Handwritten Digit Recognition with MNIST

Martin Björkquist

EC Utbildning

Kunskapskontroll 2

202503

Abstract

This report looks at how to build a handwritten digit recognition system using the MNIST dataset, trying to make machine learning models work better in real-life situations. For this I used an SVM with data augmentation, such as rotations and shifts, and got a test accuracy of 0.9878, which was better than Logistic Regression at 0.9233 and Random Forest at 0.9713. I also made a Streamlit app to test the system by drawing digits, uploading images, or using camera feeds and snapshots. Data augmentation helped the SVM handle handwriting differences, but I found that real-world testing can still be tricky. In the future, I'd like to work on getting even better accuracy with real camera data where most progress can be done.

Acknowledgements

I'd like to thank a classmate in my study group for explaining about data augmentation. That really helped me realize that augmentation was the main focus I should work on, and it led to the biggest improvements in my results. That made me see this project from another perspective and I'm really grateful for that.

Abbreviations

Ensemble: A model combining multiple classifiers to improve prediction accuracy.

imgaug: A Python library for image augmentation, used for data augmentation in this project.

MNIST: Modified National Institute of Standards and Technology dataset, a collection of handwritten digits used for training and testing machine learning models.

OpenCV: Open Source Computer Vision Library, a Python library for image processing, used in the Streamlit application.

Random Forest: A machine learning model that uses multiple decision trees to make predictions.

Scikit-learn: A Python library for machine learning, used for model training and accessing the MNIST dataset.

Streamlit: A Python framework for building interactive web applications, used to create the digit prediction app.

SVM: Support Vector Machine, a machine learning model that finds the optimal hyperplane to separate data points.

Table of Contents

1 Introduction

Machine learning and computer vision have changed a lot about how we handle data over the past ten years, and are now something widely used in many areas like sorting mail or in banking, which is pretty cool having seen how things used to be. The MNIST dataset for this project is a standard set with 70,000 images of handwritten numbers that people often used to test machine learning models (LeCun et al., 1998). My project is about making these models better, especially for real-life situations where handwriting can be tricky and hard to deal with. I aimed to build a system that's accurate and actually works well when people use it in the real world, which I learned isn't as easy as it sounds (Géron, 2022, Chapter 1).

1.1 Objectives and Research Questions

My main goal with this project was to see how data augmentation changes the accuracy of machine learning models for recognizing handwritten digits. So the question to answer was: How does data augmentation affect the accuracy of a machine learning model? The study is kept to the MNIST dataset and I built a Streamlit app to test things out in a practical way with drawing digits and using a camera (Streamlit, n.d.).

1.2 Disposition

Here's how my report is set up: Chapter 2 is about the background of the models. Chapter 3 explains how I did the project. Chapter 4 shows my results. Chapter 5 sums it up with findings and what I learned.

2 Theory

2.1 Machine Learning and Image Recognition

Machine learning is about getting computers to learn from data patterns on their own, without being explicitly programmed (Géron, 2022, Chapter 1). For image recognition, this means training models to look at pictures and figure out what's in them - in this project, that's the handwritten digits. I used supervised learning, which means I gave the model labeled images, like "this is a 5" so it could learn from them and then predict digits in images never seen before. The challenging part is that images can look very different depending on the angle, handwriting style, or lighting, and that's where data augmentation helps. It makes new versions of the images, rotated or shifted a bit, so the model gets better at handling all kinds of variations (Géron, 2022, Chapter 14).

2.2 Evaluation Metrics

How do I know if my model is doing well? I look at metrics like accuracy, which tells me how often the model picks the right digit out of all its guesses. Accuracy works well for a dataset like MNIST because each digit shows up about the same number of times, so the results aren't skewed (Géron, 2022, Chapter 3). Accuracy wouldn't be the best metric if a dataset is more imbalanced.

$$Accuracy = \frac{Number\ of\ correct\ predictions}{Total\ number\ of\ predictions}$$

I also used a confusion matrix, which showed me where the model struggled and was mistaken with the predictions, like when a "3" is an "8". That was really helpful for seeing what digits were hard for the model and very useful for real-life testing later on.

These tools helped me figure out how good my models were at recognizing digits, and I read about them and image tasks in (Géron, 2022). *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems (3rd ed.). O'Reilly Media*

2.3 Feature Selection

Each MNIST image is a 28x28 pixel grid, so that's 784 pixels (features) for the model to work with and learn patterns from. Pixels in the middle of the image are usually the ones that matter most because they hold the main part of the digit, while the ones on the edges might be empty space or noise (Géron, 2022, Chapter 14). I thought about using a method like PCA to cut down the number of features to work with while still keeping the important variance, which (Géron, 2022, Chapter 9) talks about in his book, but I decided to stick with the raw pixel values instead. I wanted to keep things simple and quick, and I didn't want to risk losing any small details by changing the data. In digit recognition, the details can make a big difference, and since MNIST doesn't have a lot of noise, I thought by keeping them it would justify the additional training cost for my computer.

2.4 Models

I tried out five models in my project: Logistic Regression, Support Vector Machine (SVM), Random Forest, an Ensemble model, and an Augmented SVM. Each one has its own pros and cons, but I picked them because they seemed like a good fit for working with images.

- **Logistic Regression:** I started with this as my baseline since it's a simple model that guesses the chances of a digit being a certain number (Géron, 2022, Chapter 4). It was pretty quick to use for a first test, but it had a hard time with more complicated patterns.

- **SVM:** I chose the SVM because it's really good at splitting digits apart by drawing a line, or a hyperplane, between them (Géron, 2022, Chapter 5). It works well for datasets like MNIST that have a lot of pixel data..

- **Random Forest:** This model puts together multiple decision trees and picks the best prediction by voting. It reduces the risk of overfitting by averaging mistakes out, and it can handle noisy image data well (Géron, 2022, Chapter 7).

- **Ensemble Model:** In this model I've mixed the SVM and Random Forest to see if working together would balance their weaknesses better (Géron, 2022, Chapter 7).

- **Augmented SVM:** This is really just the SVM but with data augmentation added, which makes new versions of the images (by rotating, stretching or shifting them) to help it deal with different handwriting styles in the real world (Géron, 2022, Chapter14).

Pictures to show a diagram of the SVM's hyperplane and an example of an augmented digit are put in the Appendix as Figure A.5.1 and Figure A.5.2.

3 Method

3.1 Dataset

I used the MNIST dataset, which has 70,000 images of handwritten digits from 0 to 9 (LeCun et al., 1998) and got the data using Scikit-learn's fetch_openml function (Scikit-learn, n.d.). Then I split the data into different sets so I could train, validate, and test my models. I took 60,000 images for training and validation together and kept 10,000 separate for the test set. After that, I split the 60,000 into 50,000 for training and 10,000 for validation, so I could evaluate my models before the final test and later retrain the best one with more data.

- 70,000 MNIST set → 60,000 train_val, 10,000 test

- 60,000 train_val → 50,000 train, 10,000 val

I also normalized the images by dividing their pixel values by 255, scaling them from a range of 0 to 255 to a smaller range of 0 to 1 (Géron, 2022, Chapter 3). This makes the data easier for the models to work with, and it's quicker than using something like StandardScaler and keeps all the sets consistent without extra processing.

3.2 Exploratory Data Analysis (EDA)

To get to know the MNIST dataset better, I looked at its structure and characteristics. Each image is a 28x28 grid of grayscale pixels, and I used Matplotlib to visualize 10 random samples to understand their variations. This helped me notice that some digits, like "3" and "8" can look similar, which might be hard for the model to tell apart and made me understand why I'd need augmentation later. I checked the number of images for each digit (0–9) to confirm that no digit was overrepresented (LeCun et al., 1998). I also looked at pixel variance across the dataset to see which ones varied the most, and noticed that the digit's main part is usually in the center, just like expected.

Pictures to show a sample digit and a mapping of pixel changes are put in the Appendix as Figure A.5.3 and Figure A.5.4.

3.3 Feature Selection

After doing the EDA, I went with the raw pixel values for my models, where each 28x28 image gives me 784 features to work with. I didn't use methods like PCA because the dataset's size was okay for my computer, and I wanted to keep all the details for the models to learn from and hoped augmentation would take care of the differences in handwriting instead of changing the data more.

3.4 Models

I worked with five models for my digit recognition project: Logistic Regression, Support Vector Machine (SVM), Random Forest, an Ensemble model, and an Augmented SVM. I built them all using Scikit-learn to keep things the same, and I tried to keep the tuning simple. I trained each model, except the Augmented SVM, on the normalized MNIST training set (50,000 images) from 3.1, and evaluated them on the validation set (10,000 images). For the Augmented SVM, which I built using the non-augmented SVM, I used the full train-validation set (60,000 images) for retraining, since the SVM (trained and validated) was already my best model by then.

Logistic Regression: I started with this as a simple model to compare the others to. I set up a parameter grid for tuning, testing the regularization strength C with values [0.01, 0.05, 0.2, 0.5]. At first, I tried a bigger range from 0.001 to 10, but after some tests, I picked these values because they worked better. I used GridSearchCV for tuning, which I'll talk about in 3.5.

SVM: I picked the SVM next because it's good at finding complex patterns that don't split with a straight line. I used SVC with an RBF kernel since my early tests showed a linear kernel didn't work well and the digits weren't linear separable. I set up a parameter grid with two settings: C at [1.0, 5.0, 10.0], which decides how much the model sticks to the training data, and gamma at ["scale", 0.01, 0.1], which helps it fit the data better. I started with a bigger range for C [0.1 to 100.0] and gamma [0.001 to 0.1], but after testing, I found 5.0 to 10.0 worked best for C. I used GridSearchCV for tuning, which I'll explain in 3.5.

Random Forest: I wanted to try a tree-based model also, so I picked Random Forest since it can work with patterns that are non-linear. I used the RandomForestClassifier and set up a parameter grid with three settings: number of trees (n_estimators) at [100, 200, 300], max depth (max_depth) at [None, 20, 30], minimum samples to split a node (min_samples_split) at [2, 5]. At first, I thought about adding more settings to avoid overfitting, but kept it simple with just these. I also started with fewer trees [50, 100] but saw that more trees worked better in my tests. I used GridSearchCV for tuning, which I'll go over in 3.5.

Ensemble Model: For my fourth model, I put together the SVM and Random Forest to see if they would do better as a set. I used the VotingClassifier with hard voting, where the model picks the digit with the most votes from the two. I tried soft voting too, but it wasn't much better, so I went with hard voting since it's simpler and faster. I thought about adding Logistic Regression to the set, but I left it out because it would probably only lower the score.

Augmented SVM: This one was the same as the regular SVM but with data augmentation added during training (explained in 3.7) to help it work better with different handwriting styles.

I was quite happy with my lineup of models and to see which one would work best for my project. Some simple, some tree-based, and some more advanced.

3.5 Hyperparameter Tuning

To find the best parameters for my models, I used GridSearchCV from Scikit-learn to adjust their settings on the training set (50,000 images). I know cross-validation already validates the models, but I still used an extra validation set (10,000 images) to test them on new data and only let the best model (SVM) go to the test set later (Géron, 2022, Chapter 2). I picked 3-fold cross-validation because it was a good middle ground, it didn't take too long and still gave me results I felt I could trust. I tried 5 folds at first but it was too slow and didn't change the results much. I used accuracy to score the models since the dataset was fairly balanced.

For Logistic Regression, I adjusted the regularization setting C, trying values [0.01, 0.05, 0.2, 0.5] to find a good fit, like mentioned in 3.4. For Random Forest, I worked on three settings, also from 3.4: the number of trees [100, 200, 300], max depth [None, 20, 30], and how many samples to split a branch [2, 5]. For the SVM, I tuned the C setting [1.0, 5.0, 10.0] and gamma ["scale", 0.01, 0.1], as mentioned in 3.4. The Ensemble model didn't need extra tuning since it used the best settings from the SVM and Random Forest and just voted on the answers. The Augmented SVM used the same tuning as the regular SVM but added augmentation during training, which I'll explain in 3.7.

3.6 Final Model

I evaluated all the models on the validation set and used accuracy and the confusion matrix to pick the best one. The winner was the SVM, so I retrained it on the full train-validation set (60,000 images) to help it learn as much as it could before testing it on the test set (10,000 images) (Géron, 2022).

To answer my research question about how data augmentation helps, I tested both the non-augmented SVM and the augmented SVM in a more real-life way using the Streamlit app. I drew digits on the app's drawing-canvas to see how they would do against each other. Choosing the drawing-canvas and not the camera kept things on a simple level since I had to try a lot of drawings and keep it as fair as possible.

3.7 Augmentation

To help my final model work better with real-life handwriting, I used the imgaug library to do data augmentation (Jung, n.d.). First, I had to change the shape of the train-validation data (60,000 samples) from a flat 784-pixel format to a 3D array (-1, 28, 28) using NumPy's reshape. Imgaug needs this shape to work with images in batches.

I picked augmentation because my project is about recognizing digits from real-life input in the Streamlit app, and real-life data can look very different. Digits might be tilted, off-center, or oddly shaped because of how people write or the image quality (Géron, 2022, Chapter 14). Augmentation simulates all this by doing things like:

- Rotating the images by ±10 degrees to look like tilted writing or camera angles.
- Shifting the images up to 10% in both directions to match off-center digits.
- Adding small distortions to make the handwriting shapes look more natural.

This way, the model could learn from a bigger set of examples and get better at working with real-life inputs. I made three new versions of each image, plus kept the original, so that's x4 per image. That made the dataset grow to about 240,000 images, which gave the model a lot more to learn from. Still, it only took a couple of minutes, which was faster than I expected.

After that, I changed the images back to the flat 784-pixel format because the SVC model needs a 2D array to work with. Then I used the best settings from the SVM tuning in 3.5 and retrained the Augmented SVM on this bigger dataset.

I tried a few things along the way. At first, I used bigger rotations (±20 degrees) and stronger distortions, but the digits started looking too odd, so I went with smaller changes. I also started with just two new versions per image, but then I added a third to give the model more examples. I found that augmentation was a good way to make my model better without doing anything too complicated with the features.

3.8 Streamlit

To test my digit recognition model in a real-life way (Géron, 2022, Chapter 2), I made a Streamlit app using the Streamlit library in Python and linked it to my trained model (Streamlit, n.d.). I built it so it could recognize handwritten digits from different inputs - drawings, uploaded pictures, and camera shots, matching real life use.

I started with a local app (app.py) and then changed it to work on Streamlit Cloud (app_cloud.py), which was harder than I thought. The three ways to input digits are set up as tabs: a drawing canvas where users can draw, an image uploader for pictures, and a camera option to take photos. For the drawing canvas, I made it 28x28 pixels to match MNIST's size, so users can draw digits with a brush tool. For uploaded pictures and camera shots, I had to adjust them using OpenCV (OpenCV, n.d.) and Pillow (Pillow, n.d.) by making them grayscale, resizing them to 28x28 pixels, and dividing the pixel values by 255 to get them between 0 and 1 (Géron, 2022). This made sure the images were the same as the training data.

The camera part was tricky to build for cloud use. I first tried a live feed for Streamlit Cloud using WebRTC, but I had problems with it and got stuck with different Python versions, which made the setup really unstable. After a lot of trouble, I switched to snapshots instead, where users take one picture at a time. That worked much better for the Streamlit Cloud app. My local app (app.py) still lets users use a live feed with a checkbox to start or stop the camera, a slider to adjust the settings, and a box to show the digit. The cloud version (app_cloud.py) just uses snapshots, but both versions send the adjusted images to the model and show the digit it guesses.

I tested the app on my computer first to try everything out, then put it on Streamlit Cloud to share with others.

4 Results and Discussion

This chapter presents the performance of the evaluated models, focusing on accuracy and real-life test results. Detailed results and additional visualisations are provided in the Appendix.

4.1 Model Performance

Table 4.1 summarizes the validation and test accuracy of the models. After tuning in 3.5, the SVM did the best on the validation set with an accuracy of 0.9852, so I picked it as my final model. I retrained it on the full train-validation set (60,000 images) and tested it on the test set (10,000 images), like I explained in 3.6.

To answer my research question about data augmentation, I tested both the non-augmented SVM and the augmented SVM. The Augmented SVM got a test accuracy of 0.9878, which was 0.0063 better than the non-augmented SVM's 0.9815. I was happy to see that augmentation made a difference! I also show the accuracies for the other models in Table 4.1 to see how they compare, but I only tested the SVM on the test set since it was the best. My results are pretty close to what LeCun et al. (1998) got with SVMs, even though I didn't tune my models much, which I think is cool.

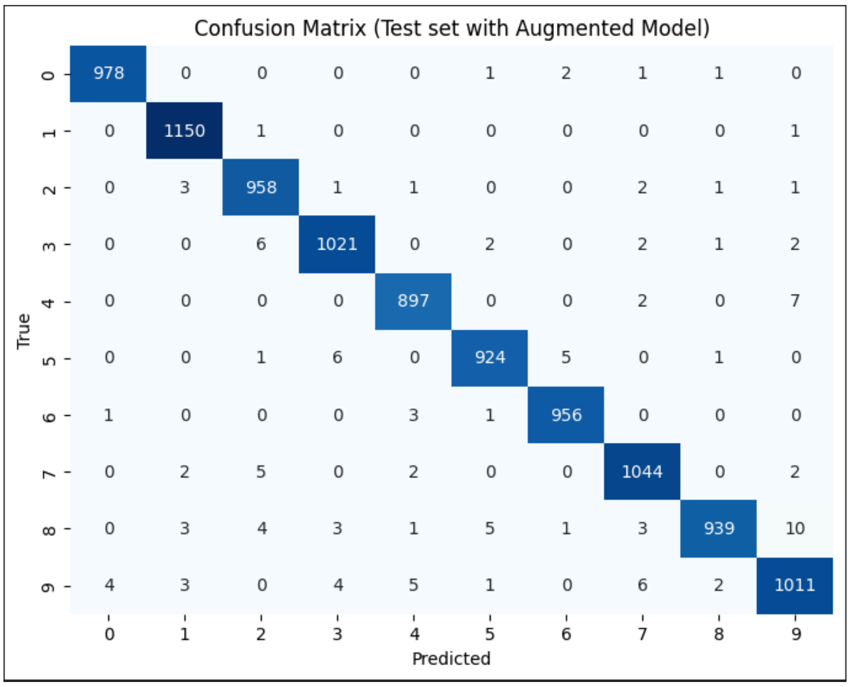
I also made a confusion matrix for the Augmented SVM, shown in Figure 4.1. It shows where the model made mistakes, like guessing a 9 when it was really an 8.

[Table 4.1: Model Performance Comparison]

Model	Training Accuracy	Validation Accuracy	Test Accuracy
Logistic Regression	0.9202	0.9233	
SVM	0.9806	0.9852	0.9815
Random Forest	0.9665	0.9713	
Ensemble		0.9782	
Augmented SVM		0.9852	0.9878

Note: Validation accuracy reflects the best model after tuning (3.5), while test accuracy comes from the final retrained model on the combined train-validation set (3.6), evaluated on the unseen test set.

[Figure 4.1: Confusion Matrix for Augmented SVM]



True \ Predicted	0	1	2	3	4	5	6	7	8	9
0	978	0	0	0	0	1	2	1	1	0
1	0	1150	1	0	0	0	0	0	0	1
2	0	3	958	1	1	0	0	2	1	1
3	0	0	6	1021	0	2	0	2	1	2
4	0	0	0	0	897	0	0	2	0	7
5	0	0	1	6	0	924	5	0	1	0
6	1	0	0	0	3	1	956	0	0	0
7	0	2	5	0	2	0	0	1044	0	2
8	0	3	4	3	1	5	1	3	939	10
9	4	3	0	4	5	1	0	6	2	1011

Caption: Confusion matrix for the Augmented SVM on the test set, showing minor misclassifications (10 actual 8's were classified as 9's).

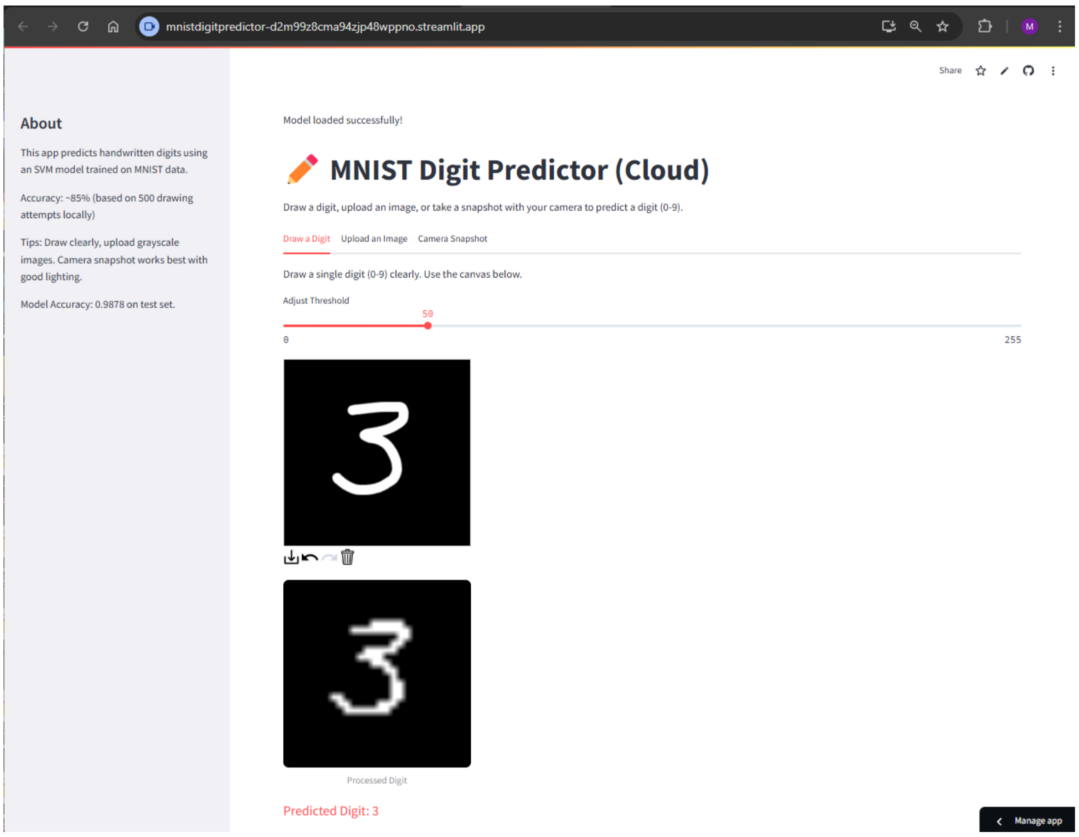
4.2 Streamlit application

I tested the non-augmented and augmented SVMs using the Streamlit app's drawing canvas to see how they would work with real-life drawings. The non-augmented SVM got an accuracy of 65.5% (131 out of 200 tries), while the augmented SVM did better at 74% (370 out of 500 tries). That's 8.5% better, which I thought was great because it showed how augmentation helped with the different ways people draw digits. Digits 6 and 8 were the hardest for the models. The non-augmented SVM only got a per-class accuracy of 25% right for 6 (5 out of 20) and 30% for 8 (6 out of 20), but the augmented SVM did better at 42% for 6 (21 out of 50) and 40% for 8 (20 out of 50). I also tried camera snapshots and image uploads, and although they worked relatively good, they were less consistent and I had to keep changing the threshold settings to deal with lighting and placing of the digits.

4.2.1 Improved Streamlit Adjustments and Updated Results

Trying to enhance the performance I made some last minute changes to the Streamlit app to see if I could make it any better by changing to cv2.INTER_CUBIC for resizing and adding a slider to adjust the threshold, hoping to help with tough digits like 6 and 8, which were hard before. I tested it again with 500 drawings (50 per digit), and the accuracy went up to 84.6% (423/500), which is 10.6% better than the earlier 74% (370/500). Digits like 7 (now 100%), 9 (86%), 8 (62%), and 6 (62%) got a lot better. Due to time limitations I couldn't perform a direct comparison with the non-augmented SVM anymore, but I'm happy with how the changes helped overall. See Table A.4.1 for the full results.

Figure 4.2: Streamlit Application Interface



mnistdigitpredictor-d2m99z8cma94zjp48wppno.streamlit.app

About

This app predicts handwritten digits using an SVM model trained on MNIST data.

Accuracy: ~85% (based on 500 drawing attempts locally)

Tips: Draw clearly, upload grayscale images. Camera snapshot works best with good lighting.

Model Accuracy: 0.9878 on test set.

Model loaded successfully!

MNIST Digit Predictor (Cloud)

Draw a digit, upload an image, or take a snapshot with your camera to predict a digit (0-9).

Draw a Digit | Upload an Image | Camera Snapshot

Draw a single digit (0-9) clearly. Use the canvas below.

Adjust Threshold: 50 (0–255)

Processed Digit

Predicted Digit: 3

Caption: Screenshot of the Streamlit app's drawing canvas (cloud version) showing a "3" I drew, with the model guessing it as a 3.

5 Conclusions

This project looked at how data augmentation changes the accuracy of machine learning models for recognizing handwritten digits. My main goal was to build a system that works well in real-life situations, which I tested with a Streamlit app that lets users draw, upload pictures, or use a camera.

The results showed that data augmentation really helped, answering my research question: How does data augmentation affect a model's accuracy? The Augmented SVM did the best with a test accuracy of 0.9878, which was 0.0063 better than the non-augmented SVM's 0.9815. In the Streamlit app's drawing tests, the augmented SVM got 74% right (370 out of 500 tries), 8.5% better than the non-augmented SVM's 65.5% (131 out of 200 tries). I'm happy to see that augmentation, by adding things like rotations and shifts, really makes the model better at handling different user drawings.

But then there's the drop in the app's accuracy (74% compared to 0.9878) which showed some problems. Augmentation helped with some real-life differences, but camera snapshots were hard to work with because of lighting, how papers were held and how digits were placed. Digits 6 and 8 were the toughest in the drawing tests, with the augmented SVM getting only 42% for 6 and 40% for 8, even though it did much better on the test set (99.5% for 6, 96.9% for 8, as shown in the confusion matrix in Figure 4.1). This made me see how different the controlled MNIST data is from real-life drawings.

Looking back, I think using both cross-validation for hyperparameter tuning and a separate validation set for picking the model might have been too much in this case. Just cross-validation could have worked, but the extra validation set nevertheless made me feel better about my choice of model, even if it took more time. For smaller datasets in the future, I'd probably stick to cross-validation to save time.

Overall, data augmentation was a good way to make my model better at recognizing digits, especially in real-life tests, but digits like 6 and 8 showed there's still work to do. In the future, I'd like to try other models like XGboost or maybe convolutional neural networks, which are better suited for image tasks (Géron, 2022, Chapter 14). My model could also use stronger augmentation or improve the Streamlit app with better image adjustments, especially for the hard digits.

I tried improving the Streamlit app like mentioned earlier, using cv2.INTER_CUBIC for resizing and a threshold slider. It worked great - the accuracy jumped to 84.6% (423/500) from 74% (370/500)! Tough digits like 9 (86%), 8 (62%), and 6 (62%) got way better now. I didn't have time to try other models or compare with the non-augmented SVM, but I think this shows how better adjustments can make a big difference.

Appendix A

This appendix has extra results and figures to help explain what I found in my study.

A.1 GitHub Repository

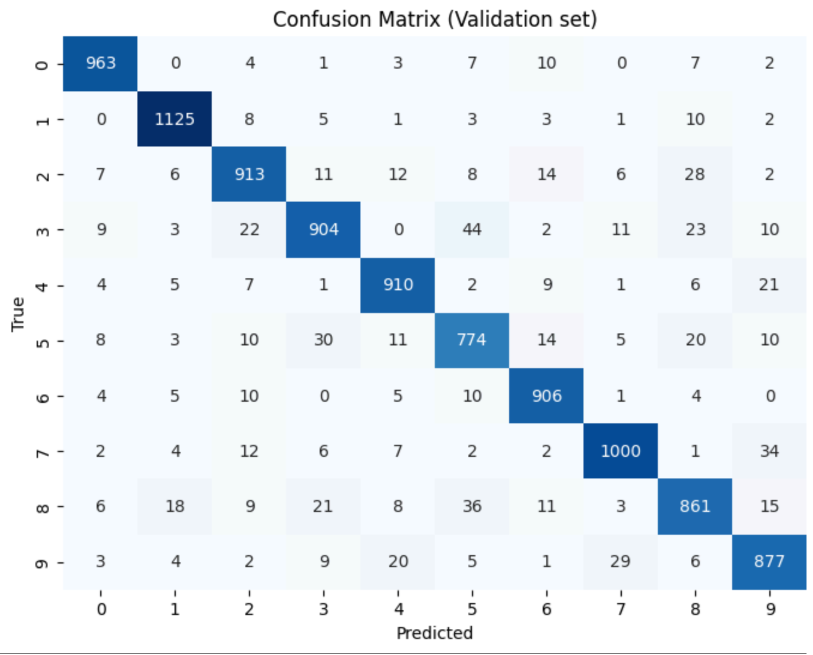
Here’s the link to my GitHub repository where I stored all my code for this project:

https://github.com/mabjq/mnist_digit_predictor

A.2 Validation Results for All Models

These figures show the confusion matrices from the validation set testings (10,000 images) for each model, as described in 3.5 and 3.6.

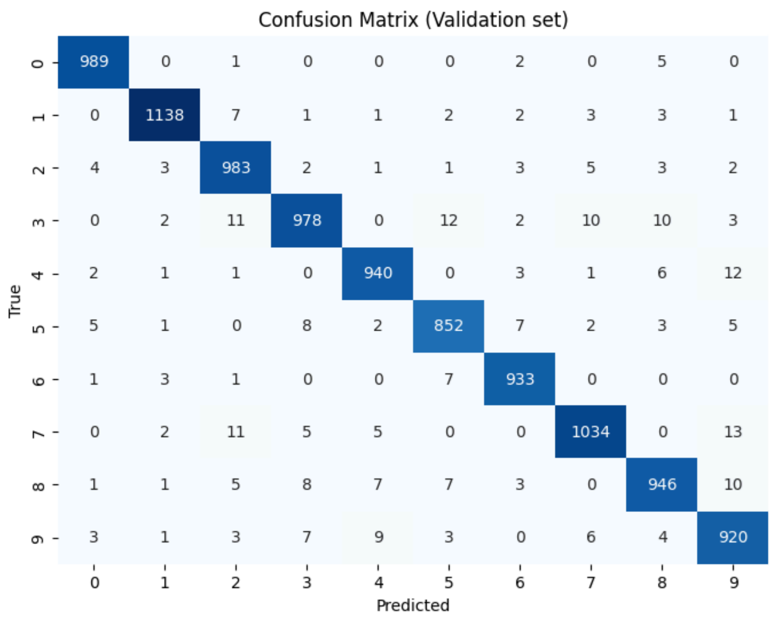
Figure A.2.1: Confusion Matrix for Logistic Regression (Validation Set)



True \ Predicted	0	1	2	3	4	5	6	7	8	9
0	963	0	4	1	3	7	10	0	7	2
1	0	1125	8	5	1	3	3	1	10	2
2	7	6	913	11	12	8	14	6	28	2
3	9	3	22	904	0	44	2	11	23	10
4	4	5	7	1	910	2	9	1	6	21
5	8	3	10	30	11	774	14	5	20	10
6	4	5	10	0	5	10	906	1	4	0
7	2	4	12	6	7	2	2	1000	1	34
8	6	18	9	21	8	36	11	3	861	15
9	3	4	2	9	20	5	1	29	6	877

Caption: Confusion matrix for Logistic Regression on the validation set, with an accuracy of 0.9233.

Figure A.2.2: Confusion Matrix for Random Forest (Validation Set)

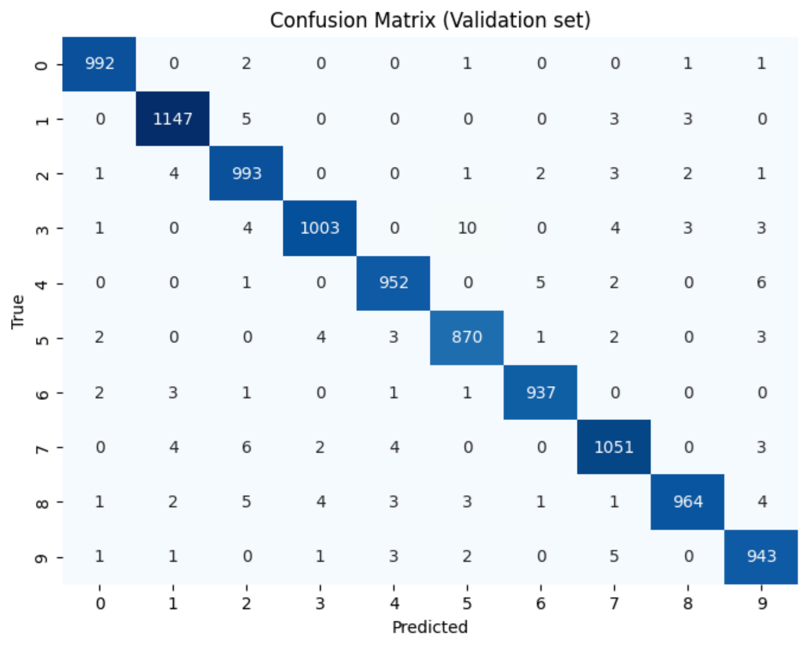


Confusion Matrix (Validation set)

True \ Predicted	0	1	2	3	4	5	6	7	8	9
0	989	0	1	0	0	0	2	0	5	0
1	0	1138	7	1	1	2	2	3	3	1
2	4	3	983	2	1	1	3	5	3	2
3	0	2	11	978	0	12	2	10	10	3
4	2	1	1	0	940	0	3	1	6	12
5	5	1	0	8	2	852	7	2	3	5
6	1	3	1	0	0	7	933	0	0	0
7	0	2	11	5	5	0	0	1034	0	13
8	1	1	5	8	7	7	3	0	946	10
9	3	1	3	7	9	3	0	6	4	920

Caption: Confusion Matrix for Random Forest on the validation set, with an accuracy of 0.9713.

Figure A.2.3: Confusion Matrix for SVM (Non-Augmented, Validation Set)

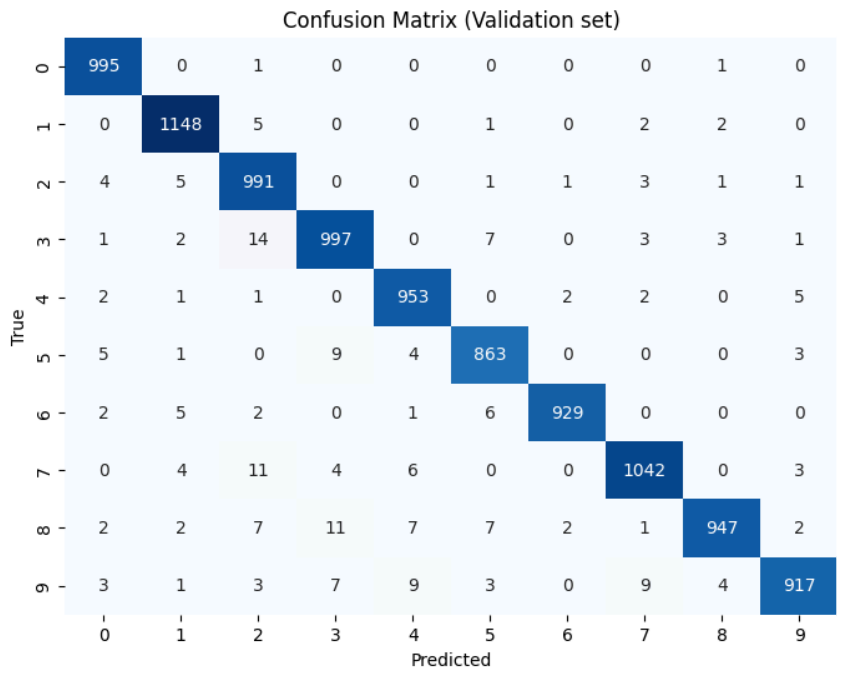


Confusion Matrix (Validation set)

True \ Predicted	0	1	2	3	4	5	6	7	8	9
0	992	0	2	0	0	1	0	0	1	1
1	0	1147	5	0	0	0	0	3	3	0
2	1	4	993	0	0	1	2	3	2	1
3	1	0	4	1003	0	10	0	4	3	3
4	0	0	1	0	952	0	5	2	0	6
5	2	0	0	4	3	870	1	2	0	3
6	2	3	1	0	1	1	937	0	0	0
7	0	4	6	2	4	0	0	1051	0	3
8	1	2	5	4	3	3	1	1	964	4
9	1	1	0	1	3	2	0	5	0	943

Caption: Confusion Matrix for the non-augmented SVM on the validation set, with an accuracy of 0.9852.

Figure A.2.4: Confusion Matrix for Ensemble Model (Validation Set)

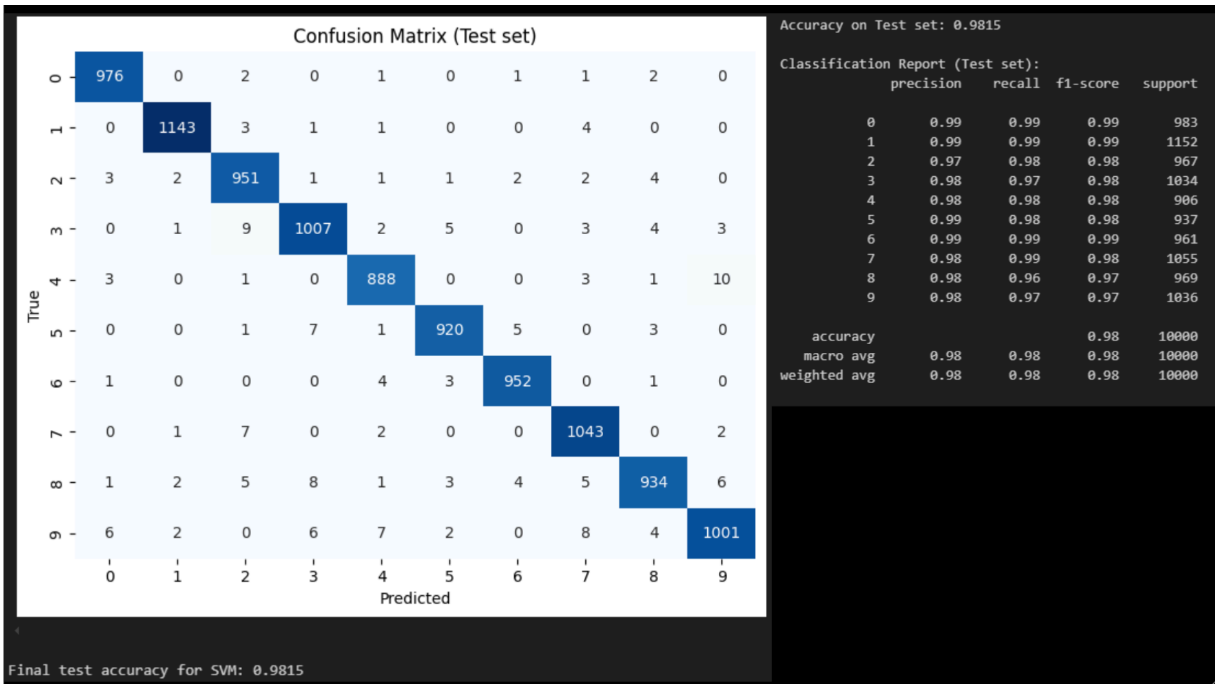


Confusion Matrix (Validation set)

True \ Predicted	0	1	2	3	4	5	6	7	8	9
0	995	0	1	0	0	0	0	0	1	0
1	0	1148	5	0	0	1	0	2	2	0
2	4	5	991	0	0	1	1	3	1	1
3	1	2	14	997	0	7	0	3	3	1
4	2	1	1	0	953	0	2	2	0	5
5	5	1	0	9	4	863	0	0	0	3
6	2	5	2	0	1	6	929	0	0	0
7	0	4	11	4	6	0	0	1042	0	3
8	2	2	7	11	7	7	2	1	947	2
9	3	1	3	7	9	3	0	9	4	917

Caption: Confusion Matrix for the Ensemble model on the validation set, with an accuracy of 0.9782.

Figure A.2.5: Confusion Matrix for SVM (Non-Augmented, Test Set)



Confusion Matrix (Test set)

True \ Predicted	0	1	2	3	4	5	6	7	8	9
0	976	0	2	0	1	0	1	1	2	0
1	0	1143	3	1	1	0	0	4	0	0
2	3	2	951	1	1	1	2	2	4	0
3	0	1	9	1007	2	5	0	3	4	3
4	3	0	1	0	888	0	0	3	1	10
5	0	0	1	7	1	920	5	0	3	0
6	1	0	0	0	4	3	952	0	1	0
7	0	1	7	0	2	0	0	1043	0	2
8	1	2	5	8	1	3	4	5	934	6
9	6	2	0	6	7	2	0	8	4	1001

```
Accuracy on Test set: 0.9815

Classification Report (Test set):
              precision    recall  f1-score   support

           0       0.99      0.99      0.99       983
           1       0.99      0.99      0.99      1152
           2       0.97      0.98      0.98       967
           3       0.98      0.97      0.98      1034
           4       0.98      0.98      0.98       906
           5       0.99      0.98      0.98       937
           6       0.99      0.99      0.99       961
           7       0.98      0.99      0.98      1055
           8       0.98      0.96      0.97       969
           9       0.98      0.97      0.97      1036

    accuracy                           0.98     10000
   macro avg       0.98      0.98      0.98     10000
weighted avg       0.98      0.98      0.98     10000
```

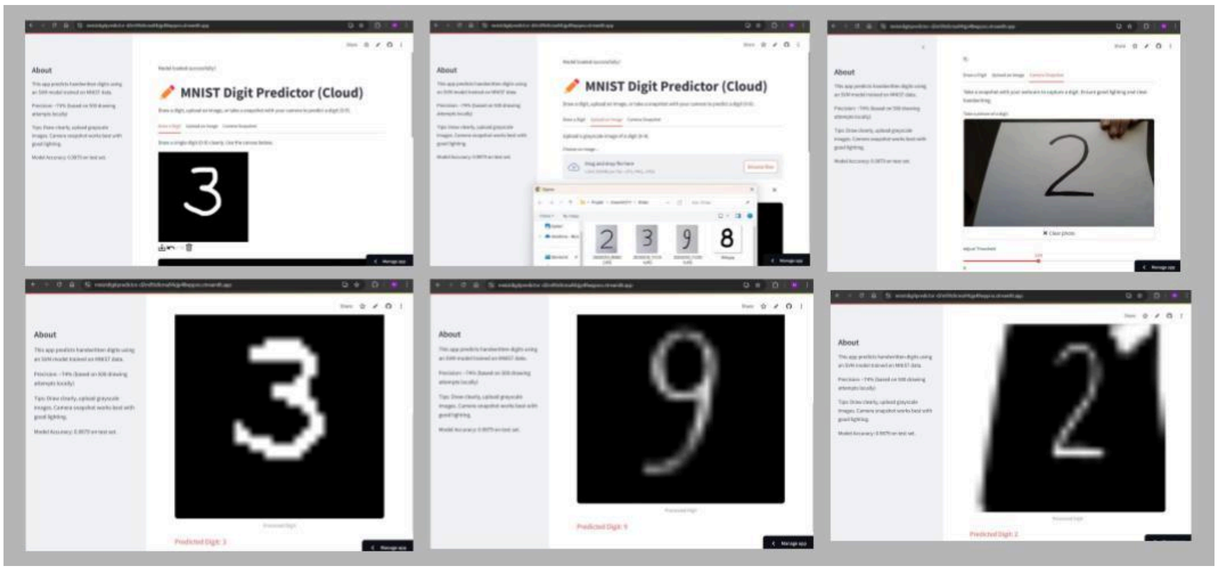
```
Final test accuracy for SVM: 0.9815
```

Caption: Confusion Matrix for the non-augmented SVM on the test set, with an accuracy of 0.9815, for comparison with the augmented SVM (Figure 4.1).

A.3 Streamlit Application Screenshots

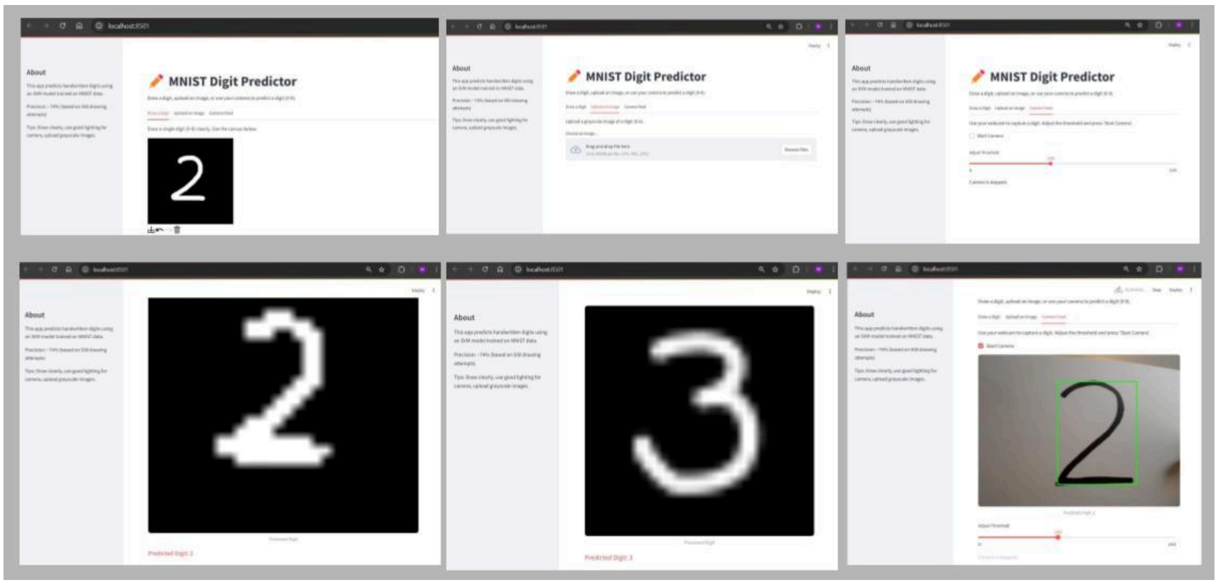
These figures show a full view of my Streamlit app's interface for both cloud and local versions, adding to the single screenshot in Figure 4.2

Figure A.3.1: Streamlit Application Screenshots (Cloud Version)



Caption: Screenshots of the cloud-based Streamlit app, showing (a) canvas written 3, (b) canvas result predicted as 3, (c) upload image of 9, (d) upload result predicted as 9, (e) camera snapshot feed, (f) camera snapshot result predicted as 2 .

Figure A.3.2: Streamlit Application Screenshots (Local Version)



Caption: Screenshots of the local Streamlit app, showing (a) canvas written 2, (b) canvas result predicted as 2, (c) upload image of 2, (d) upload result predicted as 3, (e) camera live feed, (f) camera feed result predicted as 2.

A.4 Per-Digit Accuracy from Canvas Trials

This table shows how well each digit did in the Streamlit drawing tests (from 4.2), so I could see which ones were harder for the models to predict.

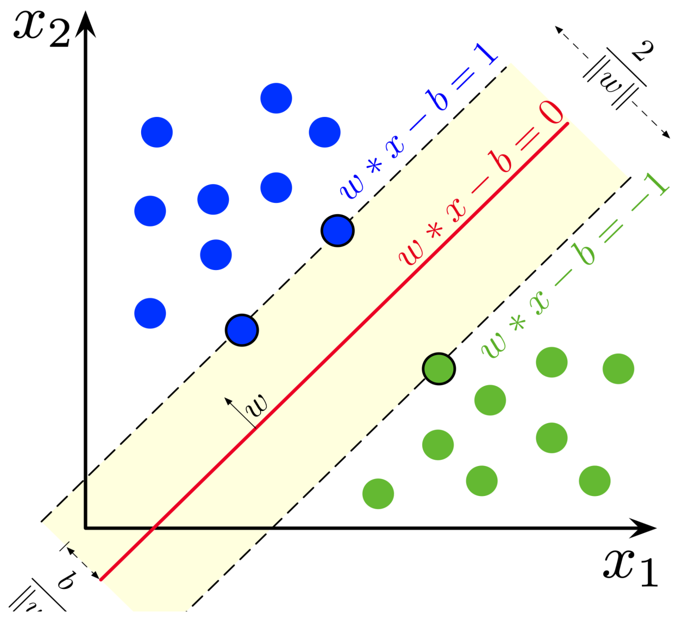
Table A.4.1: Per-Digit Accuracy from Streamlit Canvas Trials

Digit	Non-Augmented SVM (20 trials)	Augmented SVM (50 trials)	Augmented SVM with Improved Adjustment. (50 trials)
0	55% (11/20)	88% (44/50)	94% (47/50)
1	95% (19/20)	92% (46/50)	92% (46/50)
2	85% (17/20)	68% (34/50)	96% (48/50)
3	60% (12/20)	48% (24/50)	78% (39/50)
4	80% (16/20)	98% (49/50)	96% (49/50)
5	75% (15/20)	90% (45/50)	80% (40/50)
6	25% (5/20)	42% (21/50)	62% (31/50)
7	95% (19/20)	92% (46/50)	100% (50/50)
8	30% (6/20)	40% (20/50)	62% (31/50)
9	55% (11/20)	90% (45/50)	86% (43/50)
Total	65.5% (131/200)	74% (370/500	84.6% (423/500)

Note: Per-class accuracy for each digit in the Streamlit drawing tests for all models. The "Augmented SVM with Improved Adjustments" column reflects the results after implementing cv2.INTER_CUBIC for interpolation and a threshold adjustment slider.

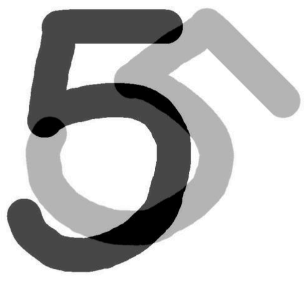
A.5 Additional Visualizations from the Report

Figure A.5.1: Maximum-Margin Hyperplane and Margins for an SVM



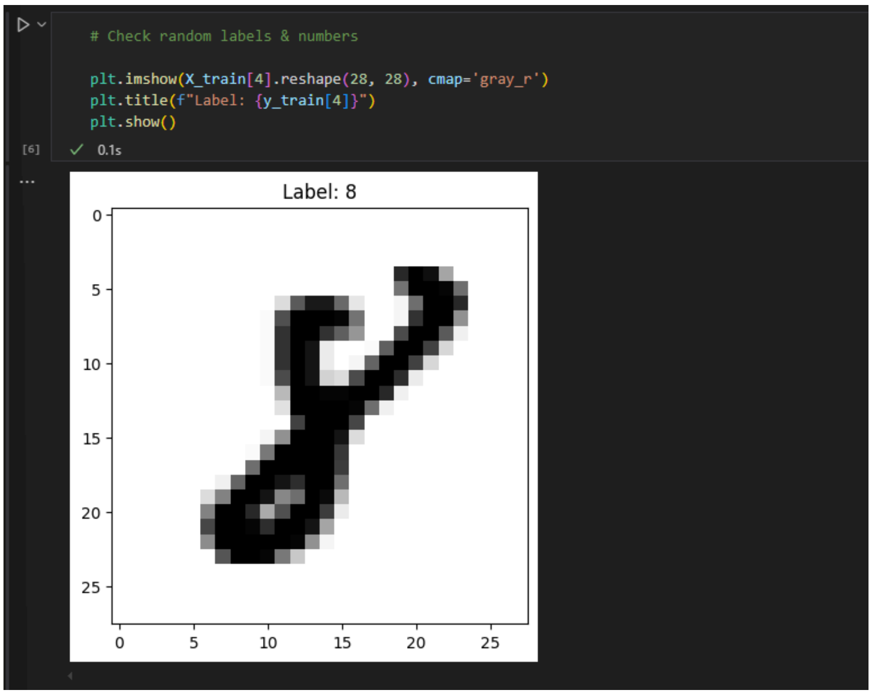
Caption: Diagram of the maximum-margin hyperplane and margins for an SVM, showing support vectors (Larhmam, 2016). Adapted from Wikipedia, "Support Vector Machine," by Larhmam, licensed under CC BY-SA 4.0.

Figure A.5.2: Example of an Augmented Digit



Caption: Example of an augmented digit from 2.4, showing an original "5" in black and a rotated "5" in grey.

Figure A.5.3: Sample MNIST Digit



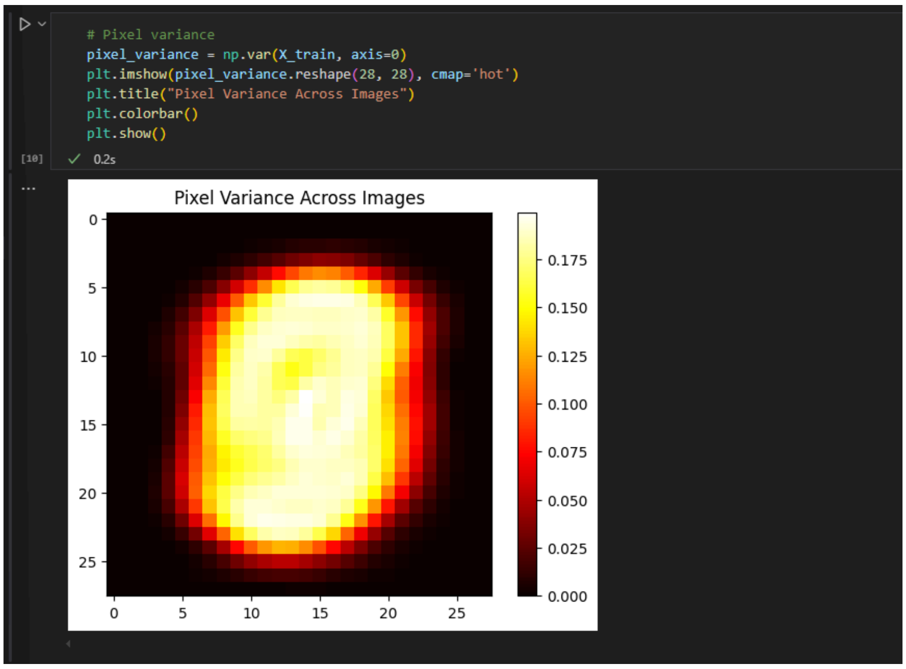
```
# Check random labels & numbers

plt.imshow(X_train[4].reshape(28, 28), cmap='gray_r')
plt.title(f"Label: {y_train[4]}")
plt.show()
```

```
[6]  ✓ 0.1s
```

Caption: Sample MNIST digit (a "7") from 3.2, showing typical handwriting differences.

Figure A.5.4: Pixel Variance Across MNIST Digits



```
# Pixel variance
pixel_variance = np.var(X_train, axis=0)
plt.imshow(pixel_variance.reshape(28, 28), cmap='hot')
plt.title("Pixel Variance Across Images")
plt.colorbar()
plt.show()
```

```
[10]  ✓ 0.2s
```

Caption: Map of pixel changes across MNIST digits from 3.2, showing that the center pixels change the most.

References

- Géron, A. (2022). *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems (3rd ed.)*. O'Reilly Media.

- Jung, A. (n.d.). *imgaug documentation.* https://imgaug.readthedocs.io/

- Larhmam. (2016). *Support vector machine* [Image]. Wikipedia. https://en.wikipedia.org/wiki/Support_vector_machine#/media/File:SVM_margin.png

- LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE, 86*(11), 2278–2324. https://doi.org/10.1109/5.726791

- OpenCV. (n.d.). *OpenCV documentation*. https://docs.opencv.org/

- Scikit-learn. (n.d.). *Scikit-learn documentation*. https://scikit-learn.org/stable/

- Streamlit. (n.d.). *Streamlit documentation*. https://docs.streamlit.io/

- Pillow. (n.d.). *Pillow (PIL Fork) documentation*. https://pillow.readthedocs.io/