Machine Learning Project

Quick, Draw! Recognition of drawings with different model types.

https://github.com/stefanpenzinger/ml-quick-draw

Contents

Goal

The aim of this project was to develop a system capable of recognizing drawings, especially those depicting specific animals. To achieve this goal, different types of models were used, and their respective accuracy was compared. Users were able to make sketches of animals on a canvas, which transforms the pictures in the necessary format. The system then analyses these drawings and makes predictions about the animals they represent.

Dataset

Google provides the data of the drawings, which were recorded in their game *Quick, Draw!* (https://quickdraw.withgoogle.com/). The dataset is available under the following link: https://github.com/googlecreativelab/quickdraw-dataset. The Quick Draw Dataset is a collection of 50 million drawings across 345 categories. The drawings were captured as timestamped vectors, tagged with metadata including what the player was asked to draw and in which country the player was located.

Google shares the following data sets:

- Raw Data
- Simplified Data (already pre-processed)
- Bitmap (simplified data converted to 28x28 greyscale bitmap=

Technology

The project was done with Python 3.10 and the Anaconda distribution. The following Python packages were used:

- NumPy
- Pandas
- Matplotlib
- scikit-learn
- TensorFlow
- Keras
  - high level API for TensorFlow
  - used for CNN
- ProtoBuf
  - used in TensorFlow
- Pillow
  - used for generating image form canvas

The packages were installed completely via a separate conda environment. The command `conda install` was used instead of `pip install` as pip did not work.

First Approach: using simplified vectors

`/src/archive`

The pre-processed simplified dataset was used for this approach: https://console.cloud.google.com/storage/browser/quickdraw_dataset/full/simplified

- simplified vectors
- removed timing information

- positioned and scaled the data into a 256x256 region
- exported in ndjson (same format and same metadata as the raw format)

The simplification process was:

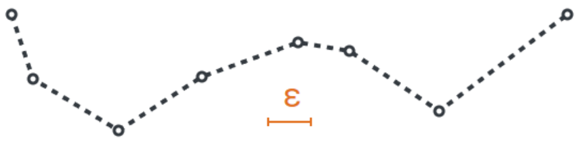
1. Align the drawing to the top-left corner, to have minimum values of 0.
2. Uniformly scale the drawing, to have a maximum value of 255.
3. Resample all strokes with a 1 pixel spacing.
4. Simplify all strokes using the Ramer–Douglas–Peucker algorithm with an epsilon value of 2.0.

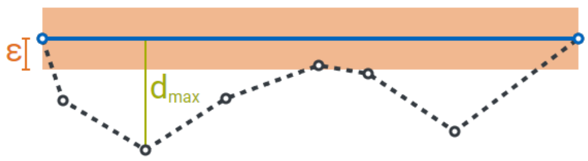
The intention of this approach was to differentiate musical instruments such as trombone, saxophone, piano and many others.

Ramer-Douglas-Peucker (RDP) algorithm

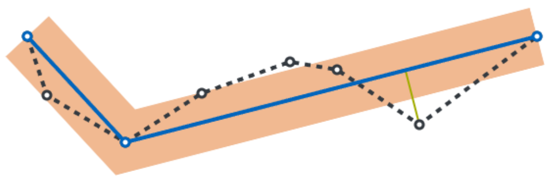
The RDP algorithm is used for the smoothing of polylines. An in-depth explanation was found at Cartography-Playground.

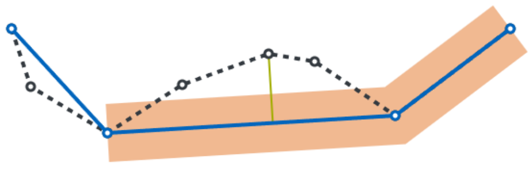
It smooths the line by reducing the number of points in a polyline, the rough shape of the curve should still be preserved. For smoothing the line, a single parameter ε which defines the maximum distance between the original points and the simplified curve is used.





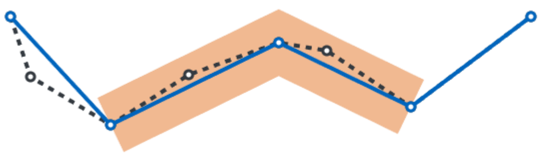
As seen in this example epsilon defines the maximum distance with the orange area. The variable *dmax* is the current furthest distance of a point to the original curve. If $d_{max} \leq$ ε the points are inside the tolerance and no further adaption is needed. As seen in the right curve above this not the case and the point can be removed, resulting in the left graph below.

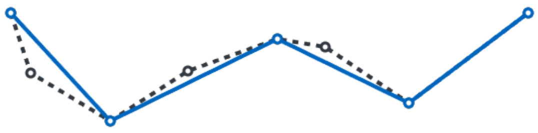




Now the next point with furthest distance to the curve is determined and again checked if $d_{max} \leq$ ε, again removing the point furthest resulting in the right curve above.

Lastly, all points are inside the defined are of ε and the curve is simplified for this ε as seen below.

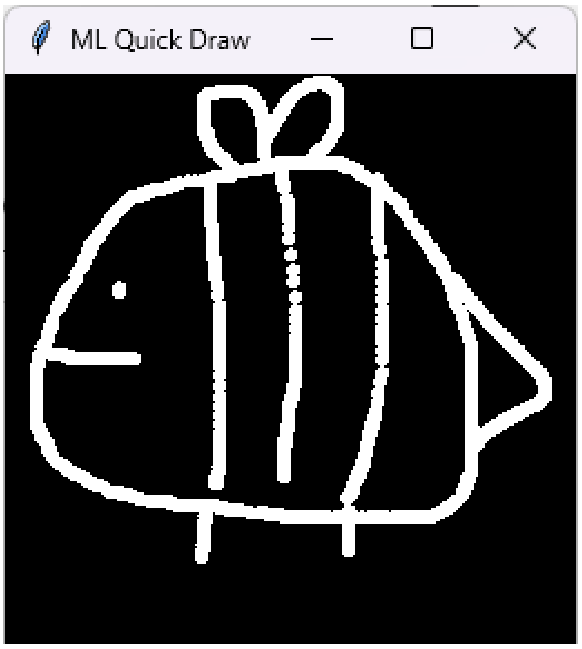




Example of the 1st approach

Drawing a bee on the canvas would lead to the following output as .ndjson

```
General:
drawing: {
   [
      [  // First stroke
         [x0, x1, x2, x3, ...],
         [y0, y1, y2, y3, ...]
      ],
      [  // Second stroke
         [x0, x1, x2, x3, ...],
         [y0, y1, y2, y3, ...]
      ],
   ... // Additional strokes
   ]
}
Example of stroke for the bee’s stinger:
drawing: {
   [ // stroke of bee’s body ],
   [
      200, 242, 242, 238, 222, 209
   ],
   [
      91, 138, 145, 149, 156, 167
   ],
   // further strokes
}
```



Pre-processing

Parsing

Due to the newline-delimited nature of the ndjson format, the file must be parsed line by line. Once parsed, the x and y coordinates are extracted and stored in separate columns within the data frame.

Feature Extraction

The following features were extracted from the Quick, Draw! dataset (partly mentioned in https://github.com/keisukeirie/quickdraw_prediction_model):

- Datapoint Count: The total number of datapoints in the drawing.
- Stroke Count: The total number of strokes used to create the drawing.
- y-Max: The highest y-coordinate reached in the drawing.
- Datapoint Percentage of Stroke 0: The percentage of datapoints that belong to stroke 0.
- Datapoint Percentage of Stroke 1: The percentage of datapoints that belong to stroke 1.
- Datapoint Percentage of Stroke 2: The percentage of datapoints that belong to stroke 2.
- Direction Stroke 1: The direction of stroke 1.
- Direction Stroke 2: The direction of stroke 2.
- x0: The x-coordinate of the starting point of the drawing.

Model Training

All models trained with the features mentioned were trained with cross validation with 10 partitions and a train-test split with 75%. In addition, several hyperparameters were used per model. KNN, Random Forest and Decision Tree were trained on the data and had an overall poor performance. This could be due to the fact that feature extraction requires many more features and the tuning of the hyperparameters to these features is also not efficient. Another reason is, that the preprocessing of data was not done well enough.

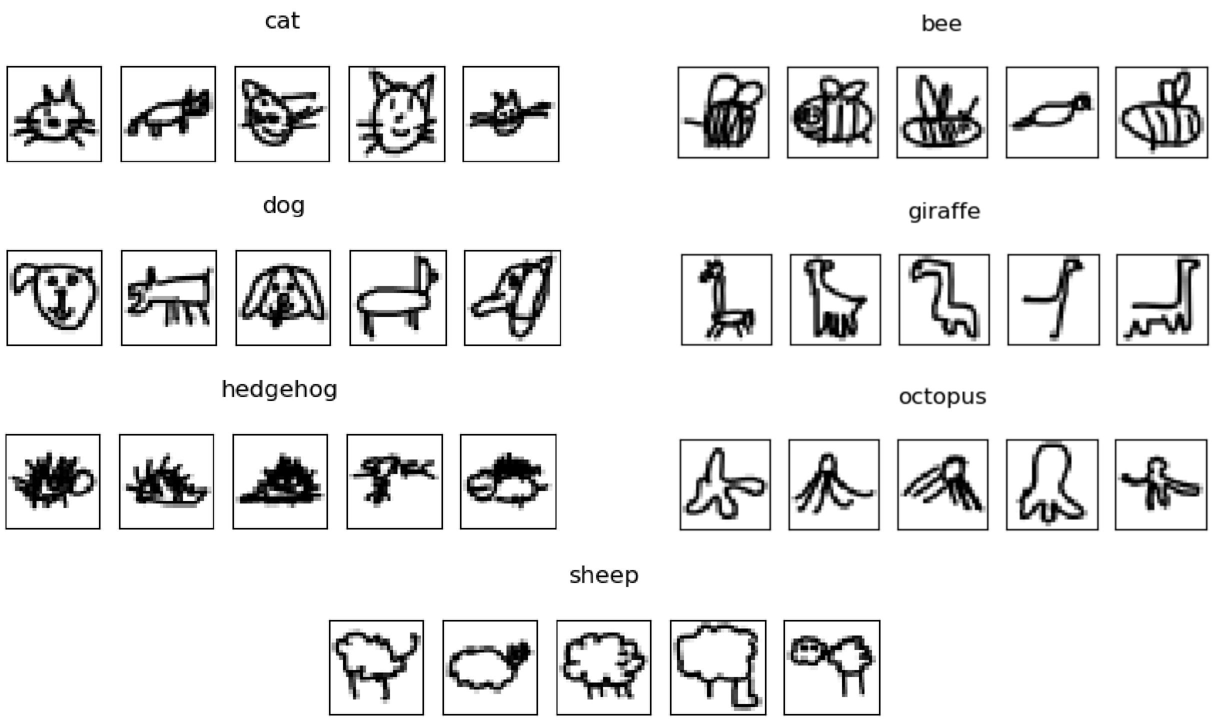
Second and working approach: using bitmaps

`/src/bitmap`

This method uses bitmaps rather than vectors. The simplified drawings were converted into 28x28 grayscale bitmaps and saved in numpy .npy format.
https://console.cloud.google.com/storage/browser/quickdraw_dataset/full/numpy_bitmap

This approach is partly based on https://github.com/kradolfer/quickdraw-image-recognition.

Unlike the first approach, different animals were used here: bee, cat, dog, giraffe, hedgehog, octopus and sheep. An extract of 25 of these animals was drawn to see what they look like:



Model training

The scaled bitmap values were used as raw features for model training. For all models except the convolutional neural network (CNN), the bitmap was flattened into a 784-dimensional vector (28 x 28 pixels). The CNN directly utilized the raster image format. Additionally, a dummy variable was appended to represent the corresponding label, assigning each class a unique numeric identifier.

To normalize the input data, the bitmaps were divided by 255 to scale their values between 0 and 1. A 50% train-test split was employed for model evaluation. Each model except the CNN underwent 10-fold cross-validation and grid search to optimize hyperparameter settings.

Furthermore, different amount of sample per animal class were examined. Starting with 2,000 samples per class, the models were trained with 5,000, 10,000, and 15,000 samples per class. 15,000 was the upper limit, as performance tended to plateau beyond this point and the increase in computing power did not need to be any greater.

For all model trainings, `n_jobs=-1` was used to utilize all cores of the CPU. This means that the computer is almost exclusively occupied with the training. Other programs do not work as well afterwards, so the training took place at night.

KNN (K nearest neighbour)

Grid Search

Number Neighbours: 1, 3, 5, 7, 9, 11

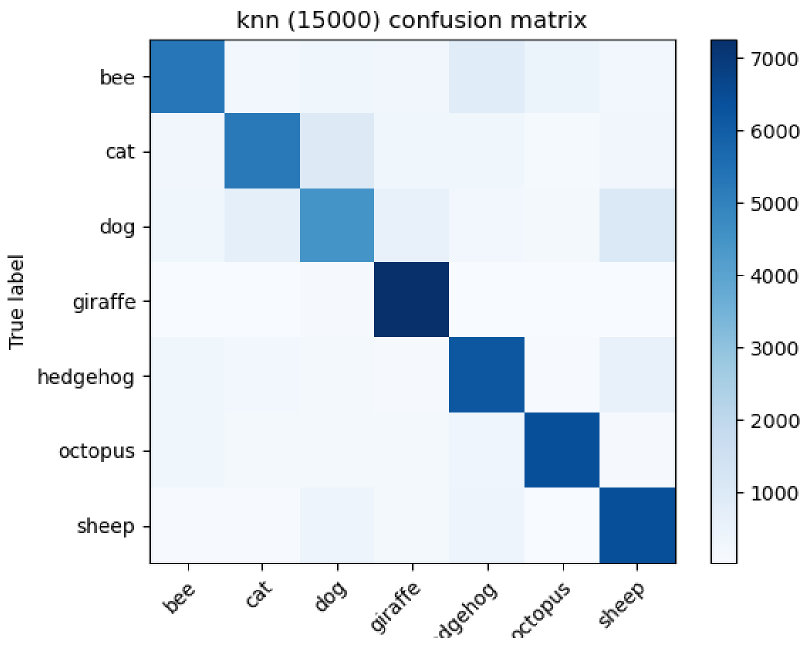
Training Results

Samples/Class	Best number of neighbours	Best training score
2,000	5	0.725
5,000	7	0.753
10,000	7	0.772
15,000	9	0.784

Test Results

Samples/Class	Test Score
2,000	0.730
5,000	0.755
10,000	0.776
15,000	0.786

Confusion Matrix with 15,000 samples per class



Random Forest

Grid Search

Number Estimators: 10, 20, 40, 60, 80, 100, 120, 140, 160

The number of estimators was not increased any further as it had reached a plateau.

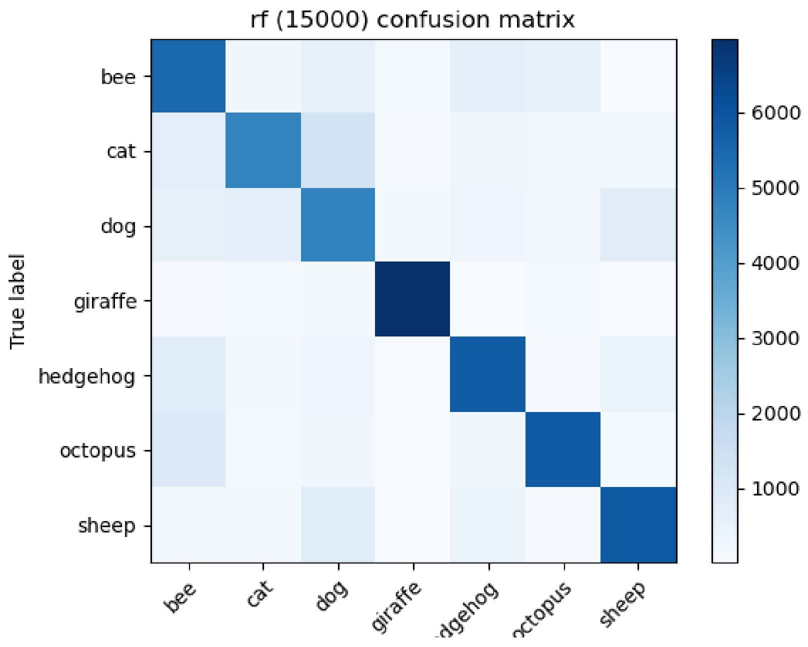
Training Results

Samples/Class	Best number of estimators	Best training score
2,000	160	0.704
5,000	160	0.722
10,000	160	0.740
15,000	160	0.748

Test Results

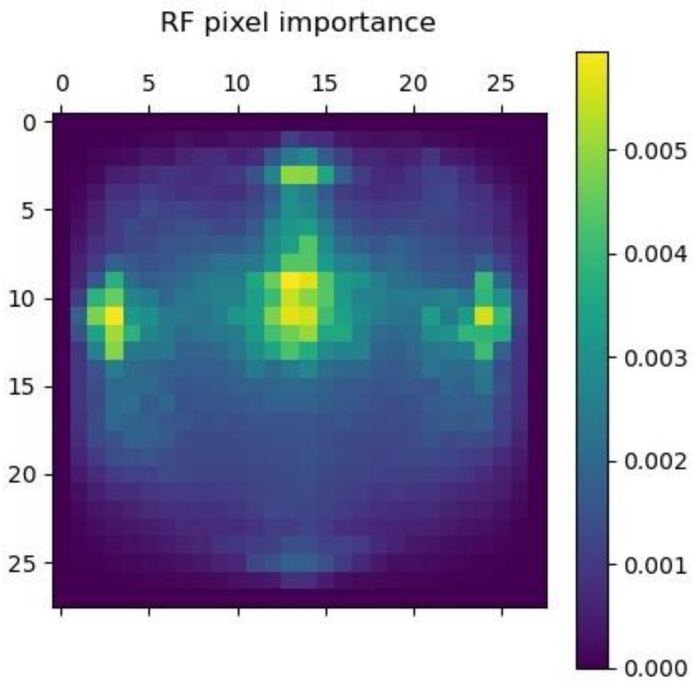
Samples/Class	Test Score
2,000	0.712
5,000	0.728
10,000	0.741
15,000	0.750

Confusion Matrix with 15,000 samples per class



Pixel Importance

As with tree-based models, it is possible to preserve the most important features (yellow) as they are located near the root. It can be seen that the pixels in the corners in particular are less important (dark blue/purple).



MLP (Multilayer Perceptron)

Grid Search

- Hidden Layer Sizes: (100), (100, 100)
- Alpha: [10^{-4}, 10^{-1}]

Since longer hidden layers have better accuracy but a long learning time, they were ignored as MLP does not seem to be very good for solving the problem.

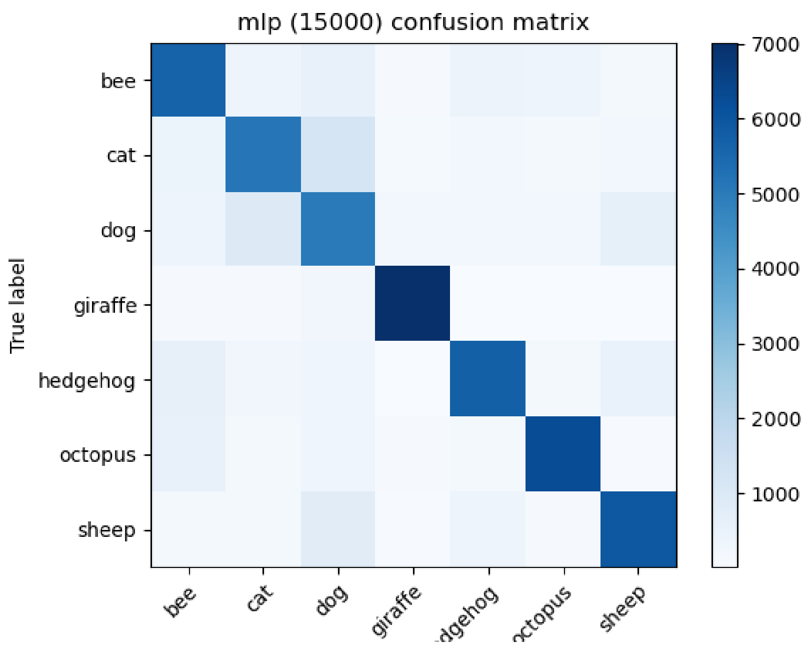
Training Results

Samples/Class	Best hidden layer sizes	Best alpha	Best training score
2,000	(100)	0.1	0.702
5,000	(100, 100)	0.01	0.734
10,000	(100, 100)	0.001	0.757
15,000	(100)	0.1	0.768

Test Results

Samples/Class	Test Score
2,000	0.720
5,000	0.737
10,000	0.757
15,000	0.776

Confusion Matrix with 15,000 samples per class



SVM (Support Vector Machine)

SVM was trained as well but not investigated further.

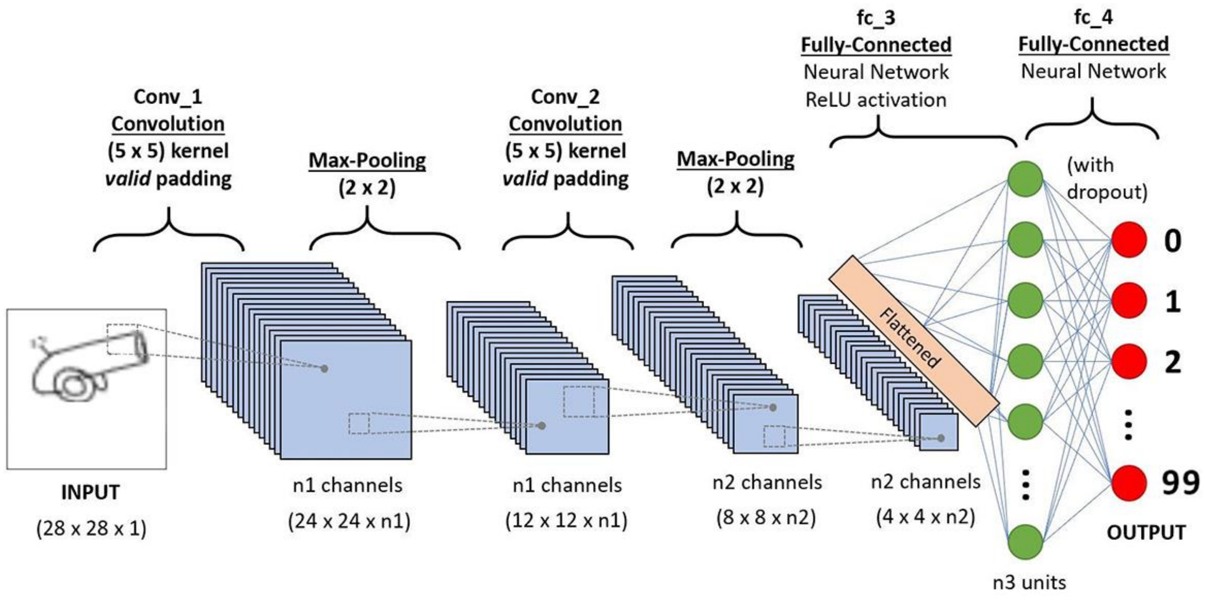
CNN (Convolutional Neural Network)

Unlike the other models, CNN retains a 28*28 input shape. It uses a batch size of 200, which means that the model processes 200 samples simultaneously during training. After completing the first batch of 200 samples, the model moves to the next batch to continue training. The number of training epochs is set to 10, as this is the point at which the model begins to show decreasing returns in performance improvement.

Attention: the trained model is uploaded to GitHub. During development, it was not possible to exchange the models between the laptops and CNN had to be trained again.

Layers

The layers of the CNN model are based on https://machinelearningmastery.com/handwritten-digit-recognition-using-convolutional-neural-networks-python-keras/.



1. Convolutional layer with 30 feature maps of size 5×5
2. Pooling layer taking the max over 2*2 patches
3. Convolutional layer with 15 feature maps of size 3×3
4. Pooling layer taking the max over 2*2 patches
5. Flatten layer
6. Fully connected layer with 128 neurons and rectifier activation
7. Fully connected layer with 50 neurons and rectifier activation
8. Output layer

Training Results

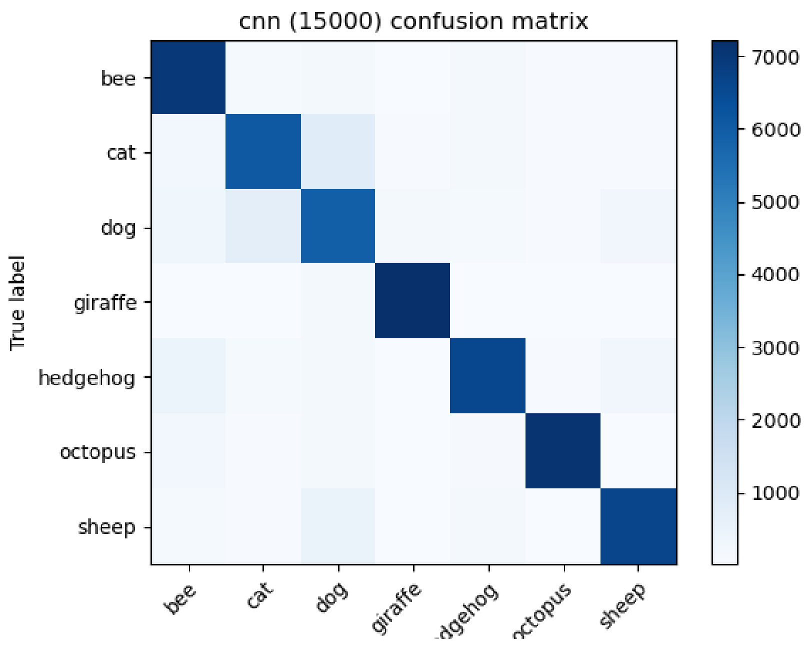
In CNN, it is possible to pass the held-back test set to the model. The model then returns the evaluation score for each epoch. Therefore, the training score is not displayed here.

Samples/Class (Time/Epoch)	Epoch 5 test acc.	Epoch 8 test acc.	Epoch 10 test acc.
2,000 (1s)	0.751	0.791	0.804
5,000 (2s)	0.822	0.840	0.847
10,000 (5s)	0.850	0.870	0.874
15,000 (9s)	0.867	0.880	0.885

Test Results

Samples/Class	Test Score
2,000	0.804
5,000	0.847
10,000	0.874
15,000	0.882

Confusion Matrix with 15,000 samples per class

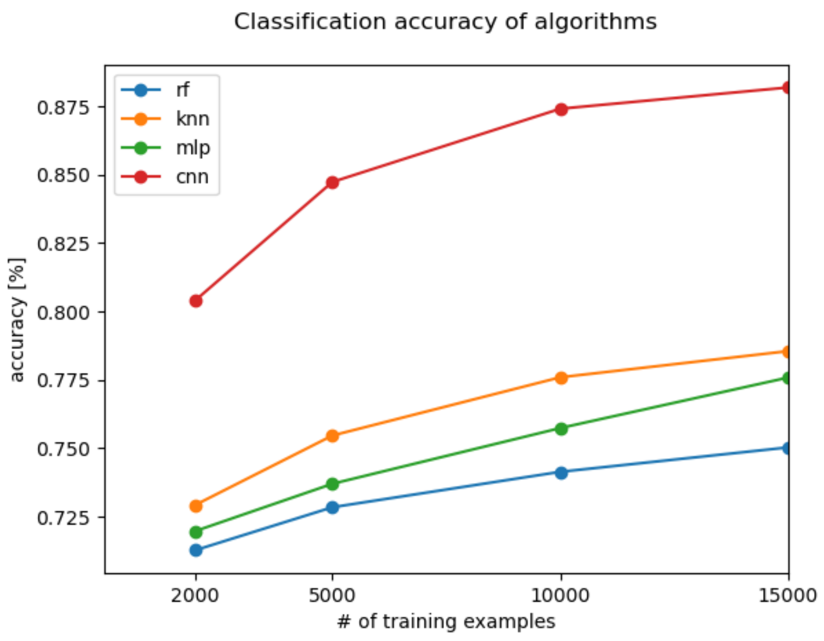


Model Evaluation

Each cross-validated and grid-searched model for all used sample sizes are temporarily stored by its model type in a dictionary. The training for all models took about 2 hours for the laptop used. After each model has been trained, the models are evaluated using the retained test set. The evaluation results are stored in a data frame for later visualization. For the best models per model type, the confusion matrix is recorded and then saved for further use.

Evaluation scores with held-back test set

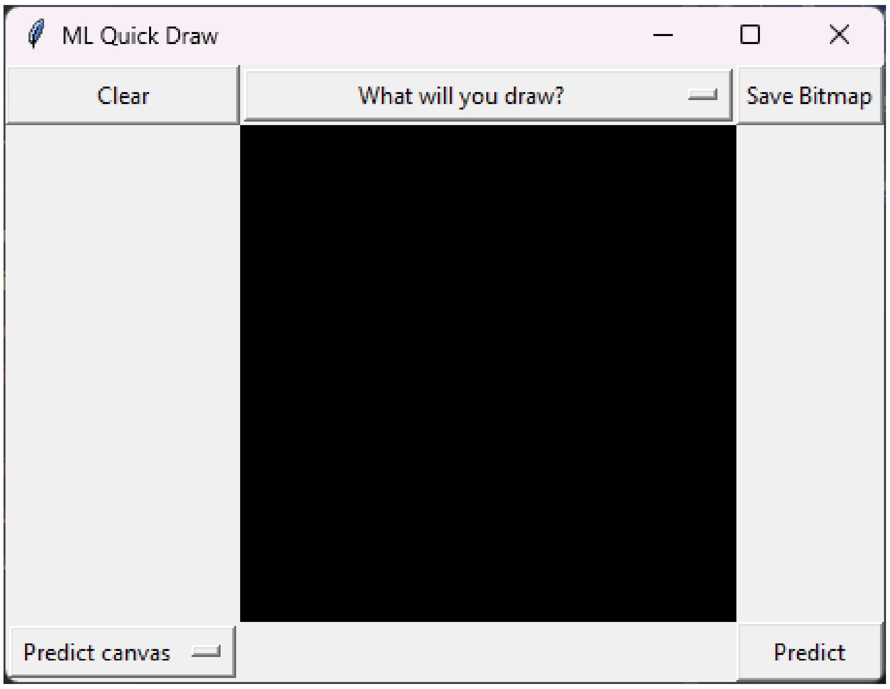
Among all model types, CNN significantly outperformed the rest. After reaching 15,000 samples, the evaluation results plateaued, leading us to restrict the sample size to this point. For instance, 120,890 samples are available for bee data, but the marginal improvement in evaluation results justified the smaller sample size.



Canvas

To generate data and test our model on data, which was created by users, we needed a canvas where we can draw the necessary pictures and save them in the right format.

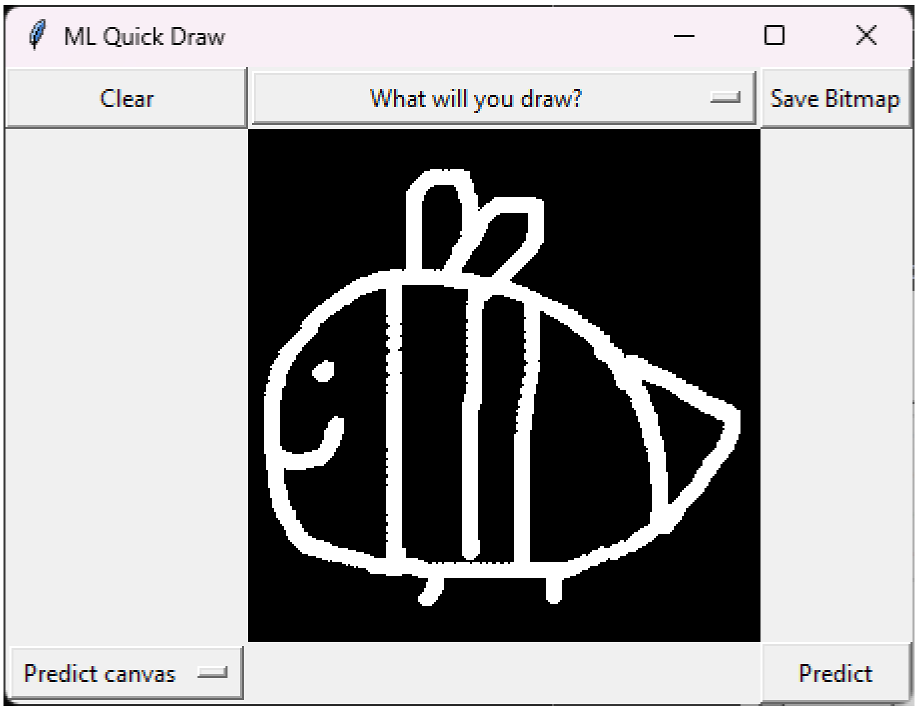
In the picture below we can see the canvas and its different functionalities.



Canvas: Drawing on the canvas

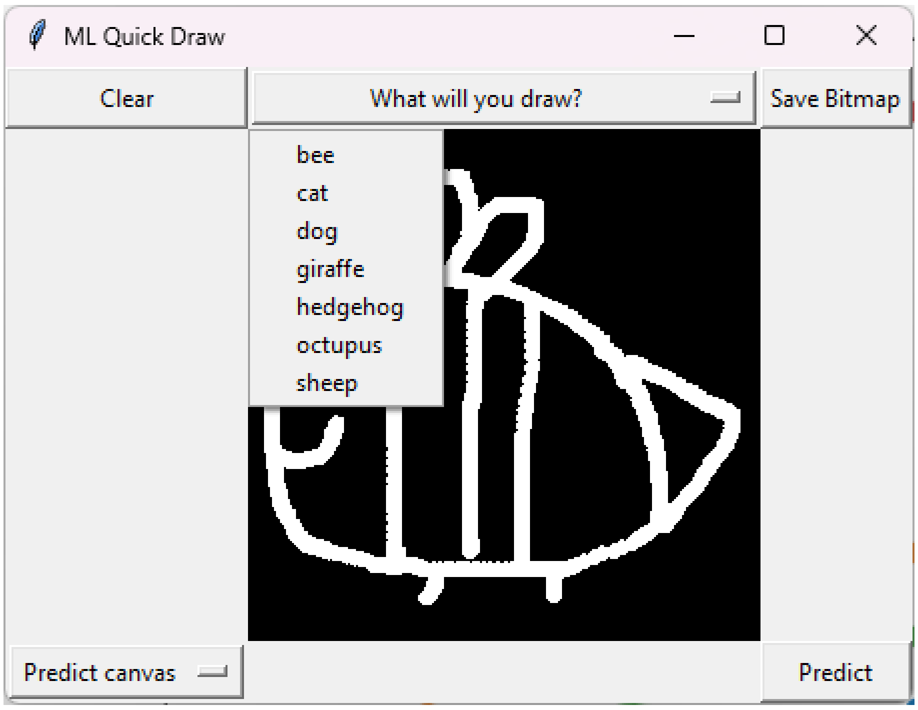
The black box is the canvas itself shaped by 256x256 pixels. This size was used so that we can transform it to 28x28 bitmap easier.

On the picture below a drawing of a bee can be seen.



Canvas: Save drawing

The picture can then be saved by choosing the kind of animal the user wanted to draw and then clicking on the “Save Bitmap” button as seen below.



When the user clicks on the button to save the image the following piece of code is being called:

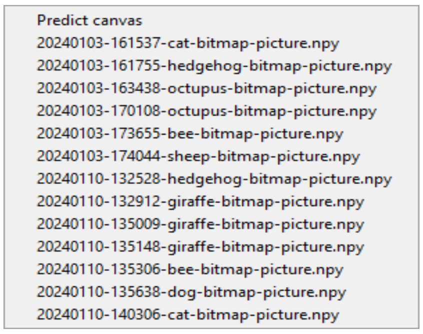
```
def __save_drawing_bitmap():
    drawing_name = clicked_example_item.get()

    if drawing_name == "What will you draw?":
        msg = messagebox.showerror(TITLE, "You need to specify a drawing!")
        return

    timestamp = datetime.now().strftime("%Y%m%d-%H%M%S")
    file_name = f"drawings/{timestamp}-{clicked_example_item.get()}-bitmap-
picture.npy"

    gray_scale_image_flattened = __create_bitmap()

    np.save(file_name, gray_scale_image_flattened)
    msg = messagebox.showinfo(TITLE, "Saved successfully!")


def __create_bitmap():
    canvas.update()
    image = ImageGrab.grab(bbox=(
        canvas.winfo_rootx(), canvas.winfo_rooty(), canvas.winfo_rootx() +
canvas.winfo_width(),
        canvas.winfo_rooty() + canvas.winfo_height(),))

    image.point(lambda x: int(x * 2.5))

    resized_image = image.resize((BITMAP_SIZE, BITMAP_SIZE), Image.LANCZOS)
    gray_scale_image = np.mean(np.array(resized_image), axis=2)
    return gray_scale_image.flatten()
```

1. First, the program checks if the user has opted in a drawing, he/she wants to draw.
2. Then the program updates the canvas, it refreshes it, ensuring that the all the data is available.
3. Then the image is grabbed from the canvas, specified by its height and width.
4. The value of the points is being increased by 2.5 to match the data provided by Google's "Quick, draw!" dataset.
5. Furthermore, the image is then resized to 28x28 pixels.
6. After that the image is turned into a greyscale image, to further match the data provided by Google.
7. Lastly, the image is flattened to a one dimensional vector (784,) and saved to the specified folder.

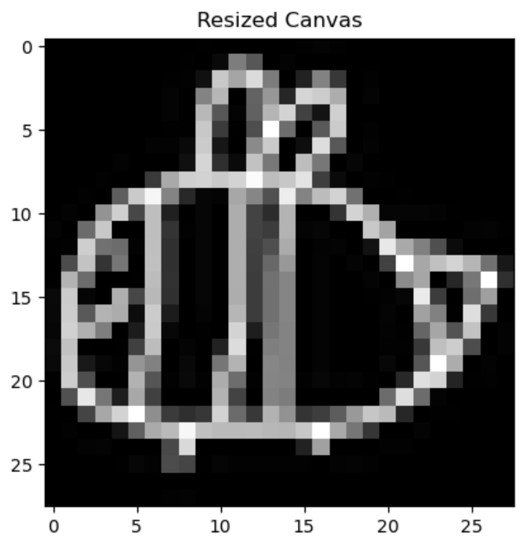
The image can be loaded later and predicted against the models created.

This is done via the dropdown menu in the bottom left corner.

Here are several files which can be used to see the performance of the models.


Predict canvas
20240103-161537-cat-bitmap-picture.npy
20240103-161755-hedgehog-bitmap-picture.npy
20240103-163438-octupus-bitmap-picture.npy
20240103-170108-octupus-bitmap-picture.npy
20240103-173655-bee-bitmap-picture.npy
20240103-174044-sheep-bitmap-picture.npy
20240110-132528-hedgehog-bitmap-picture.npy
20240110-132912-giraffe-bitmap-picture.npy
20240110-135009-giraffe-bitmap-picture.npy
20240110-135148-giraffe-bitmap-picture.npy
20240110-135306-bee-bitmap-picture.npy
20240110-135638-dog-bitmap-picture.npy
20240110-140306-cat-bitmap-picture.npy

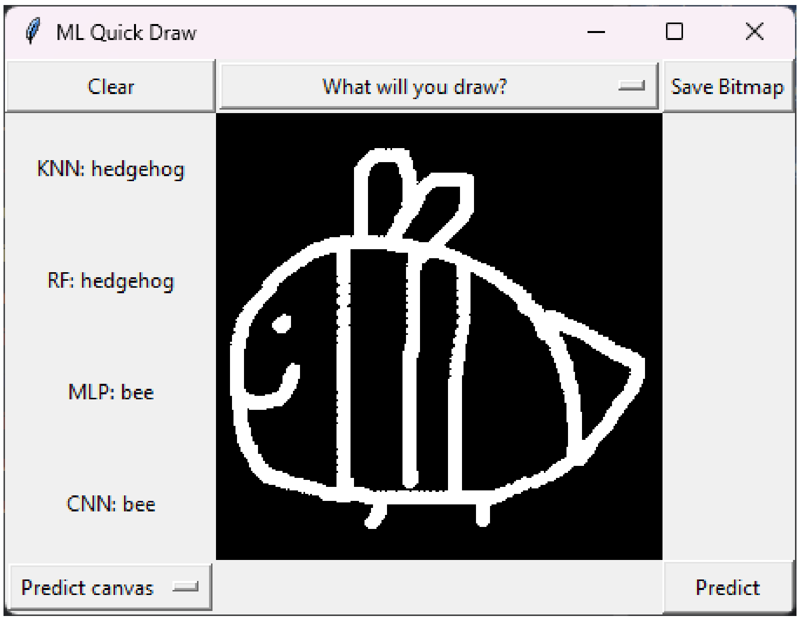
Example of saved bitmap:



Canvas: Predicting

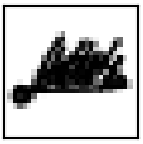
Instead of saving and loading the image, the user can directly predict against the picture drawn on the canvas. When the user is clicking on the predict button, "`__create_bitmap`" function, as seen in the code sample above, is called and predicted against the models directly.

On the left side of the screen, the results of the models are listed.



Conclusion

As seen in the results all model but CNN perform rather poor when working with Bitmaps. This is due to the fact that they only have access to the colour value of the array and try to map them with the most pixels that fit for a certain drawing. E.g. in the case of the random forest it creates a lot of if-else statements for each pixel, which are our features. KNN on the other hand looks at the pixels of each neighbour and determines on that what drawing it could be. For those two models the result was often a hedgehog. This could be because the drawings of the hedgehog are often wild scribbles which have a lot of density in the middle of the picture itself. Both KNN and Random Forest could be described as a dummy classifier because if the pixel density is high in this particular area, it will most likely always predict the hedgehog. MLP weights each pixel and creates the neural network out of it but it is not able to map the pixel very well in comparison to the CNN as it is missing the convolution of the pixels as CNN does.



If there are more drawings to classify, the models will lose performance. In the beginning, only two different animals were used, where the accuracy was much higher than it is now.

All in all, the results of this project are very good, especially when using CNN.

Improvements

The self-created bitmaps did not contain exactly the same information as the Google data set. There was a lack of precise scaling and centring of the image. Furthermore, the intensity of the strokes (intensity weight stored in the bitmap) may have been slightly incorrect.

The tuning of the hyperparameters could also be improved, but this was ignored because CNN was better than the others anyway. Another possibility would be to improve the feature extraction a bit and perhaps remove unimportant areas, as is evident in Random Forest.

Wild scribbles should be filtered out. Unfortunately, this was not possible because Google only had the labels for the correct classification inside the raw and simplified data. Perhaps it would be better to convert the data itself into a bitmap.

It would also be a step towards improvement to increase the number of samples per class, as only 15,000 samples were used out of about 120,000 samples (12.5%).