# Multimedia Retrieval Project Report

MARCELLO EIERMANN, Utrecht University, Student ID: 4940539
JESSE VROOMAN, Utrecht University, Student ID: 1112988
BAS DE BOER, Utrecht University, Student ID: 6461263

In this paper, the process of creating a content-based 3D shape retrieval system is outlined. The system will, given a 3D shape, find and display to the user the most similar shapes in a given 3D shape database. The process will be described from the beginning (preprocessing and cleaning) till the end (matching and evaluation) of the pipeline. 21 different features will be described and tested with 7 different matching algorithms, including a customized distance function (precision 38.5%), two neural network approaches and 4 dimensionality reduction strategies. For which the best algorithm (ANN) has a precision of 38.7%.

**ACM Reference Format:**
Marcello Eiermann, Jesse Vrooman, and Bas de Boer. 2023. Multimedia Retrieval Project Report. 1, 1 (November 2023), 49 pages.

## INTRODUCTION

The aim of this project is to build a retrieval system on 3-dimensional shapes. The system will query a shape database and returns objects that are similar to the object given as input. This report will describe the steps for creating and evaluating this retrieval system, the source code can be found on Github [14]. Firstly, the data from the database needs to be read and displayed. Followed by a number of preprocessing and cleaning steps in order to make the data ready for feature extraction. Next, the computation of the features will be described as well as the process for selecting the most meaningful features. These features of the 3d shapes will be used to realize an end-to-end Content-Based Shape Retrieval (CBSR) system. The CBSR system will be optimized using K-nearest neighbors (KNN) and dimensionality reduction (DR) in order let the system perform faster on larger databases. Finally, a number of quality metrics will be used to evaluate the CBSR system. The results will be displayed and discussed as well as the strong points, and possible limitations of the system.

## 1 STEP 1: READ AND VIEW THE DATA

The shape database [6] used for this project contains 2.483 shape files in .obj format of 69 different classes. Python (version 3.8) is utilized together with the Open3D library [2] to view the shapes. Figure 1 shows a visualized object of the class *Car* as shaded model, without (left) and with (right) the cell edges drawn.

## 2 STEP 2: PREPROCESSING AND CLEANING

### 2.1 Statistics over the whole database

There is a large discrepancy in the number of vertices and triangles in the objects of the dataset. The object with the lowest number of vertices and triangles (16 each) belongs to the class *Door*. There are a total of 94 outliers, which have below 100 Vertices or triangles. On the upper end there are also outliers with a large number of vertices. The most vertices are inside an object of the class *Skyscraper* counting 73440 vertices. The object with the most triangles is a *Biplane* with 129881 triangles. The histograms in Figure 2 show a clear distribution where most object have below 2000 triangles or vertices. Figure 3 is a zoomed in view, showing only the objects up to 2000 vertices and triangles.

Authors' addresses: Marcello Eiermann, r.m.eiermann@students.uu.nl, Utrecht University, Student ID: 4940539; Jesse Vrooman, j.f.vrooman@students.uu.nl, Utrecht University, Student ID: 1112988; Bas de Boer, b.k.deboer@students.uu.nl, Utrecht University, Student ID: 6461263.

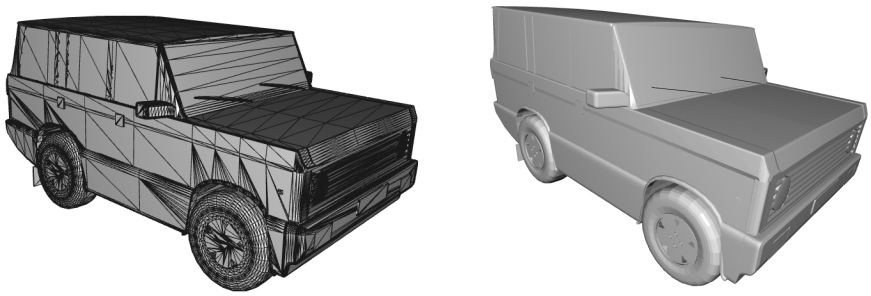


Fig. 1. An object of the class car

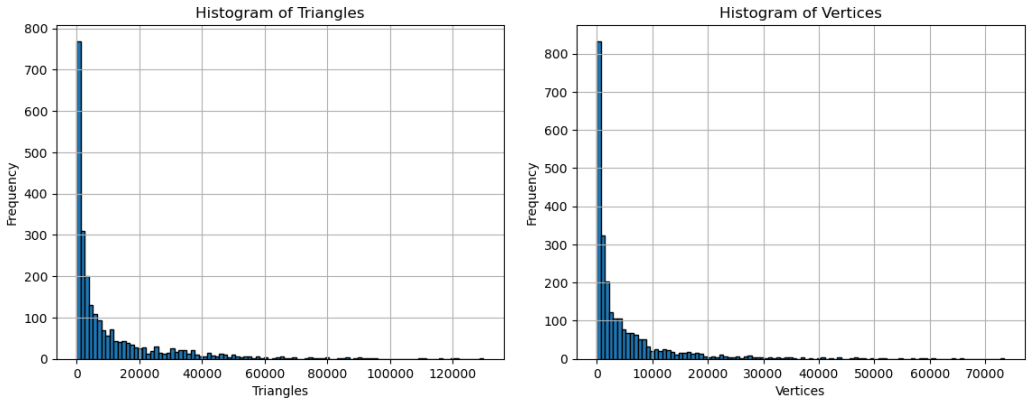



Fig. 2. Histogram of Vertices and Triangles

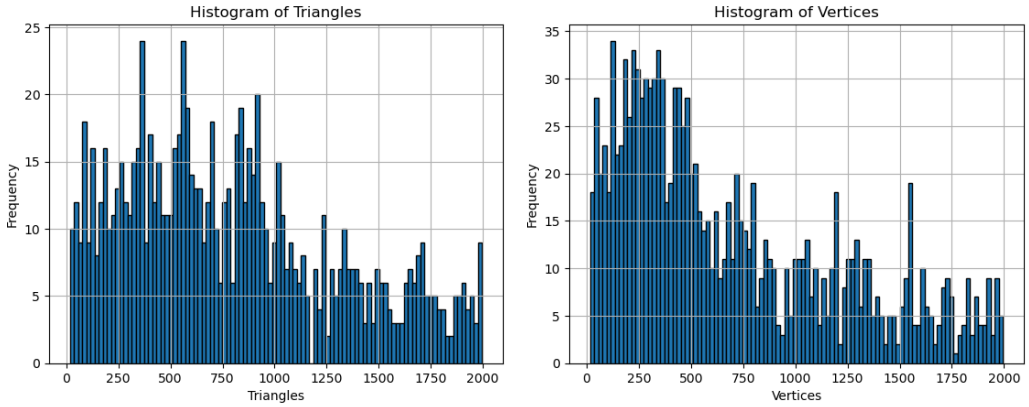



Fig. 3. Histogram of Vertices and Triangles up to 2000

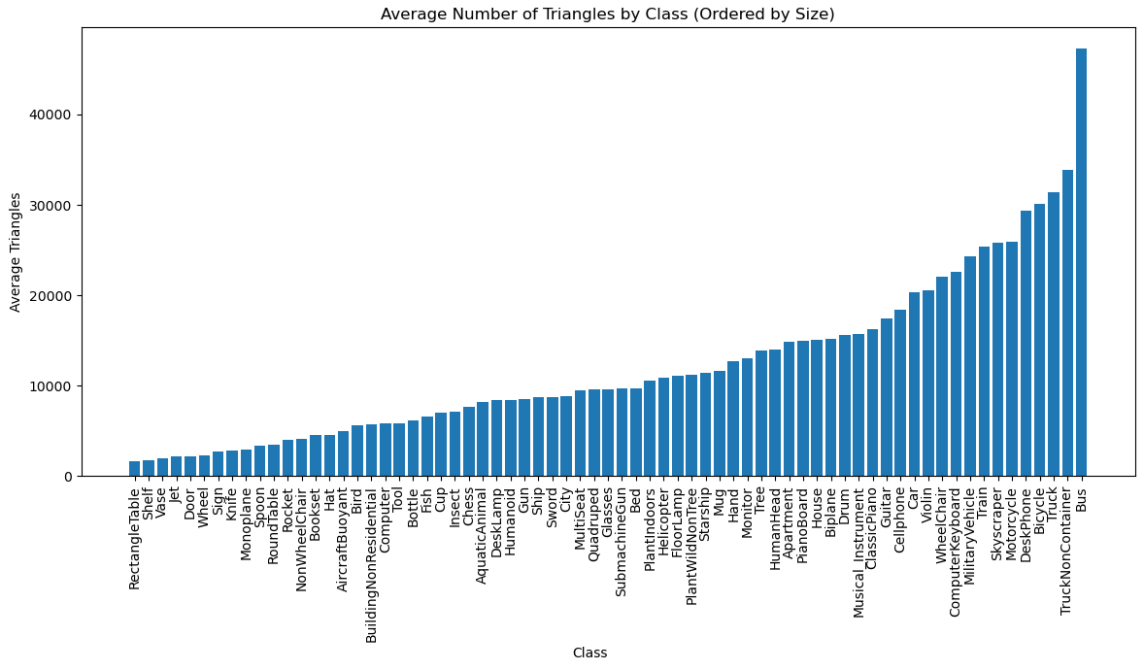



Fig. 4. Avg number of triangles per class

We further analyzed the average number of triangles per class and found substantial differences. Figure 4 shows classes with less than 2000 triangles on average while other classes have objects with over 30000 average triangles. Within each class there can be a great variation in object detail. In Figure 5 we can see two objects of the class *House*. The left object is quite detailed with 46252 vertices and 90161 triangles while the right object is made up of only 35 vertices and 48 triangles. Both are examples for outliers. The average number of vertices is 5025 and the average number of triangles is 10691. A shape that can be considered an "average shape" for this database having 5026 vertices and 9528 triangles, an example can be seen in Figure 6.

### 2.2 Resampling outliers

Multiple approaches for resampling outliers have been implemented. With our first approach we used PyMeshLabs *meshing_isotropic_explicit_remeshing* and *meshing_decimation_quadric_edge_collapse* functions and set the target vertices to 7000, while having a narrow window of between 6300 and 7700 vertices where the remeshing is accepted. The following remeshing steps were performed:

(a) Calling *meshing_isotropic_explicit_remeshing* function iteratively, checking after every step if it is close to the desired number of vertices

(b) Calling *meshing_decimation_quadric_edge_collapse* function with the parameter *targetfacenum*, which sets the desired number of triangles. In order to get an accurate number of vertices we use the ratio of triangles/vertices of the original object as factor for the parameter. Is is calculated as follows $targetfacenum = targetvertices * (triangles/vertices)$

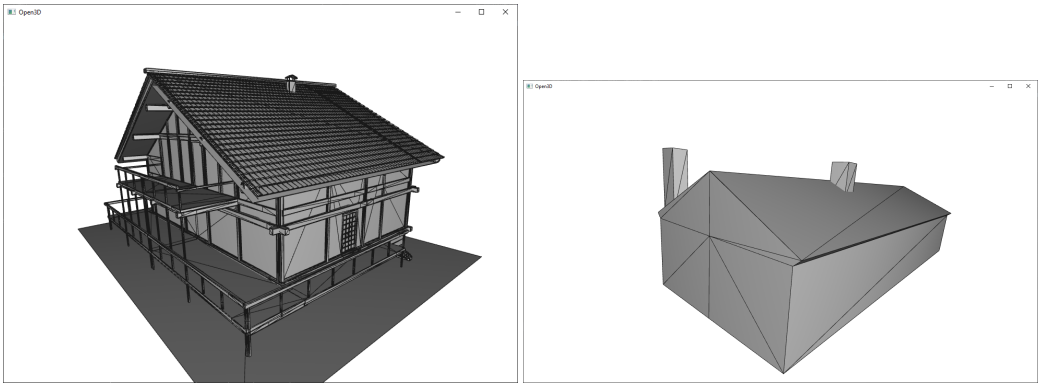


Fig. 5. Two objects of class house, left is a high outlier, right is a low outlier

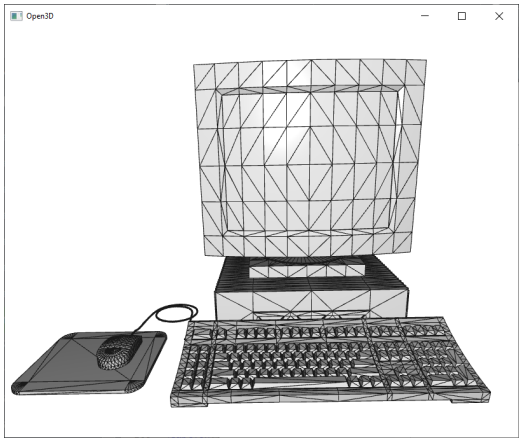


Fig. 6. Average shape of the database

After finishing all other steps of the database retrieval process and looking at the performance we came to the conclusion that the remeshing is one of the main issues. We checked several meshes by hand and while in most cases the remeshing went without problems, there was a noticeable amount of examples where the remeshing gave subpar results. Figure 7 shows such an example where the house has holes and an uneven amount of vertices after remeshing with PyMeshLab. We wanted to make the resampled meshes better, hence we choose an other approach for resampling the meshes.

Our next approach was using the Open3D library with the functions *Subdivide_loop* and *simplify_quadric_decimation*. With this method the meshes didn't have holes anymore but as seen this Figure 7 in some occasions the meshes would change their form to a round or oval shape. We decided to not leave out the problematic meshes but instead to try another remeshing approach. With the new approach we decided to use PyMeshLab again but without the *meshing_decimation_quadric_edge_collapse* function, as it seems to be responsible for some of the failed remeshes. Instead, we only remesh to above 7000 vertices and do not account for the outliers on the upper end. Figure 9b shows

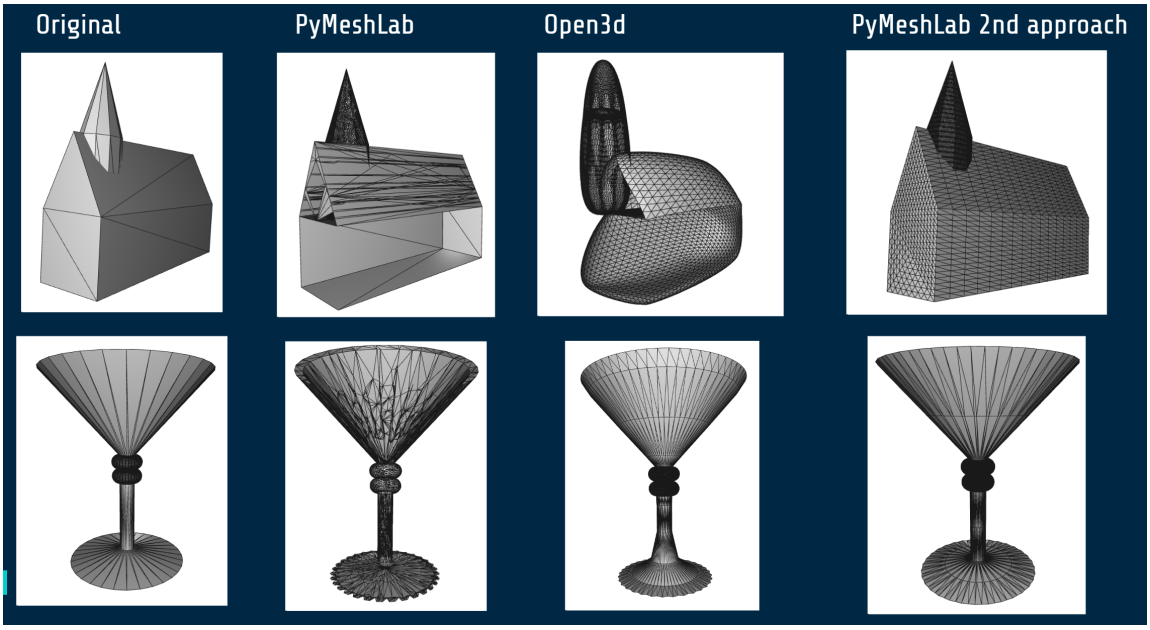



Fig. 7. Remeshing problems with Open3D and PyMeshLab

the new distribution of vertices. By having all meshes at or above 7000 vertices, we have enough sampling points to determine a meaningful distribution for the histograms. Of course the large meshes will be computed much slower, but the feature extraction still runs in under 30 seconds, which is why we deem this trade-off justified. However, the second example in Figure 7 shows that the new approach is not without its flaws. The original object of class *Cup* has many vertices in the middle part but only few vertices extending to the top and bottom part of the cup. When computing the histograms this object would be a clear outlier as the random sampling would choose vertices in the middle part much more often. The Open3D approach as well as our second PyMeshLab approach did not fix this issue, but instead intensified it by creating even more vertices in the middle part. The first PyMeshLab remeshing returned the most equally distributed object. For future work, we could make the random sampling weigh the vertices that are connected to large faces more in order to fight this imbalance.

### 2.3 Checking the resampling

In order to check if the resampling was completed successfully we first look at an example object. Figure 8 shows an object of class *Rocket* before (left) and after (right) resampling. As we can see, the large triangles have been broken into smaller and evenly sized triangles and the number of vertices is within the target range. The histogram in Figure 9a clearly shows that most objects are now at or close to 7000 vertices. The new average number of vertices is 7027, with the highest being 7694 and the lowest being 6010 (see fig. 9a. For our second approach we can see in figure 9b that all object have 7000 or more vertices.

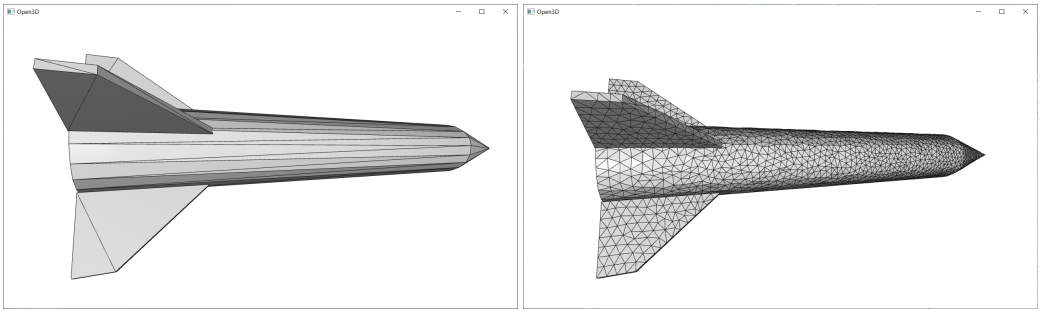
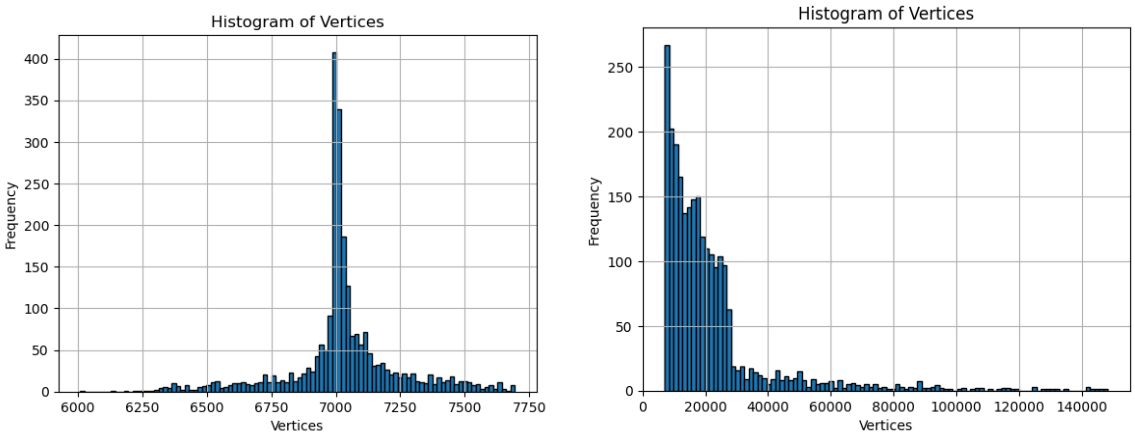


Fig. 8. Rocket before and after resampling




(a) First approach

(b) Second approach

Fig. 9. Histogram of vertices after resampling

### 2.4 Normalization

All shapes in the database will be normalized as a means to fairly compare the features that will be extracted. By normalizing the meshes there is no difference in maximum size, pose and location between the objects in the database. The main benefit is that it encloses all features in a common boundary without losing information. The following order is used:

(1) Centering to the barycenter of the coordinate frame
(2) Remeshing
(3) Rotation alignment
(4) Flipping
(5) Scaling

*2.4.1 Translation.* The first step in the normalization process is to translate the mesh so that its barycenter coincides with the coordinate-frame origin. The method get_center is used to determine the barycenter of the mesh. The barycenter is calculated by counting up all the vertices (p) and dividing these by the number of vertices, it returns the mean of the TriangleMesh vertices (3D vector).

$$bary\_center = \frac{\sum p_i}{N} \tag{1}$$

The mean is used to translate the mesh to the origin by translating all points/vertices ($p_i$) of the geometry by the negative of the barycenter vector (o), resulting in the translated mesh with vertices $pt_i$.

$$po_i = p_i - o \tag{2}$$

In order to check if this normalization step is executed correctly the distance from the center of every mesh to the center of the coordinate frame is computed and a histogram of the distribution of distances before and after normalization is plotted. The function used for computing the distance is shown here. Where $po_{mean}$ = [x1, y1, z1] is the mean of the TriangleMesh vertices and [x2, y2, z2] is the coordinate frame origin ([0,0,0]).

$$distance = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2} \tag{3}$$

The left bar chart in Figure 10 shows the distribution of distances from the center of every mesh to the coordinate frame center. Most distances are in the range 0-1, however, there are also a number of outliers. When the normalization is executed correctly, all distances from the center of every mesh to the coordinate frame origin need to be approximately 0. The right bar chart in Figure 10 shows that this is the case, with all the distances to the center in one single box (0). This proves that the centering step in the normalization process is executed correctly

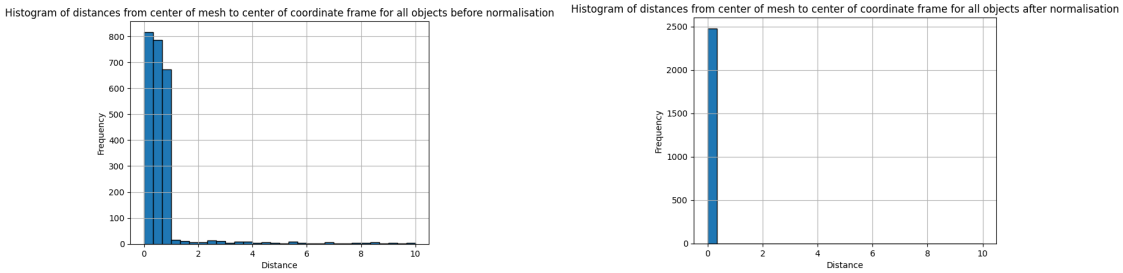



Fig. 10. Histogram of distances from the center of the mesh to the center of the coordinate frame, before and after normalization

*2.4.2 Alignment.* The eigenvectors of every mesh will be used in order to rotate all the meshes in the database to have the same orientation. These eigenvectors are computed with the principal component analysis (PCA). In order to compute the PCA, the barycenter of the mesh needs to be centered at the origin of the coordinate frame. This step is already described in section 2.4.1. After this, a 3x3 covariance matrix on the matrix of vertex positions is computed. Covariance is a generalized and unnormalized version of correlation across multiple columns. A covariance matrix is a calculation of covariance of a given matrix with covariance scores for every column with every other column, including itself. The covariance matrix is computed with the numpy.cov() function. The next step is to compute the eigenvectors and eigenvalues. This is done by the numpy.linalg.eig(cov) function. Next the eigenvectors will be ordered on their magnitude, the corresponding eigenvalues are used for this. Resulting in three eigenvectors for every mesh ordered from big to small.

The ordered eigenvectors are used as a rotation matrix for the mesh. With the function mesh.rotate(eigenvectors, center=(0, 0, 0)) the mesh is rotated so that the eigenvectors are aligned with the corresponding axis of the coordinate frame. In order to prove if the rotation is executed correctly the angles between the eigenvectors and the corresponding x,y and z axis of the coordinate will be computed, the Cosine similarity function is used for this. It is shown in Equation 4, where A is the eigenvector and B is the corresponding x,y, or z axis:

$$\cos(\theta) = \frac{A \cdot B}{\|A\| \, \|B\|} \tag{4}$$

The Cosine similarity is computed for all eigenvectors of all meshes before and after normalisation. A histogram is plotted for both in order to see if the alignment is executed successfully. In the left histogram in Figure 11 is clearly visible that there are many different Cosine distances between the eigenvectors and the corresponding axis of the coordinate frame. The right histogram in Figure 11 shows that after normalisation all these values are stacked at 1 and -1 meaning that there is no distance between the axis of the coordinate frame and the corresponding eigenvectors. So, this proves that this normalisation step is executed correctly.

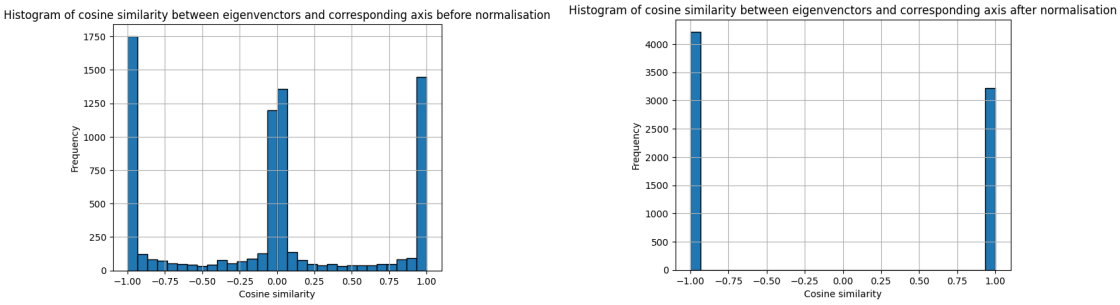



Fig. 11. Histogram of Cosine similarity between the eigenvectors of all meshes and the corresponding axis of the coordinate frame x,y and z, before and after normalisation

### 2.5 Flipping

Since eigenvectors are unoriented lines, PCA alignment cannot distinguish between flipped (mirrored) shapes. To ensure that that we consistently place the largest mass in a uniform manner across various shapes within the database, a flipping test is needed. The flipping test will ensure that most of the mass will be on the left of the xy, yz and xz planes. This is realized by creating a "flip matrix" which is a identity matrix where a 1 is changed into a -1 when a flip is needed around a specific plane. In order to get the second-order moment of the area with respect to an axis i the following formula is used:

$$f_i = \sum_t sign(C_{t,i})(C_{t,i})^2 \tag{5}$$

resulting in the following transformation matrix F:

$$\begin{bmatrix} sign(f_x) & 0 & 0 \\ 0 & sign(f_y) & 0 \\ 0 & 0 & sign(f_z) \end{bmatrix}$$

All vertex coordinates are then flipped by the corresponding transformation matrix for every mesh in the database. This should result in all objects in the database having the most mass on the left side of the planes. A histogram is plotted to prove that after the normalisation all masses are at the correct side of the plane. The right bar chart in Figure 12 shows that all centers of mass are located on the negative side of the their corresponding plane (xy, yz and xz). Before the normalisation, there was a wide distribution over both the positive and the negative values (Figure 12 (left)).

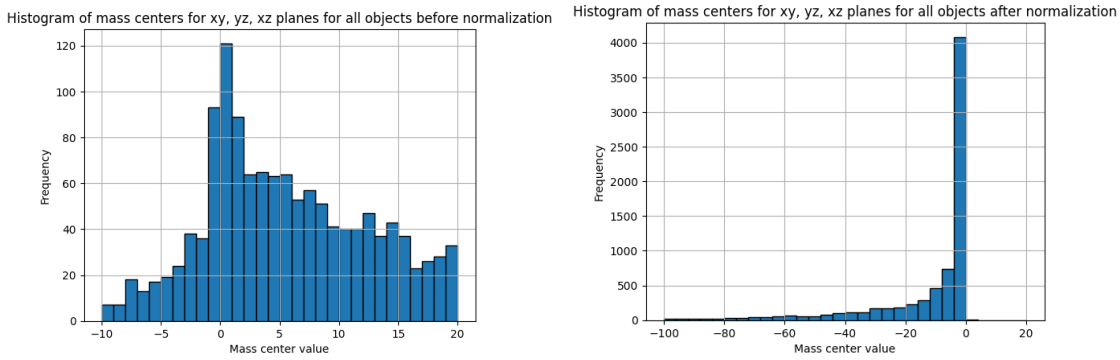



Fig. 12. Histogram of mass centers around xy, yz and xz planes for all meshes in the database, before and after normalisation

### 2.6 Scaling

Next the mesh is scaled uniformly so that it fits in a unit-sized cube. The scaling is done by using mesh.scale(factor) where

$$factor = 1/max(x_{max} - x_{min}, y_{max} - y_{min}, z_{max} - z_{min}) \tag{6}$$

In this equation mesh.get_max_bound() returns the maximum bounds for geometry coordinates and mesh.get_min_bound() returns the minimum bounds for geometry coordinates. The scaling factor is based on the greatest distance out of the x, y and z axis because if a mesh would be scaled based on every dimension shapes would be distorted due to the fact that an object than would be scaled to 1 in every direction. After normalization, the center of mass of each object coincides with the origin and is scaled uniformly so that it tightly fits in a unit-sized cube. A histogram of the maximum length for every object in the database is used in order to prove that after all normalisation steps the scaling is executed correctly (Figure 13).

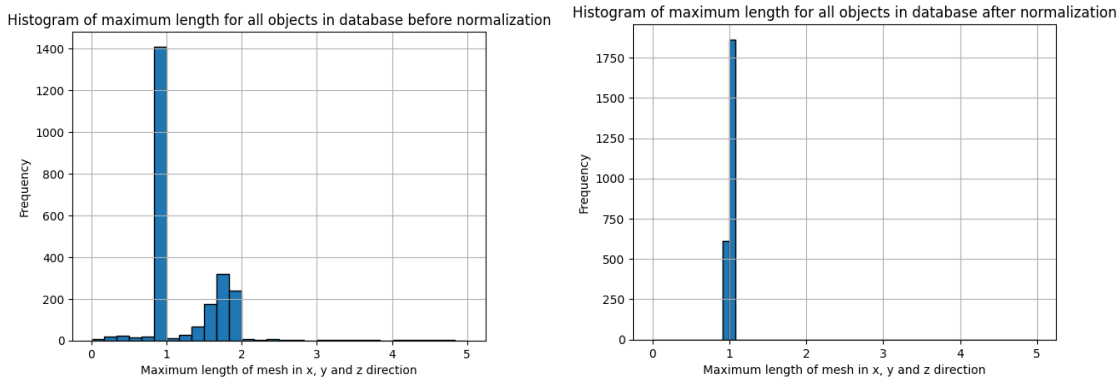



Fig. 13. Histogram of the maximum distance over the x, y and z dimension for all meshes in the database, before and after normalisation

## 3 STEP 3: FEATURE EXTRACTION

To be able to match and retrieve similar objects from the 3D shape database, it is necessary to extract features from the 3D models that can characterise them into a single vector. This section will introduce multiple features that will be extracted, which can be separated between, features that result in single scalar values, histograms, and silhouettes. The extracted values will be plotted for ten classes in order to compare within the same class to show the similarities, as well as the dissimilarities between the different classes.

### 3.1 Elementary Features

To show how the elementary features behave, the features will be calculated for a range of different basic 3D shapes. With these values it can be determined if an elementary feature is behaving as intended. The shapes that were used can be seen in Figure 14.

(a) Cube (b) Rectangle (c) Rectangle2 (d) Cone (e) Pyramid (f) Cylinder (g) Sphere

Fig. 14. Basic shapes

*3.1.1 Surface Area.* A simple formula can be used to calculate the surface area of a 3D object. This is done by summing all the surface areas of the triangles in the mesh. The formula for the surface area $S$ can be found in Equation 7, where $t_i$ is a triangle with vertices $(p1, p2, p3)$, $v1 = p1 - p2$, and $v2 = p1 - p3$.

$$S = \sum_{t_i} \frac{1}{2} \left| v1 \times v2 \right| \tag{7}$$

| cube | rectangle | rectangle2 | cone | pyramid | cylinder | sphere |
|---|---|---|---|---|---|---|
| 3.0682 | 1.8311 | 1.1408 | 2.0403 | 3.2361 | 3.7275 | 3.1211 |

Table 1. Surface area for basic 3D shapes

Table 1 shows the surface area for 7 basic shapes. It can be seen that the surface area of the sphere is 3.1211, this makes sense since the objects are scaled to fit in a $1x1x1$ box, making the sphere's radius $r \approx 0.5$. The area of a sphere is $2\pi r$, making $S \approx 1\pi = 3.14$. The rectangles have smaller surface areas then the cube, this also makes sense since they occupy less of the $1x1x1$ box then the cube can. The expectation was that a cube would have the largest surface area, since it would be able to fit the box the best. After analysis was found that, the eigenvalues of a cube make it tilted and therefor not scaled perfectly to the $1x1x1$ box.

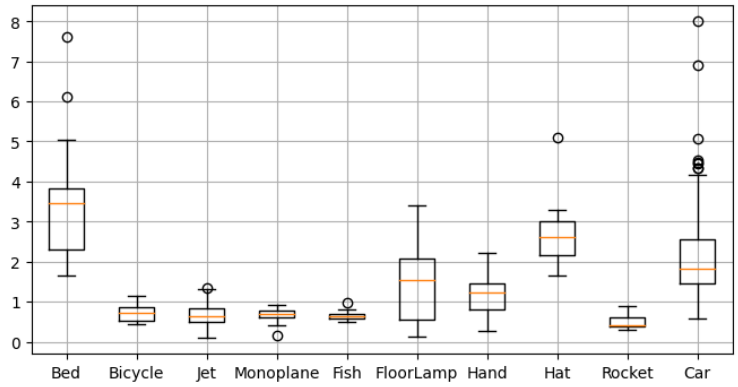



Fig. 15. Box-plot showing the distribution of surface areas per class, including outliers.

In Figure 15 the box plot show the distribution of the surface areas for 10 classes. There is decent overlap between a lot of the shapes in classes, a good example are the *Bicycle*, *Jet* and *Monoplane* classes. Since the spread over those surface areas are close they are likely to be classified the same when just using surface areas. When taking *Rocket*, *Hand* and *Hat*, the chance for classifying correctly based on surface area is more likely. There are multiple cases of this between all classes, and that shows that surface area has potential as a feature.

*3.1.2 Convex Hull Volume.* Using the Open3d library the convex hull can be created of a mesh. Then using the volume function in Equation 10 on the convex hull, gives the convex hull volume, $V_{ch}$.

| cube | rectangle | rectangle2 | cone | pyramid | cylinder | sphere |
|---|---|---|---|---|---|---|
| 0.3657 | 0.1419 | 0.0640 | 0.1886 | 0.3333 | 0.5506 | 0.5169 |

Table 2. Convex hull volume for basic 3D shapes

In Table 2 it can be seen that the convex hull volume for the cube is larger than rectangle, and rectangle2. Just like with surface area this is due to the rectangles taking up less space in the $1x1x1$ box used for scaling. Also can be seen that the convex hull volume of the pyramid is 0.3333, this makes sense since the convex hull of the pyramid, is the pyramid itself. For this pyramid with values $height = h = 1$ and $sides = a = 1$, the volume is calculated as $V = \frac{1}{3}a^2h = 0.3333$‘. This proves that the convex hull feature works correctly.

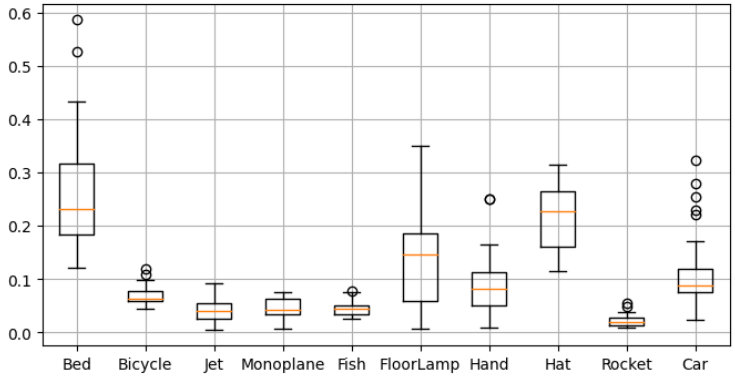



Fig. 16. Box-plot showing the distribution of the convex hull volumes per class, including outliers. For 10 classes from the database.

When taking a look at the box plots in Figure 16, the convex hull volumes actually seem distributed similarly as the surface area, so there is a correlation between the surface area and the convex hull volume. Furthermore, the box plot shows that for a number of classes there is a high in-class similarity (*Jet*, *Rocket*, *Fish* and *Bicycle*), while for other classes the convex hull volumes are more spread out. There is also a high out-class dissimilarity for this feature, except for classes containing similarly shaped objects (*Jet*, *Monoplane* and *Rocket*). This is a property we want to see in features.

*3.1.3 OBB Volume.* Using the Open3d library the orientated bounding box (OBB) can be created of a mesh. Then using the volume function provided by open3d, the volume of this bounding box can be calculated, $V_{obb}$.

| cube | rectangle | rectangle2 | cone | pyramid | cylinder | sphere |
|---|---|---|---|---|---|---|
| 0.7151 | 0.1952 | 0.0641 | 0.7586 | 1.0000 | 0.8524 | 0.9966 |

Table 3. OBB volume for basic 3D shapes

In Table 3 it can be seen that rectangle2 has the same OBB volume as convex hull volume, this is due to the eigenvectors of the object being perfectly aligned with its faces. Since the eigenvectors aren't aligned for a cube we

see a difference in those volumes. Pyramid shows that the bounding box perfectly fits inside the 1x1x1 box, since the height and sides of the pyramid are both 1. Sphere also gets close to a volume of 1, and that is the expected behaviour.

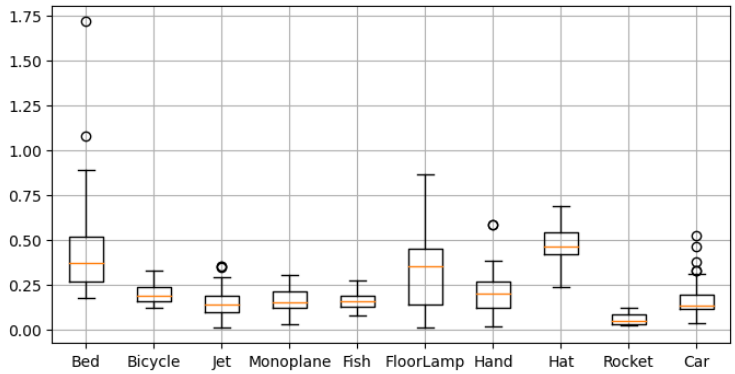



Fig. 17. Box-plot showing the distribution of the orientated bounding box volumes per class, including outliers. For 10 classes from the database.

Figure 17 shows that the OBB volume has a lower out-class dissimilarity that the previous elementary features. The reason for this is that all objects are scaled to fit the unit cube. On the other hand, there is a higher in-class similarity, which is good for matching the input object to the objects of the same class.

*3.1.4 Diameter.* The diameter indicates the longest distance between any two vertices in the mesh, and can be calculated with the formula in Equation 8, where for vertex $p_i = (x_i, y_i, z_i)$. When looking at Figure 18, it can be observed that the out-class dissimilarity is low for the diameter feature. Most classes have a value around 1, a result of the scaling step in the normalization process of the meshes. However, the is still a high in-class similarity, and also a noticeable out-class dissimilarity. For these reasons the diameter feature can be used for matching.

$$\begin{aligned} \text{euc\_d}(p_1, p_2) &= \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2} \\ \text{D} &= \max\left\{\text{euc\_d}(p_1, p_2) \,\middle|\, \forall p_1, p_2 \in \text{Vertices}\right\} \end{aligned} \tag{8}$$

| cube | rectangle | rectangle2 | cone | pyramid | cylinder | sphere |
|---|---|---|---|---|---|---|
| 1.238589 | 1.136389 | 1.062160 | 1.003119 | 1.414214 | 1.263462 | 1.000000 |

Table 4. Diameter for basic 3D shapes

In Table 4 it can be seen that for all diameter $D >= 1$. This is expected since always some vertices have a diameter of one when the object is scaled to the 1x1x1 box. Sphere also has a diameter of exactly one, what means that diameter is calculated correctly.

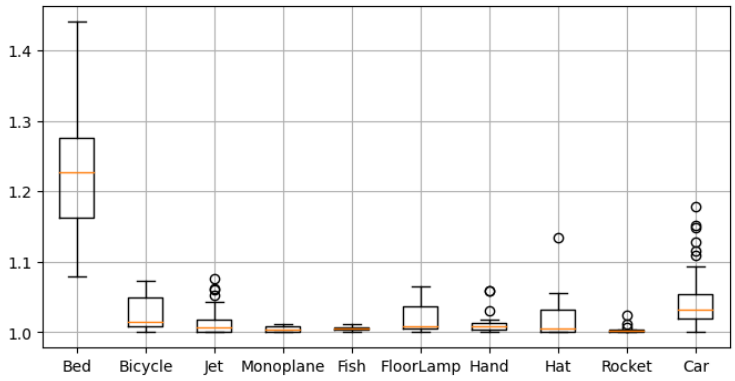



Fig. 18. Box-plot showing the distribution of the Diameters per class, including outliers. For 10 classes from the database.

*3.1.5 Eccentricity.* The Eccentricity gives the ratio between the largest and smallest eigenvalues. The formula to calculate this can be found in Equation 9, where $E$ indicates the eccentricity, and the eigenvalues = $[e_1, e_2, e_3]$.

$$E = \frac{max\{[e_1, e_2, e_3]\}}{min\{[e_1, e_2, e_3]\}} \tag{9}$$

| cube | rectangle | rectangle2 | cone | pyramid | cylinder | sphere |
|---|---|---|---|---|---|---|
| 0.818572 | 0.078629 | 0.064092 | 0.694326 | 0.740981 | 0.591674 | 0.560224 |

Table 5. Eccentricity for basic 3D shapes

In Table 5 it can be seen that rectangle and rectangle2 are the clear outliers from the basic shapes, when it comes to eccentricity. This is due to the objects being thinner then the rest, the thinner the object is the lower the value should become. Since the values calculated do represent this, it can be said that the eccentricity is behaving like expected. Figure 19 shows the distribution of eccentricity for 10 example classes. You can see that for many classes the in-class similarity is high (*Monoplane*, *Rocket* and *Jet*), while the out-class dissimilarity for differently shaped objects is also high (*hat*, *car* and *Hand*). This is a good indicator that this feature can be used well to match shapes to their similarly shaped objects in the database.

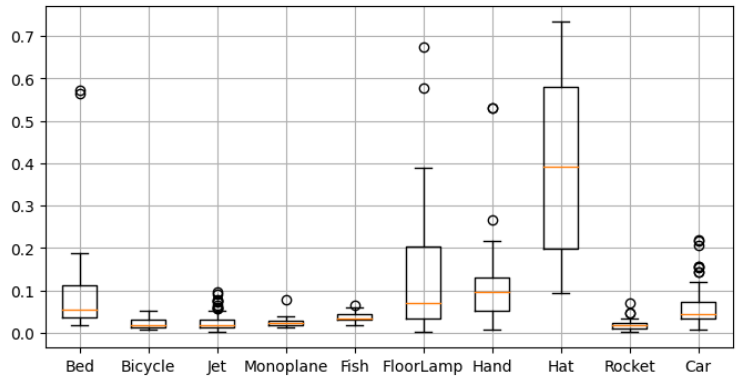



Fig. 19. Box-plot showing the distribution of the eccentricity per class, including outliers. For 10 classes from the database.

### 3.2 Discarded Elementary features

Calculating volume originally seemed solid. However, when observing the volumes more closely, the volume of many objects reached close to zero and many outliers were found. The reasoning for this is hard to discover, since it could go wrong in multiple steps of the MR system.

- Remeshing
- Inconsistent normals
- Hole stitching
- Volume calculation

The likely culprit is the inconsistent normals after the remeshing process. Many hours were invested to try and solve this issue, without good results. In the end we opted to continue without the volume based features. Due to this, convexity, compactness, rectangularity and volume can't be present in the final feature vector. When improving this system in the future, the following descriptors would be a priority.

*3.2.1 Volume.* The volume of any 3D shape (including convex shapes), can be calculated with a simple formula. When $t_i$ is a triangle of a mesh with vertices $(p1, p2, p3)$, and barycenter $o$. The volume formula can be found in Equation 10, where $v1 = p1 - o$, $v2 = p2 - o$, and $v3 = p3 - o$.

$$V = \frac{1}{6}\left|\sum_{t_i}(v1 \times v2) \cdot v3\right| \tag{10}$$

*3.2.2 3D Rectangularity.* The 3D rectangularity indicates how close the object is to a rectangular shape. It can be calculated with the formula in Equation 11, where $V_{obb}$ indicates the volume of the Oriented Bounding Box

$$R = \frac{V}{V_{obb}} \tag{11}$$

*3.2.3 Convexity.* The Convexity shows how convex the object is. The formula in Equation 12 shows how to calculate the convexity of an object, where $C$ indicates the convexity, $V$ the volume, and $V_{ch}$ the volume of the convex hull.

$$C = \frac{V}{V_{ch}} \tag{12}$$

*3.2.4 Compactness.* The Compactness shows how compact the object is relative to a sphere, and can be calculated with the formula in Equation 13, where $c$ indicates the compactness, $S$ the surface area, and $V$ the volume.

$$c = \frac{S^3}{36 \cdot \pi \cdot V^2} \tag{13}$$

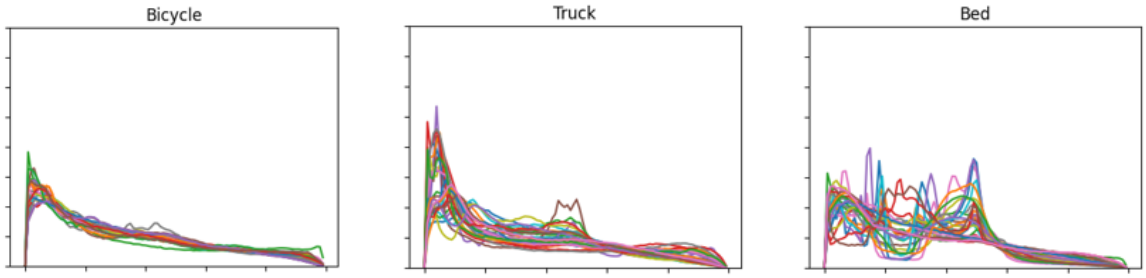
### 3.3 Histogram Features

The following histogram features are extracted by selecting random vertices in the mesh. Followed by a computation on the values of these vertices. These samples together form a histogram.

$$h_i = h_i / \sum_{i=0}^{n} h_i \tag{14}$$

The histograms are normalized to represent values between zero and one, as well as the total area of bins summing up to one. In Equation 14 the formula can be seen, that was used for normalizing the bins in the histogram, where $h_i \in H$ represents the bins for histogram $H$, with length of $n$. In the next subsections the different histogram features will be outlined, as well as examples for different classes.

*3.3.1 A3.* A3 takes three random vertices and calculates the angle between those vertices. The A3 histograms consist of $n = 1.000.000$ samples, and 100 bins. The sample value is picked based on a trade-off between time and the minimal number of vertices that need to be considered. The number of bins is set to 100 in order to represent the histograms well, while also having a good run-time performance. The number of bins is made consistent over all histogram features. Figure 20 shows the A3 histograms for three different classes. In sub-figure 20a you can see that the histograms for all objects in this class look very much alike. This is good, because we want the histograms to look very much the same for in-class objects and very different for out-class objects. However, there are some outliers. A good example of a class with bad performance on A3 is the Bed class (sub-figure 20c. These histograms show a lot of noise what can be due to that the beds are quite irregularly shaped within the same class. An overview of more classes can be found in appendix A.1. Here you can see that the shapes of the A3 histograms are differently shaped for other classes, which means that we can use this feature histogram to match our input shape to similar shapes in the database.




(a) (b) (c)

Fig. 20. Normalized A3 histograms for all objects in the Bicycle, Truck and Bed class. An overview of more classes can be found in A.1

*3.3.2 D1.* In order to compute the D1 histogram, one random vertex is selected and the distance between the barycenter and that vertex vertex is calculated. The D1 histograms consist of $n = \overline{v}$ samples where $\overline{v}$ is the amount of vertices, and 100 bins. This number of samples is chosen because it makes no sense to sample vertices double. In Figure 21 an example of a good (sub-figure 21a), an intermediate (sub-figure 21b) and a bad (sub-figure 21c) performing class is given. D1 clearly has a lot of noise within the histograms, but we can see that the amount of noise in the *Bicycle* D1 histogram shows quite some overlap between the different bicycle objects. However, for the *Wheel* class the histograms are very inconsistent. Overall, based on the graphs per class D1 does not look very promising for making a good distinction between classes. However, this will be tested in section 4.3.2 to be sure.

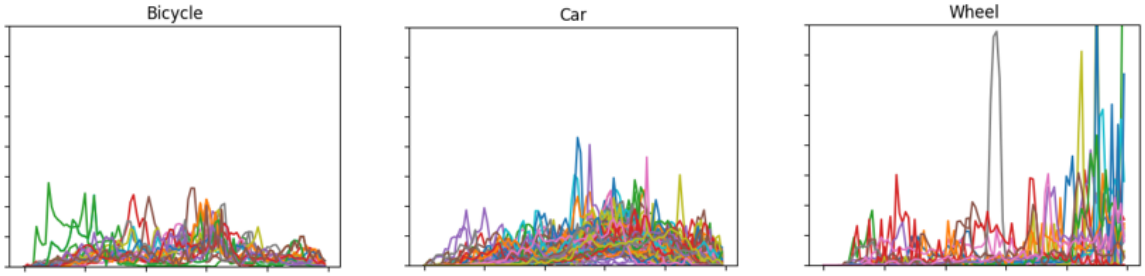



Fig. 21. Normalized D1 histograms for all objects in the Bicycle, Car and Wheel class. An overview of more classes can be found in A.2

*3.3.3 D2.* D2 is computed by taking two random vertices in the mesh and calculating the distance between those vertices. The D2 histograms consist of $n = 100.000$ samples, and 100 bins. Figure 22 shows a good (sub-figure 22a), an intermediate (sub-figure 22b) and a bad (sub-figure 22c) performing class. The D2 histograms already show a lot less noise compared to the previous D1 histograms. However, there are still a number of classes for which the D2 feature does not results in similar in-class histograms, for example the *Rocket* class. When looking at a bigger overview of D2 histograms per class (appendix A.3) it can be observed that the histograms between the different classes get closer together.

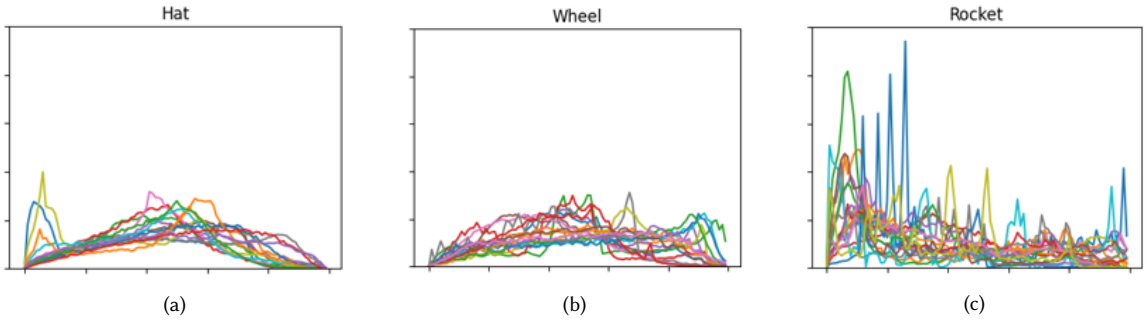
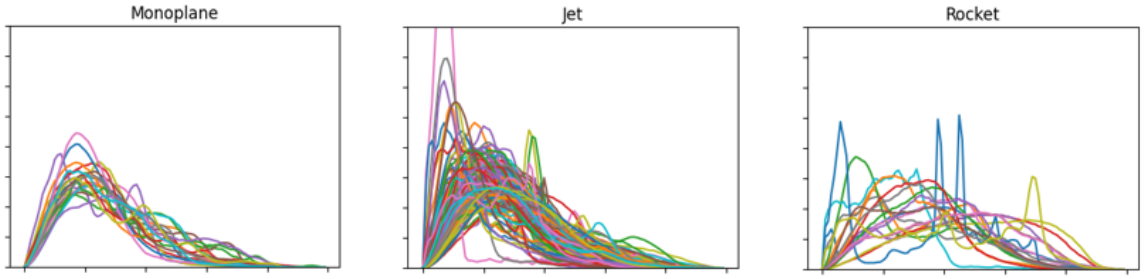



Fig. 22. Normalized D2 histograms for all objects in the Hat, Wheel and Rocket class. An overview of more classes can be found in A.3

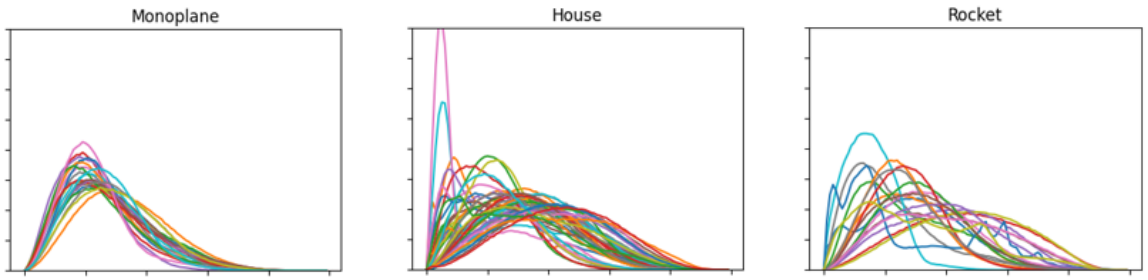
*3.3.4 D3.* D3 takes three random vertices and calculates the surface of the triangle between those vertices, of which the square root is taken. The D3 histograms consist of $n = 1.000.000$ samples, and 100 bins. In Figure 23 an overview of the D3 histograms for 3 different classes are shown. The D3 histograms show very little noise, much less compared to the D1 and D2 histograms. However, the histograms of this feature are still way more similar for some classes (sub-figure 23a than for other classes (sub-figure 23c). There is a clear difference between some classes (appendix A.4.




(a) (b) (c)

Fig. 23. Normalized D3 histograms for all objects in the Monoplane, Jet and Rocket class. An overview of more classes can be found in A.4

*3.3.5 D4.* Finally, the last feature histogram D4 is added to the feature vector. D4 takes four random vertices and calculates the volume of the tetrahedron between those vertices, and cube rooting the values. The D4 histograms consist of $n = 1.000.000$ samples, and 100 bins. In Figure 24 it can be seen that the D4 histograms of 3 different classes look way smoother. The D4 histograms show very little noise, and certain classes show clear differences in the distributions compared to other classes. The D4 histograms of the *House* class (sub-figure 24b still contains a few outlier, while there is a much higher in-class dissimilarity in the *Rocket* class (sub-figure 24c). On the other hand, for the *Monoplane* class (sub-figure 24a) the D4 histograms look very good. There is a high in-class similarity, however, when looking at a bigger overview of classes (appendix A.5) it shows that the D4 histograms look quite similar between classes. This results in a lower out-class dissimilarity, which is not good for finding the most similar object for the input query.




(a) (b) (c)

Fig. 24. Normalized D4 histograms for all objects in the monoplane, house and rocket class. An overview of more classes can be found in A.5

From looking at the histogram plots for the 5 histogram features, it looks like D1 is the least useful feature due to the high noise in the histograms. While A3 looks to have a very high in-class similarity and a high out-class dissimilarity. However, looks do not tell everything, the different histogram features will be analysed on how well they perform in section 4.3. All the histogram features will also be weighted, this process will be described in section **??**.

### 3.4 Silhouettes

Since the feature vector would be shorter than preferred with the loss of the volume based descriptors, a choice was made to include other descriptors for a more reliable matching procedure. The descriptors chosen for this project were an implementation of silhouette extraction of the 3d shapes. For each object three different silhouettes are extracted, one for each eigenvector (XY, XZ, YZ).

*3.4.1 Pixelated Images.* The silhouettes are generated along the three different axis on a pixelated grid of 150 by 150 pixels. These pixels are stored in a two dimensional array, where the pixels are binary coded ($pixel = 1$ or $pixel = 0$) dependent if the shape covers that pixel. This choice is quite arbitrary since the higher pixel count would benefit the matching process, but also heavily increase computation time. The choice was made to balance the computational time to a point where the feature extraction would not be impacted heavily, while still being able to see details within the image. In Figure 25 and 26 the silhouettes can be seen for a Jet and a Bicycle respectively.

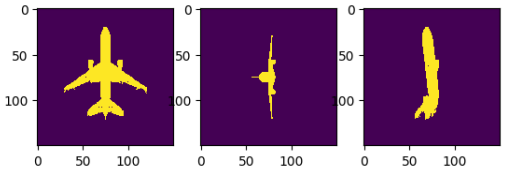



Fig. 25. Silhouettes for a Jet, where the silhouettes are: XY, XZ, YZ respectively

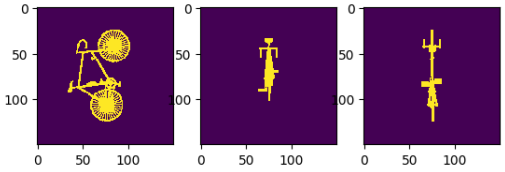



Fig. 26. Silhouettes for a Bicycle, where the silhouettes are: XY, XZ, YZ respectively

*3.4.2 Compression.* To keep the ability to match silhouettes directly together, the pixelated image would have to be added to the final feature vector. Since the silhouettes are a grid of 150 by 150 pixels, the flattened vector would have a length of 22500. This size of vector would heavily impact the matching process due to the many calculations that will have to be done. Also storing the vectors in our feature database could become to large to manage. Therefor was opted to compress these 22500 length vectors down to a length of 175. A simple compression algorithm is used and is looped seven times to get the preferred length of 175. The formula used to compress a single time is shown in Equation 15, where $\overline{v}$ is the length of the vector that will be compressed, and $c_i \in C$ where $C$ is the compressed vector of length $n$.

$$\begin{aligned} n &= \overline{v}/2 \\ for\ i &= 0..n \\ c_i &= (v_{i*2} + v_{i*2+1})/2 \end{aligned} \tag{15}$$

*3.4.3 Area.* To calculate the area of a silhouette a simple summation can be used of the matrix, since the pixels are represented by a value of zero or one. The formula is shown in Equation 16, where $I$ represents an pixel grid of size $nxm$, with pixels $p_{xy} \in I$.

$$Area(I) = \sum_{x=0}^{n} \sum_{y=0}^{m} p_{xy} \tag{16}$$

*3.4.4 Ratio over Area.* Ratios can be calculated for the differences of area between the different axis of the silhouettes. The two ratios calculated are the ratio between the largest and smallest area (formula in Equation 17), and the ratio between the largest and mean of the areas (formula in Equation 18).

$$R1 = \frac{min\{[Area(XY), Area(XZ), Area(YZ)]\}}{max\{[Area(XY), Area(XZ), Area(YZ)]\}} \tag{17}$$

$$R2 = \frac{mean\{[Area(XY), Area(XZ), Area(YZ)]\}}{max\{[Area(XY), Area(XZ), Area(YZ)]\}} \tag{18}$$

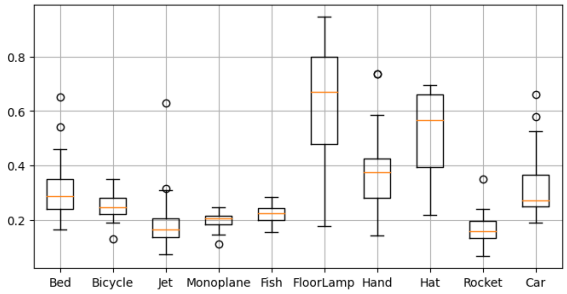



Fig. 27. Box-plot showing the distribution of ratios between the largest and smallest area of the silhouettes per class, including outliers. For 10 classes from the database.

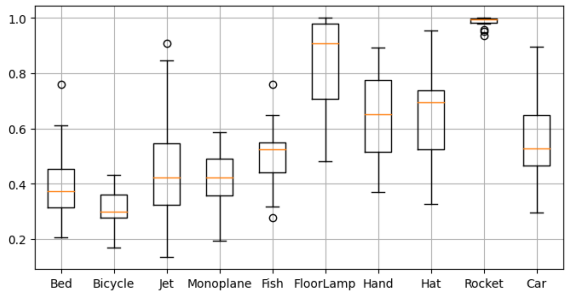



Fig. 28. Box-plot showing the distribution of ratios between the largest area and the mean of the areas of the silhouettes per class, including outliers. For 10 classes from the database.

*3.4.5 Silhouette Rectangularity.* Over each of the three different silhouettes a rectangularity can be calculated. This is done by taking the area of a silhouette and dividing it by the area of a bounding box of the silhouette. This represents how rectangular the different silhouettes are. The formula is shown in Equation 19, where $I$ represents an image, and $BB$ the bounding box of that image. The box plots showing the distribution of rectangularity over 10 example classes are shown in appendix A.6.

$$Rect(I) = \frac{Area(I)}{Area(BB)} \tag{19}$$

*3.4.6 Silhouette Circularity.* Over each of the three different silhouettes a circularity can be calculated. This is done by taking the area of a silhouette and dividing it by the area of the smallest circle that covers the whole silhouette. This represents how circular the different silhouettes are. The formula is shown in Equation 20, where $I$ represents an image, and $C$ the smallest circle that covers the whole silhouette. The box plots showing the distribution of circularity over 10 example classes are shown in appendix A.7.

$$Circ(I) = \frac{Area(I)}{Area(C)} \tag{20}$$

### 3.5 Feature Vector

After the calculation of all of the previous features, a selection had to be made, which features to keep for the final vector. Multiple different combinations of features were tested, the method of testing of these vectors will be referred to in the next section Querying. The final construction of the feature vector can be seen in the figure below.

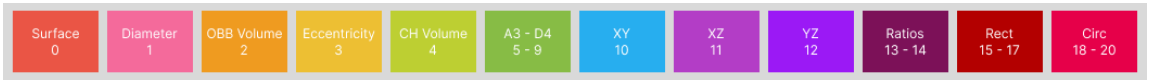



Fig. 29. Feature Vector, where $A3 - D4$ indicates all histograms as vectors, and $XY, XZ, YZ$ are the compressed silhouettes

The final feature vector consists of 21 separate features. Since the histograms are a length of 100 each and the silhouettes are a length of 175 each, when flattened the whole feature vector has a length of 1038.

## 4 STEP 4: QUERYING

There are many existing methods for comparing feature vectors like $ANN$, $Autoencoders$, and $NeuralNetworks$. These will be discussed in section 5, Scalability. These methods can be very reliable, but this section focuses on a custom distance function. Since the extracted feature vectors consists of different types of data, an attempt was made to create a custom distance function. Multiple distance functions will be used in combination to construct the final distance function. The reasoning for using different combinations, is that the scalar features could benefit from different functions than the histograms and silhouettes. The custom distance function will also make use of weights to make sure that certain features that are more descriptive will get more value in the distance. The weighting and testing of distance functions are evaluated by taking the precision. When querying object with class $c$, the precision represents the ratio of $c$ in the top 10 closest vectors.

### 4.1 Feature normalization

The features are normalized in order to ensure that no single feature disproportionately impacts the results. Standardization is used, this forces, most features to be in the same range of one standard-deviation to the average. This method is picked because it is less sensitive for outliers. The following equation is used for the standardization. Standardization is used over all scalar features since the histograms already normalized during the feature selection. The formula for standardization can be found in Equation 21, where $f_i \in$ scalar features.

$$\bar{f_i} = \frac{f_i - f_i^{average}}{f_i^{stddev}} \tag{21}$$

### 4.2 Distance Functions

As previously mentioned there are many distance functions that already are reliable for comparing vectors. In the sections below, the different distance functions that were used for constructing the custom distance function will be discussed with their benefits and downsides.

*4.2.1 Euclidean distance.* Euclidean distance is simply the length of the line between two points in Euclidean space, which is calculated using the Pythagorean theorem 22. This is one of the simplest distance function that exist, which has the advantage that the function is easy to understand. It is also applicable across multiple dimensions. However, there are a number of drawbacks on Euclidean distance. The function is sensitivity to scaling, this is why the features are standardized before computing the distance. Furthermore, for high dimensional data Euclidean distance tends to become less effective, a phenomenon known as the "curse of dimensionality", this implies that the distances between most pairs of points start to look similar, reducing the discriminatory power of Euclidean distance.

$$euc\_d(X, Y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2} \tag{22}$$

*4.2.2 Cosine distance.* While cosine similarity measures how similar two vectors are, Cosine distance measures how different they are. The equation for the Cosine similarity is already shown in Equation 4, the Cosine distance is simply 1 - the Cosine similarity. Equation 23 shows a more detailed description of this formula. If the angle between two feature vectors in n-dimensional space is small the objects have a high similarity, and so a small distance. A big advantage of

the Cosine similarity is that even if the two similar data objects are far apart by the Euclidean distance because of the size, they could still have a smaller angle between them. Smaller the angle, will result in a smaller distance.

$$cos\_d(X, Y) = 1 - \frac{X \cdot Y}{\|X\| \, \|Y\|} = 1 - \frac{\sum_{i=1}^{n} X_i Y_i}{\sqrt{\sum_{i=1}^{n} X_i^2} \cdot \sqrt{\sum_{i=1}^{n} Y_i^2}} \tag{23}$$

*4.2.3 Earth Movers Distance.* We utilize the Earth Movers Distance (EMD) to compare histograms with each other. The intuition behind this method is, that EMD shows the work that is needed to shift one distribution to the other. Capturing the minimum cost of building the smaller bars using mass taken from the larger bars and multiplying this cost over the distance that it is moved, therefore giving the distance between the distributions. It takes into account the entire distribution, rather than just specific points. This is why EMD is very useful for computing the distances between histograms. An other advantage of EMD is that the function matches perceptual similarity better than other measures, when the ground distance is perceptually meaningful. This was shown by [11] for color- and texture-based image retrieval.

$$\begin{aligned} EMD(X, Y) &= \frac{\min_F \sum_{ij} f_{ij} d_{ij}}{min(W, U)} \\ &where \\ X = (x_i, w_i) &\text{ is a feature vector } x_i \text{ with weights } w_i. \sum w_i = W \\ Y = (y_j, u_j) &\text{ is a feature vector } y_j \text{ with weights } u_j. \sum u_j = U \end{aligned} \tag{24}$$

### 4.3 Testing Distance Functions

To test what functions will be used for the final querying, it has to be determined how to deal with the different types of values in the feature vectors. The scalar features will be using a distance function together, and the for histograms and silhouettes different distance functions can be tested. The scalar features will be tested with, Euclidean distance and Cosine distance. The histogram features will be tested with, Euclidean distance, Cosine distance, and EMD. The silhouettes will be tested using, Euclidean distance, Cosine distance, EMD, and the different methods using the match shapes function in OpenCV2.

*4.3.1 Scalar Distancing.* To check how the values perform on their own, the distance for each single descriptor is calculated and precision's are tested. Table 6 shows the precision's of the scalar values when used alone to query objects in the database. These precision's can be a good indicator for weighting when combining the scalar features in one distance function. This is by no means a perfect method, but it showed good results in practice.

To check what distance function should be used for scalar features, the Euclidean distance and the Cosine distance were selected to compare. All the scalar values are used to compare these distances. They are weighted like previously mentioned based on their performance in Table 6.

| Feature | Precision | Feature | Precision |
|---|---|---|---|
| Surface | 0.0507 | Rect(XY) | 0.0508 |
| Diameter | 0.0300 | Rect(XZ) | 0.0399 |
| OBB Volume | 0.0412 | Rect(YZ) | 0.0401 |
| Eccentricity | 0.0374 | Circ(XY) | 0.0466 |
| CH Volume | 0.0497 | Circ(XZ) | 0.0499 |
| R1 | 0.0472 | Circ(YZ) | 0.0424 |
| R2 | 0.0441 | | |

Table 6. Precision's over querying with single features, with normalized values that sum to one

| Distance | Precision | Recall | F1 | Accuracy |
|---|---|---|---|---|
| Euclidean | 0.3621 | 0.1163 | 0.1691 | 0.9844 |
| Cosine | 0.3504 | 0.1136 | 0.1647 | 0.9843 |

Table 7. Precision's over querying with scalar features

When looking at Table 7, it can be seen that when maximizing precision the choice would be to make use of the Euclidean distance.

*4.3.2 Histogram Distancing.* To find out what the best distance function would be for the Histogram features, the Euclidean distance, Cosine distance and Earth's Movers Distance are compared. The histogram features include, A3, D1, D2, D3, and D4. When using all histograms combined, then they are weighted according to their precision's that they got separately using that function.

| Distance | Hist | Precision | Recall | F1 | Accuracy |
|---|---|---|---|---|---|
| Euclidean | A3 | 0.1308 | 0.0400 | 0.0585 | 0.9825 |
| | D1 | 0.0750 | 0.0213 | 0.0318 | 0.9821 |
| | D2 | 0.1326 | 0.0417 | 0.0607 | 0.9825 |
| | D3 | 0.1038 | 0.0319 | 0.0465 | 0.9822 |
| | D4 | 0.0752 | 0.0221 | 0.0327 | 0.98205 |
| | All | 0.1992 | 0.0626 | 0.0910 | 0.9831 |
| Cosine | A3 | 0.1321 | 0.0405 | 0.0591 | 0.9825 |
| | D1 | 0.0772 | 0.0220 | 0.0329 | 0.9821 |
| | D2 | 0.1344 | 0.0425 | 0.0618 | 0.9825 |
| | D3 | 0.1046 | 0.0321 | 0.0469 | 0.9823 |
| | D4 | 0.0759 | 0.0223 | 0.0329 | 0.9821 |
| | All | 0.1800 | 0.0570 | 0.0827 | 0.0569 |
| EMD | A3 | 0.1093 | 0.0327 | 0.0480 | 0.9823 |
| | D1 | 0.0515 | 0.0154 | 0.0228 | 0.9819 |
| | D2 | 0.0769 | 0.0231 | 0.0341 | 0.9821 |
| | D3 | 0.0671 | 0.0189 | 0.0282 | 0.9820 |
| | D4 | 0.0588 | 0.0165 | 0.0247 | 0.9819 |
| | All | 0.1835 | 0.0565 | 0.0825 | 0.9829 |

Table 8. Precision's over querying with histogram features

When we look at Table 8 we can see that A3 in general performs better then the rest, but the performance of all histograms clearly change when using different functions. The expectation was that EMD would perform the best of the distance functions, but after the analysis we can see that Euclidean performs better then the others.

*4.3.3 Silhouette Distancing.* To find out what the best distance function would be for the Silhouette features, the Euclidean distance, earth's movers distance, and Cosine distance. The silhouette features include, XY, XZ, and YZ. When using all silhouettes combined, then we weight according to their precision's that they got separately using that function.

| Distance | Silhouette | Precision | Recall | F1 | Accuracy |
|---|---|---|---|---|---|
| Euclidean | XY | 0.1894 | 0.0593 | 0.0863 | 0.9830 |
| | XZ | 0.1783 | 0.0553 | 0.0804 | 0.9829 |
| | YZ | 0.1652 | 0.0518 | 0.0753 | 0.9828 |
| | All | 0.2942 | 0.0941 | 0.1364 | 0.9838 |
| Cosine | XY | 0.1810 | 0.0565 | 0.0821 | 0.9829 |
| | XZ | 0.1607 | 0.0502 | 0.0729 | 0.9828 |
| | YZ | 0.1692 | 0.0517 | 0.0757 | 0.9828 |
| | All | 0.2371 | 0.0767 | 0.1106 | 0.9834 |
| EMD | XY | 0.1676 | 0.0530 | 0.0768 | 0.9828 |
| | XZ | 0.1687 | 0.0510 | 0.0747 | 0.9828 |
| | YZ | 0.1578 | 0.0483 | 0.0707 | 0.9827 |
| | All | 0.3095 | 0.0994 | 0.1441 | 0.9839 |

Table 9. Precision's over querying with silhouette features

When we look at Table 9 we can see that the performance of the different distance functions is quite similar. The expectation was that the Euclidean distance would give the best performance when looking at the precision's of the silhouettes separately, but when combining the three silhouettes the performance increases with EMD. This is an indication that EMD should be used in the custom distance function.

Silhouettes also were matched using two extra methods, without the compression of the grid. This is done using a shape matching function provided by OpenCV (matchShapes based on hueMoments). The first method just makes use of the matchShapes function, the second also compares the distances between silhouettes between all axis, and takes the closest distance. The orientation of the images might not always be the same, the matchShapes function ignores the rotation of the input shape. That together with the cross comparing would catch any object that isn't rotated similarly. After testing we found that the performance significantly decreased when using these methods, the same goes for cross comparing with EMD. Why exactly this happens is hard to say, but the hypothesis is that the current silhouette is getting an increase in performance because of the rotation. When removing the rotation as part of these features, the objects that already were rotated correctly are more likely to be matched with other objects. Due to these results, it was opted to use EMD without cross comparing for the custom distance function.

### 4.4 Combining the custom distance function

When combining the different distance functions that were selected based on the previous tests, they can be combined into a single function. This can be done by just summing them, but to allow for more weighting options, an extra layer of weights are added. This also requires an extra step of normalizing the distance outputs.

To calculate the Euclidean distance over all the scalar features, a weighted Euclidean distance function was used. This slight alteration is shown in Equation 25. In Equation 26 the distance for the scalar features is calculated, where $t_{scalar}$ is the target vector that contains only scalar features, and the same goes for the vector $v_{scalar}$, and the weights $w_{scalar}$.

$$euc\_d(X, Y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2 * w_i} \tag{25}$$

$$d\_scalar(t, v) = weigted\_euc\_d(t_{scalar}, v_{scalar}, w_{scalar}) \tag{26}$$

To calculate the Euclidean distance over the histograms, the formula in Equation 27 is used. $H$ indicates the indexes of the histograms inside the vectors, and $w_i$ the corresponding weight.

$$d\_hist(t, v) = \sum_{i \in H} euc\_d(t_i, v_i) * w_i \tag{27}$$

To calculate the earths movers distance over the silhouettes, the formula in Equation 28 is used. $S$ indicates the indexes of the silhouettes inside the feature vector, and $w_i$ the corresponding weight.

$$d\_sil(t, v) = \sum_{i \in S} EMD(t_i, v_i) * w_i \tag{28}$$

For getting the closest objects to a target object, all the distances have to be calculated for each object in the database. Then these distances are normalized so that the largest value for each distance is one. Then a weighted sum is done for these vectors, based off the precision's that the distances got in testing.

Since the hypothesis for the histograms was to use the EMD distance function, two tests were done to compare in the scope of all the features how EMD compares to Euclidean for histograms. In Table 10 the results of these tests can be seen, even though slightly the Euclidean distance does indeed perform better when being used for the histograms in the generated feature vector.

| Distance | Precision | Recall | F1 | Accuracy |
|---|---|---|---|---|
| Cdf with Euclidean for histograms | 0.3849 | 0.1254 | 0.1816 | 0.9845 |
| Cdf with EMD for histograms | 0.3756 | 0.1215 | 0.1762 | 0.9845 |

Table 10. Precision's over querying with scalar features

With this comparison done it was chosen to used the Cdf with Euclidean distance for histograms, and when the custom distance function is referenced further in the paper, this function was used. This means that our custom distance function is able to get a precision of 0.3849.

## 5 STEP 5: SCALABILITY

The speed of matching the input feature vector to the rest of the feature vectors in the database is fast at the moment. However, when using a larger database, the naive implementation of looping through all the features vectors in the dataset and comparing them to the input vector will become much slower. This is why a number of scalability options will be implemented and their performance will be analyzed.

### 5.1 Approximate Nearest Neighbours

We implemented Approximate Nearest Neighbors (ANN) using the library PyNNDescent [3] which implements the ANN search according to the paper [5]. After testing several combinations of distance functions, leaf sizes and number of trees, we achieved the best results using Cosine similarity, number of neighbors at 10 and leaving the remaining parameters as their standard values. The resulting performance was sufficient and will be further explained in the results section.

### 5.2 Neural Networks

*5.2.1 Fully connected and convolutional neural networks.* We want to compare the performance of neural networks to our own distance function. A neural network is able to find its own optimal weights using a loss function, gradient descent and back-propagation [7]. The first neural network will be used with our own features and therefore it will only try to optimize the weights. The neural network we use is a fully connected network which will take the scalar features + the flattened histogram features (+ for our second run also the silhouette features). We have 5 scalar features and 5 histogram features with 100 bins each. This equals to an input vector of size $5 + 5 * 100 = 505$ or $505 + 533 = 1038$ when including the silhouette features. The output layer will be of size 69 as we treat this as a classification between the 69 different object classes. By doing this we have to utilize the true labels of the classes, which is a piece of information that we previously did not use when creating our own distance function to match similar shapes. Therefore the results from the (Fully connected and Convolutional) neural networks cannot be adequately compared to the results of our own distance function. Additionally, a neural network can only predict the label of an unseen object in the test set and the accuracy is determined by the correctness of that prediction. The neural network cannot retrieve a set of similar shapes to the input shape. The fully connected neural network consists of the following layers. The number behind the Dense layers denote the number of neurons in that layer. Dropout layers [12] are used to prevent over-fitting of the network, the number denotes the percentage of dropped connections:

- Input (size= 505 or 1038)
- Dense (256)
- Dropout (0.2)
- Dense (512)
- Dropout (0.2)
- Dense (128)
- Dense (69)

The second test will be conducted with a Convolutional Neural Network (CNN) [9]. Here, the input will consist solely of the 2d images of the object silhouettes. The CNN is able to extract the features by using convolution layers, which contain the weights that will be trained, max pooling layers for reducing the dimensions and dense layers. The input size is 100,100,3 for the 3 different axes per object which create a 100x100 sized 2d image. The following architecture is implemented. The number behind the Convolution layers denotes the number of filters:

- Input (size=100,100,3)
- Conv2D (32)
- MaxPooling
- Conv2D (64)
- MaxPooling
- Conv2D (64)
- Flatten
- Dense(64)
- Dense(69)

The following table shows the resulting test set accuracy's after training. We should keep in mind that this refers to the classification accuracy, or what could also be stated as top-1 accuracy, when trying to retrieve the top-k most similar shapes. Therefore these results are not comparable to the other results in this report and are shown in this section instead of the evaluation section.

| | FCNN | FCNN with silhouettes | CNN |
|---|---|---|---|
| Test set Accuracy | 52.2% | 67.6% | 52.8% |

The FCNN shows a test set accuracy of 52.2% which is improved to 67.6% when adding the silhouette related features. The CNN that is purely trained on the silhouettes is able to achieve a 52.8% accuracy.

*5.2.2 Autoencoders.* Contrary to the previous neural networks, autoencoders do not perform classification nor do they require access to a ground truth such as a label. They learn by attempting to reconstruct their input data through a process of encoding and decoding. The training objective is to minimize the difference between the input data and the reconstructed output [8]. After training, the middle layer acts as small-sized, latent representation of the input. We can query all objects into the autoencoder and utilize this new vector to compute the distances between the objects. We trained two autoencoders, the first one takes the 2d silhouettes as input and tries to reconstruct them using convolution layers. The following architecture was used:

Encoder:

- Conv2D (16)
- MaxPooling
- Conv2D (32)
- MaxPooling
- Dense (10)

Decoder:

- Conv2D (32)
- UpSampling

- Conv2D (16)
- UpSampling
- Conv2D (3)

The second autoencoder tries to replicate the same 1038 sized feature vector as we used earlier. Since it only consists of flattened numerical values we do not need convolution layers but only dense layers. The architecture is the following:

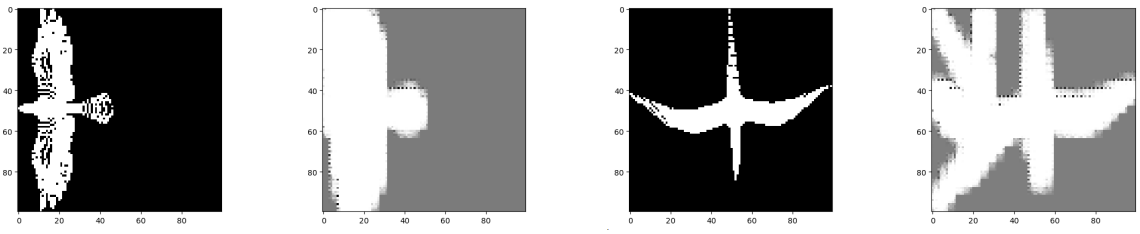
Encoder:

- Dense (128)
- Dense (256)
- Dense (256)
- Dense (128)
- Dense (30)

Decoder:

- Dense (128)
- Dense (256)
- Dense (256)
- Dense (128)
- Dense (1038)

Figure 30 shows two input images as well as the reconstructed image as created by the autoencoder.



(a) Original image (b) Reconstructed image (c) Original image (d) Reconstructed image

Fig. 30. Original images and the reconstructed output from the CNN autoencoder

The results of the autoencoders will be shown in chapter evaluation.

### 5.3 PCA

When conducting the Principal Component Analysis (PCA), the two largest eigenvectors e1, e2 are extracted and the 1038 dimensional feature vector is projected onto the subspace spanned by e1, e2. We preserve this way the most variance in the feature vector that we can describe with only 2 dimensions. PCA assumes a linear relationship between variables, for this reason we do not expect the two dimensions obtained by PCA to perform well in the matching process.

### 5.4 t-SNE

Instead of only testing the PCA for the dimensionality reduction two more methods are tested. Next, t-distributed Stochastic Neighbor Embedding (t-SNE) [13] is used. This method is preferred due to the fact that t-SNE not only retains global variance, but also retains local variance. Which is very important for the dataset of feature vectors, because there are many locally clustered classes in this data. T-SNE is a machine learning algorithm that generates slightly different results each time on the same data set, focusing on retaining the structure of neighbor points.

By using t-SNE for the dimensionality reduction the dimensions of the data is reduced from 1038 to 2 dimensions. Perplexity is set to 20, this value can be thought of as the number of neighboring points t-SNE must consider; which is 61 in this case. The number of iterations is set to 450 This number is based on the fact that after 450 iterations the precision of matching objects with the reduced 2 dimensions does not increase anymore, while the run-time does increase. The results of the reduced dimensions obtained by applying t-SNE on the feature vectors are shown in Figure 31. Objects are clustered closely with objects in the same class, which shows that the local structures are preserved well. In Figure 32 a subset of classes is shown which are very well clustered, these include mainly classes with a high similarity of object in the concerned class. On the other hand, in Figure 33 a subset of eight classes is shown for the classes for which the two dimensional vectors are not closely clustered, these classes most of the time have many differently shaped objects in them. Resulting in that the class already has a bad performance on matching before dimensionality reduction, an overview per class performance will be shown in section 6.

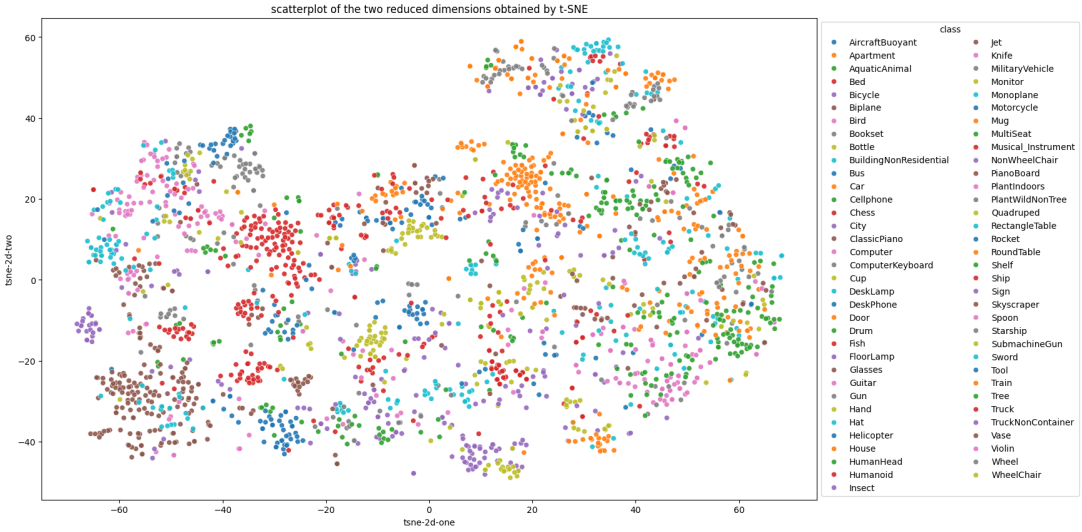



Fig. 31. The reduced dimensions obtained by t-SNE for all classes in the database.

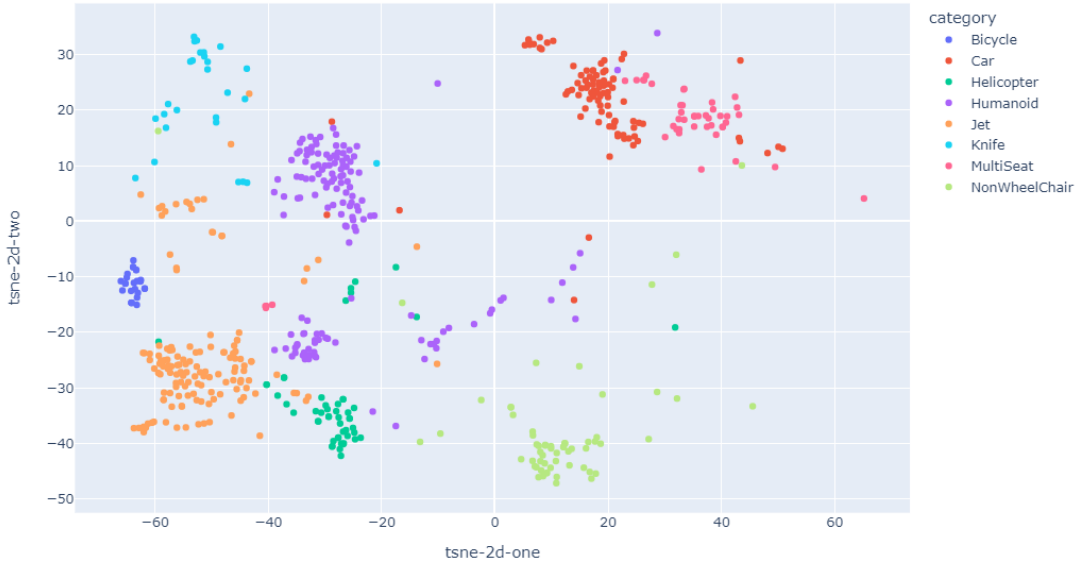



Fig. 32. The reduced dimensions obtained by t-SNE for a subset of eight classes where the objects in each class are very well clustered together.


Fig. 33. The reduced dimensions obtained by t-SNE for a subset of eight classes where the objects in each class have a wide spread and are not clustered with items from the same class.

### 5.5 UMAP

A very similar algorithm to t-SNE is the Uniform Manifold Approximation and Projection (UMAP) [10] algorithm. The main difference between t-SNE and UMAP is the interpretation between the objects and clusters. Where t-SNE mainly tries to preserve local structures. UMAP claims to preserve both local and global structures in the data. Moreover, UMAP does a better job on scalability than t-SNE, with a superior run-time performance. The n_neighbors parameter is set to 6, this resulted in the highest precision. An interesting observation is that the amount of neighboring points that t-SNE and UMAP use to perform best on this dataset are very different. This can probably be explained by the inner working of the algorithms, but that is outside the scope of this paper. The scatter plot in Figure 34 shows the resulting two dimensions computed by the UMAP algorithm.

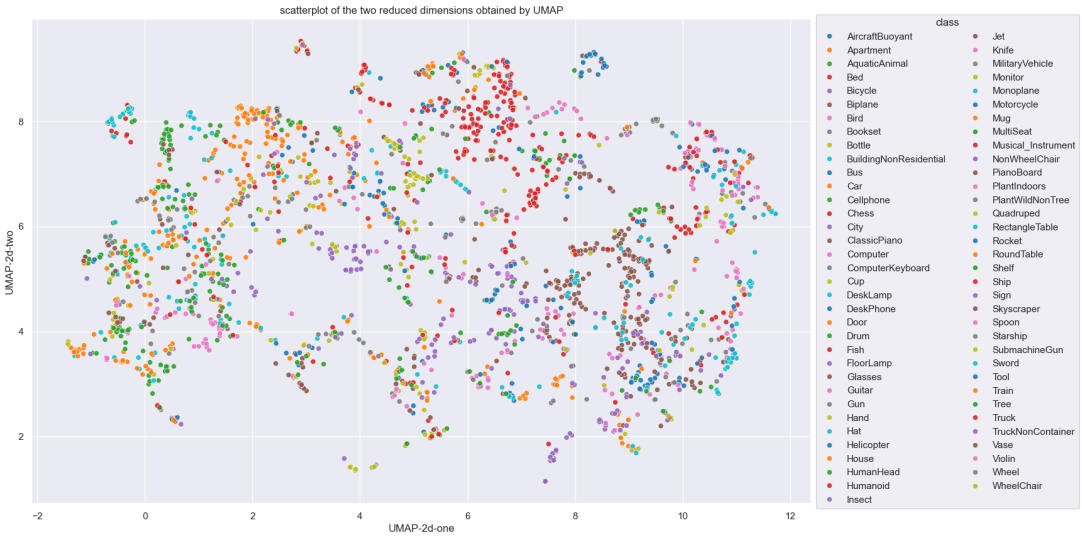



Fig. 34. The reduced dimensions obtained by UMAP algorithm for all classes in the database.

The scatter plot shown in Figure 34 shows the reduced two dimensions for for the same classes as show for the t-SNE, but now computed by the UMAP algorithm. When analysing this scatter plot visually it seems that the local structures in the data are a bit less preserved than with the t-SNE. You can see less clustering per class, however, many classes are still close to each other in the reduced dimensional space. How well these dimensionality reduction algorithms perform, compared to the each other, and to the original matching function on the 1038 dimensional vectors will be shown in the next section.

## 6 STEP 6: EVALUATION

Many strategies have been used to match the feature vector extracted from the mesh to the feature vectors of all other shapes in the database. With the goal of returning the k most similar items. In order to evaluate these strategies a number of evaluation metrics are used. The evaluation process is conducted on the top 10 retrieved items.

### 6.1 Metrics

Precision, recall, accuracy, F1-score, sensitivity and specificity are used to evaluate the results obtained by system. The most important metric for the evaluation of the matching strategies is the precision (Equation 29). This is the fraction of relevant instances among the retrieved instances. The precision will mainly used to determine how well the different retrieval approaches for the system perform. However, the other metrics are still useful to get insights in the the output of the system. The recall (Equation 30), also known as sensitivity, is the fraction of relevant instances that are retrieved. This metric will always be low when only retrieving 10 instances. The accuracy (Equation 31) shows how many items are classified correctly as positive or negative, due to the fact that the the number of True Negatives is always very high when retrieving only 10 items the accuracy will be high as well. The F1-score (Equation 32) is calculated as the harmonic mean of the precision and recall. Finally, the specificity (Equation 33)is computed, this metric shows how well the system can predict negative outcomes.

$$Precision = \frac{TP}{TP + FP} \tag{29}$$

$$Recall = \frac{TP}{TP + FN} \tag{30}$$

$$Accuracy = \frac{(TP + TN)}{(TP + FP + TN + FN)} \tag{31}$$

$$F1\ score = \frac{2 * Precision * Recall}{Precision + Recall} \tag{32}$$

$$Specificity = \frac{TN}{TN + FP} \tag{33}$$

### 6.2 Evaluation process

In order to compute the metrics of section 6.1 the True Positive (TP), False Positives (FP), True Negatives (TN) and False Negatives (FN) need to be computed. The process to obtain these values is as follows; for every object in the database:

- Target class = the ground truth the queried object
- Class size = the size of the class of the queried object
- Retrieve top 10 most similar items (without object itself).
- TP = number of object in top 10 with the same class as Target class
- FP = 10 - TP
- FN = Class size - TP
- TN = database size - TP - FP - FN
- The mean of these four computed values is taken for every class in the database.

- The mean over all classes is taken to get the final TP, FP, FN and TN values that will be used in the evaluation metrics.

The mean over all classes is taken to compute the final evaluation metrics. The reason for this is that the database is not balanced equally, some classes have a much bigger size than others. The mean for every class is used instead of over all object in order to mitigate this inequality, resulting in a fair weighting of all different object classes to the final evaluation values.

The first retrieved item is left out because this is the queried object itself. It is assumed that the system will be used to match a new 3D shape to the shapes in the database. Due to this it is fair to exclude the object itself from the resulting output, because in a real word scenario this exactly similar object would not be in the database. This makes the results of the evaluation more realistic. One should keep in mind that had we decided to calculate the top-10 metrics with the query object itself included, then they would have turned out higher, but we chose to not do so to create a more representative real-world metric.

### 6.3 Results

| K = 10 | Custom Distance Function | PCA | t-SNE | UMAP | Autoencoder CNN (2d silhouettes) | Autoencoder FCNN (all features) | ANN |
|---|---|---|---|---|---|---|---|
| Precision | 0.385 | 0.075 | 0.357 | 0.254 | 0.249 | 0.371 | 0.387 |
| Recall (sensitivity) | 0.125 | 0.022 | 0.115 | 0.082 | 0.079 | 0.120 | 0.126 |
| Accuracy | 0.985 | 0.982 | 0.984 | 0.984 | 0.983 | 0.984 | 0.985 |
| F1-score | 0.182 | 0.032 | 0.167 | 0.119 | 0.115 | 0.174 | 0.182 |
| Specificity | 0.997 | 0.996 | 0.997 | 0.997 | 0.997 | 0.997 | 0.997 |

Table 11. Evaluation of the top 10 results of all matching techniques. The first column show the computed metrics for the evaluation based on our own distance- and weighting function on the full feature vectors. Followed by three dimensionality reduction strategies, and three neural network approaches.

*6.3.1 Quality metrics.* The results of the evaluation are shown in Table 11. As mentioned before in section 6.1 the recall, accuracy, F1-score and specificity are very similar for all approaches. This is caused by only retrieving the top 10 most similar instances. The metric that is most important for measuring the performance of the system and comparing the different approaches is the precision. The precision of the custom distance function is 38.5%. Although this value seems low, there are a number of reasons why this precision score of 38.5% is good. First of all, many classes in the database have two or more sub-classes, for instance the *AircraftBuoyant* class includes hot air balloons as well as zeppelins. Better categorization of these sub-classes would improve the precision. Next to that, many classes have overlapping similarities, there are for instance three aircraft classes in the database. This results in many false positives while the object is actually very similar to the queried object.

Of the dimensionality reduction approaches t-SNE performs best (35,7%), followed by UMAP (25.4%) and PCA (7.5%). What is remarkable is that the precision of t-SNE is almost as high as the precision of the custom distance function, while only using two dimensions feature vectors instead of the full 1038 dimensional feature vectors. A possible explanation

for why t-SNE performs better than UMAP is that t-SNE looks more on local structure while UMAP also looks at global structure. Since there is a lot of local structure in our feature vector database and less global structure, this would explain the higher performance of the t-SNE. The Autoencoder CNN (37.1%) performs better than the custom distance function, this was expected since a neural network is better in computing the correct weights for the importance of the different features than the hand-tailored weighting function. Finally, the ANN has the highest performance, with an precision of 38,7%. Since the many different approaches on computing the top 10 most similar items, based on the feature vectors, results in very similar mean precision it is likely that improving the precision is held back by the computed features.

*6.3.2 Performance per class.* In Figure 35 and 36 two overviews are given of the mean precision per class for respectively the custom distance function and the t-SNE algorithm. There are many classes that have a very good performance. Figure 36 shows that the *Bicycle* class has a stunning performance of 1, meaning that for every input object in the *Bicycle* class all returned objects are bicycles as well. The two plots also show that a number of classes perform really bad, these classes decrease the overall precision of the system a lot. From these bar charts it becomes clear that there is a difference in performance per class between the different matching algorithms. For instance; the class *City* performs way worse for the custom distance function than for t-SNE. However, the overall order of best performing classes is more or less the same, except for a number of instances where a few classes have been swapped.

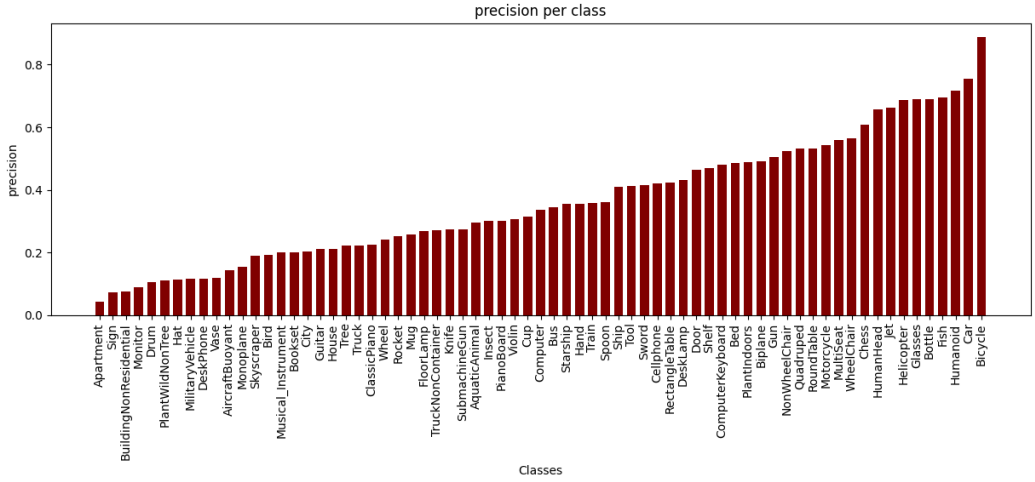



Fig. 35. Mean precision per class for the custom distance function on the full 1038 dimensional weighted feature vectors

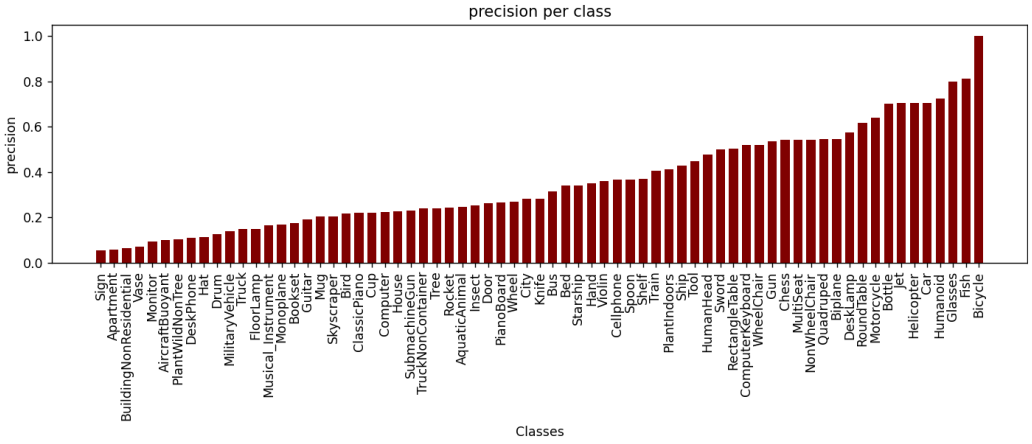



Fig. 36. Mean precision per class for the matching function using the two dimensional vectors obtained by t-SNE

*6.3.3 Relative Operating Characteristic Curve.* Relative operating characteristic (ROC) curve is a graphical representation of the relationship between sensitivity and specificity of testing over the possible query size parameters. In order to assess how the MR system performs if these metrics are inversely correlated we will 'factor out' the query size parameter, and plot the sensitivity and specificity for ALL possible values of the query-size parameter. The blue curve in Figure 37 shows the dependency between the specificity and sensitivity for the Custom Distance Function (section 4.4). An ideal system would have its ROC curve close to the upper-right square edges, which means low FP and low FN. While a straight line would mean that the performance of the system is random. The ROC curve in Figure 37 is not a straight line and is moved to the upper right corner. This shows that our MR system performs much better than random. Looking at the curve our system has an average performance, which can be improved way more in order to push the ROC curve to the upper right corner of the plot. The Area Under ROC (AUROC) value is 0.76, which indicated a better than random performance. However, there are still improvements to be made in order to get the AUROC value closer to one.

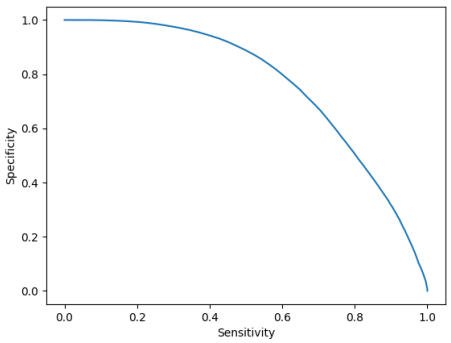



Fig. 37. Relative Operating Characteristic curve showing the dependency between the specificity and sensitivity for all possible values of the query size parameter.

### 6.4 Interface

An interface is created in order to make the content based 3D shape retrieval system easily accessible for users. PyVista [1] is used for creating the interface. In the interface, shown in Figure 38, an user can choose between selecting an object from the database or uploading a file of a 3D shape from their own computer. This object will be visualized in the small box at the left. After this the user can select a matching method, at this moment only the custom distance function and the t-SNE are possible options. These options can be easily supplemented with the other matching methods. When the user clicks on the 'run query' button the input object will go trough the pipeline of the retrieval system and the 10 closest shapes to that object will be returned. This takes around 30 seconds for the custom distance function and 25 seconds for the t-SNE. We aimed at keeping the computation time under 30 second, so this goal is accomplished. The shapes can be rotated in the interface in order to properly inspect them. The past queries of the user will be stored in the interface, in order for the user to compare different queries to each other. Every row represents one past query. At this moment the interface is designed to show the top 10 images for a query, however, this can easily be adjusted so that the user can select a top-k of retrieved items. In order to test out the interface of our system we can use the System Usability Scale (SUS) [4]. This will be an option for future work.


Fig. 38. Graphical User Interface for the 3D content based retrieval system.

## 7 DISCUSSION

The entire project has many potential sections which can cause lower than expected retrieval performance. An unclean dataset, remeshing functions that don't perform consistently, problems computing the volume and many different features with different impact on the similarity calculations. It was an interesting journey to go through all these different steps and in the end have a working content based shape retrieval system with decent performance. While a recall of 38.7% may not seem that high, there are many classes that are distinguished quite well by our system and it performs competitively when compared to neural networks that use much larger feature input. Lets examine some queries. Figure 39 shows the top 10 closest objects to a queried bicycle object. We could already see based on the per class precision metric that this class performs well and indeed all retrieved objects in this case are bicycles.

Fig. 39. Retrieval result for an object of class Bicycle

The next queried object is of the class Rocket. Most of the top 10 retrieved object are of the same class, however there is also a jet, two fish and two submarines (class Ship). While the precision in this query would only be 50% we can observe that the objects are quite similar. The long shape, with thin fins attached and a pointy tip are features present in all retrieved items.

Fig. 40. Retrieval result for an object of class Rocket

Here we can see an object that performs poorly. The class tree returns very different objects, that often do not have a high similarity. We can see that in this case, objects of the classes drums, Bird, Tables and more were returned.

The neural networks using supervised learning performed similar or worse compared to our handcrafted features and manually adjusted distance function, when looking at top-1 accuracy. The worse performance from the Convolutional

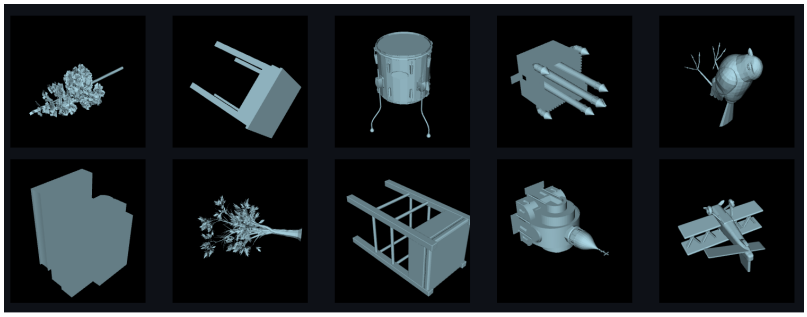


Fig. 41. Retrieval result for an object of class Tree

Neural Network is not a surprise, given that it is missing a lot of information compared to the handcrafted features. The 2d silhouettes from the three different axes are a decent representation of the object, but they do not capture the full complexity of a 3d object as well as manually handcrafted features. Overall the performance of our own features is even better compared to the Neural networks if you keep in mind that it does not use the information of true class labels and the feature vector is only of size 1038 compared to the $100 * 100 * 3 = 30000$ size of the 3d silhouettes used for the CNN. The autoencoder using our crafted feature vector shows similar performance to our own distance function. The CNN autoencoder using the silhouettes performs worse than our function for similar aforementioned reasons.

### 7.1 Limitations

There are several circumstances that hindered us from achieving greater performance. Some meshes had inverted normals, which would lead to problems when calculating the features. Other meshes had several large holes and some meshes were cut into multiple pieces. Since we decided to leave all objects in the database instead of removing these outliers we knew that we will achieve less performance. Surface reconstruction for computing the volume turned out to be one of the most challenging tasks, mostly due to the aforementioned problems with the meshes but also the several remeshing functions from Open3D and PyMeshLab sometimes caused inverted meshes or holes. Holestitching would work in some cases but it was not able to fix certain meshes with too many issues.

Another problem is that the database classes sometimes contain objects that are vastly different to each other. As an example, the class *ship* contains aircraft carriers, viking sail-ships, submarines and many more. With our features alone those shapes are too different to be classified as the same class reliably. Often these classes would have more similarity to objects of other classes, such as the submarine being similar to fish or rockets. Other classes are even worse. *BuildingNonResidential*, which performed very bad in all our tests, contains objects so different from each other that many of them can not be reasonably considered as being from the same class.

### 7.2 Future work

The most important step for future research is finding a reliable way of remeshing and hole stitching, so that the volume based features can be used for all objects effectively. This would include making the normals in the meshes consistent. There is also room for improvement when training the neural networks, such as parameter tuning. The 2d silhouettes could be taken from more angles or from different distances from the center, which would increase the number of

features and possibly also the performance, but this would have a trade-off between feature vector size and possible performance improvements.

### 7.3 Conclusion

In the limited time there was to develop the entire pipeline of this content-based 3D shape retrieval system a precision score of 38.7% is accomplished using the ANN model. At first sight this score might seem low, however, with all the challenges and complications we faced during the creation of this system we are very satisfied with this result. It all comes down to the fact that handling, adjusting and reconstructing three dimensional shapes is a challenging task. Even with the many libraries that already exist there still is none that has perfected these tasks. There are certainly several improvements possible over the entire pipeline.

## REFERENCES


[1] [n. d.]. https://docs.pyvista.org/version/stable/

[2] [n. d.]. *Open3D – A Modern Library for 3D Data Processing.* http://www.open3d.org/

[3] [n. d.]. *PyNNDescent for fast Approximate Nearest Neighbors - pynndescent 0.5.0 documentation.* https://pynndescent.readthedocs.io/en/latest/index.html

[4] John Brooke. 1996. Sus: a "quick and dirty'usability. *Usability evaluation in industry* 189, 3 (1996), 189–194.

[5] Wei Dong, Charikar Moses, and Kai Li. 2011. Efficient K-nearest neighbor graph construction for generic similarity measures. *Proceedings of the 20th international conference on World wide web* (2011). https://doi.org/10.1145/1963405.1963487

[6] prefix=van der useprefix=false family=Ree, given=Max. [n. d.]. *MaxRee94/ShapeDatabase_INFOMR.* https://github.com/MaxRee94/ShapeDatabase_INFOMR

[7] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning.* MIT Press.

[8] G. E. Hinton and R. R. Salakhutdinov. 2006. Reducing the Dimensionality of Data with Neural Networks. *Science* 313, 5786 (2006), 504–507.

[9] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[10] Leland McInnes, John Healy, and James Melville. 2018. Umap: Uniform manifold approximation and projection for dimension reduction. *arXiv preprint arXiv:1802.03426* (2018).

[11] Yossi Rubner, Carlo Tomasi, and Leonidas J Guibas. 1998. A metric for distributions with applications to image databases. In *Sixth international conference on computer vision (IEEE Cat. No. 98CH36271)*. IEEE, 59–66.

[12] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*.

[13] Laurens Van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of machine learning research* 9, 11 (2008).

[14] EIERMANN M. de Boer B. Vrooman, J. 2023. MR Project repository. https://github.com/JesseFloren/Multimedia-Retrieval/.

# A HISTOGRAMS

## A.1 A3

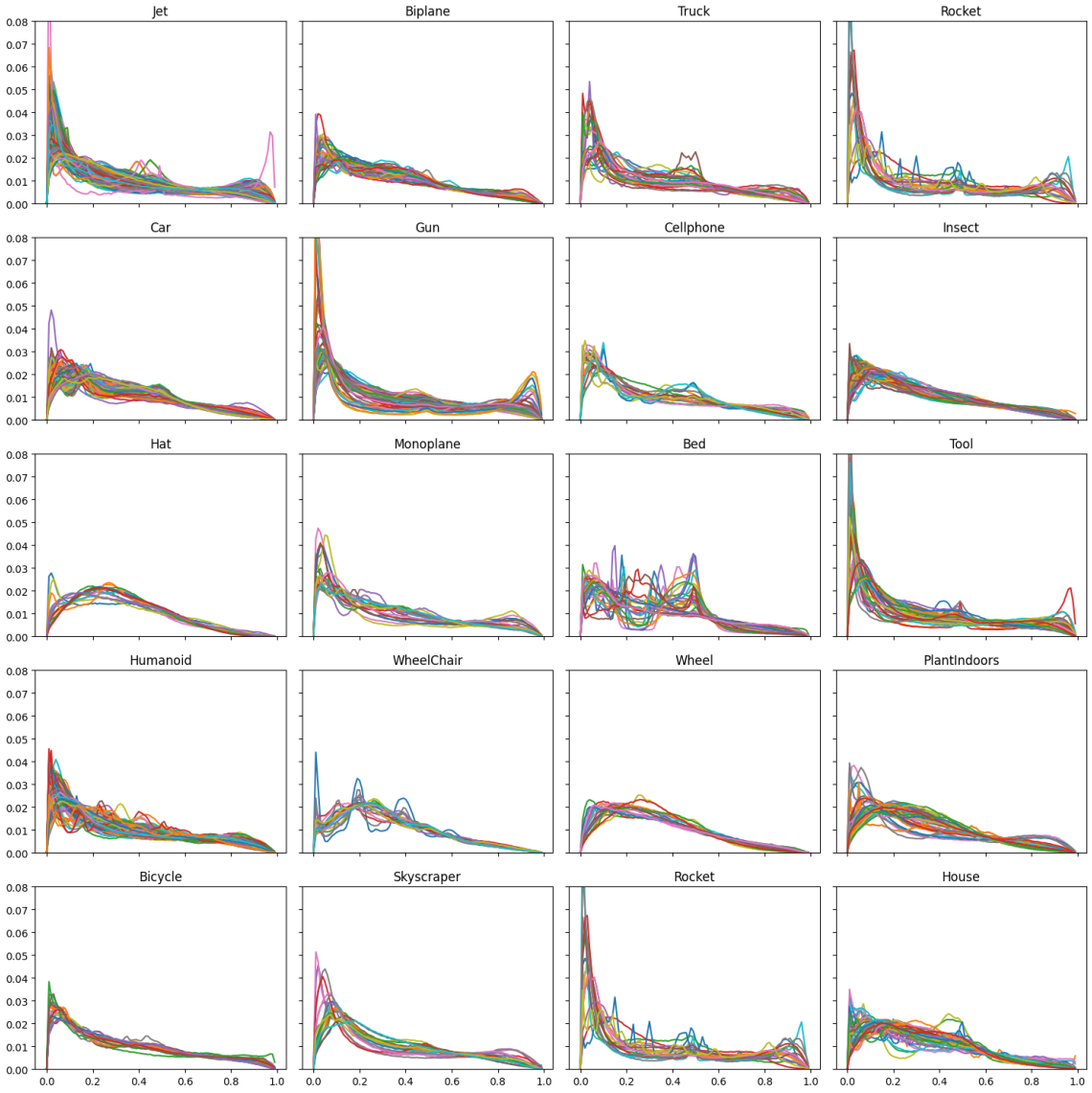



Fig. 42. A3 histogram of the objects within the classes

### A.2 D1

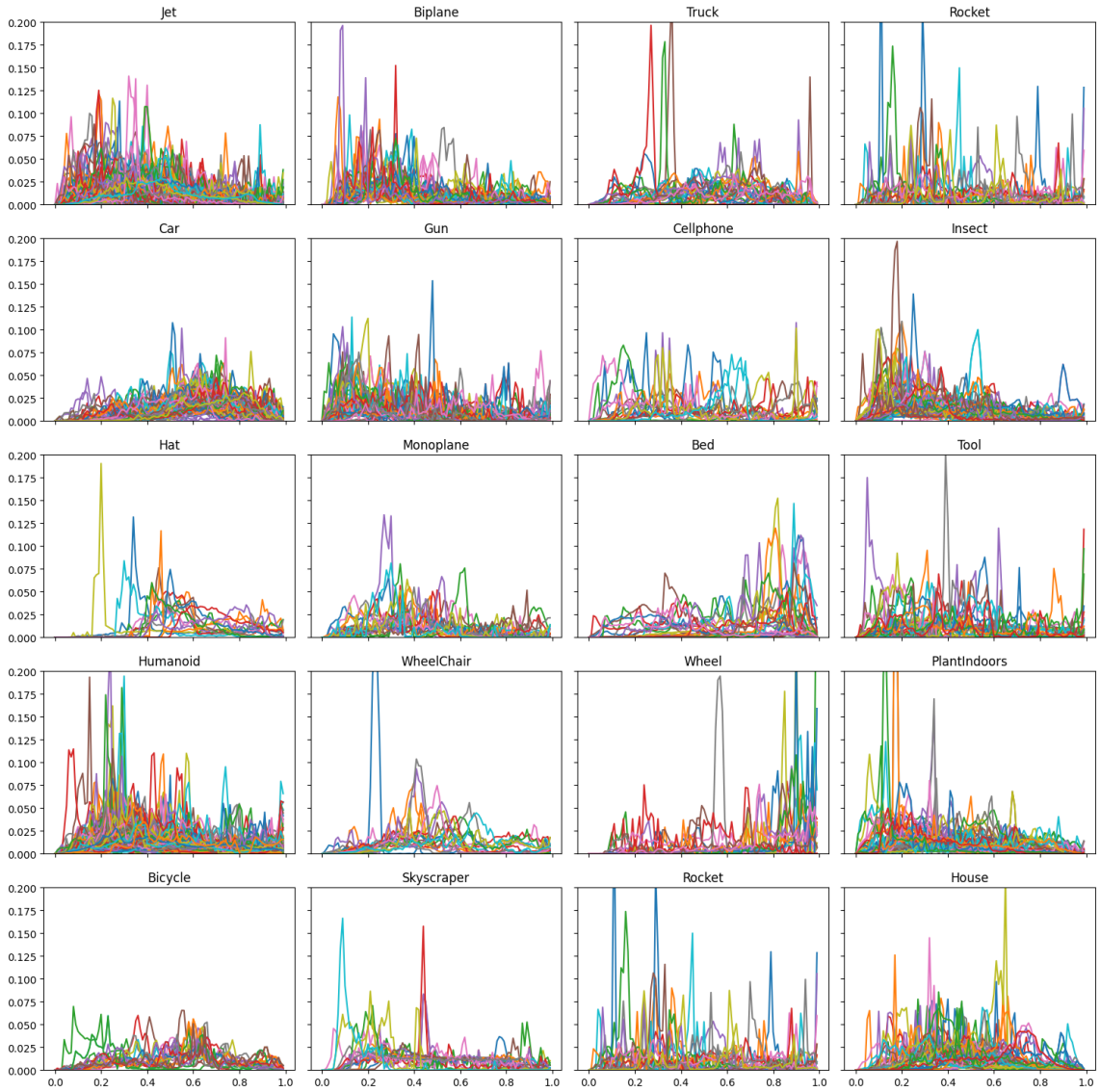



Fig. 43. D1 histogram of the objects within the classes

### A.3 D2

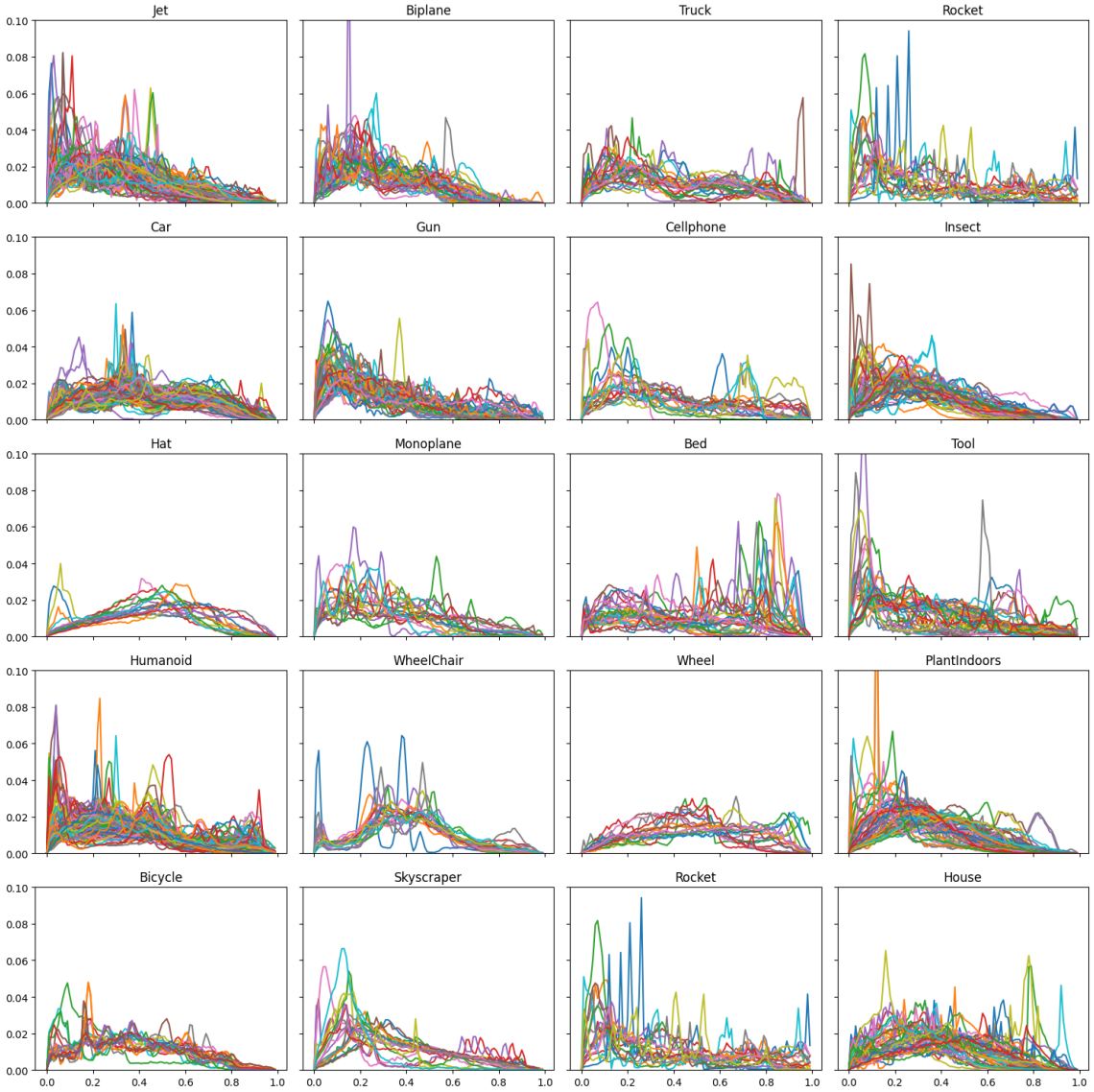



Fig. 44. D2 histogram of the objects within the classes

### A.4 D3

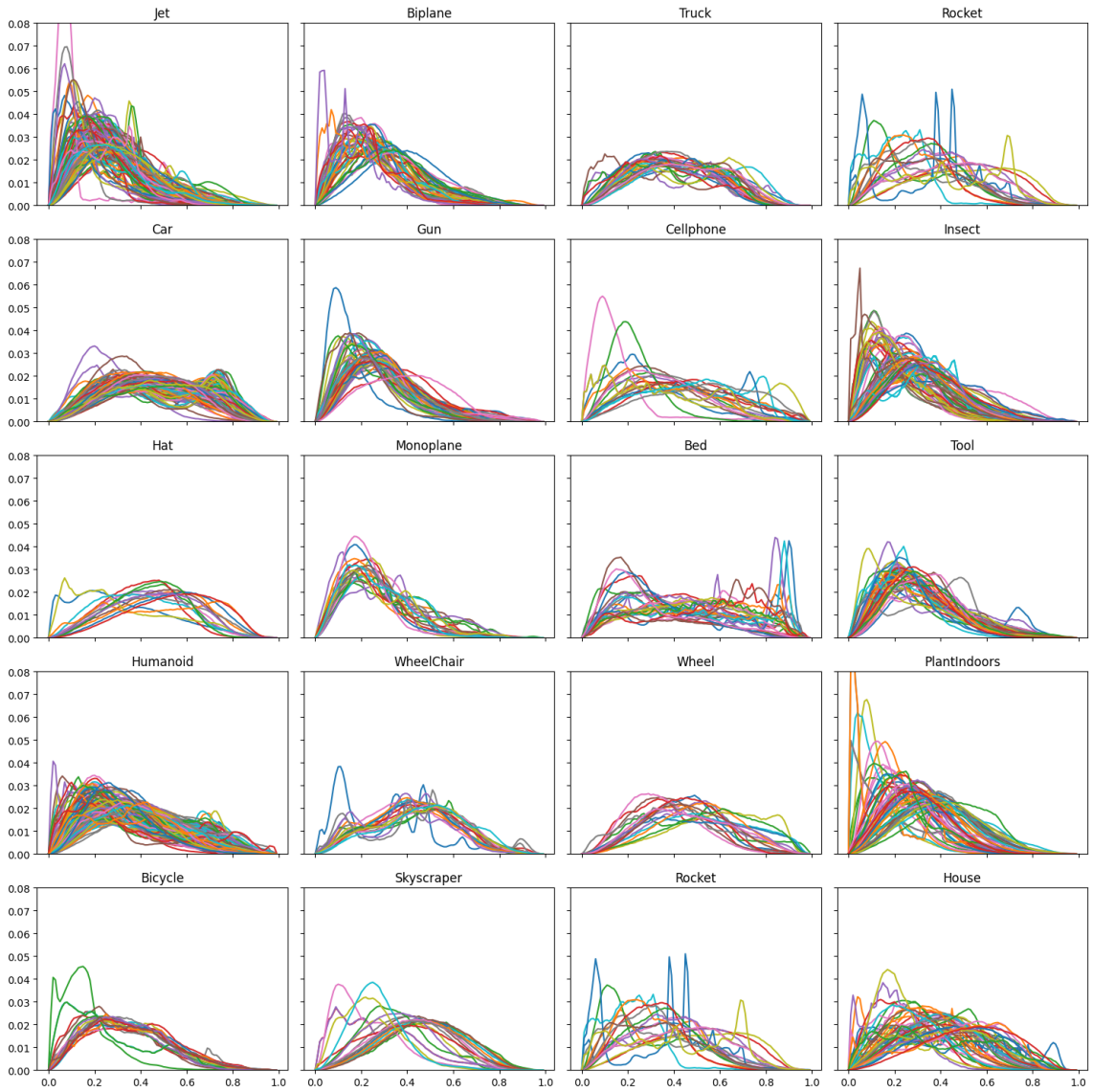



Fig. 45. D3 histogram of the objects within the classes

### A.5 D4

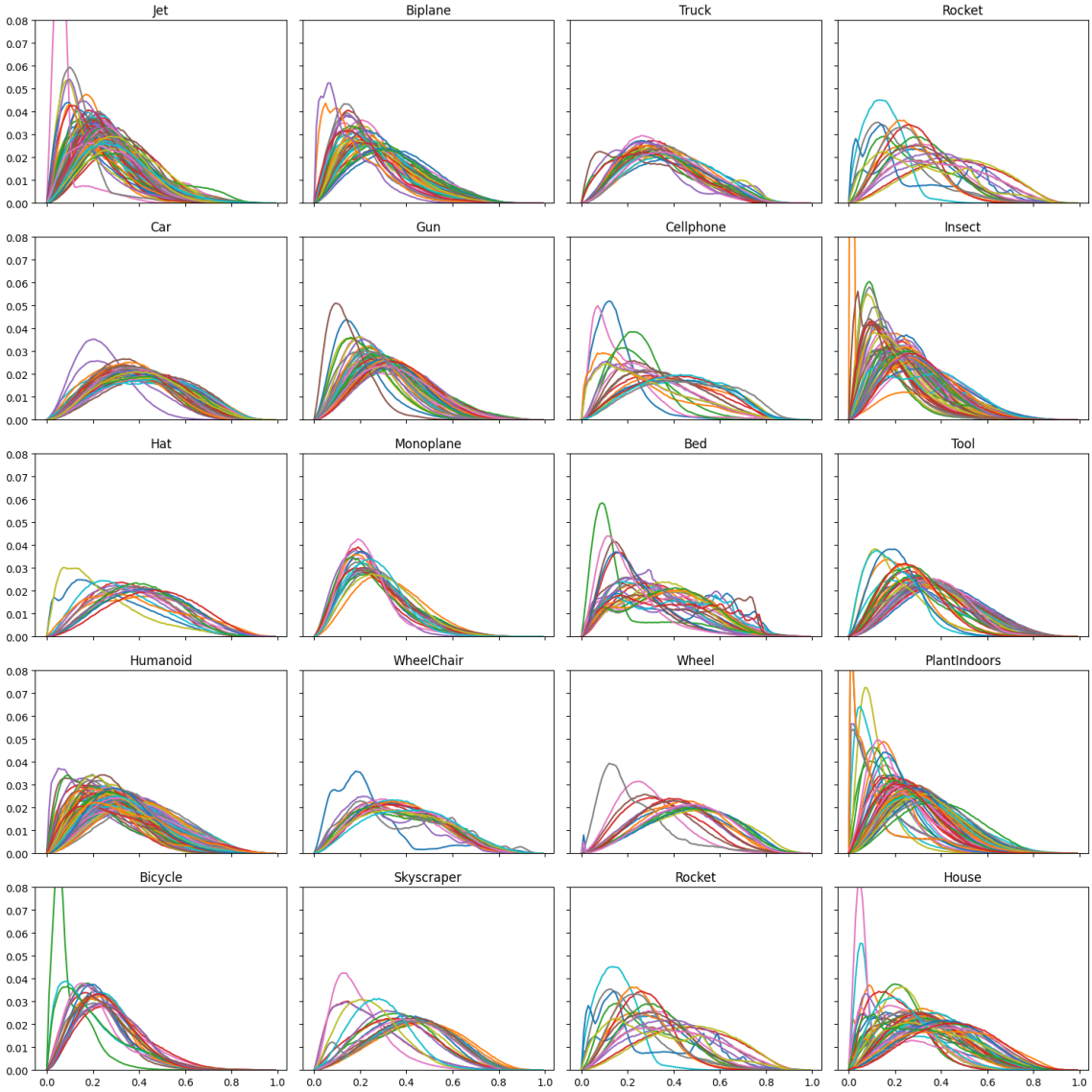



Fig. 46. D4 histogram of the objects within the classes

### A.6 Rectangularity

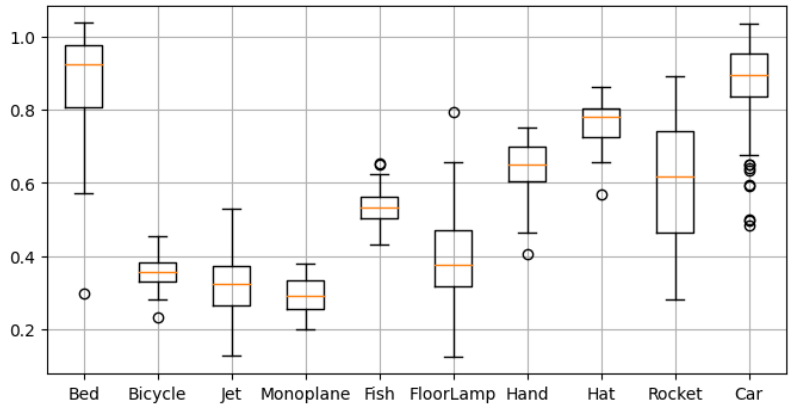



Fig. 47. Box-plot showing the distribution of the XY rectangularity per class, including outliers. For 10 classes from the database.

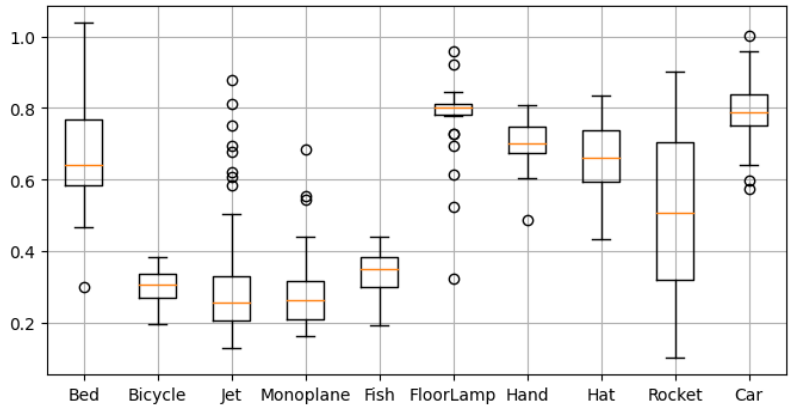



Fig. 48. Box-plot showing the distribution of the XZ rectangularity per class, including outliers. For 10 classes from the database.

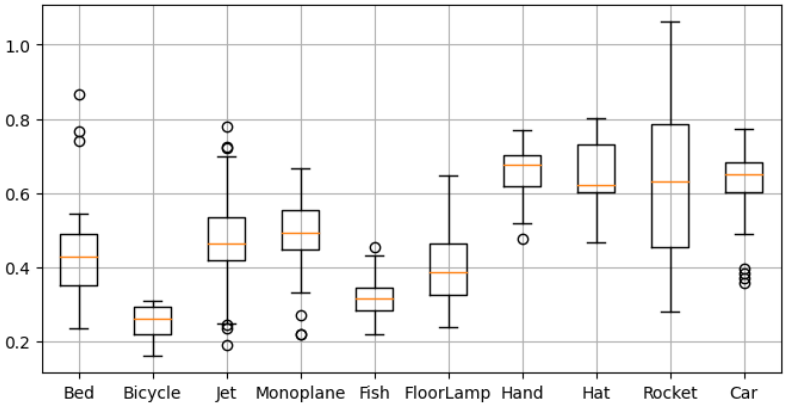



Fig. 49. Box-plot showing the distribution of the YZ rectangularity per class, including outliers. For 10 classes from the database.

### A.7 Circularity

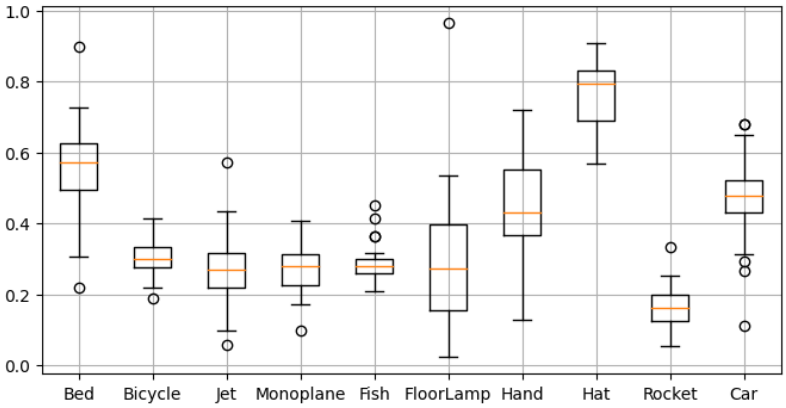



Fig. 50. Box-plot showing the distribution of the XY circularity per class, including outliers. For 10 classes from the database.

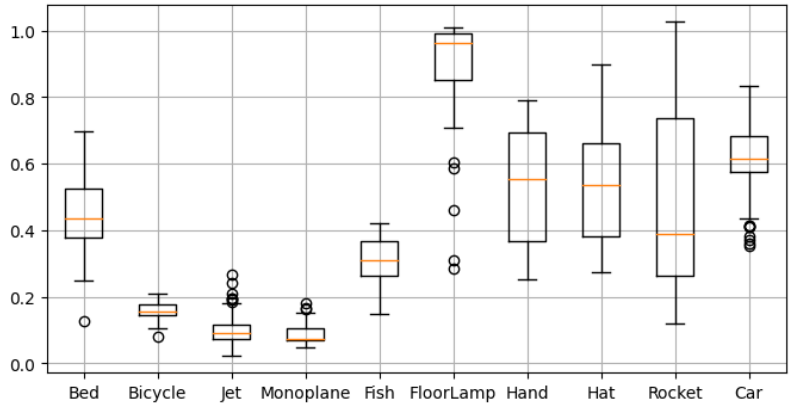



Fig. 51. Box-plot showing the distribution of the XZ circularity per class, including outliers. For 10 classes from the database.

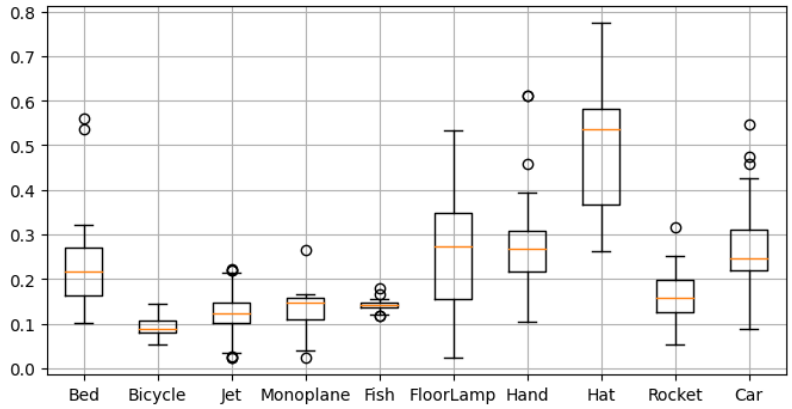



Fig. 52. Box-plot showing the distribution of the YZ circularity per class, including outliers. For 10 classes from the database.