Exercise 3: Siamese networks and zero-shot learning

Angus Munro

Introduction

A Siamese network was trained to encode plankton images into 64-dimensional vectors, i.e. as a vector space embedding. The training data comprised approximately 220000 monochromatic images from 27 classes (species). The validation and test data each comprised 2700 images (100 from each category). The code and model was based on the provided repository at https://github.com/ketil-malde/plankton-siamese (https://github.com/ketil-malde/plankton-siamese); this was not modified extensively, instead, the assignment focussed mainly on the interpretation and visualisation of the resulting encoded species "clusters".

Model and data

The model comprised an `Inception V3` base network, global average pooling, a dropout layer, followed by a dense output layer of 64 units with sigmoid activation.

The raw images were resized to 299x299 resolution, and were passed to the model, which pasted them onto a 3-channel 299x299 white template, with the monochromatic pixels constituting channel 0. Thus, the two other channels remained unused on the input side but were required due to the architecture of `Inception V3`.

Training the model

The model was trained using a triplet loss function, whereby three images (a random anchor, a random positive of the same class, and a random negative of a different class) were input separately to the base network, and the model trained towards a low loss, i.e. outputs similar for the anchor and positive, and different for the anchor and negative, as quantified by the 2-norm of output vector difference.

Curriculum learning was attempted, where the validation accuracy by class was used to alter the probabilities in the triplet generator toward difficult classes, but this was found to be unwieldy and of little benefit, so the approach was abandoned.

Transfer learning was used with `imagenet` weights. The default code's model was fully trainable and this was found to be suboptimal at the outset of training with the untrained new model head confounding the pretrained base model. Instead, the Inception base model was frozen for the first 5 epochs to pre-train the model head, after which the base model was unlocked (at a lower learning rate) to continue training the full model. Training was conducted stepwise in steps of

5 epochs; between each step, various information was logged and the learning rate was reduced by 10%. Additionally the model was saved at each step, enabling the resumption of training for model 'nursing'. An `Adam` optimiser was used, and a learning rate of 1E-3 for the head training and 1E-5 for the full training were found to be appropriate.

The standard triplet loss was observed to fall to around 0.2 after 10 cycles each of 5 epochs, where each epoch comprised 600 batches of 20 triplets (12000 triplets per epoch).

The script `train.py` performed the training, using parameters contained in `config.py`, and the output of the training epochs is reproduced below (though the training was not done in Jupyter Notebook):

Epoch 1/5 600/600 [==============================] - 639s 1s/step - loss: 1.2891 - val_loss: 0.9695

Epoch 2/5 600/600 [==============================] - 574s 956ms/step - loss: 1.0145 - val_loss: 0.8825

Epoch 3/5 600/600 [==============================] - 574s 956ms/step - loss: 0.9749 - val_loss: 0.8263

Epoch 4/5 600/600 [==============================] - 575s 958ms/step - loss: 0.8633 - val_loss: 0.7616

Epoch 5/5 600/600 [==============================] - 574s 956ms/step - loss: 0.8224 - val_loss: 0.6671

Epoch 1/5 600/600 [==============================] - 575s 958ms/step - loss: 0.7824 - val_loss: 0.7063

Epoch 2/5 600/600 [==============================] - 575s 959ms/step - loss: 0.6954 - val_loss: 0.6225

Epoch 3/5 600/600 [==============================] - 575s 959ms/step - loss: 0.7260 - val_loss: 0.5778

Epoch 4/5 600/600 [==============================] - 575s 958ms/step - loss: 0.6634 - val_loss: 0.5825

Epoch 5/5 600/600 [==============================] - 574s 957ms/step - loss: 0.6325 - val_loss: 0.4933

Epoch 1/5 600/600 [==============================] - 575s 958ms/step - loss: 0.6089 - val_loss: 0.5784

Epoch 2/5 600/600 [==============================] - 574s 957ms/step - loss: 0.5513 - val_loss: 0.4322

Epoch 3/5 600/600 [==============================] - 574s 957ms/step - loss: 0.5685 - val_loss: 0.5239

Epoch 4/5 600/600 [==============================] - 574s 956ms/step - loss: 0.5499 - val_loss: 0.4774

Epoch 5/5 600/600 [==============================] - 575s 958ms/step - loss: 0.5299 - val_loss: 0.4233

Epoch 1/5 600/600 [==============================] - 574s 957ms/step - loss: 0.5105 - val_loss: 0.3972

Epoch 2/5 600/600 [==============================] - 573s 955ms/step - loss: 0.4738 - val_loss: 0.3855

Epoch 3/5 600/600 [==============================] - 574s 957ms/step - loss: 0.4902 - val_loss: 0.4417

Epoch 4/5 600/600 [==============================] - 575s 958ms/step - loss: 0.4592 - val_loss: 0.4689

Epoch 5/5 600/600 [==============================] - 576s 960ms/step - loss: 0.4526 - val_loss: 0.5247

```
Epoch 1/5 600/600 [==============================] - 574s 957ms/step - loss: 0.4414 - val_loss: 0.3077
Epoch 2/5 600/600 [==============================] - 574s 957ms/step - loss: 0.4243 - val_loss: 0.3328
Epoch 3/5 600/600 [==============================] - 574s 957ms/step - loss: 0.3814 - val_loss: 0.4663
Epoch 4/5 600/600 [==============================] - 574s 957ms/step - loss: 0.4005 - val_loss: 0.4510
Epoch 5/5 600/600 [==============================] - 574s 957ms/step - loss: 0.4041 - val_loss: 0.3711
Epoch 1/5 600/600 [==============================] - 574s 957ms/step - loss: 0.3945 - val_loss: 0.3418
Epoch 2/5 600/600 [==============================] - 575s 959ms/step - loss: 0.3603 - val_loss: 0.3398
Epoch 3/5 600/600 [==============================] - 574s 957ms/step - loss: 0.3428 - val_loss: 0.3252
Epoch 4/5 600/600 [==============================] - 574s 957ms/step - loss: 0.3696 - val_loss: 0.3065
Epoch 5/5 600/600 [==============================] - 575s 958ms/step - loss: 0.3653 - val_loss: 0.3227
Epoch 1/5 600/600 [==============================] - 575s 959ms/step - loss: 0.3522 - val_loss: 0.3545
Epoch 2/5 600/600 [==============================] - 574s 957ms/step - loss: 0.3164 - val_loss: 0.3058
Epoch 3/5 600/600 [==============================] - 576s 960ms/step - loss: 0.3117 - val_loss: 0.3013
Epoch 4/5 600/600 [==============================] - 574s 957ms/step - loss: 0.3425 - val_loss: 0.2539
Epoch 5/5 600/600 [==============================] - 575s 958ms/step - loss: 0.3341 - val_loss: 0.2749
Epoch 1/5 600/600 [==============================] - 574s 956ms/step - loss: 0.3324 - val_loss: 0.3018
Epoch 2/5 600/600 [==============================] - 575s 958ms/step - loss: 0.3168 - val_loss: 0.3463
Epoch 3/5 600/600 [==============================] - 575s 958ms/step - loss: 0.3015 - val_loss: 0.2841
Epoch 4/5 600/600 [==============================] - 575s 959ms/step - loss: 0.3117 - val_loss: 0.3029
Epoch 5/5 600/600 [==============================] - 575s 958ms/step - loss: 0.3140 - val_loss: 0.2510
Epoch 1/5 600/600 [==============================] - 575s 958ms/step - loss: 0.3078 - val_loss: 0.2591
```

```
Epoch 2/5 600/600 [==============================] - 574s 956ms/step - loss: 0.2894 - val_loss: 0.2666
Epoch 3/5 600/600 [==============================] - 574s 957ms/step - loss: 0.2734 - val_loss: 0.2700
Epoch 4/5 600/600 [==============================] - 576s 959ms/step - loss: 0.2937 - val_loss: 0.3135
Epoch 5/5 600/600 [==============================] - 574s 957ms/step - loss: 0.2777 - val_loss: 0.2941
Epoch 1/5 600/600 [==============================] - 575s 959ms/step - loss: 0.2790 - val_loss: 0.2878
Epoch 2/5 600/600 [==============================] - 575s 959ms/step - loss: 0.2745 - val_loss: 0.2990
Epoch 3/5 600/600 [==============================] - 574s 957ms/step - loss: 0.2797 - val_loss: 0.2620
Epoch 4/5 600/600 [==============================] - 574s 957ms/step - loss: 0.2695 - val_loss: 0.2488
Epoch 5/5 600/600 [==============================] - 576s 960ms/step - loss: 0.2426 - val_loss: 0.2037
Epoch 1/5 600/600 [==============================] - 779s 1s/step - loss: 0.2478 - val_loss: 0.1980
Epoch 2/5 600/600 [==============================] - 589s 982ms/step - loss: 0.2450 - val_loss: 0.2522
Epoch 3/5 600/600 [==============================] - 574s 957ms/step - loss: 0.2356 - val_loss: 0.2577
Epoch 4/5 600/600 [==============================] - 574s 957ms/step - loss: 0.2456 - val_loss: 0.2477
Epoch 5/5 600/600 [==============================] - 578s 963ms/step - loss: 0.2415 - val_loss: 0.2736
Epoch 1/5 600/600 [==============================] - 575s 958ms/step - loss: 0.2433 - val_loss: 0.2036
Epoch 2/5 600/600 [==============================] - 574s 957ms/step - loss: 0.2288 - val_loss: 0.2292
Epoch 3/5 600/600 [==============================] - 575s 958ms/step - loss: 0.2159 - val_loss: 0.2521
Epoch 4/5 600/600 [==============================] - 576s 959ms/step - loss: 0.2317 - val_loss: 0.2343
Epoch 5/5 600/600 [==============================] - 574s 957ms/step - loss: 0.2388 - val_loss: 0.1916
Epoch 1/5 600/600 [==============================] - 624s 1s/step - loss: 0.2263 - val_loss: 0.2118
Epoch 2/5 600/600 [==============================] - 558s 930ms/step - loss: 0.2193 - val_loss: 0.2108
```

```
Epoch 3/5 600/600 [==============================] - 557s 928ms/step - loss: 0.2266 - val_loss: 0.2231
Epoch 4/5 600/600 [==============================] - 557s 928ms/step - loss: 0.2170 - val_loss: 0.2135
Epoch 5/5 600/600 [==============================] - 556s 927ms/step - loss: 0.2312 - val_loss: 0.2145
Epoch 1/5 600/600 [==============================] - 558s 930ms/step - loss: 0.2035 - val_loss: 0.2030
Epoch 2/5 600/600 [==============================] - 557s 928ms/step - loss: 0.2083 - val_loss: 0.2300
Epoch 3/5 600/600 [==============================] - 557s 928ms/step - loss: 0.2348 - val_loss: 0.1749
Epoch 4/5 600/600 [==============================] - 557s 928ms/step - loss: 0.2146 - val_loss: 0.1976
Epoch 5/5 600/600 [==============================] - 557s 928ms/step - loss: 0.2085 - val_loss: 0.1693
Epoch 1/5 600/600 [==============================] - 558s 930ms/step - loss: 0.2239 - val_loss: 0.1396
Epoch 2/5 600/600 [==============================] - 557s 928ms/step - loss: 0.2206 - val_loss: 0.1950
Epoch 3/5 600/600 [==============================] - 558s 930ms/step - loss: 0.2151 - val_loss: 0.2062
Epoch 4/5 600/600 [==============================] - 557s 929ms/step - loss: 0.2013 - val_loss: 0.2059
Epoch 5/5 600/600 [==============================] - 561s 935ms/step - loss: 0.2067 - val_loss: 0.1923
Epoch 1/5 600/600 [==============================] - 557s 929ms/step - loss: 0.1940 - val_loss: 0.2288
Epoch 2/5 600/600 [==============================] - 558s 929ms/step - loss: 0.2086 - val_loss: 0.2193
Epoch 3/5 600/600 [==============================] - 557s 929ms/step - loss: 0.2094 - val_loss: 0.2042
Epoch 4/5 600/600 [==============================] - 557s 929ms/step - loss: 0.2100 - val_loss: 0.2729
Epoch 5/5 600/600 [==============================] - 558s 930ms/step - loss: 0.2148 - val_loss: 0.2038
Epoch 1/5 600/600 [==============================] - 557s 929ms/step - loss: 0.1864 - val_loss: 0.2114
Epoch 2/5 600/600 [==============================] - 558s 929ms/step - loss: 0.2131 - val_loss: 0.2111
Epoch 3/5 600/600 [==============================] - 557s 929ms/step - loss: 0.2028 - val_loss: 0.2116
```

Epoch 4/5 600/600 [==============================] - 557s 928ms/step - loss: 0.1929 - val_loss: 0.2130

Epoch 5/5 600/600 [==============================] - 558s 930ms/step - loss: 0.1964 - val_loss: 0.1801

Epoch 1/5 600/600 [==============================] - 558s 929ms/step - loss: 0.2144 - val_loss: 0.2266

Epoch 2/5 600/600 [==============================] - 558s 930ms/step - loss: 0.1977 - val_loss: 0.2184

Epoch 3/5 600/600 [==============================] - 557s 928ms/step - loss: 0.2096 - val_loss: 0.1743

Epoch 4/5 600/600 [==============================] - 557s 929ms/step - loss: 0.2089 - val_loss: 0.1960

Epoch 5/5 600/600 [==============================] - 558s 930ms/step - loss: 0.2101 - val_loss: 0.1778

Epoch 1/5 600/600 [==============================] - 558s 930ms/step - loss: 0.2122 - val_loss: 0.2009

Epoch 2/5 600/600 [==============================] - 557s 929ms/step - loss: 0.2034 - val_loss: 0.1952

Epoch 3/5 600/600 [==============================] - 558s 929ms/step - loss: 0.1994 - val_loss: 0.1882

Epoch 4/5 600/600 [==============================] - 557s 929ms/step - loss: 0.2030 - val_loss: 0.2423

Epoch 5/5 600/600 [==============================] - 558s 930ms/step - loss: 0.2082 - val_loss: 0.2097

A plot of the progression of training and validation loss is shown below:

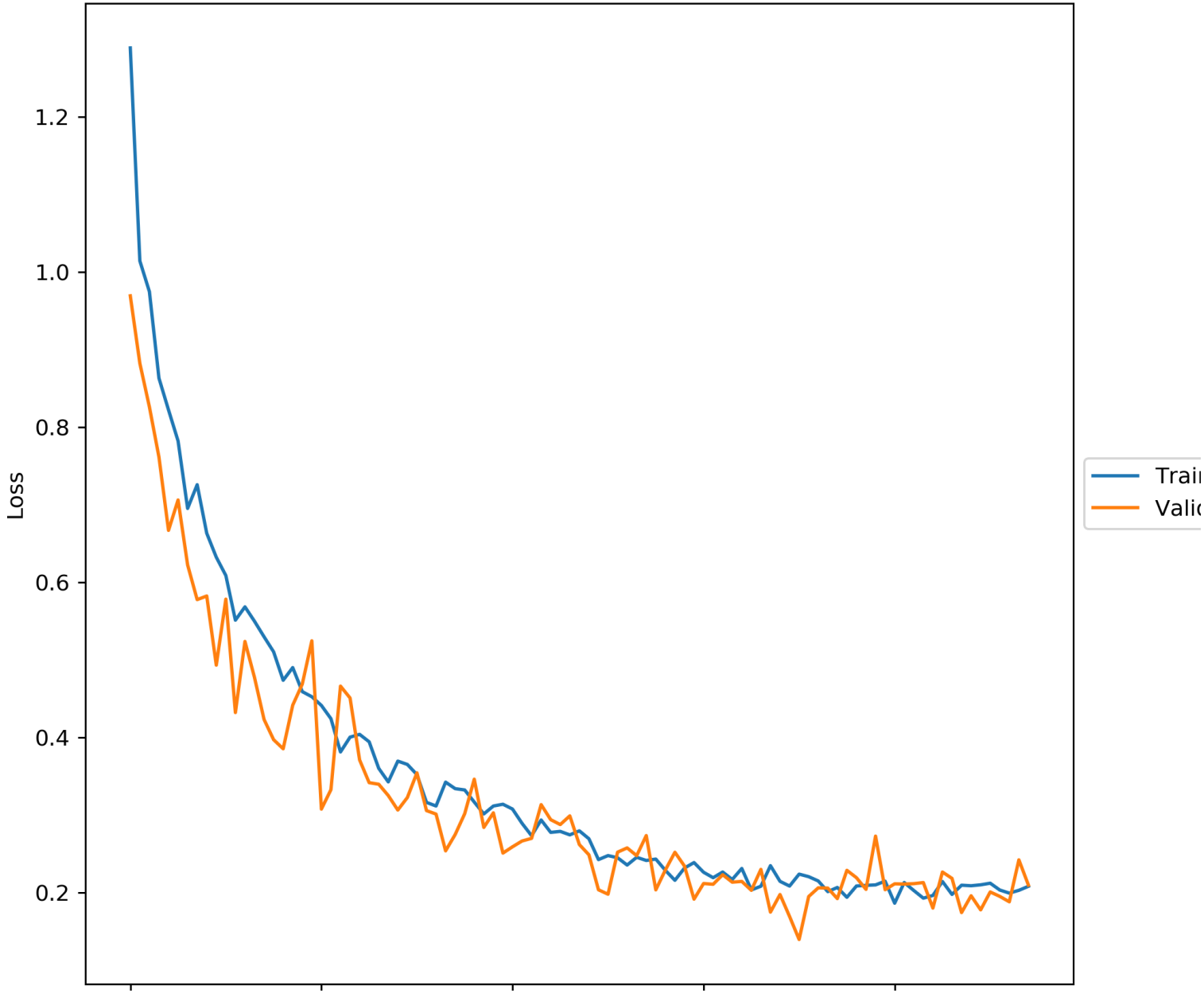
In [1]:
```
%pwd
```

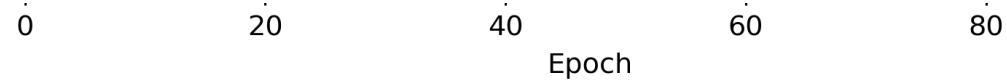
Out[1]:
```
'/home/angus_munro/projects/ex3/INF368/INF368-Ex3/notebooks'
```

In [3]:
```
%cd ..
%matplotlib notebook
from plotly.offline import download_plotlyjs, init_notebook_mode
init_notebook_mode(connected=True)
import viz as v
import config as C
import testing as T
```

```
### Testing triplet_generator ###
0 a: (4, 299, 299, 3) p: (4, 299, 299, 3) n: (4, 299, 299, 3) y: (4, 2)
1 a: (4, 299, 299, 3) p: (4, 299, 299, 3) n: (4, 299, 299, 3) y: (4, 2)
2 a: (4, 299, 299, 3) p: (4, 299, 299, 3) n: (4, 299, 299, 3) y: (4, 2)
3 a: (4, 299, 299, 3) p: (4, 299, 299, 3) n: (4, 299, 299, 3) y: (4, 2)
```

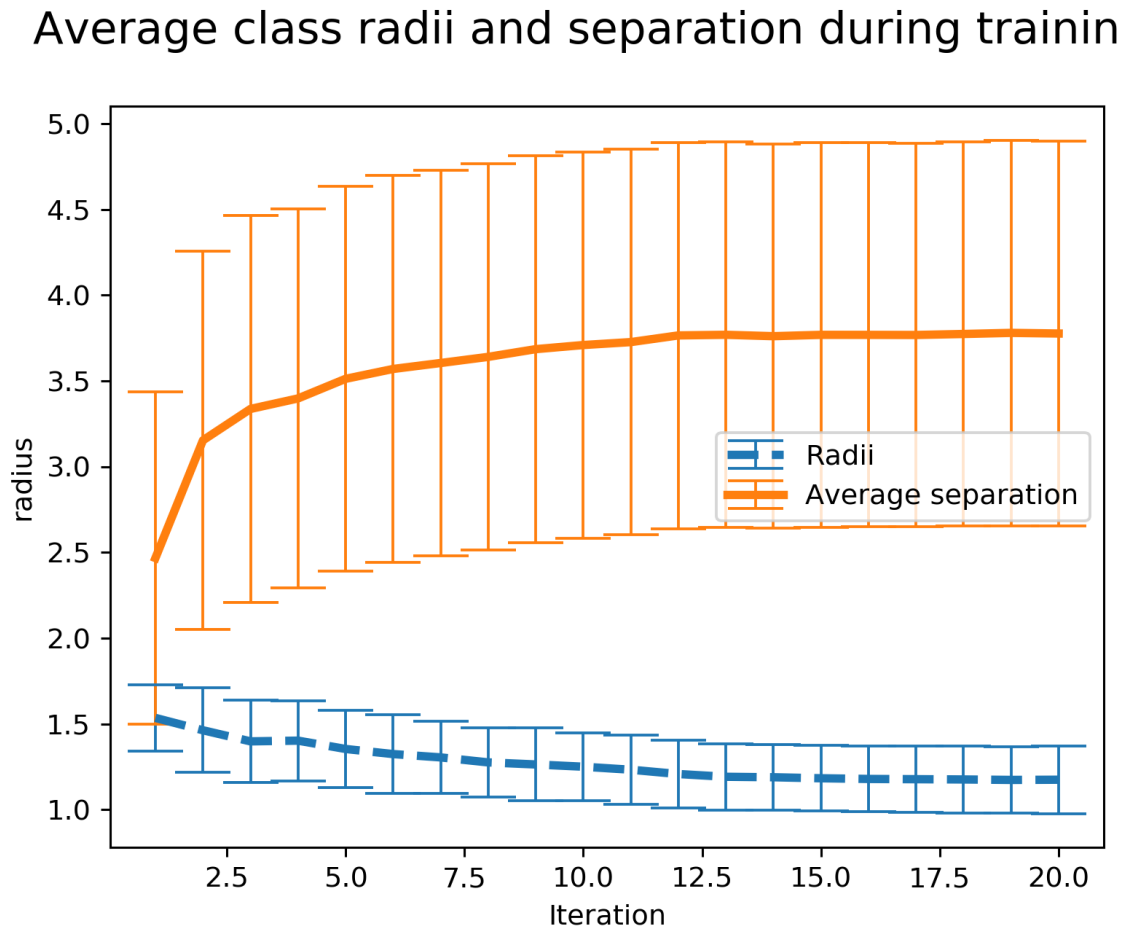
In [4]:
```
_ = v.plot_epoch_losses()
```


1.2
1.0
0.8
0.6
0.4
0.2
Loss
Trai
Vali



A plot of the average cluster radius compared to the average cluster separation is shown below. (observations...)

In [5]:
```
import config as C
_ = v.plot_train_radii_separation(C.obj_dir)
```



Visualize the embedding

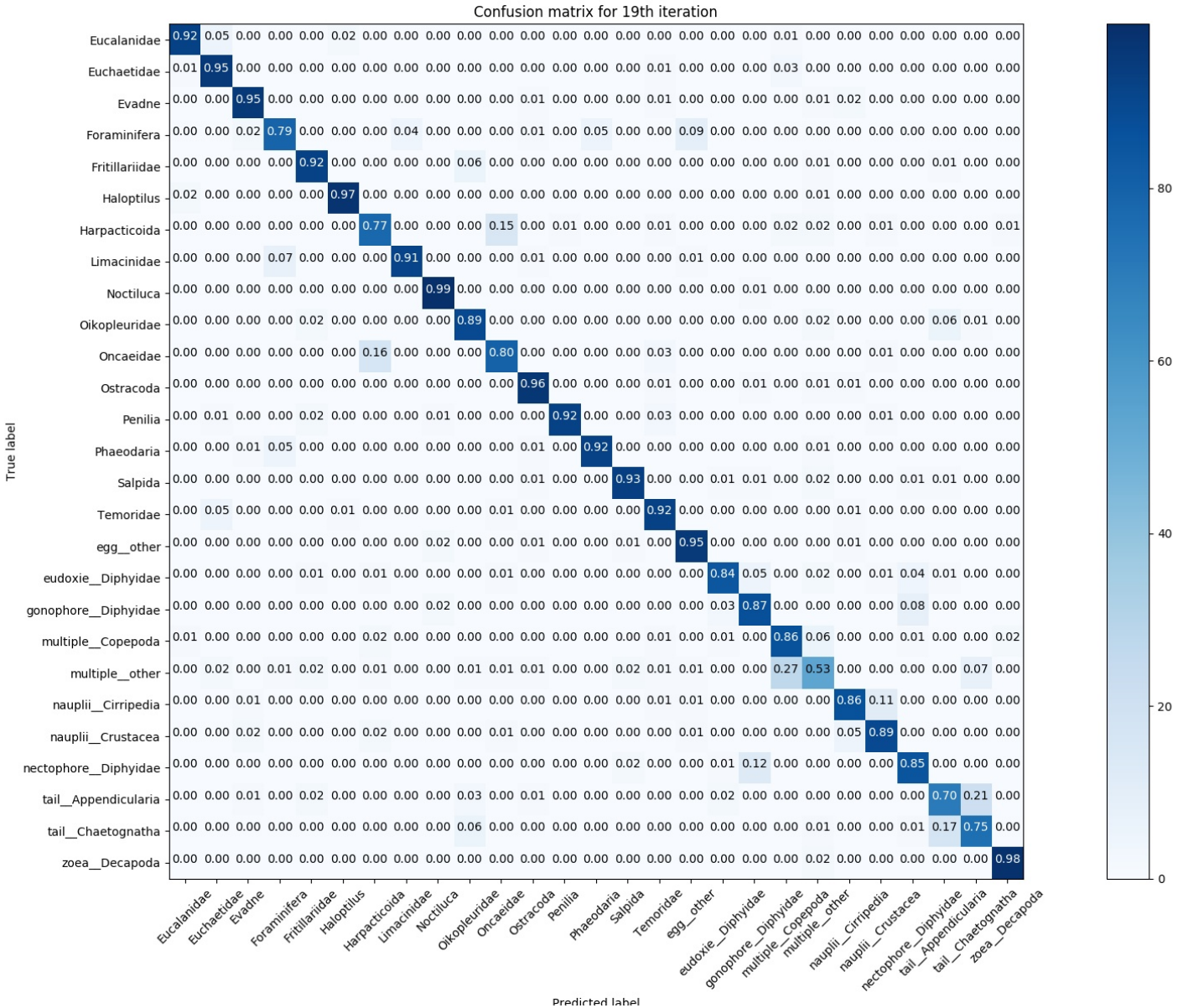
The trained embedder was investigated using several means. Firstly, a confusion plot demonstrates the class for which it performs well, and those which it confuses. Secondly, PCA was applied to elucidate the components which explain the most variance in the embedding, and the embedded validation data clusters were visualised using `ploly` with a 3D plot of the leading 3 dimensions, including an animation over the course of training.

The remaining code snippets rely on reading training history and log data from the logs directory (`config.log_dir`), and loading precomputed validation embedding vectors (in pickle files) from the obj directory (`config.log_dir`). Also from the overall training log file, `train111.log` in the main folder. The training run which was shown above, leading to the working model, was called `111` , so for example, the config log_dir is `logs/111` . The best model is from training iteration 19, so that `logs/111/val_pred_19.pkl` is the pickled file of 'best model' vector embeddings for the validation data.

Below is a confusion plot of the model's classifications, where the classification for each embedding is simply the closest class centroid:

In [6]:
```
from matplotlib import animation, rc
from IPython.display import HTML
```

In [ ]:
```
clusters = v.read_clusters()
i = 19 #which model to use
cmat, classes = v.confusion_matrix(clusters, i-1) #i-1 is index into conf vector
fig = v.plot_confusion_matrix(cmat,
                              classes,
                              title='Confusion matrix for {}th iteration'.format(i),
                              cmap=None,
                              normalize=True,
                                figsize=(20,12))
# the inline figure was not well-readable, so a jpg was saved and imported instead, see below.
fig.savefig('notebooks/images/confusion2.jpeg')
```


Confusion matrix for 19th iteration
True label
Predicted label
Eucalanidae
Euchaetidae
Evadne
Foraminifera
Fritillariidae
Haloptilus
Harpacticoida
Limacinidae
Noctiluca
Oikopleuridae
Oncaeidae
Ostracoda
Penilia
Phaeodaria
Salpida
Temoridae
egg__other
eudoxie__Diphyidae
gonophore__Diphyidae
multiple__Copepoda
multiple__other
nauplii__Cirripedia
nauplii__Crustacea
nectophore__Diphyidae
tail__Appendicularia
tail__Chaetognatha
zoea__Decapoda
0
20
40
60
80

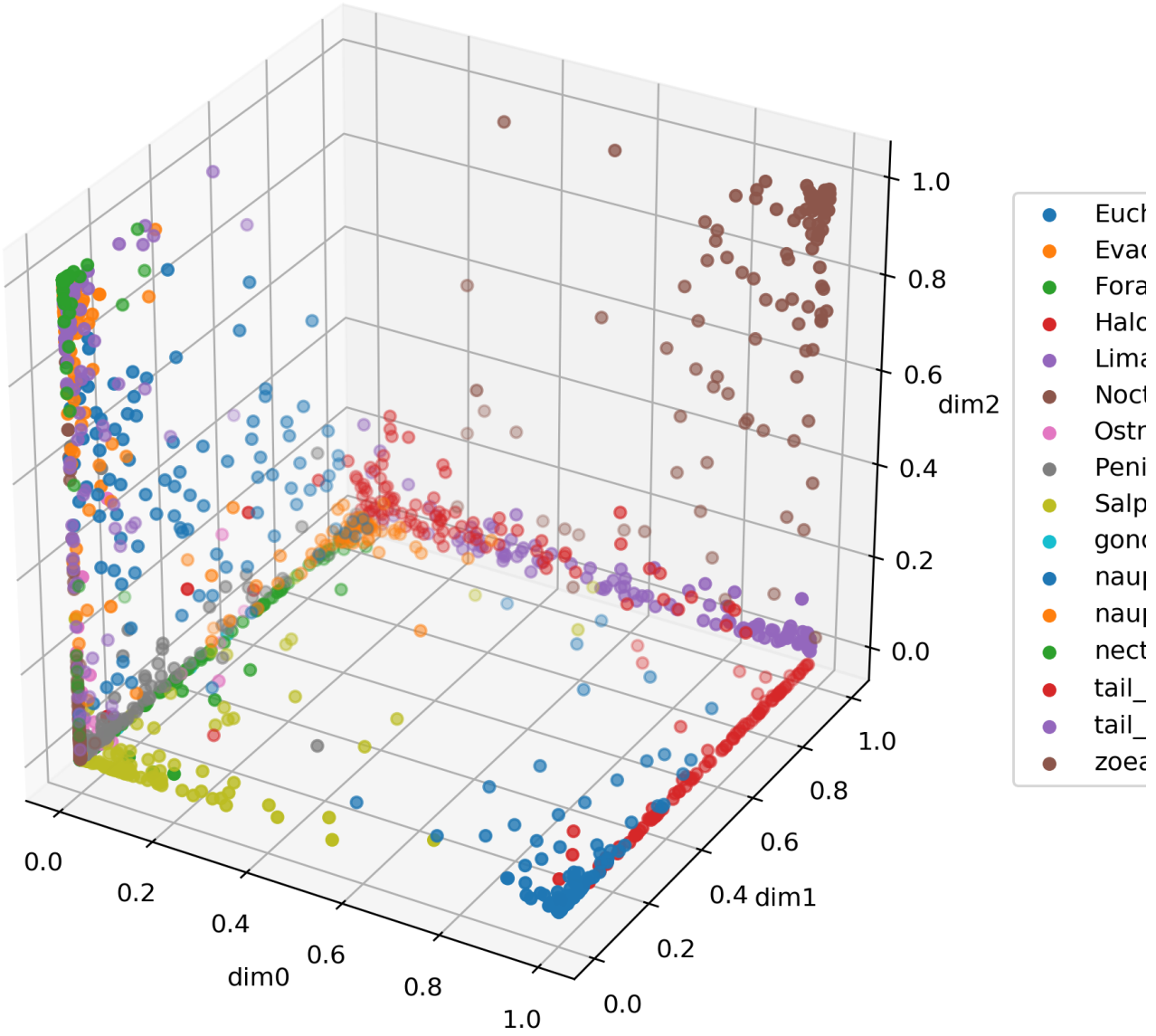
A video animating the confusion matrix over the course of training can also be created:

In [ ]:
```
clust = v.read_clusters()
v.animate_and_save_confusion(clust, "images/conf_anim.mp4")
```

We can observe the final validation data vector embedding (at least, 3 dimensions of it) using the function `plot_class_vectors_scatter`. The `dict_squeeze` (and `dict_unsqueeze`) simply convert between the format of the stored sets of vector embeddings. The sets of vectors are stored in dictionaries indexed by class name. Squeezing produces 2-D numpy arrays of dimension (number_vectors x 64), whereas unsqueezing produces lists of 1-D array (length 64) vectors. The latter matches the format used by `testing.get_vectors`, for example.

In [9]:
```
import testing as T
vs = v.dict_squeeze(T.load_obj('obj/111/val_pred_19'))
```

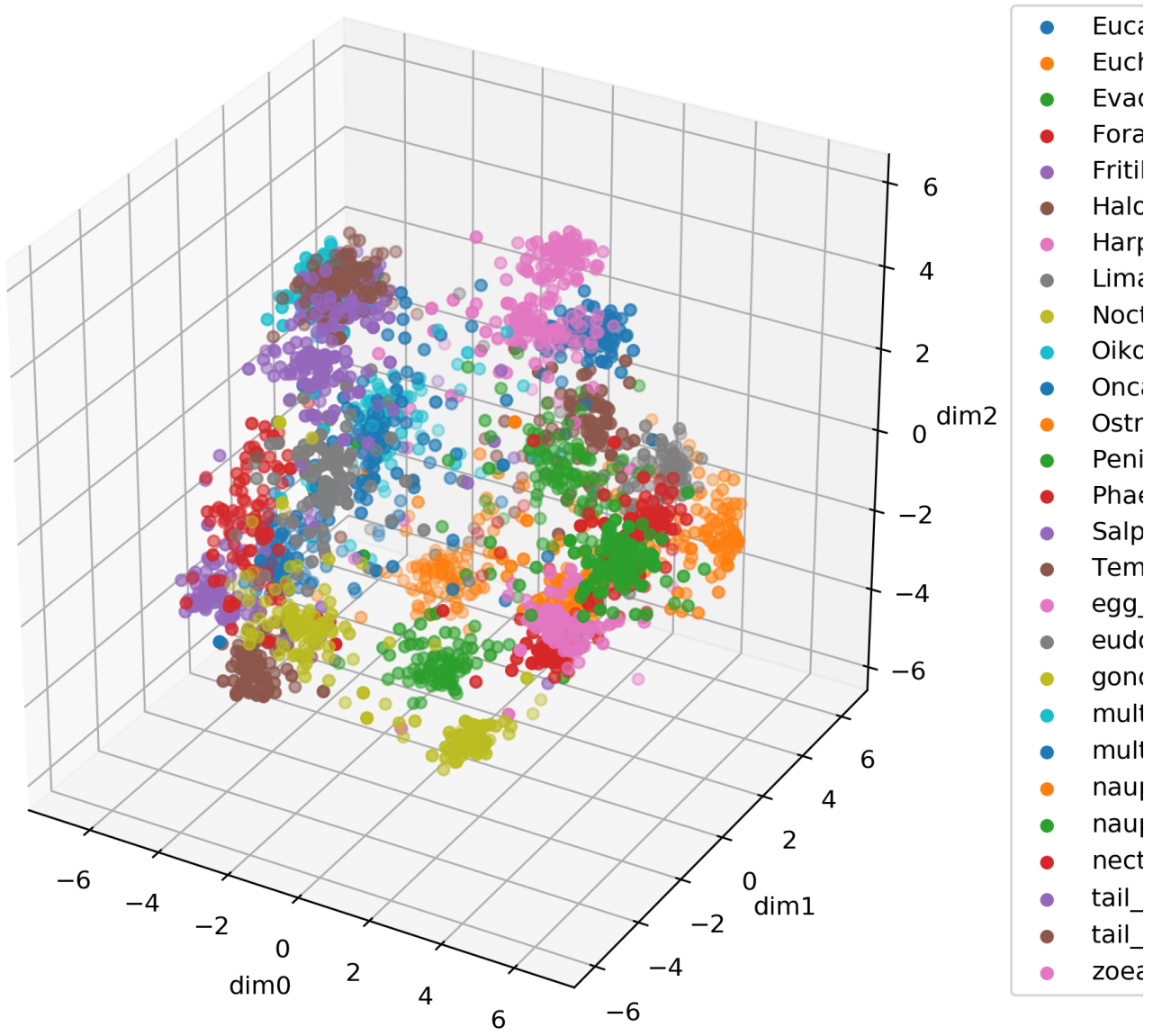
In [10]:
```
#classes = ['Harpacticoida', 'Phaeodaria', 'Evadne', 'Haloptilus', 'Oncaeidae', 'Oikopleuridae']
#_ = v.plot_class_vectors_scatter(vs, classes,
#                                 dims=[0,1,2])
classes = [c for c in vs]
_ = v.plot_class_vectors_scatter(vs, [classes[i] for i in range(16)],
                                 dims=[0,1,2])
```


dim0
dim1
dim2
0.0
0.2
0.4
0.6
0.8
1.0
Euch
Evad
Fora
Halo
Lima
Noct
Ostr
Peni
Salp
gono
naup
naup
nect
tail_
tail_
zoea

It can be seen immediately that the predicted vectors mostly are saturating along the edges of a cube in the output space. This likely reflects saturation of the sigmoid activation in the dense layer at the top of the network. A possible improvement (not done here due to time constraints) is to alter this, perhaps by using a linear activation.

We can also plot the data along the principal axes of variation as calculated using SVD:

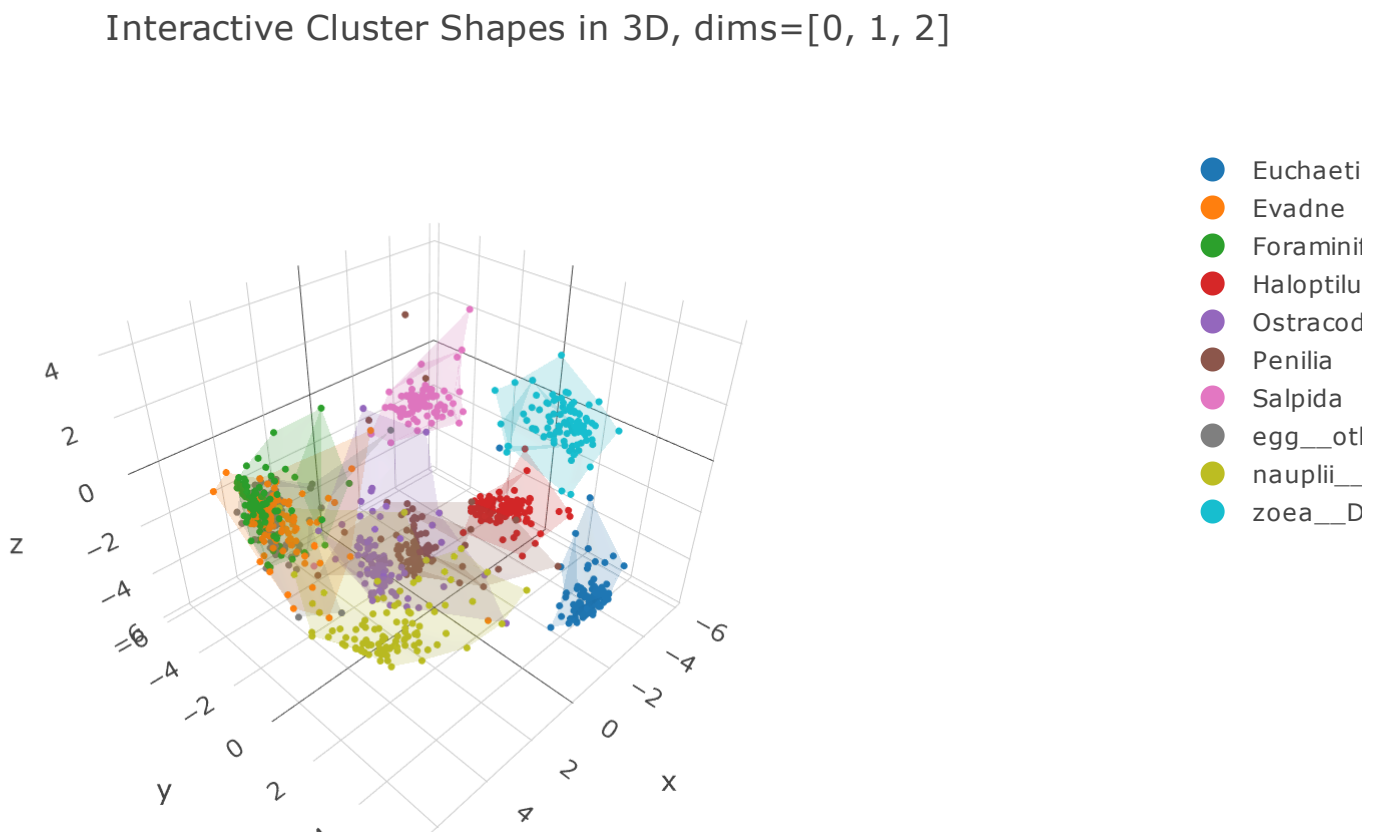
In [11]:
```
ws_n = v.svd_project(vs)
_ = v.plot_class_vectors_scatter(ws_n, classes)
#_ = v.plot_class_vectors_scatter(ws_n, [classes[i] for i in range(16)])
```


dim0
dim1
dim2
Euca
Euch
Evad
Fora
Fritil
Halo
Harp
Lima
Noct
Oiko
Onca
Ostr
Peni
Phae
Salp
Tem
egg_
eudo
gono
mult
mult
naup
naup
nect
tail_
tail_
zoea

Moreover, we can also use plotly to make 3d scatter and mesh plots of the clusters, which are rather pleasing to the eye, with the plotly package.

The plotly routines may also be run in non-notebook mode, to create an html file (here called 'temp-plot.html') containing the same interactive plot.

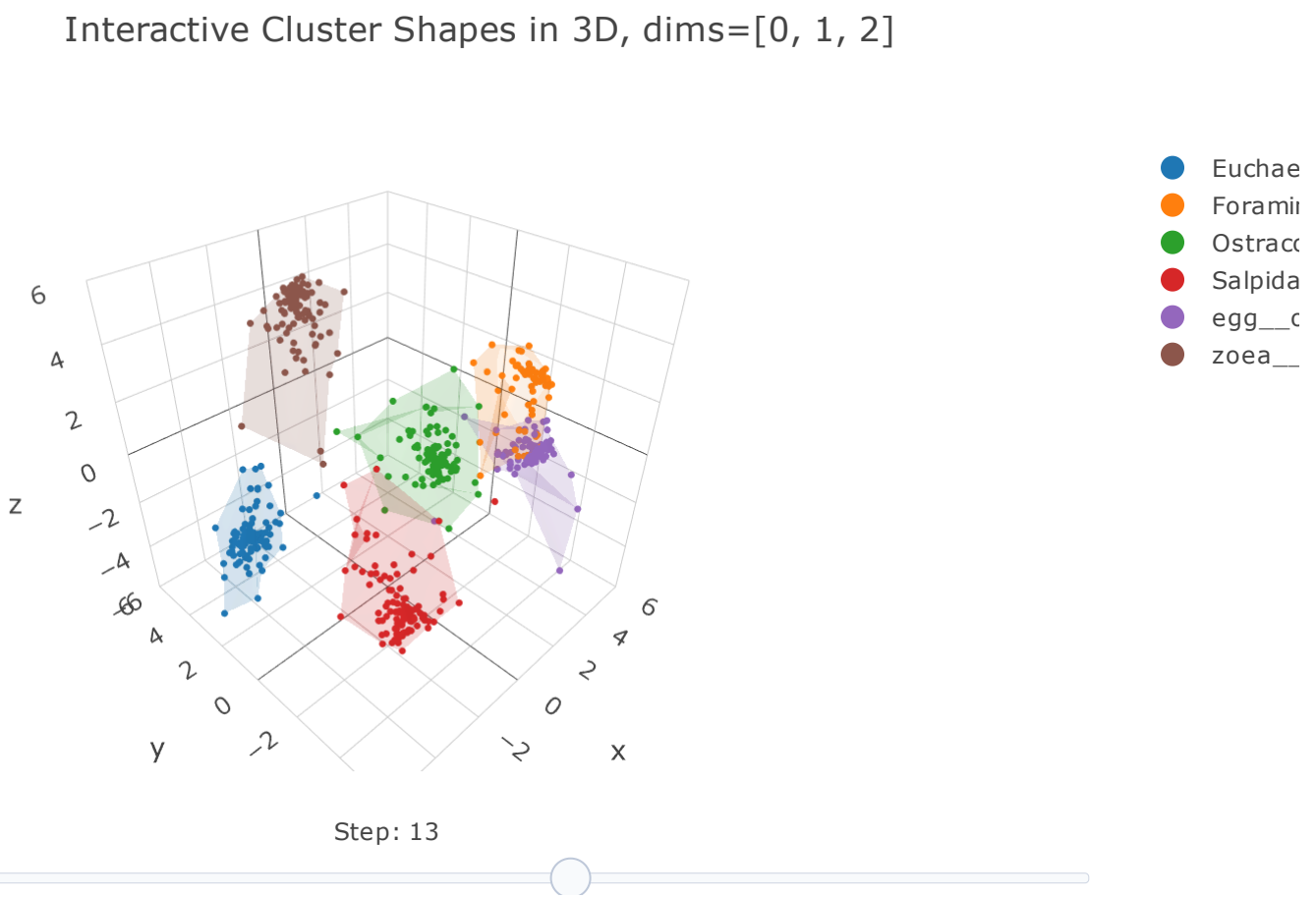
In [12]:
```
classes = list(ws_n)
v.plot_class_vectors_plotly(ws_n, [classes[i] for i in range(10)], notebook=True)
```



An animation routine was created, showing the clusters' principal components (with respect to the data at the final step), through the training process:

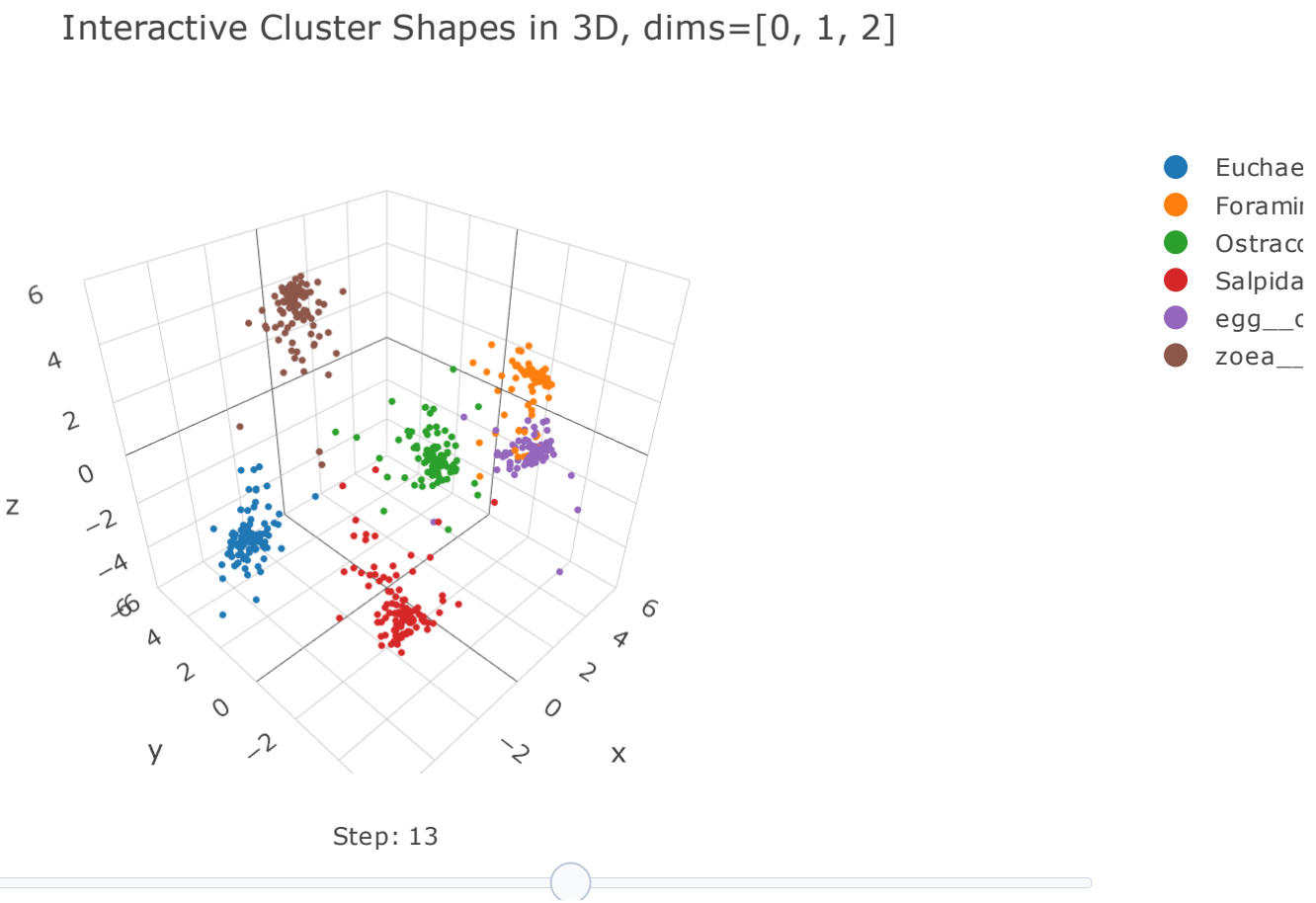
In [13]:
```
import config as C
vh = v.read_validation_history(C.obj_dir)
wh = {i: v.svd_project(vh[i]) for i in vh}
classes = list(wh[1])[:6]
```

In [14]:
```
v.plotly_animate(wh, classes=classes, ax_lims=6, notebook=True)
```



A faster animation can be got by not plotting the mesh:

In [15]:
```
v.plotly_animate(wh, classes=classes, ax_lims=6, mesh=False, notebook=True)
```

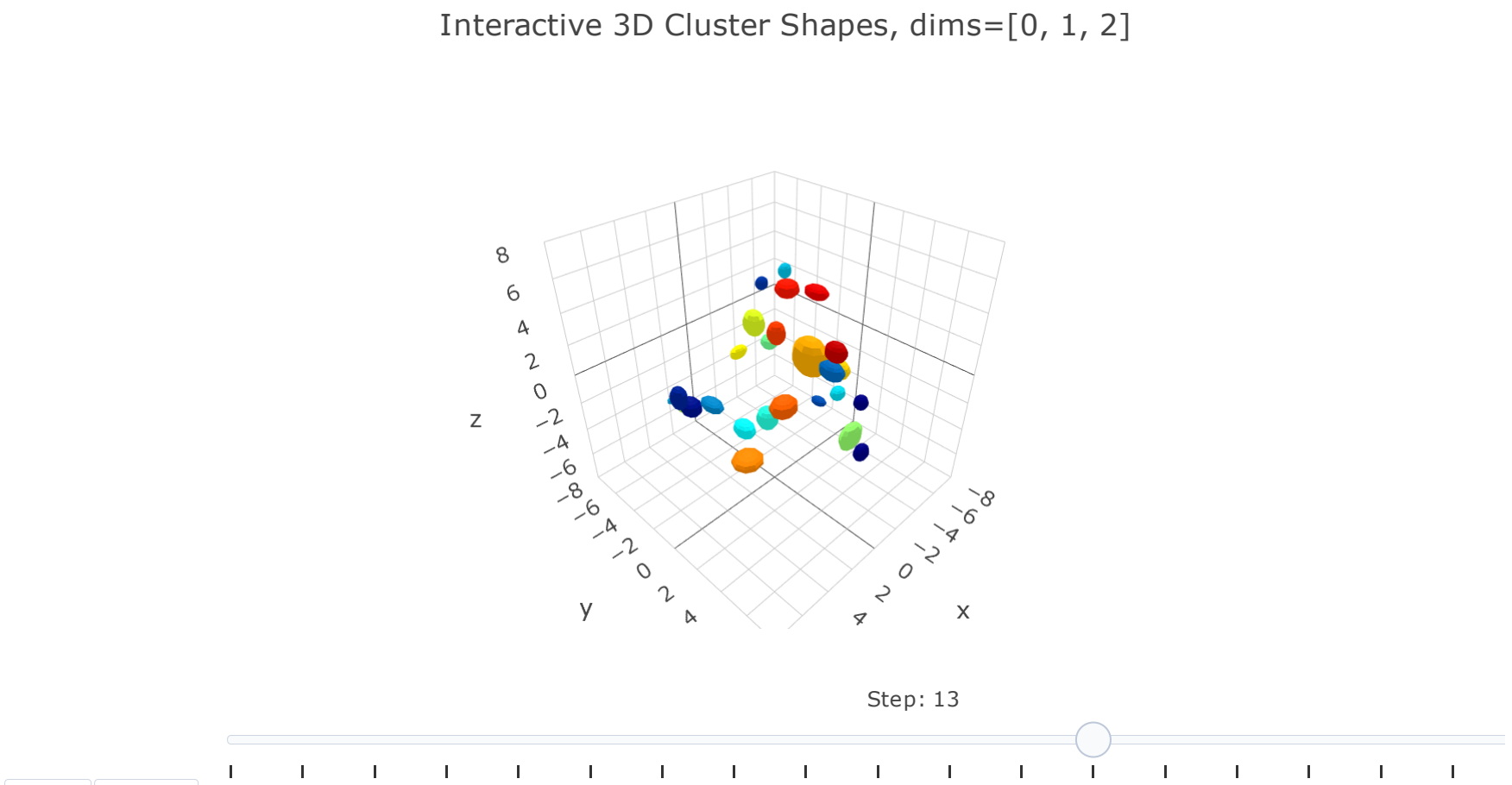


Plotly also supports smoothly animated transitions between animation frames, but only for 2D scatter plots, which were unfortunately not very sufficient for visualising this data. All plotly plots shown in this notebook do not support smooth transitions, so the frames jump from one to the next.

As this routine was quite slow, an alternative would be to show the cluster locations and radii in these coordinates as spheres. This was implemented in plotly, but it was necessary to plot the spheres as surfaces from scratch (using a spherical coordinate mesh), which did not lead to any speed increase. Hopefully the plotly package will be improved over time because it seems to be quite promising. The spheres were easily modified to be ovoids with radius in each dimension being the cluster spread in that dimension, to give a more accurate visualisation of the distribution of data spread in each dimension:

In [16]:
```
classes = list(wh[1]) #all classes
classes.sort()
```

In [17]:
```
v.plotly_spheres(wh, classes, ax_lims=8., notebook=True, n=10)
```



Design a suitable classification scheme

Once we have the encoded data, we may train an SVM or similar on the validation data, and test it on the test data. Apologies for the long code snippet below:

In [18]:
```
#Train alternative classifiers on the validation data, test on test data.
#prepare data:
from sklearn.svm import SVC, LinearSVC
from sklearn import tree
from sklearn import cluster
from sklearn.preprocessing import LabelBinarizer, LabelEncoder
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix as conf_mat
from viz import *
import config as C
import testing as T

vs = vh[19]
classes = list(vs)
classes.sort()
X_val, y_val = collect_class_vs(vs)
lb = LabelBinarizer()
le = LabelEncoder()
lb.fit(classes)
le.fit(classes)
y_val_enc = le.transform(y_val)
Y_val = lb.transform(y_val)


##load test vectors / generate test vectors using CNN:
cnn_name = 'epoch_19.model'
oname = cnn_name.split('.')[0]+'_test_pred'
ofile = os.path.join(C.obj_dir,oname)
#T.save_model_predictions(os.path.join(C.model_dir,cnn_name), tdir=C.test_dir,
#                         ofile=ofile)
test = T.load_obj(ofile)
X_test, y_test = collect_class_vs(test)
y_test_enc = le.transform(y_test)

#Decision tree
model_tree = tree.DecisionTreeClassifier()
model_tree.fit(X_val, Y_val)
Y_test = lb.transform(y_test)
Y_test_pred = model_tree.predict(X_test)
train_acc = accuracy_score(Y_val, model_tree.predict(X_val))
test_acc = accuracy_score(Y_test, Y_test_pred)
```

```
print('Train / Test accuracy decision tree      = {:.2%} / {:.2%}'.format(train_acc,test_acc))

#SVM
model_svm = SVC(kernel='linear')
model_svm.fit(X_val, y_val_enc)
y_test_pred = model_svm.predict(X_test)
train_acc = accuracy_score(y_val_enc, model_svm.predict(X_val))
test_acc = accuracy_score(y_test_enc, y_test_pred)
print('Train / Test accuracy linear SVC        = {:.2%} / {:.2%}'.format(train_acc, test_acc))

#SVM with RBF
model_svm_rbf = SVC(kernel='rbf', C=1E-7, probability=True)
model_svm_rbf.fit(X_val, y_val_enc)
y_test_pred = model_svm_rbf.predict(X_test)
train_acc = accuracy_score(y_val_enc, model_svm_rbf.predict(X_val))
test_acc = accuracy_score(y_test_enc, y_test_pred)
print('Train / Test accuracy RBF SVC           = {:.2%} / {:.2%}'.format(train_acc, test_acc))

#K-means clustering (unsupervised)
n_clusters = len(list(vs))
centersSKL = cluster.MiniBatchKMeans(n_clusters)
centersSKL.fit(X_val)
y_pred = centersSKL.predict(X_val)
#identify the clusters by class (hopefully)
centroids = {} #for the predicted clusters
radii = {}
#find centroids, radii of detected clusters
for k in set(y_pred):
    k_vecs = [X_val[i, :] for i in range(len(y_pred)) if y_pred[i]==k]
    centroids[k] = T.centroid(k_vecs)
    radii[k] = T.radius(centroids[k], k_vecs)
#find centroids, radii of the true classes
centroids_actual = {}
radii_actual = {}
for k in set(y_val):
    c_vecs = [X_val[i, :] for i in range(len(y_val)) if y_val[i]==k]
    centroids_actual[k] = T.centroid(c_vecs)
    radii_actual[k] = T.radius(centroids_actual[k], c_vecs)
cent_list = [centroids_actual[c] for c in classes]
rad_list = [radii_actual[c] for c in classes]
#match each detected cluster to closest class
#by minimising L2 norm of (centroid; radius) difference
```

```
def cluster_metric(c1, c2, r1, r2):
    return T.dist(c1, c2) + T.dist(r1,r2)
def find_nearest_cluster(centroid, radius, centroid_list, radius_list):
    (c1, r1) = (centroid, radius)
    return np.argmin([cluster_metric(c1,centroid_list[i],r1,radius_list[i]) for i in range(len(radius_list))])
k_map = {}
for k in centroids: #cluster number
    best_i = find_nearest_cluster(centroids[k], radii[k], cent_list, rad_list)
    k_map[k] = classes[best_i]
c_map = {c:k for k,c in k_map.items()}
#training score:
y_pred_cls = [k_map[k] for k in y_pred] #classes
y_pred_enc = le.transform(y_pred_cls) #numbers
train_acc = accuracy_score(y_val_enc, y_pred_enc)
#test score:
y_pred_cls = [k_map[k] for k in centersSKL.predict(X_test)]
y_test_pred = le.transform(y_pred_cls) #numbers
X_test, y_test = collect_class_vs(test) #get fresh y_test vectors
y_test_enc = le.transform(y_test)
test_acc = accuracy_score(y_test_enc, y_test_pred)
print('Train / Test accuracy K-means            = {:.2%} / {:.2%}'.format(train_acc, test_acc))

#SVM with RBF on SVD-transformed data
ws, items = svd_project(vs, return_items=True)
Z_val, y_val = collect_class_vs(ws)
y_val_enc = le.transform(y_val)
model_svm_svd = SVC(kernel='rbf', C=1E-7, probability=True)
model_svm_svd.fit(Z_val, y_val_enc)
#convert test data, with the  normalisation and transformation from the training data
X_mean = items['X_mean']
X_std = items['X_std']
test_n = {}
for c in test: #for all classes: normalise and transform
    test_n[c] = (np.squeeze(test[c]) - X_mean) / X_std
    test_n[c] = test_n[c] @ items['U']
Z_test, y_test = collect_class_vs(test_n)
y_test_enc = le.transform(y_test)
y_test_pred = model_svm_svd.predict(Z_test)
train_acc = accuracy_score(y_val_enc, model_svm_svd.predict(Z_val))
test_acc = accuracy_score(y_test_enc, y_test_pred)
print('Train / Test accuracy  RBF SVC with SVD  = {:.2%} / {:.2%}'.format(train_acc, test_acc))
```

```
Train / Test accuracy decision tree      = 100.00% / 79.30%
Train / Test accuracy linear SVC        = 91.67% / 86.96%
Train / Test accuracy RBF SVC            = 86.63% / 87.37%
Train / Test accuracy K-means            = 82.44% / 83.85%
Train / Test accuracy  RBF SVC with SVD  = 87.48% / 87.63%
```

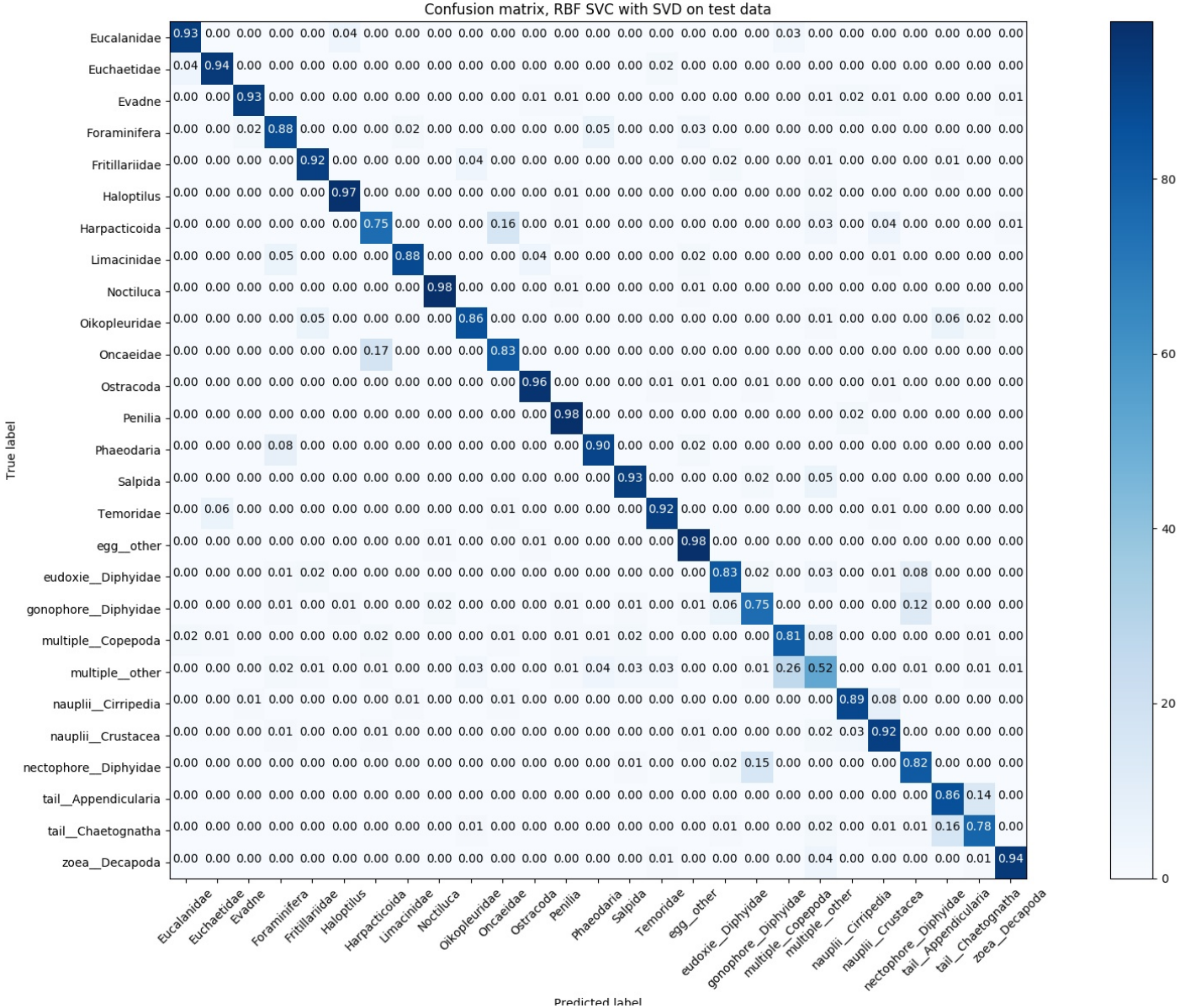
The methodology for the unsupervised K-means classifier needs a little explanation. The clustering was done on the embedded validation vectors in an unsupervised manner. Then, each cluster was assigned the class name which minimised a metric called `cluster_metric`. `Cluster_metric` was the sum of the Euclidean norm of the cluster centroid separation, and the Euclidean norm of the difference in cluster radii (ie, the absolute value difference). This metric was applied pairwise to find the best ground truth class for each detected cluster. It was noticed that one class was not matched by this method. Due to time constraints, the same was not done for the SVD-transformed data, but it is expected that the accuracy might have been better with the use of SVD.

The most accurate final classifier is the SVC using RBF, applied to the SVD-transformed data, with an accuracy of nearly 88%. This makes sense, because after transformation, the vector clusters better resemble spheroid point clouds, which should be eminently suitable for classifying with RBFs.

Confounded classes with the best classifier

We can construct a confusion plot for the best classifier to garner information on which classes confused the classifier the most:

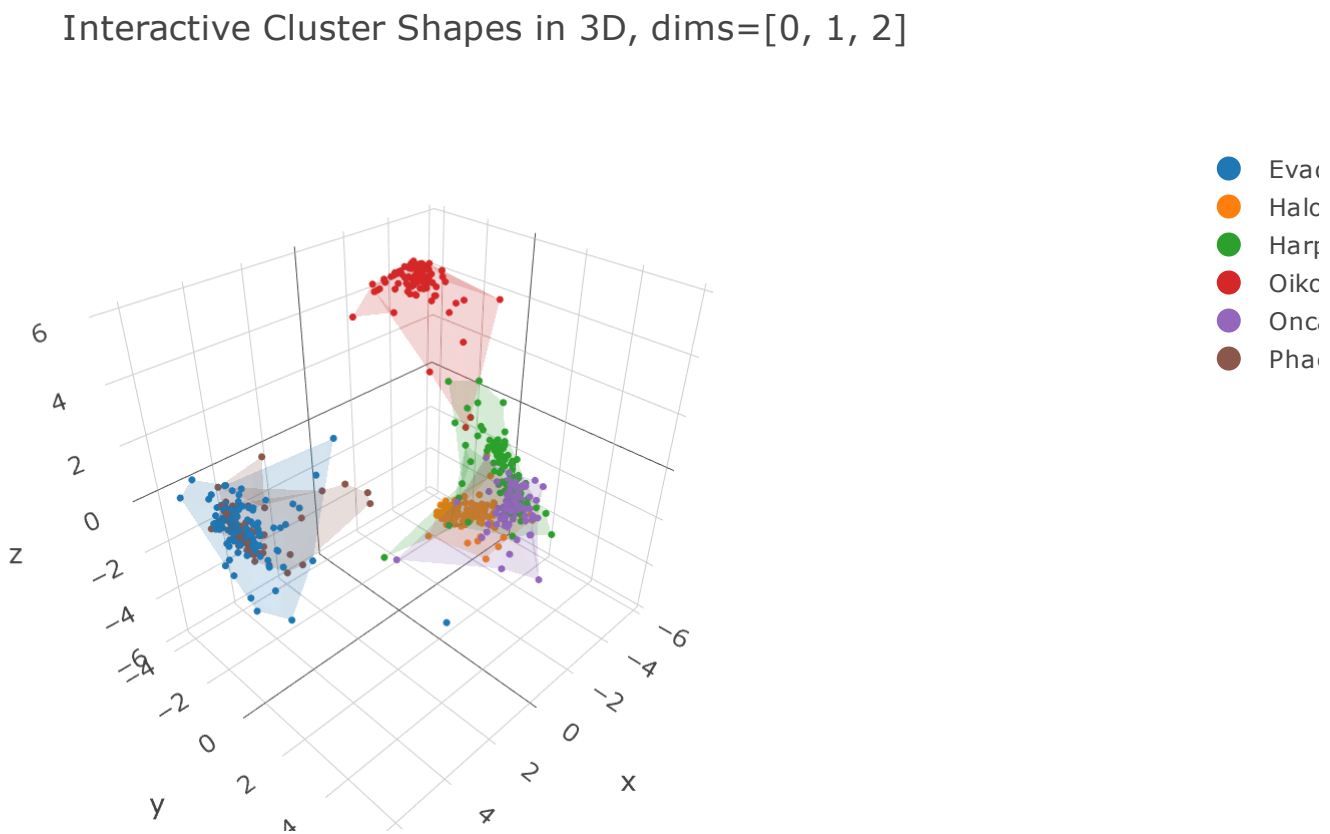
In [ ]:
```
from sklearn.metrics import confusion_matrix as conf_mat
%matplotlib notebook
cm = conf_mat(y_test_enc, y_test_pred)
classes = le.inverse_transform(range(len(set(y_test_enc))))
fig = plot_confusion_matrix(cm,
                            classes,
                            title='Confusion matrix, RBF SVC with SVD on test data',
                            cmap=None,
                            normalize=True,
                            figsize=(20,12))
fig.savefig('notebooks/images/best_confusion.jpg')
```


Confusion matrix, RBF SVC with SVD on test data
True label
Predicted label
Eucalanidae
Euchaetidae
Evadne
Foraminifera
Fritillariidae
Haloptilus
Harpacticoida
Limacinidae
Noctiluca
Oikopleuridae
Oncaeidae
Ostracoda
Penilia
Phaeodaria
Salpida
Temoridae
egg__other
eudoxie__Diphyidae
gonophore__Diphyidae
multiple__Copepoda
multiple__other
nauplii__Cirripedia
nauplii__Crustacea
nectophore__Diphyidae
tail__Appendicularia
tail__Chaetognatha
zoea__Decapoda
0
20
40
60
80

It can be seen that the worse confoundment is from Harpacticoida being predicted as Phaeodaria, Evadne as Haloptilus, Haloptilus as Evadne and Oncaeidae as Oikopleuridae.

Let's capture these classes only in a 3d scatterplot of the SVD leading components. Firstly for the validation data:

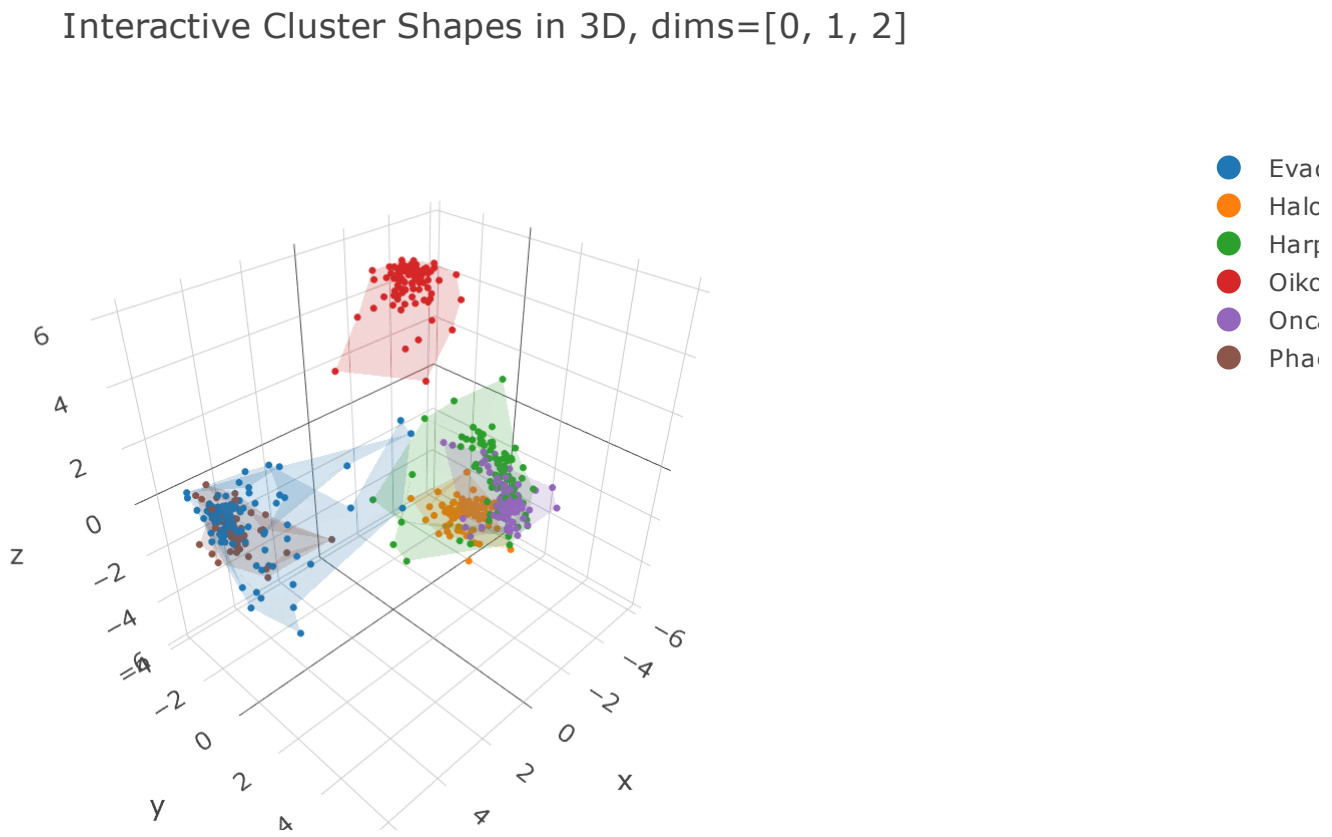
In [20]:
```
conf_classes = ['Harpacticoida', 'Phaeodaria', 'Evadne', 'Haloptilus', 'Oncaeidae', 'Oikopleuridae']
ws_n = v.svd_project(vs)
v.plot_class_vectors_plotly(ws_n, conf_classes, notebook=True, dims=[0,1,2])
```



Some overlap is seen between Evadne and Phaedoria. Also between Oncaeidae and Harpacticoida.

Let's look now at the test data. We have the normalised and singular-value transformed test data in dictionary test_n, from previously.

In [21]:
```
#we have test_n from before, which is dictionary of normalised and svd-transformed test vectors
v.plot_class_vectors_plotly(test_n, conf_classes, notebook=True)
```

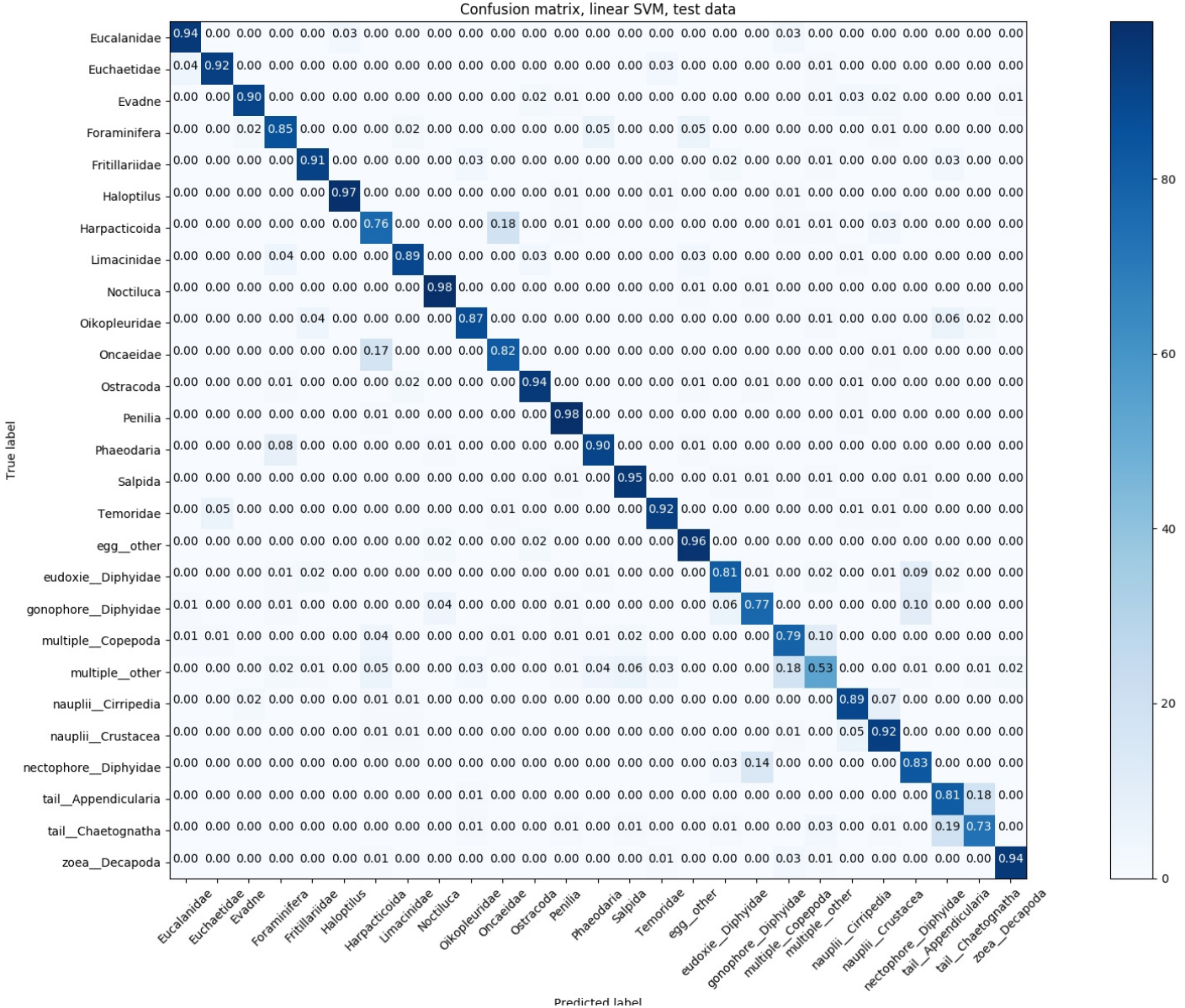


This shows overlap between Phaedoria and Evadne, as well as Harpacticoida and Oncaeidae - the same overlaps as observed for the validation data.

After checking the code, I have no explanation as to why the overlaps seen in both validation and test data appear different from the expectation from the confusion matrix. As the classifier was constructed in SVD space, the plots above should give insight into the class confusions, but they just cause more confusion.

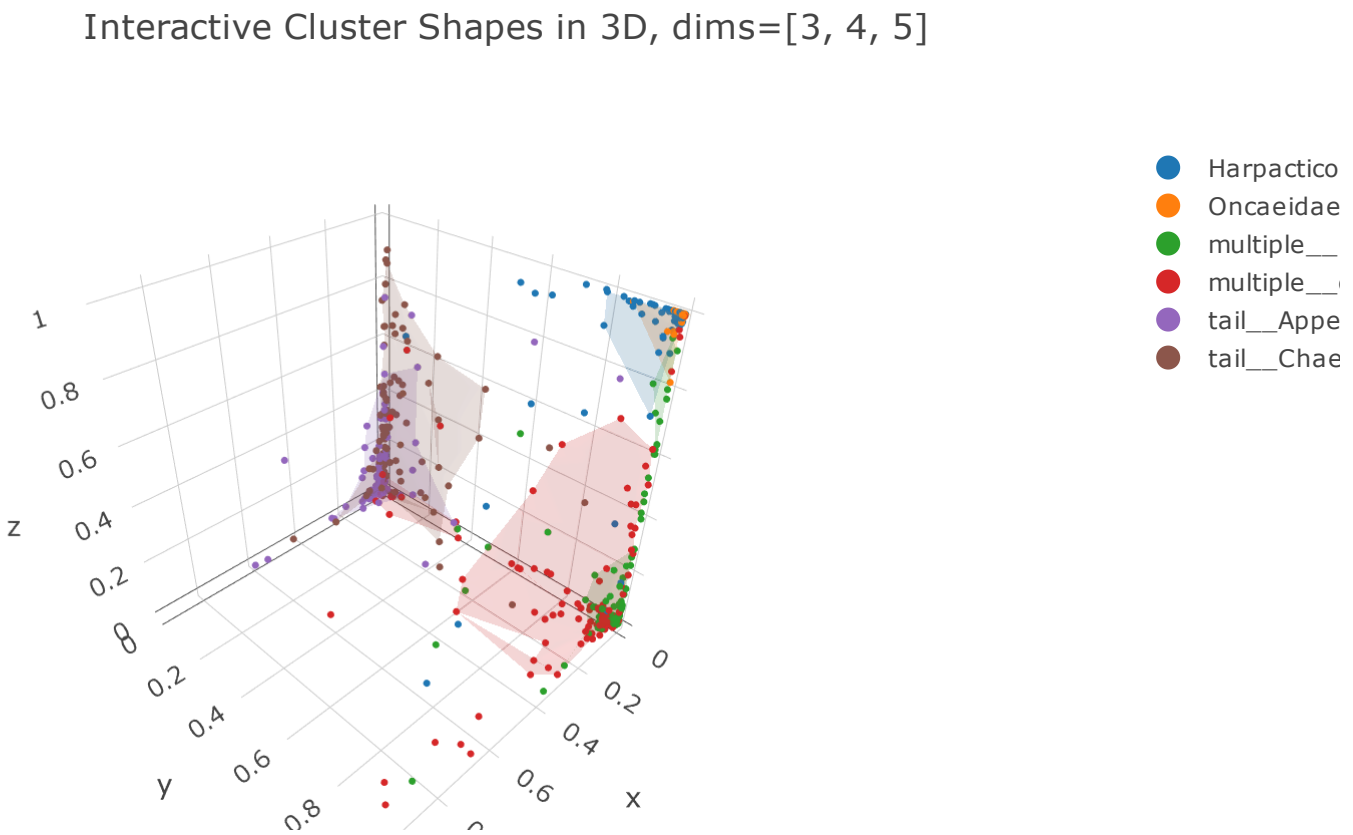
As a last resort, the same can be done for the confusion matrix for a simple classifier: the linear SVM. As SVD was not used in the training of this classifier, we will start by inspecting the class vectors in regular space.

In [ ]:
```
y_test_pred = model_svm.predict(X_test)
cm = conf_mat(y_test_enc, y_test_pred)
classes = le.inverse_transform(range(len(set(y_test_enc))))
fig = plot_confusion_matrix(cm,
                            classes,
                            title='Confusion matrix, linear SVM, test data',
                            cmap=None,
                            normalize=True,
                            figsize=(20,12))
fig.savefig('notebooks/images/svm_linear_confusion.jpg')
```

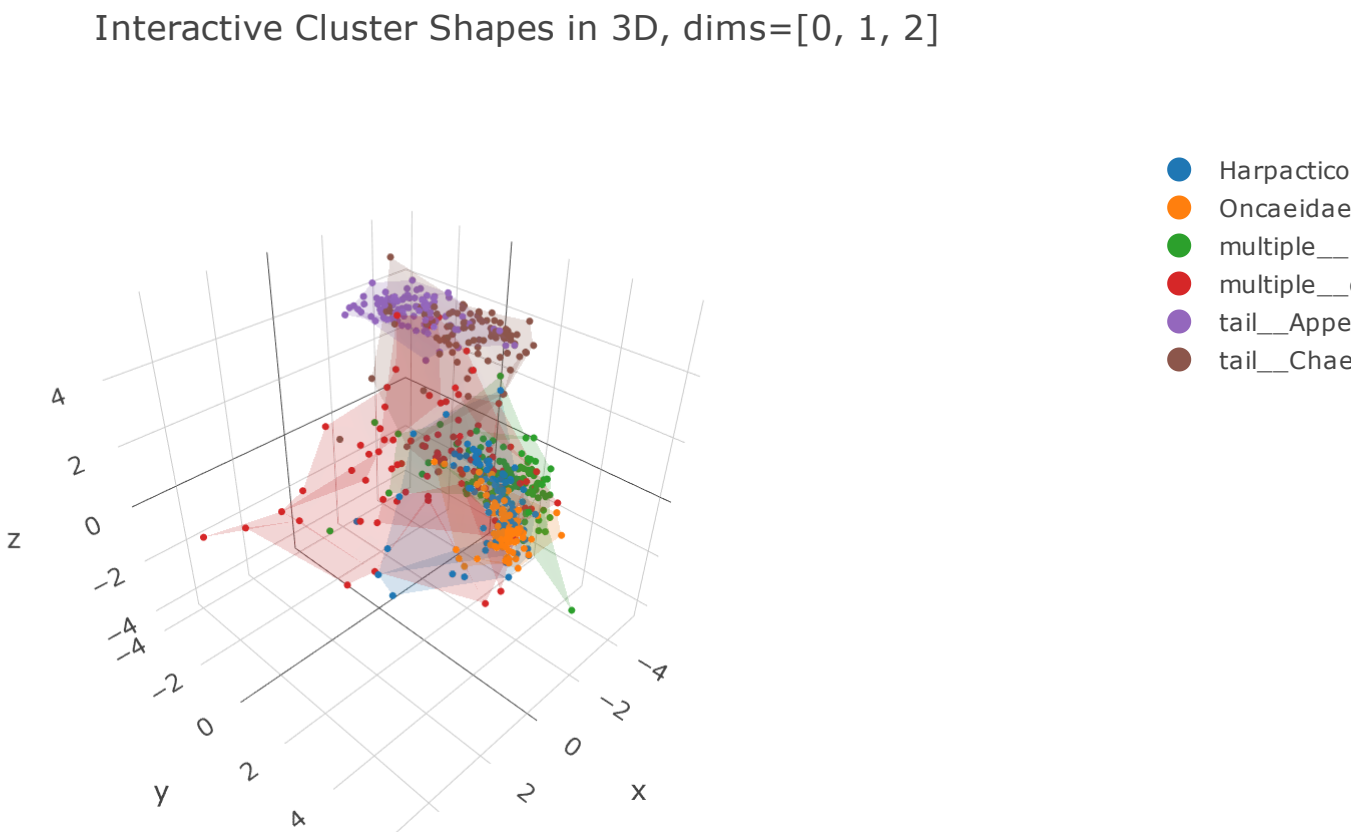

Confusion matrix, linear SVM, test data
True label
Predicted label
Eucalanidae
Euchaetidae
Evadne
Foraminifera
Fritillariidae
Haloptilus
Harpacticoida
Limacinidae
Noctiluca
Oikopleuridae
Oncaeidae
Ostracoda
Penilia
Phaeodaria
Salpida
Temoridae
egg__other
eudoxie__Diphyidae
gonophore__Diphyidae
multiple__Copepoda
multiple__other
nauplii__Cirripedia
nauplii__Crustacea
nectophore__Diphyidae
tail__Appendicularia
tail__Chaetognatha
zoea__Decapoda
0
20
40
60
80

The most confused classes here are tail**Appendicularia, tail**Chaetognatha, as well as Harpacticoida, Oncaeidae, as well as multiple**Copepoda and multiple**other. Let's inspect these classes geometrically:

In [25]:
```
conf_classes = ['tail__Appendicularia', 'tail__Chaetognatha', 'Harpacticoida', 'Oncaeidae',
                'multiple__Copepoda', 'multiple__other']
#plot in regular space
v.plot_class_vectors_plotly(vs, conf_classes, notebook=True, dims=[3,4,5])
```



In [26]:
```
#plot in svd space
v.plot_class_vectors_plotly(test_n, conf_classes, notebook=True)
```



Now it can be seen that the observed confoundments are also obvious in both of the plots. Let's inspect some pairs of images from these classes:

Harpacticoida and Oncaeidae appear very similar:

tail**Appendicularia and tail**Chaetognatha also appear very similar:

multiple**Copepoda and multiple**other were images with more than one organism in each.

multiple**other (the lower of the two) appears especially challenging because the organisms are of many and varying types: this class looks like a dog's breakfast of different species. The asymmetry of 0.18 vs 0.10 in the misclassification of multiple**other for multiple__Copepoda, and vice versa, respectively, probably reflects this.

Overall, these classes look challenging to the eye, and it is not surprising that the classifier was fooled. The surprise may indeed be that they could be identified to quite a reasonable level!

Summary and Conclusion

A vector embedding was trained for zooplankton images using an `Inceptionv3` base model and siamese triplet loss. Classifiers based on this embedding were able to attain almost 90% accuracy. The use of SVD enabled the distribution of class points in the resulting vector space to be more amenably visualised, and the `plotly` "scatter3d" and animation features enabled informative plots to be produced. Confoundment of classes was able to be confirmed visually for a basic SVM classifier, but in the case of the SVM classifier trained on SVD-transformed data, the confoundment was not able to be sensibly interpreted from the plots. This could be either from a bug in the code, or from the complexity of the classifier making visual interpretation of its discriminative ability non-obvious.