System setup

This document includes instructions for integration mathematical symbol prediction in Matistikk.

Overview

There are two projects needed in order to integrate the mathematical predictions into Matistikk. The predictor client and the predictor server.

The predictor client is a simple javascript module, which can be included as a script (see predictor server example), or as an npm module (the package has to be published first or downloaded locally).

The server is currently an independent python (Tornado) server. It can be used either as an external API (hosted on a different server than Matistikk), or by integrating the prediction code into an existing server.

Integrating the front end library

Running predictions first requires the input data to be on a specific format, as well as an implementation of drawing on a canvas. Therefore we have created a utility JavaScript module to handle the canvas events, and converting drawings to a format that the server expects.

A full example of using the front end library is found in the predictor server example.

Documentation of the client API can be found in the predictor client repository.

Using the server independent of Matistikk

The predictor server includes code to run the server independently. It is currently hosted on an Heroku server, with both the example implementation and the API. Symbol Predictor.

Running the server in your own enviroment is described in the predictor server README

Integrating the server to Matistikk

If you wish to host the server using a different library than Tornado, the best approach will be to copy the relevant code, and write your own integration.

All relevant code for prediction is available in the classification folder of predictor server. This is the code you will need to copy into your own project.

An example of interacting with the classification folder can be found in server.py.

In order to use the classification code in Django, you will have to write a POST handler to extract the relevant buffer from the POST request, and run prediction using the components in the classification folder. This can be replicated from the mentioned server.py file.

symbol-predictor-server

This a server including a simple API to run prediction on mathematical symbols and expressions.

Prerequisites

1. Python 3
2. Pip (to your python 3 version)

Running the server

Do the following steps to run the server:

1. Clone the repository (`git clone ...`)
2. Install the dependencies (`pip install -r /path/to/requirements.txt`)
3. Download the combined_model.
4. Add the model to the folder `/classification/`
5. Run the server (`python /path/to/server.py`)

Running the tests

To run the tests run the following command from the root directory:

```
python -m unittest discover test/
```

API

The server has a single endpoint, a POST handler the on /api endpoint

```
POST /api
```

Input

The endpoint expects data on the format application/JSON.

The request's body should be on the format:

```
Coordinates2D = {x: number, y: number}

Trace = Array<Coordinates2D>

{
    "buffer": "Array<Trace>"
}
```

Output

The output from the endpoint includes:

1. The prediction converted to LaTeX.
2. A list of all the symbols segmented from the traces.
3. A list of the top ten probabilities for each segmented symbol

The response body will be on the format:

```
TraceGroup = Array<number> // List of indexes which combined creates a
symbol (indexes from the "buffer" in input).

Probability = number // Number between 0 and 1.
Probabilities = Array<Probability> // List of top 10 propabilities.

Label = string // A latex representation of a single symbol.
Labels = Array<Label> // List of top 10 labels (corresponds with
Probabilities).

{
    "latex": string, // The full expression in latex
    "probabilities": {
        "tracegroup": Array<TraceGroup>, // trace indexes combined into
symbols
        "labels": Array<Labels>, // The labels corresponding with each
probability.
        "values": Array<Probabilities> // The probabilities with length
equal to the number of predicted symbols
    }
}
```

Example

```
request = {
    "url": "/api",
    "method": "POST",
    "body": {
        buffer: [
            [
                {x: 0, y: 1},
                {x: 1, y: 2}
            ],
            [
                {x: 4, y: 8},
                {x: 5, y: 9}
            ]
        ]
    }
}

response = {
    "body": {
        "latex": "\sqrt{3}",
```

```
        "probabilities": {
            "tracegroup": [
                [0],
                [1]
            ],
            "labels": [
                ["\sqrt", "\alpha", "y", ...],
                ["3", "9", "\beta", ...]
            ],
            "values": [
                [0.7, 0.2, 0.05, ...],
                [0.9, 0.03, 0.01, ...]
            ]
        }
    }
}
```

PredictorClient

Predictor client is a frontend library intended to handle common canvas use cases.

Symbol Canvas

The library includes a SymbolCanvas class, which handles most of the communication with the canvas. Its features are:

1. Drawing lines between coordinates.
2. Drawing filled circles (for erasing)
3. Converting touch - and mouse events to normalized coordinates.
4. Clearing the canvas.

API

The SymbolCanvas class exposes four public methods, as well as three events.

Constructor

```
 /**
  * @param   element The element drawing should be made on
  *
  * @returns {void}
  */
  constructor(element: HTMLCanvasElement) : void
```

Draw line

```
/**
 * @param fromCoords   The coordinates to draw from
 * @param toCoords     The coordinates to draw to
 * @param color        The color to draw the line, defaults to #A0A3A6
 * @param lineWidth    The width of the line, default to 5
 *
 * @returns {void}
 */
drawLine(fromCoords: Coordinates2D, toCoords: Coordinates2D, color =
"#A0A3A6", lineWidth = 5) : void
```

Draw circle

```
/**
 * @param coords     The coordinates in the center of the circle
 * @param radius     The radius of the circle, default to 10
```

```
 * @param fillStyle  The color to fill the circle, defaults to white
 *
 * @returns {void}
 */
drawCircle(coords: Coordinates2D, radius = 10, fillStyle = "white") : void
```

Clear canvas

```
/**
 * @description Removes all color and drawings from the canvas.
 */
clearCanvas() : void
```

Event emitters

```
/**
 * @name        click
 * @description Emits when the canvas was clicked.
 *
 * @param       {Coordinates2D} clickedCoords The coordinates where the
canvas was clicked
 *
 *
 * @name        draw
 * @description Emits when a user has drawn one step forward.

 * @param {Coordinates2D} currentCoords The coordinates where the user drew
to
 * @param {Coordinates2D} prevCoords    The coordinates the user drew from.
 *                                      Can be null if there are no
previous coords.
 *
 *
 * @name           release
 * @description    Emits when a user has released the canvas. If the user
did not draw. Click will be dispatched
 *                 instead of release.
*/
```

Canvas controller

The canvas controller class handles the communication between the canvas and your application, as well as common utility operations. Its features are:

1. Keeps a datastructure of coordinates in the format the server expects.
2. Validates touch events, in the case that a user clicked unintentionally.

3. Emits an event with the current trace buffer when a user is done drawing (on release if there is a valid buffer). This can be sent directly to the server.
4. Emits an event with the clicked trace if a user presses the canvas within a certain distance from a trace.
5. Can mark specific traces with a specified color (used for marking a group of traces)

API

The CanvasController class exposes four public methods, as well as three events.

Constructor

```
/**
 * @param canvas                                A canvas with a similar
interface to SymbolCanvas
 * @param {Object}   options                    Options for how the
controller should work.
 * @param {bool}     options.isErasing          Whether buffer should be
removed or drawn.
 *                                              Can be accessed from
symbolCanvas.options.isErasing
 * @param {number}   options.eraseRadius        Number for the circle
radius when erasing
 * @param {number}   options.minTraceCount      How many samplings in a
trace before it is valid
 * @param {number}   options.minTraceDistance   The minimum distance a
trace can be to be valid
 * @param {string}   options.canvasColor        The color of the canvas
 * @param {string}   options.canvasSelectedColor The color a selected
symbol should have
 * @param {number}   options.strokeWidth        The width of each stroke
 * @param {string}   options.strokeColor        The color of the drawn
lines
 * @param {bool}     options.validateTrace      If traces should be
validated before being included in buffer.
 *                                              If false, minTraceCount
and minTraceDistance will not be used.
 *
 * @returns {void}
 */
constructor(canvas: SymbolCanvas, options: ControllerOptions) : void
```

Redraw buffer

```
/**
 * @description Clears the canvas and redraws the buffer.
 *
 * @returns {void}
 */
drawLine() : void
```

Mark trace groups

```
/**
 * @description Draws traces in buffer at specified indexes. Uses default
values from options.
 *
 * @param traceGroupIndexes: The indexes for traces in buffer that should
be drawn
 * @param color:             The color of the drawn traces. Defaults to
value in options
 * @param strokeWidth:       The width of the drawn traces. Defaults to
value in options
 *
 * @returns {void}
 */
markTraceGroups(traceGroupIndexes: number[], color, strokeWidth): void
```

Event emitters

```
/**
 * @name               release
 * @description        Emits when the canvas was released, and at least one
valid trace has been drawn.
 *
 * @param {number[][]} buffer The current buffer at the time of release.
Includes a list of traces.
 *
 *
 * @name               symbolclick
 * @description        Emits when the canvas was clicked, and a trace was
within range of the click.

 * @param {number}     overlappingIndex The index of the closest trace to
the click.

*/
```

Example

```
<!DOCTYPE html>
<html>
<body>

 <canvas id="canvas"></canvas>
```

```
</body>
</html>
```

```
const canvas = document.getElementById("canvas")
const symbolCanvas = new SymbolCanvas(canvas);
const canvasController = new CanvasController(symbolCanvas);

canvasController.on("release", buffer => {
    // Send buffer to server
})
canvasController.on("symbolclick", traceIndex => {
  // Find which trace group the index is in
  const traceGroup = findTraceGroupByIndex(traceIndex) // Returns list of
the traces' indexes
  canvasController.markTraceGroup(traceGroup, "red") // Paint the symbol
(trace group) red.
})
```

A full example can be found in symbol predictor server.

Machine learning

This repository is used to experiment and train machine learning models for predictions. The repository also includes methods for processing InkML files into normalized traces and images.

Creating dataset and running the project

Prerequesites

- Python 3.5
- Pip (To python 3.5)

In order to run the code in this repository, you will need either InkML files or an already generated dataset. There is also a complete folder with both xml and preprocessed data available. If this file is chosen, just extract all the files within the `/online_recog/data` folder.

First clone the repository and run `pip install -r requirements.txt`.

From InkML files

If you do not have access to the preprocessed dataset. You must first run the preprocessing on InkML files.

1. Download the InkML folder
2. Create folders such that the folder structure in online_recog is: `/online_recog/data/xml/`
3. Unzip the InkML files downloaded, and paste them into the xml folder.
4. Open `/online_recog/keras_lstm.py` in a text editor.
5. Uncomment the line `generate_and_save_dataset()`
6. Run `python keras_lstm.py`

From already preprocessed

If you have downloaded the preprocessed data. You will just have to put the data in the correct folder.

1. Download the folder data.zip
2. Create the folder `/online_recog/data`.
3. Extract the zipped files into the newly created `/data` folder.

Running the training

After placing the datasets into the correct folder, the models can be ran by doing the following.

1. Uncomment`load_dataset_and_run_model()` in `/online_recog/keras_lstm_py`.
2. Choose the model you wish to train by changing `run_RNN_model()` to the method with the model you wish to train.
3. Run `python keras_lstm.py`.

Including the real data.

In order to include real data for validation during training. You will either need to create data from symbol-predictor-server, or download the already processed files (the real data is included).

In `/online_recog/keras_lstm.py`, there is a callback class which can be used to run validation and store logs from the training. This class, as well as a couple other places in `keras_lstm.py` has commented out lines that can be uncommented if real data is available in `/online_recog/data`. The regarding lines is shown in the `keras_lstm.py` file.

Python packages

- Requirements