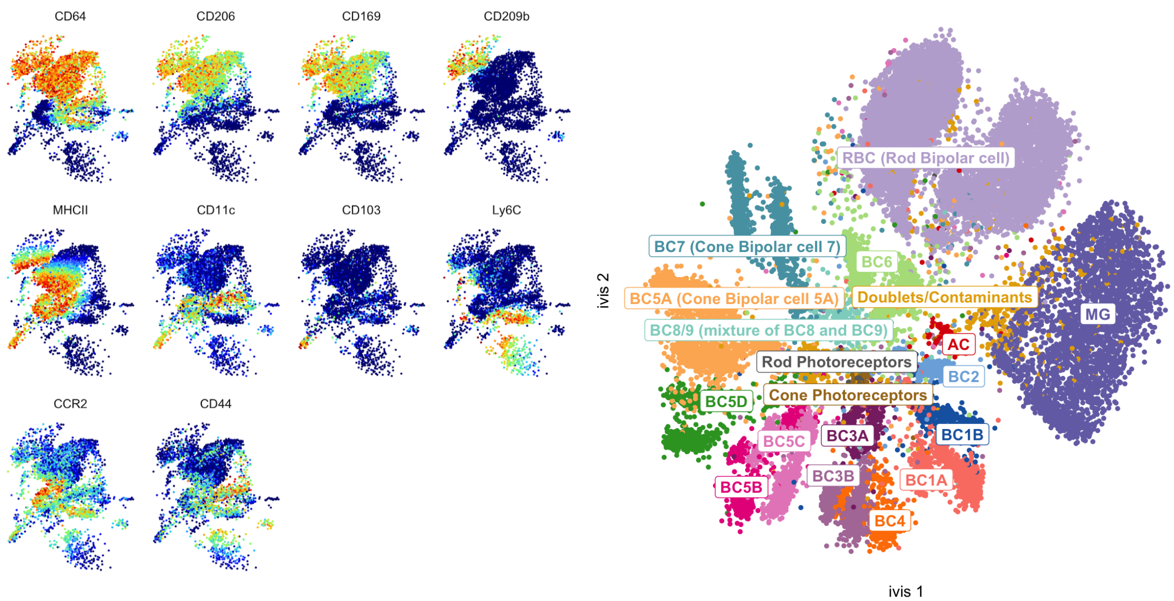
ivis Documentation

Bering Limited

Mar 10, 2022

Get Started



`ivis` is a machine learning library for reducing dimensionality of very large datasets using Siamese Neural Networks. `ivis` preserves global data structures in a low-dimensional space, adds new data points to existing embeddings using a parametric mapping function, and scales linearly to millions of observations. The algorithm is described in detail in Structure-preserving visualisation of high dimensional single-cell datasets.

CHAPTER 1

Features

- Unsupervised, semi-supervised, and fully supervised dimensionality reduction
- Support for arbitrary datasets
  - N-dimensional arrays
  - Image files on disk
  - Custom data connectors
- In- and out-of-memory data processing
- Resumable training
- Arbitrary neural network backbones
- Customizable neighbour retrieval
- Callbacks and Tensorboard integration

The latest development version is on github.

1.1 Python Package

1.1.1 Installation

The latest stable release can be installed from PyPi:

```
#TensorFlow 2 packages require a pip version >19.0.
pip install --upgrade pip
```

```
pip install ivis[cpu]
```

If you have CUDA installed and want ivis to use the tensorflow-gpu package, instead run `pip install ivis[gpu]`.

Note: ZSH users. If you're running ZSH, square brackets are used for globbing / pattern matching. That means *ivis* should be installed as `pip install 'ivis[cpu]'` or `pip install 'ivis[gpu]'`

Alternatively, you can use `pip` to install the development version directly from github:

```
pip install git+https://github.com/beringresearch/ivis.git
```

Another option would be to clone the github repository and install from your local copy:

```
git clone https://github.com/beringresearch/ivis
cd ivis
pip install -e '.[cpu]'
```

1.1.2 Dependencies

- Python 3.5+
- tensorflow
- numpy>1.14.2
- scikit-learn>0.20.0
- tqdm
- annoy

1.1.3 Quickstart

```
from ivis import Ivis
from sklearn.preprocessing import MinMaxScaler
from sklearn import datasets

iris = datasets.load_iris()
X = iris.data

# Scale the data to [0, 1]
X_scaled = MinMaxScaler().fit_transform(X)

# Set ivis parameters
model = Ivis(embedding_dims=2, k=15)

# Generate embeddings
embeddings = model.fit_transform(X_scaled)

# Export model
model.save_model('iris.ivis')
```

1.1.4 Bugs

Please report any bugs you encounter through the github issue tracker. It will be most helpful to include a reproducible example.

1.2 R Package

1.2.1 Installation

Prerequisites

R wrapper for `ivis` is provided via the `reticulate` library. Prior to installation, ensure that *reticulate* is available on your machine.

```
install.packages("reticulate")
```

Next, install virtualenv as it will be used to safely interface with the `ivis` Python package.

Note: Windows Installation. Note that virtual environment functions in the `reticulate` library are not supported on Windows. Instead, conda environment is recommended.

Finally, the easiest way to install `ivis` is using the `devtools` package:

Running install

```
devtools::install_github("beringresearch/ivis/R-package")
library(ivis)
install_ivis()
```

After `ivis` is installed, **restart your R session**.

Note: Newer versions of Keras use tensorflow as the default backend, however if for some reason this isn't the case, add the following line to your environment variables:

```
export KERAS_BACKEND=tensorflow
```

1.2.2 Quickstart

```
library(ivis)
library(ggplot2)

model <- ivis(k = 3)

X <- data.matrix(iris[, 1:4])
X <- scale(X)
model <- model$fit(X)

xy <- model$transform(X)

dat <- data.frame(x=xy[,1], y=xy[,2], species=iris$Species)

ggplot(dat, aes(x=x, y=y)) + geom_point(aes(color=species)) + theme_classic()
```

1.2.3 Vignette

The ivis package includes a vignette that demonstrates an example workflow using single-cell RNA-sequencing data.

To compile and install this vignette on your system, you need to first have a working installation of ivis. For this, please follow the instructions above.

Once you have a working installation of ivis, you can reinstall the package including the compiled vignette using the following command:

```
devtools::install_github("beringresearch/ivis/R-package", build_vignettes = TRUE,␣
↪force=TRUE)
```

1.3 Unsupervised Dimensionality Reduction

Dimensionality Reduction (DR) is the transformation of data from high-dimensional to low-dimensional space, whilst retaining properties of the original data in the low-dimensional space. Downstream applications range from data visualisation to machine learning and feature engineering.

Although many DR approaches exist (e.g. PCA, UMAP, t-SNE), Neural Network (NN) models have been proposed as effective non-linear alternatives. Generally, unsupervised NNs with multiple layers are trained by optimizing a target function, whilst an intermediate layer with small cardinality serves as a low dimensional representation of the input data.

We designed ivis to effectively capture local as well as global features of very large dataset. In our workflows we are applying ivis to millions of data points to effectively capture their behaviour.

1.3.1 The iris dataset

To demonstrate the key features of the ivis algorithm, we will use the well-established iris dataset.

```
from ivis import Ivis
from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaler

data = load_iris()
X = data.data
y = data.target

X = StandardScaler().fit_transform(X)
```

Now, let's set up ivis object.

```
ivis = Ivis(k=15)
ivis.fit(X)
embeddings = ivis.transform(X)

embeddings.shape
```

That's it! Note, that the k parameter is changed from the default value because we only have 150 observations in this dataset. Check out how *hyperparameters can be tuned* to get the most out of ivis for your dataset.

1.3.2 Reducing dimensionality of n-dimensional arrays

`ivis` easily handles n-dimensional arrays. This can be useful in datasets such as imaging, where arrays are typically in (N_SAMPLES, IMG_WIDTH, IMG_HEIGHT, CHANNELS) format. To accomplish this, all we need to do is pass a custom base neural network into ivis that ensures input shapes are captured correctly.

Let's demonstrate this feature using teh `MNSIT` dataset.

```
image_height, image_width = 28, 28
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
x_train = x_train.reshape(x_train.shape[0], image_height, image_width, 1)
x_test = x_test.reshape(x_test.shape[0], image_height, image_width, 1)
input_shape = (image_height, image_width, 1)

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')

x_train /= 255
x_test /= 255
```

We now define the custom neural network that will be used as a feature extractor. Since we are dealing with images, we can use convolutional blocks:

```
def get_base_network(in_shape):
    inputs = tf.keras.layers.Input(in_shape)
    x = tf.keras.layers.Convolution2D(32, (3,3), activation='relu', kernel_
↪initializer='he_uniform')(inputs)
    x = tf.keras.layers.MaxPool2D((2, 2))(x)
    x = tf.keras.layers.Flatten()(x)
    x = tf.keras.layers.Dense(100, activation='relu', kernel_initializer='he_uniform
↪')(x)
    x = tf.keras.layers.Dropout(0.5)(x)

    model = tf.keras.models.Model(inputs, x)
    return model

in_shape = x_train.shape[1:]
base_model = get_base_network(in_shape)
```

Once the network is set up, all we have to do is let `Ivis` know that we will be using a custom network rather than the pre-built one.

```
ivis = Ivis(model=base_model)
ivis.fit(x_train)

embeddings = ivis.transform(x_train)
embeddings.shape
```

All done - you have just reduced dimensionality of an imaging dataset!

If you're looking to extract the finetuned base model from the ivis triplet loss network, you can grab it directlu from the `ivis` object:

```
model = ivis.model_.layers[3]
```

1.3.3 Using custom KNN retreaval

ivis uses Annoy to retreave nearest neighbours during tripplet selection. Annoy was selected as the default option because its fast, accurate and a nearest neighbour index can be built on directly disk, meaning that massive datasets can be processed without the need to load them into memory.

However, many other algorithms exist and new ones are popping up continuously. To accommodate custom nearest neighbour selection, ivis can accept a nearest neighbour matrix directly through the neighbour_matrix hyperparameter.

```
from sklearn.neighbors import NearestNeighbors
nn = NearestNeighbors(n_neighbors=15).fit(X)
neighbours = nn.kneighbors(X, return_distance=False)

ivis = Ivis(neighbour_matrix=neighbours)
ivis.fit(X)
```

1.4 Supervised Dimensionality Reduction

ivis is able to make use of any provided class labels to perform supervised dimensionality reduction. Supervised ivis can thus be used in Metric Learning applications, as well as classical supervised classifier/regressor problems. Supervised embeddings can combine the distance-based characteristics of the unsupervised ivis algorithm with clear class boundaries between the class categories when trained to classify inputs simulateously to embedding them. The resulting embeddings encode relevant class-specific information into lower dimensional space, making them useful for enhancing the performance of a classifier.

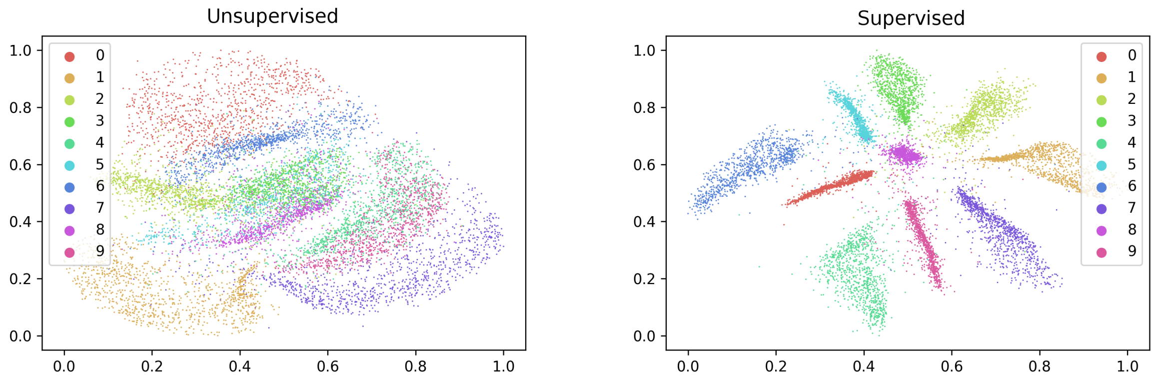
ivis supports both classification and regression problems and makes use of the losses included with keras, so long as the labels are provided in the correct format.

1.4.1 Classification

To train ivis in supervised mode using the default softmax classification loss, simply provide the labels to the fit method's Y parameter. These labels should be a list of 0-indexed integers with each integer corresponding to a class.

```
import numpy as np
from tensorflow.keras.datasets import mnist
from ivis import Ivis

(X_train, Y_train), (X_test, Y_test)  = mnist.load_data()

# Rescale to [0,1]
X_train = X_train / 255.
X_test = X_test / 255.

# Flatten images to 1D vectors
X_train = np.reshape(X_train, (len(X_train), 28 * 28))
X_test = np.reshape(X_test, (len(X_test), 28 * 28))

model = Ivis(n_epochs_without_progress=5)
model.fit(X_train, Y_train)
```

Experimental data has shown that ivis converges to a solution faster in supervised mode. Therefore, our suggestion is to lower the value of the n_epochs_without_progress parameter from the default to around 5. Here are the resulting embeddings:

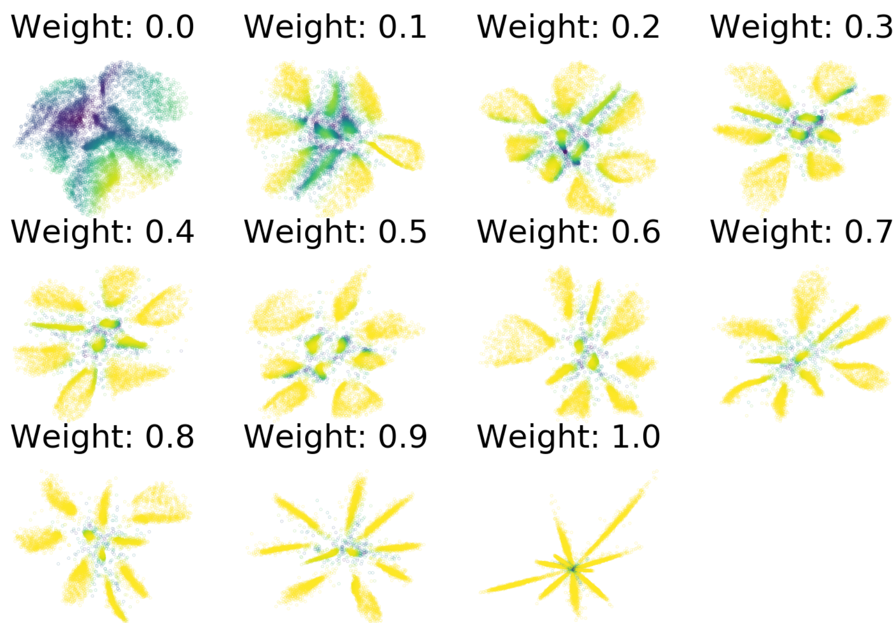


Obtaining Classification Probabilities

Since training `ivis` in supervised mode causes the algorithm to optimize the supervised objective in conjunction with the triplet loss function, it is possible to obtain the outputs of the supervised network using the `score_samples` method. These may be useful for assessing the quality of the embeddings by examining the performance of the classifier, for example, or for predicting the labels for unseen data.

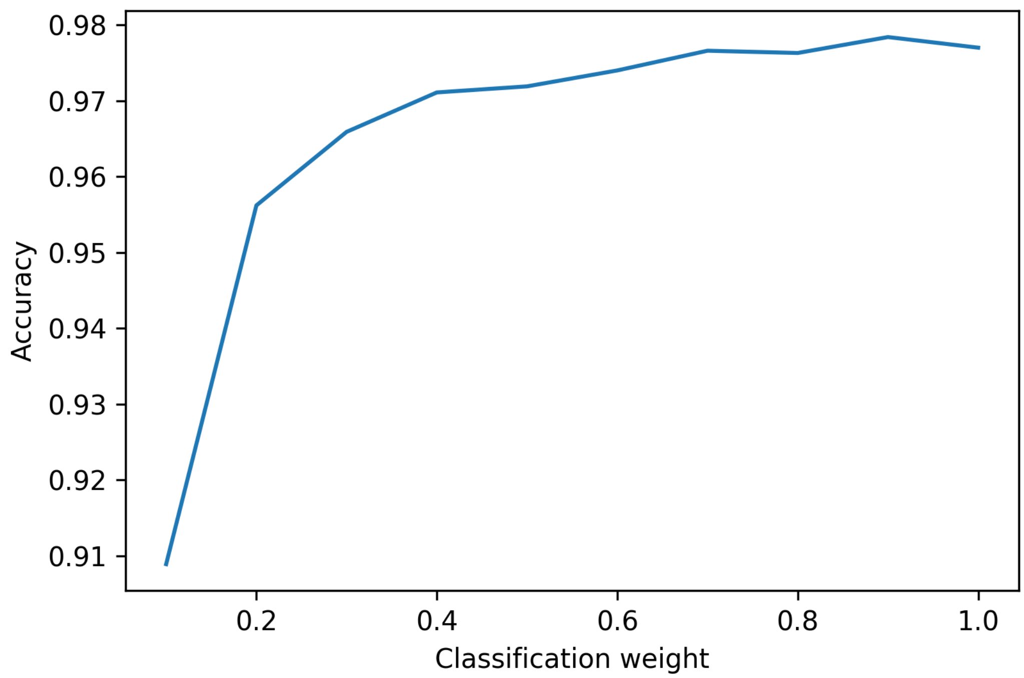
```
weight = 0.8
model = Ivis(n_epochs_without_progress=5,
             supervision_weight=weight)
model.fit(X_train, Y_train)

embeddings = model.transform(X_test)
y_pred = model.score_samples(X_test)
```

As before, we can train several supervised `ivis` models on the MNIST dataset, varying the supervision_weight parameter, coloring the plots according to the max of the returned softmax probabilities.



Coloring by the max softmax probabilities shows the degree of certainty in the supervised network's predictions - areas that are yellow are predicted with a higher degree of confidence while those in blue and green have a lower degree of confidence. With low supervision weight, more of the data is classified with a low degree of certainty. Additionally, points floating in the centre between clusters tend to have lower class predictions associated with them.

We also checked the accuracy of the ivis classifiers when used to predict the test set labels across the different supervision weights. In general, increasing the supervision weight improved the classifier's predictive performance on the test set, with maximum performance achieved with a weight of 0.9. At this weight the triplet loss continues to have a small regularizing effect on the results, which may improve the generalizability of the classifier compared to a pure softmax classifier.



Linear-SVM classifier

It's also possible to utilize different supervised metrics to train the supervised network by adjusting the `supervsed_metric` parameter. By selecting `categorical_hinge` it is possible to optimize a linear SVM on the data in conjunction with the triplet loss.

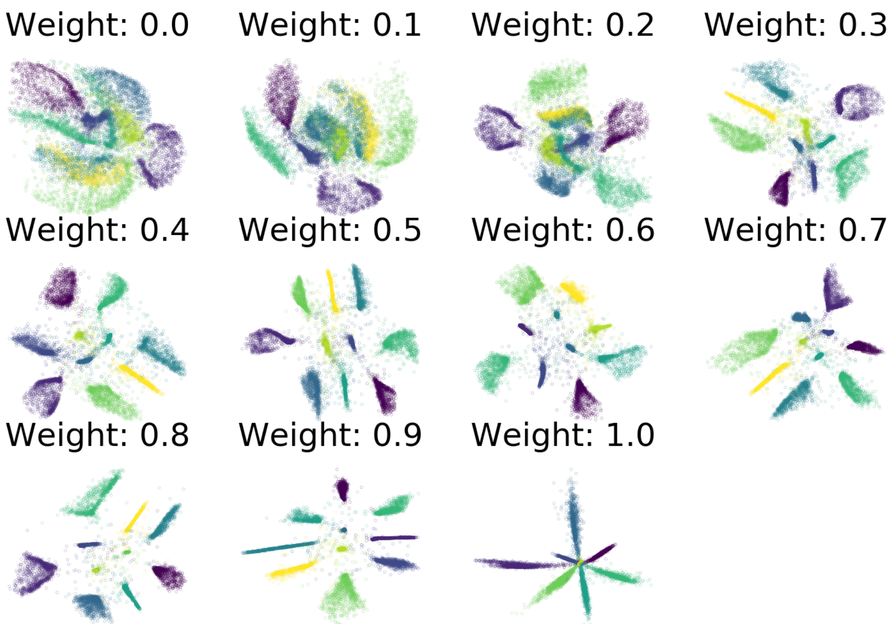
Below is an example of training `ivis` in supervised mode in tandem with a linear SVM on the Fashion MNIST dataset. Note that the `categorical_hinge` loss function expects one-hot encoded labels. We can achieve this using the `to_categorical` function from keras utils.

```
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
from ivis import Ivis
import numpy as np

(X_train, Y_train), (X_test, Y_test) = mnist.load_data()

# Flatten images
X_train = np.reshape(X_train, (len(X_train), 28 * 28)) / 255.
X_test = np.reshape(X_test, (len(X_test), 28 * 28)) / 255.

# One-hot encode labels
Y_train = to_categorical(Y_train)
Y_test = to_categorical(Y_test)

model = Ivis(n_epochs_without_progress=5,
             supervision_metric='categorical_hinge')
model.fit(X_train, Y_train)
```

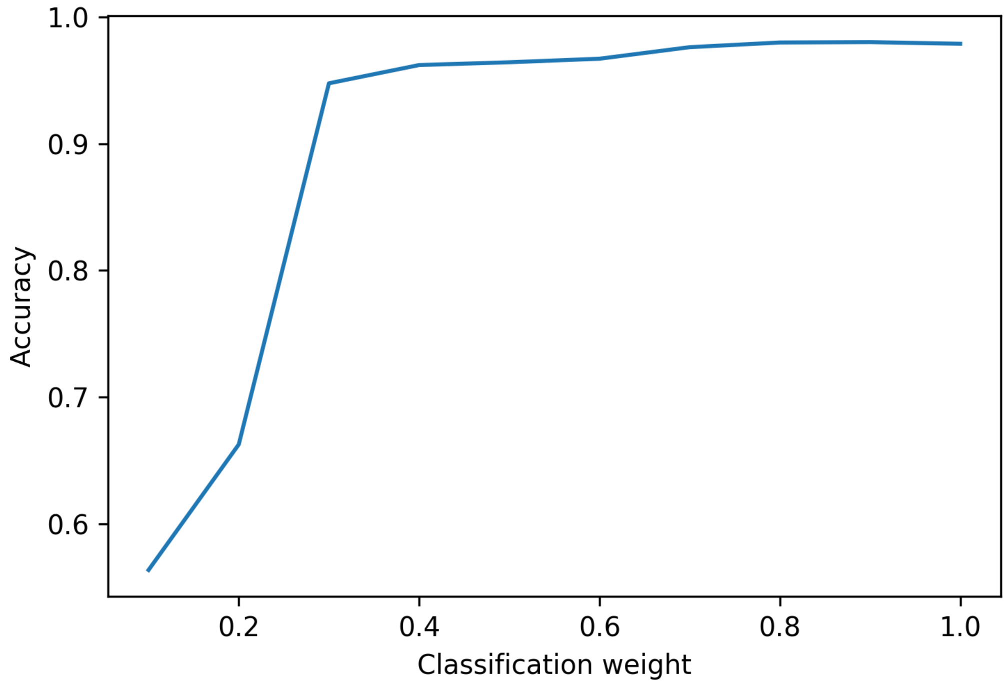
(continues on next page)

(continued from previous page)

```
embeddings = model.transform(X_test)
y_pred = model.score_samples(X_test)
```



The resulting embeddings show ivis trained with a Linear SVM using the `categorical_hinge` metric over a variety of supevision_weight values. The maximum achieved accuracy on the test set was 98.02% - once again, a supervision weight of 0.9 led to the highest classification performance.



Multi-label classification

In cases where a single observation is accompanied by multiple response variables, `ivis` implements support for multi-label classification. Ensuring that `y` is a multi-dimensional array (N x L), where L is the number of unique labels, multi-label model can be fitted as:

```
ivis = Ivis(k=30, model='maaten',
            supervision_metric='binary_crossentropy')
ivis.fit(x, y)
```

Note that the only requirement is that supervision metric is set to `binary_crossentropy`.

1.4.2 Regression

It is also possible to perform supervised training on continous labels. To do this, a regression metric should be provided to `supervision_metric` when constructing the Ivis object. Many of these exist in Keras, including mean-absolute-error, mean-squared error, and logcosh.

In the example below, `ivis` is trained on the boston housing dataset using the mean-absolute-error supervised metric (mae).

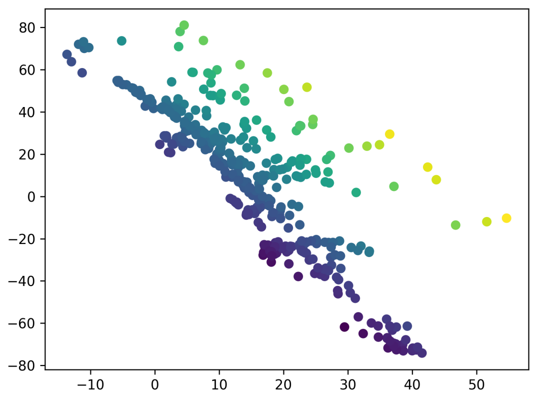
```
from ivis import Ivis
from tensorflow.keras.datasets import boston_housing
import numpy as np

(X_train, Y_train), (X_test, Y_test) = boston_housing.load_data()

supervision_metric = 'mae'
```

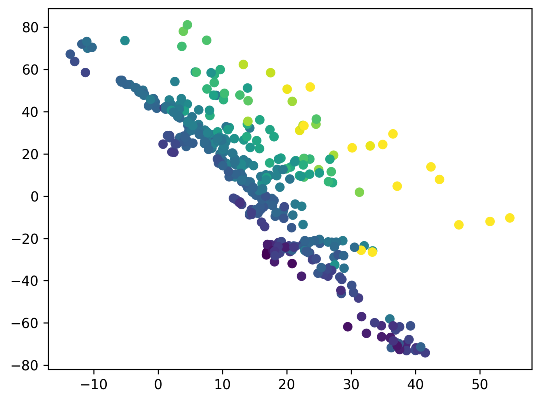
(continues on next page)

(continued from previous page)

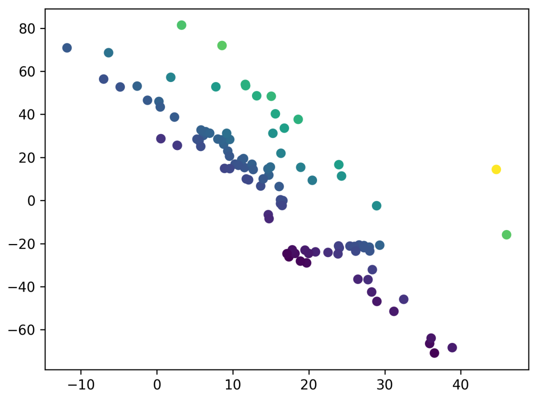
```
ivis_boston = Ivis(k=15, batch_size=16, supervision_metric=supervision_metric)
ivis_boston.fit(X_train, Y_train)

train_embeddings = ivis_boston.transform(X_train)
y_pred_train = ivis_boston.score_samples(X_train)

test_embeddings = ivis_boston.transform(X_test)
y_pred_test = ivis_boston.score_samples(X_test)
```

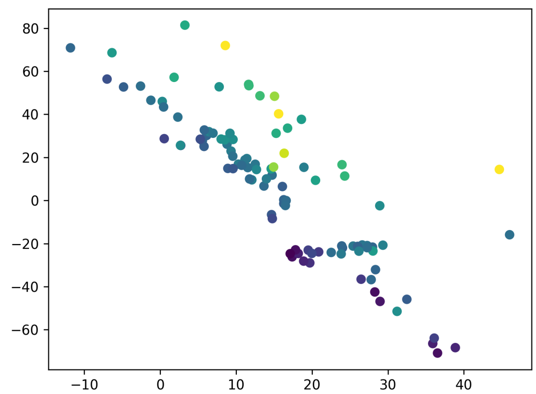
The embeddings on the training set are shown below. On the left are the embeddings are colored by the ground truth label; the right is colored by predicted values. There is a high degree of correlation between the predicted and actual values, with an R-squared value of 0.82.





The embeddings on the test set are below. Again, the left is colored by the ground truth label, while the right is colored by predicted values. There is a also a high degree of correlation between the predicted and actual values on the test set, although it is lower than on the training set - the R-squared value is 0.63.



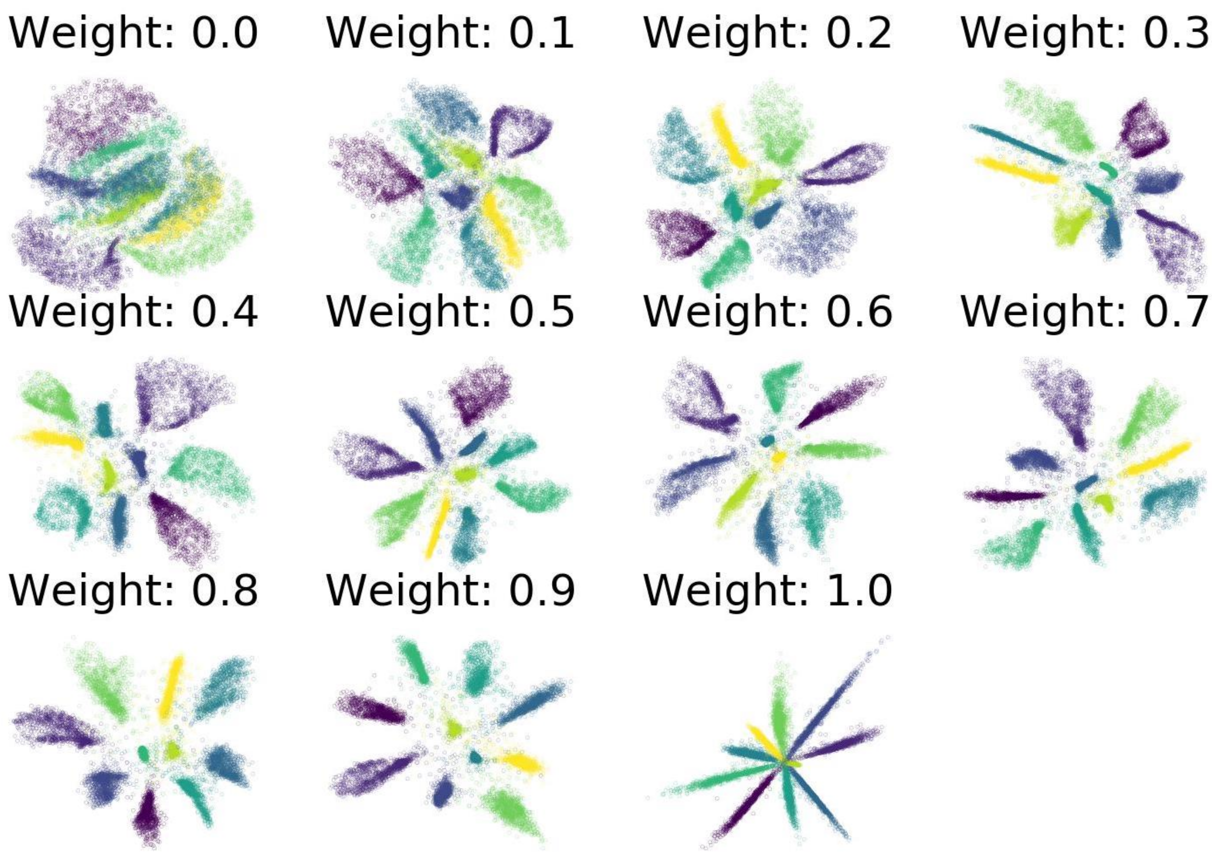


1.4.3 Supervision Weight

It is possible to control the relative importance `ivis` places on the labels when training in supervised mode with the `supervision_weight` parameter. This variable should be a float between 0.0 to 1.0, with higher values resulting in supervision affecting the training process more, and smaller values resulting in it impacting the training less. By default, the parameter is set to 0.5. Increasing it to 0.8 will result in more cleanly separated classes.

```
weight = 0.8
model = Ivis(n_epochs_without_progress=5,
             supervision_weight=weight)
model.fit(X_train, Y_train)
```

As an illustration of the impact the `supervision_weight` has on the resulting embeddings, see the following plot of supervised `ivis` applied to MNIST with different weight values:



1.5 Semi-supervised Dimensionality Reduction

Sometimes only part of a dataset has ground-truth labels available. In such a scenario, `ivis` is still able to make use of existing label information in conjunction with the inputs to do dimensionality reduction when in semi-supervised mode. When in semi-supervised mode, `ivis` will use labels when available as well as the unsupervised triplet loss. However, when label information is not available, only the unsupervised loss will be used. By training in semi-supervised mode, we can make full use of the data available, even if it is only partially labeled.

In order to use semi-supervised learning, mark missing labeled points as -1 in the Y vector provided to `ivis` when calling *fit* or *fit_transform*. Currently, only *sparse_categorical_crossentropy* loss works with semi-supervised inputs.

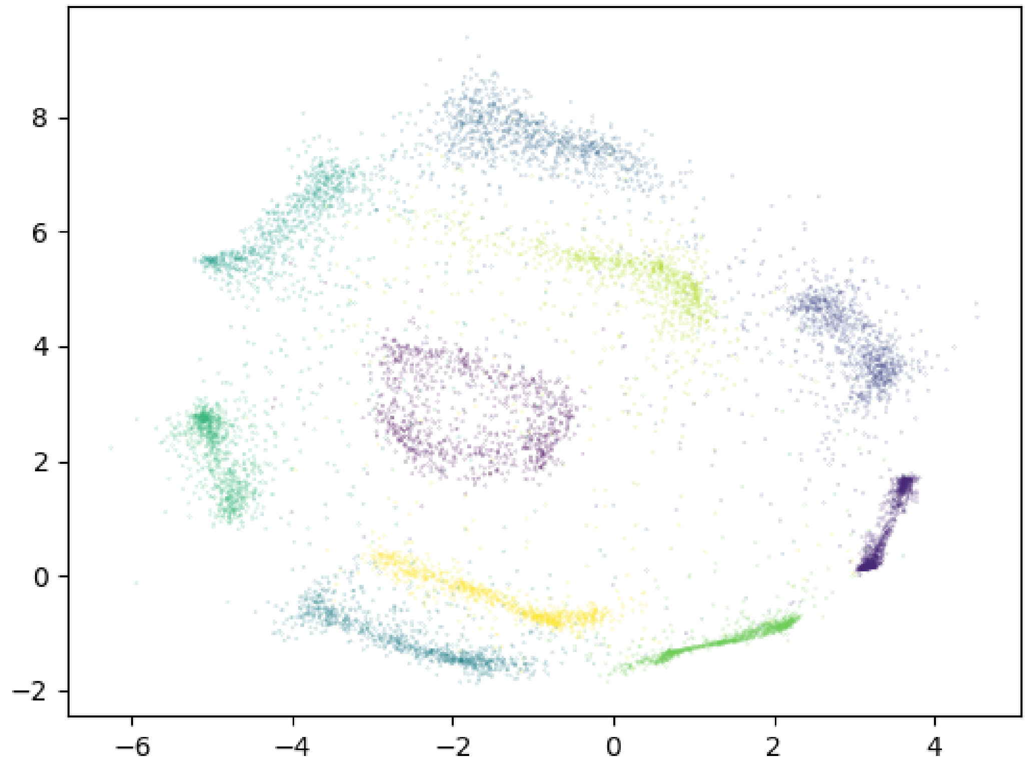
1.5.1 Semi-supervised Classification

To train `ivis` in semi-supervised mode using the default softmax classification loss, simply provide the labels to the fit method's `Y` parameter. These labels should be a list of 0-indexed integers with each integer corresponding to a class. Missing labels should be denoted with -1.

In the example below, we will mask 50% of the available labels for the MNIST dataset.

```
import numpy as np
from tensorflow.keras.datasets import mnist
from ivis import Ivis

(X_train, Y_train), (X_test, Y_test)  = mnist.load_data()

# Rescale to [0,1]
X_train = X_train / 255.
X_test = X_test / 255.

# Flatten images to 1D vectors
X_train = np.reshape(X_train, (len(X_train), 28 * 28))
X_test = np.reshape(X_test, (len(X_test), 28 * 28))

# Mask labels
mask = np.random.choice(range(len(Y_train)), size=len(Y_train) // 2, replace=False)
Y_train_masked = np.array(Y_train, dtype=np.int8) # Can't use uint to represent negative numbers
Y_train_masked[mask] = -1

model = Ivis(n_epochs_without_progress=5)
model.fit(X_train, Y_train_masked)
```

Experimental data has shown that `ivis` converges to a solution faster in supervised mode. Therefore, our suggestion is to lower the value of the `n_epochs_without_progress` parameter from the default to around 5. Here are the resulting embeddings on the testing set:

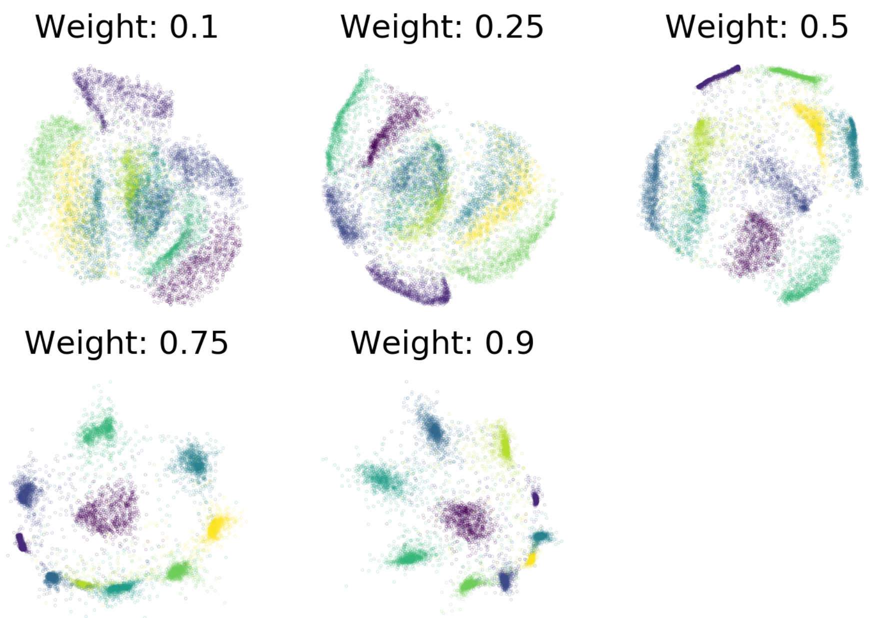


1.5.2 Supervision Weight

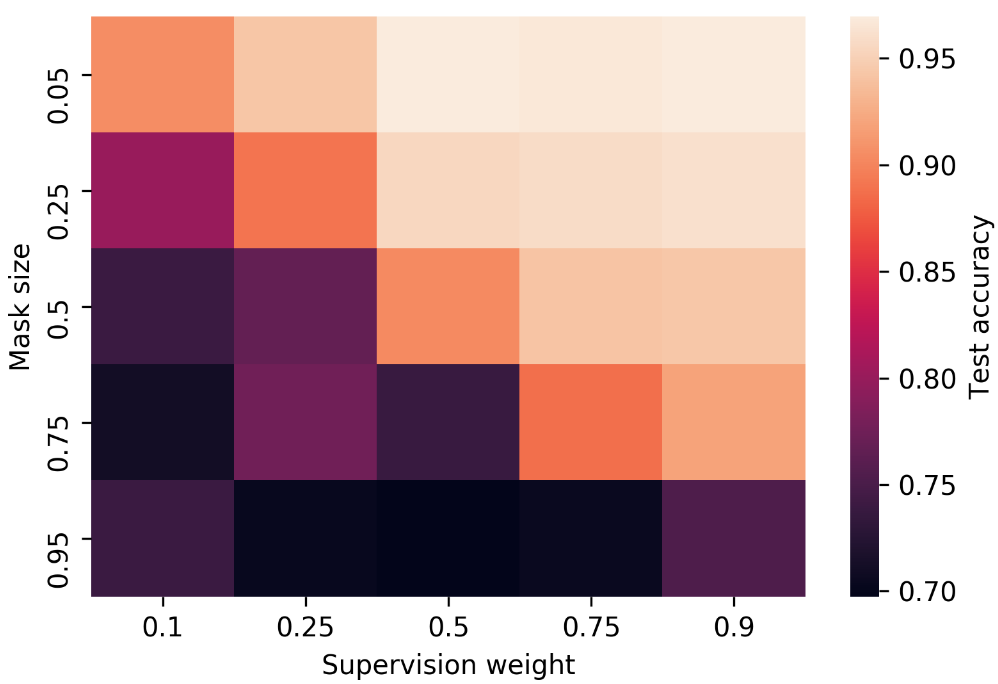
As in supervised mode, it is still possible to control the relative importance ivis places on the labels when training in supervised mode with the supervision_weight parameter. This variable should be a float between 0.0 to 1.0, with higher values resulting in supervision affecting the training process more, and smaller values resulting in it impacting the training less. By default, the parameter is set to 0.5. Increasing it will result in more cleanly separated classes.

```
weight = 0.8
model = Ivis(n_epochs_without_progress=5,
             supervision_weight=weight)
model.fit(X_train, Y_train)
```

As an illustration of the impact the supervision_weight has on the resulting embeddings, see the following plot of supervised ivis applied to MNIST with different weight values:



In semi-supervised mode, the supervision weight may need to be higher to have the same effect on the resulting embeddings as in supervised mode, depending on the dataset. This is because when unlabeled points are encountered, unsupervised loss will still have an impact, while the supervised loss will not apply. The more of the dataset is unlabeled, the higher the supervision weight should be to have an impact on the embeddings.



1.6 Hyperparameter Selection

`ivis` uses several hyperparameters that can have an impact on the desired embeddings:

- `embedding_dims`: Number of dimensions in the embedding space.
- `k`: The number of nearest neighbours to retrieve for each point.
- `n_epochs_without_progress`: After n number of epochs without an improvement to the loss, terminate training early.
- `model`: the keras model that is trained using triplet loss. If a model object is provided, an embedding layer of size `embedding_dims` will be appended to the end of the network. If a string is provided, a pre-defined network by that name will be used. Possible options are: 'szubert', 'hinton', 'maaten'. By default the 'szubert' network will be created, which is a selu network composed of 3 dense layers of 128 neurons each, followed by an embedding layer of size 'embedding_dims'.

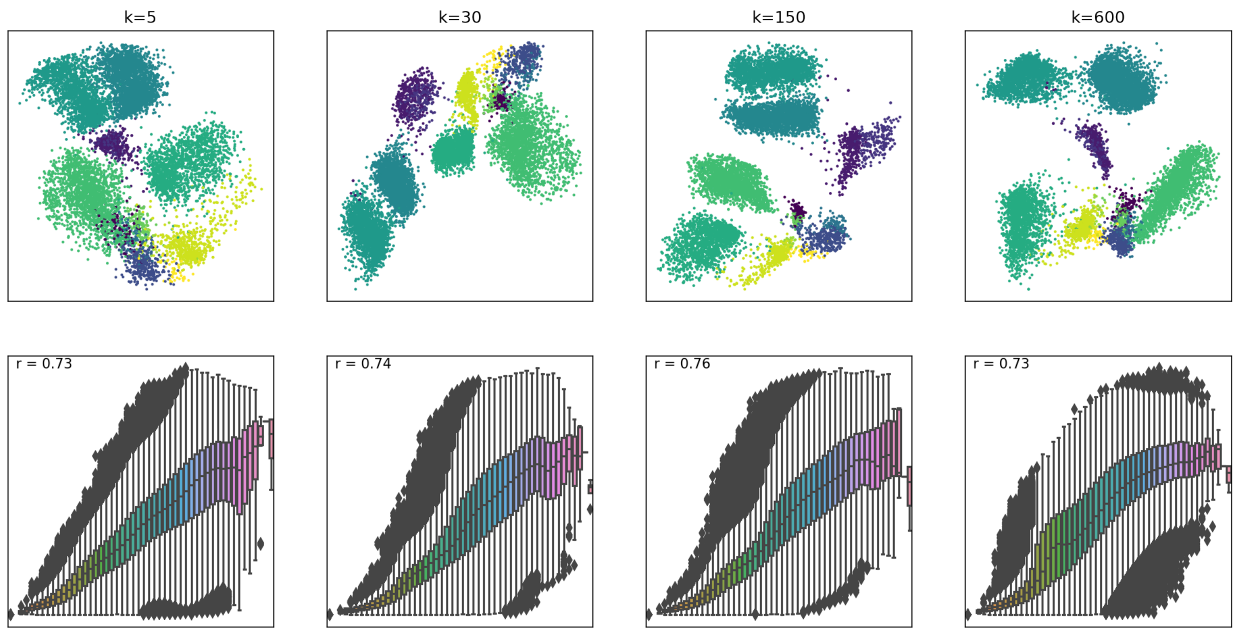
`k` , `n_epochs_without_progress`, and `model` are tunable parameters that should be selected on the basis of dataset size and complexity. The following table summarizes our findings:

Observations	`k`	`n_epochs_without_progress`	`model`
< 1000	10-15	20-30	maaten
1000-10000	10-30	10-20	maaten
10000-50000	15-150	10-20	maaten
50K-100K	15-150	10-15	maaten
100K-500K	15-150	5-10	maaten
500K-1M	15-150	3-5	szubert
> 1M	15-150	2-3	szubert

We will now look at each of these parameters in turn.

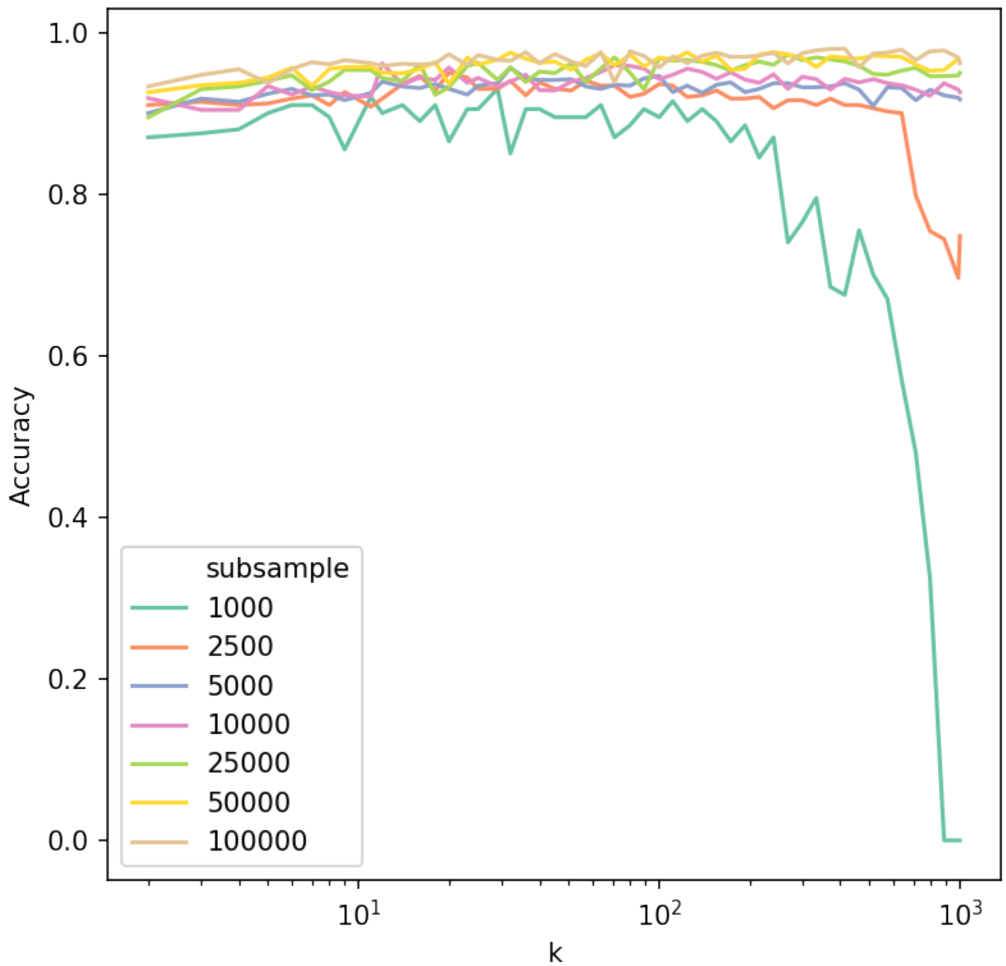
1.6.1 `k`

This parameter controls the balance between local and global features of the dataset. Low `k` values will result in prioritisation of local dataset features and the overall global structure may be missed. Conversely, high `k` values will force `ivis` to look at broader aspects of the data, losing desired granularity. We can visualise effects of low and large values on `k` on the Levine dataset (104,184 x 32).



Box plots represent distances across pairs of points in the embeddings, binned using 50 equal-width bins over the pairwise distances in the original space using 10,000 randomly selected points, leading to 49,995,000 pairs of pairwise distances. For each embedding, the value of the Pearson correlation coefficient computed over the pairs of pairwise distances is reported. We can see that where `k=5`, smaller distances are better preserved, whilst larger distances have higher variability in the embedding space. As `k` values increase, larger distances are beginning to be better preserved as well. However, for very large `k`, smaller distances are no longer preserved.

To establish an appropriate value of `k`, we evaluated a range of values across a severao subsamples of varying sizes, keeping `n_epochs_without_progress` and `model` hyperparameters fixed.



Accuracy was calculated by training a Support Vector Machine classifier on 75% of each subsample and evaluating the classifier performance on the remaining 25%, whilst predicting manually assigned cell types in the Levine dataset. Accuracy was high and generally stable for `k` between 10 and 150. A decrease was observed when `k` was considerably large in relation to subsample size.

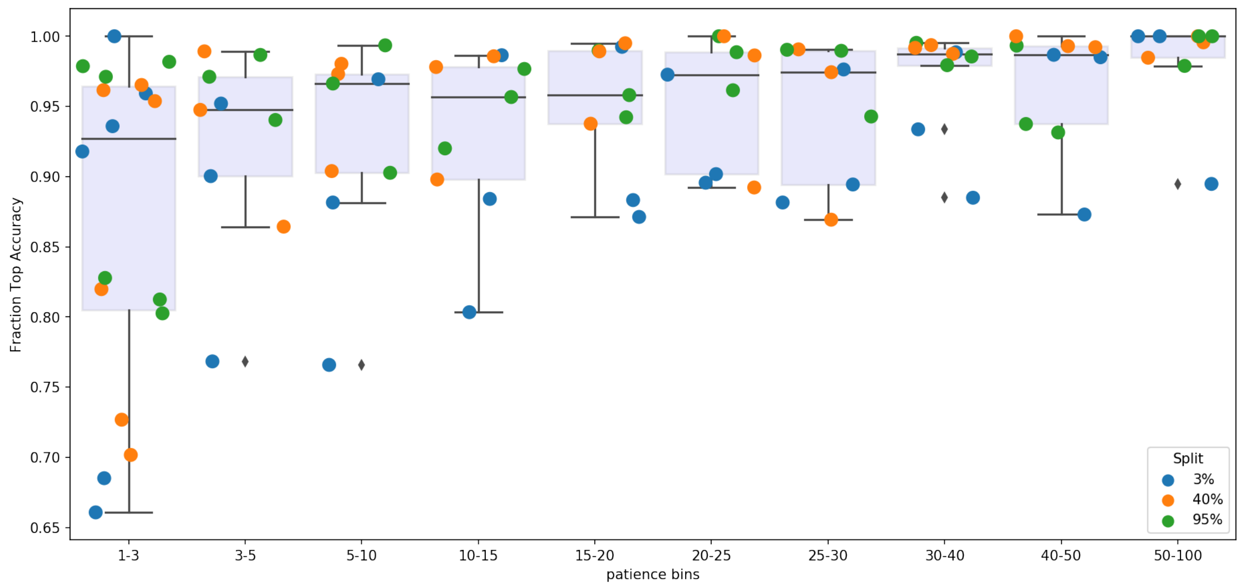
Overall, `ivis` is fairly robust to values of `k`, which can control the local vs. global trade off in the embedding space.

1.6.2 `n_epochs_without_progress`

This patience hyperparameter impacts both the quality of embeddings and speed with which they are generated. Generally, the higher `n_epochs_without_progress` are, the more accurate are the low-dimensional features. However, this comes at a computational cost. Here we examine, the speed vs. accuracy trade-off and recommend sensible defaults. For this experiment `ivis` hyperparameters were set to `k=15` and `model='maaten'`.

Three datasets were used Levine (104,184 x 32), MNIST (70,000 x 784), and Melanoma (4,645 x 23,686). The Melanoma featurespace was further reduced to n=50 using Principal Component Analysis.

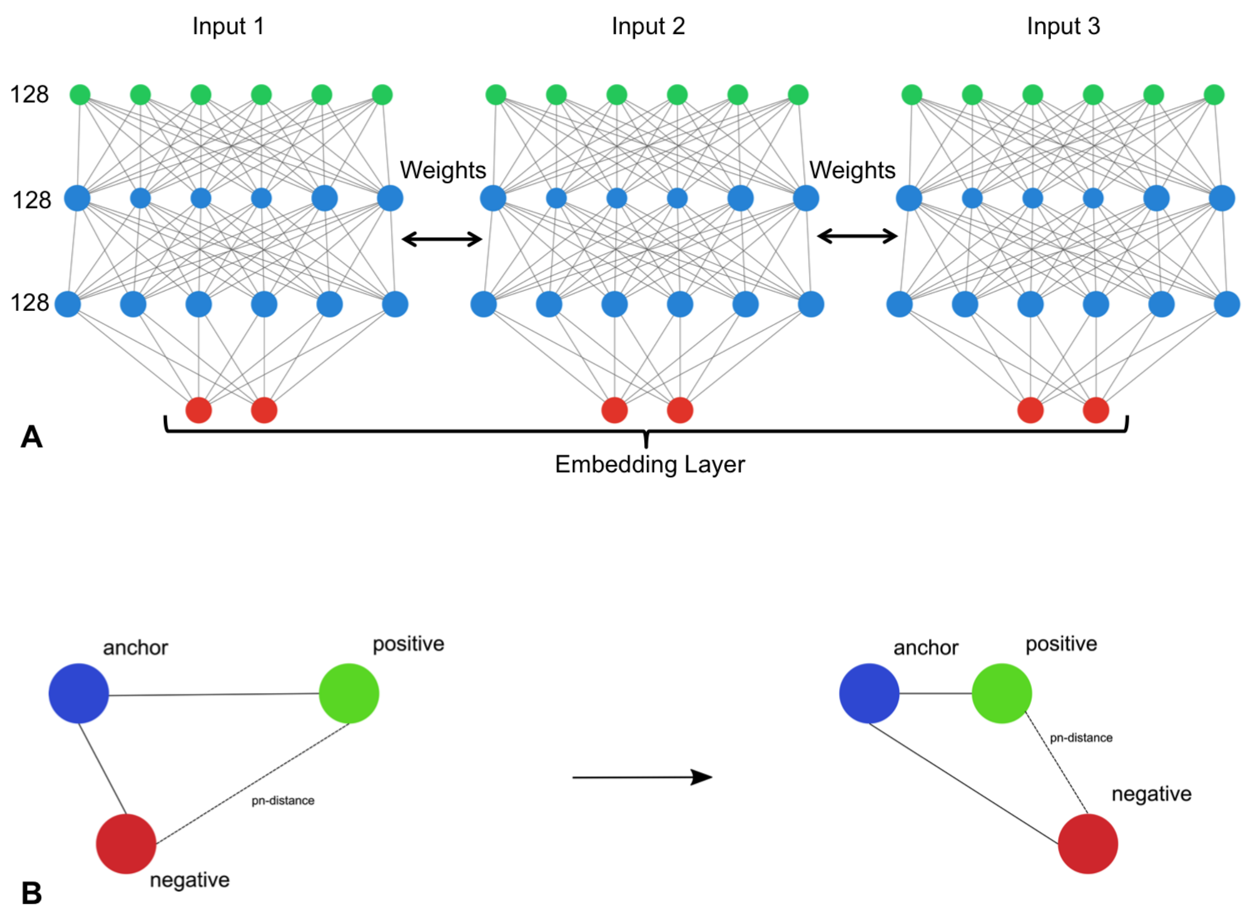
For each dataset, we trained a Support Vector Machine classifier to assess how well `ivis` embeddings capture manually supplied response variable information. For example, in case of an MNIST dataset, the response variable is the digit label, whilst for Levine and Melanoma datasets it is the cell type. SVM classifier was trained on `ivis` embeddings representing 3%, 40%, and 95% of the data obtained using a stratified random subsampling. The classifier was then validated on the `ivis` embeddings of the remaining 97%, 60%, and 5% of data. For each training set split, an `ivis` model was trained by keeping the `k` and `model` hyperparameters constat, whilst varying n_epochs_without_progress. Finally, classification accuracies were noramlised to a 0-1 range to facilitate comparisons between datasets.



Our final results indicate that oveall accuracy of embeddings is a function of dataset size and n_epochs_without_progress. However, only marginal gain in performance is achieved when n_epochs_without_progress>20. For large datasets (n_observations>10000), n_epochs_without_progress between 3 and 5 comes to within 85% of optimal classification accuracy.

1.6.3 model

The model hyperparameter is a powerful way for ivis to handle complex non-linear feature-spaces. It refers to a trainable neural network that learns to minimise a triplet loss loss function. Structure-preserving dimensionality reduction is achieved by creating three replicates of the baseline architecture and assembling these replicates using a siamese neural network (SNNs). SNNs are a class of neural network that employ a unique architecture to naturally rank similarity between inputs. The ivis SNN consists of three identical base networks; each base network is followed by a final embedding layer. The size of the embedding layer reflects the desired dimensionality of outputs.



model parameter is defined using a keras model. This flexibility allows ivis to be trained using complex architectures and patterns, including convolutions. Out of the box, ivis supports three styles of baseline architectures - **szubert**, **hinton**, and **maaten**. This can be passed as string values to the model parameter.

'szubert'

The **szubert** network has three dense layers of 128 neurons followed by a final embedding layer (128-128-128). The size of the embedding layer reflects the desired dimensionality of outputs. The layers preceding the embedding layer use the SELU activation function, which gives the network a self-normalizing property. The weights for these layers are randomly initialized with the LeCun normal distribution. The embedding layers use a linear activation and have their weights initialized using Glorot's uniform distribution.

'hinton'

The **hinton** network has three dense layers (2000-1000-500) followed by a final embedding layer. The size of the embedding layer reflects the desired dimensionality of outputs. The layers preceding the embedding layer use the SELU activation function. The weights for these layers are randomly initialized with the LeCun normal distribution. The embedding layers use a linear activation and have their weights initialized using Glorot's uniform distribution.

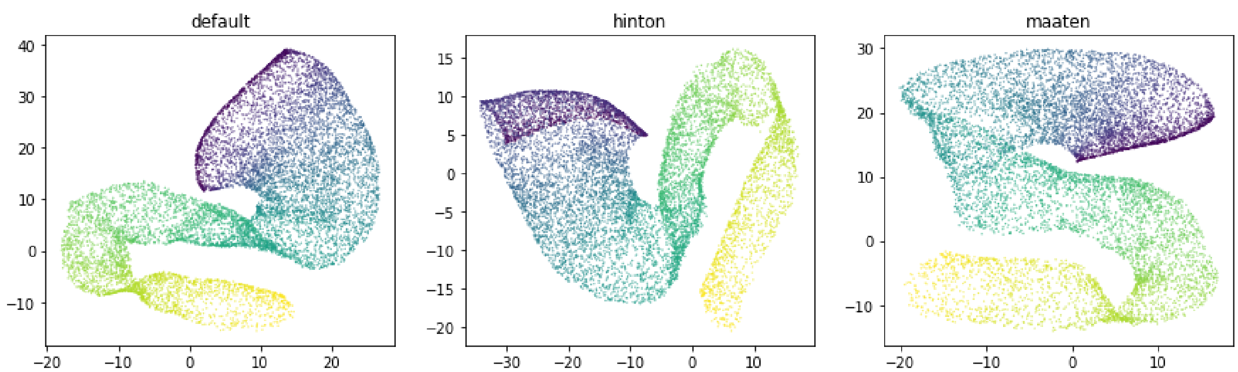
'maaten'

The **maaten** network has three dense layers (500-500-2000) followed by a final embedding layer. The size of the embedding layer reflects the desired dimensionality of outputs. The layers preceding the embedding layer use the SELU activation function. The weights for these layers are randomly initialized with the LeCun normal distribution. The embedding layers use a linear activation and have their weights initialized using Glorot's uniform distribution.

Let's examine each architectural option in greater detail:

```
architecture = ['szubert', 'hinton', 'maaten']
embeddings = {}
for a in architecture:
    ivis = Ivis(k=150).fit(X_poly)
    embeddings[a] = ivis.transform(X_poly)
```

```
fig, axs = plt.subplots(1, 3, figsize=(15, 4), facecolor='w', edgecolor='k')
fig.subplots_adjust(hspace = 0.3, wspace = 0.2)

axs = axs.ravel()
for i, nn in enumerate(architecture):
    xy=embeddings[nn]
    axs[i].scatter(xy[:, 0], xy[:, 1], s = 0.1, c = y)
    axs[i].set_title(nn)
```



Selecting an appropriate baseline architecture is a data-driven task. Three unique architectures that are shipped with ivis perform consistently well across a wide array of tasks. A general rule of thumb in our own experiments is to use the **szubert** network for computationally-intensive processing on large datasets (>1 million observations) and select **maaten** architecture for smaller real-world datasets.

Custom backbone

Ivis also supports construction of arbitrary headless models, which can be helpful when dealing with multi-dimensional datasets such as images or text.

```
from tf.keras.applications.inception_v3 import InceptionV3

base_model = InceptionV3(include_top=False, pooling='avg')
ivis = Ivis(model=base_model)
ivis.fit()
```

1.7 Examples

You can find here a list of notebooks demonstrating key Ivis functionalities. Also, we would like to list here interesting content created by the community. If you wrote some notebook(s) leveraging Ivis and would like to be listed here, please open a Pull Request so it can be included under the Community notebooks.

Notebook	Description	Colab
How to reduce dimensionality of structured data	Introduction to Ivis - reduce dimensionality of structured data	
How to reduce dimensionality of image data	Reduce dimensionality of image data using Ivis algorithm and a custom convolutional neural network	
Using callbacks to assess model training	Apply callbacks during ivis training to log and assess intermediate model states.	
Concept drift detection on image data	Detect Concept Drift in image datasets using Ivis.	

1.8 Using `ivis` for Dimensionality Reduction of Single Cell Experiments

This example will demonstrate how `ivis` can be used to visualise single cell experiments. Data import, preprocessing and normalisation are handled by the Scanpy module. The data that will be used in this example consists of 3,000 PBMCs from a healthy donor and is freely available from 10x Genomics. Now, let's download the data and get started.

```
mkdir data
wget http://cf.10xgenomics.com/samples/cell-exp/1.1.0/pbmc3k/pbmc3k_filtered_gene_bc_
↪matrices.tar.gz -O data/pbmc3k_filtered_gene_bc_matrices.tar.gz
cd data; tar -xzf pbmc3k_filtered_gene_bc_matrices.tar.gz
```

```
import numpy as np
import pandas as pd
import scanpy as sc

sc.settings.verbosity = 3
sc.logging.print_versions()
results_file = './write/pbmc3k.h5ad'

adata = sc.read_10x_mtx(
    './data/filtered_gene_bc_matrices/hg19/',  # the directory with the `.mtx` file
    var_names='gene_symbols',                   # use gene symbols for the variable␣
↪names (variables-axis index)
    cache=True)                                 # write a cache file for faster␣
↪subsequent reading
```

We can now carry out basic filtering and handling of mitochondrial genes:

```
adata.var_names_make_unique()
sc.pp.filter_cells(adata, min_genes=200)
sc.pp.filter_genes(adata, min_cells=3)

mito_genes = adata.var_names.str.startswith('MT-')
# for each cell compute fraction of counts in mito genes vs. all genes
# the `.A1` is only necessary as X is sparse (to transform to a dense array after␣
↪summing)
```

(continues on next page)

(continued from previous page)

```
adata.obs['percent_mito'] = np.sum(
    adata[:, mito_genes].X, axis=1).A1 / np.sum(adata.X, axis=1).A1
# add the total counts per cell as observations-annotation to adata
adata.obs['n_counts'] = adata.X.sum(axis=1).A1

adata = adata[adata.obs['n_genes'] < 2500, :]
adata = adata[adata.obs['percent_mito'] < 0.05, :]
```

Let's normalise the data and apply log-transformation:

```
sc.pp.normalize_per_cell(adata, counts_per_cell_after=1e4)
sc.pp.log1p(adata)
adata.raw = adata
```

Identify highly-variable genes and do the filtering:

```
sc.pp.highly_variable_genes(adata, min_mean=0.0125, max_mean=3, min_disp=0.5)

adata = adata[:, adata.var['highly_variable']]

sc.pp.regress_out(adata, ['n_counts', 'percent_mito'])
```

It's recommended to apply PCA-transformation of normalised data - this step tends to denoise the data.

```
sc.pp.scale(adata, max_value=10)
sc.tl.pca(adata, svd_solver='arpack')
```

1.8.1 Reducing Dimensionality Using `ivis`

```
import matplotlib.pyplot as plt
from ivis import Ivis
```

For most single cell datasets, the following hyperparameters can be used:

- `k=15`
- `model='maaten'`
- `n_epochs_without_progress=5`

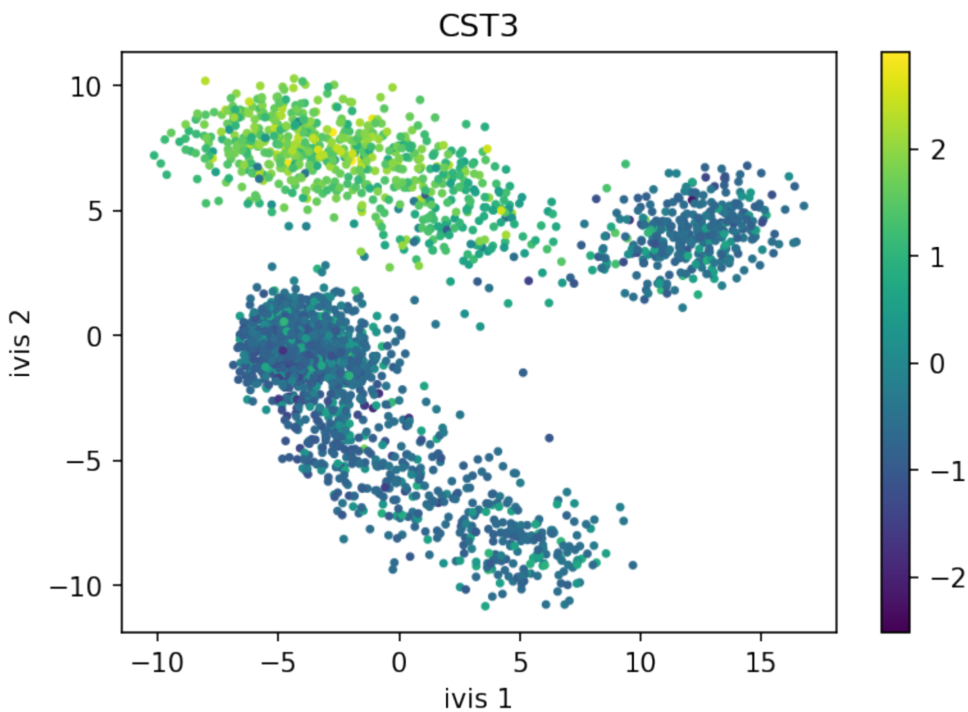
Note: Keep in mind that this is a very small experiment (<3000 observations) and there are plenty of fast and accurate algorithm designed for these kinds of datasets e.g. UMAP. However, if you have >250,000 cells, `ivis` considerably outperforms state-of-the-art both in speed and accuracy of embeddings. See our *timings benchmarks* for more information on this.

```
X = adata.obsm['X_pca']

ivis = Ivis(k=15, model='maaten', n_epochs_without_progress=5)
ivis.fit(X)
embeddings = ivis.transform(X)
```

Finally, let's visualise our embeddings, coloured by the CST3 gene!

```
fill = adata.X[:, adata.var.gene_ids.index=='CST3']
fill = fill.reshape((X.shape[0], ))
```

```
plt.figure(figsize=(6, 4), dpi=150)
sc = plt.scatter(x=embeddings[:, 0], y=embeddings[:, 1], c=fill, s=5)
plt.xlabel('ivis 1')
plt.ylabel('ivis 2')
plt.title('CST3')
plt.colorbar(sc)
```



ivis effectively captured three distinct cellular populations in this small dataset. Note that ivis is an "honest" algorithm and distances between observations are meaningful. Our benchmarks show that ivis is ~10% better at preserving local and global distances in low-dimensional space than comparable state-of-the-art algorithms. Additionally, ivis is robust against noise and outliers, ulike t-SNE, which tends to group random noise into well-defined clusters that can be potentially misleading.

1.9 Comparing ivis with other dimensionality reduction algorithms

Ivis aims to reduce data dimensionality whilst preserving both global and local structures. There are a number of real-world applications where this feature could be useful. For example:

- Anomaly detection
- Biological interpretation of high-throughput experiments
- Feature extraction

Several algorithms have been proposed to address the problem of dimensionality reduction, including UMAP and t-SNE. UMAP in particular, has been succesfully applied in machine learning pipelines. Ivis is different to these approaches in several ways.

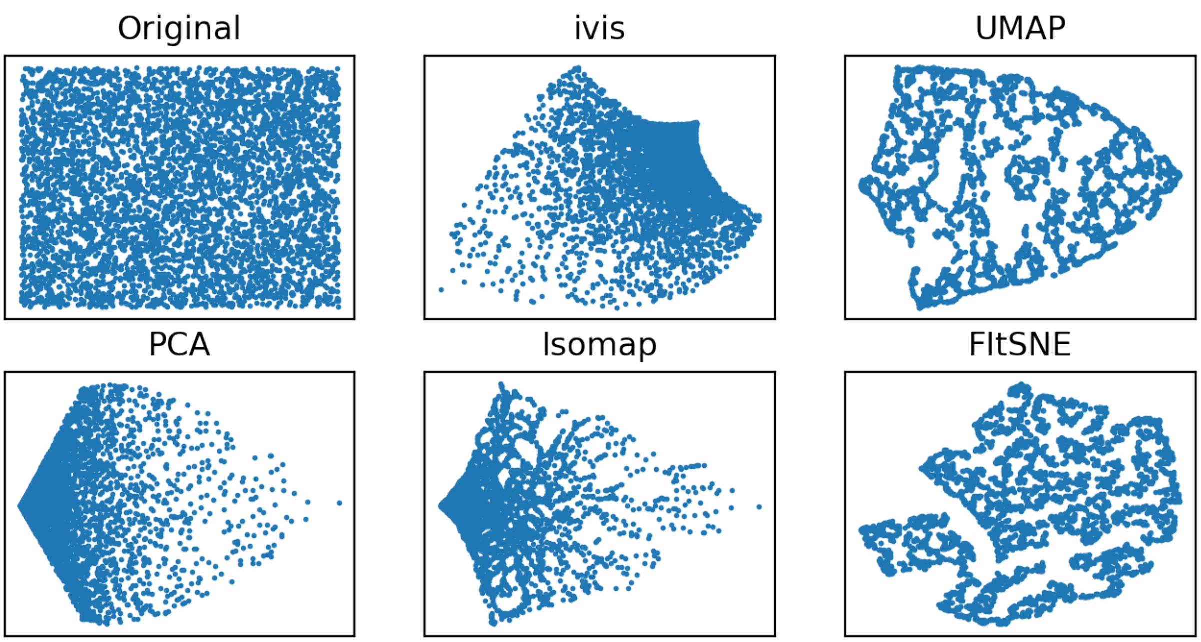
First, `ivis` does not make any assumptions as to the inherent structure of the dataset. Second, `ivis` is designed to handle both small and extremely large datasets. Ivis performs well on toy datasets such as the *iris* dataset, and scales linearly to datasets with millions of observations. Indeed, we see that the main usecase for `ivis` are datasets with > 250,000 observations. Finally, `ivis` prioritises interpretation over visual apperance - this is accomplished by imposing meaning to distances between points in the embedding space. As such, `ivis` does not create spurious clusters nor does it artificially pack clusters closer together. Embeddings aim to be true to the original structure of the data, which can be noisy in a real-world setting.

1.9.1 Visual Assessment

We will visually examine how popular dimensionality reduction algorithms - UMAP, t-SNE, Isomap, MDS, and PCA - approach two synthetic datasets with 5,000 observations in each. Since we are concerned with a dimensionality reduction problem, we will artificially add reduntant features to the original datasets using polynomial combinations (degree 10) of the original features.

Random Noise

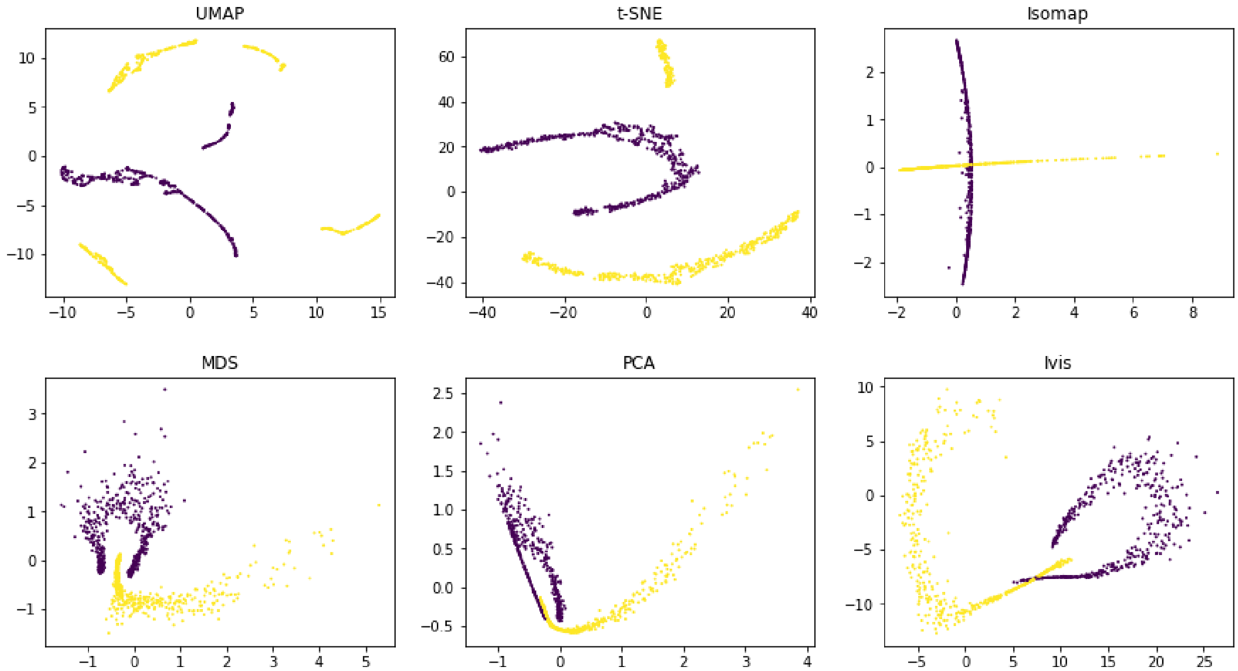
To start, let's examine how various dimensionality reduction methods behave in the presence of random noise. We generated 5000 uniformly distributed random points in a two-dimensional space and expanded the feature set using polynomial combinations. In all cases default parameters were used to fit multiple models.



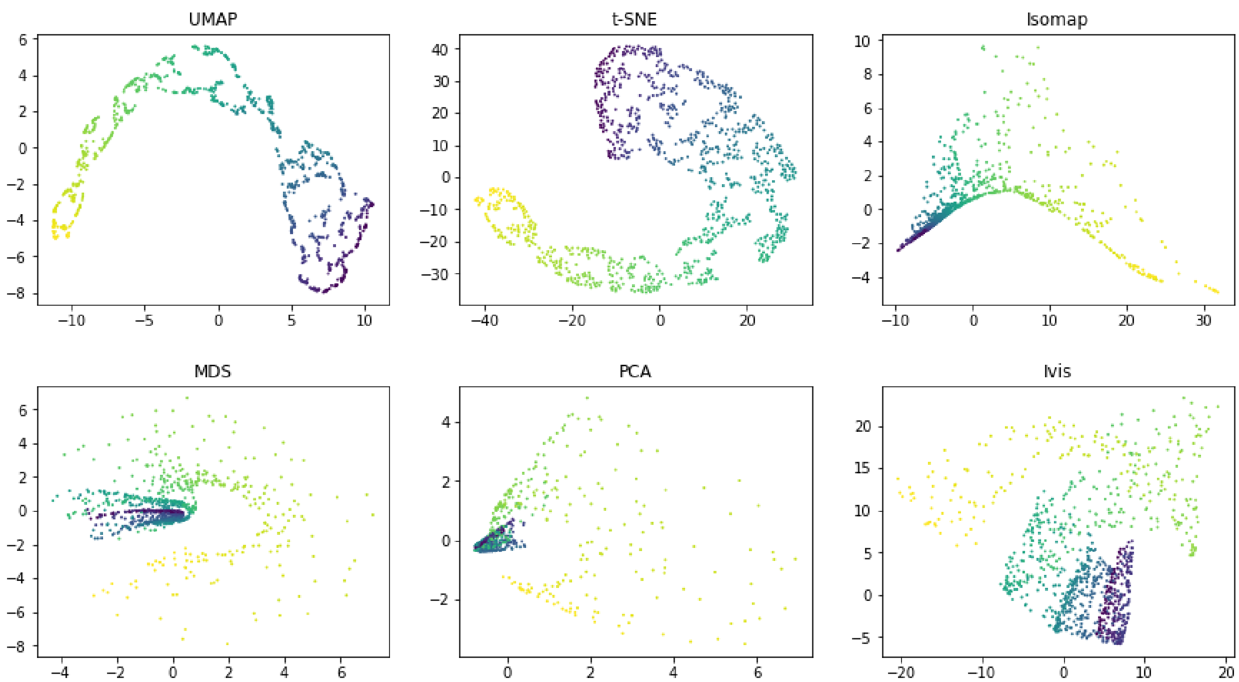
Both `ivis` and PCA reliably recovered the random nature of our dataset. Conversely, Isomap, UMAP, and t-SNE appeared to pack certain points together, creating an impression of clusters within uniform random noise.

Structured Datasets

Next, we examine how well global features of a dataset, such as relative position of clusters, can be recovered in a low-dimensional space.



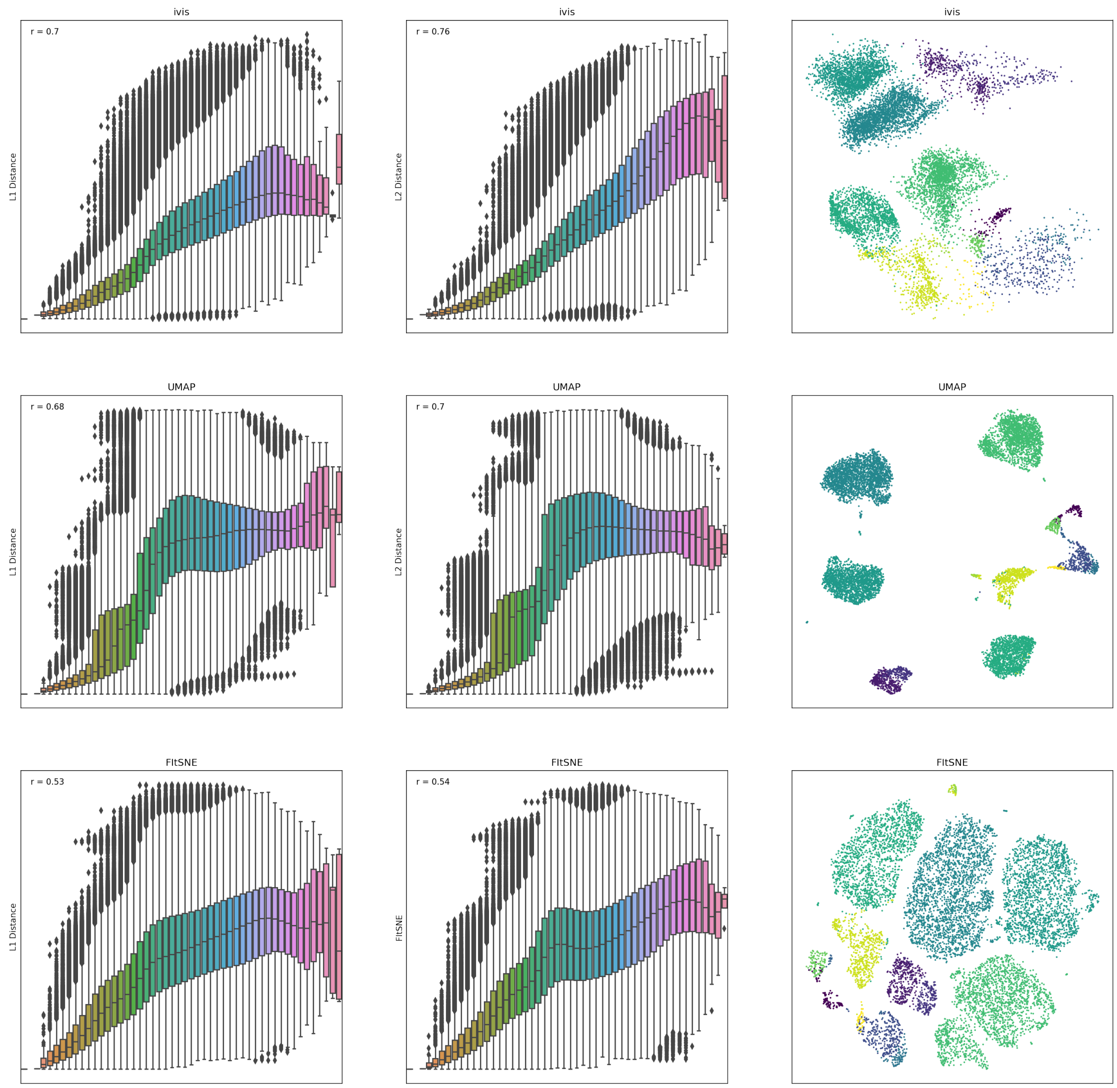
Using default parameters, we can see that `ivis` captures both the general structure of each half-moon, as well as their relative positions to one another. Both UMAP and t-SNE appear to introduce spurious clusters and global relationships between the half-moons appear to be disrupted.



Similarly as above, UMAP and t-SNE appear to generate a large number of small clusters along the continuous distribution of the dataset. Although the global structure is relatively well-preserved. `ivis` maintains both global and local structures of the dataset.

1.9.2 Quantitative Evaluation

To measure how well each algorithm preserves global distances, we examined correlation between points in the original dataset and the embedding space. For this analysis, 10,000 observations were chosen from the Levine dataset (104,184 x 32) using random uniform sampling. Box plots represent distances across pairs of points in the embeddings, binned using 50 equal-width bins over the pairwise distances in the original space. Pearson correlation coefficients were also computed over the pairs of distances.



`ivis` appeared to preserve both a small-, mid-, and large-scale L1 and L2 distances, whilst UMAP and t-SNE seemed to ignore mid- to large-scale distances. Interestingly, `ivis` was particularly good at preserving L2 distances in low-dimensional space.

1.10 Metric Learning with Application to Supervised Anomaly Detection

1.10.1 Introduction

Metric Learning

Metric Learning is a machine learning task that aims to learn a distance function over a set of observations. This can be useful in a number of applications, including clustering, face identification, and recommendation systems.

`ivis` was developed to address this task using concepts of the Siamese Neural Networks. In this example, we will demonstrate that Metric Learning using `ivis` can effectively deal with class imbalance, yielding features resulting in state-of-the-art classification performance.

Supervised Dimensionality Reduction

`ivis` is able to make use of any provided class labels to perform supervised dimensionality reduction. Supervised embeddings combine the distance-based characteristics of the unsupervised `ivis` algorithm with clear class boundaries between the class categories. This is achieved by simultaneously minimising the tripplet loss and softmax loss functions. The resulting embeddings encode relevant class-specific information into lower dimensional space. It is possible to control the relative importance `ivis` places on class labels when training in supervised mode with the `supervision_weight` parameter. This variable should be a float between 0.0 to 1.0, with higher values resulting in classification affecting the training process more, and smaller values resulting in it impacting the training less. By default, the parameter is set to 0.5. Increasing it to 0.8 will result in more cleanly separated classes.

1.10.2 Results

Data Selection

In this example we will make use of the Credit Card Fraud Dataset. The datasets contains transactions made by credit cards in September 2013 by european cardholders. This dataset presents transactions that occurred in two days, where we have 492 frauds out of 284,807 transactions. The dataset is highly unbalanced, the positive class (frauds) account for 0.172% of all transactions. Traditional supervised classification approaches would typically balance the training dataset either by over-sampling the minority class or down-sampling the majority class. Here, we investigate how `ivis` handles class embalance.

Data Preparation

```
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, average_precision_score, roc_auc_score,␣
↪classification_report
from sklearn.linear_model import LogisticRegression

from ivis import Ivis
```

```
data = pd.read_csv('../input/creditcard.csv')
Y = data['Class']
```

The Credit Card Fraud dataset is highly skewed, consisting of 492 frauds in a total of 284,807 observations (0.17% fraud cases). The features consist of numerical values from the 28 'Principal Component Analysis (PCA)' transformed features, as well as Time and Amount of a transaction.

In this analysis we will train `ivis` algorithm using a 5% stratified subsample of the dataset. Our previous experiments have shown that `ivis` can yield >90% accurate embeddings using just 1% of the total data.

```
train_X, test_X, train_Y, test_Y = train_test_split(data, Y, stratify=Y,
                                                    test_size=0.95, random_state=1234)
```

Next, because `ivis` will learn a distance over observations, scaling must be applied to features. Additionally, transforming the data to a range [0, 1] allows the neural network to extract more meaningful features.

```
standard_scaler = StandardScaler().fit(train_X[['Time', 'Amount']])
train_X.loc[:, ['Time', 'Amount']] = standard_scaler.transform(train_X[['Time',
↪'Amount']])
test_X.loc[:, ['Time', 'Amount']] = standard_scaler.transform(test_X[['Time', 'Amount
↪']])

minmax_scaler = MinMaxScaler().fit(train_X)
train_X = minmax_scaler.transform(train_X)
test_X = minmax_scaler.transform(test_X)
```

Dimensionality Reduction

Now, we can run `ivis` using default hyperparameters for supervised embedding problems:

```
ivis = Ivis(embedding_dims=2, model='maaten',
            k=15, n_epochs_without_progress=5,
            supervision_weight=0.80,
            verbose=0)
ivis.fit(train_X, train_Y.values)
```

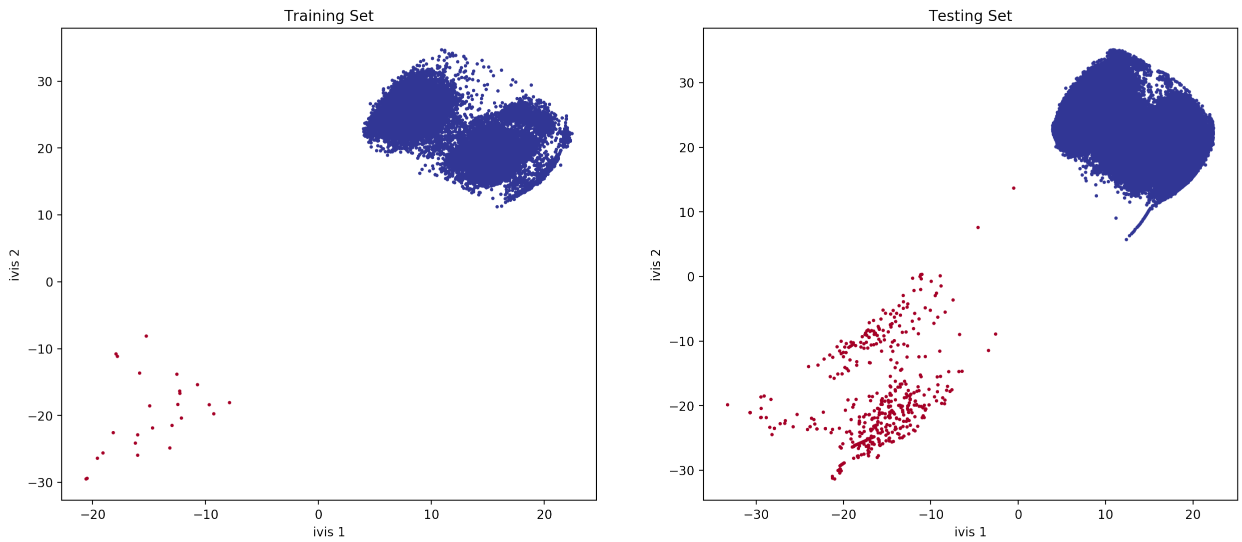
```
ivis.save_model('ivis-supervised-fraud')
```

Finally, let's embed the training set and extrapolate learnt embeddings to the testing set.

```
train_embeddings = ivis.transform(train_X)
test_embeddings = ivis.transform(test_X)
```

Visualisations

```
fig, ax = plt.subplots(1, 2, figsize=(17, 7), dpi=200)
ax[0].scatter(x=train_embeddings[:, 0], y=train_embeddings[:, 1], c=train_Y, s=3,␣
↪cmap='RdYlBu_r')
ax[0].set_xlabel('ivis 1')
ax[0].set_ylabel('ivis 2')
ax[0].set_title('Training Set')

ax[1].scatter(x=test_embeddings[:, 0], y=test_embeddings[:, 1], c=test_Y, s=3, cmap=
↪'RdYlBu_r')
ax[1].set_xlabel('ivis 1')
ax[1].set_ylabel('ivis 2')
ax[1].set_title('Testing Set')
```



With anomalies being shown in red, we can see that `ivis`:

1. Effectively learnt embeddings in an unbalanced dataset.
2. Succesfully extrapolated learnt metrics to a testing subset.

Linear Classifier

We can train a simple linear classifier to assess how well `ivis` learned the class representations.

```
clf = LogisticRegression(solver="lbfgs").fit(train_embeddings, train_Y)
```

```
labels = clf.predict(test_embeddings)
proba = clf.predict_proba(test_embeddings)
```

```
print(classification_report(test_Y, labels))

print('Confusion Matrix')
print(confusion_matrix(test_Y, labels))
print('Average Precision: '+str(average_precision_score(test_Y, proba[:, 1])))
print('ROC AUC: '+str(roc_auc_score(test_Y, labels)))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    270100
           1       1.00      0.99      1.00       467

    accuracy                           1.00    270567
   macro avg       1.00      1.00      1.00    270567
weighted avg       1.00      1.00      1.00    270567

Confusion Matrix
[[270100      0]
 [     3    464]]
Average Precision: 0.9978643591710002
ROC AUC: 0.9967880085653105
```

1.10.3 Conclusions

`ivis` effectively learns a distance metric over an unbalanced dataset. The resulting feature set can be used with a simple linear model classifier to achieve state-of-the-art performance on a classification task.

1.11 Training `ivis` on Out-of-memory Datasets

1.11.1 Introduction

Out-of-memory Datasets

Some datasets are so large that it becomes infeasible to load them into memory all at the same time. Other visualisation techniques might only be able to run on a smaller subset of the data; however, this runs the risk of potentially missing out on important smaller patterns in the data.

`ivis` was developed to address the issue of dimensionality reduction in very large datasets through batch-wise training of the neural network on data stored HDF5 format. Since training occurs in batches, the whole dataset does not need to be loaded into memory at once, and can instead be loaded from disk in chunks. In this example, we will show how `ivis` can scale up and be used to visualize massive datasets that don't fit into memory.

1.11.2 Example

Data Selection

In this example we will make use of the KDD Cup 1999 dataset. Although the dataset can be easily read-in to RAM, it provides a toy example for a general use case. The KDD99 dataset contains network traffic, with the competition task being to detect network intruders. The dataset is unbalanced, with the majority of traffic being normal.

Data Preparation

To train `ivis` on an out-of-memory dataset, the dataset must first be converted into the `h5` file format. There are numerous methods of doing this using various external tools such as Apache Spark. In this example, we will assume that the dataset has already been preprocessed and converted to .h5 format.

Dimensionality Reduction

To train on a h5 file that exists on disk you need the `h5py` package. A HDF5 dataset stored inside a file can be directly passed to an Ivis object's `fit` and `transform` methods. We will train `ivis` in unsupervised mode for 5 epochs to speed up training; other hyperparameters are left at their default values.

Note: When training on a h5 dataset, we recommend to use the `shuffle_mode='batch'` option in the `fit` method. This will speed up the training process by shuffling batches of data, rather than shuffling across the whole dataset.

```
import h5py

with h5py.File(h5_filepath, 'r') as f:
    X = f['data']
```

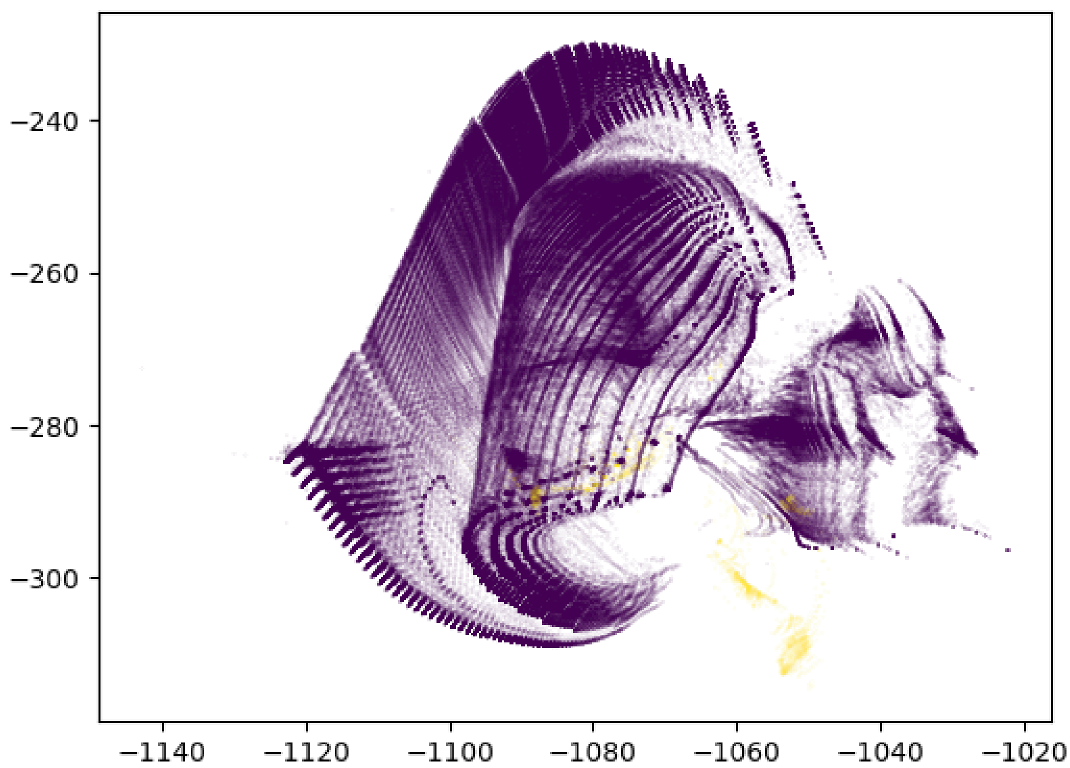
(continues on next page)

(continued from previous page)

```
    y = f['labels']

    model = Ivis(epochs=5)
    model.fit(X, shuffle_mode='batch') # Shuffle batches when using h5 files

    y_pred = model.transform(X)
```

Visualisations

```
plt.figure()
plt.scatter(x=y_pred[:, 0], y=y_pred[:, 1], c=y)
plt.set_xlabel('ivis 1')
plt.set_ylabel('ivis 2')
plt.show()
```



With anomalies being shown in yellow, we can see that `ivis` is able to pin point anomalous observations.

1.11.3 Conclusions

`ivis` is able to scale and deal with the massive, out-of-memory datasets found in the real world by training directly on h5 files. Additionally, it can effectively learn embeddings in an unbalanced dataset without labels.

1.11.4 Advanced: Custom dataset loaders

In addition to h5 files, ivis is also able to train on arbitrary out-of-memory datasets using custom classes that implement the *ivis.data.sequence.IndexableDataset* class methods. For example, the *ivis.data.sequence.ImageDataset* inherits from the *IndexableDataset* and reads in images from disk when indexed, allowing for training on image datasets of any size. The instance of *ImageDataset* is simply provided to ivis as the argument to the *fit* or *transform* methods.

By writing a custom class tailored to your specific scenario you can use ivis with whatever data storage you are using. For example, it's possible to have ivis train directly on data stored within a database with just a few lines of code.

1.12 Ivis Runtime Benchmarks

Real-world datasets are becoming increasingly complex, both due to the number of observations and the ever-growing feature space. For example, single cell experiments can easily monitor 20,000 features across 1,000,000 observations.

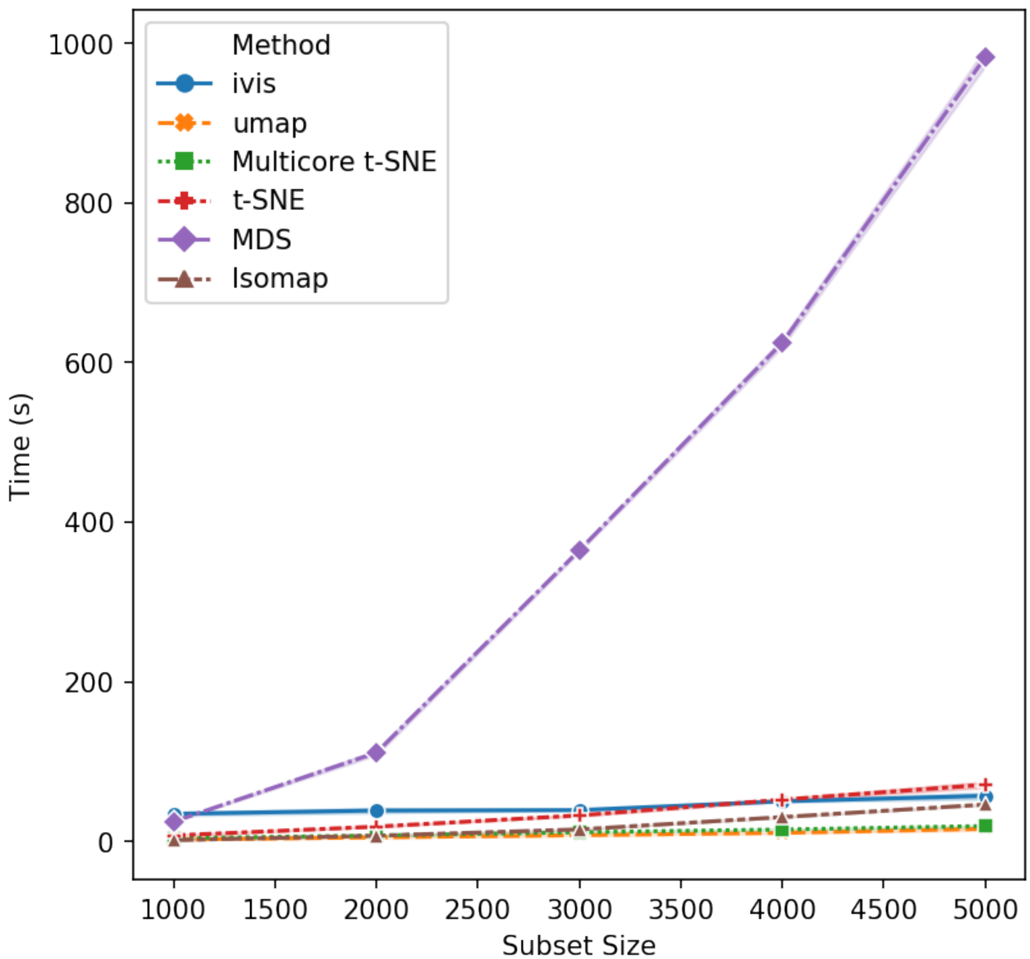
Dimensionality reduction (DR) algorithms enable useful exploration of feature-rich datasets. Nevertheless, each algorithm has different computational complexity that impacts its real-world use case. We will now investigate how runtime performance of the `ivis` algorithm scales with increasing dataset size.

Algorithm implementation has significant impact on performance. In these experiments, we will use mainly scikit-learn implementation, with the exception of multicore t-SNE. Two benchmark datasets will be used to asses runtimes: MNIST (up to 70,000 observations) and the first 1,000,000 integers represented as binary vectors indicating their prime factors. For all algorithms, default settings were adopted. Ivis hyperparameters were fixed to: `embedding_dims=2`, `k=15`, `model='szubert'`, and `n_epochs_without_progress=3`. Our previous experiments have shown that these defaults yield accurate embeddings.

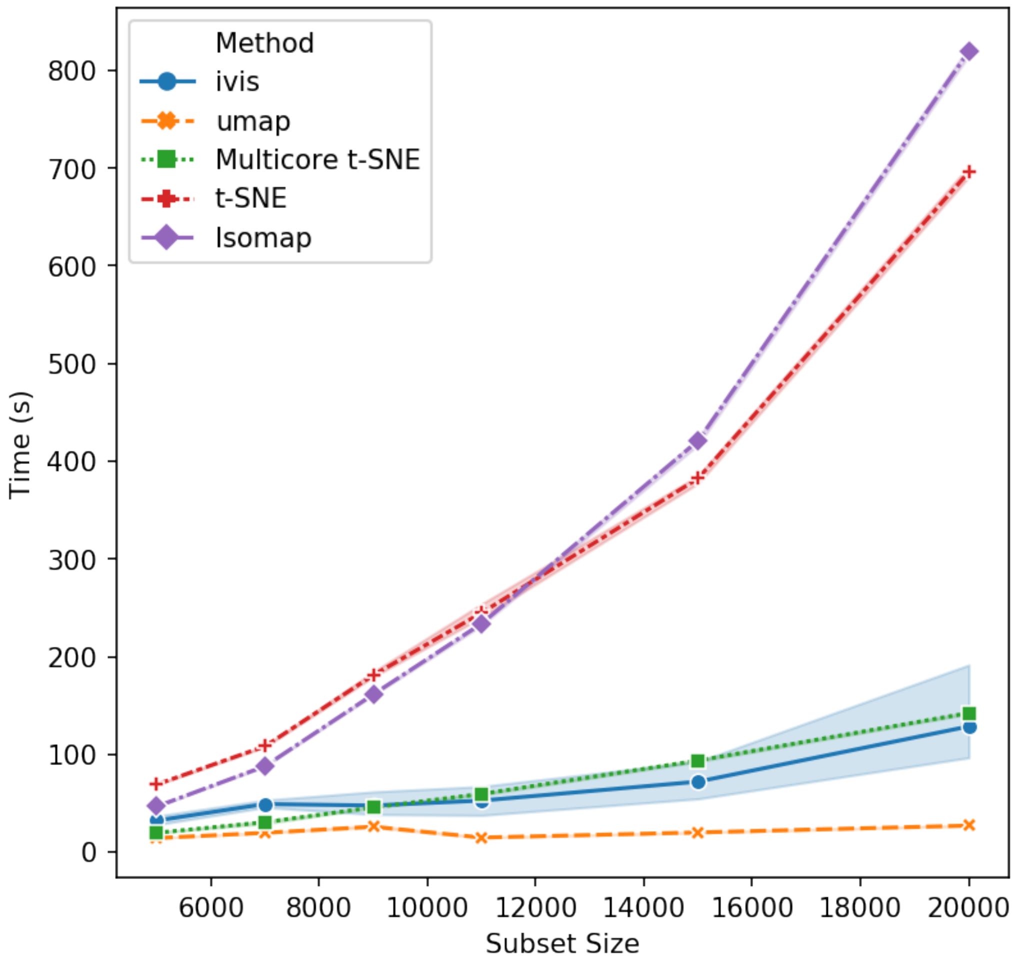
Subsamples were created using scikit-learn's `resample` method, producing stratified random sub-samples. For each run, three random subsamples were generated to create a distribution of values. All runs were carried out on a 16-core machine with 32GB of RAM.

1.12.1 Effects of Data Size on Performance

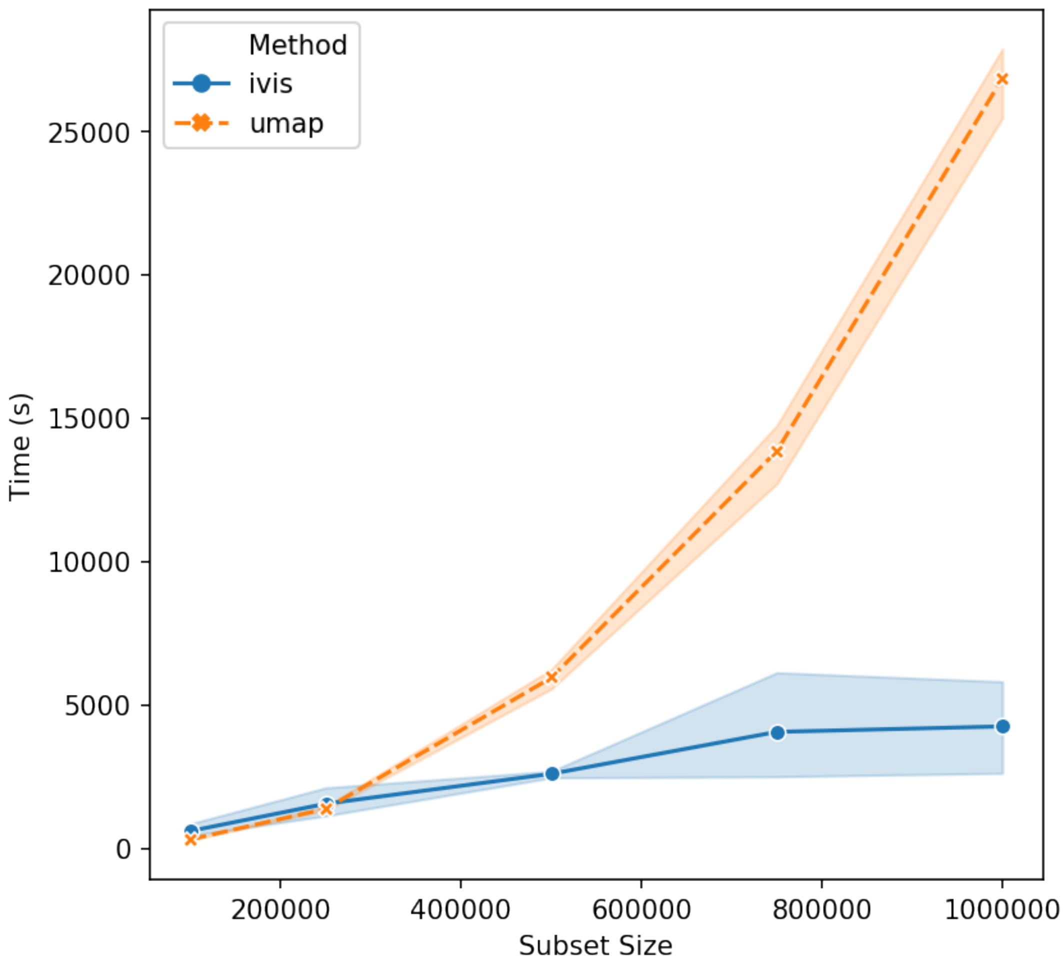
We begin with small subsample sizes – 1,000 to 5,000 observations. It becomes clear that MDS will not be usable as we increase subsample sizes. Additionally, scikit-learn's implementation of t-SNE is beginning to slow down as we approach 5,000 subsamples. UMAP and multicore t-SNE perform very well.



That's a reasonable start - let's increase the subset size. Isomap and scikit-learn's t-SNE seem to have reached their performance threshold and are now experiencing considerable slow down. `ivis` appears to be on-par with multicore t-SNE, albeit a little faster, whilst UMAP is the winner hands down!



Now, let's push beyond toy datasets and examine sizes that are more likely to be encountered in real-world problems. For this experiment we generated 1,000,000 integers (observations) with corresponding binary vectors indicating their prime factors (features). We immediately see that `ivis` is fast. Additionally, whilst UMAP timings increase exponentially, `ivis` execution speed does not change much on subsamples with greater than 750,000 observations.



We can conclude that for smaller datasets (< 100,000 observations), UMAP and multicore t-SNE are excellent options. However, `ivis` excels at dealing with very large datasets. Furthermore, `ivis` appears to *generate more accurate embeddings* – a perk that comes with a slightly longer runtime for smaller datasets.

1.13 Distance Preservation Benchmarks

Dimensionality reduction is crucial for effective manipulation of high-dimensional datasets. However, low-dimensional representations often fail to capture complex global and local relationships in many real-world datasets. Here, we assess how well `ivis` preserves inter-cluster distances in two well-characterised datasets and benchmark performance across several linear and non-linear dimensinality reduction approaches.

1.13.1 Datasets Selection

Two benchmark datasets were used - MNIST database of handwritten digits (70,000 observations, 784 features) and Levine dataset (104,184 observations, 32 features). The Levine dataset was obtained from Data-Driven Phenotypic Dissection of AML Reveals Progenitor-like Cells that Correlate with Prognosis. The 32-dimensional Levine dataset can be downloaded directly from Cytobank.

Both datasets have target `Y` variables. For MNIST, targets take on values [0, 9] and represent hand-written digits, whilst in the Levine dataset targets are manually annotated cell populations [0-13]. Prior to preprocessing, values in both datasets were scaled to [0, 1] range.

- MNIST preprocessing:

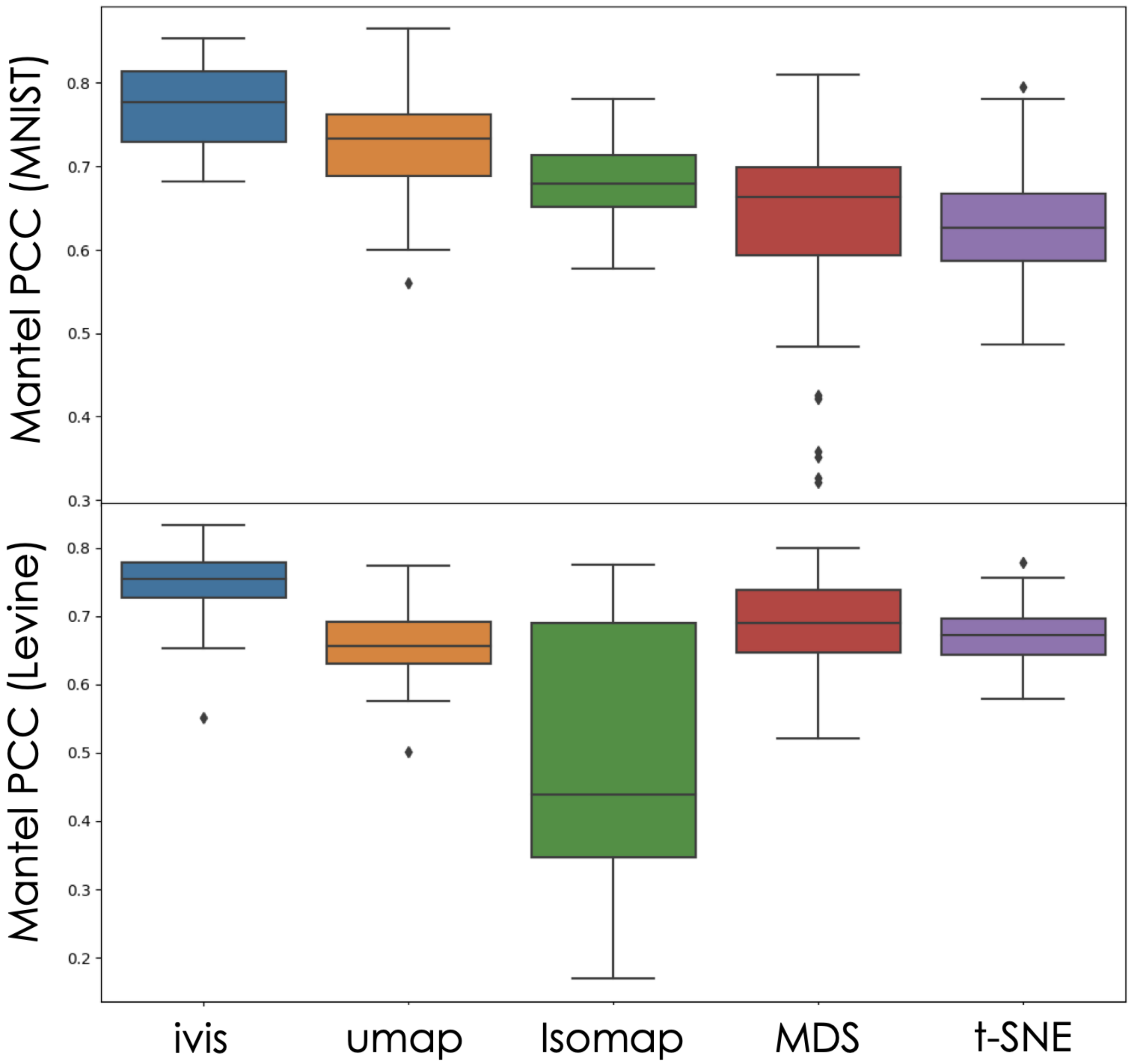
```
from sklearn.datasets import fetch_openml
from sklearn.preprocessing import MinMaxScaler
X, Y = fetch_openml('mnist_784', version=1, return_X_y=True)
X = MinMaxScaler().fit_transform(X)
```

- Levine preprocessing:

```
import pandas as pd
from sklearn.preprocessing import LabelEncoder, MinMaxScaler

data = pd.read_csv('../data/levine_32dm_notransform.txt')
data = data.dropna()

features = ['CD45RA', 'CD133', 'CD19', 'CD22', 'CD11b', 'CD4', 'CD8',
            'CD34', 'Flt3', 'CD20', 'CXCR4', 'CD235ab', 'CD45', 'CD123',
            'CD321', 'CD14', 'CD33', 'CD47', 'CD11c', 'CD7', 'CD15', 'CD16',
            'CD44', 'CD38', 'CD13', 'CD3', 'CD61', 'CD117', 'CD49d',
            'HLA-DR', 'CD64', 'CD41', 'label']
data = data[features]

X = data.drop(['label'], axis=1).values
X = np.arcsinh(X/5)
X = MinMaxScaler().fit_transform(X)
```

1.13.2 Accuracy of Low-Dimensional Embeddings

To establish how well `ivis` and other dimensionality reduction techniques preserve data structure in low-dimensional space, a Euclidean distance matrix between centroids of the target values in Levine and MNIST datasets was created for the original datasets, respective `ivis` embeddings, as well as UMAP, t-SNE, MDS, and Isomap embeddings. The level of correlation between the original distance matrix and the distance matrices in the embedding spaces was then assessed using the Mantel test. Pearson's product-moment correlation coefficient (PCC) was used to quantitate concordance between original data and low-dimensional representations. Random stratified subsamples (n=50) of 1000 observations were used to generate a continuum of PCC values for each embedding technique. For all `ivis` runs, only two hyperparameters were set: `k=15` and `model="maaten"`. These are recommended defaults for datasets with <500,000 observations. For other dimensionality reduction methods, default parameters were used.



The Mantel Test measures correlation between two distance matrices - embedding space and original space Euclidean distances of cluster centroids. From our experiment, we can conclude that `ivis` preserves inter-cluster distances well, with average PCC being ~0.75 in the MNIST and Levine datasets. Importantly, `ivis` outperformes other dimensionality reduction techniques.

1.14 Ivis

class ivis.**Ivis**(*embedding_dims=2*, ***, *k=150*, *distance='pn'*, *batch_size=128*, *epochs=1000*, *n_epochs_without_progress=20*, *n_trees=50*, *ntrees=None*, *knn_distance_metric='angular'*, *search_k=-1*, *precompute=True*, *model='szubert'*, *supervision_metric='sparse_categorical_crossentropy'*, *supervision_weight=0.5*, *annoy_index_path=None*, *callbacks=None*, *build_index_on_disk=True*, *neighbour_matrix=None*, *verbose=1*)

Bases: `sklearn.base.BaseEstimator`, `sklearn.base.TransformerMixin`

Ivis is a technique that uses an artificial neural network for dimensionality reduction, often useful for the purposes of visualization. The network trains on triplets of data-points at a time and pulls positive points together, while pushing more distant points away from each other. Triplets are sampled from the original data using KNN approximation using the Annoy library.

Parameters

- **embedding_dims** (*int*) – Number of dimensions in the embedding space
- **k** (*int*) – The number of neighbours to retrieve for each point. Must be less than one minus the number of rows in the dataset.
- **distance** (*Union[str,Callable]*) – The loss function used to train the neural network.
  - If string: a registered loss function name. Predefined losses are: "pn", "euclidean", "manhattan_pn", "manhattan", "chebyshev", "chebyshev_pn", "softmax_ratio_pn", "softmax_ratio", "cosine", "cosine_pn".
  - If Callable, must have two parameters, (y_true, y_pred). y_pred denotes the batch of triplets, and y_true are any corresponding labels. y_pred is expected to be of shape: (3, batch_size, embedding_dims).
    - When loading model loaded with a custom loss, provide the loss to the constructor of the new Ivis instance before loading the saved model.
- **batch_size** (*int*) – The size of mini-batches used during gradient descent while training the neural network. Must be less than the num_rows in the dataset.
- **epochs** (*int*) – The maximum number of epochs to train the model for. Each epoch the network will see a triplet based on each data-point once.
- **n_epochs_without_progress** (*int*) – After n number of epochs without an improvement to the loss, terminate training early.
- **n_trees** (*int*) – The number of random projections trees built by Annoy to approximate KNN. The more trees the higher the memory usage, but the better the accuracy of results.
- **ntrees** (*int*) – Deprecated. Use *n_trees* instead.
- **knn_distance_metric** (*str*) – The distance metric used to retrieve nearest neighbours. Supports "angular" (default), "euclidean", "manhattan", "hamming", or "dot".
- **search_k** (*int*) – The maximum number of nodes inspected during a nearest neighbour query by Annoy. The higher, the more computation time required, but the higher the accuracy. The default is n_trees * k, where k is the number of neighbours to retrieve. If this is set too low, a variable number of neighbours may be retrieved per data-point.
- **precompute** (*bool*) – Whether to pre-compute the nearest neighbours. Pre-computing is a little faster, but requires more memory. If memory is limited, try setting this to False.
- **model** (*Union[str,tf.keras.models.Model]*) – The keras model to train using triplet loss.
  - If a model object is provided, an embedding layer of size 'embedding_dims' will be appended to the end of the network.
  - If a string, a pre-defined network by that name will be used. Possible options are: 'szubert', 'hinton', 'maaten'. By default the 'szubert' network will be created, which is a selu network composed of 3 dense layers of 128 neurons each, followed by an embedding layer of size 'embedding_dims'.
- **supervision_metric** (*str*) – The supervision metric to optimize when training keras in supervised mode. Supports all of the classification or regression losses included with

keras, so long as the labels are provided in the correct format. A list of keras' loss functions can be found at https://keras.io/losses/ .

- **supervision_weight** (*float*) – Float between 0 and 1 denoting the weighting to give to classification vs triplet loss when training in supervised mode. The higher the weight, the more classification influences training. Ignored if using Ivis in unsupervised mode.
- **annoy_index_path** (*str*) – The filepath of a pre-trained annoy index file saved on disk. If provided, the annoy index file will be loaded and used. Otherwise, a new index will be generated and saved to disk in a temporary directory.
- **callbacks** (*[keras.callbacks.Callback]*) – List of keras Callbacks to pass model during training, such as the TensorBoard callback. A set of ivis-specific callbacks are provided in the ivis.nn.callbacks module.
- **build_index_on_disk** (*bool*) – Whether to build the annoy index directly on disk. Building on disk should allow for bigger datasets to be indexed, but may cause issues.
- **neighbour_matrix** (*Union[np.array,collections.abc.Sequence]*) – Providing a neighbour matrix will cause Ivis to skip computing the Annoy KNN index and instead use the provided neighbour_matrix.
  - A pre-computed neighbour matrix can be provided as a numpy array. Indexing the array should retrieve a list of neighbours for the data point associated with that index.
  - Alternatively, dynamic computation of neighbours can be done by providing a class than implements the collections.abc.Sequence class, specifically the *__getitem__* and *__len__* methods.
    * See the ivis.data.neighbour_retrieval.AnnoyKnnMatrix class for an example.
- **verbose** (*int*) – Controls the volume of logging output the model produces when training. When set to 0, silences outputs, when above 0 will print outputs.

fit (*X*, *Y=None*, *shuffle_mode=True*)

Fit an ivis model.

Parameters

- **X** (*np.array*, *ivis.data.sequence.IndexableDataset*, *tensorflow.keras.utils.HDF5Matrix*) – Data to be embedded. Needs to have a *.shape* attribute and a *__getitem__* method.
- **Y** (*array, shape (n_samples)*) – Optional array for supervised dimensionality reduction. If Y contains -1 labels, and 'sparse_categorical_crossentropy' is the loss function, semi-supervised learning will be used.

Returns **self** – Returns estimator instance.

Return type ivis.Ivis object

fit_transform (*X*, *Y=None*, *shuffle_mode=True*)

Fit to data then transform

Parameters

- **X** (*np.array*, *ivis.data.sequence.IndexableDataset*, *tensorflow.keras.utils.HDF5Matrix*) – Data to train on and then embedded. Needs to have a *.shape* attribute and a *__getitem__* method.
- **Y** (*array, shape (n_samples)*) – Optional array for supervised dimensionality reduction. If Y contains -1 labels, and 'sparse_categorical_crossentropy' is the loss function, semi-supervised learning will be used.

Returns **X_new** – Embedding of the data in low-dimensional space.

Return type array, shape (n_samples, embedding_dims)

get_params(*deep=True*)

Get parameters for this estimator.

Parameters **deep** (*bool, default=True*) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns **params** – Parameter names mapped to their values.

Return type dict

load_model(*folder_path*)

Load ivis model

Parameters **folder_path** (*string*) – Path to serialised model files and metadata

Returns **self** – Returns estimator instance.

Return type ivis.Ivis object

save_model(*folder_path*, *save_format='h5'*, *overwrite=False*)

Save an ivis model

Parameters

- **folder_path** (*string*) – Path to serialised model files and metadata
- **save_format** (*string*) – Format to save ivis model as. Either ".h5" for a .h5 file or "tf" for TensorFlow SavedModel format.
- **overwrite** (*bool*) – Whether to overwrite the specified folder path.

score_samples(*X*)

Passes X through classification network to obtain predicted supervised values. Only applicable when trained in supervised mode.

Parameters **X** (*np.array, ivis.data.sequence.IndexableDataset, tensorflow.keras.utils.HDF5Matrix*) – Data to be passed through classification network. Needs to have a *.shape* attribute and a *__getitem__* method.

Returns **X_new** – Softmax class probabilities of the data.

Return type array, shape (n_samples, embedding_dims)

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Parameters ****params** (*dict*) – Estimator parameters.

Returns **self** – Estimator instance.

Return type estimator instance

transform(*X*)

Transform X into the existing embedded space and return that transformed output.

Parameters **X** (*np.array, ivis.data.sequence.IndexableDataset, tensorflow.keras.utils.HDF5Matrix*) – Data to be transformed. Needs to have a *.shape* attribute and a *__getitem__* method.

Returns **X_new** – Embedding of the data in low-dimensional space.

Return type array, shape (n_samples, embedding_dims)

1.15 Neighbour Retrieval

class `ivis.data.neighbour_retrieval.`**`NeighbourMatrix`**

Bases: `collections.abc.Sequence`

A matrix A_{ij} where i is the row index of the data point and j refers to the index of the neigbouring point.

`get_batch`(*idx_seq*)

Gets a batch of neighbours corresponding to the provided index sequence.

Non-optimized version, can be overridden by child classes to be made be efficient

`k`

The width of the matrix (number of neighbours retrieved)

class `ivis.data.neighbour_retrieval.`**`AnnoyKnnMatrix`**(*index*, *nrows*, *index_path='annoy.index'*, *metric='angular'*, *k=150*, *search_k=-1*, *precompute=False*, *include_distances=False*, *verbose=False*, *n_jobs=-1*)

Bases: `ivis.data.neighbour_retrieval.knn.NeighbourMatrix`

A matrix A_{ij} where i is the row index of the data point and j refers to the index of the neigbouring point.

Neighbouring points are KNN retrieved using an Annoy Index.

Parameters

- **index** (*AnnoyIndex*) – AnnoyIndex instance to use when retrieving KNN
- **nrows** (*tuple*) – Number of rows in data matrix was built on
- **index_path** (*string*) – Location of the AnnoyIndex file on disk
- **k** (*int*) – The number of neighbours to retrieve for each point
- **search_k** (*int*) – Controls the number of nodes searched - higher is more expensive but more accurate. Default of -1 defaults to n_trees * k
- **precompute** (*boolean*) – Whether to precompute the KNN index and store the matrix in memory. Much faster when training, but consumes more memory.
- **include_distances** (*boolean*) – Whether to return the distances along with the indexes of the neighbouring points
- **verbose** (*boolean*) – Controls verbosity of output to console when building index. If False, nothing will be printed to the terminal.

`__getitem__`(*idx*)

Returns neighbours list for the specified index. Supports both integer and slice indices.

`__getstate__`()

Return object serializable variable dict

`__init__`(*index*, *nrows*, *index_path='annoy.index'*, *metric='angular'*, *k=150*, *search_k=-1*, *precompute=False*, *include_distances=False*, *verbose=False*, *n_jobs=-1*)

Constructs an AnnoyKnnMatrix instance from an AnnoyIndex object with given parameters

__len__()
Number of rows in neighbour matrix

classmethod build(*X*, *path*, *k=150*, *metric='angular'*, *search_k=-1*, *include_distances=False*, *ntrees=50*, *build_index_on_disk=True*, *precompute=False*, *verbose=1*, *n_jobs=-1*)
Builds a new Annoy Index on input data *X*, then constructs and returns an AnnoyKnnMatrix object using the newly-built index.

delete_index(*parent=False*)
Cleans up disk resources used by the index, rendering it unusable. First will *unload* the index, then recursively removes the files at index path. If parent is True, will recursively remove parent folder.

get_batch(*idx_seq*)
Returns a batch of neighbours based on the index sequence provided.

get_neighbour_indices(*n_jobs=-1*)
Retrieves neighbours for every row in parallel

classmethod load(*index_path*, *data_shape*, *k=150*, *metric='angular'*, *search_k=-1*, *include_distances=False*, *precompute=False*, *verbose=1*, *n_jobs=-1*)
Constructs and returns an AnnoyKnnMatrix object from an existing Annoy Index on disk.

save(*path*)
Saves internal Annoy index to disk at given path.

unload()
Unloads the index from disk, allowing other processes to read/write to the index file. After calling this, the index will no longer be usable from this instance.

class `ivis.data.neighbour_retrieval.`**LabeledNeighbourMap**(*labels*)
Bases: `collections.abc.Sequence`

Retrieves neighbour indices according to class labels provided in constructor. Rows with the same label will be regarded as neighbours.

__getitem__(*idx*)
Retrieves the neighbours for the row index provided

__init__(*labels*)
Constructs a LabeledNeighbourMap instance from a list of labels. :param labels list: List of labels for each data-point. One label per data-point.

__len__()
Returns the number of rows in the data

`ivis.data.neighbour_retrieval.knn.`**build_annoy_index**(*X*, *path*, *metric='angular'*, *ntrees=50*, *build_index_on_disk=True*, *verbose=1*, *n_jobs=-1*)
Build a standalone annoy index.

Parameters

- **X** (*array*) – numpy array with shape (n_samples, n_features)
- **path** (*str*) – The filepath of a trained annoy index file saved on disk.
- **ntrees** (*int*) – The number of random projections trees built by Annoy to approximate KNN. The more trees the higher the memory usage, but the better the accuracy of results.
- **build_index_on_disk** (*bool*) – Whether to build the annoy index directly on disk. Building on disk should allow for bigger datasets to be indexed, but may cause issues.

- **metric** (*str*) – Which distance metric Annoy should use when building KNN index. Supports "angular", "euclidean", "manhattan", "hamming", or "dot".
- **verbose** (*int*) – Controls the volume of logging output the model produces when training. When set to 0, silences outputs, when above 0 will print outputs.

1.16 Indexable Datasets

class ivis.data.sequence.**IndexableDataset**

Bases: collections.abc.Sequence

A sequence that also defines a shape attribute. This indexable data structure can be provided as input to ivis.

get_batch(*idx_seq*)

Returns a batch of data points based on the index sequence provided.

Non-optimized version, can be overridden by child classes to be made be efficient

shape()

Returns the shape of the dataset. First dimension corresponds to rows, the other dimensions correspond to features.

class ivis.data.sequence.**ImageDataset**(*filepath_list*, *img_shape*, *color_mode='rgb'*, *resize_method='bilinear'*, *preserve_aspect_ratio=False*, *dtype=<sphinx.ext.autodoc.importer._MockObject object>*, *preprocessing_function=None*, *n_jobs=-1*)

Bases: ivis.data.sequence.sequence.IndexableDataset

When indexed, loads images from disk, resizes to consistent size, then returns image. Since the returned images will consist of 3 dimensions, the model ivis uses must be capable of dealing with this dimensionality of data (for example, a Convolutional Neural Network). Such a model can be constructed externally and then passed to ivis as the argument for 'model'.

Parameters

- **filepath_list** (*list*) – All image filepaths in dataset.
- **img_shape** (*tuple*) – A tuple (height, width) containing desired dimensions to resize the images to.
- **str** (*resize_method*) – Either "rgb", "rgba" or "grayscale". Determines how many channels present in images that are read in - 3, 4, or 1 respectively.
- **str** – Interpolation method to use when resizing image. Must be one of: "area", "bicubic", "bilinear", "gaussian", "lanczos3", "lanczos5", "mitchellcubic", "nearest".
- **boolean** (*preserve_aspect_ratio*) – Whether to preserve the aspect ratio when resizing images. If True, will maintain aspect ratio by padding the image.
- **tf.dtypes.DType** (*dtype*) – The dtype to read the image into. One of tf.uint8 or tf.uint16.
- **Callable** (*preprocessing_function*) – A function to apply to every image. Will be called at the end of the pipeline, after image reading and resizing. If None (default), no function will be applied.

__init__(*filepath_list*, *img_shape*, *color_mode='rgb'*, *resize_method='bilinear'*, *preserve_aspect_ratio=False*, *dtype=<sphinx.ext.autodoc.importer._MockObject object>*, *preprocessing_function=None*, *n_jobs=-1*)

Initialize self. See help(type(self)) for accurate signature.

get_batch (*idx_seq*)
Returns a batch of data points based on the index sequence provided.

read_image (*filepath*)
Reads an image from disk into a numpy array

resize_image (*img*)
Resizes an numpy array image to desired dimensions

class ivis.data.sequence.**FlattenedImageDataset** (*filepath_list, img_shape, color_mode='rgb', resize_method='bilinear', preserve_aspect_ratio=False, dtype=<sphinx.ext.autodoc.importer._MockObject object>, preprocessing_function=None, n_jobs=None*)
Bases: `ivis.data.sequence.image.ImageDataset`

Returns flattened versions of images read in from disk. This dataset can be used with the default neighbour retrieval method used by ivis (Annoy KNN index) since it is 2D.

__init__ (*filepath_list, img_shape, color_mode='rgb', resize_method='bilinear', preserve_aspect_ratio=False, dtype=<sphinx.ext.autodoc.importer._MockObject object>, preprocessing_function=None, n_jobs=None*)
Initialize self. See help(type(self)) for accurate signature.

1.17 Losses

Triplet loss functions for training a siamese network with three subnetworks. All loss function variants are accessible through the *triplet_loss* function by specifying the distance as a string.

class ivis.nn.losses.**ChebyshevPnLoss** (*margin=1, name=None*)
Calculates the pn loss (a variant of triplet loss) between anchor, positive and negative examples in a triplet based on chebyshev distance.

class ivis.nn.losses.**ChebyshevTripletLoss** (*margin=1, name=None*)
Calculates the standard triplet loss between anchor, positive and negative examples in a triplet based on chebyshev distance.

class ivis.nn.losses.**CosinePnLoss** (*margin=1, name=None*)
Calculates the pn loss (a variant of triplet loss) between anchor, positive and negative examples in a triplet based on cosine distance.

class ivis.nn.losses.**CosineTripletLoss** (*margin=1, name=None*)
Calculates the standard triplet loss between anchor, positive and negative examples in a triplet based on cosine distance.

class ivis.nn.losses.**EuclideanPnLoss** (*margin=1, name=None*)
Calculates the pn loss (a variant of triplet loss) between anchor, positive and negative examples in a triplet based on euclidean distance.

class ivis.nn.losses.**EuclideanSoftmaxRatioLoss** (*name=None*)
Calculates the standard softmax ratio between anchor, positive and negative examples in a triplet based on euclidean distance.

class ivis.nn.losses.**EuclideanSoftmaxRatioPnLoss** (*name=None*)
Calculates a pn variant of the softmax ratio between anchor, positive and negative examples in a triplet based on euclidean distance.

class ivis.nn.losses.**EuclideanTripletLoss**(*margin=1*, *name=None*)
Calculates the standard triplet loss between anchor, positive and negative examples in a triplet based on euclidean distance.

class ivis.nn.losses.**ManhattanPnLoss**(*margin=1*, *name=None*)
Calculates the pn loss (a variant of triplet loss) between anchor, positive and negative examples in a triplet based on manhattan distance.

class ivis.nn.losses.**ManhattanTripletLoss**(*margin=1*, *name=None*)
Calculates the standard triplet loss between anchor, positive and negative examples in a triplet based on manhattan distance.

ivis.nn.losses.**register_loss**(*loss_fn=None*, ***, *name=None*)
Registers a class definition or Callable as an ivis loss function. A mapping will be created between the name and the loss function passed. If a class definition is provided, an instance will be created, passing the name as an argument.

If no name is provided to this function, the name of the passed function will be used as a key.

The loss function must have two parameters, (y_true, y_pred) and calculates the loss for a batch of triplet inputs (y_pred). y_pred is expected to be of shape: (3, batch_size, embedding_dims).

Usage:

```
@register_loss
def custom_loss(y_true, y_pred):
    pass
model = Ivis(distance='custom_loss')
```

ivis.nn.losses.**semi_supervised_loss**(*loss_function*)
Wraps the provided ivis supervised loss function to deal with the partially labeled data. Returns a new 'semi-supervised' loss function that masks the loss on examples where label information is missing.

Missing labels are assumed to be marked with -1.

ivis.nn.losses.**triplet_loss**(*distance='pn'*)
Returns a previously registered triplet loss function associated with the string 'distance'. If passed a callable, just returns it.

1.18 Callbacks

A collection of callbacks that can be passed to ivis to be called during training. These provide utilities such as saving checkpoints during training (allowing for resuming if interrupted), as well as periodic logging of plots and model embeddings. With this information, you may decide to terminate a training session early due to a lack of improvements to the visualizations, for example.

To use a callback during training, simply pass a list of callback objects to the Ivis object when creating it using the callbacks keyword argument. The ivis.nn.callbacks module contains a set of callbacks provided for use with ivis models, but any *tf.keras.callbacks.Callbacks* object can be passed and will be used during training: for example, *tf.keras.callbacks.TensorBoard*. This means it's also possible to write your own callbacks for ivis to use.

class ivis.nn.callbacks.**EmbeddingsImage**(*data*, *labels=None*, *log_dir='./logs'*, *filename='{}_embeddings.png'*, *epoch_interval=1*)
Bases: `sphinx.ext.autodoc.importer._MockObject`

Periodically generates and plots 2D embeddings of the data provided to `data` using the latest state of the `Ivis` model. By default, saves plots of the embeddings every epoch; increasing the `epoch_interval` will save the plots less frequently.

Parameters

- **data** (*list[float]*) – Data to embed and plot with the latest Ivis model
- **labels** (*list[int]*) – Labels with which to colour plotted embeddings. If *None* all points will have the same color.
- **log_dir** (*str*) – Folder to save resulting embeddings.
- **filename** (*str*) – Filename to save each file as. *{}* in string will be substituted with the epoch number.

Example usage:

```
from ivis.nn.callbacks import EmbeddingsImage
from ivis import Ivis
from tensorflow.keras.datasets import mnsit

(X_train, Y_train), (X_test, Y_test)  = mnist.load_data()

# Plot embeddings of test set every epoch colored by labels
embeddings_callback = EmbeddingsImage(X_test, Y_test,
                                        log_dir='test-embeddings',
                                        filename='{}_test_embeddings.npy',
                                        epoch_interval=1)

model = Ivis(callbacks=[embeddings_callback])

# Train on training set
model.fit(X_train)
```

class ivis.nn.callbacks.**EmbeddingsLogging**(*data*, *log_dir='./embeddings_logs'*, *filename='{}_embeddings.npy'*, *epoch_interval=1*)

Bases: `sphinx.ext.autodoc.importer._MockObject`

Periodically saves embeddings of the data provided to `data` using the latest state of the `Ivis` model. By default, saves embeddings every epoch; increasing the `epoch_interval` will save the embeddings less frequently.

Parameters

- **data** (*list[float]*) – Data to embed with the latest Ivis object
- **log_dir** (*str*) – Folder to save resulting embeddings.
- **filename** (*str*) – Filename to save each file as. *{}* in string will be substituted with the epoch number.

Example usage:

```
from ivis.nn.callbacks import EmbeddingsLogging
from ivis import Ivis
from tensorflow.keras.datasets import mnsit

(X_train, Y_train), (X_test, Y_test)  = mnist.load_data()

# Save embeddings of test set every epoch
embeddings_callback = EmbeddingsLogging(X_test,
                                        log_dir='test-embeddings',
                                        filename='{}_test_embeddings.npy',
```

(continues on next page)

(continued from previous page)

```
                                              epoch_interval=1)

model = Ivis(callbacks=[embeddings_callback])

# Train on training set
model.fit(X_train)
```

class ivis.nn.callbacks.**ModelCheckpoint**(*log_dir='./model_checkpoints', filename='model-checkpoint_{}.ivis', epoch_interval=1*)

Bases: `sphinx.ext.autodoc.importer._MockObject`

Periodically saves the model during training. By default, it saves the model every epoch; increasing the `epoch_interval` will make checkpointing less frequent.

If the given filename contains the `{}` string, the epoch number will be subtituted in, resulting in multiple checkpoint folders with different names. If a filename such as 'ivis-checkpoint' is provided, only the latest checkpoint will be kept.

Parameters

- **log_dir** (*str*) – Folder to save resulting embeddings.
- **filename** (*str*) – Filename to save each file as. *{}* in string will be substituted with the epoch number.

Example usage:

```
from ivis.nn.callbacks import ModelCheckpoint
from ivis import Ivis

# Save only the latest checkpoint to current directory every 10 epochs
checkpoint_callback = ModelCheckpoint(log_dir='.',
                                      filename='latest-checkpoint.ivis',
                                      epoch_interval=10)

model = Ivis(callbacks=[checkpoint_callback])
```

class ivis.nn.callbacks.**TensorBoardEmbeddingsImage**(*data, labels=None, log_dir='./logs', epoch_interval=1*)

Bases: `sphinx.ext.autodoc.importer._MockObject`

Periodically generates and plots 2D embeddings of the data provided to `data` using the latest state of the `Ivis` model. The plots are designed to be viewed in Tensorboard, which will provide an image that shows the history of embeddings plots through training. By default, saves plots of the embeddings every epoch; increasing the `epoch_interval` will save the plots less frequently.

Parameters

- **data** (*list[float]*) – Data to embed and plot with the latest Ivis
- **labels** (*list[int]*) – Labels with which to colour plotted embeddings. If *None* all points will have the same color.
- **log_dir** (*str*) – Folder to save resulting embeddings.
- **filename** (*str*) – Filename to save each file as. *{}* in string will be substituted with the epoch number.

Example usage:

```python
from ivis.nn.callbacks import TensorBoardEmbeddingsImage
from ivis import Ivis
from tensorflow.keras.datasets import mnsit

(X_train, Y_train), (X_test, Y_test)  = mnist.load_data()

# Plot embeddings of test set every epoch colored by labels
embeddings_callback = TensorBoardEmbeddingsImage(X_test, Y_test,
                                                  log_dir='test-embeddings',
                                                  filename='{}_test_embeddings.npy',
                                                  epoch_interval=1)

model = Ivis(callbacks=[embeddings_callback])

# Train on training set
model.fit(X_train)
```

Python Module Index

i

Index

Symbols

A

B

C

D

E

F

G

I

K

L

M

N

R

S

T

U