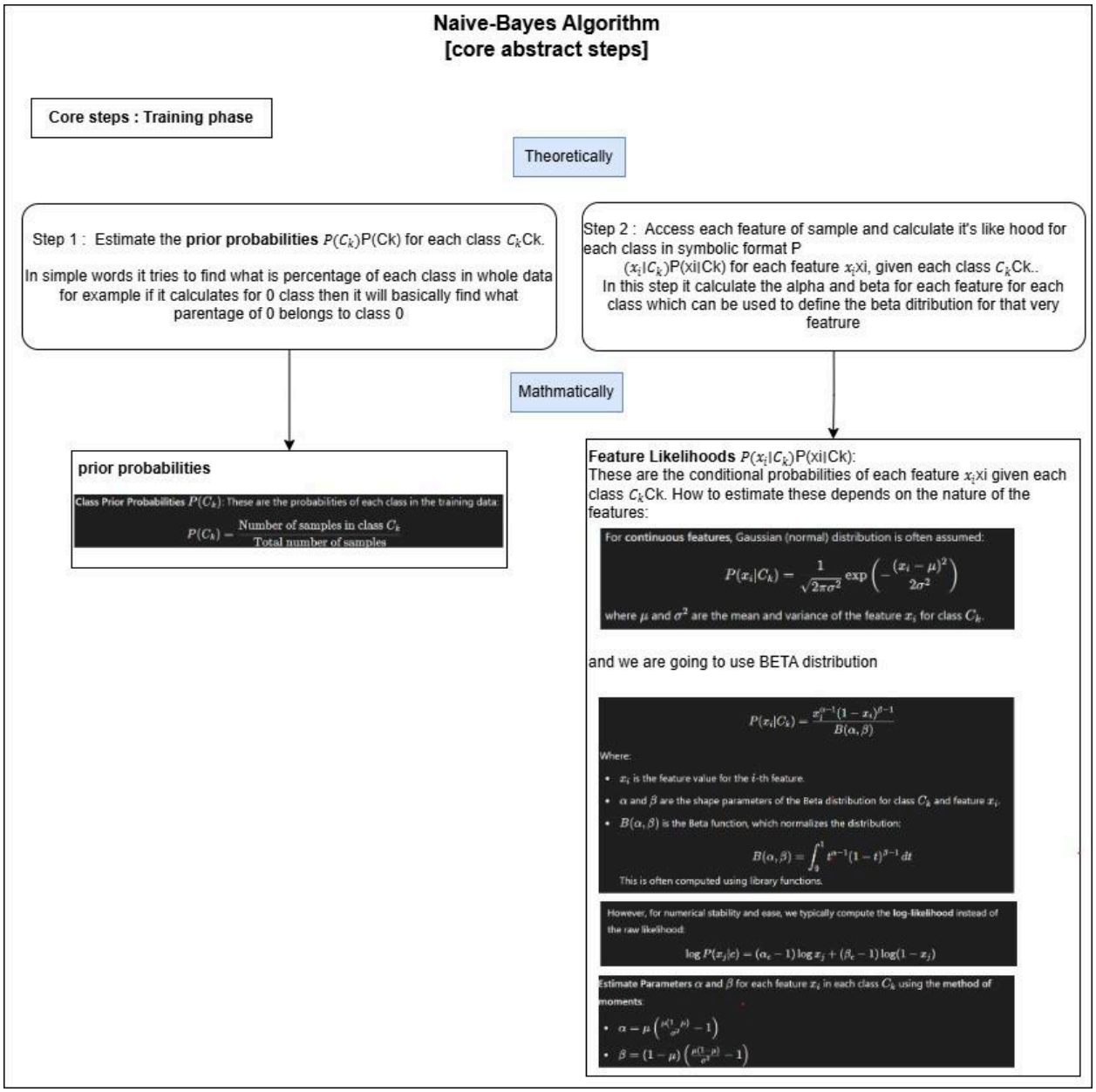
Best Choice for MNIST:

Based on the above comparison, if **accuracy** is the top priority, **Random Forest** stands out as the best choice due to its **94.83% accuracy**. However, if **time efficiency** is more critical, **k-NN with Manhattan Distance** would be a good option, as it balances accuracy (**92.98%**) with faster execution times compared to other models.

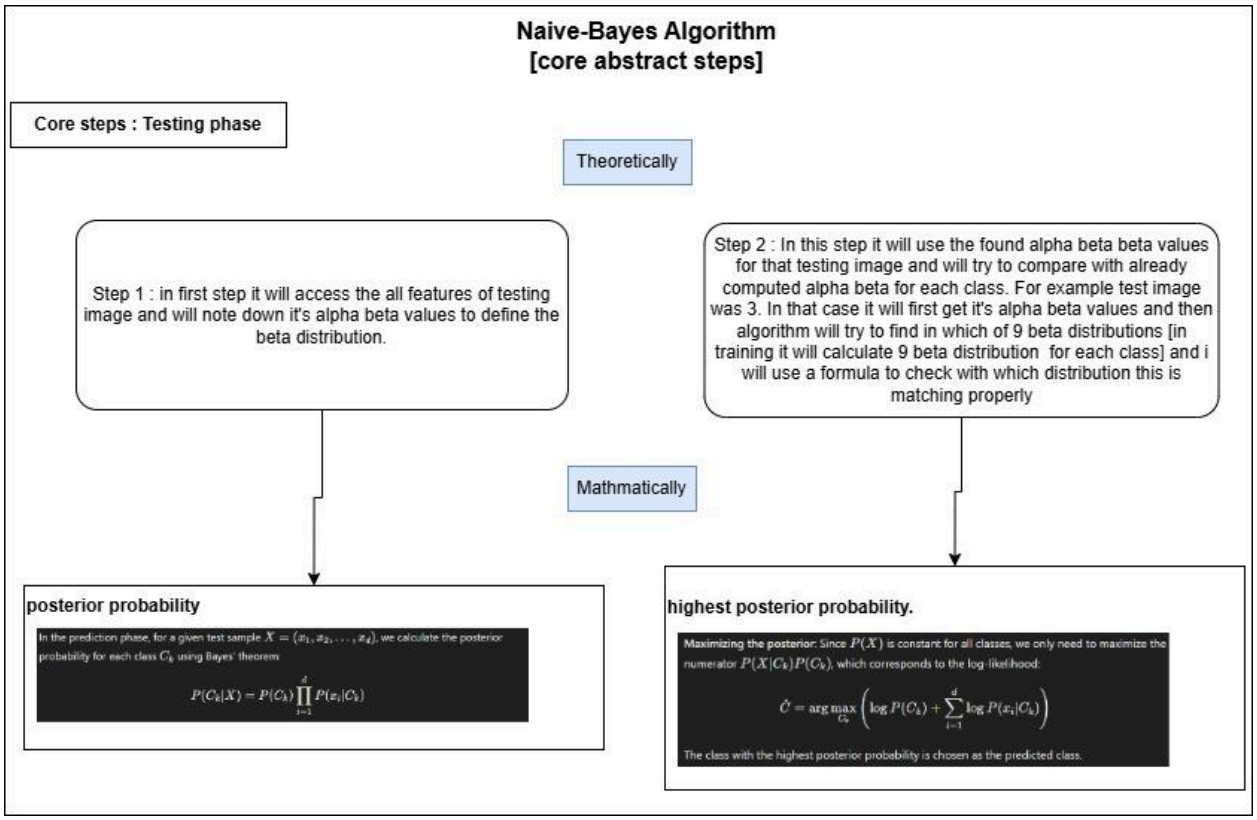
In conclusion, the choice between **Random Forest** and **k-NN with Manhattan Distance** largely depends on the desired balance between **accuracy** and **computation time**. If computational resources are not a major concern, **Random Forest** is the best option, while for faster results, **k-NN with Manhattan Distance** would be preferable.

Implementation of custom Naive Bayes Classifier

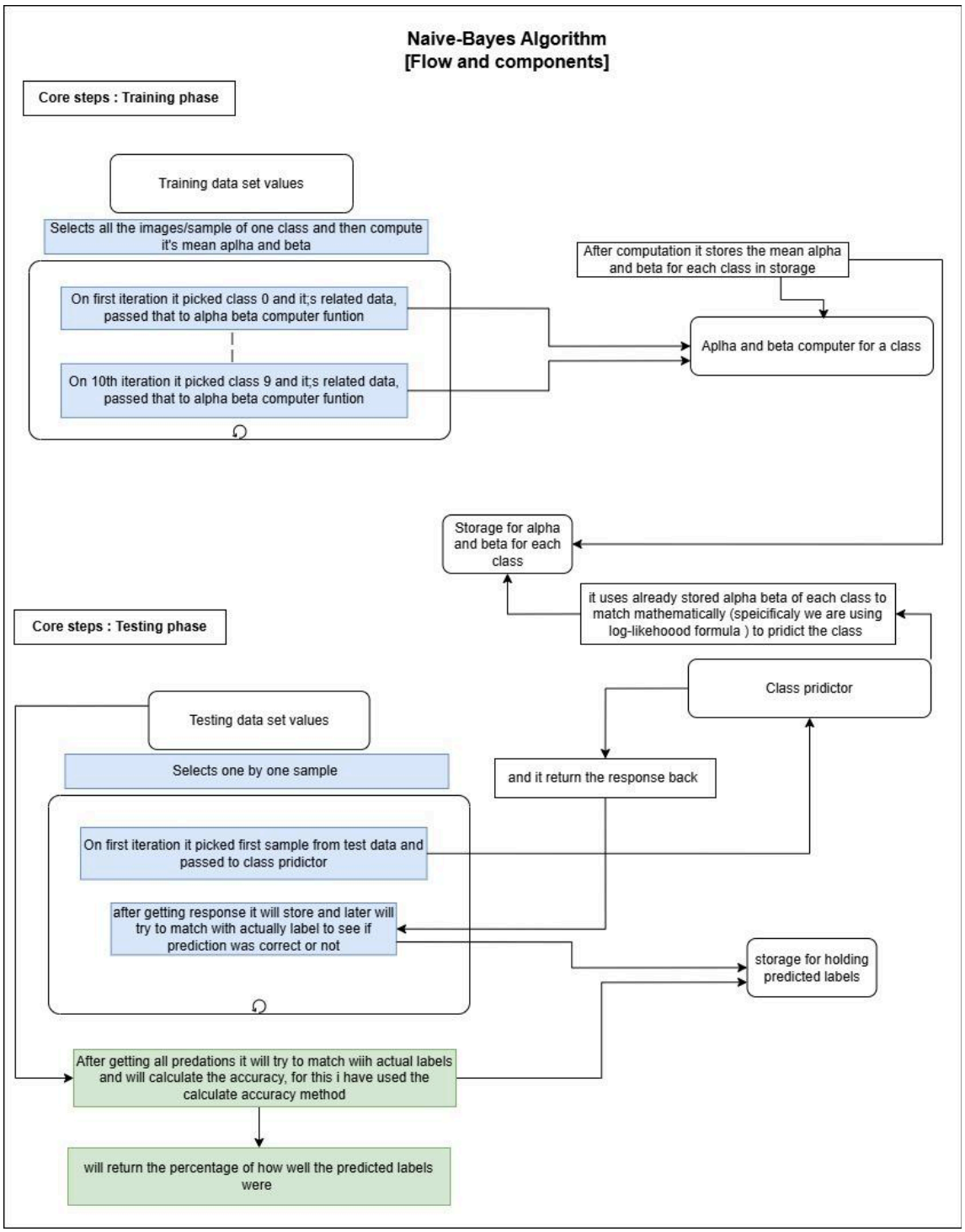
Following diagram shows the core steps of the training process of naive bayes (theoretical and mathematical concepts). The first step is about calculating the prior probabilities of each class and storing them for later usage in calculating the likelihood. And in the second step we have to calculate the alpha and beta for each feature across all the images of all classes and after that mean alpha and beta are calculated for each class.



Following diagram shows the core steps involved in the testing phase (theoretical and mathematical concepts). While testing or trying to make prediction of class for a given sample image, classifiers first get mean alpha beta calculated for each feature of that image features and then by using log-likelihood formula it calculates the likelihood with all of the classes and pick the label one with highest similarity.

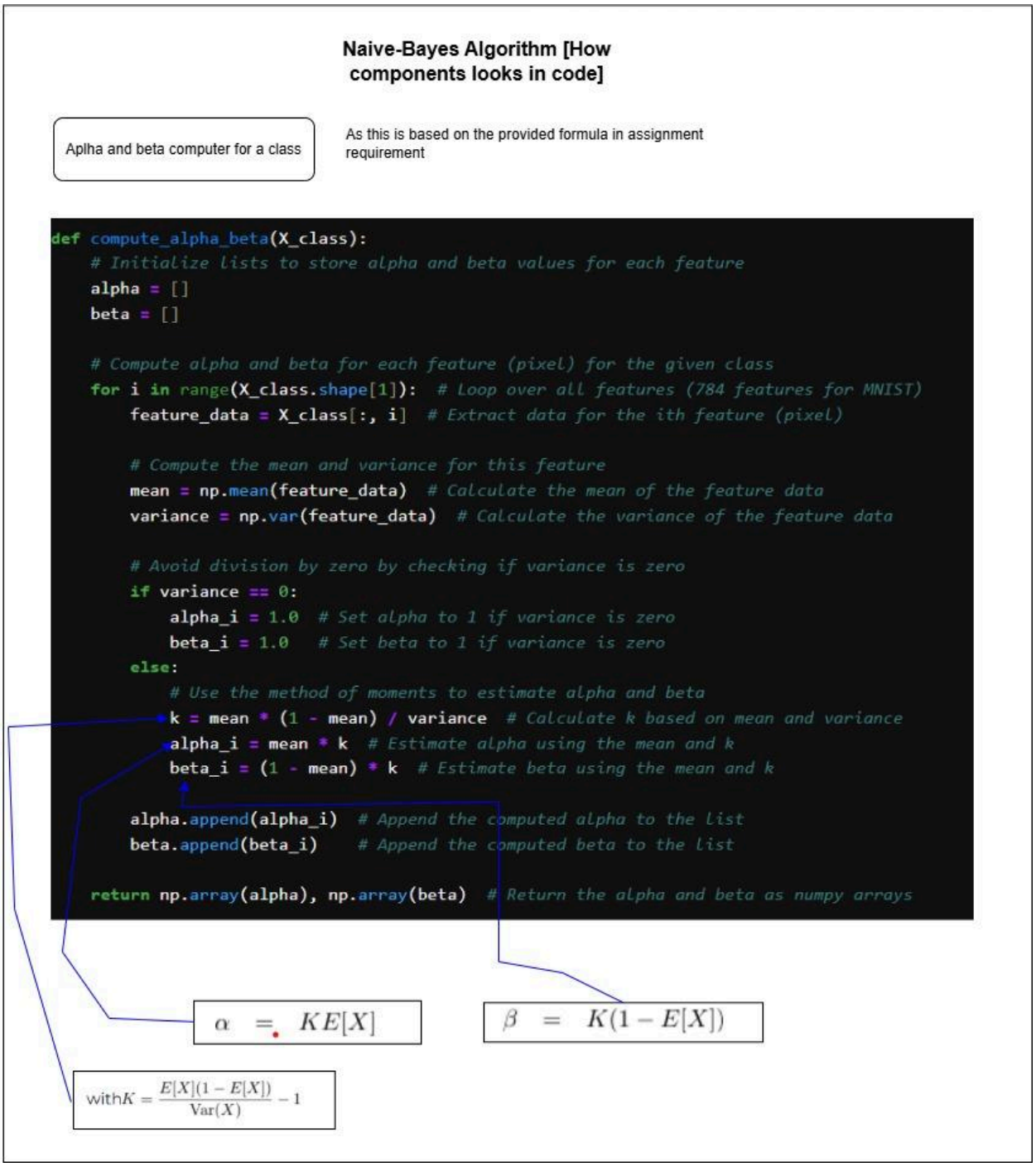


Following diagram shows the complete actual flow of execution of code I have implemented for Naive bayes. It includes core components like two functions one is for calculating the mean alpha beta for a given image for all the features and storing them in globally declared variables. Another core function is one that predicts the class label of a given sample image by using a log-likelihood formula.



Following diagram shows the implementation of key components (alpha beta computer and class predictor) and actual code implementation I have used. I have tried to map the requirements by arrows (look at the calculation of alpha and beta, they are exactly counted based on given formula)

Implementation of alpha beta computer function.



Implementation of alpha beta computer function.

Naive-Bayes Algorithm [How components looks in code]

Class predictor

As this is based on the provided formula in assignment requirement

```python
def predict_class(sample, alpha_dict, beta_dict, class_priors):
    max_log_prob = -np.inf  # Initialize maximum log probability to negative infinity
    predicted_class = -1  # Initialize predicted class to an invalid value
    epsilon = 1e-10  # Small constant to avoid log(0)

    # Ensure sample values are between epsilon and 1 - epsilon to avoid log(0) or log(1)
    sample = np.clip(sample, epsilon, 1 - epsilon)

    # Iterate over each class label (0 to 9)
    for class_label in range(10):
        alpha = alpha_dict[class_label]  # Retrieve alpha parameter for the current class
        beta = beta_dict[class_label]  # Retrieve beta parameter for the current class

        # Compute log-likelihood for the current class using the beta-binomial distribution
        log_likelihoods = (alpha - 1) * np.log(sample) + (beta - 1) * np.log(1 - sample)

        # Calculate total log probability by summing log-likelihoods and adding the log of class pri
        total_log_prob = np.sum(log_likelihoods) + np.log(class_priors[class_label])

        # Update the predicted class if the current total log probability is greater than the maximu
        if total_log_prob > max_log_prob:
            max_log_prob = total_log_prob  # Update maximum log probability
            predicted_class = class_label  # Update predicted class to the current class label

    return predicted_class  # Return the predicted class
```

given PDF for Beta-distribution

$$d(x;a,b) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} x^{\alpha-1}(1-x)^{\beta-1}$$

By using Chat-GPT i reformulated to be used in code and got log-like hood which is equivalent in functionality we want to achieve (predicting class)

$$P(x_i|C_k) = \frac{x_i^{\alpha-1}(1-x_i)^{\beta-1}}{B(\alpha,\beta)}$$

Where:

- x_i is the feature value for the i-th feature.
- α and β are the shape parameters of the Beta distribution for class C_k and feature x_i.
- $B(\alpha,\beta)$ is the Beta function, which normalizes the distribution:

$$B(\alpha,\beta) = \int_0^1 t^{\alpha-1}(1-t)^{\beta-1}\,dt$$

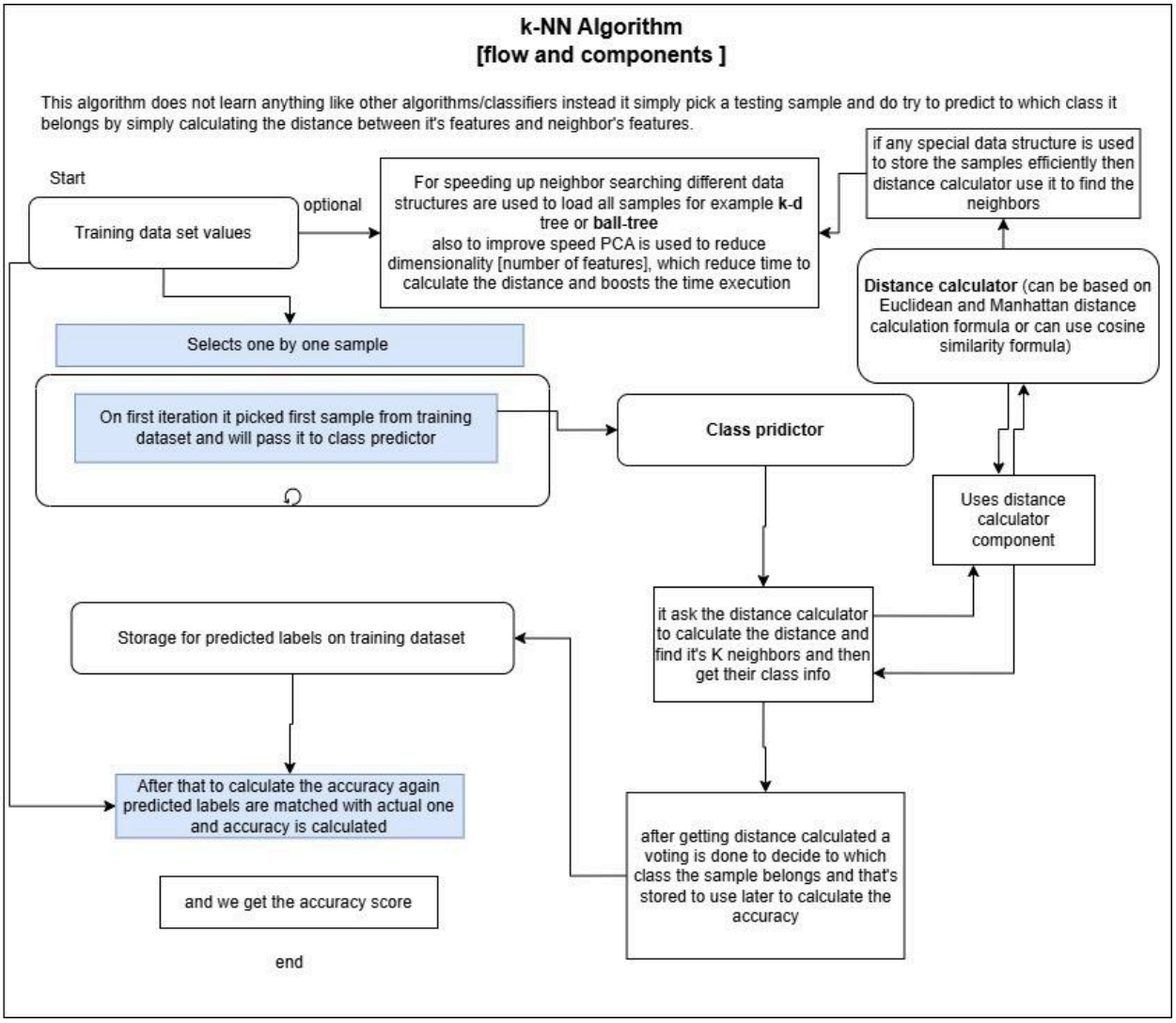
This is often computed using library functions.

However, for numerical stability and ease, we typically compute the **log-likelihood** instead of the raw likelihood:

$$\log P(x_j|c) = (\alpha_c - 1)\log x_j + (\beta_c - 1)\log(1-x_j)$$

Implementation of custom k-NN Classifier

Following diagram shows the flow of code I have implemented for k-nn classifiers and flow of usage of each component it involves. In this diagram I have mentioned PCA and usage of trees for efficiency. I have built-in libraries for both inside my custom code to observe the effect In results. You can find the detailed implementation in code files.



Following diagram shows what these components look like in code. It shows how the distance is calculated by all three methods and also how the k-nn class predictor function looks in code

Conclusion

To sum up, this study began by investigating the mathematical and theoretical underpinnings of classification before putting both discriminative and generative classifiers into practice. We looked closely at each classifier's specifics and evaluated the 10-fold cross-validation results.

It was evident from the data that performance was significantly impacted by the model selection and parameters. For example, SVM illustrated how various kernels could impact outcomes, whereas k-NN illustrated the significance of choosing suitable k-values and the impact of PCA. Although this was more obvious in the theoretical aspects and the underlying code, we also considered how performance could be slightly impacted by the data structures used to store datasets.

SVM and Random Forest were the best classifiers for the MNIST_784 dataset, both of which achieved high accuracy when we compared them. The trade-off between model complexity and execution time was highlighted by the discovery that, despite its strong performance, Random Forest required more computing time than SVM.

The built-in Naive Bayes model was more accurate than Custom Naive Bayes, which used a Beta distribution. This disparity implies that the Gaussian distribution, which is commonly employed in Naive Bayes, suited the MNIST dataset more successfully than the Beta distribution. The fact that Naive Bayes was the fastest classifier in spite of its lower accuracy highlights how effective it is in circumstances where speed is more important than accuracy.

Lastly, while k-NN with Manhattan distance demonstrated respectable accuracy, it was less appropriate for huge datasets such as MNIST due to its high computational cost. This demonstrates the difficulty of applying distance-based models to massive data sets.

All things considered, this investigation provided important new information about how each classifier's execution time and accuracy are traded off. It emphasized how crucial it is to choose the best model for the given issue, taking into account both the required level of accuracy and the available computing power.