CM3015 Mid-term Coursework

Machine Learning Algorithms for Handwritten Digit Classification

Abstract

In this project, the effectiveness of five classification-based machine learning algorithms: K-Nearest Neighbours, Decision Trees, Logistic Regression, Support Vector Machines, and Random Forest are evaluated with the aim of identifying the most effective algorithm for classifying handwritten digits. Through hyperparameter tuning and K-Fold cross validation, the algorithms are compared using accuracy as the primary metric while also considering precision, recall, and F1-score to understand each algorithm's strengths and weaknesses. The results reveal that the Support Vector Machine performed best with an accuracy of 98.06% both before and after hyperparameter tuning, while the Decision Tree algorithm showed subpar performance with an accuracy of 88.89%.

Table of Contents

1. Introduction

Handwriting recognition is an essential task within the domain of machine learning which revolves around the ability of machines to interpret handwritten characters. Handwriting recognition benefits numerous industries by streamlining processes such as automating data entry, digitising documents, verifying signatures, processing checks, and even deciphering handwritten medical prescriptions. However, efficiently and accurately recognizing handwriting poses a significant challenge given how handwriting style differs from person to person.

1.1 Literature Review

Challenges of handwritten character recognition

The challenges associated with handwritten character recognition are highlighted in a study [1] that while significant advancements have been made in the field of Optical Character Recognition (OCR) especially in the ability to accurately recognise machine-printed characters, handwritten character recognition on the other hand remains a challenging problem primarily due to the variability in each individual's handwriting style. As a result, handwriting recognition continues to be an active area of research.

A study comparing different handwriting recognition techniques [2] highlights two main steps involved in recognising handwritten characters. Firstly, extracting handwritten characters from a scanned image of handwritten characters. Secondly, identifying individual extracted characters. It also highlights a flaw of two characters that are closely placed together mistakenly being extracted as one single character. This further highlights how challenging the task of recognising handwriting is.

Machine learning algorithms for character recognition

The study [2] further discusses the pros and cons of two classification algorithms - support vector machine (SVM) and k-nearest neighbour (KNN). When used to classify the MNIST handwritten digits dataset, it was found that the KNN algorithm was simple to implement and only required a small training sample. However, it consumes higher physical memory resources which is a limitation for larger datasets. SVM was found to work well even on unstructured data, and was able to solve complex classification problems. However, it is challenging to implement and set up, especially in determining the most suitable kernel function.

SUMMARY

In this literature review, the challenges of handwritten character recognition are highlighted, along with discussion of numerous machine learning algorithms that are commonly used to recognise handwriting. Most importantly, it highlights how every single algorithm has its own strengths and weaknesses. This motivated my project idea of comparing different classification-based machine learning algorithms to find their strengths and weaknesses, and to identify the most optimal one for my chosen dataset of handwritten digits.

1.2 Project Aim

This project aims to evaluate five classification-based machine learning algorithms: K-Nearest Neighbours, Decision Trees, Logistic Regression, Support Vector Machines, and Random Forest, with the goal of determining the most efficient algorithm for handwritten digit classification by evaluating their performance using accuracy as the key metric, while also evaluating their precision, recall, and F1-score to understand the strength and weakness of each algorithm.

The findings of this project will not only provide valuable insights for automating industrial processes that are reliant on handwriting recognition, but also enhance the reliability of existing ones.

1.3 Dataset - Handwritten Digits

The Digits dataset [3] from scikit-learn library is used in this project. This dataset is a copy of the test set of the “Optical Recognition of Handwritten Digits” dataset from UCI ML Repository. I chose to use the scikit-learn library’s version as it contains data that has already been standardised, which aligns with my project aim of comparing between different algorithms as it allows me to focus on analysing the algorithms without having to implement additional pre-processing steps on the data.

The digits dataset contains 1797 samples, each corresponding to an image of a handwritten digit ranging from 0 to 9, along with 1797 labels that match the samples. The images are represented as pixel values in a 8x8 grid, with each image having 64 features.

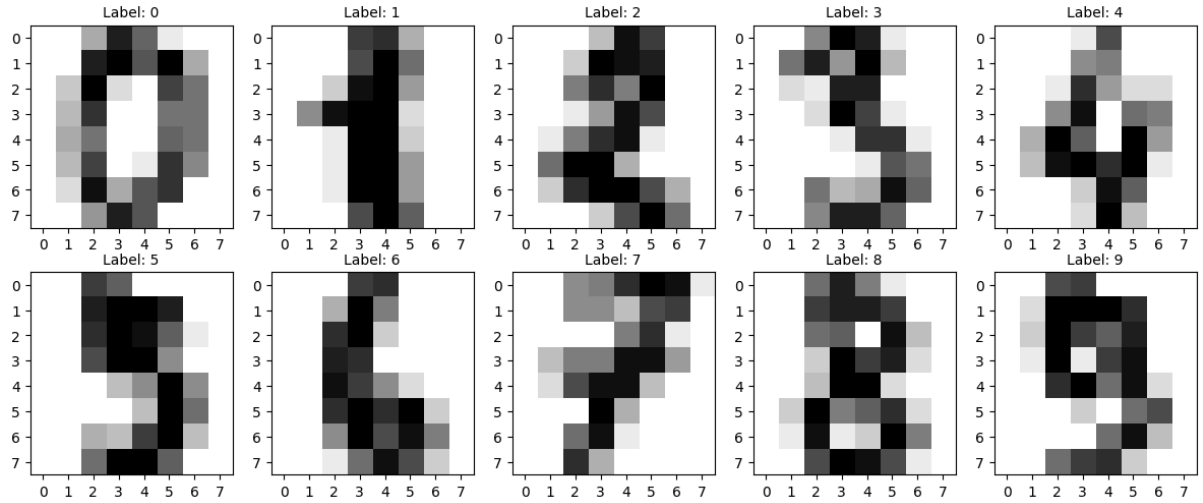


Figure 1. Plot of first 10 samples

2. Background

In this project, five machine learning algorithms were evaluated: K-Nearest Neighbours, Decision Trees, Logistic Regression, and Support vector machines. The K-Nearest Neighbours algorithm was implemented from scratch, while the others are implemented from the scikit-learn library.

2.1 K-Nearest Neighbours

KNN is an algorithm used for classification and regression tasks. It is distance-based, which means that prediction is made by assuming that the smaller the distance between two points, the more similar they are.

How this algorithm works in the context of the digits dataset is:

1. For each handwritten digit image, the similarity between that image's feature vector and those of the training data is calculated using a distance metric.
2. It then identifies the K training digits that are most similar (the nearest distance) to the new digit.
3. Finally, it assigns the class label that is most frequent among these K nearest neighbours to the new digit, effectively classifying it.

I. Distance metrics

For the KNN algorithm to perform classification well, an appropriate distance metric has to be determined because each metric has a different way of calculating how close two points are. This means that the similarity of the same two points can be different when calculated with different metrics, resulting in different prediction outcomes.

The most used metrics are Euclidean distance, Manhattan distance and Cosine distance, all of which I have included in my own implementation of the KNN algorithm.

Euclidean distance
Measures the true straight line distance between two points in Euclidean space. It works well especially in cases where features have the same units and are similarly scaled.

Manhattan distance
Measured by the sum of absolute differences of their cartesian coordinates. It works well especially when the dataset has many features (high dimensionality).

Cosine distance
Measured by the cosine of the angle between two vectors. It is often used to measure document similarity in text analysis.

II. My KNN Implementation (from scratch)

The KNN algorithm was implemented from scratch as a Python class (called myKNN) with three main methods: fit, compute_distance, and predict.

(See Appendix A for the code implementation.)

Methods

The “fit” method within myKNN class takes in X_train (features of training data) and y_train (corresponding labels of training data) as params, and stores them for use during prediction. The “compute_distance” method computes the distance using either Euclidean, Manhattan, or Cosine distance. Lastly, the “predict” method makes predictions on new, unseen data (X_test). It computes the distances of each test instance to all training instances, identifies the k nearest neighbours, and predicts the label based on the most frequent label among these neighbours.

Verification

To verify my implementation, I compared the results of my algorithm with sklearn’s KNeighborsClassifier. Using the same k value and distance metric, both produced the same predictions and accuracy, which lets me confirm that my implementation of the KNN algorithm is working as intended.

Figure 2. Verification of myKNN implementation

2.2 Decision Tree

A decision tree is an algorithm used for classification and regression tasks. The structure of a decision tree is similar to a family tree where there is a hierarchical layout which starts from the root at the very top, then branches out into multiple paths down the tree. Prediction is made by first starting at the root of the tree where a question relating to certain characteristics of the data is asked. Depending on the answer, the next path down the tree is determined. This repeats till it reaches the end of the tree (the leaf node), and that would be the final prediction.

In the context of the digits dataset, the entire set of test data will start at the root of the tree where a question such as “Does the handwritten image contain one straight line?” is asked, and the data is split accordingly. This continues till the leaf node which would then be the predicted class for each of the data.

2.3 Logistic Regression

Logistic Regression is an algorithm designed for binary classification tasks, and is often used to set a baseline as it is simple to implement. In this algorithm, prediction is made by calculating how likely each data point belongs to a particular class. This probability is calculated using the sigmoid function which takes in feature values of the data, then converts it to a value between 0 and 1 representing the probability. If the probability value is below 0.5, the data point would be labelled as false (if assuming 0 is false). If the probability value is above 0.5, it would be labelled as true.

For multi-class classification tasks like the digits dataset used in this project, an additional method called One-vs-Rest is applied which involves splitting the dataset based on the number of classes. The Logistic Regression algorithm described above is then used on each of the sets to predict the probability. Then, the set with the highest probability value is chosen as the final prediction. Figure 3 below shows an illustration I have created to visualise the process.

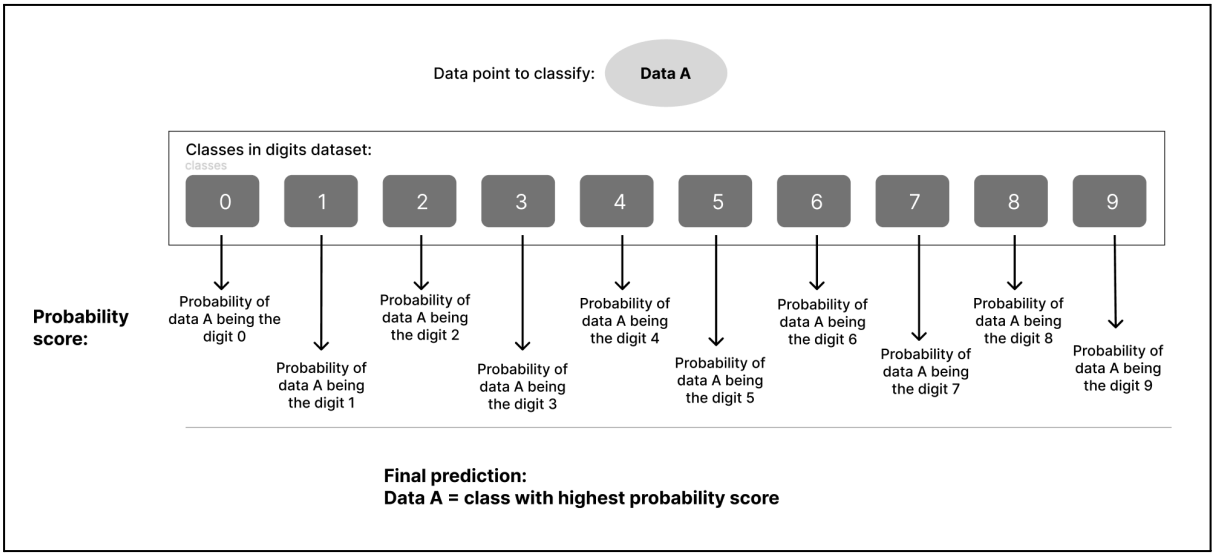


Figure 3. One-vs-Rest on digits dataset

2.4 Support Vector Machine

Support Vector Machine is an algorithm used mostly for classification tasks where it finds the best way to separate different classes so that there is a clear boundary of which area belongs to which class. Using this boundary, the algorithm uses each new data point's features to determine which point in the boundary it belongs to. Based on which class's area it falls on, the final prediction is made.

In terms of the digits data, during the training process the algorithm first draws clear boundaries between each handwritten digit image based on the pattern of the pixels. When used to predict on test data, the algorithm will determine which area each data point belongs to based on its features (pixel values) by matching it to what was learned during training, and prediction is made.

2.5 Random Forest

Random Forest is an algorithm used for classification tasks, and is made up of a collection of decision trees which it uses in the decision making process to make predictions. It starts by first creating multiple decision trees through random selection of features and data points. Afterwards, each tree will make predictions of the data based on its features. Finally, the most "popular" prediction among all the predictions generated will be chosen as the final prediction.

In the context of the digits dataset, during training the Random Forest algorithm will first create multiple decision trees where each of the trees are made up of a random subset of the data and the pixel values, which they use to learn how to classify into the digits 0 to 9. During testing, each tree will try to predict which digit the new data point belongs to. Finally, the most common prediction among all the trees will be selected as the final prediction by the Random Forest algorithm.

3. Methodology

This section discusses the steps and methods adopted, following the same sequence as my Jupyter notebook implementation.

3.1 Dataset Exploration

The digits dataset was first loaded from the sklearn library, followed by an exploratory data analysis to better understand the data that I am working with. A plot of the distribution across classes was then created to determine whether they are balanced. As seen in Figure 4, all classes are relatively well distributed. This also justifies the use of Accuracy as the key metric to measure performance.

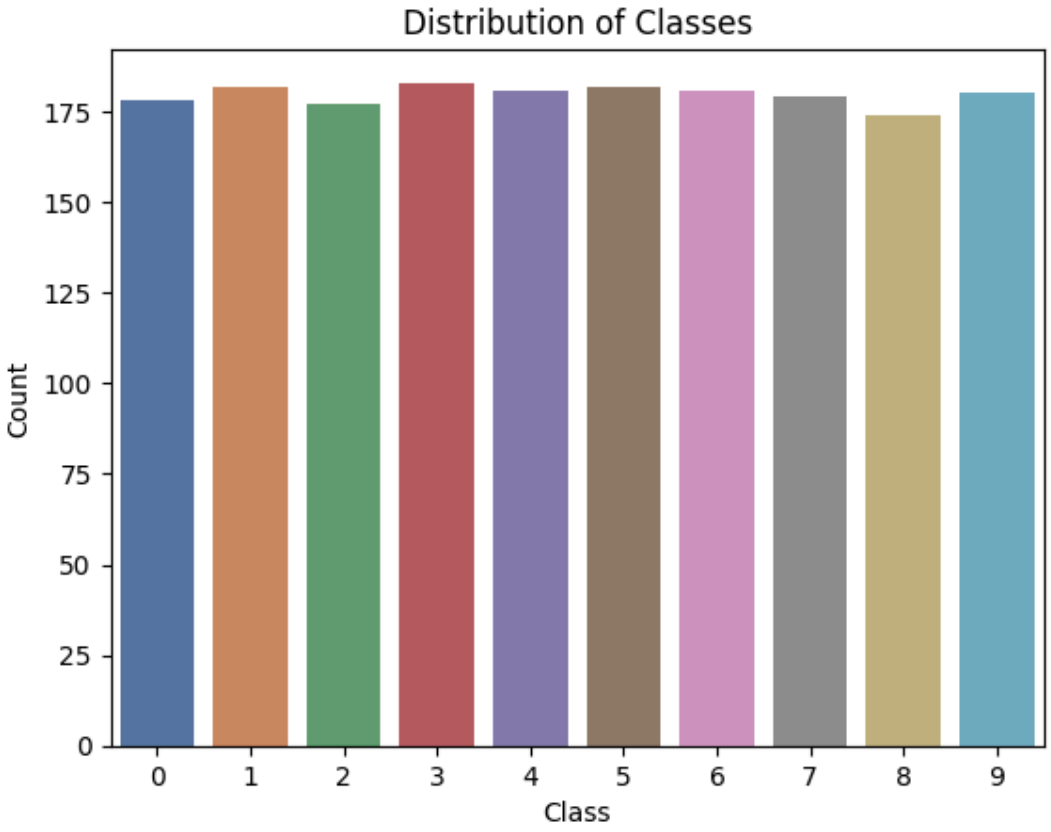


Figure 4. Class distribution plot

3.2 Data Pre-processing

In this step, the dataset was first split into training and testing sets in the ratio of 80% training and 20% testing. This process is required as it allows me to train the classifiers on the training data, then evaluate it on the unseen test data.

Next, feature scaling was carried out to normalise the feature values. This is needed as some algorithms like KNN are sensitive to unscaled data, which would lead to unfair comparison when compared with the performance of other algorithms like Decision tree that are not impacted by unscaled data.

3.3 Algorithm selection

In this project, the machine learning algorithms involved are: K-Nearest Neighbours, Decision Tree, Logistic regression, Support Vector Machine, and Random Forest.

These algorithms were chosen as they are all classification based, and are also often used for handwritten character recognition tasks, which is evident from my literature review.

The K-Nearest Neighbours algorithm is implemented from scratch, while the others are from the sklearn library.

3.4 Hyperparameter tuning

Hyperparameter tuning was carried out to find the most optimal hyperparameter configurations for each of the algorithms, with K-fold cross validation used to evaluate performance of the different hyperparameter configurations. The choice to use K-fold cross validation technique is due to the digits dataset being relatively small with 1797 data points, which could easily lead to overfitting.

In the hyperparameter tuning process, different hyperparameter combinations of each algorithm are tested and evaluated using K-fold cross validation. During validation, the dataset is divided into “k” number of folds, and each hyperparameter combination is tested to identify the configuration that results in the highest accuracy in classifying the handwritten digits. A k value of 5 is used in my implementation.

This entire process ensures that each algorithm is given the best chance to perform well, which would make comparison more meaningful.

The following hyperparameters are tuned:

- **KNN**: k number of neighbours and distance metric.
- **Decision Tree**: criterion, maximum depth, minimum sample needed to split a node, and minimum sample needed to reach leaf node.
- **Logistic Regression**: regularisation strength.
- **SVM**: Regularisation strength, gamma and kernel.
- **Random Forest**: n number of trees, maximum depth, and minimum samples needed to split a node.

3.5 Evaluation

Once the most optimal hyperparameters are found, the models are retrained on the training data using those hyperparameters settings. Additionally, I also trained each of the models on the training data with default hyperparameters.

After that, both sets of trained models were used to predict on the test data. The accuracy, precision recall and F1-Score are then measured.

4. Results

4.1 Hyperparameters

Below are the most optimised hyperparameter combinations for each algorithm identified through hyperparameter tuning as highlighted in the methodology.

```
Best hyperparameters for KNN:
  >> best_k = 1
  >> best_metric = manhattan
  >> avg_accuracy = 0.9749540263259776

Best hyperparameters for Decision Tree:
  >> criterion = entropy
  >> max_depth = 15
  >> min_samples_leaf = 1
  >> min_samples_split = 2

Best hyperparameters for Logistic Regression:
  >> C = 0.2682695795279725

Best hyperparameters for SVM:
  >> C = 100
  >> gamma = 0.01
  >> kernel = rbf

Best hyperparameters for Random Forest:
  >> max_depth = None
  >> min_samples_split = 2
  >> n_estimators = 100
```

Figure 5. Optimised hyperparameter for each algorithm

4.2 Hyperparameters

Figure 6 below shows an overview of the accuracy, precision, recall and f1-score achieved by each algorithm. The top table represents results achieved by the default models, and the bottom table are results achieved by the optimised models (using hyperparameters identified during hyperparameter tuning).

Results of default algorithms:

	Accuracy	Precision	Recall	F1-Score
K-Nearest Neighbours	97.50%	97.65%	97.69%	97.65%
Decision Tree	84.17%	84.47%	83.60%	83.85%
Logistic Regression	97.22%	97.37%	97.44%	97.40%
Support Vector Machine	98.06%	98.30%	98.07%	98.17%
Random Forest	97.22%	97.40%	97.27%	97.32%

Results of optimised algorithms:

	Accuracy	Precision	Recall	F1-Score
K-Nearest Neighbours	97.78%	97.82%	97.98%	97.88%
Decision Tree	88.89%	89.16%	88.66%	88.84%
Logistic Regression	97.50%	97.68%	97.69%	97.67%
Support Vector Machine	98.06%	98.08%	98.15%	98.11%
Random Forest	97.22%	97.40%	97.27%	97.32%

Figure 6. Final results overview

Impact of Hyperparameter Tuning

The results overview in Figure 6 shows that the Decision Tree algorithm had the most improvement with an accuracy of 84% initially, to almost 89% after it was optimised through hyperparameter tuning. This is followed by K-Nearest Neighbours, Logistic Regression, and Support Vector Machine where there were very minor improvements of around 0.25% across all metrics. On the other hand, the Random Forest algorithm showed no difference at all across all metrics before and after it was optimised.

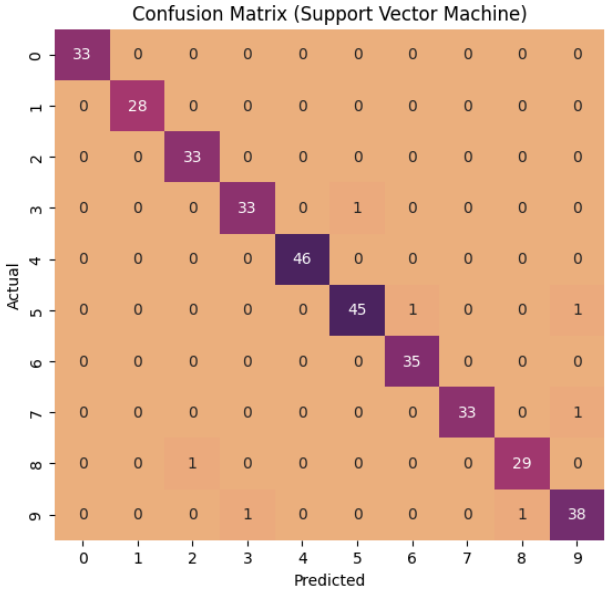
These findings highlight the positive impact that hyperparameter tuning can bring to certain algorithms, while some algorithms are not affected much by parameter changes in a handwritten digit classification task.

5. Evaluation

As stated in the aims, the algorithms will be evaluated using accuracy as the key metric, followed by the several other metrics to highlight the strength and weaknesses of each algorithm.

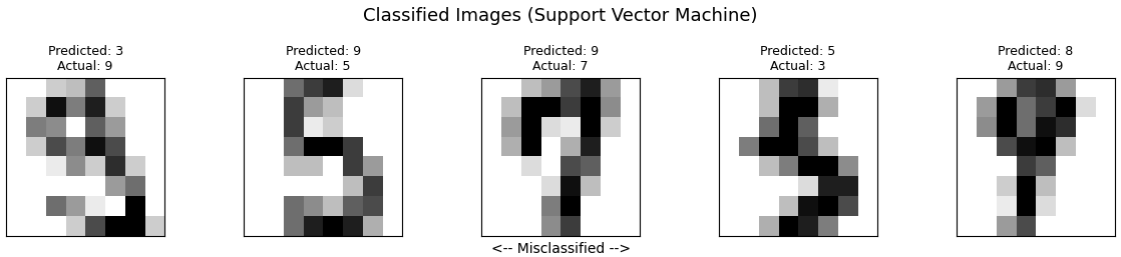
	Accuracy	Precision	Recall	F1-Score
Support Vector Machine	98.06%	98.08%	98.15%	98.11%
K-Nearest Neighbours	97.78%	97.82%	97.98%	97.88%
Logistic Regression	97.50%	97.68%	97.69%	97.67%
Random Forest	97.22%	97.40%	97.27%	97.32%
Decision Tree	88.89%	89.16%	88.66%	88.84%

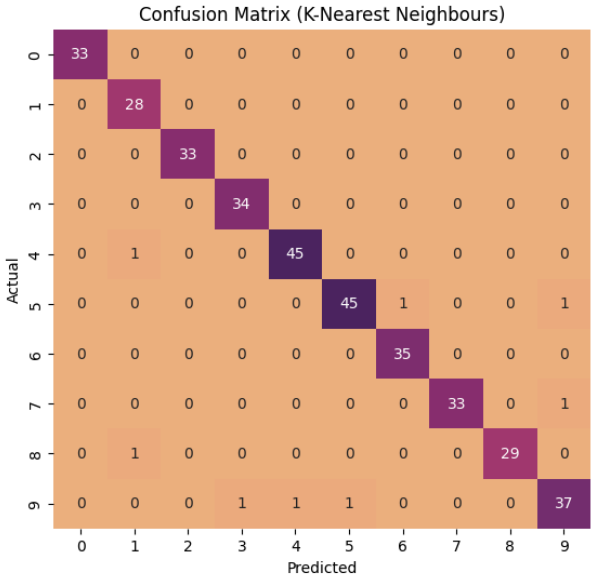
Figure 7. Performance of algorithms ranked by accuracy



1. Support Vector Machine

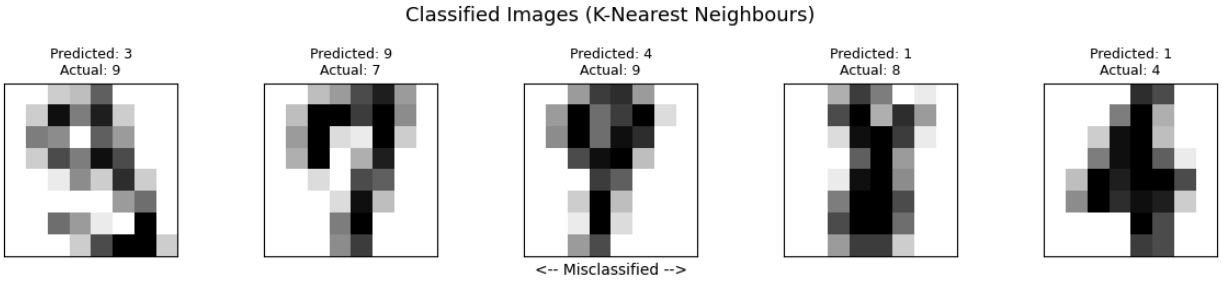
The Support Vector Machine algorithm ranked first in performance with an accuracy of 98.06% both before and after optimization, which shows its effectiveness in classifying handwritten digits. This can also be seen from the confusion matrix where most classes had all correct predictions. However, it can be observed that it struggles with predicting the digit “9” as shown in misclassified images below.

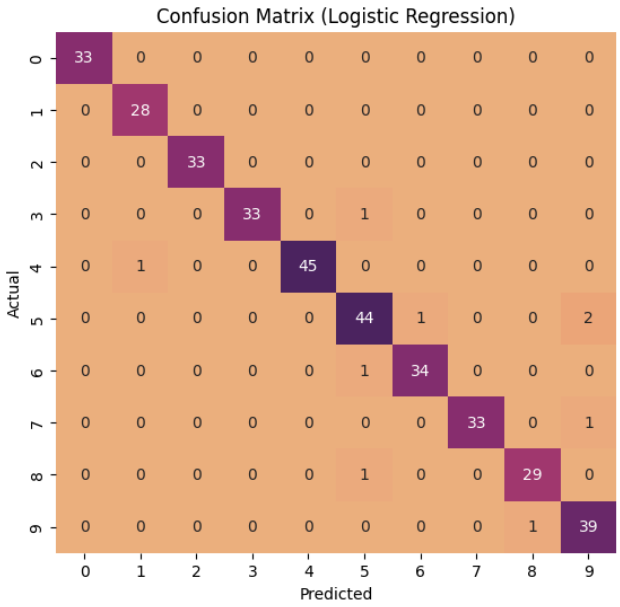




2. K-Nearest Neighbours

The KNN algorithm ranked second with an accuracy of 97.78% which suggests that it performs well in classifying handwritten digits. This can also be seen from its confusion matrix where most predictions were accurate. It can also be observed that the wrong predictions are spread out across different classes which tells me that it has a consistent performance.

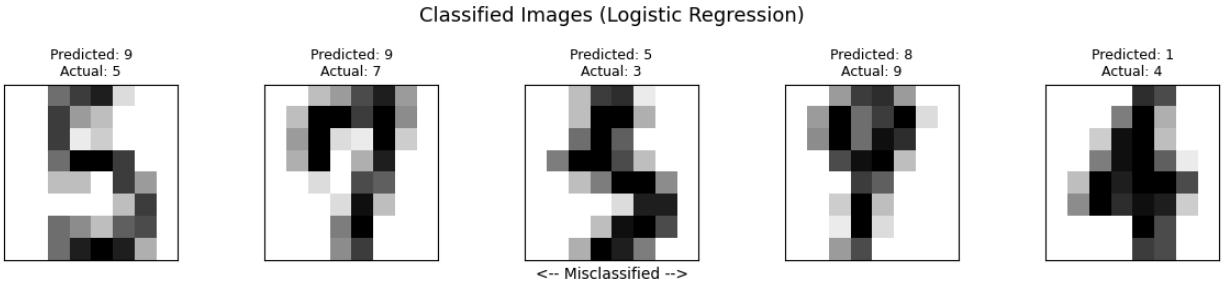


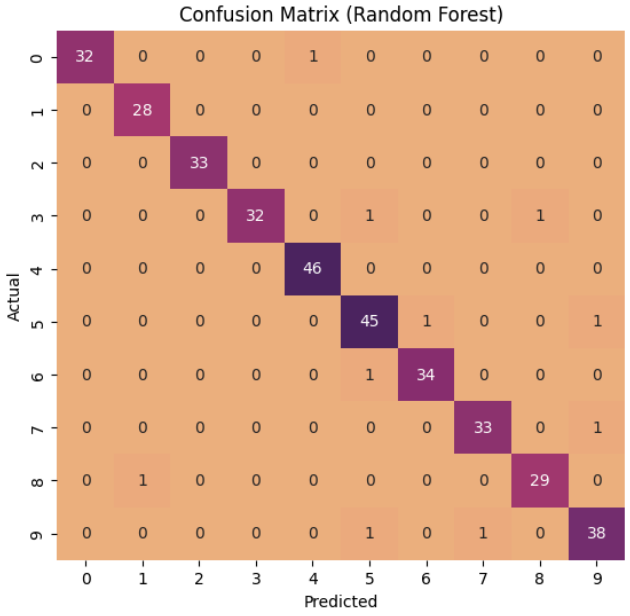


3. Logistic Regression

Logistic regression ranked third with an accuracy of 97.5%. From the confusion matrix, it can be seen that it predicts digits 0,1 and 2 well with no wrong predictions. On the other hand, it does not do well on the digit “5” where it made the most wrong predictions, and even predicted it as “9” on two occasions.

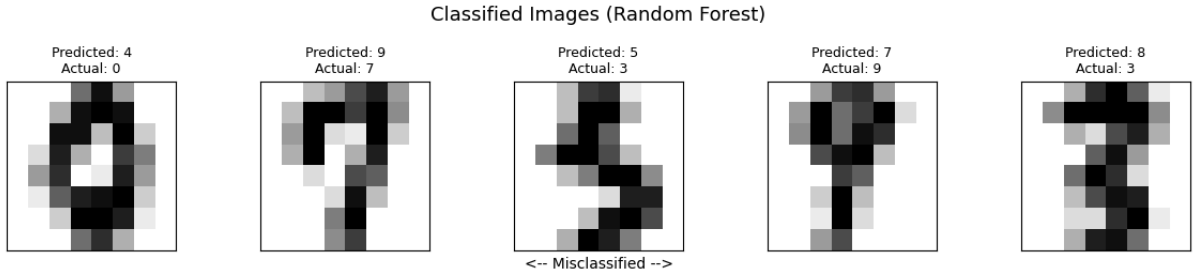
Interestingly, from the first classified image below, the digit 5 which was misclassified by the Logistic regression algorithm as digit 9 was also observed in the results of SVM where it predicted the same image of digit 5 as the digit 9 even though the digit 5 is clearly written. I also noticed that the middle image where the actual class is 3 has an image which resembles the digit “5” even to me.

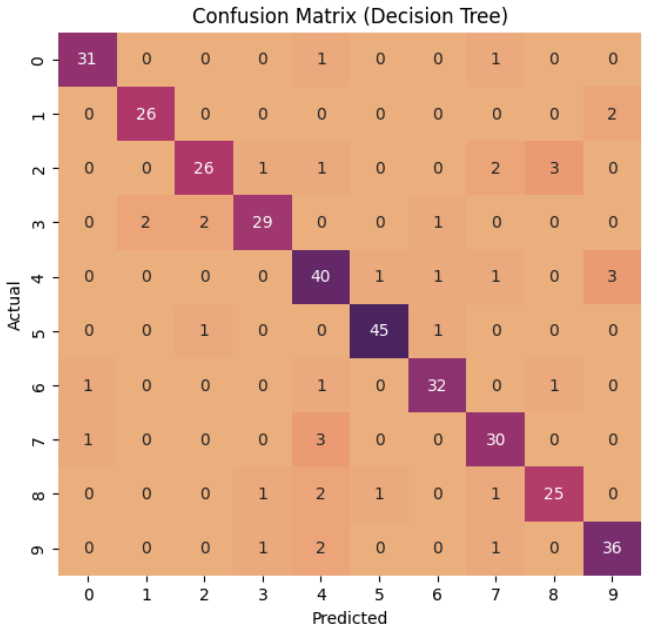




4. Random Forest

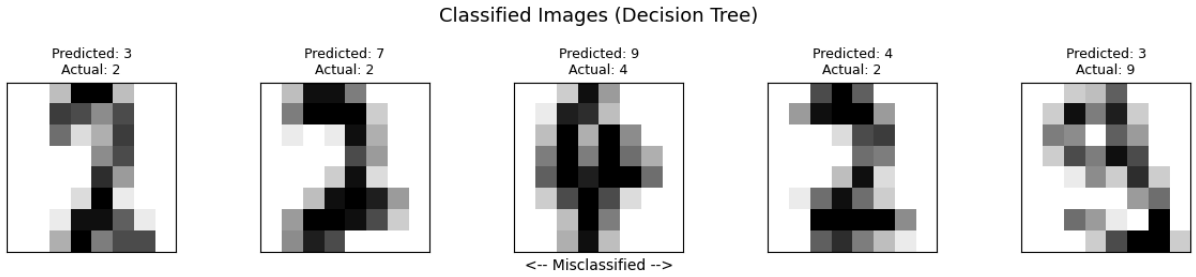
Random Forest ranked forth with an accuracy of 97.22%. It was able to predict digits 1, 2 and 4 perfectly, while it did not perform well in classifying digits 3, 5, and 9 with two instances of misclassifications each.





5. Decision Tree

Decision tree ranked last both before and after hyperparameter tuning with an accuracy of 88.89% achieved by the final optimised algorithm. This suggests that the Decision Tree algorithm is not suitable for classifying handwritten digits which is also seen in the confusion matrix where it made at least one wrong prediction for each class. From the classified images below, it can also be seen that the digit “2” was misclassified thrice, each time with a different predicted value even though the handwritten image does not seem to vary much.



6. Conclusion

In conclusion, this project's aim to evaluate five machine learning algorithms for the task of handwritten digit classification has been achieved successfully. The performance metrics of accuracy, precision, recall, and F1-score, as well as the confusion matrix have provided a comprehensive understanding of each algorithm's strengths and weaknesses.

The Support Vector Machine emerged as the top-performing algorithm with an accuracy of 98.06% and very similar precision, recall, and F1-scores. This indicates that the SVM algorithm offers a highly effective balance between correctly identifying true cases and limiting false classifications. This makes the SVM the most effective algorithm in classifying handwritten digits.

The K-Nearest Neighbours algorithm followed closely, demonstrating robustness with a 97.78% accuracy rate which suggests that it is almost as effective as SVM for the task of classifying handwritten digits. Logistic Regression and Random Forest also showed commendable performance, with accuracy scores above 97%, which signifies their potential utility in scenarios where interpretability and model complexity are significant factors.

However, the Decision Tree algorithm lagged behind the other algorithms with an accuracy of 88.89%, which may be due to its less sophisticated approach in handling the complex patterns in handwritten digits compared to the others’. While the Decision Tree algorithm had faster implementation and simpler interpretation, its lower performance in classifying handwritten digits highlights some trade-off between simplicity and efficacy.

All in all, this project's findings suggest that while all the evaluated algorithms achieved satisfactory performance in digit classification, Support Vector Machines stand out in terms of efficiency and effectiveness. These insights not only contribute to the field of handwritten character classification but also serve as a guide for choosing appropriate algorithms for similar image classification tasks.

7. References

[1] Vamvakas, G., Gatos, B., and Perantonis, S. J. 2010. Handwritten character recognition through two-stage foreground sub-sampling. Pattern Recognition 43, 8 (2010), 2807-2816. DOI: https://doi.org/10.1016/j.patcog.2010.02.018.

[2] P. C. Vashist, A. Pandey, and A. Tripathi. 2020. A Comparative Study of Handwriting Recognition Techniques. In *Proceedings of the 2020 International Conference on Computation, Automation and Knowledge Management (ICCAKM)*, Dubai, United Arab Emirates, 2020, 456-461. DOI: 10.1109/ICCAKM46823.2020.9051464.

[3] Scikit-learn developers. n.d. "Digits Classification Example — scikit-learn 0.24.2 documentation." Accessed [January 01, 2024]. https://scikit-learn.org/stable/auto_examples/datasets/plot_digits_last_image.html.

8. Appendices

8.1 Appendix A - KNN implementation

```python
class myKNN:
    """
    K-Nearest Neighbors algorithm for classification.
    """

    def __init__(self, k=5, dist_metric="euclidean"):
        """
        Initialize the KNN classifier with k neighbors and distance metric to use.
        By default, k is 5 and Euclidean distance is used.
        """
        self.k = k
        self.dist_metric = dist_metric

    def fit(self, X_train, y_train):
        """Fit the model using X_train as training data and y_train as target values."""
        if len(X_train) != len(y_train):
            raise ValueError("Length of X_train and y_train must be equal")
        self.X_train = X_train
        self.y_train = y_train
```

(implementation of the KNN algorithm)

```python
    def compute_distance(self, x1, x2):
        """Compute the distance between two data points using the distance metric specified."""
        if self.dist_metric == "euclidean":
            return np.sqrt(np.sum((x1 - x2) ** 2, axis=1))
        elif self.dist_metric == "manhattan":
            return np.sum(np.abs(x1 - x2), axis=1)
        elif self.dist_metric == "cosine":
            return np.dot(x1, x2) / (np.sqrt(np.dot(x1, x1)) * np.sqrt(np.dot(x2, x2)))
        else:
            raise ValueError(f"Unknown distance metric: {self.dist_metric}")

    def predict(self, X_test):
        """Predict the class labels for the provided data."""
        predictions = []
        for x in X_test:
            # compute distances using the specified distance metric
            dists = self.compute_distance(self.X_train, x)

            # get the k nearest neighbors & labels
            nearest_idx = np.argsort(dists)[: self.k]
            nearest_labels = [self.y_train[i] for i in nearest_idx]

            # use label with highest frequency as prediction
            predictions.append(np.bincount(nearest_labels).argmax())
        return predictions
```

(implementation of the KNN algorithm (2))