**Identifying Handwriting Using Decision-Tree, Naïve Bayes, Random Forest, and Support Vector Machines**

David Caspers

2024-08-29

**Introduction**

In the fast-paced world of logistics and postal services, efficiency and accuracy are paramount. Every day, millions of packages and letters are processed and delivered across the country. At the heart of this massive operation is the ability to quickly and accurately read and interpret handwritten addresses on packages and mail. This is where handwriting recognition technology plays a crucial role, enabling postal services to streamline their operations, reduce errors, and ensure timely deliveries.

One prominent example of how handwriting recognition is utilized is within the United States Postal Service (USPS). USPS has implemented advanced systems that automatically scan and process handwritten addresses, dramatically reducing the time it takes to sort and route mail. This technology is also integral to services like USPS Informed Delivery, which provides customers with a digital preview of their incoming mail. Through Informed Delivery, images of the front of mail pieces are captured and sent to recipients, allowing them to see what will arrive in their mailbox each day. The ability to accurately recognize handwritten addresses on these mail pieces is critical to the success of such services.

Given the widespread use and importance of handwriting recognition, it is essential to develop models that are both accurate and efficient. This project focuses on recognizing handwritten digits, a fundamental task within the broader scope of handwriting recognition. By analyzing various approaches to this problem, we aim to identify the most effective methods for accurate and reliable digit recognition, which could potentially enhance a multitude of applications that rely on this technology.

**Exploratory Analysis**

**About the data**

The dataset used in this project is derived from the Kaggle Digit Recognizer competition, which is based on the MNIST database of handwritten digits. The dataset consists of images representing digits from 0 to 9. Each image is a 28x28 pixel grid, resulting in 784 features per image. These pixel values are grayscale, with a range from 0 (white) to 255 (black).

**Understanding the Pixel Representation**

As described in the Kaggle dataset repository:

"The data files train.csv and test.csv contain gray-scale images of hand-drawn digits, from zero through nine.

Each image is 28 pixels in height and 28 pixels in width, for a total of 784 pixels in total. Each pixel has a single pixel-value associated with it, indicating the lightness or darkness of that pixel, with higher numbers meaning darker. This pixel-value is an integer between 0 and 255, inclusive.

The training data set, (train.csv), has 785 columns. The first column, called 'label', is the digit that was drawn by the user. The rest of the columns contain the pixel-values of the associated image.

Each pixel column in the training set has a name like pixelx, where x is an integer between 0 and 783, inclusive. To locate this pixel on the image, suppose that we have decomposed x as x = i \* 28 + j, where i and j are integers between 0 and 27, inclusive. Then pixelx is located on row i and column j of a 28 x 28 matrix, (indexing by zero).

For example, pixel31 indicates the pixel that is in the fourth column from the left, and the second row from the top, as in the ascii-diagram below:

A number with numbers on it

Description automatically generated with medium confidence

**Data Preparation & Visualization**

Data preprocessing involved several key steps. Initially, the dataset was checked for missing values, confirming that all entries were complete. The labels, which indicate the digit in each image, were converted into categorical factors to facilitate the classification tasks.

To ensure that the dataset was balanced across all digit classes, an analysis of the distribution of the labels was conducted. The distribution plot shows that the dataset is fairly balanced, with each digit class (from 0 to 9) having a similar number of samples, ranging from approximately 3,800 to 4,500. This balance is important because it ensures that each class has enough representation for training the models effectively. If certain digits were underrepresented, they could be considered rare classes, which would require special handling, such as resampling techniques or model adjustments, to avoid biased predictions and poor performance for those underrepresented digits.

A colorful bar chart with text

Description automatically generatedAdditionally, visualizing the digit images helps verify the dataset's accuracy and provides insight into the variety and style differences within each digit class. An example set of images from the dataset, displaying digits from 0 to 9, is shown below:

A number symbols with text

Description automatically generated with medium confidence

These images demonstrate the range of handwriting styles captured in the dataset, from clear and well-formed digits to more stylized or ambiguous examples. This visualization underscores the challenges the models face in distinguishing between similar-looking digits.

Given that some of the models applied in this analysis assume a particular distribution of pixel values, exploratory data analysis was conducted by selecting several randomly chosen pixels to visually assess their distributions. Given the large number of pixels (784) in the dataset, it was infeasible to inspect each pixel individually. However, the histograms for the selected pixels, as shown below, indicate that their distributions are highly skewed, with most pixel values concentrated at the extremes (0 or 255).

A graph of a number of data

Description automatically generated with medium confidence

This lack of normal distribution suggests that the Gaussian Naive Bayes assumption of normally distributed features may not hold, potentially leading to suboptimal performance when applied to this dataset. Despite these findings, Gaussian Naive Bayes was still applied as part of the analysis, although its performance might be limited due to the violation of its core assumption.

**Models**

**Decision Tree**

**Training**

A Decision Tree model was implemented using the rpart method in R. This model works by recursively splitting the data based on feature values, creating a tree-like structure where each branch represents a decision rule, and each leaf node represents a class label. To optimize the model's performance, a grid search was conducted over the complexity parameter (cp), which controls the depth and pruning of the tree, and various minbucket values, which set the minimum number of observations per leaf node. The model was trained with 3-fold cross-validation to ensure that the results were robust and generalizable.

**Model Tuning:**

To optimize the performance of the Decision Tree, two key parameters were tuned: the complexity parameter (cp) and the minbucket parameter.

* **Complexity Parameter (cp)**: This parameter controls the complexity of the tree by determining the minimum improvement in the model's fit required for a split to be made. A smaller cp value allows for a more complex tree with more splits, potentially capturing more detailed patterns in the data. Conversely, a larger cp value leads to a simpler tree with fewer splits, reducing the risk of overfitting.
* **Minbucket Parameter**: The minbucket parameter sets the minimum number of observations that must be present in any terminal node (leaf) of the tree. A smaller minbucket value allows the tree to grow with more leaves, which can improve accuracy but may also increase the risk of overfitting. A larger minbucket value forces the tree to have fewer leaves, leading to a more generalized model.

To determine the optimal values for the Decision Tree model's parameters, a grid search was conducted, and the model was trained using 3-fold cross-validation. The caret package was utilized to automatically select the highest performing complexity parameter (cp) across the folds. Simultaneously, the cross-fold validated accuracy of each model at different levels of the minbucket parameter was recorded, as shown in the table below. This approach ensures that the model's performance is evaluated on different subsets of the data, providing a robust measure of its generalization ability.A screenshot of a computer

Description automatically generated

After tuning, the optimal parameters were found to be a cp value of 0 and a minbucket size of 5. These settings resulted in a well-balanced tree that effectively captured the structure of the data without overfitting. The table clearly shows that as the minbucket size increased, the tree became simpler with fewer nodes and leaves, but this came at the cost of lower accuracy. The smallest minbucket value of 5 provided the highest accuracy, confirming its effectiveness in this context.

**Results**

The Decision Tree model achieved an overall accuracy of approximately 85% on the cross-validation set. Below is the confusion matrix for the model's out-of-fold predictions:

A screenshot of a computer

Description automatically generated

The confusion matrix and accompanying statistics reveal several insights into how well the model performed in distinguishing between different digits:

* **Strong Classification Performance:**
  + The model exhibited high sensitivity (true positive rate) for digits like 1 (94.58%), 0 (91.75%), and 6 (87.28%). This indicates that the model was particularly effective at correctly identifying these digits when they appeared in the dataset.
  + The model also demonstrated high specificity (true negative rate) across all digits, with values consistently above 97%. For instance, the specificity for digit 0 was 99.02%, meaning that the model rarely misclassified other digits as 0.
* **Frequent Misclassifications:**
  + **Digits 3 and 5**: The model frequently confused these two digits, as reflected by the number of times a 3 was predicted as a 5 and vice versa. This is likely due to the visual similarity between these digits in handwritten form. Specifically, the model incorrectly classified 232 instances of digit 3 as 5.
  + **Digits 8 and 3**: Another common source of confusion was between digits 8 and 3, where the model misclassified 141 instances of digit 8 as 3. This is also understandable, given that parts of these digits can appear similar, especially if the handwritten strokes are incomplete or merged.
* **Balanced Accuracy:**
  + The balanced accuracy, which accounts for both sensitivity and specificity, was reasonably high across most digits. For example, digit 0 had a balanced accuracy of 95.39%, indicating that the model was equally effective in detecting the digit when present and correctly identifying other digits when it was absent.
* **Challenges with Digit 5:**
  + Digit **5** had a relatively lower sensitivity of 78.37%, indicating that the model had more difficulty correctly identifying this digit compared to others. This is further supported by its positive predictive value (the probability that the digit is actually 5 when predicted as 5) being lower at 79.84%. This suggests that while the model could often detect digit 5, it struggled to classify it accurately.
* **High Specificity for Digit 1:**
  + Digit 1 was classified with near-perfect specificity (99.01%), meaning that when the model predicted a digit was not 1, it was almost always correct. The high sensitivity and specificity for digit 1 indicate that this digit was very easy for the model to distinguish from others.

Overall, the model demonstrated strong performance in recognizing most digits, particularly those with distinct shapes like 0, 1, and 6. However, digits with more similar features, such as 3, 5, and 8, posed challenges, leading to more frequent misclassifications.

**Naïve Bayes**

The Naive Bayes algorithm was employed to classify the handwritten digits based on pixel intensity values. Given the assumption that the features (pixels) are normally distributed, Gaussian Naive Bayes was initially applied. However, based on the exploratory data analysis, which indicated that the normal distribution assumption might not hold for all pixel values, a Kernel Naive Bayes model was also evaluated to determine if it could provide better performance by relaxing the normality assumption.

**Gaussian Naïve Bayes**

**Training**

Gaussian Naive Bayes (GNB) was selected as the first model due to its simplicity and effectiveness in classification tasks where the features are normally distributed. The model was trained using 3-fold cross-validation to ensure that the results were robust and generalizable across different subsets of the data. The primary assumption of GNB is that each feature (in this case, each pixel) follows a normal distribution, which influences how the model calculates the probabilities used for classification.

**Results**

The Gaussian Naive Bayes model achieved an overall accuracy of 53% on the cross-validation set. Below is the confusion matrix, which provides a detailed view of how the model performed across different digit classes:

A screenshot of a computer

Description automatically generated

The confusion matrix reveals several key insights:

* **Sensitivity and Specificity:**
  + The model showed high sensitivity for digits 0 (88.77%) and 1 (96.35%), indicating that it was effective at correctly identifying these digits when they appeared in the dataset.
  + Specificity was high across all digits, with values consistently above 95%, meaning that the model was generally good at avoiding false positives for most digits.
* **Frequent Misclassifications:**
  + Digits 3 and 5 were particularly challenging for the model, with the sensitivity for digit 3 being only 34.11% and for digit 5 a mere 4.74%. This suggests that the model struggled significantly to correctly classify these digits, often confusing them with others.
  + The low positive predictive value for digit 5 (60%) further indicates that when the model predicted a digit as 5, it was correct only 60% of the time, highlighting a significant weakness in classifying this digit.
* **Balanced Accuracy:**
  + Balanced accuracy, which considers both sensitivity and specificity, was relatively low for digits like 2 (58.24%) and 5 (52.21%), further emphasizing the model's difficulty in accurately identifying these digits.

The overall performance of the Gaussian Naive Bayes model appears to be limited by the violation of the normal distribution assumption, as indicated by the exploratory data analysis. This led to suboptimal classification, particularly for digits with more complex or variable pixel distributions.

**Kernal Naïve Bayes**

**Training**

Given the limitations observed with Gaussian Naive Bayes, Kernel Naive Bayes (KNB) was also evaluated. Kernel Naive Bayes relaxes the normality assumption by using kernel density estimation to model the distribution of the features, allowing for more flexibility in cases where the true distribution deviates from normality. This approach was expected to better capture the variations in pixel intensities and potentially improve classification accuracy.

**Results**

The Kernel Naive Bayes model achieved an overall accuracy of 39.48% on the cross-validation set. The confusion matrix below provides insight into the model’s performance:

The confusion matrix shows the following:

**A screenshot of a computer

Description automatically generated**

* **Sensitivity and Specificity:**
  + The model had high sensitivity for digit 1 (99.87%), indicating strong performance in detecting this digit. However, the sensitivity for other digits, such as 3 (27.35%) and 5 (0.15%), was significantly lower, indicating challenges in correctly identifying these digits.
  + Specificity remained high across most digits, particularly for digits like 0 (97.78%) and 1 (55.51%). However, the lower specificity for some digits suggests that the model occasionally misclassified other digits as these digits.
* **Misclassification Trends:**
  + The model showed significant misclassification for digit 5, with only 0.15% sensitivity, meaning it almost never correctly identified this digit. This result underscores the difficulties in using Naive Bayes models (even with kernel estimation) when the pixel distributions deviate significantly from the expected form.
  + Digits 3 and 8 also had low balanced accuracy scores, indicating that the model struggled with these digits similarly to the Gaussian Naive Bayes model.

Overall, while Kernel Naive Bayes provided an alternative to Gaussian Naive Bayes, its performance was still limited by the complexity of the pixel distributions. The model struggled to effectively classify digits that deviated significantly from its underlying assumptions, leading to overall lower accuracy compared to Gaussian Naive Bayes.

**Random Forest**

Random Forest is an ensemble learning method used for classification and regression tasks. It builds multiple decision trees during training and merges them to improve predictive accuracy and control overfitting. Each decision tree in a random forest is created from a different bootstrap sample from the original dataset, and at each node, a subset of features is randomly selected to determine the best split. This randomness helps create a diverse set of trees, enhancing the model's robustness and reducing variance. In the context of handwriting recognition, Random Forest can effectively capture the complex patterns and variations in handwriting by leveraging multiple decision trees that individually learn different features from the data.

**Training**

In this implementation, the model was validated using 3-fold cross-validation, which ensured that the model was trained and evaluated on different subsets of the data to evaluate the model's generalization capability more robustly.

**Model Tuning:**

In this particular implementation, the only hyperparameter that the caret package automatically tuned was mtry, which is the number of predictors randomly selected at each split in the decision trees. The Random Forest model in caret does not provide the option to tune other hyperparameters such as the number of trees (ntree) or the minimum node size (nodesize). In Random Forest, trees are typically grown to their maximum possible depth, and nodesize is not a tunable parameter within caret.

The tuning process for mtry involved evaluating the model's performance across different values of this parameter. As depicted in the below graph, the model's accuracy varied with the number of predictors selected randomly at each split. The optimal model was achieved with an mtry of 39, which was determined to best balance the trade-off between model complexity and predictive accuracy.

**A graph with a line

Description automatically generated**

**Results**

The Random Forest model achieved an overall accuracy of **96.45%**, a notable improvement over the decision tree model, albeit with increased computational complexity. The confusion matrix provides several key insights into the model's performance:

* **Sensitivity and Specificity**:
  + The model demonstrated high sensitivity (true positive rate) across all digits, ranging from **94% to 98%**, indicating that it correctly identified most true positives. However, it frequently misclassified the digit **9**, often confusing it with **3** and **4**. Additionally, the digit **3** was commonly misclassified as **2**.
  + The model's specificity (true negative rate) exceeded **99%** for all digits, showing a strong ability to correctly identify true negatives and avoid false positives.
* **Balanced Accuracy**:
  + The balanced accuracy, which accounts for both sensitivity and specificity, was consistently high across all digits. However, the model struggled the most with accurately classifying the digit **9**, reflecting the challenges noted with sensitivity.

Overall, the Random Forest model demonstrated strong performance due to its lack of assumptions about pixel distribution and its ability to train multiple trees to capture diverse patterns in the dataset. This versatility ensures robust generalization to unseen data, making it a powerful tool for handwriting recognition tasks.

**A screenshot of a computer

Description automatically generated**

**K-Nearest Neighbors**

K-Nearest Neighbors (KNN) is a simple yet effective non-parametric classification method used in machine learning. KNN classifies a new data point based on the majority class of it’s ‘K’ nearest neighbors in the training set. This distance metric can be defined by the user though Euclidian distance is commonly utilized to determine the “closeness” of the neighbors. The model can be tuned by selecting the number ‘k’ used to determine the majority class. It is particularly useful for the handwriting contest as it uses the similarity between handwritten digits to classify new ones.

One potential challenge this model faces is the "curse of dimensionality," which occurs when the number of features (dimensions) in the dataset increases. As the dimensionality grows, the volume of the space increases exponentially, and data points become sparse. This sparsity makes it difficult for distance-based algorithms like KNN to differentiate between neighbors, as all points tend to appear almost equidistant from each other in high-dimensional space. Consequently, the distinction between close and far neighbors becomes less meaningful, which can reduce the effectiveness of KNN for classification tasks, including handwriting recognition where there may be many features (such as pixel intensities).

To mitigate this issue, Principal Component Analysis (PCA) is applied to reduce the number of dimensions while retaining most of the data's variance. PCA transforms the original features into a smaller set of orthogonal components, capturing the most important variations in the data. By reducing the number of dimensions, PCA helps to combat the curse of dimensionality, improving the KNN model's ability to effectively distinguish between neighbors and maintain high classification accuracy.

**Pre-Processing**

* **Zero Variance Filter (zv):** Removed 76 features with zero variance to reduce dimensionality and noise.
* **Centering:** Adjusted features to have a mean of zero, which can improve the distance-based calculations in KNN.
* **Principal Component Analysis:**
  + Principal Component Analysis (PCA) is a dimensionality reduction technique used to reduce the number of features in a dataset while retaining as much variance as possible. In this KNN model, PCA was applied as part of the preprocessing to handle the high-dimensional data effectively. Initially, the dataset contained 785 features, representing pixel intensities of handwritten digits. However, **76 features were identified as having zero variance** (constant values across all observations) and were dropped at the start. These features provided no useful information for distinguishing between different classes and could potentially add noise or computational overhead.

After removing the zero variance features, PCA was applied to the remaining features to reduce dimensionality. To retain **95% of the variance** in the data, PCA selected **154 principal components** out of the original set of features. This step was crucial to improve the computational efficiency of the KNN model and potentially enhance its accuracy by focusing on the most informative features.

**Tuning and Training**

The tuning of the KNN model involved selecting the optimal number of neighbors (k). A tuneLength of 2 was specified, meaning that two different values for k were tested (5 and 7). The plot shows that the model's accuracy decreased slightly as k increased from 5 to 7, indicating that fewer neighbors may lead to better performance in this context.

The model was validated using 3-fold cross-validation, ensuring that the data was split into three subsets, each serving as a validation set once while the remaining two subsets formed the training set. This process provides a robust estimate of the model's generalization performance.

A graph with a line

Description automatically generated

**Results**

The final KNN model achieved an overall accuracy of **94.04%**, as indicated by the confusion matrix and overall statistics:

* **Sensitivity and Specificity**:
  + The model showed high sensitivity across all digits, ranging from **92.37% to 99.62%**, suggesting it correctly identified most true positives. However, digits **8** and **9** were more challenging for the model, showing slightly lower sensitivity.
  + The specificity was also high across all digits, ranging from **99.10% to 99.94%**, reflecting the model's strong ability to correctly identify true negatives and avoid false positives.
* **Balanced Accuracy**:
  + The balanced accuracy, which considers both sensitivity and specificity, was consistently high across all digits. However, digits **8** and **9** were the most problematic, causing some classification errors, as indicated by their relatively lower balanced accuracy.
* **Positive Predictive Value (Precision)**: The model maintained high precision across most digits, suggesting that when the model predicted a digit, it was usually correct.

A screenshot of a computer

Description automatically generated

Overall, the KNN model provided strong performance in the handwriting recognition task, with its preprocessing steps and careful tuning of the number of neighbors contributing to its effectiveness. While it slightly underperformed compared to the Random Forest model, KNN still offers a valuable approach, especially when leveraging dimensionality reduction techniques like PCA to handle high-dimensional data efficiently. The decrease in accuracy with increasing k suggests that the model benefits from fewer neighbors, likely due to the distinctive features of different handwritten digits being more easily distinguishable in smaller neighborhoods.

**SVM Linear Model**

Support Vector Machines (SVM) are powerful supervised learning models used for classification and regression tasks. An SVM works by finding the optimal hyperplane that maximally separates data points of different classes in a high-dimensional feature space. The model’s effectiveness depends on the choice of kernel, which transforms the input data into a higher-dimensional space. In this case, an SVM with a linear kernel was used. A linear kernel is particularly well-suited for high-dimensional data, such as handwriting recognition, because it assumes that classes are linearly separable. In high-dimensional spaces, data points are often more easily separable by a hyperplane, which makes a linear kernel competitive with other, more complex kernel tricks.

**Pre-Processing**

The same pre-processing steps used for the KNN model were applied to the dataset prior to running the SVM Linear Kernal model to save on the computational cost of running the model:

* **Zero Variance Filter (zv)**: Removed features with zero variance to reduce dimensionality and noise, as such features do not contribute to the model's decision boundary.
* **Centering**: Standardized the features by centering them to have a mean of zero, which is beneficial for SVMs because it ensures all features are on a similar scale.
* **Principal Component Analysis (PCA)**: Applied to further reduce dimensionality while retaining most of the variance in the data, improving computational efficiency and potentially enhancing accuracy.

**Tuning and Training**

The SVM model was tuned using a grid search over different values of the cost parameter C (0.1, 1, and 10). The cost parameter controls the trade-off between maximizing the margin and minimizing classification error. A lower value of C creates a wider margin but allows some misclassifications, while a higher value of C attempts to classify all training examples correctly, which may lead to a narrower margin and potentially overfitting.

The performance plot shows that the accuracy of the model was highest when C was set to 0.1, and it gradually decreased as C increased to 10. This suggests that a simpler model (with a wider margin) performed better in this context, likely due to better generalization to unseen data.

**A graph with a line

Description automatically generated**

**Results**

The final SVM model with a linear kernel achieved an overall accuracy of **92.45%** on the test dataset, as demonstrated by the confusion matrix and accompanying statistics:

* **Sensitivity (Recall)**: The model achieved consistently high sensitivity across all digit classes, ranging from **92.21%** to **99.63%**. This high sensitivity indicates that the model successfully identified most of the actual positive instances for each digit. Digits '0' and '6' had particularly high sensitivity values, reflecting the model's strength in accurately identifying these classes with minimal false negatives.
* **Specificity**: The specificity, which measures the model's ability to correctly identify true negatives and avoid false positives, ranged from **99.04%** to **99.84%** across all digit classes. This high specificity demonstrates the model's robustness in distinguishing non-target digits and minimizing false positive rates.
* **Balanced Accuracy**: The balanced accuracy, an average of sensitivity and specificity, was high across all digit classes, ensuring robust performance even in cases of class imbalance. While most digits showed strong balanced accuracy, digits '2' and '9' had slightly lower scores, suggesting that these classes were more challenging for the model to classify correctly. This indicates a potential area for further refinement or additional feature engineering.
* **Precision (Positive Predictive Value)**: The model exhibited high precision across most digit classes, indicating that when the model predicted a specific digit, it was generally correct. This high precision minimizes the number of false positives, enhancing the reliability of the model's predictions, especially for digits that the model could classify with high confidence.

**A screenshot of a computer

Description automatically generated**

Using a linear kernel in SVM provides a significant computational efficiency advantage over other kernel methods, such as Radial Basis Function (RBF) or polynomial kernels, particularly in high-dimensional spaces like digit classification tasks. The linear SVM model achieved an overall accuracy of 92.45% on the test dataset, demonstrating strong performance with high sensitivity (92.21% to 99.63%) and specificity (99.04% to 99.84%) across all digit classes. Although the linear SVM's accuracy was slightly lower than the Random Forest model's 92.98% and comparable to the K-Nearest Neighbors (KNN) model's 92.80%, its computational efficiency makes it a compelling choice for applications where speed and resource management are critical.

The linear SVM strikes a good balance between simplicity, interpretability, and computational efficiency while delivering competitive classification performance. Its lower computational cost compared to Random Forest and non-linear SVM models makes it particularly suitable for high-dimensional datasets, such as handwriting recognition, where quick training and prediction times are crucial. Despite a small trade-off in accuracy compared to more complex models, the linear SVM is an attractive option for real-world applications where computational resources are a priority.

**SVM Polynomial**

The polynomial kernel allows the SVM to create non-linear decision boundaries, which can capture more complex relationships in the data compared to a linear kernel. However, the choice of kernel introduces additional hyperparameters, such as the degree of the polynomial, that need to be carefully tuned to balance model complexity and performance.

**Pre-Processing**

Similar to the KNN and SVM linear models, this model underwent preprocessing steps to optimize its performance:

* **Zero Variance Filter (zv)**: Removed features with zero variance to reduce dimensionality and noise.
* **Centering**: Standardized the features to have a mean of zero, improving the effectiveness of the SVM.
* **Principal Component Analysis (PCA)**: Applied to reduce dimensionality while retaining most of the variance in the data, improving computational efficiency and potentially enhancing accuracy.

The model was validated using 2-fold cross-validation, ensuring a robust evaluation of its generalization capability.

**Tuning and Training**

The tuning of the SVM model involved a grid search over different values of the cost parameter (C), polynomial degree (degree), and a fixed scale value calculated as 1 / number of features. The tuning grid tested combinations of C = {0.1, 1, 10}, degree = {2, 3, 4}, and scale was set to approximately 0.00127551.

The plot indicates that the model's accuracy improved with higher polynomial degrees and increased values of C. The best performance was achieved with the parameters degree = 4, scale = 0.00127551, and C = 10. This combination allowed the model to capture more complex relationships in the data, resulting in a more accurate decision boundary.

**A graph of a graph with different colored lines

Description automatically generated**

**Results**

The final SVM model with a polynomial kernel achieved an overall accuracy of 94.61%, as indicated by the confusion matrix and overall statistics:

* **Sensitivity and Specificity**:
  + The model demonstrated high sensitivity (true positive rate) across all digits, ranging from 92.21% to 99.63%, similar to the linear kernel SVM.
  + Specificity was also high across all digits, ranging from 99.04% to 99.84%, indicating the model's strong ability to correctly identify true negatives and avoid false positives.
* **Balanced Accuracy:**
  + The balanced accuracy, which considers both sensitivity and specificity, was consistently high across all digits. However, digits 2 and 9 were slightly more challenging for the model, as indicated by their relatively lower balanced accuracy.
* **Positive Predictive Value (Precision):** The model maintained high precision across most digits, suggesting that when the model predicted a digit, it was usually correct.

**A screenshot of a computer

Description automatically generated**

While the SVM model with a polynomial kernel provided competitive accuracy, achieving similar results to the linear kernel SVM and slightly better than KNN, the computational cost was significantly higher. Polynomial kernels are computationally more intensive than linear kernels due to the additional complexity introduced by higher-degree polynomials. Given the relatively modest improvement in performance over the simpler linear SVM and the much higher computational requirements, this model is not recommended for practical use in this context. The linear SVM or Random Forest models provide a better trade-off between accuracy and computational efficiency.

**Algorithm Comparison**

**Accuracy Comparison**

When comparing the overall accuracy of the algorithms, the Random Forest model outperformed the SVM (linear and polynomial kernels), K-Nearest Neighbors (KNN), Decision Tree, and Naive Bayes models. The Random Forest achieved an accuracy of **96.45%**, demonstrating its superior ability to handle high-dimensional feature spaces and capture complex patterns within the dataset.

The SVM with a linear kernel achieved an accuracy of **92.45%**, while the SVM with a polynomial kernel performed better, achieving an accuracy of **94.61%**. The polynomial SVM slightly outperformed the KNN model, which had an accuracy of **94.04%**, indicating that both models are effective, but the polynomial SVM offers a marginal improvement in accuracy. The Decision Tree model achieved an accuracy of **85.06%**, which was significantly better than both the Gaussian Naive Bayes and Kernel Naive Bayes models, with accuracies of **53%** and **39.48%**, respectively.

The variations in accuracy across these models can be attributed to their underlying assumptions and their ability to manage the dataset's characteristics:

* **Random Forest** is a non-parametric model that uses an ensemble of decision trees to capture complex patterns, making it highly effective for this dataset.
* **SVM (Linear and Polynomial)** models performed well in this high-dimensional space. The linear kernel provided a computationally efficient solution for linearly separable data, while the polynomial kernel, capable of modeling more complex relationships, achieved slightly better accuracy but with increased computational demands.
* **KNN** achieved competitive performance with an accuracy of **94.04%**, closely matching the polynomial SVM's accuracy. However, the slight difference suggests that the polynomial SVM may be better suited for capturing complex patterns in this particular dataset.
* **Decision Tree** outperformed the Naive Bayes models but was less effective than the ensemble and distance-based methods due to its sensitivity to noise and tendency to overfit.
* **Naive Bayes (Gaussian and Kernel)** models assumed feature independence and specific distributions, which did not align well with the dataset's characteristics, resulting in lower performance.

**Speed Comparison**

In terms of computational efficiency, the Naive Bayes models (both Gaussian and Kernel) were the fastest to train and evaluate due to their simple probabilistic framework and linear time complexity with respect to the number of features and samples.

The KNN model also had relatively low computational requirements since it involves straightforward distance calculations during prediction, although this was still relatively computationally intensive given the dataset size and dimensionality.

The SVM with a linear kernel was more computationally efficient than the SVM with a polynomial kernel, which required significantly more computation to determine the optimal decision boundaries due to the higher complexity of polynomial transformations. Despite this, the linear SVM's performance was comparable to the polynomial SVM, making the linear SVM a preferable choice for high-dimensional datasets where computational efficiency is a concern.

The Random Forest model, while more computationally intensive than the Naive Bayes and KNN models, provides a good balance between accuracy and training time. The Decision Tree model, although slower than Naive Bayes, was faster than Random Forest and SVM with a polynomial kernel but at the cost of lower accuracy compared to the Random Forest.

**Overall Model Comparison**

Given the accuracy and computational trade-offs, the **Random Forest** model emerges as the best performer, providing high accuracy with manageable computational requirements. The **SVM with a linear kernel** offers a good balance between accuracy and efficiency, especially in high-dimensional spaces. In contrast, the **SVM with a polynomial kernel** did not provide sufficient accuracy gains to justify its computational expense. The **KNN** model, while effective, is slightly less accurate and efficient in high-dimensional spaces. The **Decision Tree** offers moderate accuracy with reasonable computational costs, and the **Naive Bayes models** are the least accurate but the fastest, making them suitable only for applications where training speed is more critical than accuracy.

**Conclusion**

The results of this project show both the potential and the limitations of the evaluated methods for recognizing handwritten digits. While some models performed well, achieving around 96% accuracy, this means that for every 100 digits, about 4 could be incorrectly identified. In real-world settings, such as sorting mail or processing handwritten forms, even a few mistakes can add up to big problems. Errors could lead to delays, higher costs, and frustration for customers.

Models that use more straightforward approaches, like decision trees or basic statistical methods, struggled more with accuracy and made more errors. This indicates that these simpler models may not be reliable enough for tasks where precision is crucial.

Overall, while the current models offer a good starting point, they are still likely not accurate enough for applications where mistakes are costly. To achieve the level of accuracy needed for real-world use, more advanced and precise methods should be considered. This would help ensure that tasks involving handwritten text are handled efficiently, accurately, and without causing disruptions.