KOÇ UNIVERSITY INTELLIGENT USER INTERFACES LABORATORY

SUMMER RESEARCH PROJECT REPORT

Tevfik Metin Sezgin, Ozan Can Altıok, Koç University Intelligent User Interfaces Laboratory
mtsezgin@ku.edu.tr, oaltiok15@ku.edu.tr

Ahmet Bağlan, Boğaziçi University, ahmet.baglan@boun.edu.tr
Amirhossein Sayyafan, Sharif University of Technology, sayyafan@ee.sharif.edu
Arda İçmez, Galatasaray University, aicmez@gmail.com
Elif Yağmur Eyrice, Semih Günel, Tuğrulcan Elmas, Bilkent University
yagmureyrice@gmail.com, semih.gunel@ug.bilkent.edu.tr, tugrulcanelmas95@gmail.com

1 Project Description

Video Browser Showdown [1] is an annual competition among video retrieval engines. In the competition, the participants are asked to find several videos by giving queries to their own video search engine. The team that has found the given videos most accurately becomes the winner. iMotion [2] is one of the video retrieval engines that have been in the competition for years.

As a contributor to this project, in order to find the videos faster and trying more queries out in a limited time, we noticed that we need to reduce the time for giving a single query to the system. As the queries are given as sketches, we decided on the integration of the sketch auto-completion system, which enables the classification of the partially-sketched symbols and thus reduces the time spent for drawing sketches. Since this work [3] was done by a Ph.D student of our lab, we had the code and the integration was expected to take a little time. Nevertheless, the training of the system on a large dataset of sketches took a long time. The constrained K-means clustering task of the training couldn't finish even in a week. By digging into the source code of the framework, we identified 2 reasons for this problem:

- **Poor runtime performance of MATLAB** [4]**:** The source code of the framework is available only in MATLAB. Our calculations showed that the clustering operation whose original code is optimized would still have taken about 30 days on the high performance clusters of the university. This implies that MATLAB has a poor runtime performance.
- **O(N^2) performance of the constrained K-means algorithm** [5]**:** The constrained K-means algorithm assigns the clusters by checking given background knowledge. If we have *N* instances to get clustered, for every instance, the algorithm checks the constraint relations (must-link or cannot link) to all of the *N* instances, therefore the running time of the algorithm is $O(N^2)$, which is inefficient if we have a huge number of instances (i.e., *N* is large). In the project, we planned to train the framework on Eitz [6] dataset, a dataset of everyday object sketches. Due to the fact that the number of instances in the dataset extended by the partial sketches is about 360.000, the runtime performance gets considerably slow.

In this project, we asked the students to implement the auto-completion system in Python, which has a better runtime performance than MATLAB and which is suitable for computational tasks. Moreover, since they completed the implementation of the framework as it is in the paper earlier, we asked them to do some optimization on the code in order to make constrained K-means operation faster. The running time of the entire training algorithm on this dataset has been reduced from ~60 days to an hour (see figures *1 and 2* for details). The resulting system was then integrated to iMotion system. Furthermore, for the demonstration of the system in real time, some students worked on an Android UI and its server back-end.

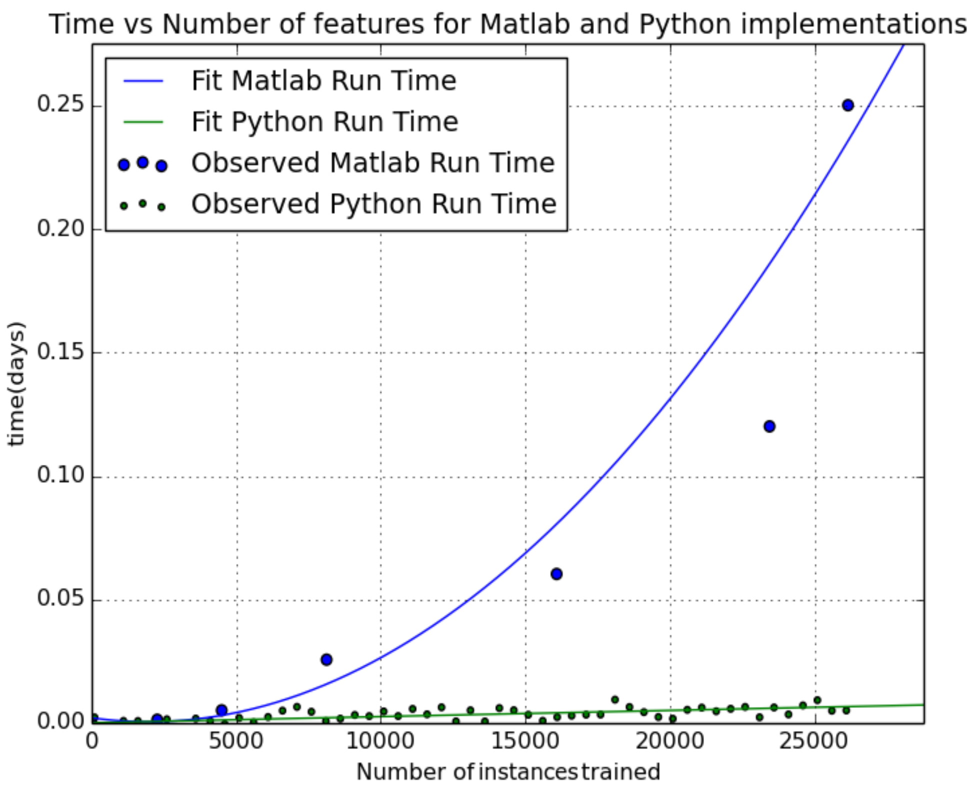


Figure 1: The runtime performance of the training algorithm on NicIcon [7] dataset. The runtime of the original algorithm increases parabolic-ally, whereas the runtime of our algorithm increases linearly.

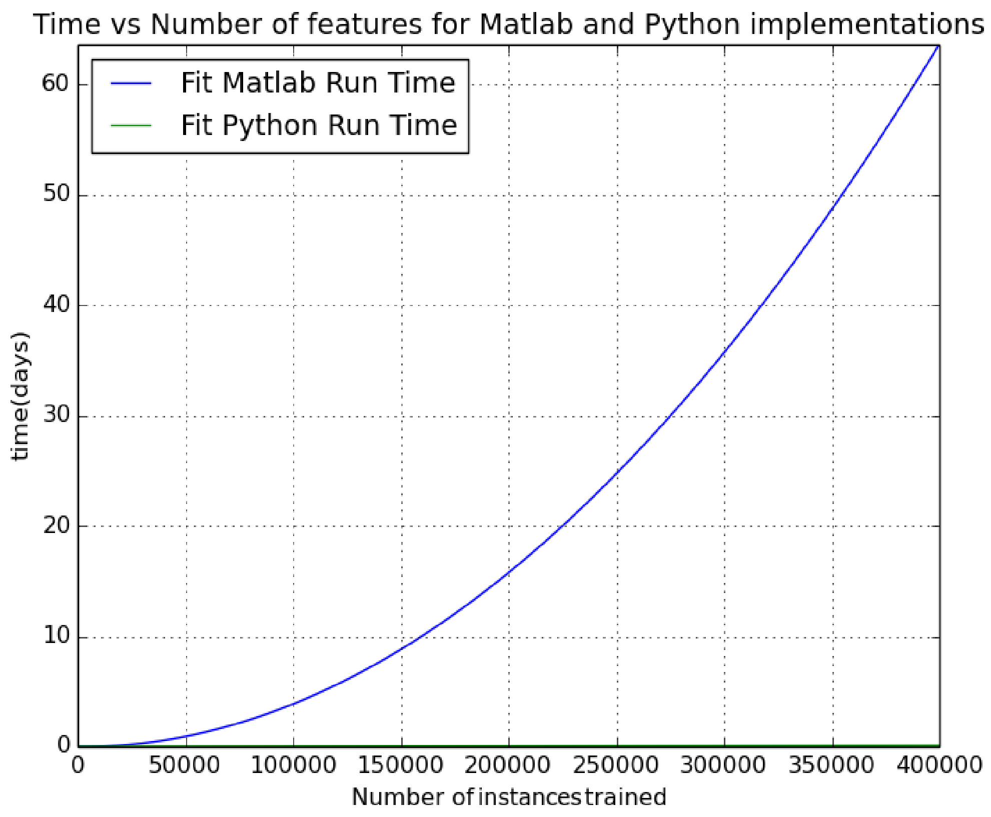


Figure 2: The runtime estimations on a larger interval of instance count, using the curves fit in Figure 2.. Once the partial sketches are created, the number of instances in Eitz dataset raises up to 360.000. The original pipeline takes about 60 days, whereas our implementation takes less then a day (~an hour).

2 Distribution of Work

Here is the pipeline of the framework, (a) is for training and (b) is for testing.

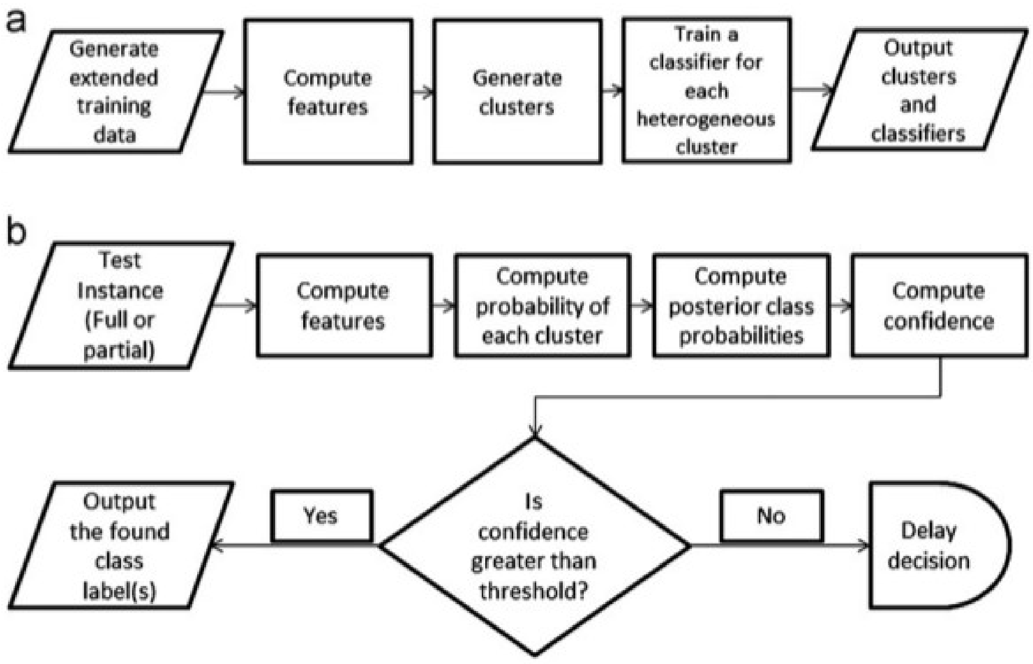

a
Generate extended training data
Compute features
Generate clusters
Train a classifier for each heterogeneous cluster
Output clusters and classifiers
b
Test Instance (Full or partial)
Compute features
Compute probability of each cluster
Compute posterior class probabilities
Compute confidence
Is confidence greater than threshold?
Yes
Output the found class label(s)
No
Delay decision

Figure 3: The pipeline of the sketched symbol auto-completion framework

As mentioned in *Introduction*, some students in our team also implemented an Android UI and a server back-end for demonstration, and integrated the resulting auto-completion system into iMotion system. The distribution of these parts is given below.

Task	**Team Members**
Extension of the datasets	Arda, Semih
Feature extraction (debugging)	Amir
Constrained K-means implementation	Arda, Amir, Semih, Tuğrulcan
SVM training on clusters	Ahmet, Arda
Probability calculation	Ahmet, Arda, Semih
Saving trained model on disk	Ahmet, Arda, Semih
Alternative pipeline	Ahmet
Predictor implementation	Ahmet, Arda, Semih
Accuracy testing	Ahmet, Semih
Android UI	Amir, Yağmur
HTTP Server for Android UI	Tuğrulcan, Yağmur, Amir
iMotion integration	Tuğrulcan

Table 1: Work distribution among team members

3 Work Done

3.1 Model Training

3.1.1 Dataset Extension

Eitz and Niclcon datasets contain human sketches of various objects and are commonly used to evaluate the performance of sketch recognition systems. They are available in various formats, mainly in XML or MATLAB variable files (*.mat*). In both datasets, sketches are stored stroke by stroke. In order to use these datasets in this project, we first had to transform them into JSON format. While doing this transformation, we also had to create partial sketches of every full sketch in these datasets, because the main goal of this project is to give suggestions to partial drawings. As we had every individual stroke of every sketch in the datasets, creating partial sketch was a matter of approach. We decided that it would be a good start to take the first stroke of sketch as the first partial sketch, and develop upon that with other strokes in an incremental order. We have written a script in MATLAB which would serve to this purpose. To make sure we are in the right track, we also contacted with the authors of the original paper, both of whom also agreed they also used the same approach while creating the partial sketches.

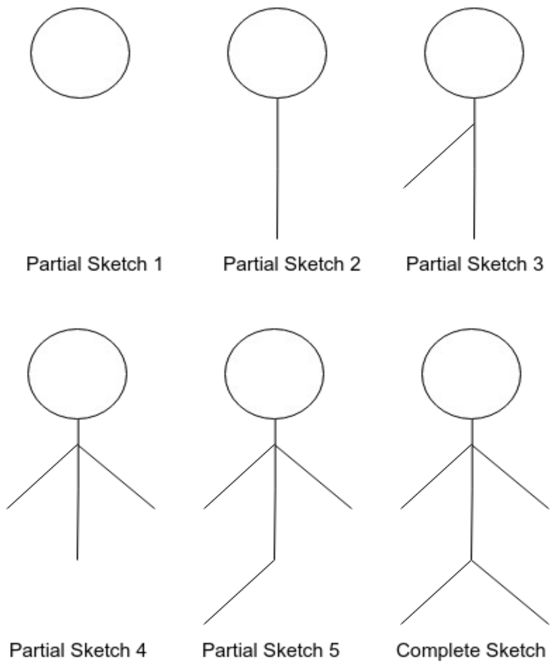


Figure 3: An example partial sketch generation from a full stick figure

3.1.2 Feature Extraction

In the original paper, the authors indicate that the sketches are represented in feature domain using IDM [8] feature representation. The pipeline of this feature extraction method is shown below.

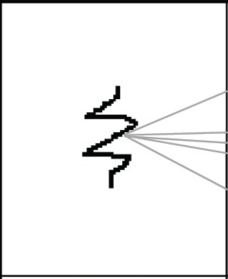
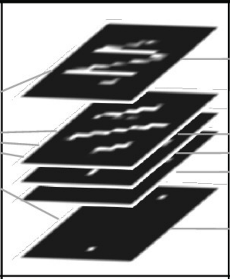
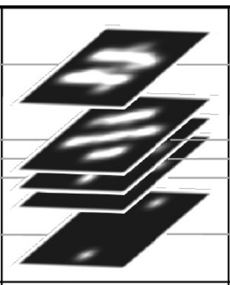
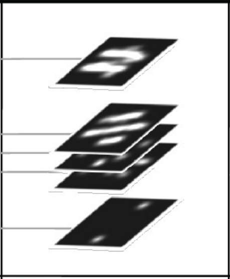
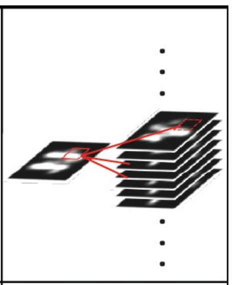
				
Input Normalize and compute feature values at each point in the pen trajectory.	**Feature Images** Extract feature images corresponding to four orientations and stroke endpoints.	**Smoothing** Apply Gaussian smoothing to reduce sensitivity to local distortions.	**Down-sampling** Down-sample images using a local max filter to improve robustness and performance.	**Classification** Match the input feature images against those for the prototypes in the training set, using IDM.

Table 2: The steps of IDM feature extraction

To extract the features of the extended dataset, we used IDM feature extractor of this laboratory [9]. While extracting the features of the entire dataset, we noticed that the features of some sketches couldn't get extracted. After the sketches giving these errors were identified, we found that the current implementation was not able to extract the features of pure horizontal/vertical lines and single points. By drilling into the code, we tried to mark the point where this error happened. While normalizing the sketches such that the standard deviation in both x and y coordinates is 1, if the standard deviation in at least one of those axis is zero, a “division by zero” error used to occur.

To fix this problem, we added a conditional check, whose pseudo-code is given below.

```
if std_x_axis == 0 do
  std_x_axis := 1
endif

if std_y_axis == 0 do
  std_y_axis := 1
endif

// continue normalization
```

Pseudo-code 1: Conditional checkers for standard deviation of x and y axis

Moreover, once the code was corrected, in order to make sure the code works correctly, we created three sketches, one pure horizontal line, one pure vertical line and one single point and run the modified code on them. The visualizations are presented below.

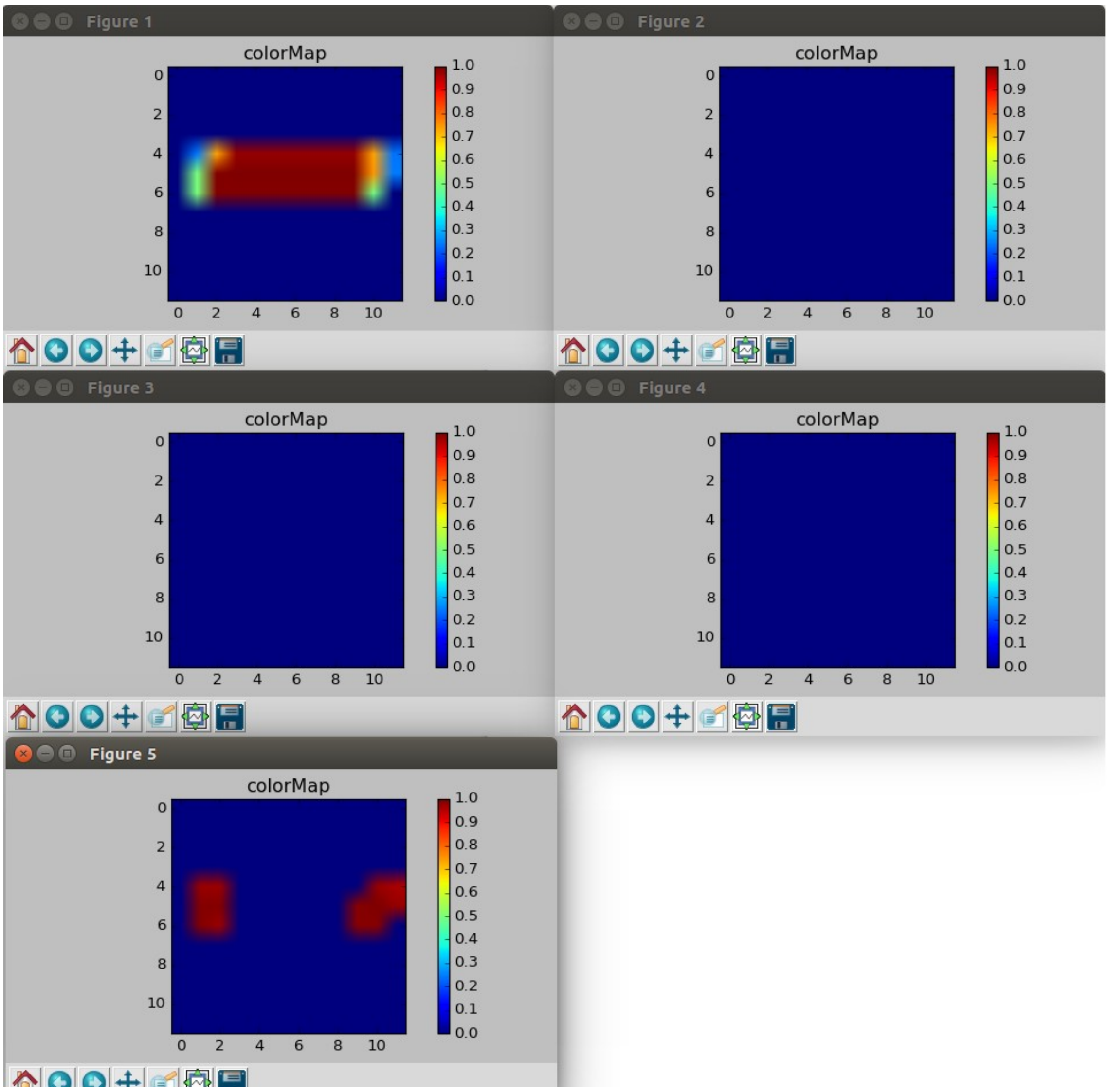


Figure 4: Feature representation of a horizontal line

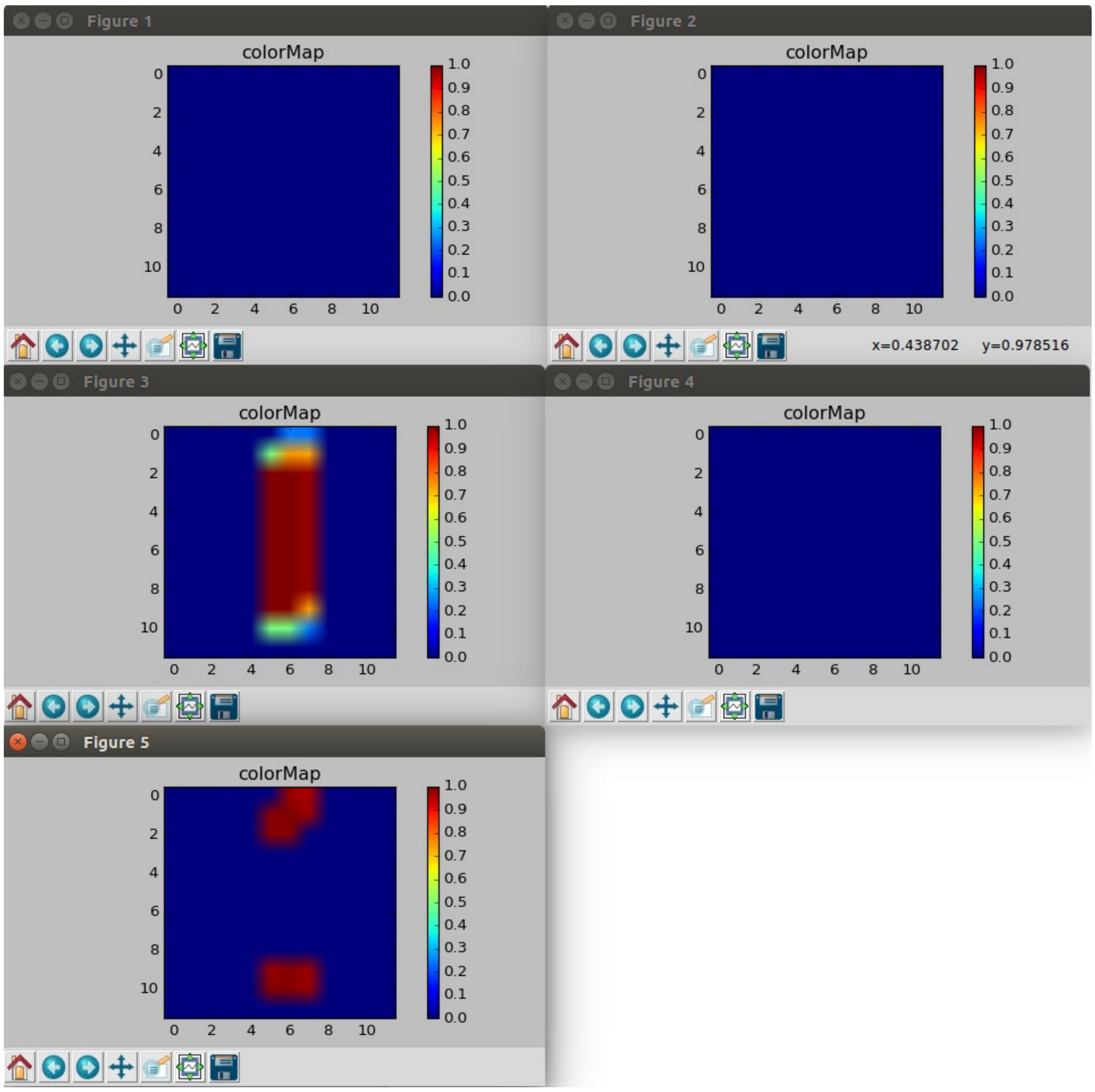


Figure 5: Feature representation of a vertical line

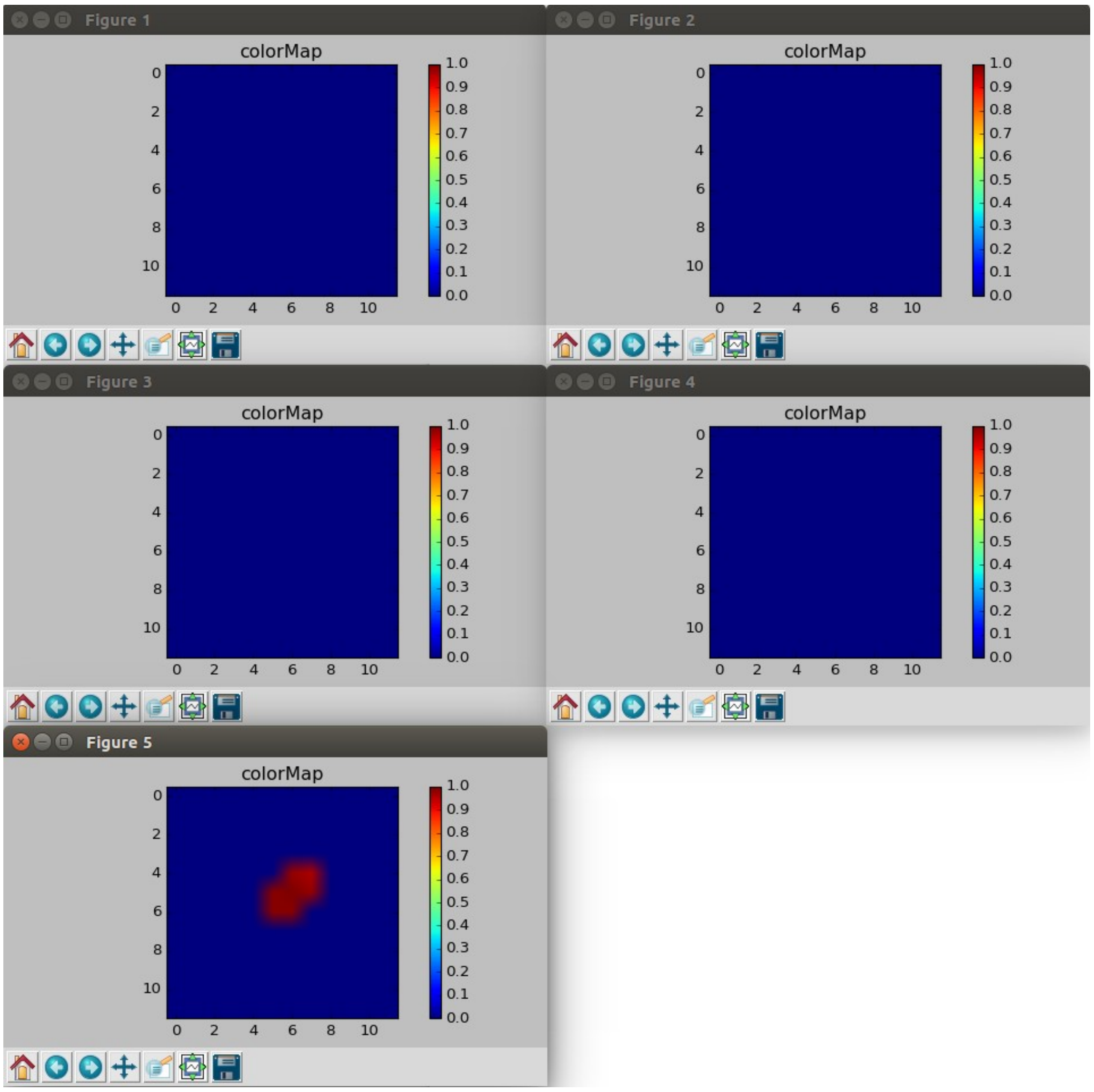


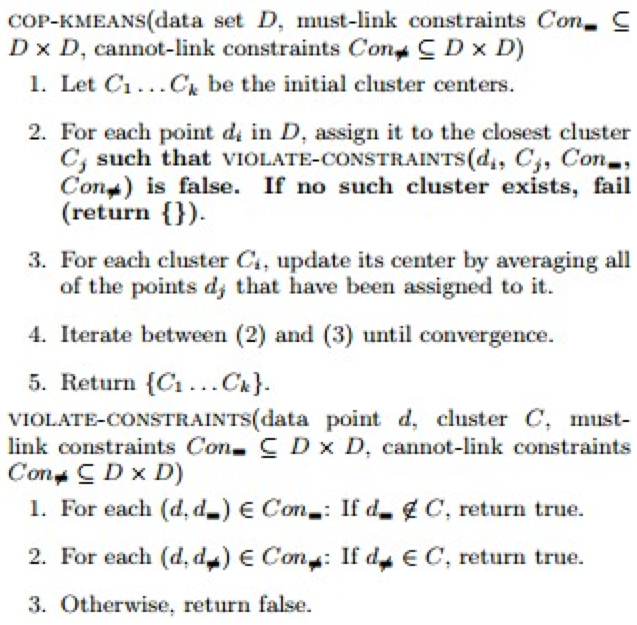
Figure 6: Feature representation of a single point

3.1.3 Constrained K-Means Algorithm

Naive constrained K-means algorithm we initially started to work with was the algorithm designed by Wagstaff et. al.. Basically, the constraints are that the full sketches of the same class should be clustered and that the full sketches of different classes mustn't be in the same cluster, disregarding the closest distance part of the K-Means algorithm. However, to partial sketches, standard K-Means algorithm is applied, where each partial sketch is directly assigned to the closest cluster center. Wagstaff's proposed algorithm is an iterative one, and fairly simple to implement. In order to apply the constraints, for each instance (sketch), the algorithm checks all other instances to see their assigned clusters. This provides a time complexity of $O(N^2)$, which makes it extremely slow and unfavorable on big datasets. Even on the small portions of the dataset, execution time consisted of several hours.

For an instance of 15,000 sketches, execution time of the MATLAB implementation is being held back by constrained K-means implementation and producing the output model took approximately 7 hours. Our educated guess is that on the full dataset, execution time would be up to nearly three months even if we assume the constraint matrix included in the algorithm, which will eventually include 350,000 x 350,000 integers, can fit into main memory. For this reason it seems infeasible to integrate the naive algorithm into iMotion system and it seems obvious that the naive Constrained K-Means algorithm is not designed for big datasets.

However with further inspection, we found several workarounds which can potentially reduce the execution time of the naive Constrained K-Means algorithm from $O(N^2)$ to $O(N)$. In general, as we are not restricted by space constraints, we made a trade-off between computation and space and precomputed most of the necessary information constrained K-Means algorithm checks every time repeatedly. The biggest contribution was getting rid of the constraint matrix which has a space complexity of $O(N^2)$. Although constraint matrix provided us with the important structure of dependency, this information was already available to us, without creating the expensive constraint matrix, as we were aware of that dependency is only among the full sketches of the same class.

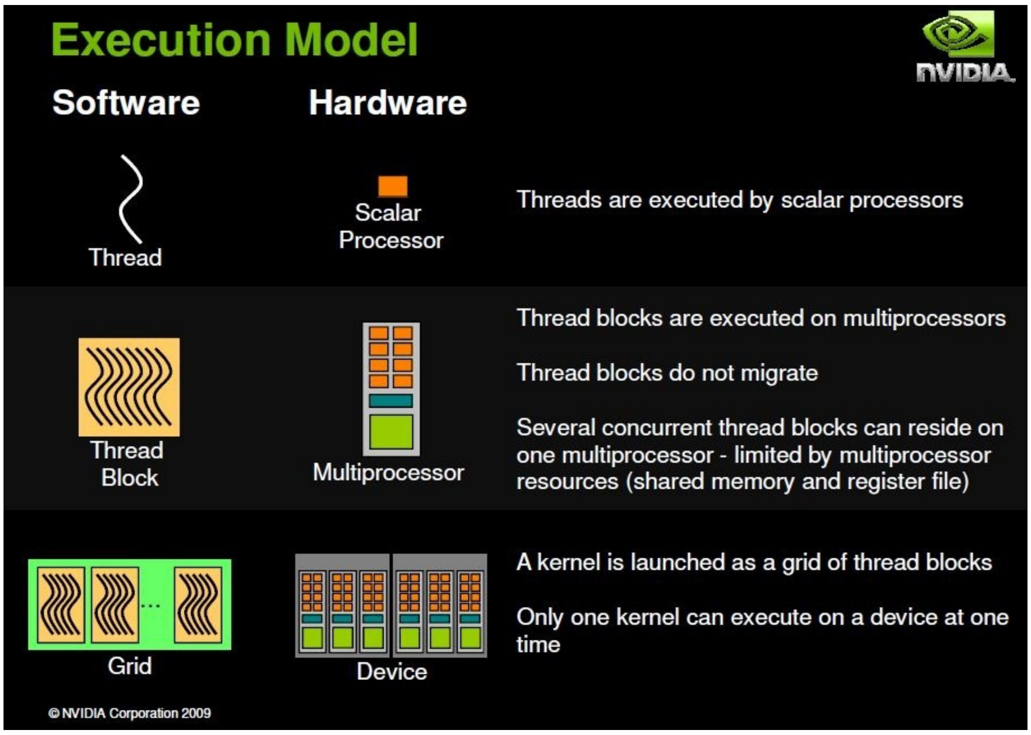


```
COP-KMEANS(data set $D$, must-link constraints $Con_= \subseteq D \times D$, cannot-link constraints $Con_{\neq} \subseteq D \times D$)
  1. Let $C_1 \ldots C_k$ be the initial cluster centers.
  2. For each point $d_i$ in $D$, assign it to the closest cluster $C_j$ such that VIOLATE-CONSTRAINTS($d_i$, $C_j$, $Con_=$, $Con_{\neq}$) is false. If no such cluster exists, fail (return {}).
  3. For each cluster $C_i$, update its center by averaging all of the points $d_j$ that have been assigned to it.
  4. Iterate between (2) and (3) until convergence.
  5. Return $\{C_1 \ldots C_k\}$.
VIOLATE-CONSTRAINTS(data point $d$, cluster $C$, must-link constraints $Con_= \subseteq D \times D$, cannot-link constraints $Con_{\neq} \subseteq D \times D$)
  1. For each $(d, d_=) \in Con_=$: If $d_= \notin C$, return true.
  2. For each $(d, d_{\neq}) \in Con_{\neq}$: If $d_{\neq} \in C$, return true.
  3. Otherwise, return false.
```

Pseudo-code 2: Naive constrained K-means algorithm

Furthermore, we realized that it is possible to ameliorate this algorithm through parallelization, although the naive version of that algorithm had considerable amount of data dependency, the iterative implementation was redundant as mentioned. Before we optimized the algorithm, we broke it down the into 4 separate parts: calculating the distance between sketches and clusters, applying constraints, assigning sketches to clusters and finding new cluster centers. It was obvious that the bottleneck of the execution would be the calculation of distances. We rewrote the calculation code so that it became suitable for an embarrassingly parallel code block. From this point on, we needed to find a viable development platform to apply our parallel algorithm. After looking through multiple options, we chose to implement the code in CUDA, as we had some experience with that platform beforehand.

NVIDIA graphics cards from 2007 and onwards have the capability to run CUDA [10], as their architecture is designed that way. Simply put, one graphics card, which is called grid, contains thread blocks (multiprocessor), in which threads (scalar processor) are being stored. Dimensions of grid and thread blocks depend on the hardware where the program is running. We looked through the limitations of computers on which the program will run (2 computers in the lab), and designed the memory structure of the program accordingly. Each sketch (full and partial alike) has been assigned to a thread, and every 512 thread has been assigned to a thread block. We have used the shared memory of the GPU to hold sketch information of every thread block, which greatly reduced read/write time of threads, as GPU shared memory nearly attains speeds that of registers. Given that the hardware limitations were the main obstacle during the implementation of this part, when processing large amounts of data which exceeds the memory of the GPU, we divide sketches into biggest blocks that GPU can hold, and process them iteratively. As a result of the implementation, this part of the algorithm was relieved from being the bottleneck of the program, and thanks to the calculation power of GPUs, it has shown approximately 400 times faster execution time.



Execution Model

Threads are executed by scalar processors

Thread blocks are executed on multiprocessors

Thread blocks do not migrate

Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)

A kernel is launched as a grid of thread blocks

Only one kernel can execute on a device at one time

Figure 7: Execution model in CUDA architecture

For the dependency constraints, we've decided to use the voting mechanics of constrained K-Means implementation (see *3.1.4* for details) directly from MATLAB code as they give far superior accuracy results than any of the algorithms we had previously tried. It seemed like Tirkaz et. al., in the implementation mentioned in the paper about sketched symbol auto-completion, had already materialized our idea of voting in his Constrained K-

Means implementation in the best possible way with a better insight.

Our contribution is shrinking Tirkaz et. al.'s algorithms time complexity to *O(N)* from *O(N²)* by eliminating the use of constraints matrix and expensive queries on that matrix.

The implementation of Tirkaz et. al. uses negative votes so that instances are encouraged to fall into the same cluster with the majority of their classes, however they are not forced to -as opposed to the naive CK-Means, where we force instances of the same classes to fall in the same cluster-.

Although this does not fully realize our background knowledge -that the full sketches of the same class should be in the same cluster-, it finds a middle ground; if an instance of a full sketch is too far away form the preferred cluster of its class, instance can still prefer a cluster other than the majority of its classmates suggest.

Now we will describe the full algorithm we implemented in Python, which mostly resembles the variant Tirkaz et. al. derives:

```
Algorithm  naive k-means
1: procedure K-MEANS(DATASET, K)
2:     initiliaze cluster centers
3:     while Not in local minima do
4:         Assign each data point to nearest cluster
5:         Recalculate the cluster centers
6:     return cluster centers
```

Pseudo-code 3: Naive K-means algorithm

In the standard K-means implementation each data point is directly assigned to the closest center and algorithm proceeds with calculating the cluster centers by taking means. Naive K-means algorithm does not consider the background knowledge we naturally possess from the distribution of the full sketches of the same classes.

On the other hand, our final CK-means algorithm works as follows:

Arguments: We take the data-set, a boolean indicating whether each sketch is full or not, an array to indicate the class of every sketch. With these three inputs, we can apply K-means algorithm using our constraints since we can deduce the constraints using only those inputs.

Initializing cluster centers: First we initialize the cluster centers via calculating the mean of the full sketches for each class. In other words, each cluster center is assigned to mean of the full sketches of the same class. If number of cluster centers is bigger than the number of sketch classes, we initialize other cluster centers randomly.

GPU computation (part of the loop): At the beginning of each loop, cluster centers and an empty matrix of size *number of instances* x *number of clusters* is send to GPU to obtain the following:

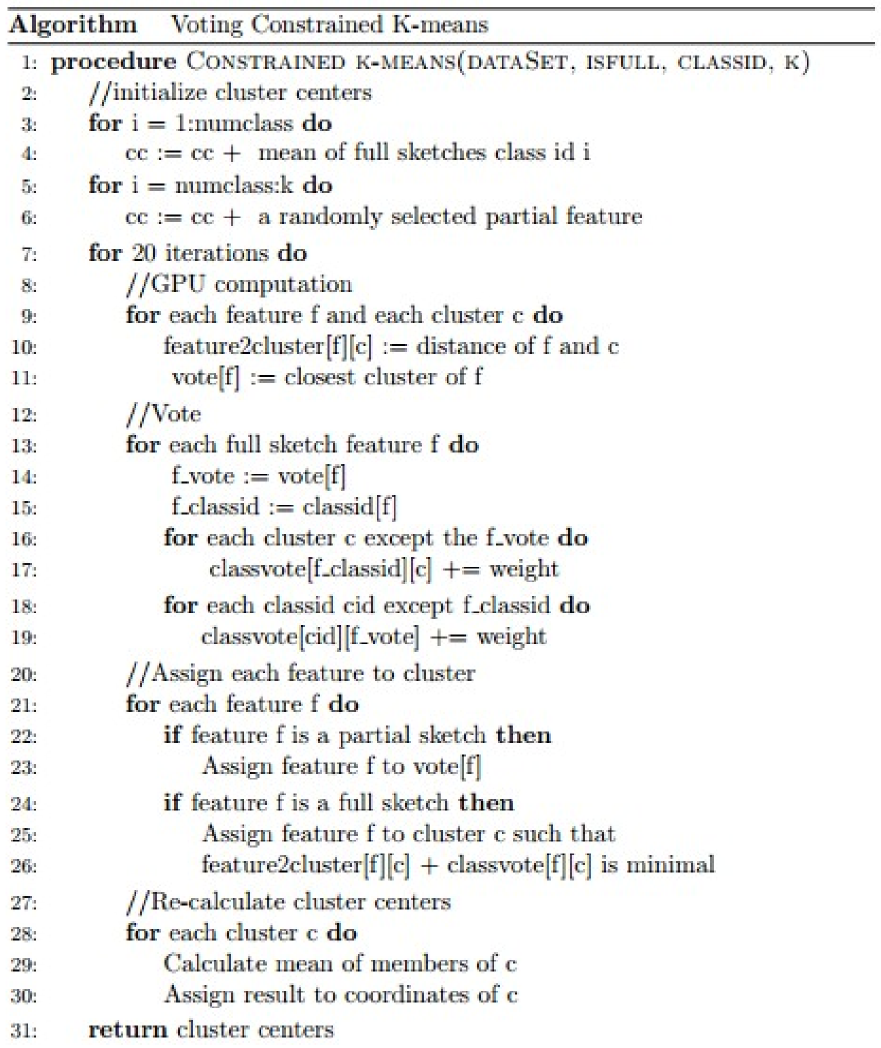
- The distance of every instance to every cluster
- The closest cluster center for each instance (e.g. vote of this instance)

Voting (part of the loop): Since GPU returns the closest center for each feature, this particular cluster center constitutes the necessary information for voting. Every feature votes for its own class, however the main difference of this implementation is that instead of voting to promote a certain cluster, they vote for preventing themselves from going to other clusters. Hence, every feature -for its class- negatively votes for the following:

- For full sketches of the same class to be assigned to other clusters
- For full sketches of different classes to be assigned to the same cluster

As partial sketches do not have any constraints, they do not vote, and are treated as we are doing simple k-means iteration. They are directly assigned to their respective cluster centers at the beginning of voting, without being included.

Assigning full sketches (part of the loop): When the time for assigning a cluster has arrived, a full sketch looks for each cluster, considers the negative votes together with distance to every individual cluster, and applies a weighted sum of those two factors. Eventually it gets assigned to a cluster with the smallest sum. Partial sketches are assigned in the same way as the K-means without constraints. Hence they do not consider negative votes but the distance for each cluster center. Weight parameters are used to balance negative votes and distance metric. The total sum is calculated as (weight*number_of_negative_votes + distance) so weight = 0 implies K-means without constraints, and a large weight implies neglecting distances, and only considering the votes for each class.



```
Algorithm  Voting Constrained K-means
procedure CONSTRAINED K-MEANS(DATASET, ISFULL, CLASSID, K)
    //initialize cluster centers
    for i = 1:numclass do
        cc := cc + mean of full sketches class id i
    for i = numclass:k do
        cc := cc + a randomly selected partial feature
    for 20 iterations do
        //GPU computation
        for each feature f and each cluster c do
            feature2cluster[f][c] := distance of f and c
             vote[f] := closest cluster of f
        //Vote
        for each full sketch feature f do
            f_vote := vote[f]
            f_classid := classid[f]
            for each cluster c except the f_vote do
                classvote[f_classid][c] += weight
            for each classid cid except f_classid do
                classvote[cid][f_vote] += weight
        //Assign each feature to cluster
        for each feature f do
            if feature f is a partial sketch then
                Assign feature f to vote[f]
            if feature f is a full sketch then
                Assign feature f to cluster c such that
                feature2cluster[f][c] + classvote[f][c] is minimal
        //Re-calculate cluster centers
        for each cluster c do
            Calculate mean of members of c
            Assign result to coordinates of c
    return cluster centers
```

Pseudo-code 4: Our constrained K-means algorithm (feature refers to an instance)

3.1.4 Reducing Time and Space Complexity of CK-Means Algorithm

The problem that the original MATLAB code cannot train on Eitz dataset can be explained by the space and time complexity of this implementation, and mainly of the constrained K-Means algorithm, which constituted the bottleneck of the implementation. Constrained K-Means implementation made it impossible to even store necessary data on the main memory, as the complete training of the Eitz dataset requires 19 gigabytes of RAM only for the constraints matrix. Even if we assume that we can handle the necessary memory shortage, training can still take up to two months.

The aim of using CK-Means algorithm is to exploit background knowledge of the class information of the sketches to encourage full sketches of the same class to fall into the same cluster, so to preserve full sketch accuracy preventing too much confusion on full sketches.

This was achieved by putting must-link constraints between full sketches of the same class, and putting cannot-link constraints on full sketches of different classes. This encourages respective instances to get clustered together, or encourage them to get separated, if cannot-link constraint is the case. Of course it is more than obvious that constraint matrix can be calculated from the information of whether or not a sketch is full, and the class information for sketches. Instead of calculating constraint matrix, replacing this with checking the condition of two sketches to be full and from the same class -or checking whether they are from different classes- will get rid of the constraint matrix, however this even hurts the run time.

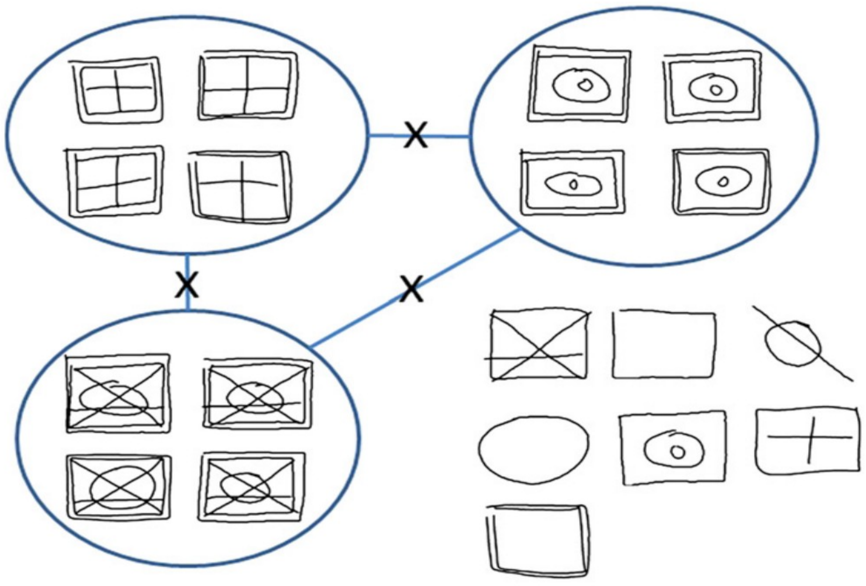


Figure 8: Showing must-link and cannot-link constraints

The way that the algorithm of Tirkaz et. al. works is to change the distance metric, using the sum of the must-link and cannot-link instances in addition to Euclidean distance.

Must-link distance of an instance to a cluster: Number of features not in this cluster having must-link constraint with the instance

Cannot-link distance of an instance to a cluster: Number of features *in* this cluster having cannot-link constraint with the instance

The way that algorithm calculates the must-link and cannot-link distances is the most expensive way, for each feature, iterating a row of the constraint matrix -which has width of ~350.000- to check the dependencies, and check the cluster of the instances which has the constraint relationship. Therefore in each iteration, two matrices with the size of ~300.000x300.000 are *and*'ed, and every row is summed up. This constitutes the main drawback of the algorithm.

It seems that the current algorithm implementation is the most general, and it can be changed to exploit the regularity in the data. The speed-up we can generate is to see that *must-link distance + cannot-link distance* is unique for a *(class, cluster)* pair, since all the features inside a class has exactly the same cannot-link and must-link constraints. Instead of calculating *must-link distance + cannot-link distance* for each instance to be clustered, we can calculate it once for a specific class and cluster and use it again for each instance of the same class. Then the new strategy is to iterate over instances twice, first to calculate the (class, cluster) distances, and then find the cluster which minimizes our new distance metric.

Hence, the mere contribution is calculating the class to cluster distances for the must-link and cannot-link constraints.

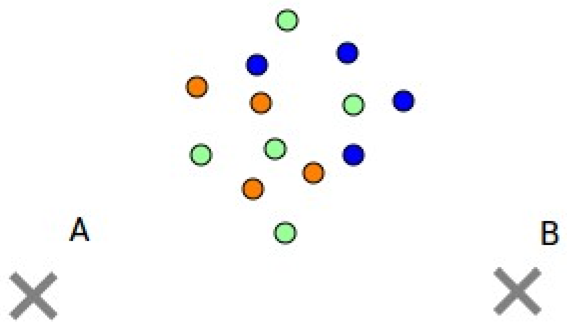


Figure 9: An example scene before voting

	A	B
Blue	0	0
Green	0	0
Orange	0	0

Table 3: Class vs. cluster distances of Figure 9

Now we will iterate over instances and fill the class to cluster (*must-link* + *cannot-link*) distance matrix.

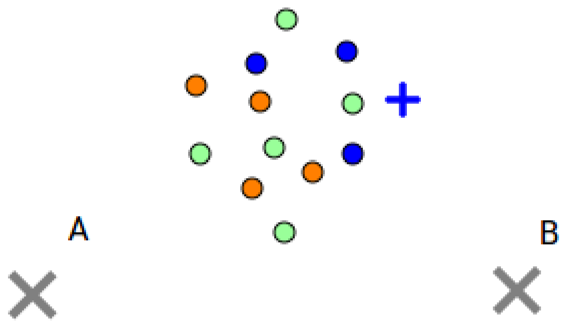


Figure 10: First voting

	A	B
Blue	1	0
Green	0	1
Orange	0	1

Table 4: Class vs. cluster distances in the first voting

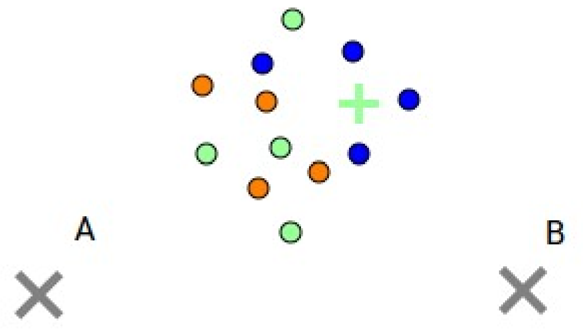


Figure 11: Second voting

	A	B
Blue	1	0+1
Green	0+1	1
Orange	0	1+1

Table 5: Class vs. cluster distances in the second voting

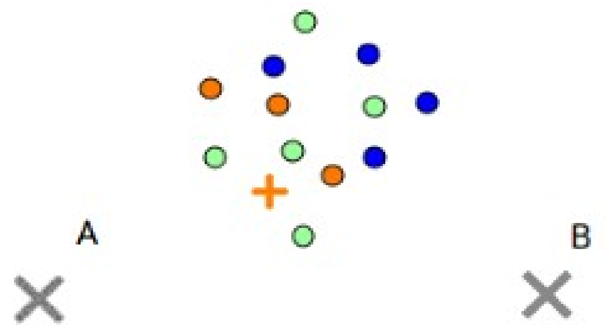


Figure 12: Third voting

	A	B
Blue	1+1	1
Green	1+1	1
Orange	0	2+1

Table 6: Class vs. cluster distances in the third voting

Once we iterated over instances, since then we know all the *from instance to cluster* distances, we can continue running constrained K-means algorithm, without the burden of calculating these distances for each feature separately.

3.1.5 Probability Calculation

In order to make a prediction given a test instance *x*, we compute the posterior probability of each symbol class, by running SVM prediction function over clusters and then multiplying probabilities in clusters with the probability of the instance being in the cluster.

$$P(s_i|x) = \sum_{k=1}^{K} P(s_i, c_k|x) = \sum_{k=1}^{K} P(s_i|c_k, x)P(c_k|x)$$

where x represents the input symbol; K is the total number of clusters; $P(s_i/ c_k,x)$ is the probability of symbol class s_i given cluster c_k and input x; and

$P(c_k|x)$ denotes the posterior probability of cluster c_k given x. Given the distance from x to each cluster center, we estimate *$P(c_k|x)$* as

$$P(c_k|x) = e^{(-\|x-\mu_k\|^2 + min_k(\|x-\mu_k\|^2))/\sigma^2} P(c_k) / P(x)$$

where μ_k is the mean of *k*th cluster c_k and *$P(c_k)$* is the prior probability of c_k estimated using the proportion of the number of instances the cluster *k* and the total number of instances and the coefficient σ is set to 0.3.

To realize the above formula, we have written the python class named **`Predictor`**. In this script, the probability *$P(c_k|x)$* is calculated using the function named **`clusterProb`**, the probability *$P(s_i|c_k,x)$* is calculated using the Python package named LibSVM [11] and the function named **`calculatePosteriorProb`** computes *$P(s_k|x)$*.

3.1.6 Using Support Vector Machines on Clusters

In this project, we have adopted a different approach to the conventional use of support vector machines. In a more general situation, vector machines would be trained by passing raw data as argument. Here, training of vector machines are being done by passing clusters containing sketch data as arguments and training clusters in an orderly fashion to create support vector machine models.

For this particular work, LibSVM API seemed the best option, given the availability on Python with well defined functions, minimum overhead on memory and multiple options to tweak the support vector machine itself to serve to purpose of this project.

After analyzing the work done in the auto-completion paper, we have concluded that our classification method for SVM would be Cost-Support Vector Classification, and our kernel type would be Radial Basis Function. We have set γ coefficient to 0.125, as it would give out the most desirable results according to the paper, and set the cost coefficient of Cost-Support Vector Classification to 8. These options are reflected in Python in the following form:

```
svm_parameter('-s 0 -t 2 -g 0.125 -c 8 -b 1 -q')
```

where *-b* option lets us train an SVM model for probability estimates and *-q* option lets us omit the output of the function, which is given as default by LibSVM API.

In the program, after setting the parameters, we need to define `svm_problem`. This built-in function takes 2 arguments, first one is the supervised classes of sketches and the second one is the sketch data with all dimensions included. As our approach is to train clusters individually, for each cluster, we set the `svm_problem` according to the data inside the cluster at hand. After this step, the only action required is to call `svm_train` function using `svm_problem` as a parameter.

```
Algorithm SVM Training
1: procedure TRAINSVM(CLUSTERIDARR, DIRECTORY, PROCID)
2:     define x and y as list
3:     for each cluster c do
4:         for each feature f in c do
5:             append class label of f to y
6:             append features of f to x
7:         problem := svm_problem(y, x)
8:         param := svm_parameter('-s 0 -t 2 -g 0.125 -c 8 -b 1 -q')
9:         svm_train(problem, param)
```

Pseudo-code 5: Pseudo-code of SVM training on clusters

3.1.7 Saving Trained Model on Disk

At the first stage of the implementation, runtime of feature extraction used to be slow and redundant. Since dataset being used for testing was fixed (ie. Eitz and NicIcon datasets), in order to reduce running time of testing we decided to generate features of datasets and save them. Via using this approach, features can be directly loaded and used for training, thus we overcame the redundancy problem.

Another concern was using models multiple times. To overcome this issue, keeping clusters and their SVM outputs was essential. In this way, trained models can be loaded multiple times and testing can be done easily.

For the reasons explained above, we have implemented a class named `FileIO`. Using Pandas [12] library this class can save and load necessary information such as extracted features, cluster centers as .csv files. This format was chosen for its readability and easy implementation. SVM models trained are saved and loaded using LibSVM built-in functions `svm_save_models` and `svm_load_models`.

For saving testing results, Python serialization library Pickle [13] was used. In this way a variety of plots can be generated without running the same test again.

3.1.8 Increasing Performance by Applying Multiprocessor Parallelization

During the development cycle of the project, we have seen some issues regarding the performance of the application. If the system were to be tested more rigorously, the execution time had to be reduced. For this reason we implemented the framework for CUDA architecture for the sake of utilizing the power of the GPU. Using CUDA, we've obtained fantastic results, and we had the chance to test the entire Eitz dataset (~350.000 sketches, including user-generated partial sketches), In light of this successs, we have decided to start searching for parts of the program where similar amelioration can be achieved.

In SVM training and saving section, we have found that clusters passed to the vector machine are independent from each other, they do not require being sent in an iterative manner. It seemed that the structure of the algorithm could be re-programmed with parallel programming as the state of the algorithm was already an embarrassingly parallel problem. Pool library of Python was suitable for this job, so we've implemented a process pool with 4 processes inside. Each process takes ¼ of clusters with data in them, and train the vector machine. When a cluster is trained, the resulting model is saved. Having implemented the parallelization, we've tested the iterative and parallel implementation of SVM training/saving.

Results are the following:

	Iterative	**Parallel**
5 clusters	3m45s	2m
25 clusters	24m	9m
50 clusters	42m	16m
150 clusters	1h56m	41m

Table 7: Running time of the iterative and parallelized training pipelines

The image below best illustrates how the processes work with clusters in an orderly but parallel fashion.

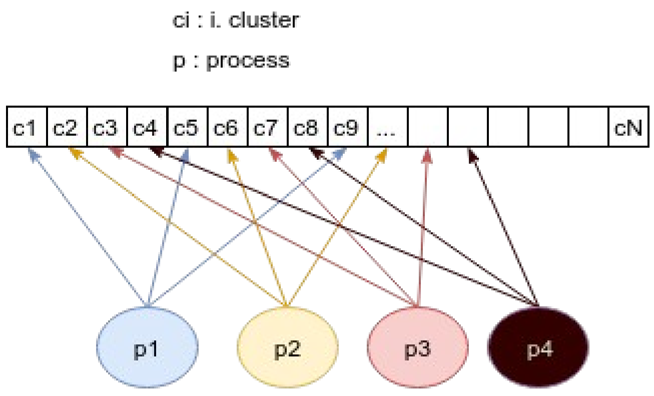


Figure 13: The illustration of the parallelization of SVM training

3.1.9 A New Method

Following the implementation of the standard pipeline, we also started research for a new method that was hoped to increase partial prediction accuracy.

In order to deal with the wide range of sketch classes, one idea was to divide the class space into several groups. After grouping, standard pipeline can be run on every group which contains a subdivision of the huge dataset. The goal of this step is to collect similar classes in a group where the standard pipeline will run on a much specialized data set. For example, running standard recognition training on a group which contains round natured sketches, is expected to create a model which have more ability to differentiate round natured sketch inquiries.

Another motivation for dividing the data into several groups was reducing the time taken for clustering operation of the standard pipeline which was the bottleneck of the training.

During recognition, the system firstly finds the distance of each symbol to every group and then computes the posterior probabilities via running standard pipeline in every single group.

Grouping: There is an ambiguity in finding similar sketch classes. Any random sketch can not be chosen as the representative of a class. In order to solve this ambiguity, we compute the mean of the sketches (using two method: by using only full sketches and by using all extended sketches) based on their feature representation. Computed mean vector is accepted as the representative of the class.

For grouping, class representatives are clustered using iterative K-Means algorithm where the iteration number is given as an input argument. Throughout the iterations, the output with the minimum total distance to their centers is chosen. Moreover, to obtain evenly populated groups, at each iteration, the clusters where there are too few elements get rejected.

The algorithm used for grouping is given below.

```
Algorithm  Grouping algorithm
  procedure K-MeansGroupGenerator($numOfGroups, classReps, iterNum, coeff$)
      $threshold \leftarrow (len(classReps)/numberOfGroups) \times coeff$    ▷ Minimum number of class in each group
      $bestDistance \leftarrow Inf$
      while $i < iterNum$ do
          $i \leftarrow i + 1$
          $nowGroups, nowCenters \leftarrow kMeansClusterer(numOfGroups, classRep)$
          $nowDistance \leftarrow getDistance(nowGroups, nowCenters)$
          if $nowDistance < bestDistance$ then
              if $len(smallestGroup) > threshold$ then
                  $bestDistance \leftarrow nowDistance$
                  $bestGroups \leftarrow nowGroups$
              end if
          end if
      end while
      return $bestGroups$
  end procedure
```

Pseudo-code 6: Pseudo-code of grouping algorithm

To experiment and observe the effect of the grouping using K-means algorithm, we tried another method named random grouping. Running the new pipeline using random grouping was expected to give lower accuracies since distances to groups becomes no longer meaningful. Moreover, when the experiment is repeated several times, this method leads to a larger standard deviation since the division of groups might change the model drastically.

Training: Having grouped the instances, in every single group, the standard pipeline is run. Here we should state that in every group, the number of clusters is given by hand. Thus, at the training stage, alternative method trainer takes an array input including the number of clusters in groups.

In experiments, we used different cluster number sets and obtained the best result when they are set to the number of class in every group.

The diagram summarizing the training pipeline is given below.

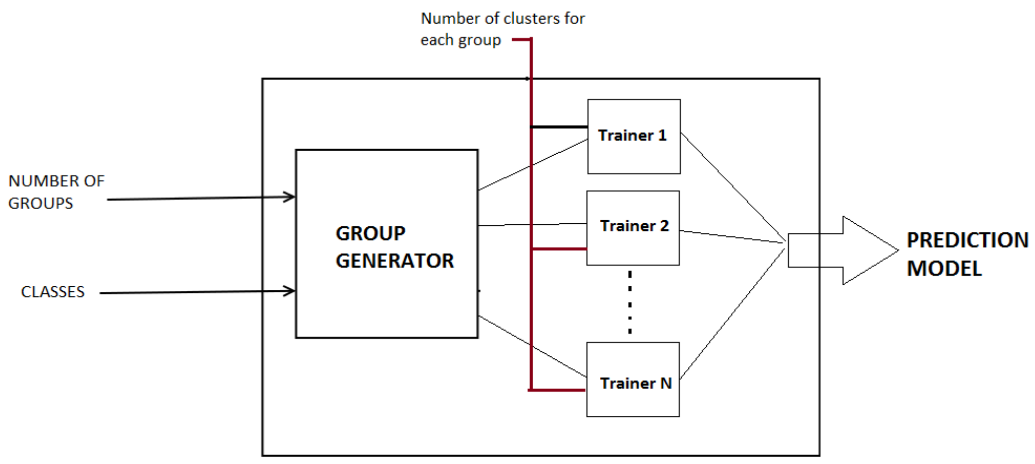


Figure 14: The training pipeline of the alternative method

Posterior Class Probabilities: In order to make a prediction given a test symbol x, we compute the posterior probability of each symbol class s_i, by computing the probability of x, to be in a group and then running standard probability calculation method in the specific group.

$$P(s_i|x) = \sum_{k=1}^{K} P(s_i, g_k|x) = \sum_{k=1}^{K} P(s_i|g_k, x)P(g_k|x)$$

Using the above formula, the class probabilities of the instances can be calculated. To explain the formula further, x represents the input symbol, K is the total number of groups, $P(s_i/g_k,x)$ denotes the probability of symbol class s_i given group g_k and input x; and $P(g_k/x)$ denotes the posterior probability of group g_k given x. The probability $P(s_i/g_k,x)$ is computed using the standard pipeline.

Given the distance from x to each group center, we estimate $P(g_k/x)$ as

$$P(g_k|x) = e^{(-\|x-\mu_k\|^2 + min_k(\|x-\mu_k\|^2))/\sigma 2} P(g_k)/\,P(x)$$

where μ_k is the mean of the kth group g_k and $P(g_k)$ is the prior probability of g_k estimated using the proportion of the number of instances that resides in the group k and the total number of instances and the coefficient σ is set to 0.3.

The entire pipeline including both training and testing is given below.

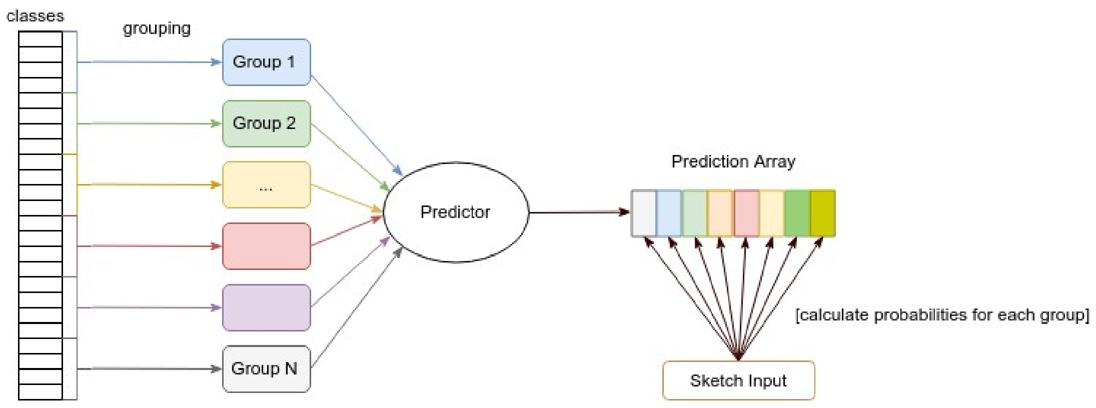


Figure 15: The diagram summarizing the whole pipeline

3.2 Prediction

3.2.1 Implementation

Our implementation strictly followed the paper written by Tirkaz et. al.. More than that, we found that results in the paper cannot be reproduced and observed some ambiguities and unspecified hard-coded parameters some of which disastrously degraded our accuracy results. We could only realize the differences -paper versus implementation- through a further inspection on the demonstration code after we found our accuracy results cannot match the ones given in the paper, although we were sure that we implemented the pipeline correctly. In general, there were three parts which are not mentioned in the paper, or not stated precisely.

- The details of CK-means implementation
- Calculation of posterior class probabilities
- How partial sketches are added to the dataset of pure full sketches
- Applying multiprocessor parallelization to accelerate the pipeline

The details of CK-means implementation: Since the paper does not point out a specific implementation of the constrained K-means implementation, we started implementing the most popular and straightforward one by Wagstaff et al.. This implementation has a time complexity of O(N^2) if we keep the other parameters fixed. This can become problematic when the number of instances increases. We reduced time complexity to O(N) via exploiting the structure of our data. However in the actual implementation of the paper, an entirely different approach of constrained K-means was used, although the work cited was the paper of Wagstaff et. al., which is recently discussed. Understanding how instances are clustered was not possible without digging into the demonstration code provided by the author, and subtle differences between the work cited and the actual implementation was not explained in the paper.

Calculation of posterior class probabilities: During the calculation of the posterior probabilities, we firstly calculate the posterior cluster probabilities, which is defined in the paper as:

This calculation gives us the probability that an instance belongs to a cluster. Nevertheless, further inspection on the code provided by the author revealed that the actual implementation is quite different, and posterior cluster probability, as in the paper, dropped our accuracy results up to 10%, especially in full sketch classification. The actual implementation is the following:

$$P(c_k|x) = [P(x|c_k)P(c_k)/P(x)]\, e^{(-\|x-\mu_k\|^2 + min_k(\|x-\mu_k\|^2))/2\sigma^2} P(c_k)/P(x)$$

where σ is set to 0.3, without giving any particular reason neither in the paper nor in the original MATLAB implementation. It is sad to see a hard-coded value without any explanation -even in the paper-, on absence of which reduces the accuracy rates to 10%.

How partial sketches are added to the dataset of pure full sketches: The process of generating partial sketches is not explicitly stated in the paper, which makes it hard to reproduce the accuracy results. However since we could contact the authors, and we confirmed that the partial sketches are generated with respect to timestamps, so that assuming *n* strokes in a sketches, *n-1* partial sketches are generated, where *k*th partial sketch includes the first *k* strokes, sorted by the time-stamp. On the other hand we also questioned whether not all full sketches are used in the training -and possibly in testing-. The reason for this curiosity is that full sketch accuracies are higher than the paper and partial sketch accuracies are lower than the paper. These observations can be reversed, and accuracies can be made equal with the paper by reducing the number of partial sketches used in the training, since we have observed that number of full sketches greatly reduces the full sketch accuracy. However we couldn't find the answers by even asking the authors, since they seem to forgot the details of the project.

Applying multiprocessor parallelization to accelerate the pipeline: At the implementation phase, we have noticed that the predictor component of the program is also suitable for parallelization, and the performance (execution time) can be ameliorated, with an approach similar to the parallelization of SVM training/saving. A process pool is being initialized at the beginning of the prediction code block in a way that the pool spawns 4 processes and executes the job at hand. Then, each process takes up to *1/4* of data to work on. The main difference between this process pool and the former one is that the end results of the prediction tasks are executed simultaneously, so the results from each process is merged in a neat fashion. Having adopted this approach, we've seen that the prediction is sped up, nearly 4 times the speed of the iterative (former) one.

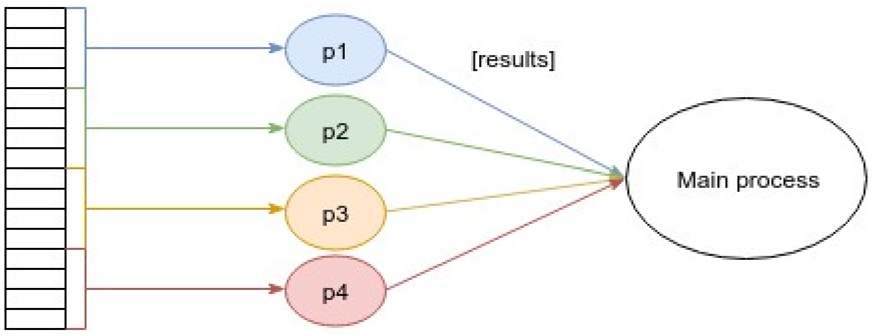


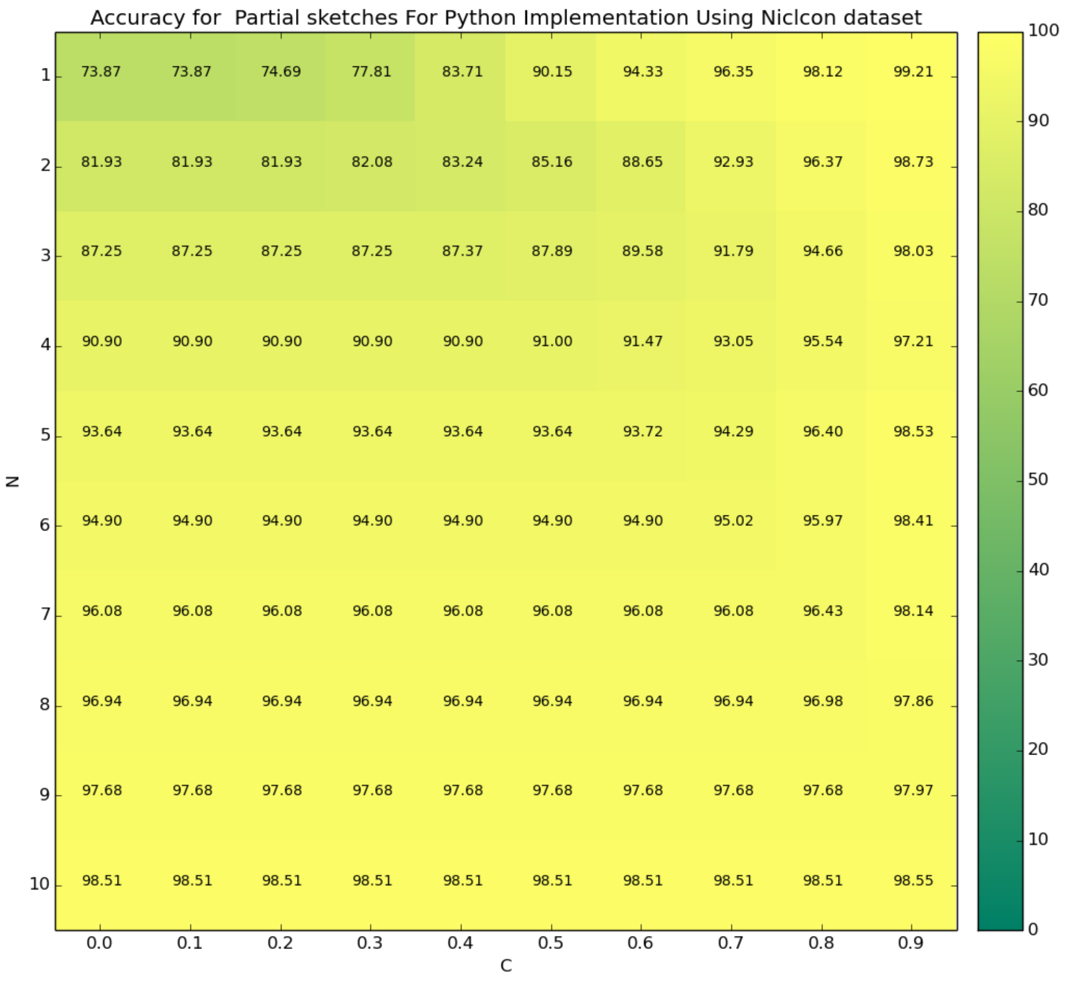
Figure 16: Illustration of parallelized prediction

Using both, we have ameliorated the performance of the execution time of the project by a significant margin, which enables us to test and further develop the program much easily. Only negative effect of this approach is that in order to gain performance, we had to sacrifice the memory, which is usually expected in multiprocessor programming. At the end, through a trade-off between performance/memory, the implementation of parallelization became successful.

3.3 Accuracy Testing & Results

3.3.1 On NicIcon Dataset

3.3.1.1 Standard Pipeline



N \ C	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
1	73.87	73.87	74.69	77.81	83.71	90.15	94.33	96.35	98.12	99.21
2	81.93	81.93	81.93	82.08	83.24	85.16	88.65	92.93	96.37	98.73
3	87.25	87.25	87.25	87.25	87.37	87.89	89.58	91.79	94.66	98.03
4	90.90	90.90	90.90	90.90	90.90	91.00	91.47	93.05	95.54	97.21
5	93.64	93.64	93.64	93.64	93.64	93.64	93.72	94.29	96.40	98.53
6	94.90	94.90	94.90	94.90	94.90	94.90	94.90	95.02	95.97	98.41
7	96.08	96.08	96.08	96.08	96.08	96.08	96.08	96.08	96.43	98.14
8	96.94	96.94	96.94	96.94	96.94	96.94	96.94	96.94	96.98	97.86
9	97.68	97.68	97.68	97.68	97.68	97.68	97.68	97.68	97.68	97.97
10	98.51	98.51	98.51	98.51	98.51	98.51	98.51	98.51	98.51	98.55

Figure 17: Accuracy performance of our implementation, on partial sketches of NicIcon dataset

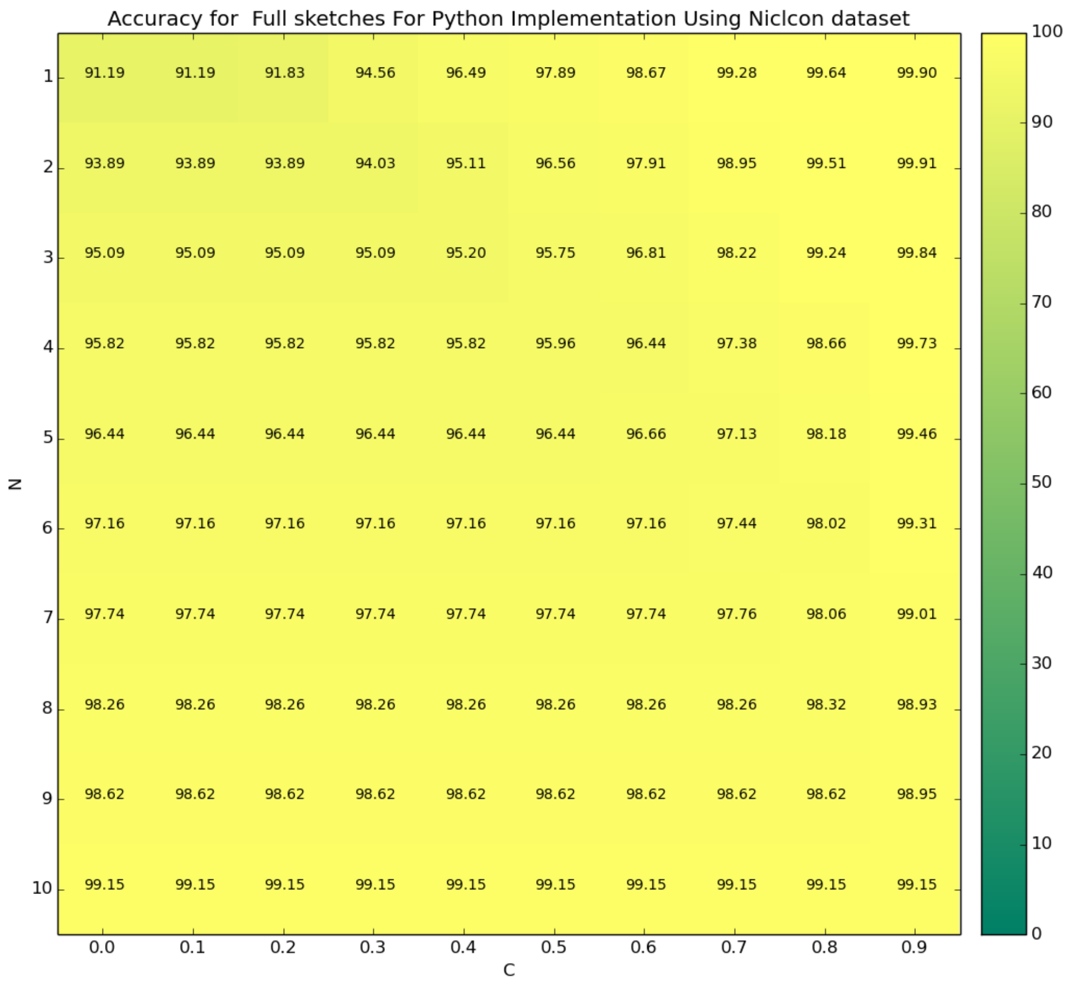


Figure 18: Accuracy performance of our implementation, on full sketches of NicIcon dataset

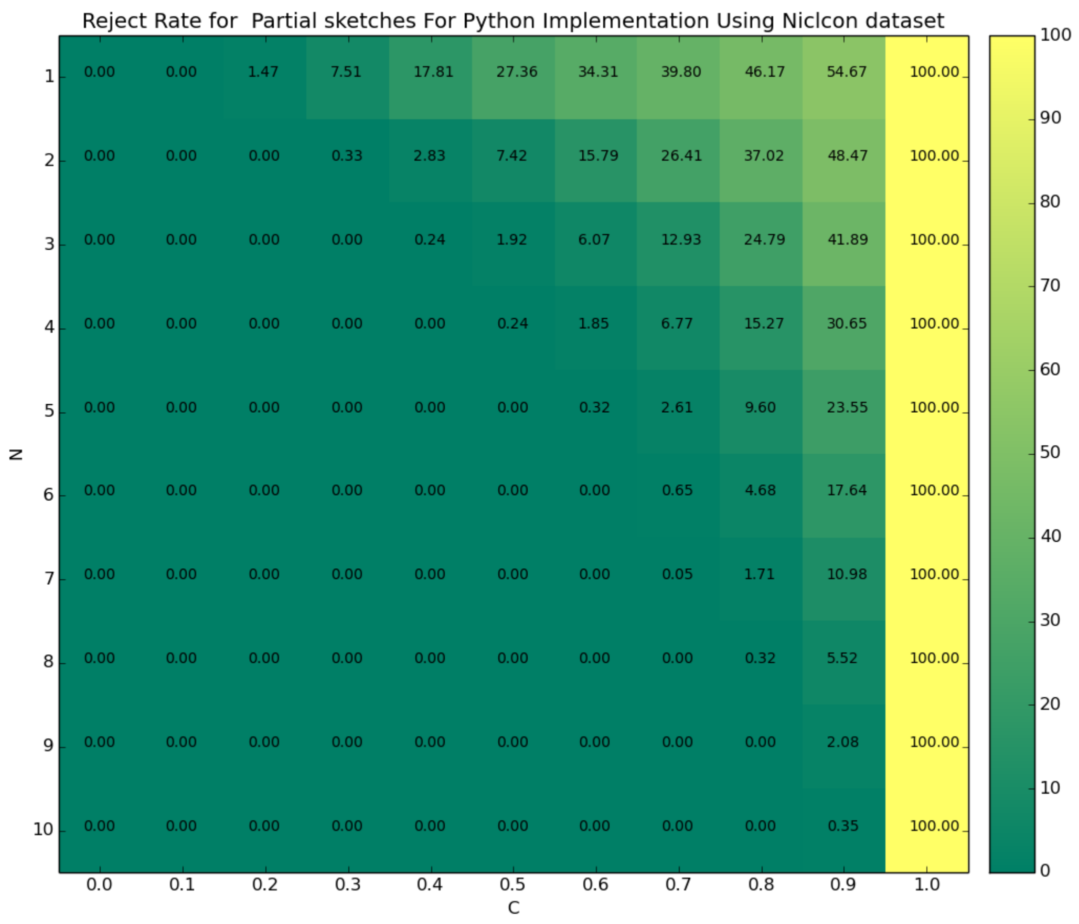


Figure 19: Reject rates of NicIcon partial sketches

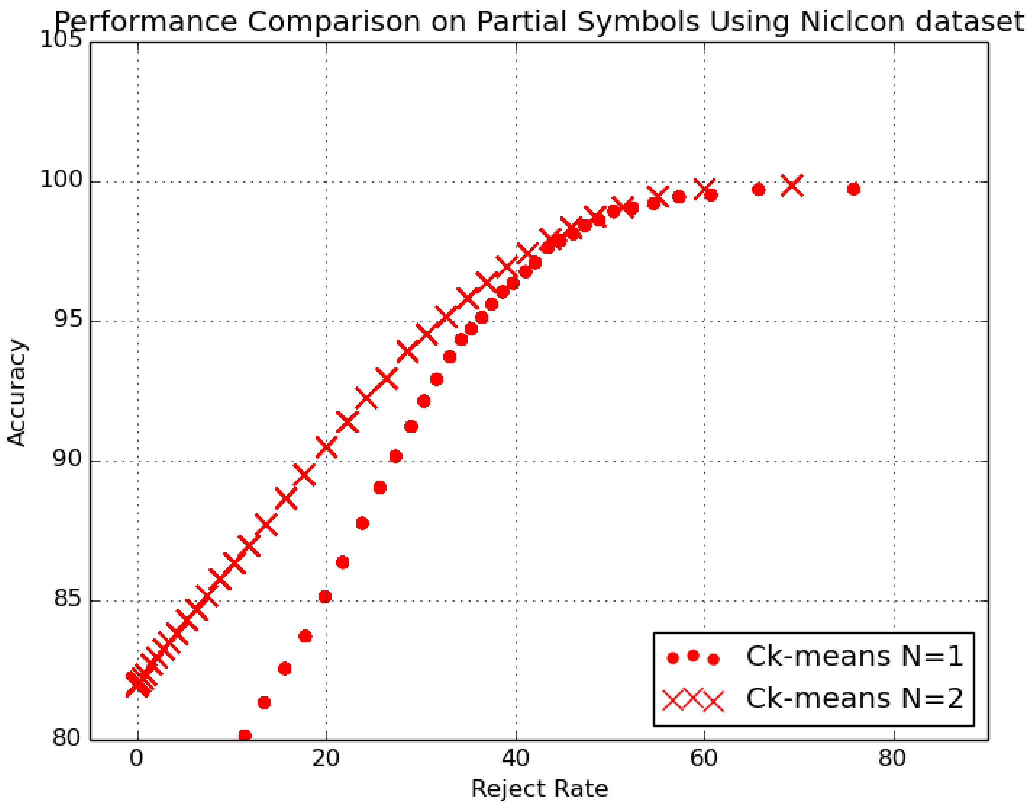


Figure 20: Reject rate vs. accuracy for NicIcon partial sketches

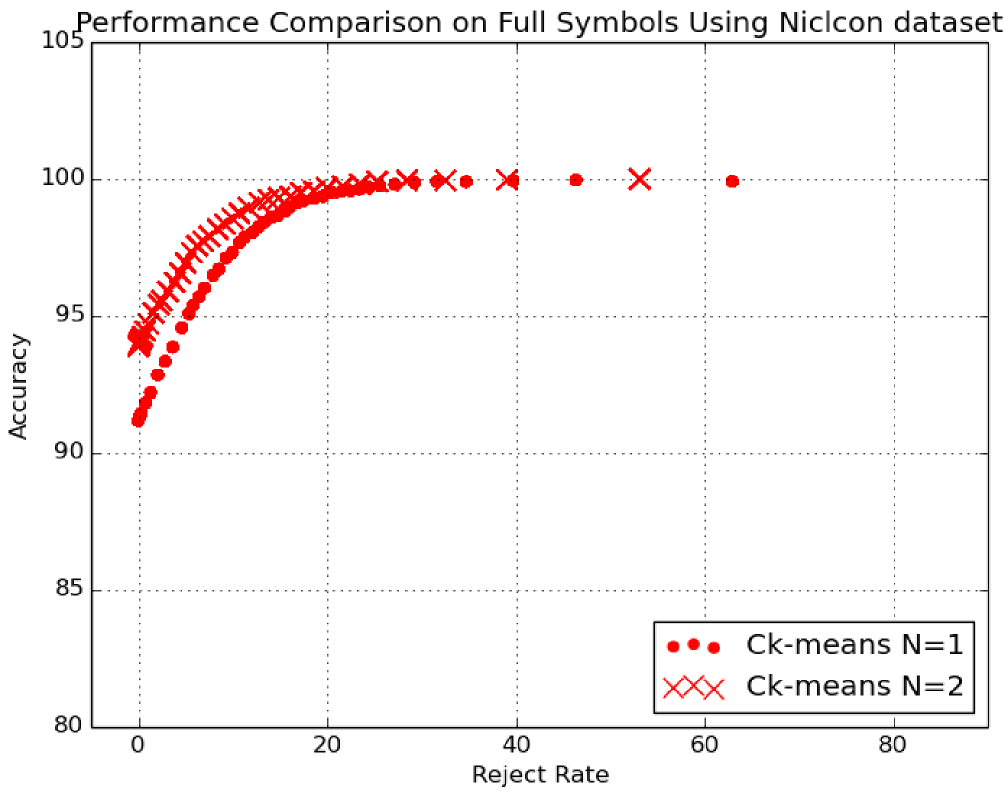
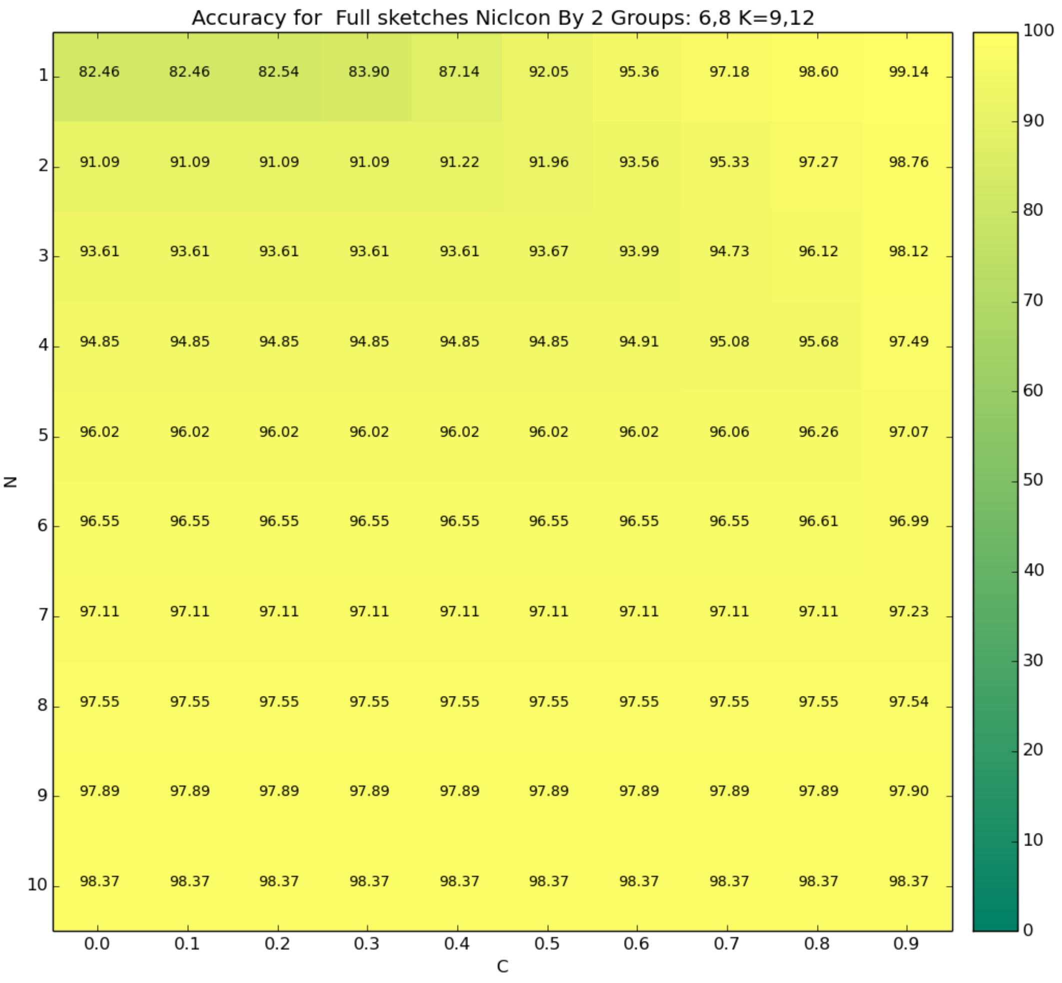


Figure 21: Reject rate vs. accuracy for NicIcon full sketches

3.3.1.2 Alternative Pipeline

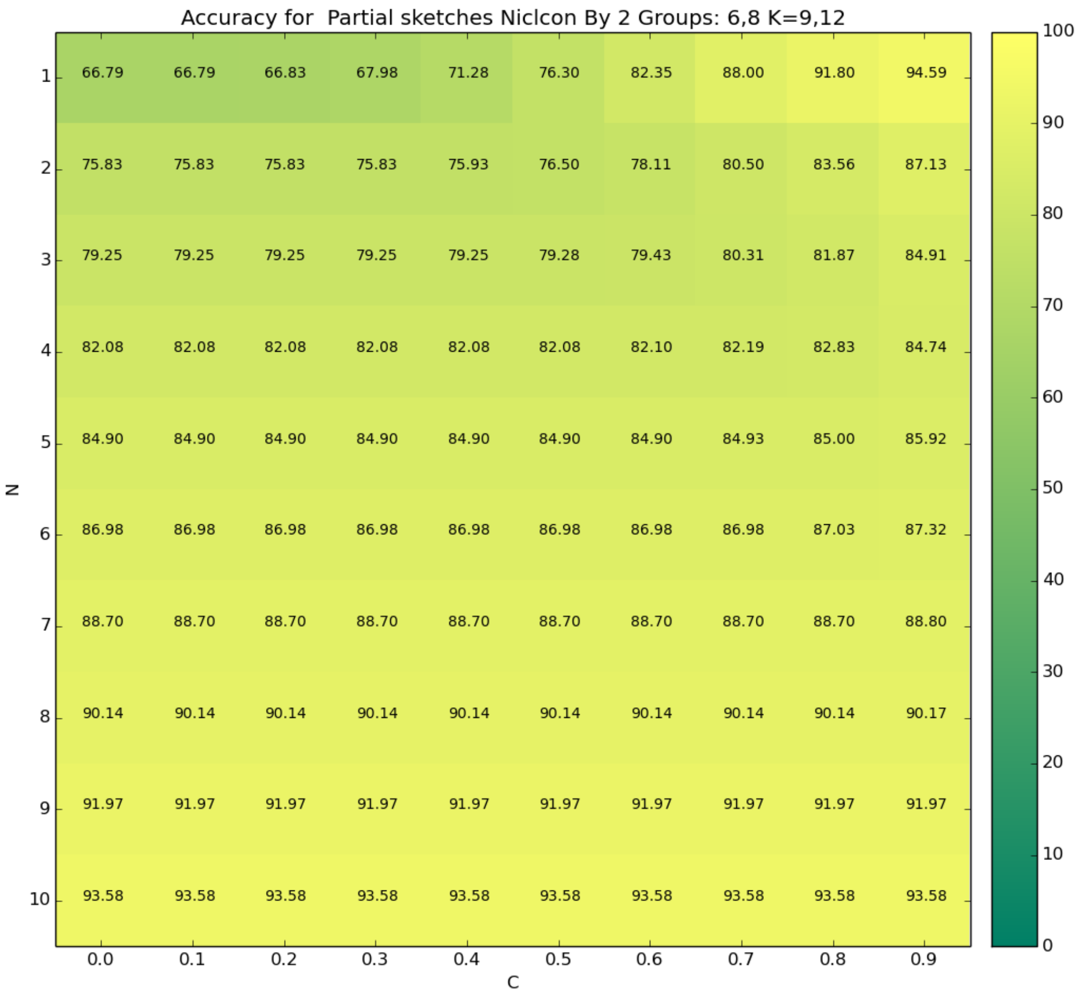
Test results that we have obtained by testing it on NicIcon dataset are given below, in figures *22* and *23*. While testing, the sketches were grouped by taking the representative of each class as the mean of full sketches within the class. As expected, when there is a single group, the accuracy results were identical to what we have obtained using the standard pipeline.

The map below shows accuracy results when the data classes were divided into 2 and 3 groups.



N \ C	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
1	82.46	82.46	82.54	83.90	87.14	92.05	95.36	97.18	98.60	99.14
2	91.09	91.09	91.09	91.09	91.22	91.96	93.56	95.33	97.27	98.76
3	93.61	93.61	93.61	93.61	93.61	93.67	93.99	94.73	96.12	98.12
4	94.85	94.85	94.85	94.85	94.85	94.85	94.91	95.08	95.68	97.49
5	96.02	96.02	96.02	96.02	96.02	96.02	96.02	96.06	96.26	97.07
6	96.55	96.55	96.55	96.55	96.55	96.55	96.55	96.55	96.61	96.99
7	97.11	97.11	97.11	97.11	97.11	97.11	97.11	97.11	97.11	97.23
8	97.55	97.55	97.55	97.55	97.55	97.55	97.55	97.55	97.55	97.54
9	97.89	97.89	97.89	97.89	97.89	97.89	97.89	97.89	97.89	97.90
10	98.37	98.37	98.37	98.37	98.37	98.37	98.37	98.37	98.37	98.37

Figure 22: Accuracy results of the alternative pipeline, on NicIcon full sketches (the sketches were divided into 2 groups)



N \ C	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
1	66.79	66.79	66.83	67.98	71.28	76.30	82.35	88.00	91.80	94.59
2	75.83	75.83	75.83	75.83	75.93	76.50	78.11	80.50	83.56	87.13
3	79.25	79.25	79.25	79.25	79.25	79.28	79.43	80.31	81.87	84.91
4	82.08	82.08	82.08	82.08	82.08	82.08	82.10	82.19	82.83	84.74
5	84.90	84.90	84.90	84.90	84.90	84.90	84.90	84.93	85.00	85.92
6	86.98	86.98	86.98	86.98	86.98	86.98	86.98	86.98	87.03	87.32
7	88.70	88.70	88.70	88.70	88.70	88.70	88.70	88.70	88.70	88.80
8	90.14	90.14	90.14	90.14	90.14	90.14	90.14	90.14	90.14	90.17
9	91.97	91.97	91.97	91.97	91.97	91.97	91.97	91.97	91.97	91.97
10	93.58	93.58	93.58	93.58	93.58	93.58	93.58	93.58	93.58	93.58

Figure 23: Accuracy results of the alternative pipeline, on NicIcon partial sketches (the sketches were divided into 2 groups)

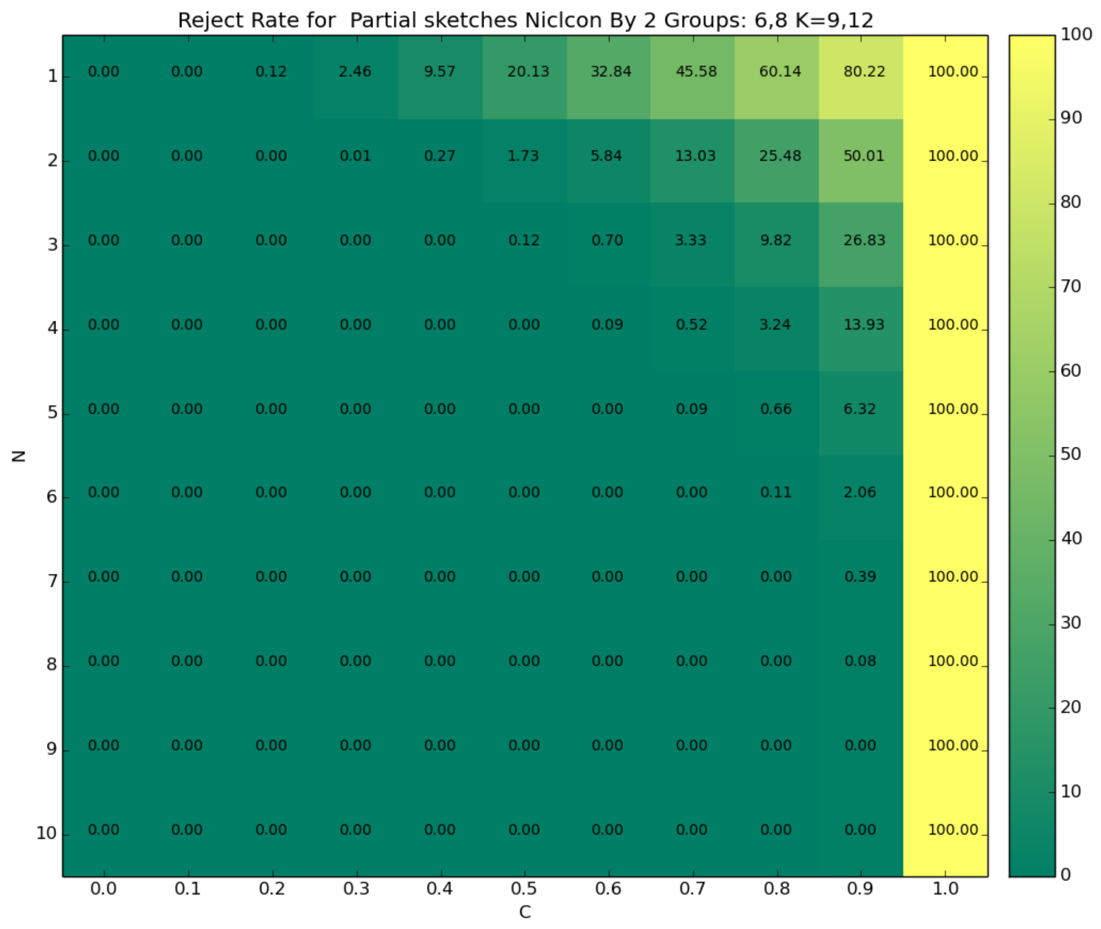
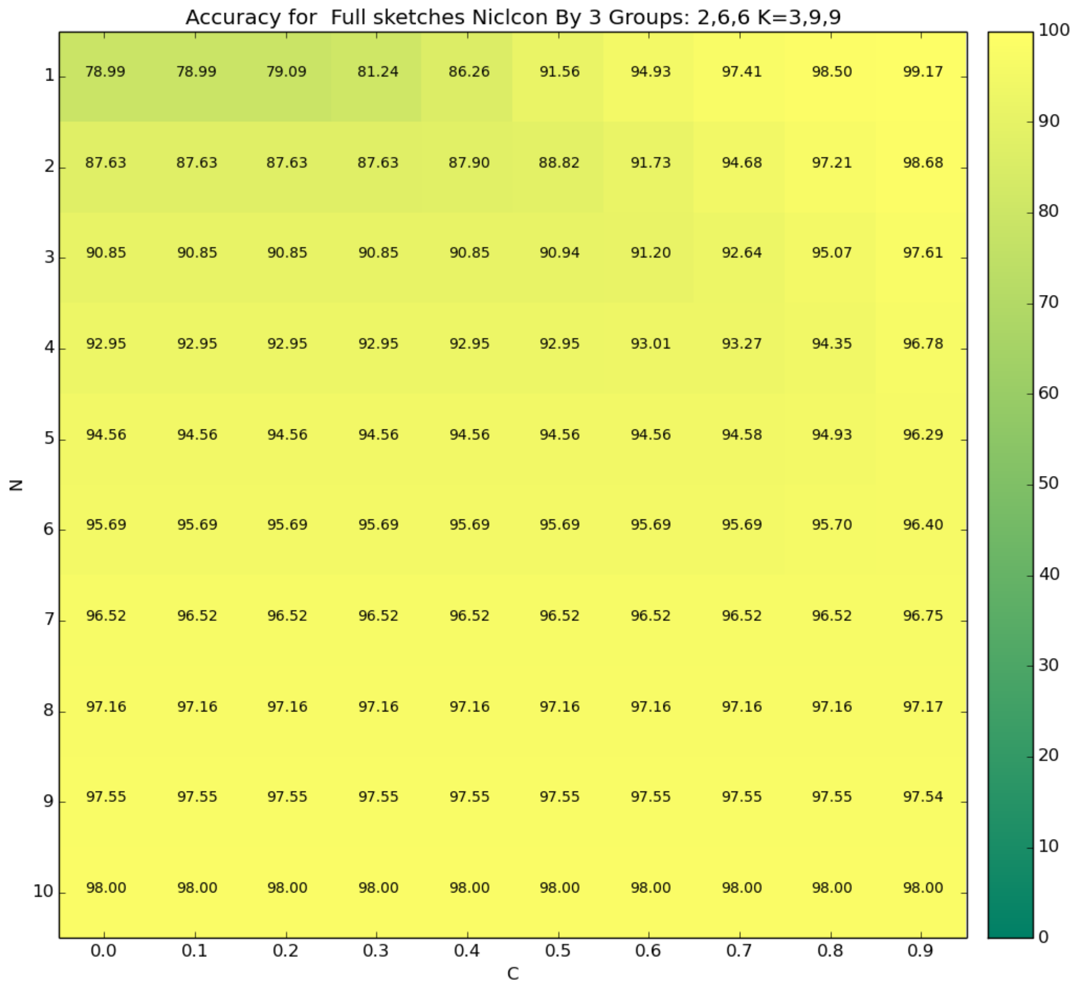


Figure 24: Reject rates of NicIcon partial sketches (the sketches were divided into 2 groups)

As seen in the heat maps above, compared to the original pipeline, the accuracies of this method were found to be less efficient. We believe in that the reason is that probabilities of being in a group should be computed using a different method rather than using the same approach as in the classical pipeline. Moreover, rejection rates are lower than the classical pipeline. This shows that the alternative system is more confident of the answers. Since we used standard predictor in every group, the most probable reason is that the calculation of the probability of the instance to be in groups which is calculated using the formula below

$$P(g_k|x) = e^{(-\|x-\mu_k\|^2 + min_k(\|x-\mu_k\|^2))/\sigma^2} P(g_k)/P(x)$$

as already explained.



N \ C	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
1	78.99	78.99	79.09	81.24	86.26	91.56	94.93	97.41	98.50	99.17
2	87.63	87.63	87.63	87.63	87.90	88.82	91.73	94.68	97.21	98.68
3	90.85	90.85	90.85	90.85	90.85	90.94	91.20	92.64	95.07	97.61
4	92.95	92.95	92.95	92.95	92.95	92.95	93.01	93.27	94.35	96.78
5	94.56	94.56	94.56	94.56	94.56	94.56	94.56	94.58	94.93	96.29
6	95.69	95.69	95.69	95.69	95.69	95.69	95.69	95.69	95.70	96.40
7	96.52	96.52	96.52	96.52	96.52	96.52	96.52	96.52	96.52	96.75
8	97.16	97.16	97.16	97.16	97.16	97.16	97.16	97.16	97.16	97.17
9	97.55	97.55	97.55	97.55	97.55	97.55	97.55	97.55	97.55	97.54
10	98.00	98.00	98.00	98.00	98.00	98.00	98.00	98.00	98.00	98.00

Figure 25: Accuracy results of the alternative pipeline, on NicIcon full sketches (the sketches were divided into 3 groups)

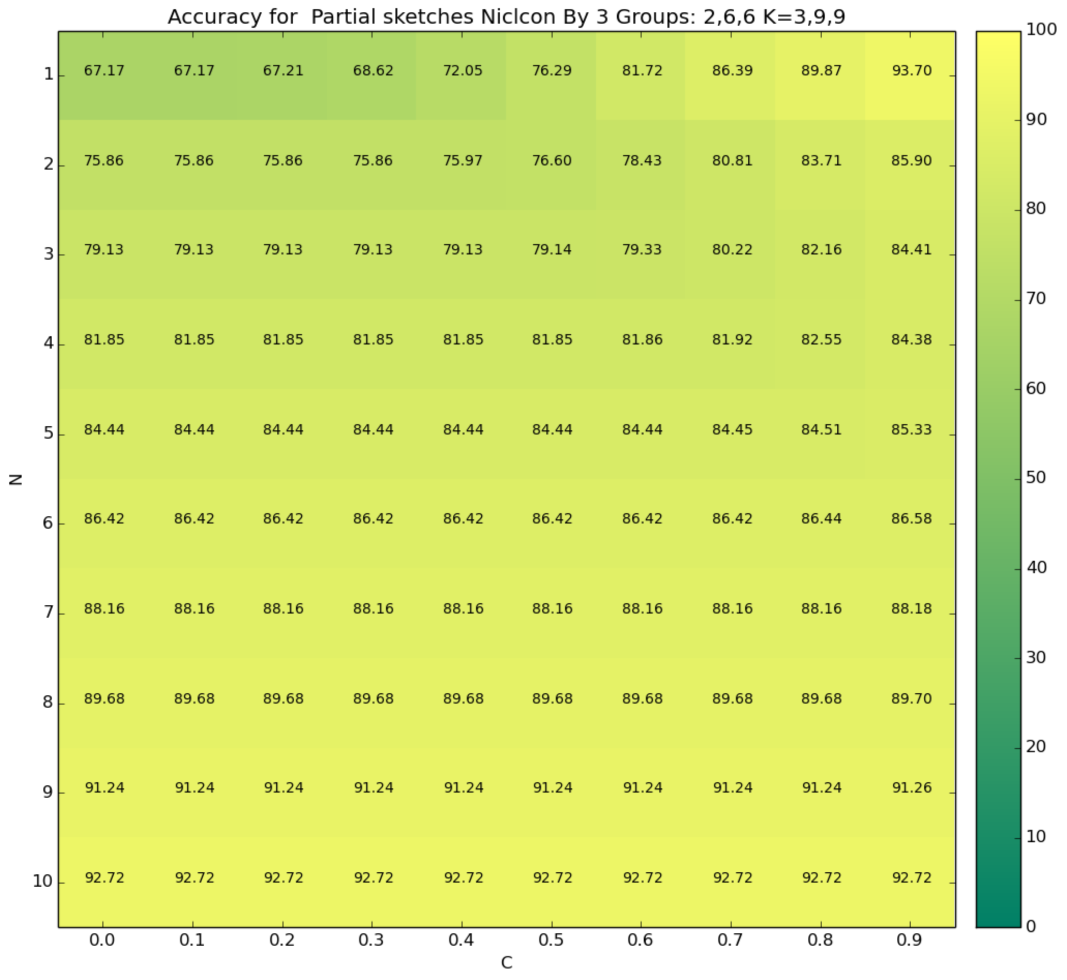


Figure 26: Accuracy results of the alternative pipeline, on NicIcon partial sketches (the sketches were divided into 3 groups)

Figures *25* and *26* show the partial accuracies when the sketches were divided into 3 groups. Compared to 2-grouping, the results of 3-grouping are slightly better. This indicates that the accuracy results heavily depend on how well classes can be grouped.

3.3.2 On Eitz Dataset

3.3.2.1 Standard Pipeline

We also tried to verify our implementation by running the same experiments as Altıok et. al. describes [14]. The results are illustrated below, in figures from *27* to *29*.

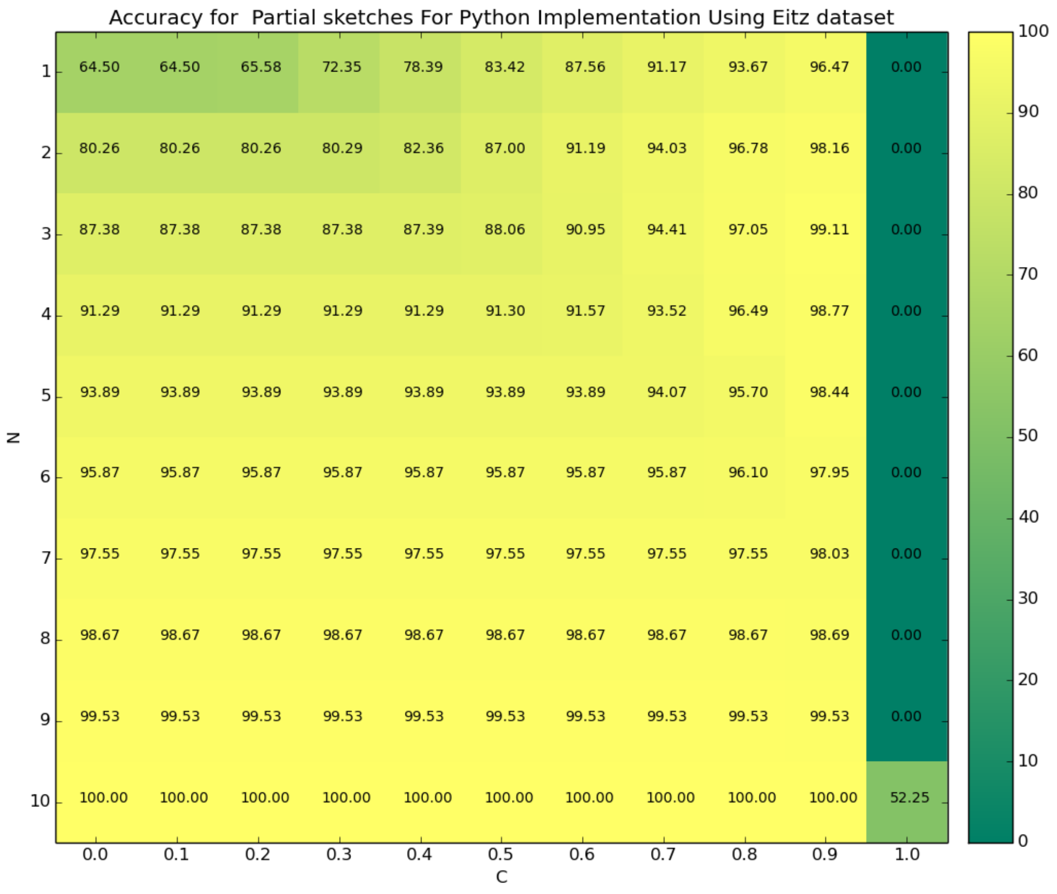


Figure 27: Eitz partial sketch accuracy results of the standard pipeline by running the same experiments as Altıok et. al. describes

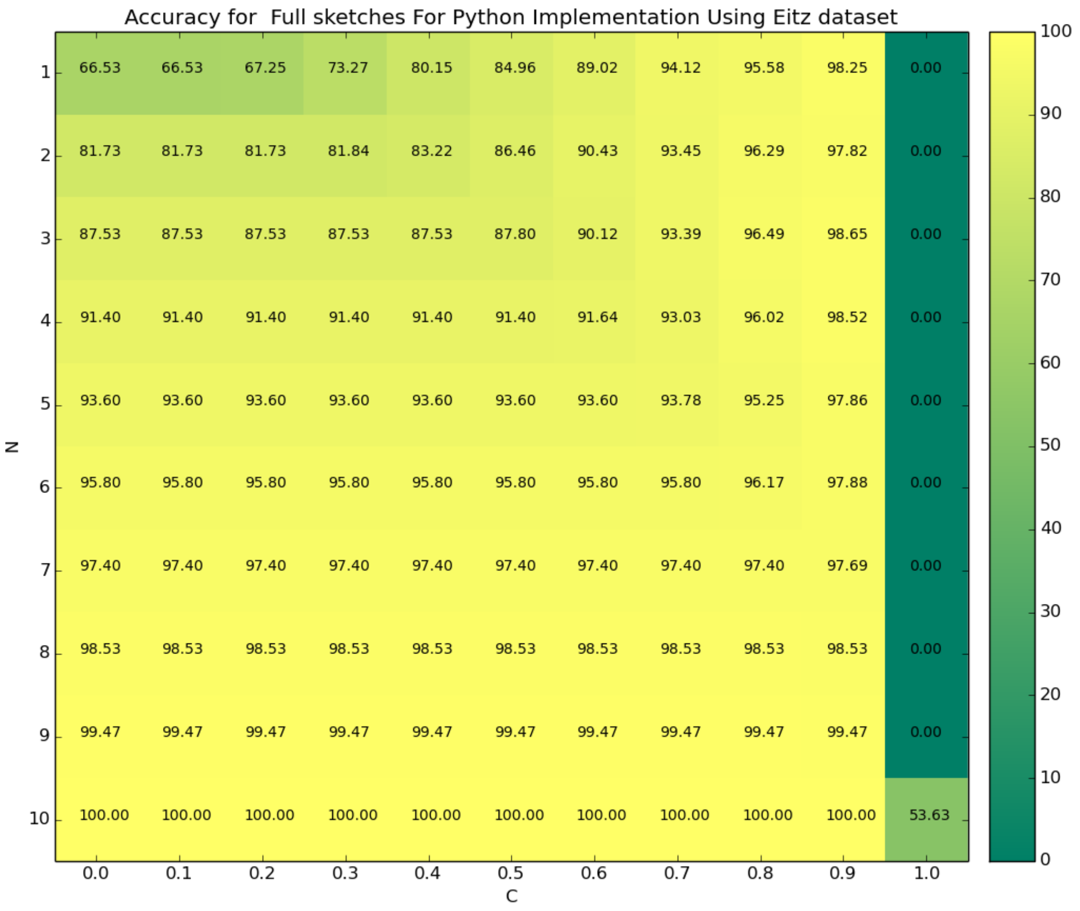
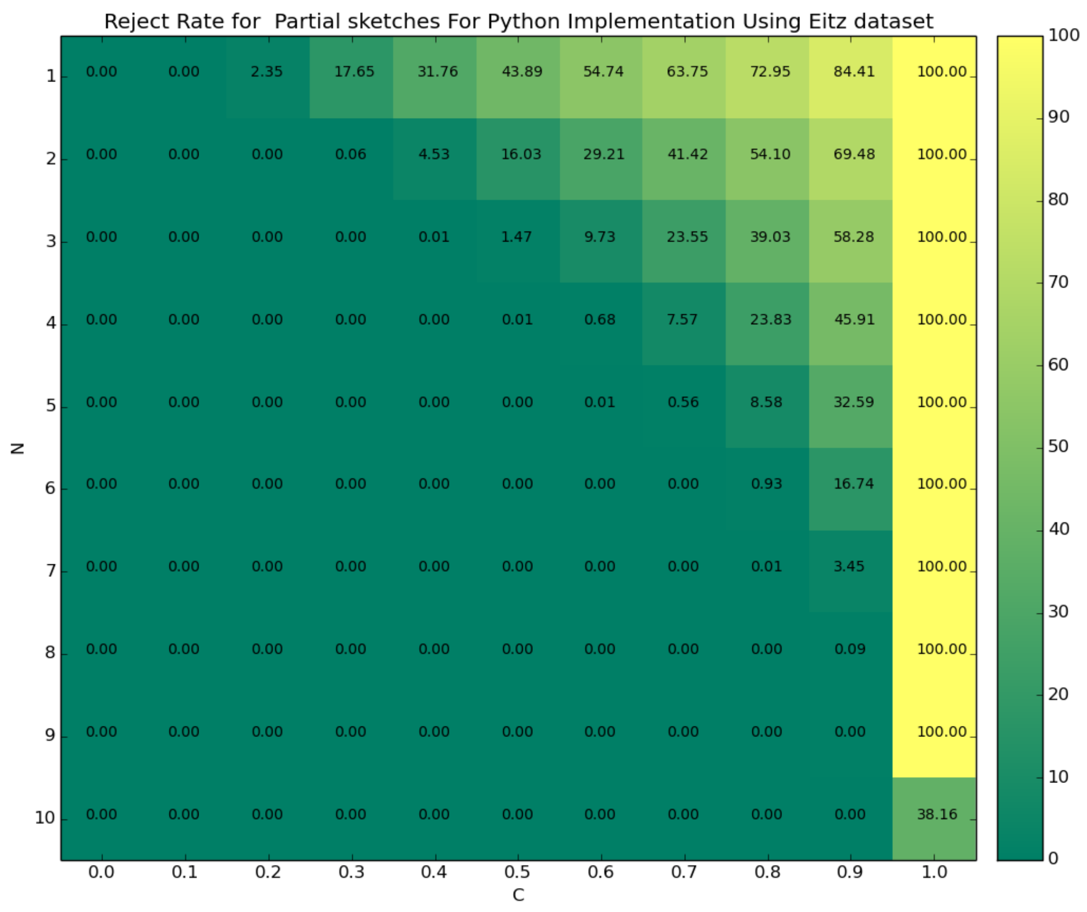


Figure 28: Eitz full sketch accuracy results of the standard pipeline by running the same experiments as Altıok et. al. describes

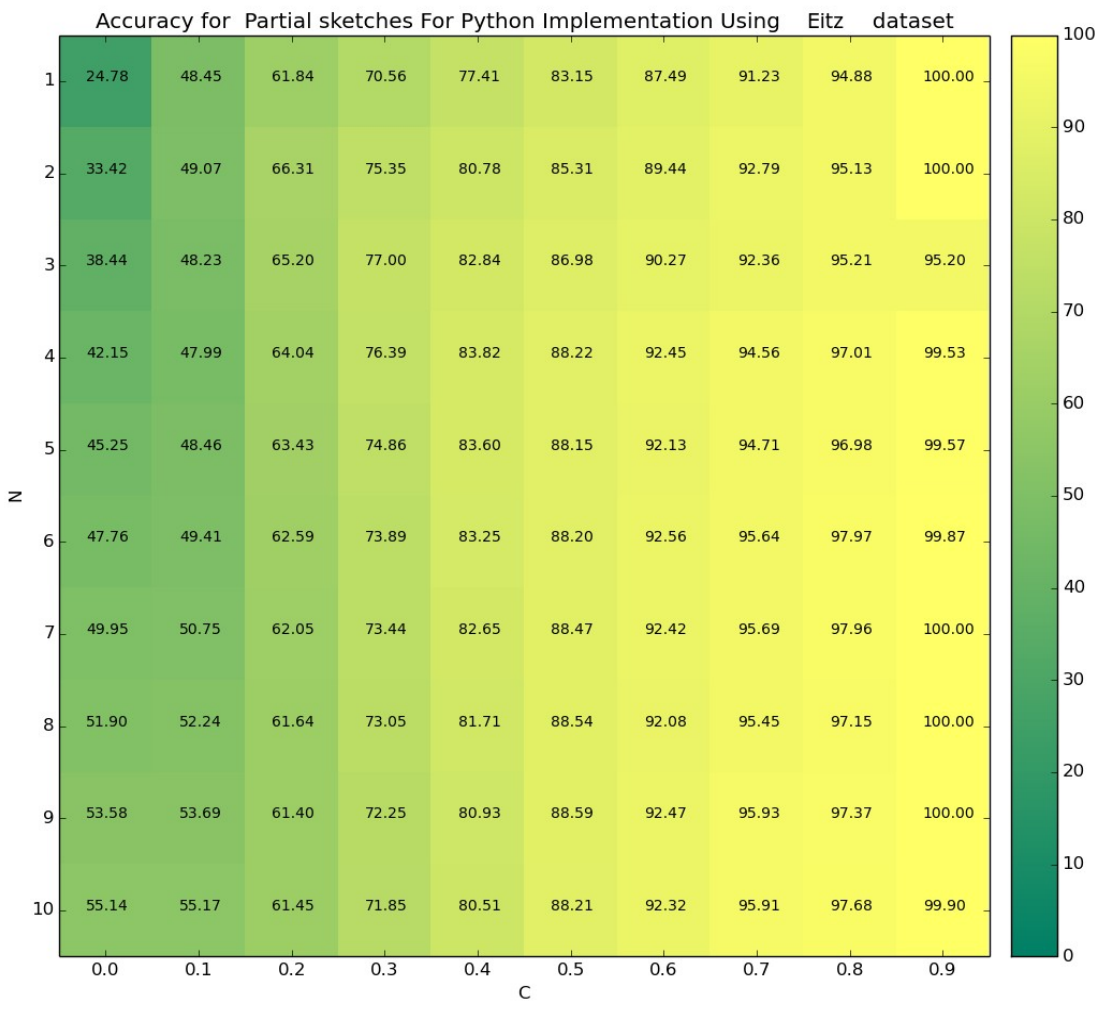


N \ C	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
1	0.00	0.00	2.35	17.65	31.76	43.89	54.74	63.75	72.95	84.41	100.00
2	0.00	0.00	0.00	0.06	4.53	16.03	29.21	41.42	54.10	69.48	100.00
3	0.00	0.00	0.00	0.00	0.01	1.47	9.73	23.55	39.03	58.28	100.00
4	0.00	0.00	0.00	0.00	0.00	0.01	0.68	7.57	23.83	45.91	100.00
5	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.56	8.58	32.59	100.00
6	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.93	16.74	100.00
7	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	3.45	100.00
8	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.09	100.00
9	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	100.00
10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	38.16

Figure 29: Eitz partial sketch reject rates of the standard pipeline by running the same experiments as Altıok et. al. describes

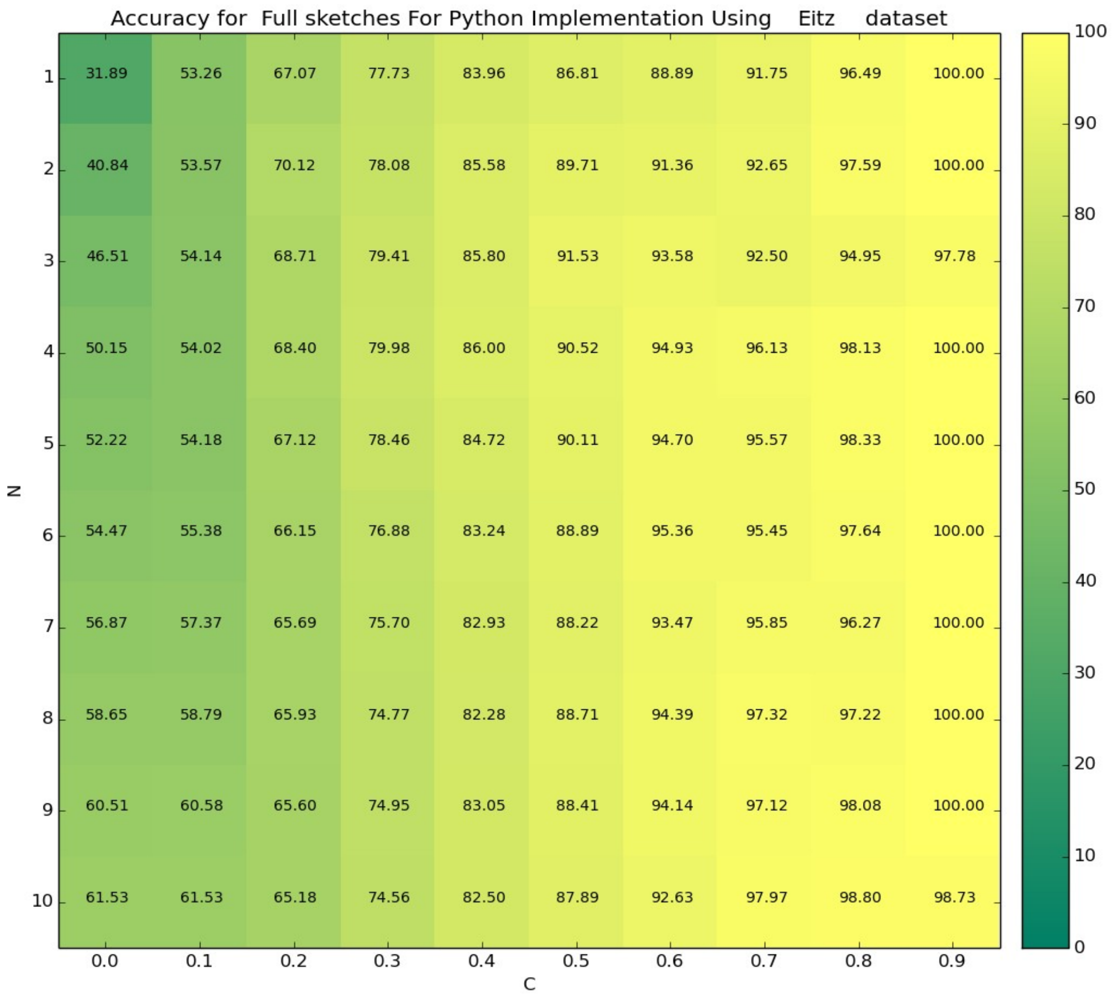
Compared to the results given in Altıok et. al.'s paper, there is a negligibly small difference, which can be explained by the fact that the experiment is divided into sub-experiments each of which has 10 randomly picked classes. The chance of having the same classes in every sub-experiment is very low.

We also tested the pipeline on the entire dataset. We took the first 70 instances of every class for training, and the last 10 for testing. When the partial sketches are generated, the number of instances used for testing becomes around 40.000 and SVM classification for a single instance takes around a second. We noticed that the time left after debugging the implementation was not going to be enough to use all of the sketches allocated for testing, then we randomly picked 5.000 of them. The results are given below, in figures from *30* to *33*.



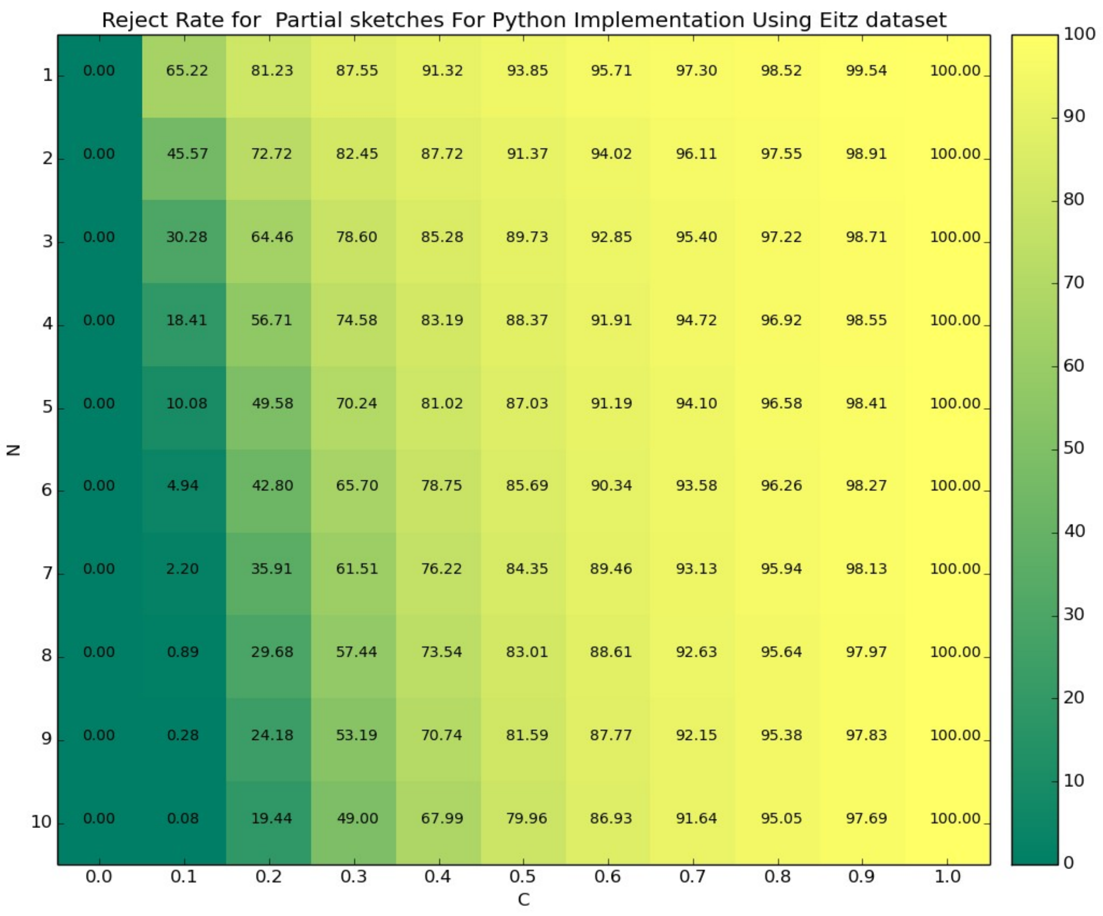
N \ C	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
1	24.78	48.45	61.84	70.56	77.41	83.15	87.49	91.23	94.88	100.00
2	33.42	49.07	66.31	75.35	80.78	85.31	89.44	92.79	95.13	100.00
3	38.44	48.23	65.20	77.00	82.84	86.98	90.27	92.36	95.21	95.20
4	42.15	47.99	64.04	76.39	83.82	88.22	92.45	94.56	97.01	99.53
5	45.25	48.46	63.43	74.86	83.60	88.15	92.13	94.71	96.98	99.57
6	47.76	49.41	62.59	73.89	83.25	88.20	92.56	95.64	97.97	99.87
7	49.95	50.75	62.05	73.44	82.65	88.47	92.42	95.69	97.96	100.00
8	51.90	52.24	61.64	73.05	81.71	88.54	92.08	95.45	97.15	100.00
9	53.58	53.69	61.40	72.25	80.93	88.59	92.47	95.93	97.37	100.00
10	55.14	55.17	61.45	71.85	80.51	88.21	92.32	95.91	97.68	99.90

Figure 30: Accuracy performance of the standard pipeline, on partial sketches of the whole Eitz dataset



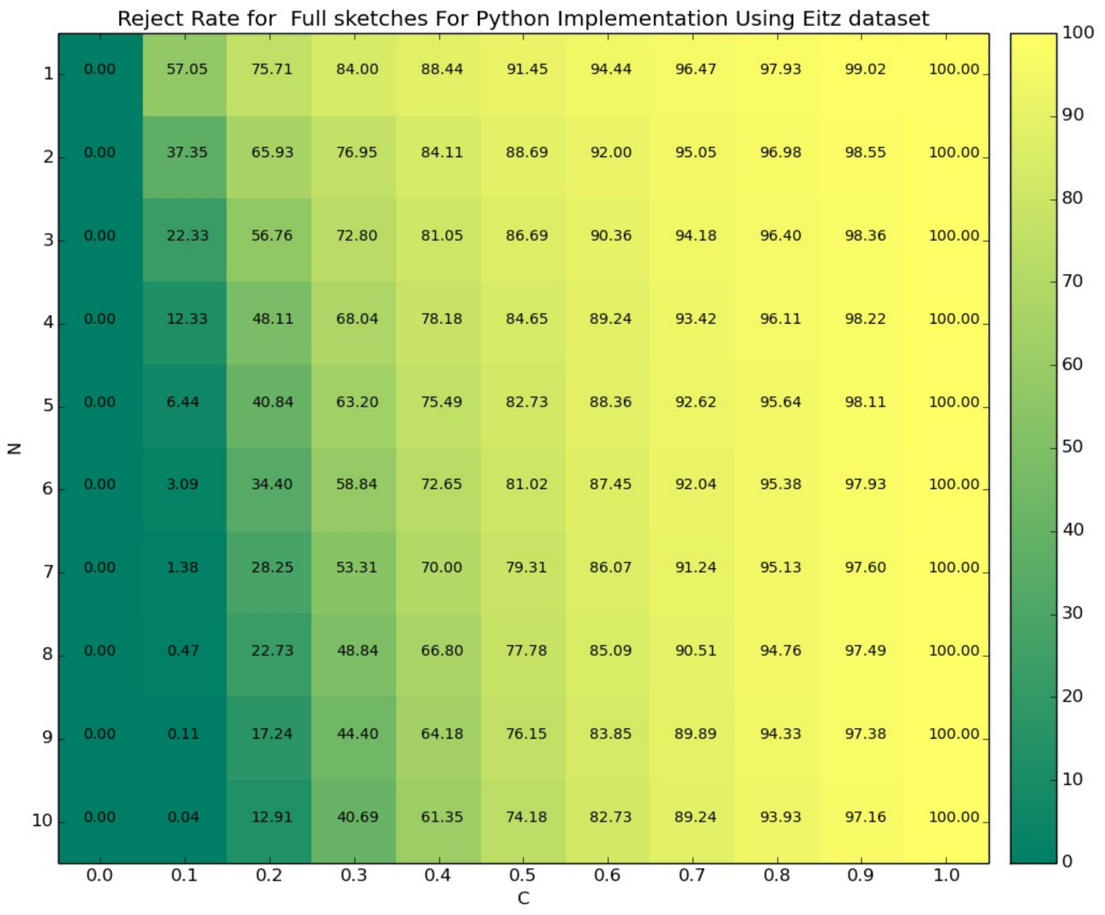
N \ C	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
1	31.89	53.26	67.07	77.73	83.96	86.81	88.89	91.75	96.49	100.00
2	40.84	53.57	70.12	78.08	85.58	89.71	91.36	92.65	97.59	100.00
3	46.51	54.14	68.71	79.41	85.80	91.53	93.58	92.50	94.95	97.78
4	50.15	54.02	68.40	79.98	86.00	90.52	94.93	96.13	98.13	100.00
5	52.22	54.18	67.12	78.46	84.72	90.11	94.70	95.57	98.33	100.00
6	54.47	55.38	66.15	76.88	83.24	88.89	95.36	95.45	97.64	100.00
7	56.87	57.37	65.69	75.70	82.93	88.22	93.47	95.85	96.27	100.00
8	58.65	58.79	65.93	74.77	82.28	88.71	94.39	97.32	97.22	100.00
9	60.51	60.58	65.60	74.95	83.05	88.41	94.14	97.12	98.08	100.00
10	61.53	61.53	65.18	74.56	82.50	87.89	92.63	97.97	98.80	98.73

Figure 31: Accuracy performance of the standard pipeline, on full sketches of the whole Eitz dataset



N \ C	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
1	0.00	65.22	81.23	87.55	91.32	93.85	95.71	97.30	98.52	99.54	100.00
2	0.00	45.57	72.72	82.45	87.72	91.37	94.02	96.11	97.55	98.91	100.00
3	0.00	30.28	64.46	78.60	85.28	89.73	92.85	95.40	97.22	98.71	100.00
4	0.00	18.41	56.71	74.58	83.19	88.37	91.91	94.72	96.92	98.55	100.00
5	0.00	10.08	49.58	70.24	81.02	87.03	91.19	94.10	96.58	98.41	100.00
6	0.00	4.94	42.80	65.70	78.75	85.69	90.34	93.58	96.26	98.27	100.00
7	0.00	2.20	35.91	61.51	76.22	84.35	89.46	93.13	95.94	98.13	100.00
8	0.00	0.89	29.68	57.44	73.54	83.01	88.61	92.63	95.64	97.97	100.00
9	0.00	0.28	24.18	53.19	70.74	81.59	87.77	92.15	95.38	97.83	100.00
10	0.00	0.08	19.44	49.00	67.99	79.96	86.93	91.64	95.05	97.69	100.00

Figure 32: Reject rates of the standard pipeline, on partial sketches of the whole Eitz dataset

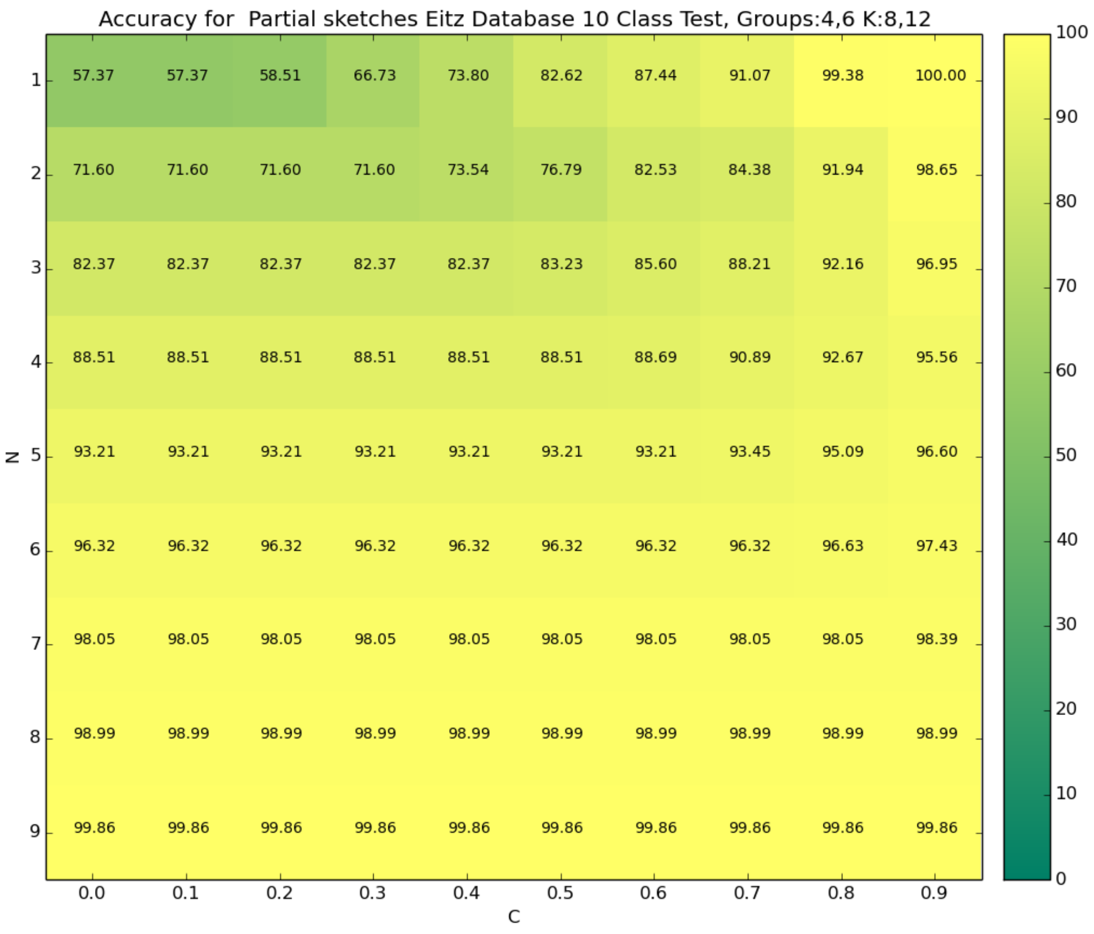


N \ C	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
1	0.00	57.05	75.71	84.00	88.44	91.45	94.44	96.47	97.93	99.02	100.00
2	0.00	37.35	65.93	76.95	84.11	88.69	92.00	95.05	96.98	98.55	100.00
3	0.00	22.33	56.76	72.80	81.05	86.69	90.36	94.18	96.40	98.36	100.00
4	0.00	12.33	48.11	68.04	78.18	84.65	89.24	93.42	96.11	98.22	100.00
5	0.00	6.44	40.84	63.20	75.49	82.73	88.36	92.62	95.64	98.11	100.00
6	0.00	3.09	34.40	58.84	72.65	81.02	87.45	92.04	95.38	97.93	100.00
7	0.00	1.38	28.25	53.31	70.00	79.31	86.07	91.24	95.13	97.60	100.00
8	0.00	0.47	22.73	48.84	66.80	77.78	85.09	90.51	94.76	97.49	100.00
9	0.00	0.11	17.24	44.40	64.18	76.15	83.85	89.89	94.33	97.38	100.00
10	0.00	0.04	12.91	40.69	61.35	74.18	82.73	89.24	93.93	97.16	100.00

Figure 33: Reject rates of the standard pipeline, on full sketches of the whole Eitz dataset

3.3.2.2 Alternative Pipeline

For evaluating the alternative pipeline, we run several tests on Eitz dataset (the experiment described by Altıok et. al., testing on the whole dataset). The sketches were divided into 5 groups. The results are given below.



N \ C	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
1	57.37	57.37	58.51	66.73	73.80	82.62	87.44	91.07	99.38	100.00
2	71.60	71.60	71.60	71.60	73.54	76.79	82.53	84.38	91.94	98.65
3	82.37	82.37	82.37	82.37	82.37	83.23	85.60	88.21	92.16	96.95
4	88.51	88.51	88.51	88.51	88.51	88.51	88.69	90.89	92.67	95.56
5	93.21	93.21	93.21	93.21	93.21	93.21	93.21	93.45	95.09	96.60
6	96.32	96.32	96.32	96.32	96.32	96.32	96.32	96.32	96.63	97.43
7	98.05	98.05	98.05	98.05	98.05	98.05	98.05	98.05	98.05	98.39
8	98.99	98.99	98.99	98.99	98.99	98.99	98.99	98.99	98.99	98.99
9	99.86	99.86	99.86	99.86	99.86	99.86	99.86	99.86	99.86	99.86

Figure 34: Accuracy performance of the alternative implementation by running the experiments Altıok et. al. describes, on partial sketches of Eitz dataset

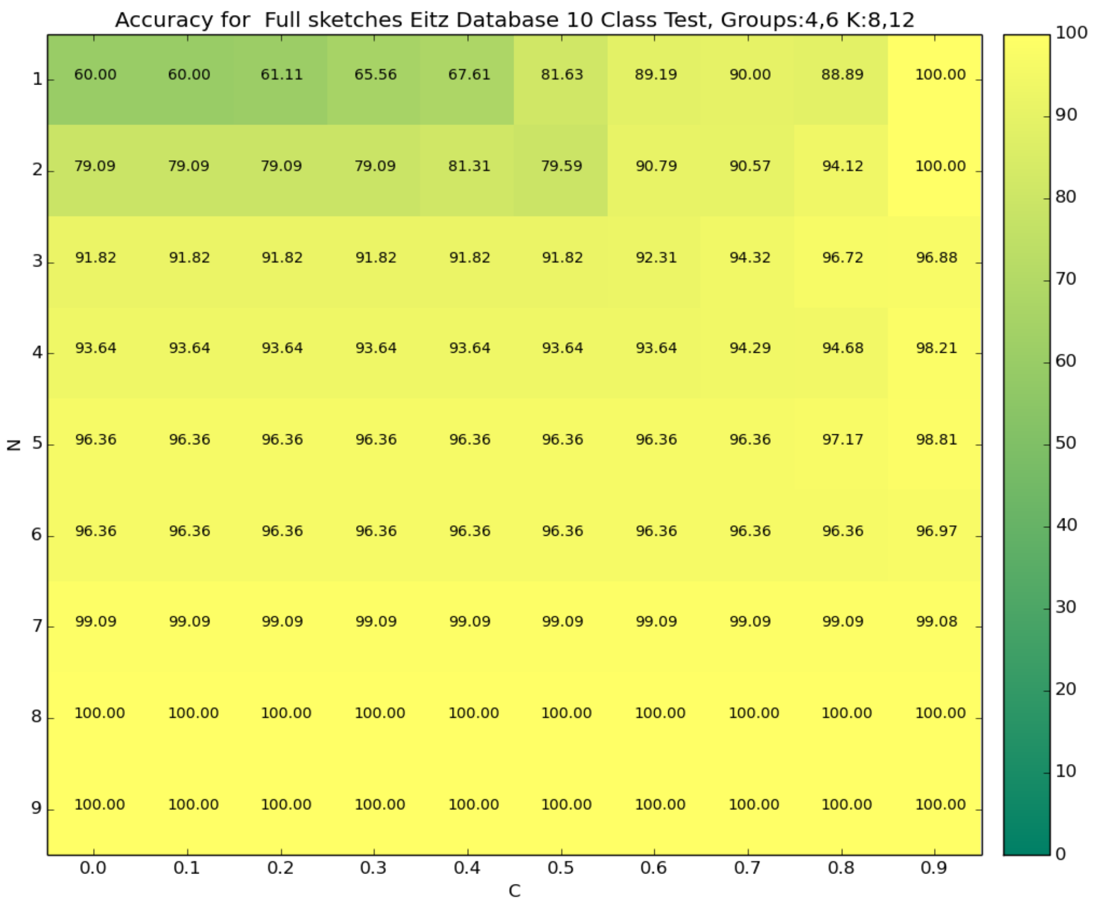


Figure 35: Accuracy performance of the alternative implementation by running the experiments Altıok et. al. describes, on full sketches of Eitz dataset

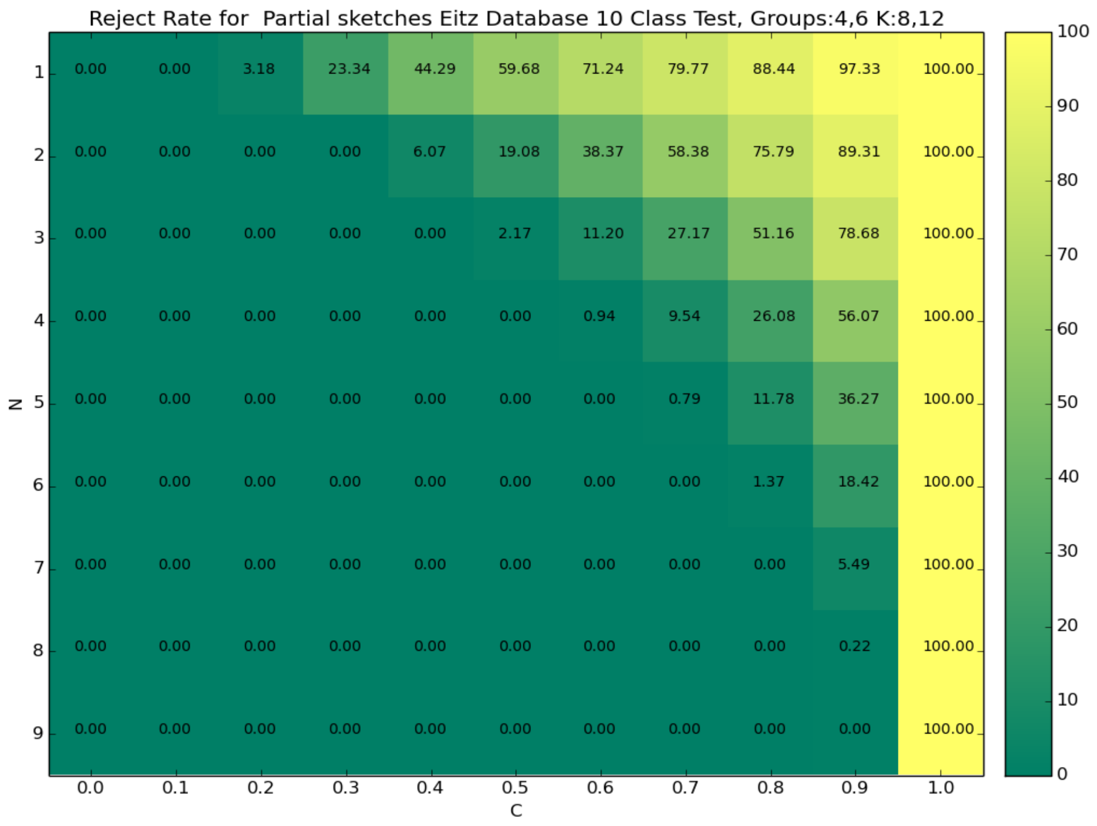


Figure 36: Reject rates of the alternative implementation by running the experiments Altıok et. al. describes, on partial sketches of Eitz dataset

As seen in figures from *34* to *36*, the alternative implementation has nearly caught the performance of the original implementation (see *3.3.2.1* for details), but couldn't beat it. We believe in that a more robust grouping method will improve the results and beat the accuracy performance of the standard one.

Moreover, we also tested the accuracy performance of this pipeline on the whole Eitz dataset, in the same way as described in 3.3.2.1. The results of that experiment are given below.

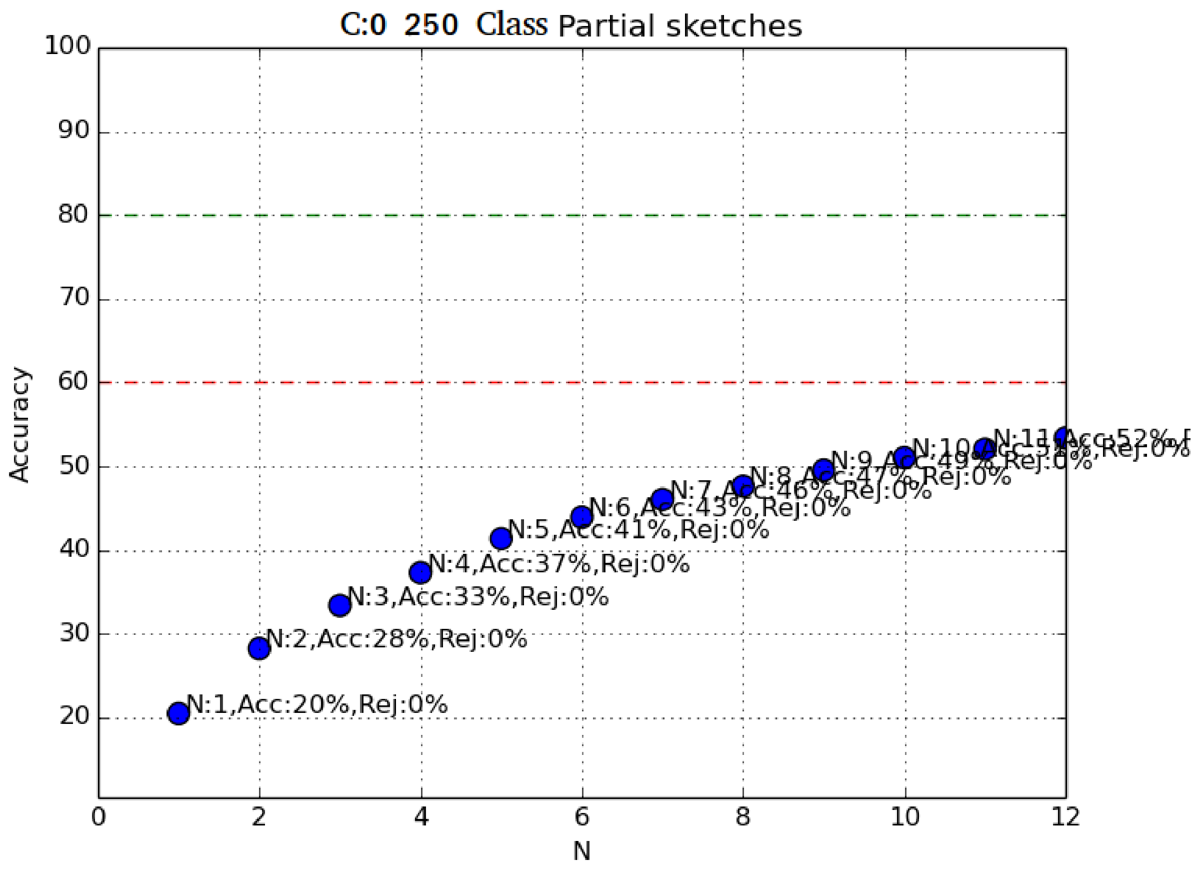


Figure 37: Accuracy performance of the alternative pipeline on partial sketches of the whole Eitz dataset, when C (confidence threshold) is set to 0

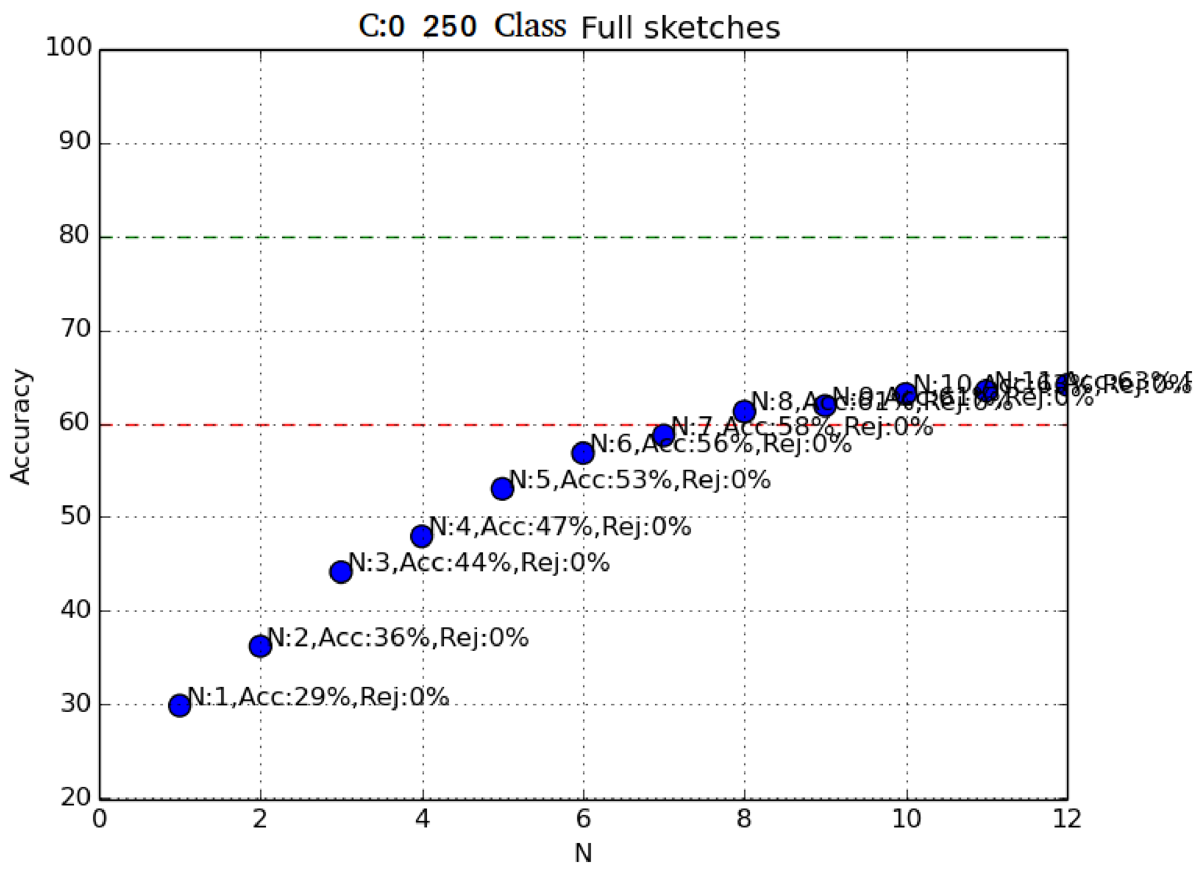


Figure 38: Accuracy performance of the alternative pipeline on full sketches of the whole Eitz dataset, when C (confidence threshold) is set to 0

As seen in figures *37* and 38, the results are lower than the standard pipeline, with a small difference. However, using different group numbers or grouping techniques may improve the accuracies. Actually we aimed to finally try out different number of groups, but we didn't have sufficient time for testing and hence reduced the scale such that we are able to repeat the experiments numerous times in order to see the effect of number of groups by changing the value every repetition. We only used the instances of first 40 classes. For each class, we chose the first 70 full instances for training, and the last 10 for testing. Results of this experiment are given below, in a single figure.

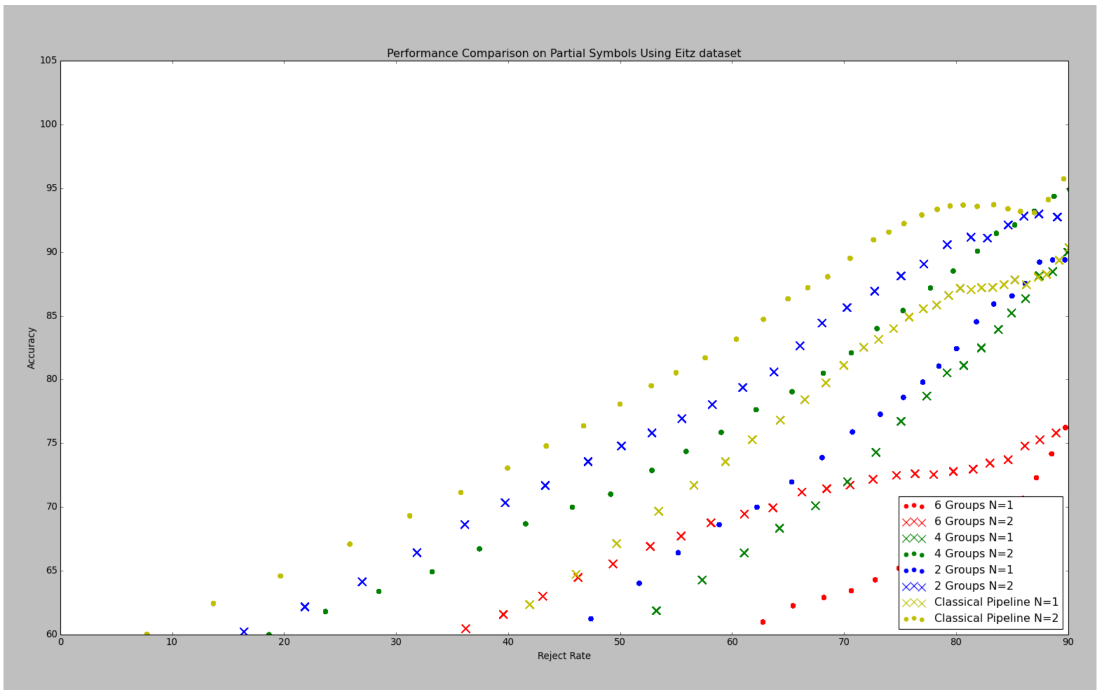


Figure 39: Trying different number of groups out in our alternative pipeline (6,4 and 2)

As seen in the figure above, increasing the number of groups deteriorates the classification performance, but improves confidence of the classifier. These results encourages us to try out different grouping techniques, however, since we didn't have enough time in the summer research period, we couldn't do a deeper research on it. We have also pointed out this issue in *Future Work & Conclusion* section.

3.4 Android UI for Demonstration

We developed an application that provides users an Android graphical interface to try out the auto-completion pipeline in real life. The application has basic features as shown in figure *40*.

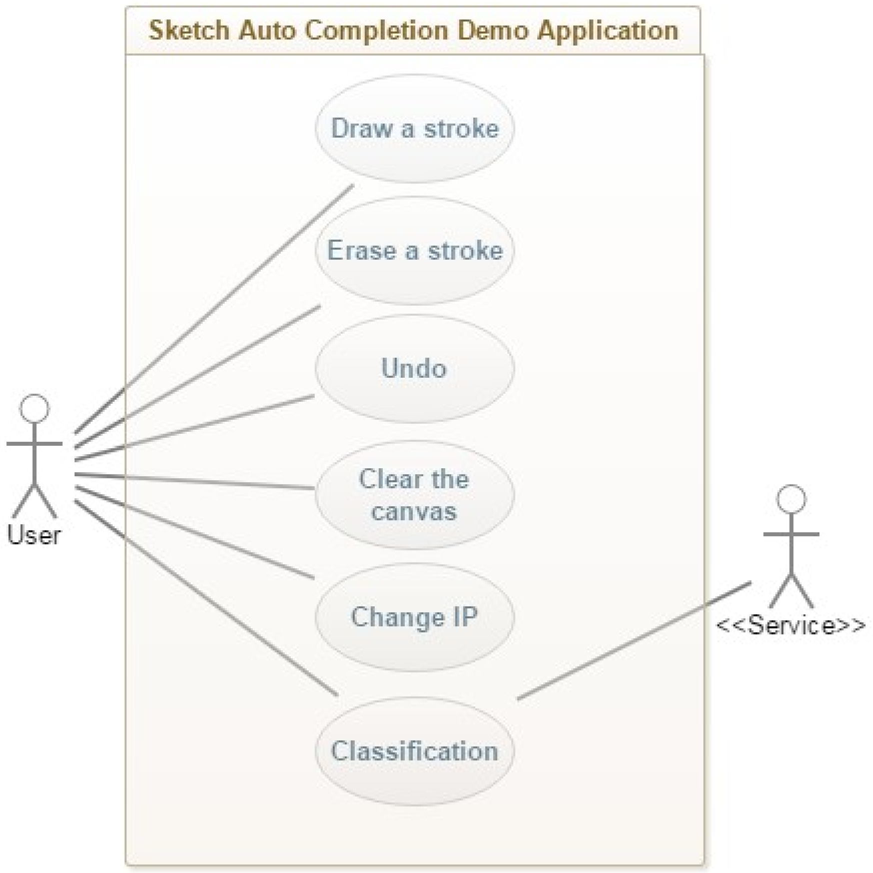


Figure 40: Use case diagram of the application

3.4.1 How to Use

As seen in figure *41*, there are four circle buttons on the right side of the screen and there is scroll bar on the left side of the screen. Unless user clicks on one of those buttons, application will continue to draw whatever user draws on the screen.

If user clicks **undo** button after drawing a stroke, the recently drawn stroke will be removed and class names with their probabilities will be recalculated as shown in figure *41*.

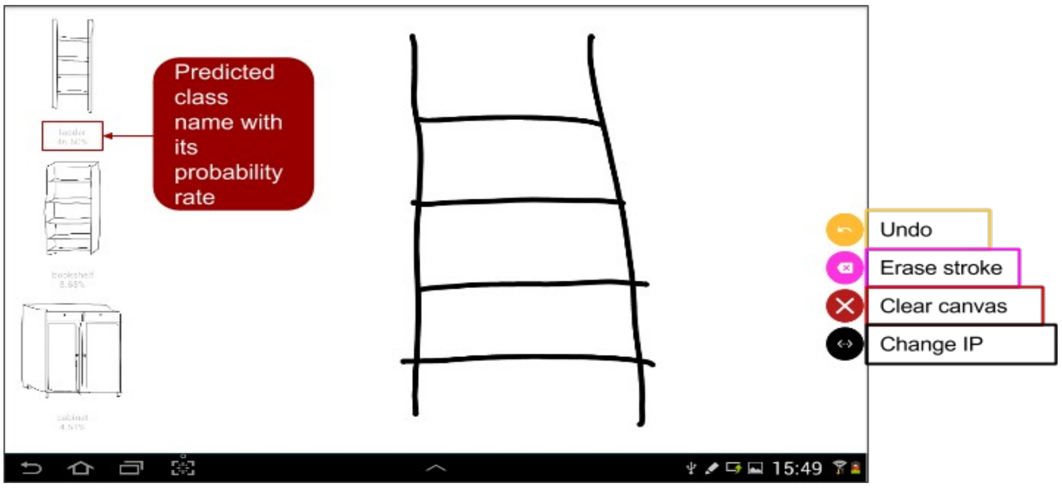


Figure 41: A screen-shot after ladder is drawn

If user clicks one of the pictures on the scroll bar the picture will be shown on the screen as shown in figure *42*.

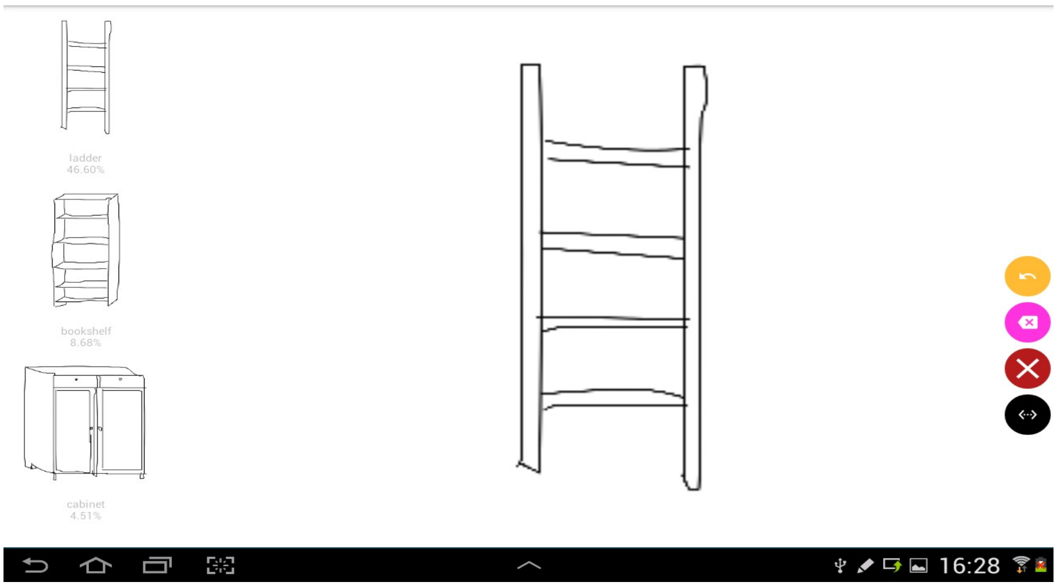


Figure 42: A screen-shot after one of the images on the scroll bar is clicked

User can also remove strokes by choosing the stroke, after s/he clicked on **erase stroke** button. For instance, as seen in figure *43*, user can click on this button and then can pick a stroke to remove. Having clicked this button, class probabilities of the remaining sketch will be recalculated. Afterwards, scroll bar will be refreshed as seen in figure *44*.

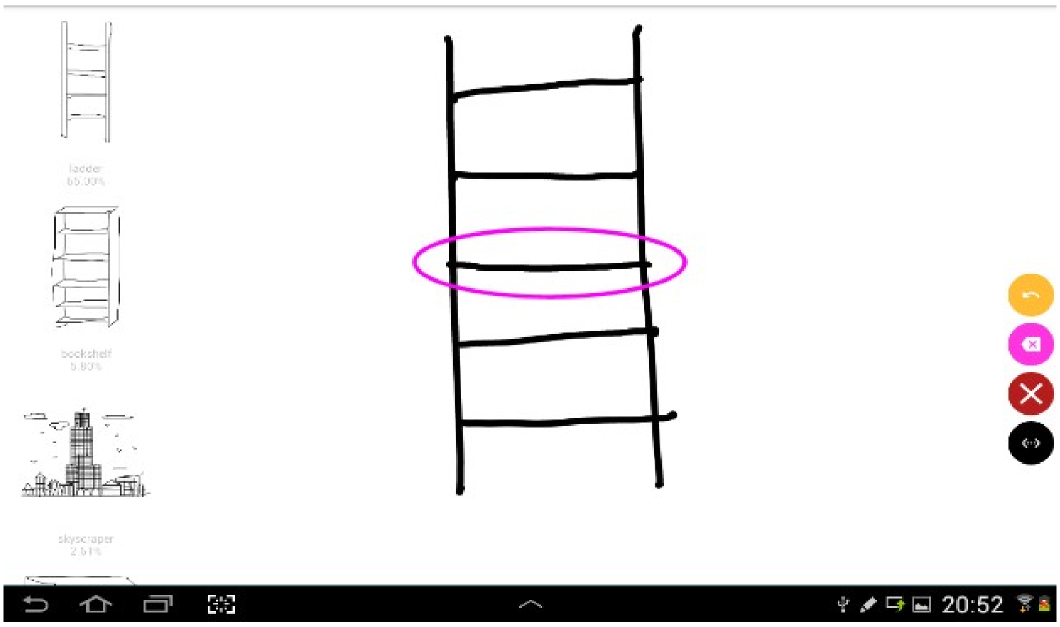


Figure 43: A screen-shot after a ladder is drawn on the canvas

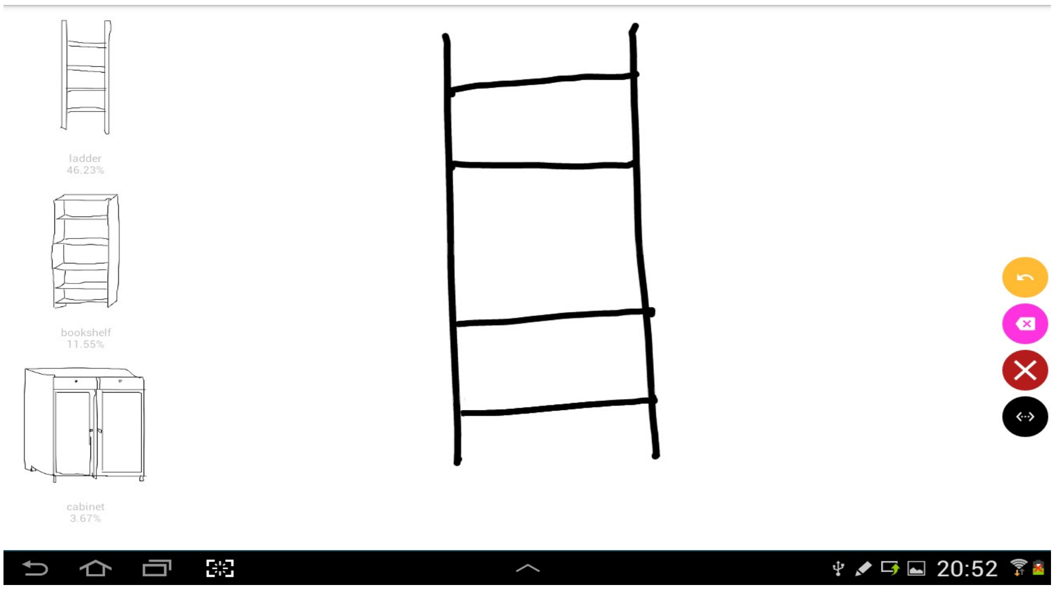


Figure 44: A screen-shot after highlighted stroke in figure 43 is removed.

Moreover, user can reset IP address of the machine where the classification program is running by clicking **change IP** button. Once this button is clicked, a pop-up window will appear to input a new IP address, as seen in figure 45. With the aid of that property, IP address of the classification server can be changed without digging into the source code and building the application again.

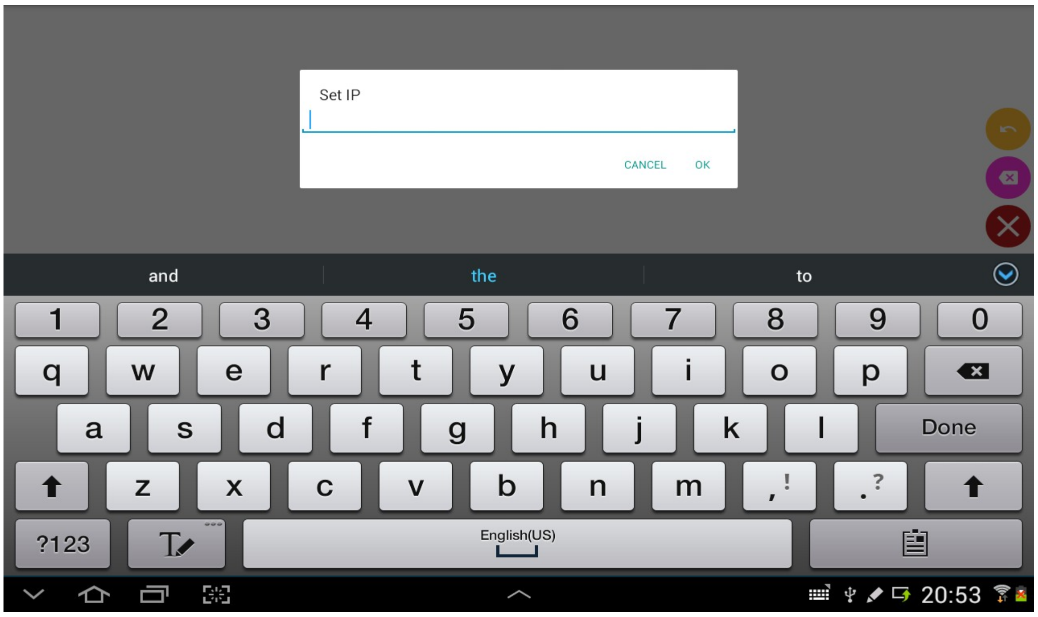


Figure 45: A screen-shot after change IP button is clicked.

3.4.2 Implementation

3.4.2.1 Drawing Strokes

In this application, user can draw a sketch on the canvas. The application receives the sketch and converts it into JSON format, the textual way of representing a sketch. Every time a sketch has been changed, application sends JSON of this sketch to the server.

Every time user draws a stroke on the screen or removes a stroke from the sketch, the JSON string of the entire sketch is updated and sent to the server as described in *3.4.2.6*.

3.4.2.2 Erasing Strokes

In this application, user can modify the sketch by removing particular stroke(s). We consider an array including all of the strokes. When user pushes **erase** button and chooses the stroke to remove, application finds this stroke on the list and removes it. This functionality allows users to remove any stroke.

3.4.2.3 Undo

Another functionality is keeping a stack of all the actions taken by users, e.g. adding or removing a stroke. The last action can be undone by pushing and removing this action from the stack, going back to the sketch which was drawn before the action and classifying it once more.

3.4.2.4 Clearing Canvas

Another feature is that the application enables users to clear the canvas via **clear all** button. Once this button is clicked, results in the scroll bar are also cleared. Additionally, the objects referring to the sketch abstraction, the drawing canvas and the stack keeping the actions are reset.

3.4.2.5 Classification

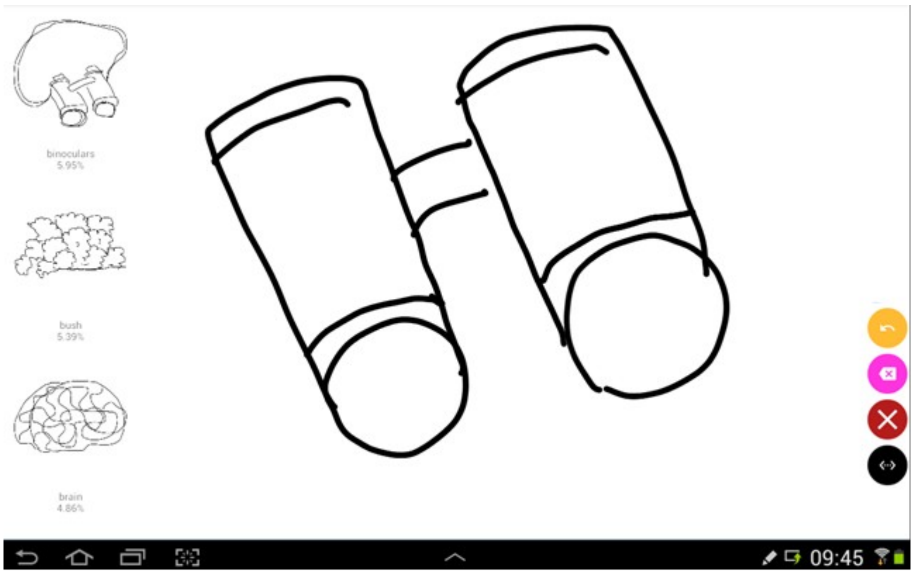
The application communicates with HTTP server via a service that we implemented. One of the most significant features of the application is that it can give suggestions in real time. We implemented a service that communicates with the server in the background and provides fast responses to users. Moreover, every time user updates the sketch, the recent sketch is sent to the server using asynchronous threads. Server classifies the sketch and tries to recognize it through the standard auto-completion pipeline. Having classified the sketch, server sends top 5 class names with their probabilities. The application receives names and possibilities of these classes and shows them to the user on the scroll view. User can choose one of these classes by clicking on its representative class image. When an image is clicked, the sketch in canvas gets replaced by that image.

3.4.2.6 HTTP Communication

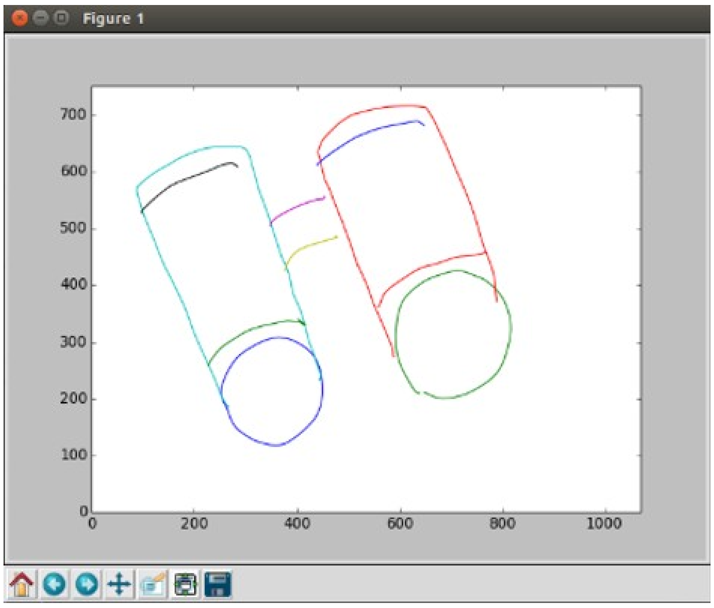
3.4.2.6.1 Implementation

Every time the sketch is updated by adding/removing a stroke, the resulting sketch on the canvas is sent to the server for classification. When the classification is done, the most probable 5 classes with their probabilities is sent back to the application. Since a prediction task takes less than a second, all these events happen in real time. We preferred Hyper Text Transfer Protocol [15] to maintain the communication between application and server.

To implement the server-side of communication, we chose Flask [16] as the HTTP server framework in Python. To make sure the sketch is coming to the server with no corruption, before sending the sketch to the classifier, it is also visualized on the screen of the machine where the server is running.



a



b

Figures 46: A binoculars drawn on the client application (a) and its visualization on the server (b)

3.4.2.6.2 Sketch Abstraction

A **Sketch** object consists of **Stroke** objects that are kept in an **ArrayList**, and sketch ID. **Sketch** class implements **JSONable** interface in order to convert sketch object into JSON string. However, while testing to program we realized that every time creating new JSON string to send to the server is time-consuming. Therefore, the

JSON string of strokes is kept in the sketch and after every update operation this JSON string is also updated. With the aid of this, application can give faster response to users, since JSON string is not recreated every time but updated.

Moreover, **Strokes** of a **Sketch** consist of **ArrayList**s that store **Point** objects and stroke ID. Moreover, **Point** objects keep coordinates, point IDs and time information. **Stroke** and **Point** classes implement **JSONable** interface to convert their data into JSON string.

A diagram explaining the abstraction and an example representation in JSON are given below.

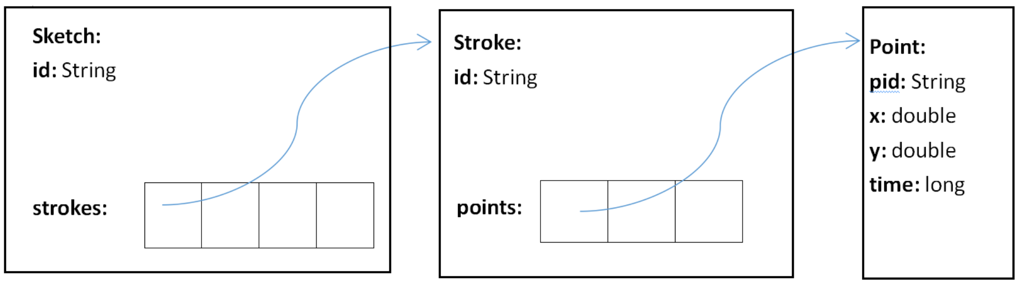
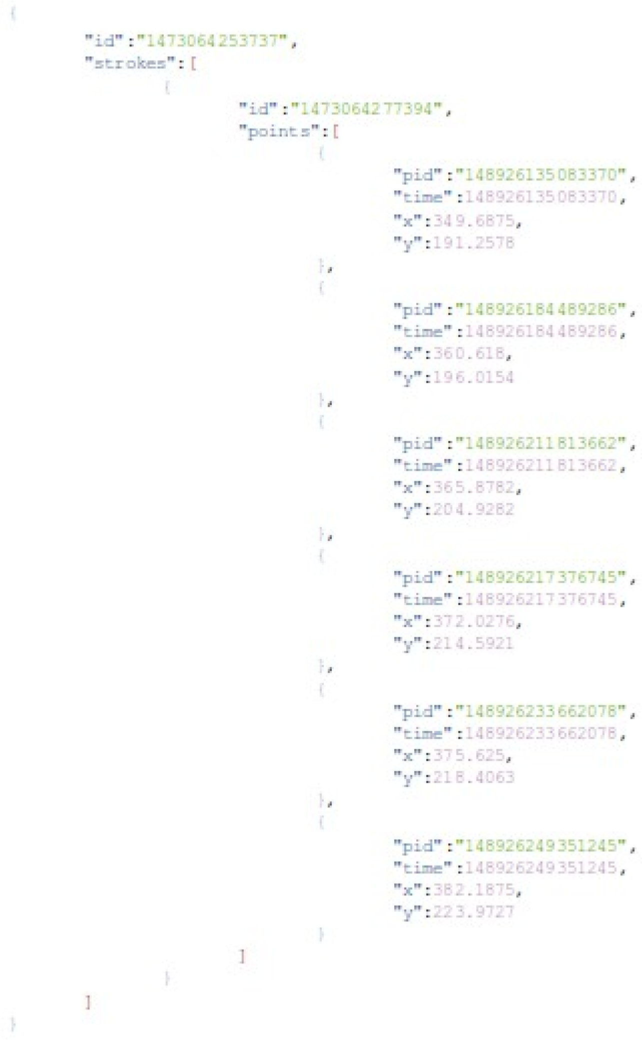


Figure 47: A diagram explaining the sketch abstraction



```
{
        "id":"1473064253737",
        "strokes":[
                {
                        "id":"1473064277394",
                        "points":[
                                {
                                        "pid":"148926135083370",
                                        "time":148926135083370,
                                        "x":349.6875,
                                        "y":191.2578
                                },
                                {
                                        "pid":"148926184489286",
                                        "time":148926184489286,
                                        "x":360.618,
                                        "y":196.0154
                                },
                                {
                                        "pid":"148926211813662",
                                        "time":148926211813662,
                                        "x":365.8782,
                                        "y":204.9282
                                },
                                {
                                        "pid":"148926217376745",
                                        "time":148926217376745,
                                        "x":372.0276,
                                        "y":214.5921
                                },
                                {
                                        "pid":"148926233662078",
                                        "time":148926233662078,
                                        "x":375.625,
                                        "y":218.4063
                                },
                                {
                                        "pid":"148926249351245",
                                        "time":148926249351245,
                                        "x":382.1875,
                                        "y":223.9727
                                }
                        ]
                }
        ]
}
```

Figure 48: An example of JSON string of a sketch

3.5 Integration into iMotion

To integrate our system into iMotion, we firstly modified the UI in a way that it also records the sketch in the abstraction discussed in 3.4.2.6.2. Secondly, the standard implementation of auto-completion has been integrated to the sketch server, which is the back-end for partial sketch classification. Finally, we have implemented the communication between the UI and the sketch server in a way that the UI sends both image and JSON version of the sketches in frames and presents the results of the recent (created by University of Basel) and our classification pipeline. The details of the integration are given in the subsections below.

3.5.1 Front-end Integration

The recent system of iMotion used to send the sketch in frames as bitmap images. Yet, our project uses a different representation that is described in 3.4.2.6.2. This structure is implemented in JavaScript, and included in the

UI.

This structure was supposed to be created as soon as a user starts sketching. Therefore, the canvas functions in the original file, named **sketchObjectCanvas.js,** have been modified. Now a new **Sketch** is instantiated as soon as a new **Canvas** is instantiated or the canvas is cleared. Then every time the user clicks on the canvas (which indicates start of a draw) a new **Stroke** is created and the points where the stylus passed are sampled and included in this newly created **Stroke**. Releasing the stylus (or going out of Canvas) indicates the end of the current **Stroke** and all of the opened canvases starts waiting for another new **Stroke** to be drawn. All those **Point**s and **Stroke**s are already recorded on the canvas by **`stroke()`** method of HTML 5 **Canvas** structure.

The current system sends an auto-completion request after a stroke is addded. If not, the system waits for 5 seconds to send the request. The auto completion request is an **HttpPostRequest** using *Oboe.js* [17] API. The request contains a stringified version of a JSON which has three elements: the request ID (Date object), array of image bitmaps (which is used in original iMotion) and the JSON representation of the sketch the user has drawn.

The server sends a response containing six elements: request ID, top instances computed by the classifier of University of Basel, probabilities of those instances, classes of top instances computed by our classifier (standard pipeline), probabilities of those instances and best full sketches computed by k-nearest neighbor algorithm.

When the response has been successfully received from the server, the client takes the top instances and show them in two **`<div>`** block elements with a green background, one for results acquired by the implementation of University of Basel and one for the results acquired by our standard implementation in a pop-up fashion. The probabilities are only shown in console. CSS code of the respective elements has been modified to show the pop-up in a fixed position which is a bit higher than the original code. The suggestions are removed and the pop-up is disappeared after 5 seconds.

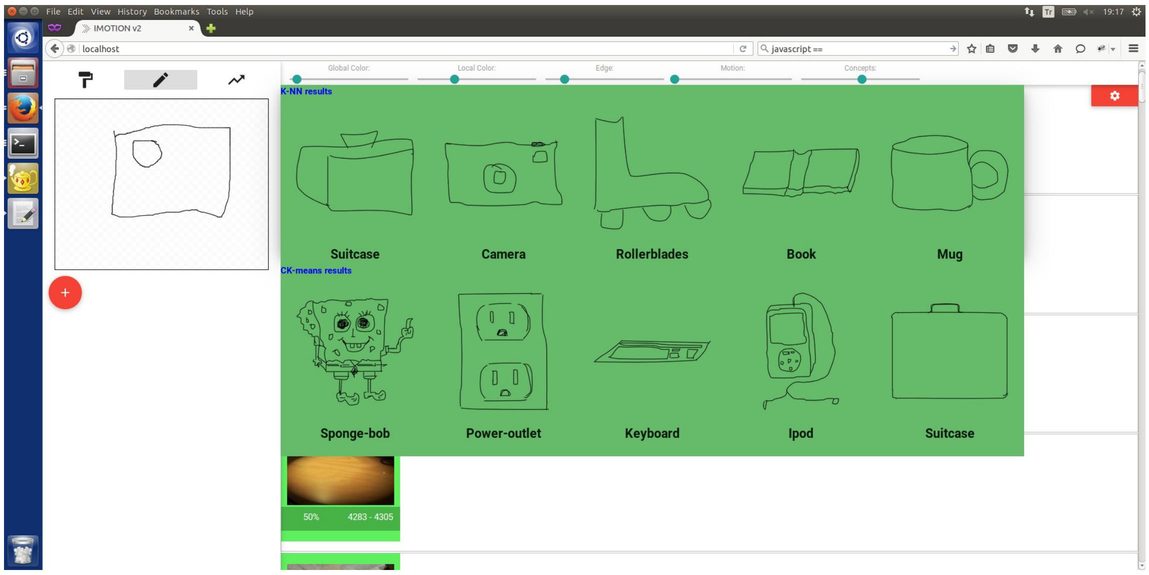


Figure 49: An example screen-shot of giving suggestions

3.5.2 Back-end Integration

Back-end integration has three main steps: providing an interface to the server, communicating with the client, and training for the k-nearest neighbor of all sketches stored in *.csv* files.

To provide an interface to the sketch server, we imported the entire prediction pipeline of our standard implementation. The paths of those classes are appended to the system path respective to their relative path to the working directory. Modules such as **extractor.py** had a name clash with the original iMotion code. Those classes are merged for simplicity. Then a main module is created and a method named **`runPrediction()`** has been written in order for the server to run directly. This method specifies the path to load and put already trained data a new **`Predictor`** object and the predictor gives desired number of predictions (in our case, 5) for the given JSON string. Having received the predictions, the server also finds the best full sketch for every class included in the predictions to be displayed in the website by using K-nearest neighbor algorithm.

The server has been built on the top of **`BaseHTTPServer`**, that handles the requests using **`do_POST()`** function. It reads the content of the request as raw data. In our case, the raw data is a JSON string. **`json.loads()`**

function converts this raw data into a dictionary. Through this dictionary, the server extracts the bitmap representation and the representation we have implemented in the client side and provides them as input to the respective functions (in our project it is **`runPrediction()`**). Having received the class names as output, it finds the best full sketch match to be displayed by putting the bitmap and sketch representations to the k-nearest neighbor functions. Finally, the server formats the response and sends it back to client.

To find the best full sketch match for auto-completion, the k-nearest neighbor function of **scikit-learn** [20] was used. For every sketch in Eitz dataset, we firstly extracted the features of all sketch instances in the respective *.csv* files, then put them in an array and trained this in a new **`NearestNeighbors`** object. To avoid doing this process every time we started the sketch server, **`NearestNeighbors`** objects of the sketch classes have been saved as files using Pickle serialization of Python.

One issue with iMotion is that the server sends the file names and probabilities instead of the sketch directly. This means only the server can act as a client to itself. We did not do anything about it since iMotion was created for testing purposes and this addition would make the system slower. In case iMotion is opened the public, the developers can send sketches encoded in **base64** through **Oboe** requests which will display the sketches one by one.

4 Future Work & Conclusion

To conclude, during the summer research period, we ported an existing work regarding sketched symbol auto-completion to Python in order to raise the runtime performance. Moreover, to see it in action, our students developed an Android user interface together with a server back-end. In order to make this framework usable in iMotion system, we also used the server implementation and modified the UI in a way that it shows the suggestions of our system together with the suggestions coming from the auto-completion pipeline written by University of Basel. At the beginning of the period, our expectation was only to rewrite the original pipeline without any amelioration. Thanks to their great effort, students implemented and tested the pipeline earlier than we expected, then we were able to finish all of the things we mentioned here.

Yet, we believe in that there are still some missing parts in the puzzle. As the time wasn't sufficient, we couldn't focus on different techniques of grouping sketches in the alternative pipeline. The way sketch classes are grouped is the key part of the alternative implementation and we believe in that there can be some other ways which are waiting for being discovered and which are useful to beat the standard pipeline. Moreover, accuracy performance of the pipeline of University of Basel hasn't been compared to our standard and alternative implementations yet. We don't still have any analyzable results to make a comparison among each other. In the near future, our plans are to conduct a deeper research about sketch grouping and to complete the analysis of the auto-completion pipeline of University of Basel and deliver the integrated system to all the partners in the project.

5 References

1. Schoeffmann, Klaus, et al. "The video browser showdown: a live evaluation of interactive video search tools." *International Journal of Multimedia Information Retrieval* 3.2 (2014): 113-127.
2. Rossetto, Luca, et al. "IMOTION–Searching for Video Sequences Using Multi-Shot Sketch Queries." *International Conference on Multimedia Modeling.* Springer International Publishing, 2016.
3. Tirkaz, Caglar, Berrin Yanikoglu, and T. Metin Sezgin. "Sketched symbol recognition with auto-completion." *Pattern Recognition* 45.11 (2012): 3926-3937.
4. MATLAB, http://www.mathworks.com/products/matlab/
5. Wagstaff, Kiri, et al. "Constrained k-means clustering with background knowledge." *ICML.* Vol. 1. 2001.
6. Eitz, Mathias, James Hays, and Marc Alexa. "How do humans sketch objects?." *ACM Trans. Graph.* 31.4 (2012): 44-1.
7. Niels, Ralph, Don Willems, and Louis Vuurpijl. "The nicicon database of handwritten icons for crisis management." *Nijmegen Institute for Cognition and Information Radboud University Nijmegen, Nijmegen, The Netherlands* 2 (2008).
8. Ouyang, Tom Y., and Randall Davis. "A visual approach to sketched symbol recognition." (2009).
9. https://github.com/ozymaxx/sketchfe
10. NVIDIA, Parallel Programming and Computing Platform | CUDA, http://www.nvidia.com/object/cuda_home_new.html
11. Hsu, Chih-Wei, Chih-Chung Chang, and Chih-Jen Lin. "A practical guide to support vector classification." (2003): 1-16.
12. pandas – Python data analysis library, http://pandas.pydata.org/
13. pickle – Python object serialization, https://docs.python.org/2/library/pickle.html
14. O. C. Altıok, K. T. Yesilbek, T. M. Sezgin, "What Auto Completion Tells Us About Sketch Recognition." In Proceedings of Expressive 2016, Posters, Artworks, and Bridging Papers, the Eurographics Association, Lisbon, Portugal, May 7-9, (2016).
15. HTTP – Hypertext Transfer Protocol Overview, https://www.w3.org/Protocols/
16. Flask (A Python Microframework), http://flask.pocoo.org/
17. Streaming JSON loading for Node and browsers: Oboe.js, http://oboejs.com/