INF264 Project 2

Language: **Python**
Group: **Project 2-1**
Students:

- Mats Omland Dyrøy (**mdy020**)
- Linus Krystad Raaen (**zec018**)

Summary

The Chief Elf Officer (CEO) of Santa's Workshop have reached out to us for help sorting gifts.
All the gifts are marked with a handwritten digit 0-9, A-F.
Our task was to train a machine learning model to identify these digits that the CEO could then use.

We estimate the accuracy of our final model to be `96%` on data it has never seen before.
Naturally, a child not getting their present is unacceptable for Santa, and thus our model is not good enough for its intended purpose.

Run program

If you want to run the program yourself, do the following:

1. Ensure current working directory is the root of this project.
2. Start `code/main.py`.

Logs and images will be created in `dump/<today>/`.

NOTE: The program takes around `2 hours` to run.
For the convenience of the person grading this assignment, we have already run the program multiple times.

Technical report

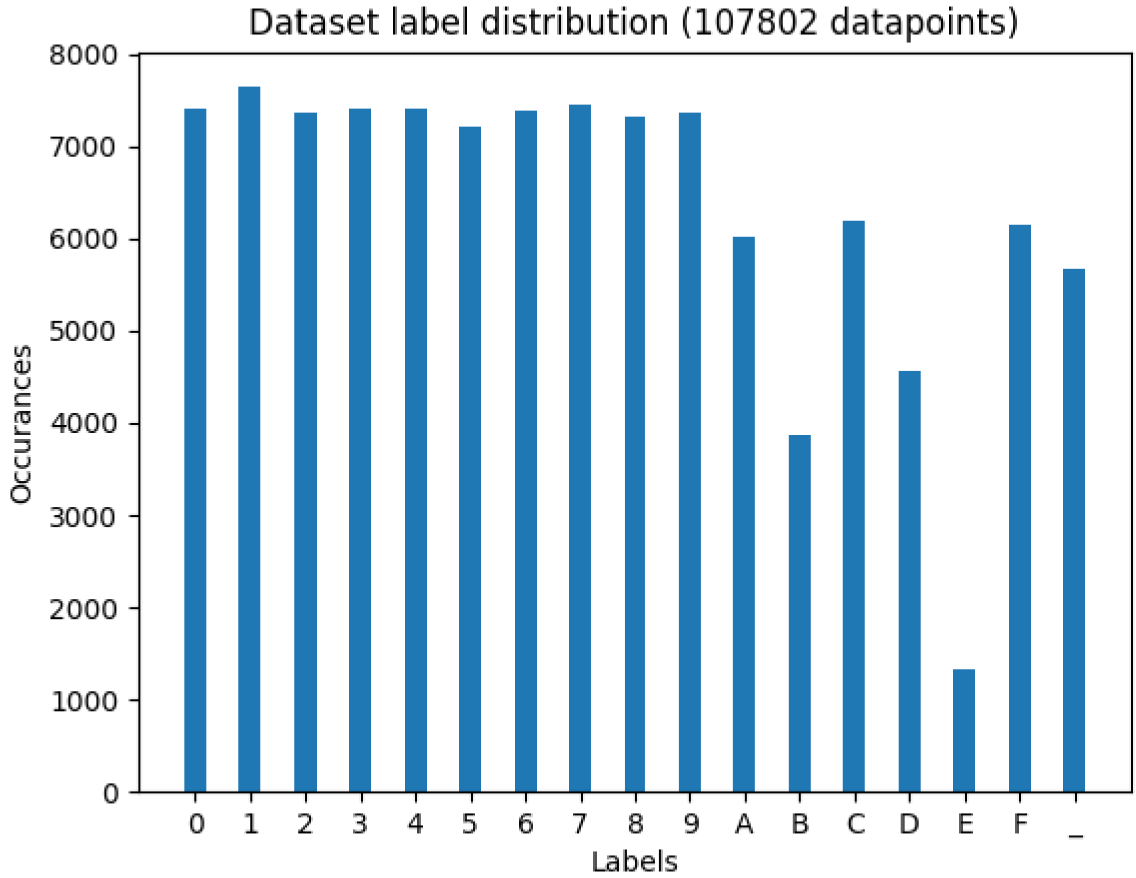
In this section we will discuss the following topics.

- Data observations
- Model selection
- Model candidates
- Final classifier
- Future improvements

The report is based on the files in `dump/report/` which is a copy of `dump/2023-10-13-1441/`.
The time estimates will vary from computer to computer and sometimes from run to run.

Data observations

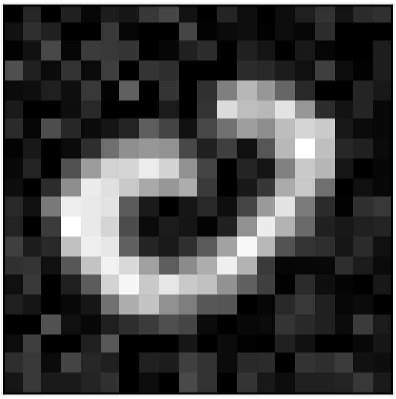
Before choosing models and hyper-parameters, we need to look at the dataset we are trying to generalize into knowledge.



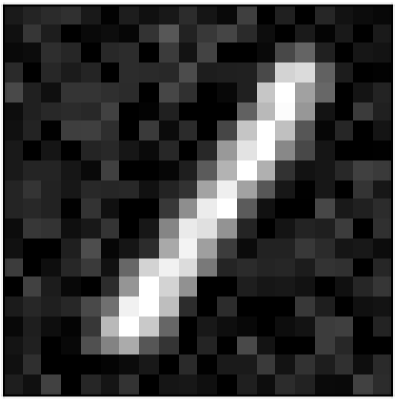
(Figure 1-1: Label distribution)
Initially we expected to se a uniform dataset, but it appears the `E`, `B` and `D` labels are significantly underrepresented.
This might impact model performance, but out final model `sklearn.svm-poly3` does not seem to take issue with labelling `E`.

The following image is very long and includes an example of all the different possible labels.
We immidiatly notice that the images are very noisy, but to avoid overfitting and to make our models more resilient to small changes, we decided not to remove the noise.

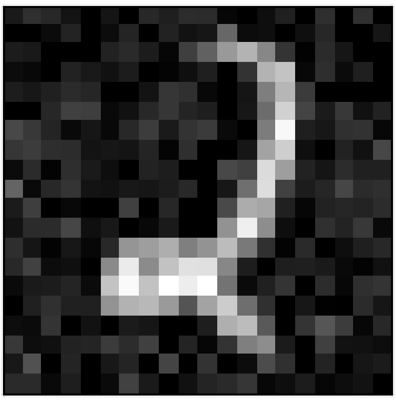
Actual 0



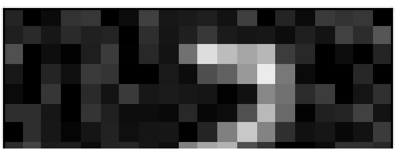
Actual 1

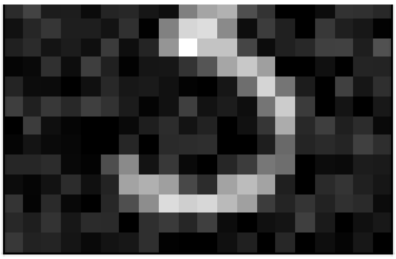


Actual 2

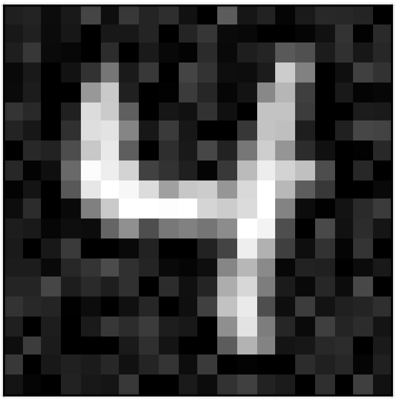


Actual 3

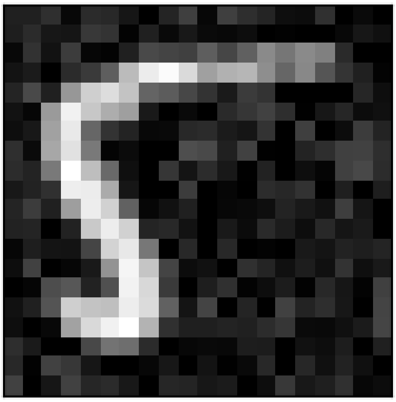




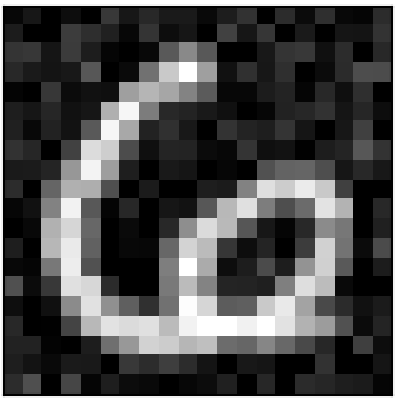
Actual 4



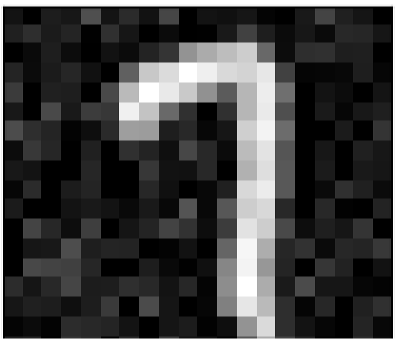
Actual 5



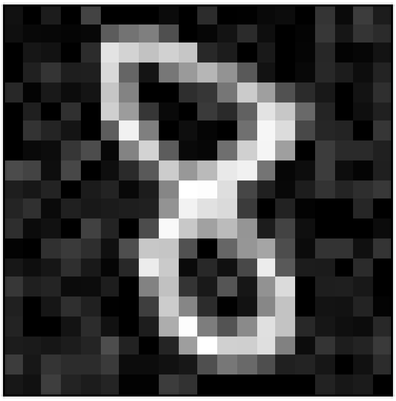
Actual 6



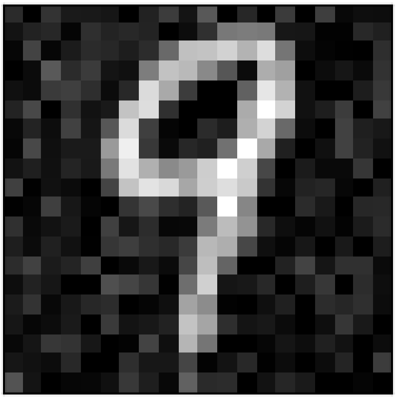
Actual 7



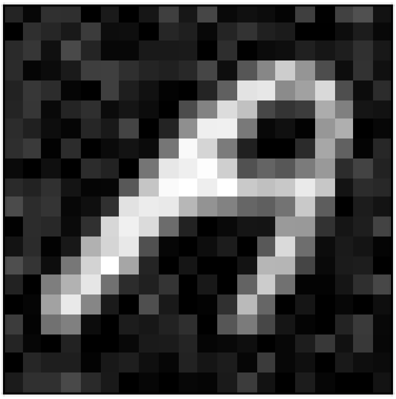
Actual 8



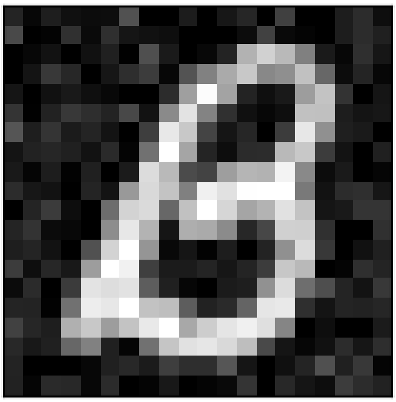
Actual 9



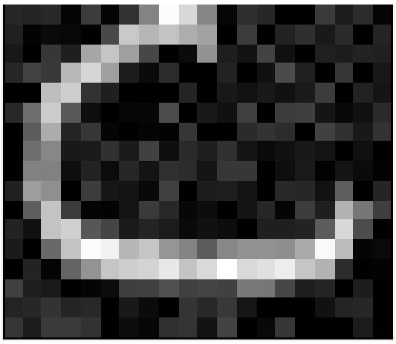
Actual A



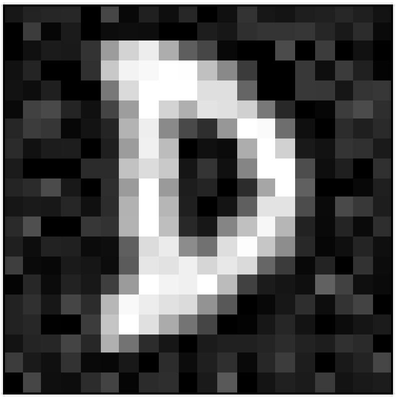
Actual B



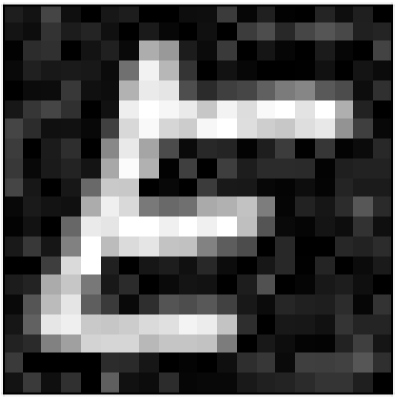
Actual C



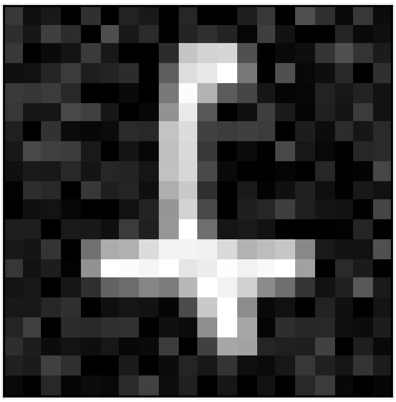
Actual D



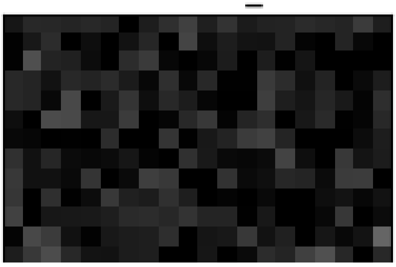
Actual E

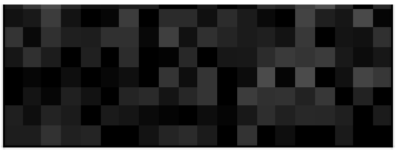


Actual F



Actual _





(Figure 1-2, Label examples)

Model selection

Here we go into detail about how we select the best model.
If you want to see which models are competing, see Model candidates.
We use `accuracy` as a performance measure because in the context of the model usage, an accurate model is a good model.
We also measure the time per prediction (*TPP in the logs*) and training time.
Note that the time measurements vary greatly from computer to computer and some from run to tun.

1. **Split the data**
   First we split the data into three datasets.
   `train` - Used to train the models.
   `val` - Used to pick the best model.
   `test` - Used to estimate real performance on best model.
   The reason we need both `val` and `train` is that when we choose the best performing model on `val`,

the selected model might have gotten lucky on the datapoints and performed *too* well.
In a way, we optimized for `val`.

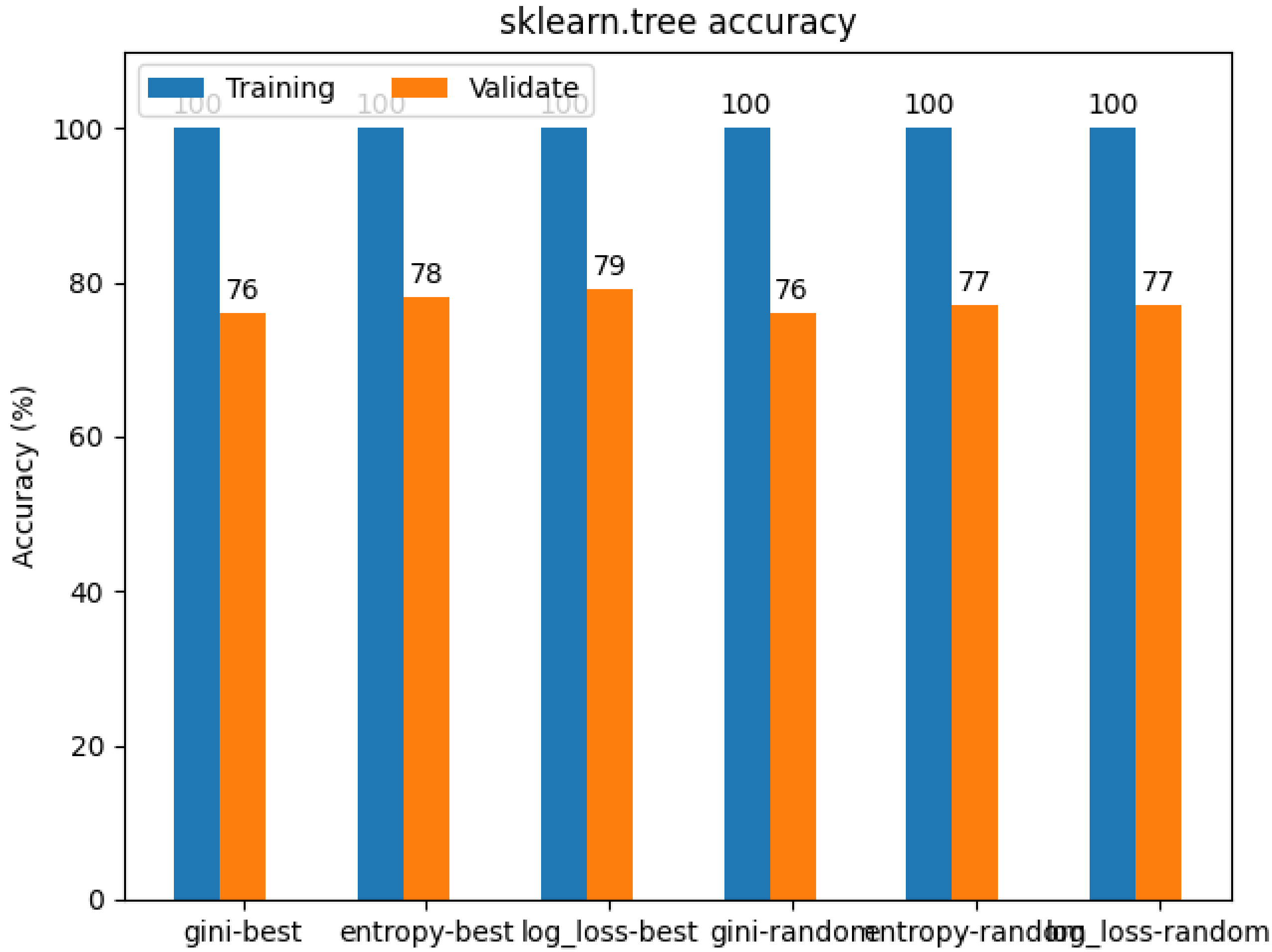
2. **Train and measure**
   The `train` dataset is given to the model trainers located in `code/model_trainers/`.
   The trainer located in `code/model_trainers/trainer.py` then splits the original `train` set into smaller `train` and `val` sets.
   The models are then trained on `train`-set and measured on `train` and `val`-set.
   We measure on both sets to identify overfitting.

3. **Pick local winner**
   Once all the models of a given type (like *Decision tree*) are trained and measured, we pick a winner amongst that type which is a candidate for out final model.
   We also log the performance of all the models of the type like this:

```
====== Group: sklearn.tree
Best model: sklearn.tree-log_loss-best

=== Model:        sklearn.tree-gini-best
Training size:    66203 pts.
Training time:    33.27s
train: Accuracy=100%, TPP=899ns, Size=66203, Duration=59.53ms
validate: Accuracy=76%, TPP=3730ns, Size=11683, Duration=43.58ms

=== Model:        sklearn.tree-entropy-best
Training size:    66203 pts.
Training time:    33.02s
train: Accuracy=100%, TPP=868ns, Size=66203, Duration=57.43ms
validate: Accuracy=79%, TPP=1226ns, Size=11683, Duration=14.32ms

=== Model:        sklearn.tree-log_loss-best
Training size:    66203 pts.
Training time:    33.29s
train: Accuracy=100%, TPP=932ns, Size=66203, Duration=61.68ms
validate: Accuracy=79%, TPP=1378ns, Size=11683, Duration=16.09ms

=== Model:        sklearn.tree-gini-random
Training size:    66203 pts.
Training time:    6.40s
train: Accuracy=100%, TPP=938ns, Size=66203, Duration=62.07ms
validate: Accuracy=75%, TPP=1399ns, Size=11683, Duration=16.35ms

=== Model:        sklearn.tree-entropy-random
Training size:    66203 pts.
Training time:    6.34s
train: Accuracy=100%, TPP=1095ns, Size=66203, Duration=72.47ms
validate: Accuracy=77%, TPP=1529ns, Size=11683, Duration=17.86ms

=== Model:        sklearn.tree-log_loss-random
Training size:    66203 pts.
Training time:    5.82s
```

```
train: Accuracy=100%, TPP=1421ns, Size=66203, Duration=94.10ms
validate: Accuracy=76%, TPP=1335ns, Size=11683, Duration=15.59ms
```

And generate a plot like this:



4. **Pick final classifier**
   Once all the models have been trained and measured, we pick the final classifier.
   We do this by testing all the *local winners* on the original `val`-dataset.
   We then pick the best-performing classifier.

5. **Evaluate final classifier**
   The current estimates we have for the final classifier are optimistic because we picket the model that performed best on these measurements.
   To estimeate real world performance, the final classifier is now tested on the `test`-dataset.
   The final test is logged like this:

```
====== Best model
 === Model:  sklearn.svm-poly3
 Training size:  66203 pts.
 Training time:  1.99min
 train: Accuracy=100%, TPP=3.26ms, Size=66203, Duration=3.60min
 validate: Accuracy=96%, TPP=2.98ms, Size=11683, Duration=34.81s
 test: Accuracy=96%, TPP=2.98ms, Size=13745, Duration=41.02s
 estimate: Accuracy=96%, TPP=2.99ms, Size=16171, Duration=48.43s
```

Here `training size` is the number of datapoints the model was trained on.
`Training time` is the time it took to train the model.
`train` contains the measurements from the training dataset.
`validate` contains the measurements from the validation set derrived from the training set.
`test` contains the meaurements from the original validation set. (*very poor naming, we know...*)
`estimate` contains the measurements from the test dataset.

We also make various other plots and measurements of the final classifier which you can read about here.

Model candidates

We decided to try four different types of models:

- K-Nearest Neighbor
- Decision Tree
- Support Vector Machine
- Multi-layer perception

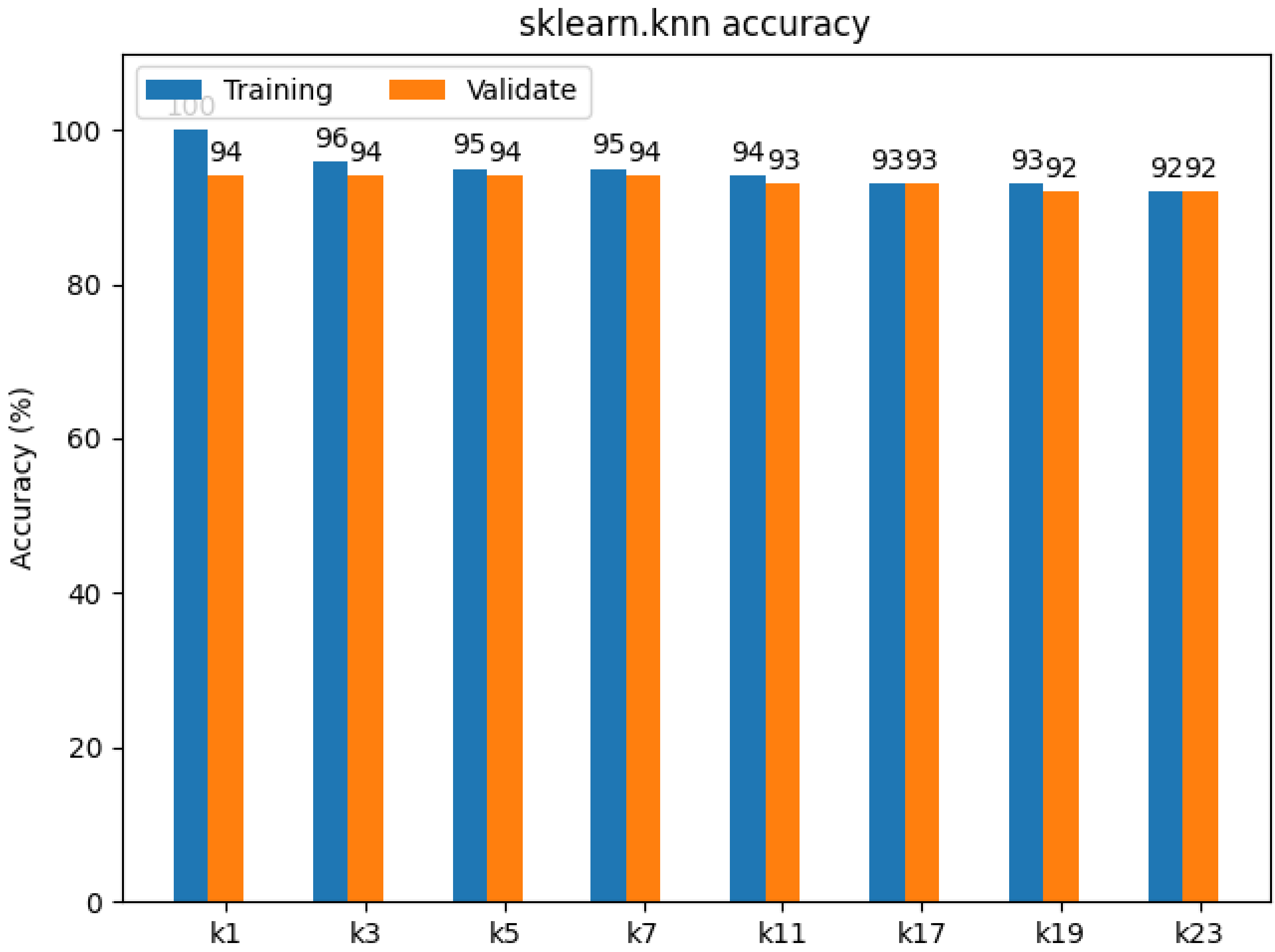
All the model implementations are from `sklearn` and were trained with various hyper-parameters.

K-Nearest Neighbor

(Code: `code/model_trainers/sklearn_knn.py`)

For the K-nearest neighbor models, we only varied the value of `k`.
Spesifically we tried `1`, `3`, `5`, `7`, `11`, `17`, `19` and `23`. After `23` we noticed a worsening trend, so we stopped there.
To out surprise, `k=1` performed very well. It reached an accuracy of `94%`. (*If you look at out other runs, it usually lands between `94%` and `95%`*)
It usually spent abount `1.2ms` per prediction.

Here is the accuracy of the knn models:



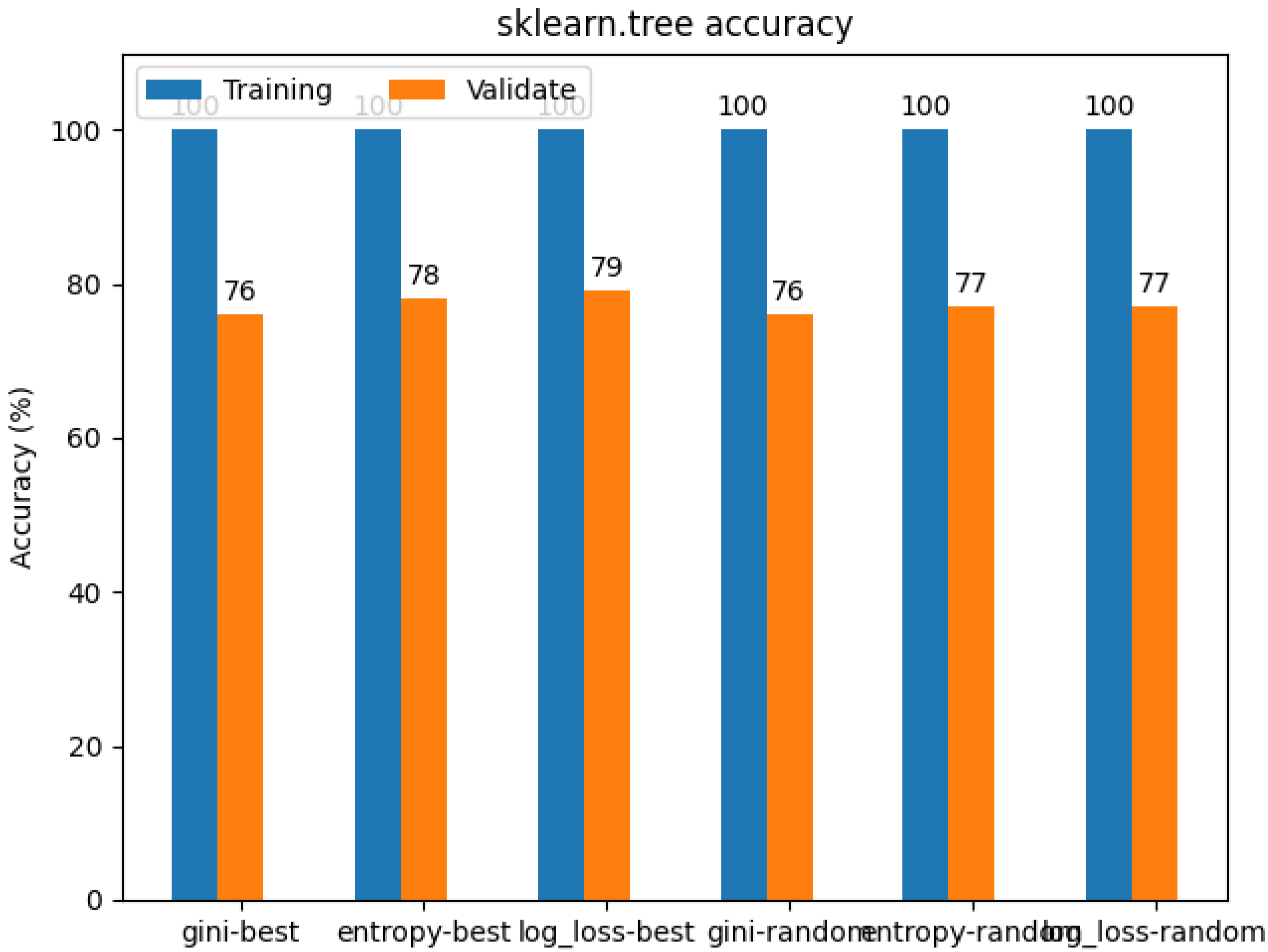
It also seems the knn models did not overfit due to the low difference between training and validation accuracy.

Decision Tree

(Code: `code/model_trainers/sklearn_tree.py`)

For the decition tree models, we varied the impurity measure (`gini`, `entropy` and `log_loss`) and feature selection (`random` and `best`).
We could probably have done more work here, but the models were so bad we decided to focus on other classifiers.
Although the performed badly, it at least did so very fast (Spending around `1000ns` or `0.001ms` per prediction).

Here is the accuracy of the `decition tree` models:



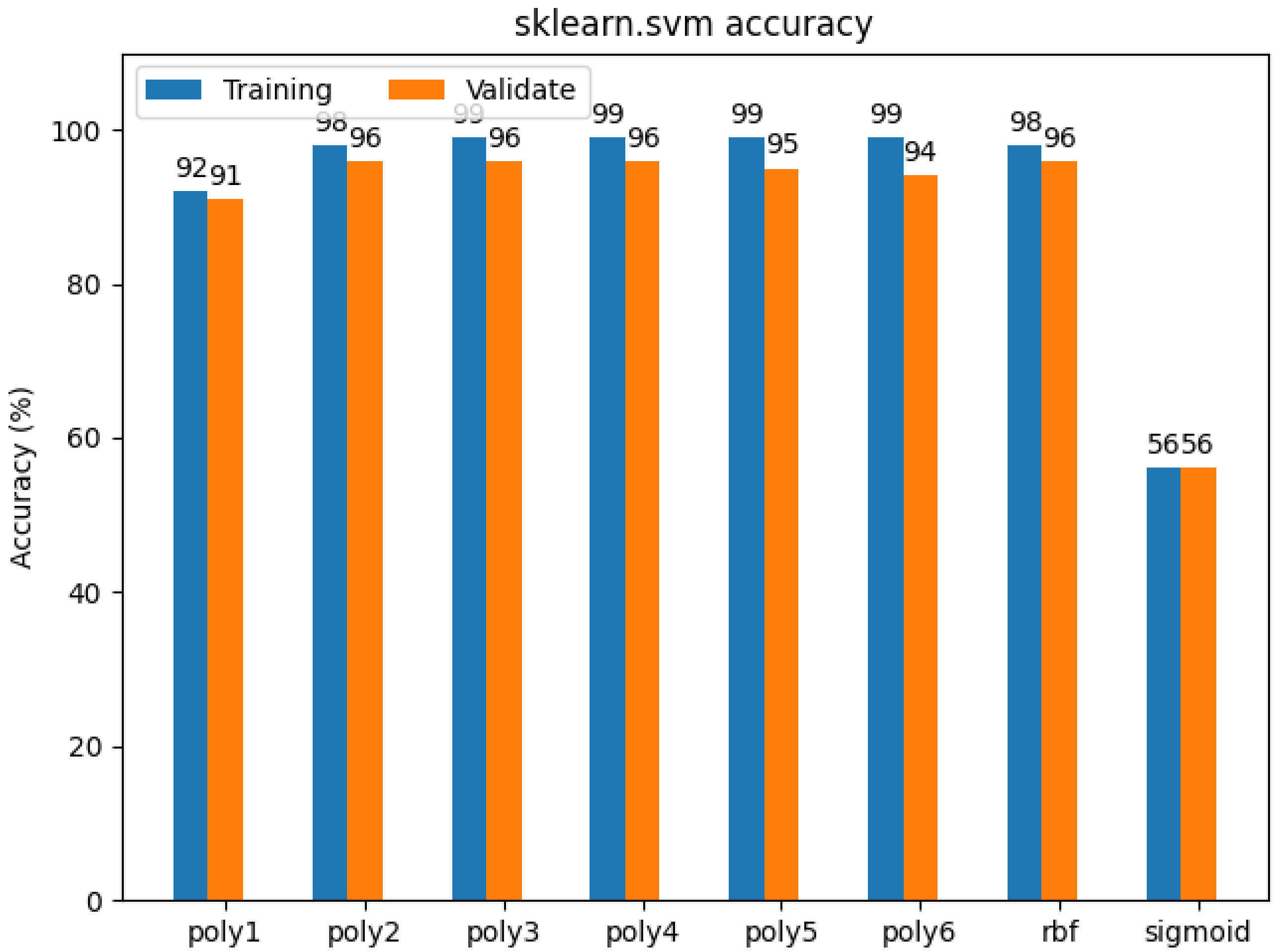
We originally planned to use our own decision tree classifier from `project 1`, but it was incredibly slow and very inaccurate (less than `30%` at best), so we decided to use the sklearn instead. As to why it was so much more inaccurate we aren't fully sure, but it could have to do with more labels in this dataset.

Support Vector Machine

(Code: `code/model_trainers/sklearn_svm.py`)

For the `svm` models, we decided to try different kernels (`poly`, `rbf` and `sigmoid`) and degrees (`1`-`6`).
The `linear` kernel was too slow in training to be included. We let it run for `12 hours` on a reasonably fast computer, but when it still would not finish, we decided to persue the other parameters instead.
In the end, we experienced the best accuracy when using an `svm` with kernel set to `poly` and degree set to `3`.
The accuracy was `96%` and it spent around `3ms` per prediction.
This ended up being our final classifier, and we go into more details in here.

Here is the accuracy of the `svm` models:

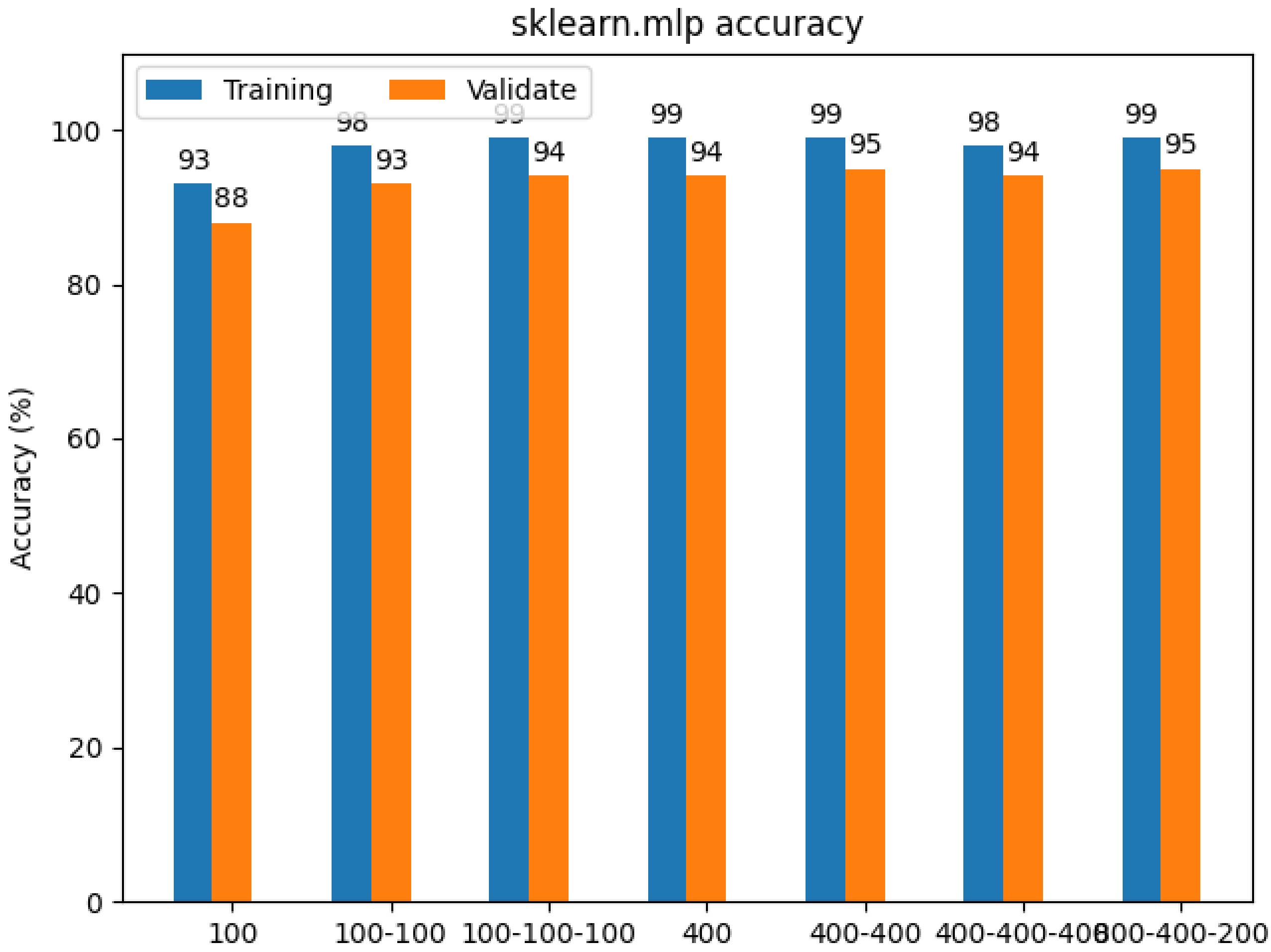


Multi-layer Perception

(Code: `code/model_trainers/sklearn_mlp.py`)

On multi-payer perception, we decided only to vary the size and number of hidden layers.
This was mostly because we did not understand the difference and consequence of the different activation functions.
The best `mlp` model reached an accuracy of `95%`. The timer per prediction (`TPP` in the logs) varied greatly by how many hidden layers there were, but always less than `20 000ns` or `0.02ms`.

Here is the accuracy of the `mlp` models:



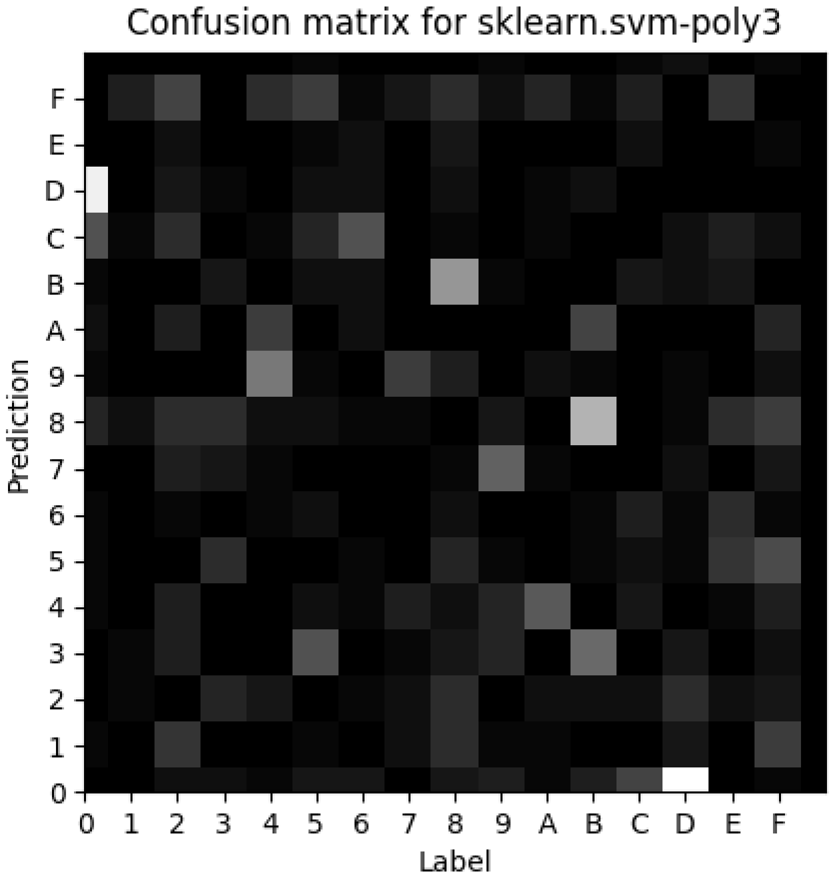
Left to right: `100`, `100-100`, `100-100-100`, `400`, `400-400`, `400-400-400` and `800-400-200`.

The model name indicate the size of the hiden layers.

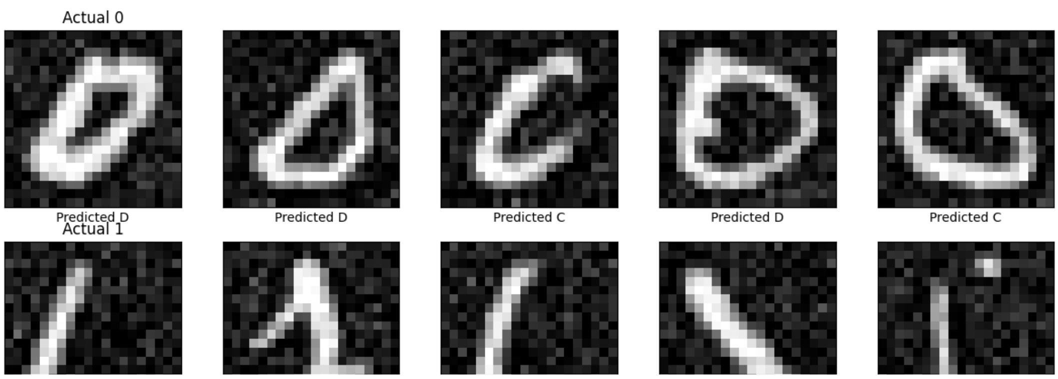
Final classifier

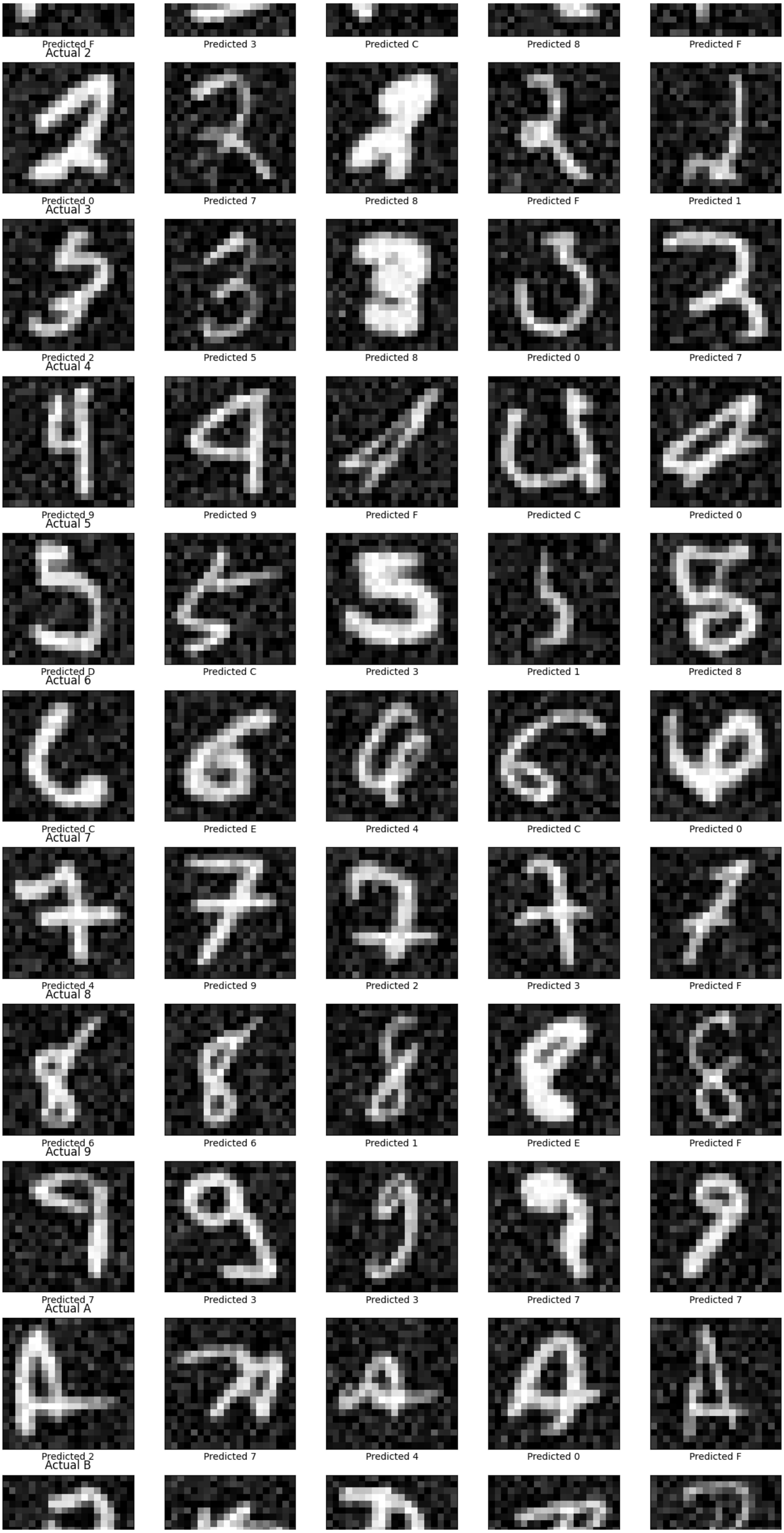
The final classifier was `sklearn.svm-poly3`.
An `svm` classifier from `sklearn` with a `poly` kernel of degree `3`. It reached an accuracy of `96%` and spends `3.2ms` per prediction.

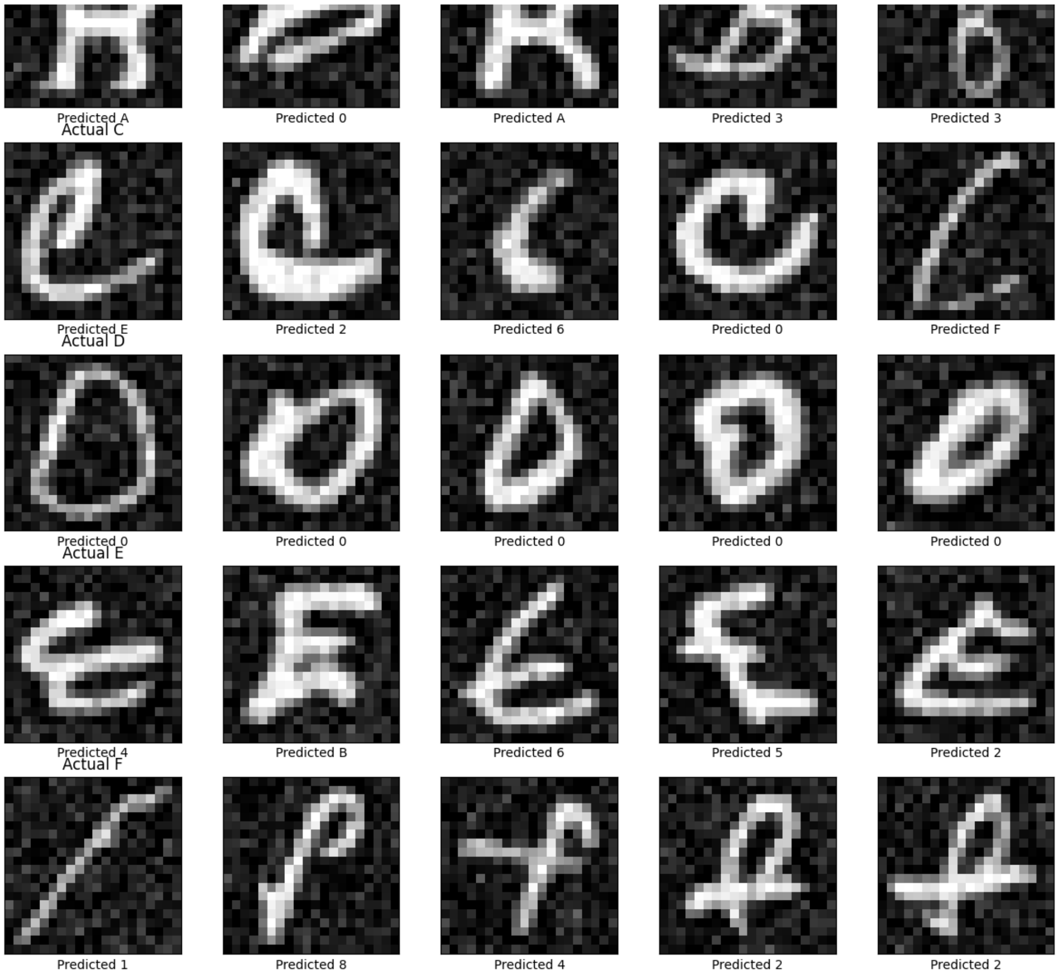
We decided to take a look at the `4%` that went wrong.
Here is the confusion matrix:



The bright squares indicate a higher error rate.
We initially expected the error rate for `E` to be higher due to its lower representation in the dataset, but it appears to be just fine.
The two primary confusions are `D`-`0` and `B`-`8`.
To understand these errors better, we compiled a list of 5 examples of each label that the model got wrong:




Predicted F
Predicted 3
Predicted C
Predicted 8
Predicted F
Actual 2
Predicted 0
Predicted 7
Predicted 8
Predicted F
Predicted 1
Actual 3
Predicted 2
Predicted 5
Predicted 8
Predicted 0
Predicted 7
Actual 4
Predicted 9
Predicted 9
Predicted F
Predicted C
Predicted 0
Actual 5
Predicted D
Predicted C
Predicted 3
Predicted 1
Predicted 8
Actual 6
Predicted C
Predicted E
Predicted 4
Predicted C
Predicted 0
Actual 7
Predicted 4
Predicted 9
Predicted 2
Predicted 3
Predicted F
Actual 8
Predicted 6
Predicted 6
Predicted 1
Predicted E
Predicted F
Actual 9
Predicted 7
Predicted 3
Predicted 3
Predicted 7
Predicted 7
Actual A
Predicted 2
Predicted 7
Predicted 4
Predicted 0
Predicted F
Actual B



We find some of these errors to be completely understandable (and we would have struggled to identify them ourself), and others to be not so much.

Future improvements

If given more time and resources we would like to look more into MLPs.
We would like to learn more about the different activation functions, how they work and their pros and cons.

We would also like to look into optimalization.
Currently we only utilize one thread, but the different trainers could absolutely run in parallell.

`sklearn` also has many tools and utilities that we have not taken advantage of.
If we were to continute this project, those helpers might be worth looking into.