# University of Cape Town

STA5003W

Multivariate Statistics

# Multivariate Report

*Authors:*
Gawronsky, Marcus
Singo, Una

*Student Number:*
GWRMAR002
SNGUNA003

October 7, 2019

**Abstract**

Variational Autoencoders (VAEs) are a class of generative artificial neural network commonly used for dimensionality reduction in machine learning applications. In this paper, we implement a stable Variational Autoencoder and benchmark its performance against traditional methods in multivariate statistics using common open data sets. We build on our implementation with two experiments; the first is an application to noncolumnar data in the medical field, and the second is an original generative sampling problem.

# Contents

**Literature Review** **2**

**Setup** **3**

**Data** **3**
- Iris Flowers . . . 3
- Wine . . . 3
- Breast Cancer . . . 4
- Heartbeats Audio Recordings . . . 4

**Analysis** **4**
- Benchmarking . . . 4
  - Dimensionality reduction for clustering . . . 4
  - Manifold Learning . . . 5
- Application to Medical Heartbeats Audio Data . . . 6
- Generative Modeling . . . 8

**Appendixes** **8**
- Benchmarking . . . 8
- Manifold Learning . . . 9
- Heartbeats Audio . . . 10
- Generative Sampling . . . 11
- Code . . . 12
  - Exploratory Analysis . . . 12
  - Dataset Benchmark Code . . . 19
  - Manifold Learning Comparison . . . 23
  - Latent Space Sampling . . . 26
  - Heartbeats Dataset . . . 29
  - Library Code . . . 36
- Environment . . . 55

# Literature Review

Variational Autoencoders (VAEs) are a class of generative artificial neural network commonly used for dimensionality reduction in machine learning applications. The technique extends on the application of traditional autoencoders and deep belief networks by introducing techniques from Variational Bayesian Methods.

An autoencoder consists of two neural networks, an encoder network and decoder network. The encoder network takes an input and produces a lower-dimensional representation of the data. The decoder network takes the lower-dimensional representation of the data and learns a reconstruction of the data back into its original vector-space. These neural networks can have a varying number of hidden-layers, activation functions and neurons to learn non-linear representations of the data (Bengio et al., 2007, 2013).

Variational Bayesian Methods are a class of techniques used in approximating intractable integrals found in Bayesian Modeling. Using an approximate distribution, $q^{\star}(x)$, a loss function is used to minimize the difference between the approximate distribution and true posterior. This loss function is often chosen to be the Evidence Lower Bound (ELBO) or Kullback–Leibler divergence and is weighted in the optimization procedure to ensure the validity of the applied approximate distribution.

The Variational Autoencoder aims to extend on encoder-decoder model architectures by placing distributional assumptions on the low-dimensional latent space. By making this assumption, these models can be generative, allowing one to sample data across a complex data manifold. Figure 1 aims to show the basic architecture of a VAE. The input data $\mathbf{X}$ is assumed to be an *i.i.d.* high dimensional dataset and the latent variables are denoted as $\mathbf{z}$ are assumed to follow an unobservable distribution. The encoder network is defined as the conditional probability distribution $q_\theta(\mathbf{z}|\mathbf{x})$ and outputs estimates to $q_\theta(\mathbf{z}|\mathbf{x})$ which is assumed to follow a parametric probability distribution. Using the estimated parameters from the encoder neural network, sample values of $\mathbf{z}$ are computed and used as an input to the decoder neural network $p_\theta(\mathbf{x}|\mathbf{z})$ which outputs estimates to the conditional distribution $p_\theta(\mathbf{x}|\mathbf{z})$ which can be used to sample values of $\mathbf{x}$. A key aspect to the model is the distributional assumption on the latent space. While several methods can be used to ensure the model distributional assumption, the original authors include the Kullback–Leibler divergence into the loss function of the model to ensure a distribution over latent variable which is approximately standard normal (Kingma and Welling, 2013; Zhao et al., 2017; Tolstikhin et al., 2017).

Using these techniques, Variational Autoencoders provide a flexible method which can be applied to solve a wide range of problems across domains. In the original

paper, Kingma and Welling (2013) use the popular MNIST dataset to both embed and generate new images of handwritten digits. A significant number of subsequent research on VAEs has focused on the learning latent distributions of images and generating new images. Generative Adversarial Networks are a popular example of such research outputs where the objective is to generate fake images that appear to be authentic to human observers (Goodfellow et al., 2014).
Apart from image generation VAEs have been shown to solve anomaly detection problems. An and Cho (2015) propose a method using the reconstruction probability from a variational autoencoder to identify outliers. Their experimental results show that this method outperforms traditional auto-encoder and principal components based methods.

# Setup

Code for this assignment was written in Python version 3.7.1, using a random seed of 1234, a full list of the Python dependencies are listed in the appendencies in a YML file. This can be used, along with the code [1], to reproduce the analysis provided.

# Data

Five open data sets and two simulated are used in this assignment. Of the open data sets, three are column-oriented and two audio based. This section will provide a short description of the data, its source, and some information about its characteristics.

## Iris Flowers

The Iris flower dataset (Fisher, 1936) measures 4 distinguishing features of three related species of Iris flowers, Setosa, Veriscolour and Virginica. It contains 150 observations with 5 variables, a classifier, the sepal length, sepal width, petal length and petal width. It is commonly used as an introduction to solving classification, clustering and dimensional reduction problems.

## Wine

The Wine dataset (Aeberhard et al., 1994) contains results from a chemical study of wine grown in the same region in Italy but grown by three different farmers.

[1] The code is accessible on github: https://github.com/marcusinthesky/super-spirals.git

The data contains 178 observations, 13 predictive numerical features that measure chemicals properties.

### Breast Cancer

The Breast Cancer (Street et al., 1993) data contains features computed from an image of a fine needle aspirate (FNA). Fine-needle aspiration (FNA) is a diagnostic procedure used to investigate lumps. Using a fine needle cells from the lump are extracted and studied under a microscope, from the image 30 features can be measured that characterise the mass and distinguish it from being Benign or Malignant.

### Heartbeats Audio Recordings

The Heartbeats dataset (Bentley et al.) will form part of our non column-oriented datasets.
Heartbeats is an audio based data set that contains recordings of heartbeats from a stethoscope. There are four classes of audio, artifact, extrahls, murmer and normal. The data set was originally created to identify S1 (dub) and S2 (dub) sounds and classify beats into one of the four classes.

# Analysis

In the literature, we have seen a variety of VAE implementations to solve multivariate problems. In this section, we present benchmark the applications of VAEs to problems in dimensionality reduction for clustering and manifold learning. We also demonstrate the application of VAEs to real-world datasets from the medical field and test an original generative modelling example.

## Benchmarking

#### Dimensionality reduction for clustering

Clustering high dimensional data can be difficult as regions in the space become increasingly sparse and data tends to equidistance. To solve this 'curse of dimensionality', dimensionality reduction is often applied in order to control for the correlation between dimensions and to aid in identifying separable clusters. We explored the feasibility of using a VAE to map high dimensional data to a lower-dimensional space for use in clustering.

We trained VAEs, on the Iris, Cancer and Wine dataset and visualised their latent space. As a benchmark, we also implemented Principal Component Analysis (PCA), Independent Component Analysis (ICA) and Kernal Principal Component Analysis (KPCA). Figures 2, 3 and 4 show the input data mapped to the latent space, point is coloured according to class.

By observation, the PCA for all data sets produces latent spaces with significantly more variation within classes compared to the other models. The KPCA seems to be the most consistent across datasets in the way that it maps inputs to the latent space. The ICA seems to produce tightly condensed clusters that will be difficult to separate.

To get a quantitative sense of the feasibility of clustering data in the latent space, we needed to create a ground truth clustering measure. We used the original class labels computed silhouette scores. These scores summarise the level of cohesion within clusters and separation between clusters. Scores range from -1 to 1, and high values mean that objects of a cluster are cohesive and well separated to other clusters.

Figure 5 shows the silhouette scores for each method trained on Iris, Cancer and Wine. The KPCA methods consistently produce scores that are above 0.4. The KPCA(RBF) method produces the highest score with the Wine dataset. The VAEs performance varies significantly between datasets. The VAE(Tanh) performs the best when trained on the Iris and Cancer dataset but is the weakest performer on the wine dataset. The VAE(ReLu) only performs well with the Cancer dataset and performs poorly with the Iris and Wine dataset. It appears that the VAEs performance is quite sensitive to the type of activation function used in training. Given the lack of consistent performance, the feasibility of using a VAE for efficient dimensionality reduction as a pre-processing step for clustering is questionable. While methods such as KPCA produced consistent results, there is a trade-off. The KPCA method is are basis expansion methods, and so time complexity is likely to be a problem.

**Manifold Learning**

# Application to Medical Heartbeats Audio Data

For the application section of this report, a public medical dataset of patient heartbeats was chosen. This dataset, provided by Bentley et al., was gathered from both general public via the iStethoscope Pro iPhone app, and from clinical trials in hospitals using the digital stethoscope DigiScope. Many of these 832 are noisy and contain artefacts in the data. The dataset has labels for the recordings, marking them as recording either normal heartbeats, recordings with heart murmurs or extrahls or recordings with noise artefacts which make classification by physicians challenging. Even though the distributions of subsets of the data may be different the aims of this task was to use the data collected from the clinical trial using the digital stethoscope DigiScope to train a Variational Autoencoder to encode data from the iStethoscope Pro iPhone app collected from the general public and identify whether, using this data we could accurately identify recordings with artefacts, as well as heart conditions.

**Data processing**

The original data structure was stored in wav files. A wav file is a raw uncompressed audio file that stores audio data, sample rate, and bit rate. The sample rate was 44,100Hz per second, and so in the time domain, there are too many observations to efficient perform machine learning. Data was read from the wav files and normalized in order to ensure that the audio tracks were roughly the same volume. A simple filter was used in order to clip loud pops from the recordings and split in individual heartbeats by searching for extrema in the signal from within a specified neighbourhood of one-thousand time-steps and then subsampling to ensure the recordings were of equal length. By splitting the data into individual heartbeats, we massively increased the sample size and also managed to better control for elements of the waveform which determined heart-rate as opposed to artefacts, heart murmurs or normal heartbeats. We used a Fast Fourier Transform (FFT) method to transform the high dimensional audio data from an amplitude-time domain to an amplitude-frequency domain of 40 dimensions. The final dataset yielded 11483 samples of heartbeats for in training our model.

While we did experiment with various hyper-parameters our final Variational Autoencoder featured 594 parameters on the encoder with two hidden layers of 15 and 10 nodes each using the tanh activation function, the model was trained over 1000 epochs using the ADAM optimizer with an l2 regularization term of 0.0001 and a one-to-one weighting between our reconstruction loss and the weighting of our KL Divergence term.

**Results**

Figure 8 shows the latent space of the trained VAE across three random initializations. While the data is characterized by imperfect sample recordings, based on the figure 8 and results of tables 1 and 2, we can see that the model tends to separate our the data artefacts at the tails of the distribution, with normal heartbeats closest to the mean. This suggests that for data-cleaning and anomaly detection, the Variational Autoencoder may present value to further downstream analysis.

| | **Component 1** | |
|---|---|---|
| | mean | std |
| label | | |
| artifact | -0.565199 | 0.736880 |
| extrahls | -0.634950 | 0.719695 |
| murmur | -0.981001 | 0.404676 |
| normal | -0.795812 | 0.568068 |

Table 1: Summary statistics on component 1 for each heartbeat label

| | **Component 2** | |
|---|---|---|
| | mean | std |
| label | | |
| artifact | 0.172482 | 1.096362 |
| extrahls | -0.313284 | 1.023796 |
| murmur | -0.077684 | 0.546320 |
| normal | -0.198015 | 0.840402 |

Table 2: Summary statistics on component 2 for each heartbeat label

## Generative Modeling

Since their popularization, a widely publicized application of Variational Autoencoders has been in generating new images. Using the prior distribution of the latent space, researchers will commonly sample from this distribution and use the decoder model to generate sampled from the high dimensional data manifold. While this method is novel, this method does make several assumptions around the size of the data, the capacity of the network and the underlying distribution of the data.

In order to experiment and test the notion of sampling from complicated distributions, we took to experiment with an extremely shallow neural network architecture- with only 6 and 3 hidden nodes with Rectified Linear Unit (ReLU) activation functions- and a simple distribution- the truncated Bivariate Gaussian. In this example, we aimed to compare the ability of the Variational Autoencoder- given enough samples- to embed this truncated bivariate data onto a one-dimensional line in a manner which maintained the original distributional properties of the data.

Figure 9 shows the newly sampled data superimposed with the original Gaussian dataset. The random samples generated from the VAE seem to have low variance and low bias- sampling points from around the mean of the truncated distribution. This turns out to be a complicated yet straightforward problem to solve for methods in dimensionality reduction as similar properties arise when applying methods such as Principle Component Analysis, as shown in figure 10, and Kernel Principal Component Analysis, as shown in figure 11. When trying to use linear dimensionalty reduction, it is impossible to find a reconstruction which accurately reconstructs the data and when using a kernel method methods either tend to high variance or the mean of the distribution as in the case of the kernel method.

While the model architecture and choice of loss function weighting make a huge difference in the stability and capacity of the model to generate samples accurately, it is clear that these methods rely on the correlations in the data in order to accurately embed the data in a lower-dimensional space. This is not easy for all datasets, as shown when given non-symmetric data from the truncated Bivariate Gaussian.

# Appendixes

## Benchmarking

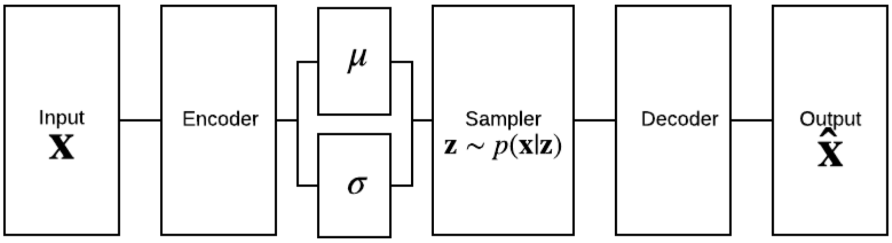


Figure 1: Basic architecture of a Variational Autoencoder

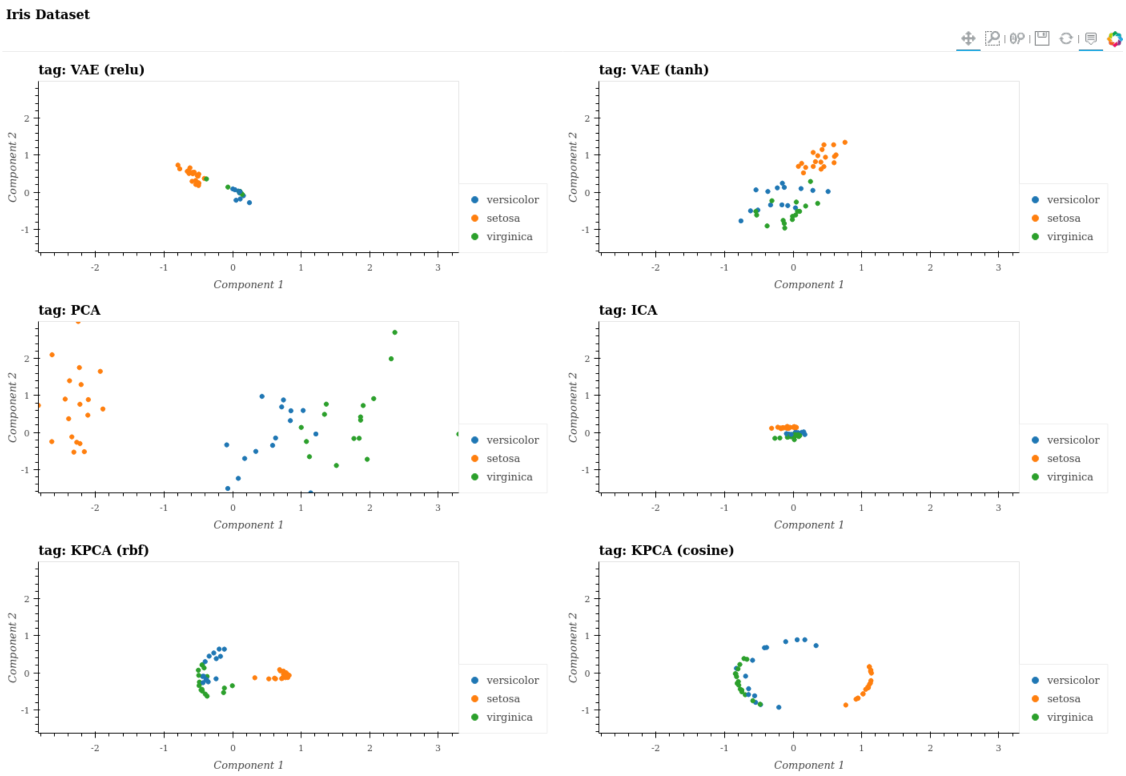


Figure 2: Iris dataset Latent Space representation

# Manifold Learning

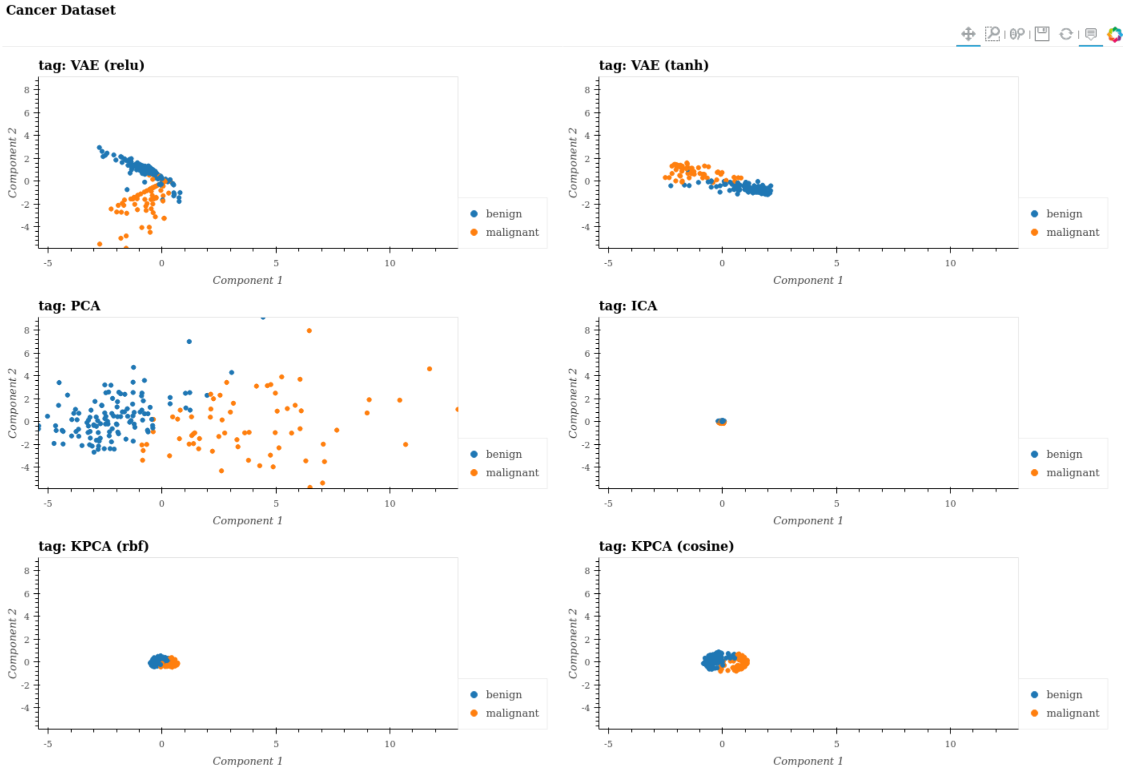


Figure 3: Cancer dataset Latent Space representation

# Heartbeats Audio

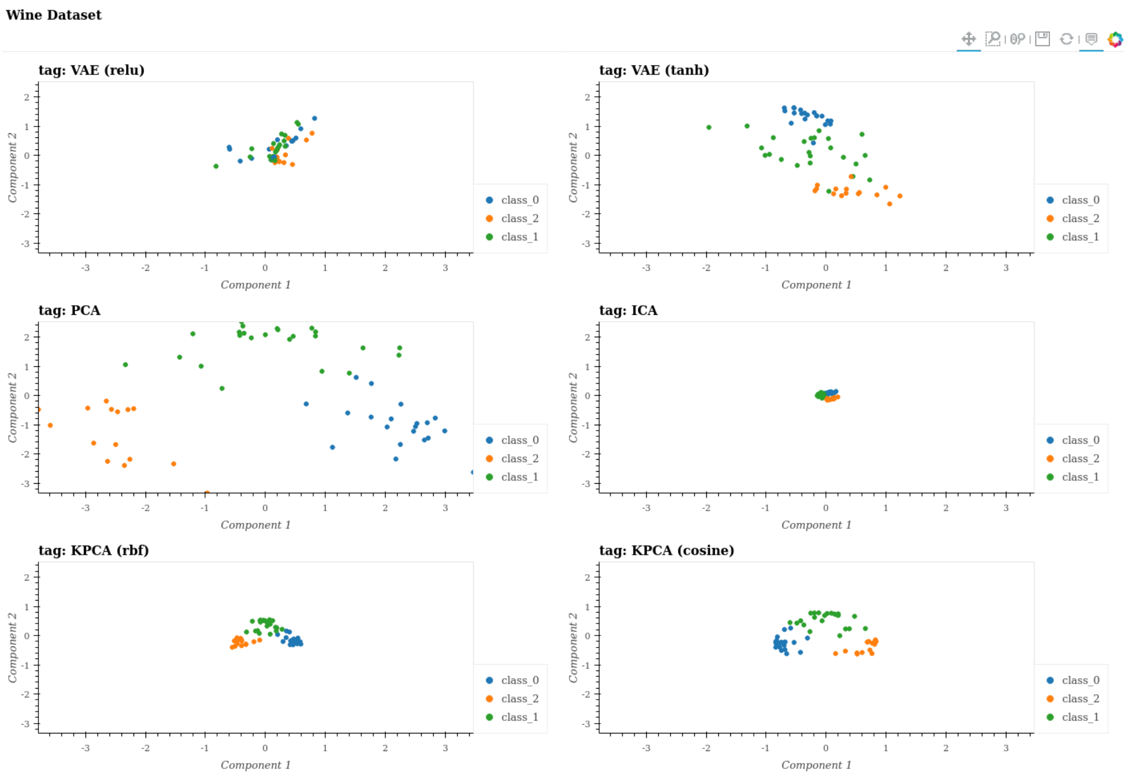


Figure 4: Wine dataset Latent Space representation

# Generative Sampling

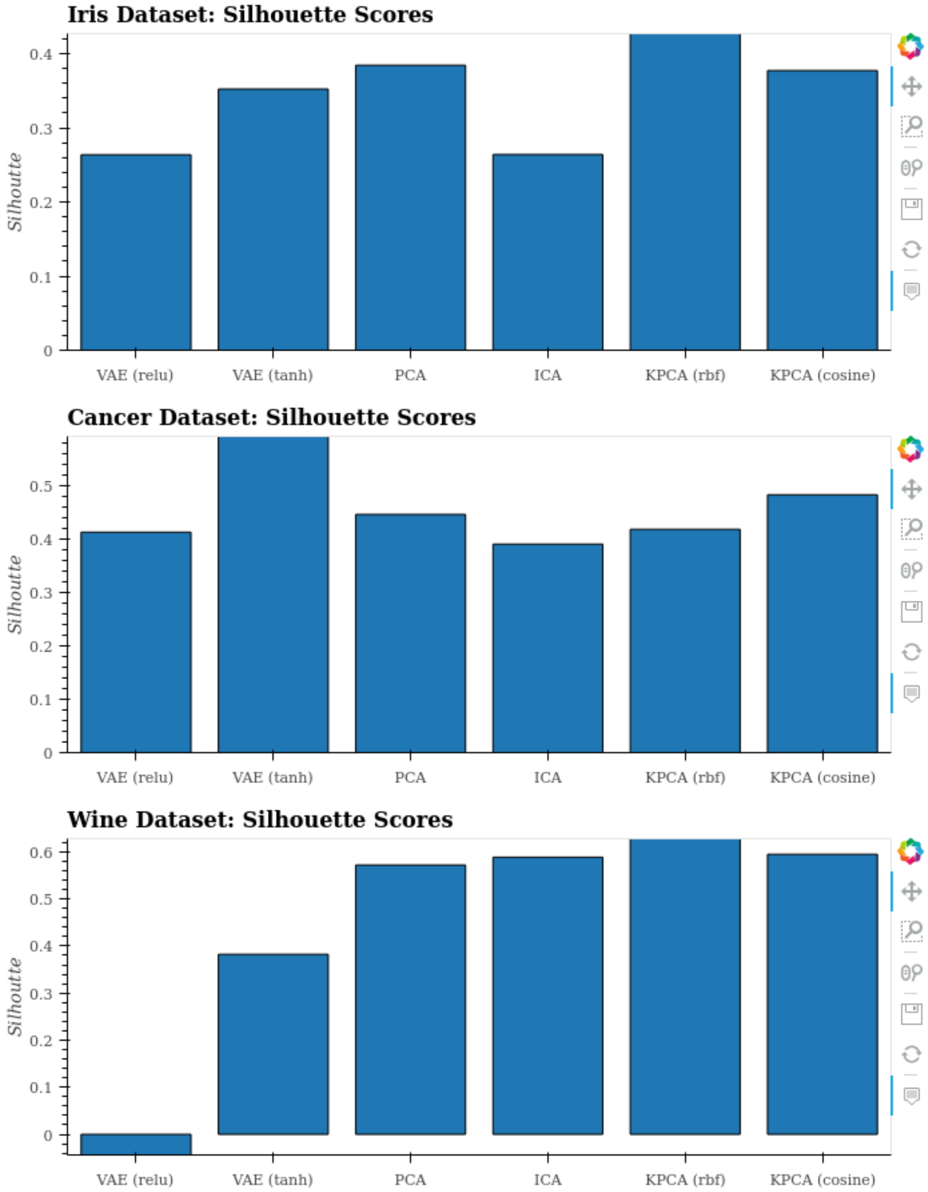


Figure 5: Silhouette Scores

# Code

## Exploratory Analysis

```
# ---
# jupyter:
#   jupytext:
#     text_representation:
```

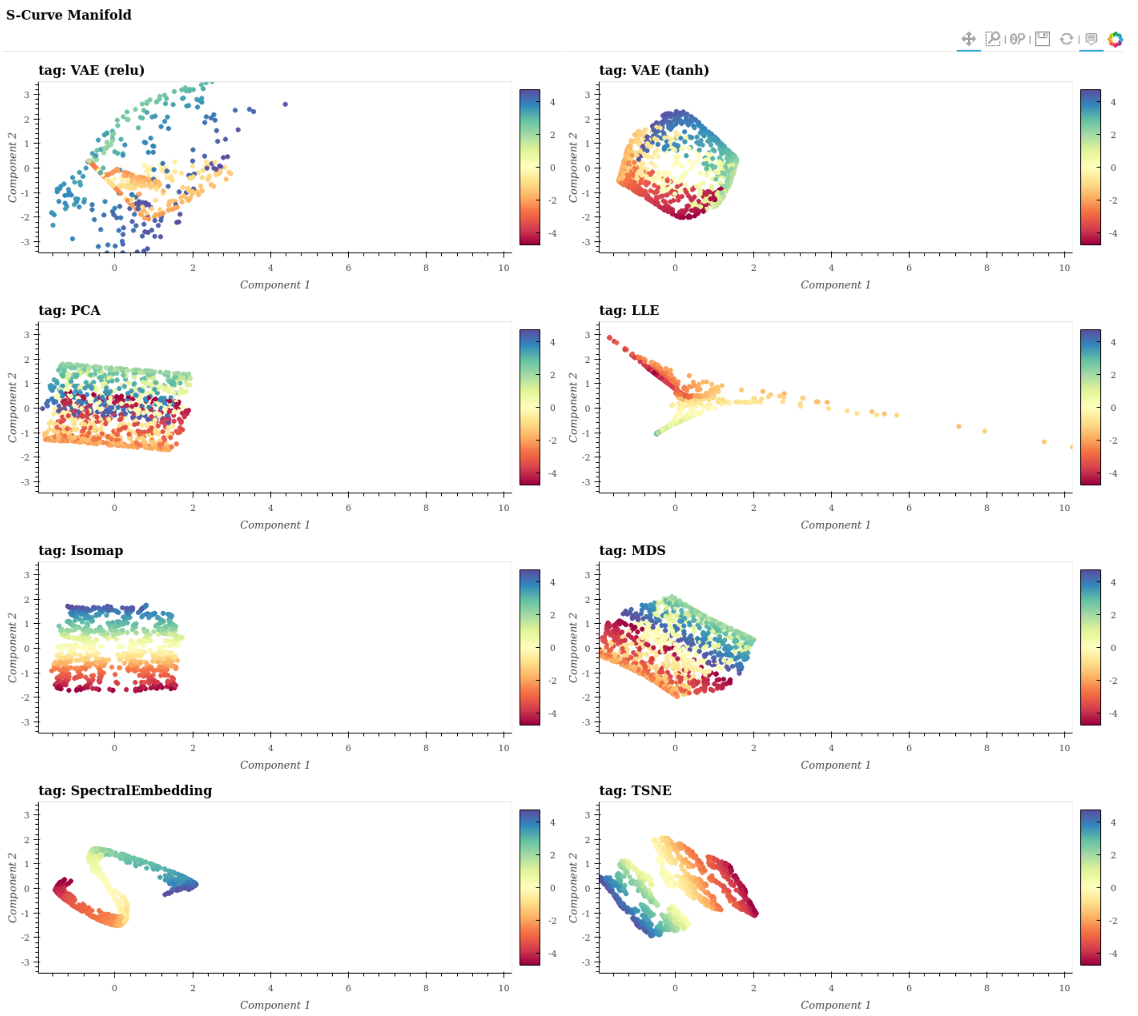


Figure 6: S-Curve Manifold Learning

```
#         extension: .py
#         format_name: percent
#         format_version: '1.2'
#         jupytext_version: 1.2.4
#     kernelspec:
#       display_name: Python 3
#       language: python
#       name: python3
# ---

# %% {"slideshow": {"slide_type": "skip"}, "language": "html"}
# <style>
# div.input {
```

```
#     display:none;
# }
# </style>

# %% {"slideshow": {"slide_type": "skip"}, "language": "html"}
# <style>
# div.input {
#     display:contents;
# }
# </style>

# %% {"slideshow": {"slide_type": "skip"}}
import sys
```

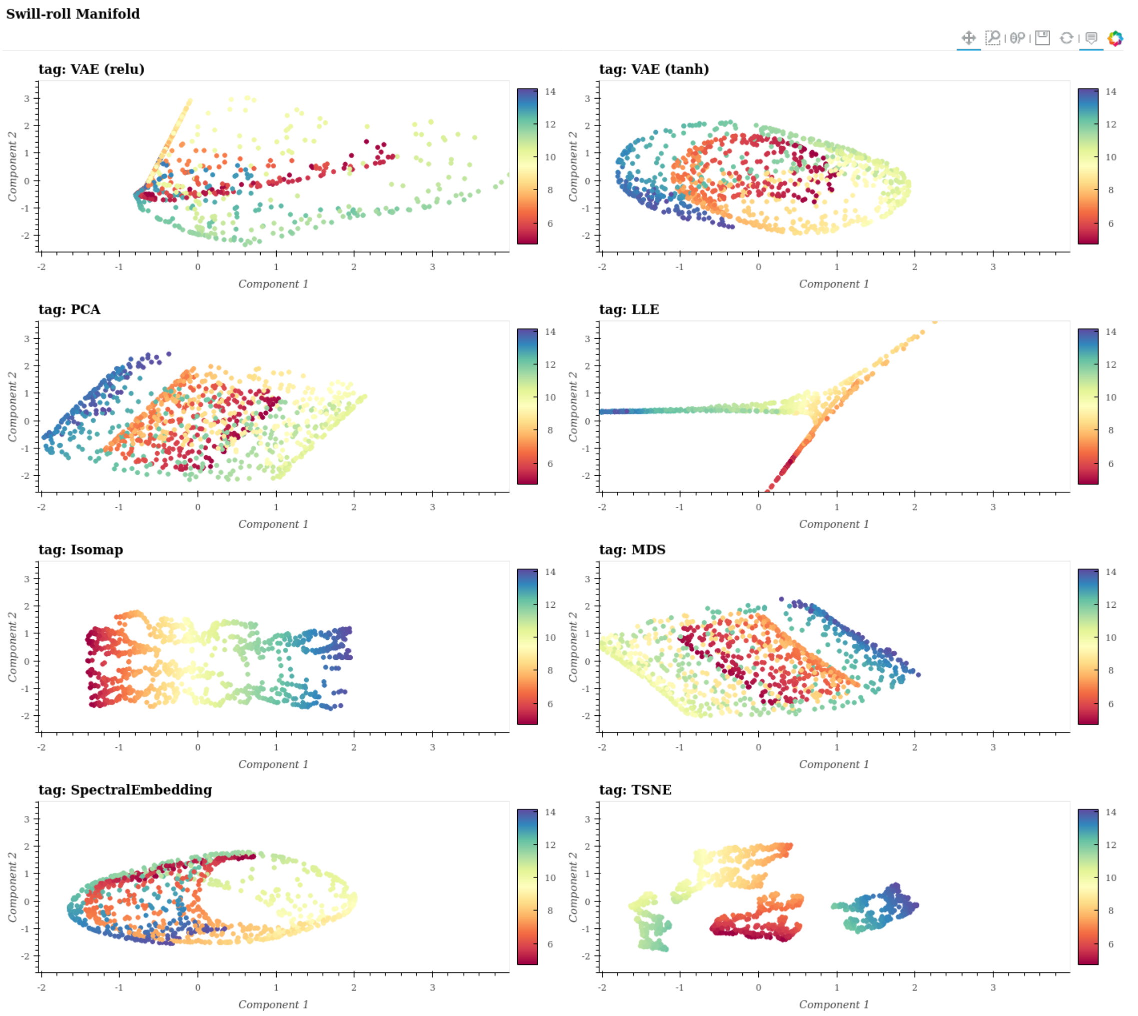


Figure 7: Swiss Roll manifold learning

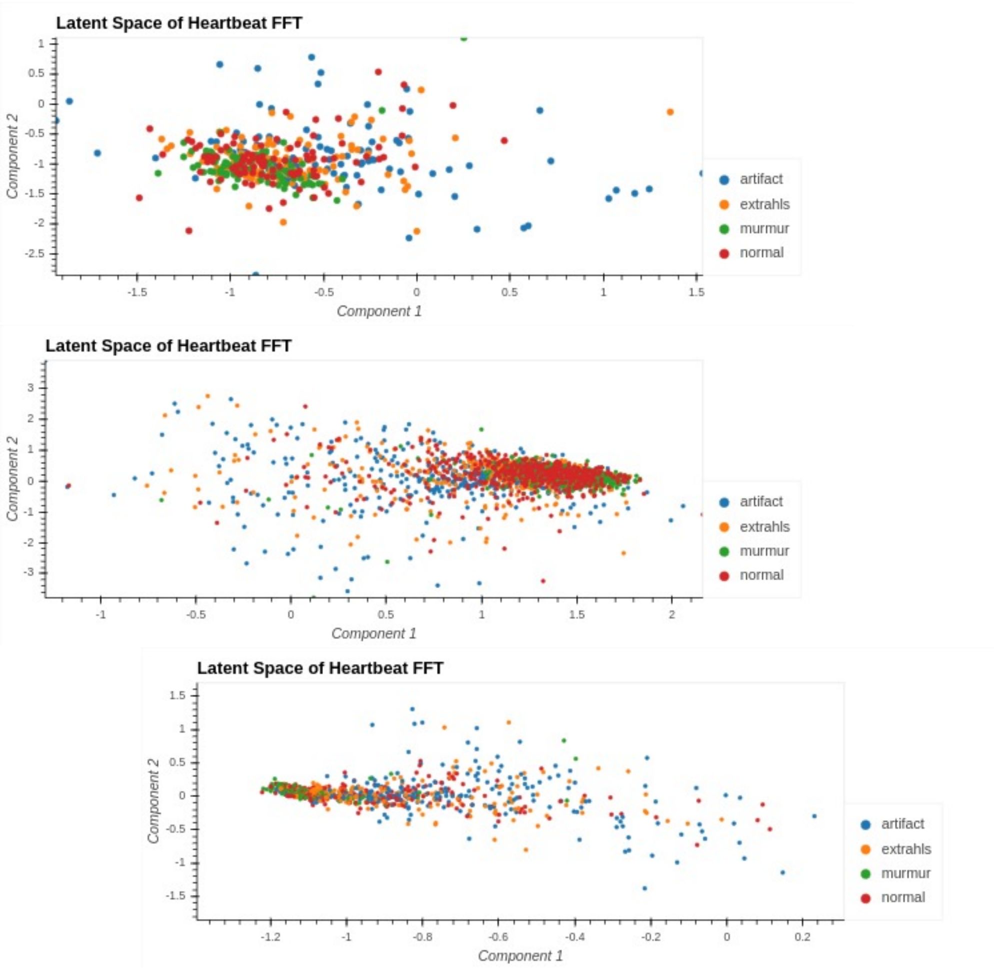


Figure 8: Heartbeat Latent Space Representation

```
from sklearn.datasets import load_iris
import pandas as pd
import numpy as np
import holoviews as hv
import hvplot.pandas
from toolz.curried import *
from sklearn.decomposition import PCA
from sklearn.manifold import MDS, TSNE
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import train_test_split, KFold


sys.path.append("../")
```

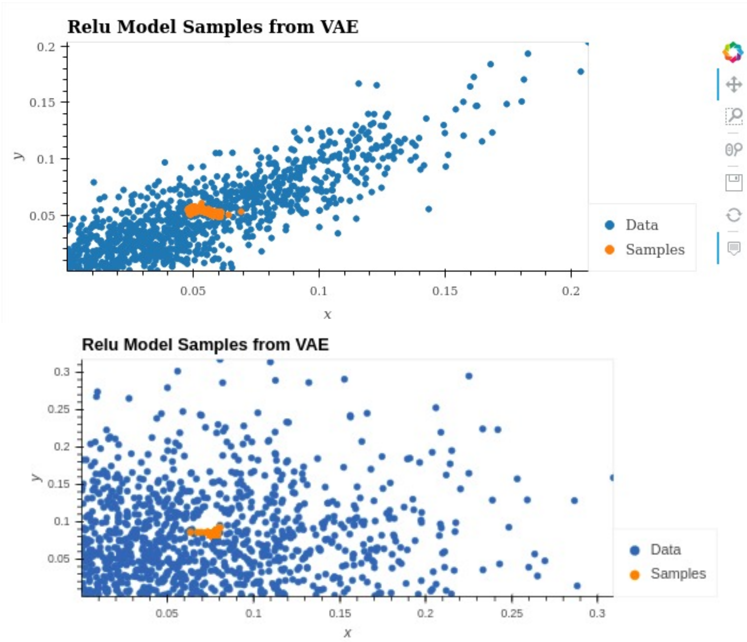


Figure 9: Gaussian Generative Modeling example

```
hv.extension("bokeh")

```

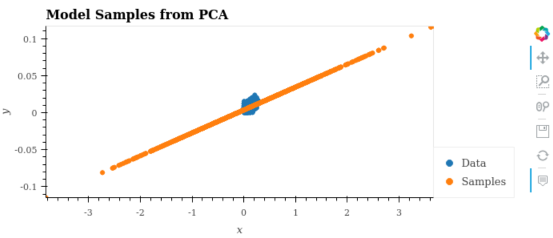


Figure 10: PCA on Bivariate Gaussian Generative Modeling example

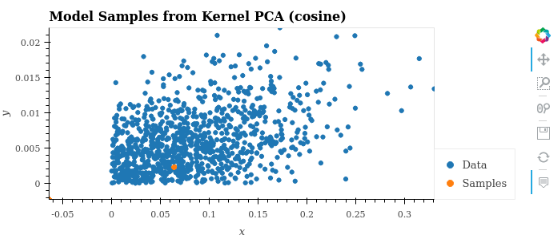


Figure 11: Cosine Kernel PCA on Bivariate Gaussian Generative Modeling

```
# %% {"slideshow": {"slide_type": "skip"}}
from super_spirals.neural_network import VAE

# %% [markdown] {"slideshow": {"slide_type": "slide"}}
# # Preliminary Analysis

# %% {"slideshow": {"slide_type": "skip"}}
X = load_iris()

# %% {"slideshow": {"slide_type": "skip"}}
pipeline = make_pipeline(
    StandardScaler(),
    VAE(
        hidden_layer_sizes=(5, 2), activation="tanh", divergence_weight
=5, max_iter=500
    ),
)

# %% {"slideshow": {"slide_type": "skip"}}
pipeline.fit(X=X.data)

# %% [markdown] {"slideshow": {"slide_type": "slide"}}
# __Inpect model__

# %% [markdown] {"slideshow": {"slide_type": "-"}}
# Encoder

# %% {"slideshow": {"slide_type": "-"}}
pipeline.named_steps["vae"].encoder.summary()

# %% [markdown] {"slideshow": {"slide_type": "slide"}}
# __Decoder__

# %% {"slideshow": {"slide_type": "-"}}
pipeline.named_steps["vae"].decoder.summary()

# %% [markdown] {"slideshow": {"slide_type": "slide"}}
# __Denoising__

# %% [markdown] {"slideshow": {"slide_type": "-"}}
# Original data

# %% {"slideshow": {"slide_type": "-"}}
# original data
sample = X.data[:10, :]
pipe(sample, partial(pd.DataFrame, columns=X.feature_names))

# %% [markdown] {"slideshow": {"slide_type": "slide"}}
# 'Denoised' Data
```

```
# %% {"slideshow": {"slide_type": "-"}}
# denoised
pipe(
    sample,
    pipeline.transform,
    pipeline.inverse_transform,
    partial(pd.DataFrame, columns=X.feature_names),
)

# %% [markdown] {"slideshow": {"slide_type": "slide"}}
# __Generate new samples__

# %% {"slideshow": {"slide_type": "-"}}
pipe(
    np.random.multivariate_normal(np.zeros(2), np.diag(np.ones(2)),
    size=10),
    pipeline.inverse_transform,
)

# %% [markdown] {"slideshow": {"slide_type": "slide"}}
# __Dimensionality Reduction__

# %% {"slideshow": {"slide_type": "-"}}
# %%output filename='../media/01-iris-latent' fig='png'
(
    pipe(
        X.data,
        pipeline.transform,
        partial(pd.DataFrame, columns=["Component 1", "Component 2"]),
    )
    .assign(label=X.target)
    .assign(label=lambda d: d.label.replace(dict(enumerate(X.
    target_names))))
    .hvplot.scatter(
        x="Component 1",
        y="Component 2",
        color="label",
        label="Variational Autoencoder Latent Encoding",
    )
)

# %% [markdown] {"slideshow": {"slide_type": "slide"}}
# __Anomaly Detection__

# %% [markdown] {"slideshow": {"slide_type": "-"}}
# Assume normal, use Z-scores to filter outliers
```

### Dataset Benchmark Code

```python
# ---
# jupyter:
#   jupytext:
#     text_representation:
#       extension: .py
#       format_name: percent
#       format_version: '1.2'
#       jupytext_version: 1.2.4
#   kernelspec:
#     display_name: Python 3
#     language: python
#     name: python3
# ---

# %% {"slideshow": {"slide_type": "skip"}, "language": "html"}
# <style>
# div.input {
#     display:none;
# }
# </style>

# %% {"slideshow": {"slide_type": "skip"}, "language": "html"}
# <style>
# div.input {
#     display:contents;
# }
# </style>

# %% [markdown] {"slideshow": {"slide_type": "slide"}}
# # Dimensionality Reduction
# 1. Compressed representation which capture all relevant structure in
  the data
# 2. Representation useful in supervised and unsupervised learning
  tasks

# %% [markdown] {"slideshow": {"slide_type": "slide"}}
# ### Models
# 1. VAE (Tanh)
# 2. VAE (ReLU)
# 3. PCA
# 4. ICA
# 5. Kernel PCA (RBF)
# 6. Kernel PCA (Cosine)

# %% [markdown] {"slideshow": {"slide_type": "slide"}}
# ### Data
# 1. Fisher's Iris data
```

```
# 2. UCI ML Wine Data
# 3. UCI ML Breast Cancer Wisconsin (Diagnostic)
#

# %% {"slideshow": {"slide_type": "skip"}}
import sys
from sklearn.datasets import (
    load_iris,
    load_wine,
    load_breast_cancer,
    make_multilabel_classification,
)
import pandas as pd
import numpy as np
import holoviews as hv
import hvplot.pandas
from typing import Dict
from toolz.curried import *
from sklearn.decomposition import PCA, FastICA, KernelPCA
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline


sys.path.append("../")
hv.extension("bokeh")

# %% {"slideshow": {"slide_type": "skip"}}
from super_spirals.metrics import reconstruction_benchmark
from super_spirals.neural_network import VAE


# %% {"slideshow": {"slide_type": "skip"}}
def get_models():
    return {
        "VAE (relu)": make_pipeline(
            StandardScaler(),
            VAE(activation="relu", max_iter=300, hidden_layer_sizes=(4,
    2)),
        ),
        "VAE (tanh)": make_pipeline(
            StandardScaler(),
            VAE(activation="tanh", max_iter=300, hidden_layer_sizes=(4,
    2)),
        ),
        "PCA": make_pipeline(StandardScaler(), PCA(n_components=2)),
        "ICA": make_pipeline(StandardScaler(), FastICA(n_components=2))
  ,
        "KPCA (rbf)": make_pipeline(
            StandardScaler(),
```

```
            KernelPCA(n_components=2, kernel="rbf", fit_inverse_transform=True),
        ),
        "KPCA (cosine)": make_pipeline(
            StandardScaler(),
            KernelPCA(n_components=2, kernel="cosine", fit_inverse_transform=True),
        ),
    }


# %% [markdown] {"slideshow": {"slide_type": "slide"}}
# # Iris

# %% {"slideshow": {"slide_type": "skip"}}
iris_models = get_models()
iris_df, iris_reconstruction, iris_silhouette, iris_plot = reconstruction_benchmark(
    load_iris(), iris_models, "Iris Dataset"
)

# %% {"slideshow": {"slide_type": "slide"}}
# %%output filename='../media/02-iris-loss' fig='png'
iris_reconstruction

# %% {"slideshow": {"slide_type": "slide"}}
# %%output filename='../media/02-iris-silhouette' fig='png'
iris_silhouette

# %% {"slideshow": {"slide_type": "slide"}}
# %%output filename='../media/02-iris-latent' fig='png'
iris_plot

# %% [markdown] {"slideshow": {"slide_type": "slide"}}
# # Wine

# %% {"slideshow": {"slide_type": "skip"}}
wine_models = get_models()
wine_df, wine_reconstruction, wine_silhouette, wine_plot = reconstruction_benchmark(
    load_wine(), wine_models, "Wine Dataset"
)

# %% {"slideshow": {"slide_type": "slide"}}
# %%output filename='../media/02-wine-loss' fig='png'
wine_reconstruction

# %% {"slideshow": {"slide_type": "slide"}}
# %%output filename='../media/02-wine-silhouette' fig='png'
```

```
wine_silhouette

# %% {"slideshow": {"slide_type": "slide"}}
# %%output filename='../media/02-wine-latent' fig='png'
wine_plot

# %% [markdown] {"slideshow": {"slide_type": "slide"}}
# # Cancer

# %% {"slideshow": {"slide_type": "skip"}}
cancer_models = get_models()
cancer_df, cancer_reconstruction, cancer_silhouette, cancer_plot =
    reconstruction_benchmark(
    load_breast_cancer(), cancer_models, "Cancer Dataset"
)

# %% {"slideshow": {"slide_type": "slide"}}
# %%output filename='../media/02-cancer-loss' fig='png'
cancer_reconstruction

# %% {"slideshow": {"slide_type": "slide"}}
# %%output filename='../media/02-cancer-silhouette' fig='png'
cancer_silhouette

# %% {"slideshow": {"slide_type": "slide"}}
# %%output filename='../media/02-cancer-latent' fig='png'
cancer_plot


# %% {"slideshow": {"slide_type": "skip"}}
class get_random:
    random = make_multilabel_classification(n_samples=1000, n_features
    =20, n_classes=5)
    data = random[0]
    target = pd.DataFrame(random[1]).idxmax(1).to_numpy()


# %% {"slideshow": {"slide_type": "skip"}}
random_models = get_models()
random_df, random_reconstruction, random_silhouette, random_plot =
    reconstruction_benchmark(
    get_random(), random_models, "Random Dataset"
)

# %% {"slideshow": {"slide_type": "skip"}}
# %%output filename='../media/02-random-loss' fig='png'
random_reconstruction

# %% {"slideshow": {"slide_type": "skip"}}
```

```
# %%output filename='../media/02-random-silhouette' fig='png'
random_silhouette

# %% {"slideshow": {"slide_type": "skip"}}
# %%output filename='../media/02-random-latent' fig='png'
random_plot

# %% {"slideshow": {"slide_type": "skip"}}
```

### Manifold Learning Comparison

```
# ---
# jupyter:
#   jupytext:
#     text_representation:
#       extension: .py
#       format_name: percent
#       format_version: '1.2'
#       jupytext_version: 1.2.4
#   kernelspec:
#     display_name: Python 3
#     language: python
#     name: python3
# ---

# %% [markdown] {"slideshow": {"slide_type": "slide"}}
# # Manifold Learning
# 1. Data Exists in Manifold within the high dimensional space
# 2. Often non-linear surface

# %% [markdown] {"slideshow": {"slide_type": "slide"}}
# ### Models
# - VAE (relu)
# - VAE (tanh)
# - PCA
# - LLE
# - Isomap
# - MDS
# - SpectralEmbedding
# - TSNE

# %% [markdown] {"slideshow": {"slide_type": "slide"}}
# ### Data
# - S-curve
# - Swiss Roll

# %% {"slideshow": {"slide_type": "skip"}, "language": "html"}
# <style>
# div.input {
```

```
#       display:none;
# }
# </style>

# %% {"slideshow": {"slide_type": "skip"}, "language": "html"}
# <style>
# div.input {
#       display:contents;
# }
# </style>

# %% {"slideshow": {"slide_type": "skip"}}
import sys
from sklearn.datasets import samples_generator
import pandas as pd
import numpy as np
import holoviews as hv
import hvplot.pandas
from toolz.curried import *
from sklearn import manifold
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import train_test_split, KFold


sys.path.append("../")
hv.extension("bokeh")

# %% {"slideshow": {"slide_type": "skip"}}
from super_spirals.neural_network import VAE

# %% {"slideshow": {"slide_type": "skip"}}
n_points = 1000


# %% {"slideshow": {"slide_type": "skip"}}
def get_models():
    return {
        "VAE (relu)": make_pipeline(
            StandardScaler(),
            VAE(activation="relu", max_iter=300, hidden_layer_sizes=(4, 5, 2)),
        ),
        "VAE (tanh)": make_pipeline(
            StandardScaler(),
            VAE(activation="tanh", max_iter=300, hidden_layer_sizes=(4, 5, 2)),
        ),
```

```
        "PCA": make_pipeline(StandardScaler(), PCA(n_components=2)),
        "LLE": make_pipeline(
            StandardScaler(), manifold.LocallyLinearEmbedding(n_components=2)
        ),
        "Isomap": make_pipeline(StandardScaler(), manifold.Isomap(n_components=2)),
        "MDS": make_pipeline(StandardScaler(), manifold.MDS(n_components=2)),
        "SpectralEmbedding": make_pipeline(
            StandardScaler(), manifold.SpectralEmbedding(n_components=2)
        ),
        "TSNE": make_pipeline(StandardScaler(), manifold.TSNE(n_components=2)),
    }


# %% {"slideshow": {"slide_type": "skip"}}
def get_components(model, X, y, tag):

    latent = pipe(
        X,
        model.fit_transform,
        StandardScaler().fit_transform,
        PCA(whiten=True).fit_transform,
        partial(pd.DataFrame, columns=["Component 1", "Component 2"]),
    )

    return latent.assign(y=y).assign(tag=tag)


# %% {"slideshow": {"slide_type": "skip"}}
s_curve_models = get_models()
s_curve_X, s_curve_color = samples_generator.make_s_curve(n_points, random_state=0)

s_curve_components = pd.concat(
    [get_components(m, s_curve_X, s_curve_color, t) for t, m in s_curve_models.items()]
)

# %% {"slideshow": {"slide_type": "slide"}}
# %%output filename='../media/03-scurve-latent' fig='png'
(
    s_curve_components.hvplot.scatter(
        x="Component 1", y="Component 2", color="y", groupby="tag", cmap="spectral"
    )
```

```
    .layout()
    .opts(title="S-Curve Manifold", shared_axes=False)
    .cols(2)
)

# %% {"slideshow": {"slide_type": "skip"}}
swissroll_models = get_models()
swissroll_X, swissroll_color = samples_generator.make_swiss_roll(
    n_points, random_state=0
)

swissroll_components = pd.concat(
    [
        get_components(m, swissroll_X, swissroll_color, t)
        for t, m in swissroll_models.items()
    ]
)

# %% {"slideshow": {"slide_type": "slide"}}
# %%output filename='../media/03-swissroll-latent' fig='png'
(
    swissroll_components.hvplot.scatter(
        x="Component 1", y="Component 2", color="y", groupby="tag",
    cmap="spectral"
    )
    .layout()
    .opts(title="Swill-roll Manifold", shared_axes=False)
    .cols(2)
)
```

### Latent Space Sampling

```
# ---
# jupyter:
#   jupytext:
#     text_representation:
#       extension: .py
#       format_name: percent
#       format_version: '1.2'
#       jupytext_version: 1.2.4
#   kernelspec:
#     display_name: Python 3
#     language: python
#     name: python3
# ---

# %% [markdown] {"slideshow": {"slide_type": "skip"}}
# # Sampling

```

```
# %% [markdown] {"slideshow": {"slide_type": "slide"}}
# __Aim__
# Truncated Bivariate Gaussian $\rightarrow$ 1D Representation $\rightarrow$ Truncated Bivariate Gaussian

# %% {"slideshow": {"slide_type": "skip"}}
import sys
import pandas as pd
import numpy as np
from sklearn.decomposition import PCA, KernelPCA
import holoviews as hv
import hvplot.pandas
from toolz.curried import *


sys.path.append("../")
hv.extension("bokeh")

# %% {"slideshow": {"slide_type": "skip"}}
from super_spirals.neural_network import VAE

# %% {"slideshow": {"slide_type": "skip"}}
X = np.random.multivariate_normal(
    mean=np.zeros(2), cov=np.diag(np.ones(2)), size=100000
)

beta = np.random.uniform(-0.1, 0.1, size=(2, 2))
data = pd.DataFrame(X @ beta).where(lambda d: d > 0).dropna(how="any").to_numpy()

# %% {"slideshow": {"slide_type": "skip"}}
vae = VAE(
    hidden_layer_sizes=(6, 3, 1),
    max_iter=500,
    activation="relu",
    alpha=0.0005,
    divergence_weight=0,
    batch_size=100000 / 10,
)

# %% {"slideshow": {"slide_type": "skip"}}
vae.fit(x=data)

# %% {"slideshow": {"slide_type": "slide"}}
# %%output filename='../media/04-generative-samples' fig='png'
(
    (
        pd.DataFrame(data, columns=["x", "y"])
        .sample(1000)
```

```
        .hvplot.scatter(x="x", y="y", label="Data")
    )
    * (
        pd.DataFrame(vae.sample(1000), columns=["x", "y"]).hvplot.scatter(
            x="x", y="y", label="Samples"
        )
    )
).opts(title="Relu Model Samples from VAE", tools=[])

# %%
kpca = KernelPCA(1, kernel="rbf")
kpca.fit(X=data)

# %% {"slideshow": {"slide_type": "skip"}}
# %%output filename='../media/04-generative-pca' fig='png'
(
    (
        pd.DataFrame(data, columns=["x", "y"])
        .sample(1000)
        .hvplot.scatter(x="x", y="y", label="Data")
    )
    * (
        pd.DataFrame(
            pca.inverse_transform(np.random.normal(size=(1000,)).reshape(-1, 1)),
            columns=["x", "y"],
        ).hvplot.scatter(x="x", y="y", label="Samples")
    )
).opts(title="Relu Model Samples from PCA", tools=[])


# %%
pca = PCA(1)
pca.fit(X=data)

# %%
# %%output filename='../media/04-generative-pca' fig='png'
(
    (
        pd.DataFrame(data, columns=["x", "y"])
        .sample(1000)
        .hvplot.scatter(x="x", y="y", label="Data")
    )
    * (
        pd.DataFrame(
            pca.inverse_transform(np.random.normal(size=(1000,)).reshape(-1, 1)),
            columns=["x", "y"],
```

```
        ).hvplot.scatter(x="x", y="y", label="Samples")
    )
).opts(title="Relu Model Samples from PCA", tools=[])
```

### Heartbeats Dataset

```
# ---
# jupyter:
#   jupytext:
#     text_representation:
#       extension: .py
#       format_name: percent
#       format_version: '1.2'
#       jupytext_version: 1.2.4
#   kernelspec:
#     display_name: Python 3
#     language: python
#     name: python3
# ---

# %% {"language": "html"}
# <style>
# div.input {
#     display:none;
# }
# </style>

# %% {"language": "html"}
# <style>
# div.input {
#     display:contents;
# }
# </style>

# %% [markdown] {"slideshow": {"slide_type": "slide"}}
# # Application

# %% [markdown] {"slideshow": {"slide_type": "slide"}}
# __Data__
# ![](../media/05-heartbeat-kaggle.png)

# %% {"slideshow": {"slide_type": "skip"}}
# # ! kaggle datasets download -d kinguistics/heartbeat-sounds -p ../data/raw

# %% {"slideshow": {"slide_type": "skip"}}
import os
import glob
import zipfile
```

```
import sys
from sklearn.datasets import load_iris
import pandas as pd
import numpy as np
import holoviews as hv
import hvplot.pandas
from toolz.curried import *
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import train_test_split, KFold
from scipy.io import wavfile  # get the api
from scipy.fftpack import fft, irfft
from scipy.signal import find_peaks, resample_poly
from scipy.spatial import procrustes
from scipy.stats import iqr
import panel as pn
import param
from random import sample


sys.path.append("../")
hv.extension("bokeh")

# %% {"slideshow": {"slide_type": "skip"}}
from super_spirals.neural_network import VAE

# %% [markdown] {"slideshow": {"slide_type": "skip"}}
# ### Load Data

# %% {"slideshow": {"slide_type": "skip"}}
data_path = os.path.join("..", "data", "raw", "heatbeat-sounds")

if not os.path.exists(data_path):
    with zipfile.ZipFile(
        os.path.join("..", "data", "raw", "heartbeat-sounds.zip"), "r"
    ) as zip_ref:
        zip_ref.extractall(data_path)

# %% {"slideshow": {"slide_type": "skip"}}
heartbeat_path = os.path.join(data_path, "set_b")

# %% {"slideshow": {"slide_type": "skip"}}
files = pipe(
    os.listdir(heartbeat_path),
    map(str),
    map(lambda f: os.path.join(heartbeat_path, f)),
    list,
)
```

```

# %% {"slideshow": {"slide_type": "skip"}}
set_a = pipe(data_path, lambda f: os.path.join(f, "set_a.csv"), pd.
    read_csv)

# %% {"slideshow": {"slide_type": "skip"}}
set_b = pipe(data_path, lambda f: os.path.join(f, "set_b.csv"), pd.
    read_csv)

# %% [markdown] {"slideshow": {"slide_type": "slide"}}
# __Preprocessing__
# Data $\rightarrow$ Standardize $\rightarrow$ Clip $\rightarrow$ Split
    into single beats $\rightarrow$ Resample resolution $\rightarrow$
    Fourier Transform

# %% [markdown] {"slideshow": {"slide_type": "skip"}}
# High pass filter

# %% {"slideshow": {"slide_type": "skip"}}
filter_signal = lambda a: (
    pipe(
        a,
        partial(find_peaks, distance=1000),
        get(0),
        lambda x: pipe(a[x], np.median),
        lambda x: np.clip(a, -x, x),
    )
)

# %% {"slideshow": {"slide_type": "skip"}}
get_signal = lambda f: pipe(f, wavfile.read, get(1))

# %% {"slideshow": {"slide_type": "slide"}}
a = pipe(files[0], get_signal)

b = pipe(files[0], get_signal, filter_signal)

c = pipe(
    b,
    partial(find_peaks, distance=1000),
    get(0),
    lambda x: np.vstack((np.arange(b.shape[0]), b)).T[x, :],
)

(
    hv.Curve(a) * hv.Curve(b).opts(color="orange") * hv.Scatter(c).opts
    (color="green")
).opts(width=600)

```

```
# %% [markdown] {"slideshow": {"slide_type": "skip"}}
# split into individual heartbeats, resample to equal length and get Fourier Transform

# %% {"slideshow": {"slide_type": "skip"}}
def explode(b, components=10):
    return pipe(
        b,
        partial(find_peaks, distance=1000),
        get(0),
        sliding_window(2),
        map(lambda x: b[x[0] : x[1]]),
        map(lambda x: (x) / (np.quantile(np.abs(x), 0.9))),
        map(
            lambda x: x[
                np.round(np.linspace(0, x.shape[0] - 1, num=1000)).astype(np.int)
            ]
        ),
        map(partial(fft, n=components)),
        map(lambda x: np.hstack((np.real(x))).reshape(-1)),
        list,
    )


# %% {"slideshow": {"slide_type": "skip"}}
def get_explotion(f, components=25):
    try:
        return pipe(f, wavfile.read, get(1), partial(explode, components=components))

    except:
        return [np.zeros(int(round(components))) * np.nan]


# %% {"slideshow": {"slide_type": "skip"}}
frequencies_a = set_a.fname.apply(lambda f: os.path.join(data_path, f)).apply(
    get_explotion
)

# %% {"slideshow": {"slide_type": "skip"}}
frequencies_b = pipe(files, pd.Series).apply(get_explotion)

# %% [markdown] {"slideshow": {"slide_type": "skip"}}
# Merge data with frequencies

# %% {"slideshow": {"slide_type": "skip"}}
```

```
set_a_freq = (
    set_a.assign(frequencies=frequencies_a)
    .explode("frequencies")
    .reset_index(drop=True)
    .assign(label=lambda d: d.label.fillna("None"))
    .where(lambda d: ~d.label.str.startswith("None"))
    .dropna(how="all")
)

# %% {"slideshow": {"slide_type": "skip"}}
X_a = set_a_freq.frequencies.apply(pd.Series)

# %% {"slideshow": {"slide_type": "skip"}}
set_a_filtered = set_a_freq.loc[~X_a.isna().all(axis=1), :]

# %% {"slideshow": {"slide_type": "skip"}}
X_b = (
    pd.DataFrame({"frequencies": frequencies_b})
    .explode("frequencies")
    .reset_index(drop=True)
    .frequencies.apply(pd.Series)
)

# %% [markdown] {"slideshow": {"slide_type": "skip"}}
# ### Train

# %% {"slideshow": {"slide_type": "skip"}}
pipeline = make_pipeline(
    StandardScaler(),
    PCA(whiten=True),
    VAE(
        hidden_layer_sizes=(15, 10, 2),
        max_iter=1000,
        divergence_weight=10,
        activation="tanh",
    ),
)

# %% {"slideshow": {"slide_type": "skip"}}
pipeline.fit(X_b.dropna())

# %% [markdown] {"slideshow": {"slide_type": "slide"}}
# __Model__

# %% {"slideshow": {"slide_type": "-"}}
pipeline.named_steps["vae"].encoder.summary()

# %% [markdown] {"slideshow": {"slide_type": "slide"}}
# ### Dashboard
```

```

# %% {"slideshow": {"slide_type": "skip"}}
latent_a = pipeline.transform(X_a.dropna())  # .loc[~X_a.isna().all(
    axis=1),:])

# %% {"slideshow": {"slide_type": "skip"}}
latent_a_df = (
    pd.DataFrame(latent_a, columns=["Component 1", "Component 2"])
    .assign(label=set_a_filtered.label.fillna("None"))
    .reset_index()
    .groupby("label")
    .apply(lambda d: d.sample(250, replace=False))
    .reset_index(drop=True)
    .set_index("index")
)

# %% {"slideshow": {"slide_type": "skip"}}
latent_a_df.head()

# %% {"slideshow": {"slide_type": "skip"}}
print(
    latent_a_df.loc[:, ["Component 1", "label"]]
    .groupby("label")
    .describe()
    .iloc[:, 1:3]
    .to_latex()
)

# %% {"slideshow": {"slide_type": "skip"}}
print(
    latent_a_df.loc[:, ["Component 2", "label"]]
    .groupby("label")
    .describe()
    .iloc[:, 1:3]
    .to_latex()
)

# %% {"slideshow": {"slide_type": "skip"}}
clips = set_a_filtered.loc[latent_a_df.index, "fname"].to_list()


# %% {"slideshow": {"slide_type": "skip"}}
class Dashboard(param.Parameterized):
    files = pn.widgets.Select(name="Audio Clip", value=clips[0],
    options=clips)

    @pn.depends("files.value")
    def update(self, index):
        if index:
```

```
            self.files.value = clips[index[0]]
        wav_file = pipe(self.files.value, lambda f: os.path.join(data_path, f))

        data = pipe(wav_file, wavfile.read, get(1))

        time = pipe(data, lambda x: x[::400] / np.max(np.abs(x)), hv.Curve).opts(
            width=400, xlabel="time", ylabel="waveform", height=300
        )

        frequency = pipe(
            data,
            partial(fft, n=1000),
            np.real,
            lambda x: x / np.max(np.abs(x)),
            hv.Curve,
        ).opts(xlabel="frequency", ylabel="aplitude", width=400, height=300)

        return time + frequency

    @pn.depends("files.value")
    def view(self):

        latent = latent_a_df.hvplot.scatter(
            x="Component 1",
            y="Component 2",
            color="label",
            title="Latent Space of Heartbeat FFT",
            width=800,
            size=10,
            height=300,
            tools=["tap"],
        )

        stream = hv.streams.Selection1D(source=latent)

        reg = hv.DynamicMap(self.update, kdims=[], streams=[stream])

        audio = pn.widgets.Audio(
            name="Audio",
            value=pipe(self.files.value, lambda f: os.path.join(data_path, f)),
        )

        return pn.Column(latent, reg, audio)


```

```
# %% {"slideshow": {"slide_type": "skip"}}
d = Dashboard()

# %% {"slideshow": {"slide_type": "-"}}
pn.Column(d.files, d.view)

# %% {"slideshow": {"slide_type": "skip"}}
```

### Library Code

```
# wavfile.py (Enhanced)
#
# Mod by X-Raym
# Date: 20181906_1222
# * corrected loops
# * unsupported chunk read and write
# * LIST-INFO support
# * renamed variables to avoid conflict with python native functions
# * correct bytes error
# * correct write function
#
# URL: https://gist.github.com/josephernest/3f22c5ed5dabf1815f16efa8fa53d476
# Source: scipy/io/wavfile.py
#
# Mod by Joseph Basquin
# Date: 20180430_2335
# * read: also returns bitrate, cue markers + cue marker labels (sorted), loops, pitch
# * read: 24 bit & 32 bit IEEE files support (inspired from wavio_weckesser.py from Warren Weckesser)
# * read: added normalized (default False) that returns everything as float in [-1, 1]
# * read: added forcestereo that returns a 2-dimensional array even if input is mono
#
# * write: can write cue markers, cue marker labels, loops, pitch
# * write: 24 bit support
# * write: can write from a float normalized in [-1, 1]
# * write: 20180430_2335: bug fixed when size of data chunk is odd (previously, metadata could become unreadable because of this)
#
# * removed RIFX support (big-endian) (never seen one in 10+ years of audio production/audio programming), only RIFF (little-endian) are supported
# * removed read(..., mmap)
#
#
# Test:
```

```
# ..\wav\_____wavfile_demo.py


"""
Module to read / write wav files using numpy arrays

Functions
---------
`read`: Return the sample rate (in samples/sec) and data from a WAV
    file.

`write`: Write a numpy array as a WAV file.

"""
# from __future__ import division, print_function, absolute_import

import numpy
import struct
import warnings
import collections

# from operator import itemgetter


class WavFileWarning(UserWarning):
    pass


_ieee = False

# assumes file pointer is immediately
#  after the 'fmt ' id
def _read_fmt_chunk(fid):
    res = struct.unpack("<ihHIIHH", fid.read(20))
    size, comp, noc, rate, sbytes, ba, bits = res
    if comp != 1 or size > 16:
        if comp == 3:
            global _ieee
            _ieee = True
            # warnings.warn("IEEE format not supported", WavFileWarning
    )
        else:
            warnings.warn("Unfamiliar format bytes", WavFileWarning)
        if size > 16:
            fid.read(size - 16)
    return size, comp, noc, rate, sbytes, ba, bits


# assumes file pointer is immediately
```

```python
#   after the 'data' id
def _read_data_chunk(fid, noc, bits, normalized=False):
    size = struct.unpack("<i", fid.read(4))[0]

    if bits == 8 or bits == 24:
        dtype = "u1"
        bytes_val = 1
    else:
        bytes_val = bits // 8
        dtype = "<i%d" % bytes_val

    if bits == 32 and _ieee:
        dtype = "float32"

    data = numpy.fromfile(fid, dtype=dtype, count=size // bytes_val)

    if bits == 24:
        a = numpy.empty((len(data) // 3, 4), dtype="u1")
        a[:, :3] = data.reshape((-1, 3))
        a[:, 3:] = (a[:, 3 - 1 : 3] >> 7) * 255
        data = a.view("<i4").reshape(a.shape[:-1])

    if noc > 1:
        data = data.reshape(-1, noc)

    if bool(
        size & 1
    ):  # if odd number of bytes, move 1 byte further (data chunk is word-aligned)
        fid.seek(1, 1)

    if normalized:
        if bits == 8 or bits == 16 or bits == 24:
            normfactor = 2 ** (bits - 1)
        data = numpy.float32(data) * 1.0 / normfactor

    return data


def _skip_unknown_chunk(fid):
    data = fid.read(4)
    size = struct.unpack("<i", data)[0]
    if bool(
        size & 1
    ):  # if odd number of bytes, move 1 byte further (data chunk is word-aligned)
        size += 1
    fid.seek(size, 1)

```

```

def _read_unknown_chunk(fid, name):
    data = fid.read(4)
    size = struct.unpack("<i", data)[0]
    # string = fid.read(size).rstrip(bytes('\x00', 'UTF-8')).decode("utf-8")
    offset = 0
    if bool(
        size & 1
    ):  # if odd number of bytes, move 1 byte further (data chunk is word-aligned)
        offset = 1
    string = fid.read(size)
    fid.seek(offset, 1)
    return string


def _read_riff_chunk(fid):
    str1 = fid.read(4)
    if str1 != b"RIFF":
        raise ValueError("Not a WAV file.")
    fsize = struct.unpack("<I", fid.read(4))[0] + 8
    str2 = fid.read(4)
    if str2 != b"WAVE":
        raise ValueError("Not a WAV file.")
    return fsize


def read_wav(
    file,
    readmarkers=False,
    readmarkerlabels=False,
    readmarkerslist=False,
    readloops=False,
    readpitch=False,
    normalized=False,
    forcestereo=False,
    log=True,
    readlistinfo=True,
    readunsupported=True,
):
    """
    Return the sample rate (in samples/sec) and data from a WAV file

    Parameters
    ----------
    file : file
        Input wav file.

```

```
    Returns
    -------
    rate : int
        Sample rate of wav file
    data : numpy array
        Data read from wav file

    Notes
    -----

    * The file can be an open file or a filename.

    * The returned sample rate is a Python integer
    * The data is returned as a numpy array with a
      data-type determined from the file.

    """
    if hasattr(file, "read"):
        fid = file
    else:
        fid = open(file, "rb")

    fsize = _read_riff_chunk(fid)
    noc = 1
    bits = 8
    # _cue = []
    # _cuelabels = []
    _markersdict = collections.defaultdict(lambda: {"position": -1, "
label": ""})
    unsupported = {}
    loops = []
    list_info_index = [
        "IARL",
        "IART",
        "ICMS",
        "ICMT",
        "ICOP",
        "ICRD",
        "IENG",
        "IGNR",
        "IKEY",
        "IMED",
        "INAM",
        "IPRD",
        "ISBJ",
        "ISFT",
        "ISRC",
        "ISRF",
        "ITCH",
```

```
    ]
    info = {}
    pitch = 0.0
    while fid.tell() < fsize:
        # read the next chunk
        chunk_id = fid.read(4)
        chunk_id_str = chunk_id.decode("utf-8")
        if chunk_id == b"fmt ":
            size, comp, noc, rate, sbytes, ba, bits = _read_fmt_chunk(fid)
        elif chunk_id == b"data":
            data = _read_data_chunk(fid, noc, bits, normalized)
        elif chunk_id == b"cue ":
            str1 = fid.read(8)
            size, numcue = struct.unpack("<ii", str1)
            for c in range(numcue):
                str1 = fid.read(24)
                idx, position, datachunkid, chunkstart, blockstart, sampleoffset = struct.unpack(
                    "<iiiiii", str1
                )
                # _cue.append(position)
                _markersdict[idx][
                    "position"
                ] = position  # needed to match labels and markers

        elif chunk_id == b"LIST":
            str1 = fid.read(8)
            size, datatype = struct.unpack("<ii", str1)
        elif (
            chunk_id_str in list_info_index
        ):  # see http://www.pjb.com.au/midi/sfspec21.html#i5
            s = _read_unknown_chunk(fid, chunk_id_str)
            info[chunk_id_str] = s.decode("UTF-8")
        elif chunk_id == b"labl":
            str1 = fid.read(8)
            size, idx = struct.unpack("<ii", str1)
            size = size + (
                size % 2
            )  # the size should be even, see WAV specfication, e.g. 16=>16, 23=>24
            label = fid.read(size - 4).rstrip(
                bytes("\x00", "UTF-8")
            )  # remove the trailing null characters
            # _cuelabels.append(label)
            _markersdict[idx]["label"] = label  # needed to match labels and markers

        elif chunk_id == b"smpl":
```

```
            str1 = fid.read(40)
            size, manuf, prod, sampleperiod, midiunitynote, midipitchfraction, smptefmt, smpteoffs, numsampleloops, samplerdata = struct.unpack(
                "<iiiiiIiiii", str1
            )
            cents = midipitchfraction * 1.0 / (2 ** 32 - 1)
            pitch = 440.0 * 2 ** ((midiunitynote + cents - 69.0) / 12)
            for i in range(numsampleloops):
                str1 = fid.read(24)
                cuepointid, datatype, start, end, fraction, playcount = struct.unpack(
                    "<iiiiii", str1
                )
                loops.append(
                    {
                        "cuepointid": cuepointid,
                        "datatype": datatype,
                        "start": start,
                        "end": end,
                        "fraction": fraction,
                        "playcount": playcount,
                    }
                )
        else:
            if log:
                warnings.warn("Chunk " + str(chunk_id) + " skipped", WavFileWarning)
            if readunsupported:
                # print( chunk_id.decode("utf-8")  + " unsupported")
                unsupported[chunk_id] = _read_unknown_chunk(fid, chunk_id_str)
            else:
                _skip_unknown_chunk(fid)
    fid.close()

    if data.ndim == 1 and forcestereo:
        data = numpy.column_stack((data, data))

    _markerslist = sorted(
        [_markersdict[l] for l in _markersdict], key=lambda k: k["position"]
    )  # sort by position
    _cue = [m["position"] for m in _markerslist]
    _cuelabels = [m["label"] for m in _markerslist]

    return (
        (rate, data, bits)
        + ((_cue,) if readmarkers else ())
```

```python
        + ((_cuelabels,) if readmarkerlabels else ())
        + ((_markerslist,) if readmarkerslist else ())
        + ((loops,) if readloops else ())
        + ((pitch,) if readpitch else ())
        + ((info,) if readlistinfo else ())
        + ((unsupported,) if readunsupported else ())
    )


def write_wav(
    filename,
    rate,
    data,
    bitrate=None,
    markers=None,
    loops=None,
    pitch=None,
    normalized=False,
    infos=None,
    unsupported=None,
):
    """
    Write a numpy array as a WAV file

    Parameters
    ----------
    filename : file
        The name of the file to write (will be over-written).
    rate : int
        The sample rate (in samples/sec).
    data : ndarray
        A 1-D or 2-D numpy array of integer data-type.

    Notes
    -----
    * Writes a simple uncompressed WAV file.
    * The bits-per-sample will be determined by the data-type.
    * To write multiple-channels, use a 2-D array of shape
      (Nsamples, Nchannels).

    """

    # normalization and 24-bit handling
    if (
        bitrate == 24
    ):  # special handling of 24 bit wav, because there is no numpy.int24...
        if normalized:
            data[data > 1.0] = 1.0
```

```
            data[data < -1.0] = -1.0
            a32 = numpy.asarray(data * (2 ** 23 - 1), dtype=numpy.int32)
        else:
            a32 = numpy.asarray(data, dtype=numpy.int32)
        if a32.ndim == 1:
            a32.shape = a32.shape + (1,)  # Convert to a 2D array with a single column.
        a8 = (
            a32.reshape(a32.shape + (1,)) >> numpy.array([0, 8, 16])
        ) & 255  # By shifting first 0 bits, then 8, then 16, the resulting output is 24 bit little-endian.
        data = a8.astype(numpy.uint8)
    else:
        if normalized:  # default to 32 bit int
            data[data > 1.0] = 1.0
            data[data < -1.0] = -1.0
            data = numpy.asarray(data * (2 ** 31 - 1), dtype=numpy.int32)

    fid = open(filename, "wb")
    fid.write(b"RIFF")
    fid.write(b"\x00\x00\x00\x00")
    fid.write(b"WAVE")

    # fmt chunk
    fid.write(b"fmt ")
    if data.ndim == 1:
        noc = 1
    else:
        noc = data.shape[1]
    bits = data.dtype.itemsize * 8 if bitrate != 24 else 24
    sbytes = rate * (bits // 8) * noc
    ba = noc * (bits // 8)
    fid.write(struct.pack("<ihHIIHH", 16, 1, noc, rate, sbytes, ba, bits))

    if unsupported:
        for key, val in unsupported.items():
            if len(key) % 2 == 1:
                key += b"\x00"
            if len(val) % 2 == 1:
                val += b"\x00"
            info = key
            size = len(val)  # because \x00
            info = struct.pack("<i", size)
            info += val
            fid.write(key)
            size = len(info)
```

```
            fid.write(info)

    # cue chunk
    if markers:  # != None and != []
        if isinstance(
            markers[0], dict
        ):  # then we have [{'position': 100, 'label': 'marker1'}, ...]
            labels = [m["label"] for m in markers]
            markers = [m["position"] for m in markers]
        else:
            labels = ["" for m in markers]

        fid.write(b"cue ")
        size = 4 + len(markers) * 24
        fid.write(struct.pack("<ii", size, len(markers)))
        for i, c in enumerate(markers):
            s = struct.pack(
                "<iiiiii", i + 1, c, 1635017060, 0, 0, c
            )  # 1635017060 is struct.unpack('<i',b'data')
            fid.write(s)

        lbls = b""
        for i, lbl in enumerate(labels):
            lbls += b"labl"
            label = lbl + (b"\x00" if len(lbl) % 2 == 1 else b"\x00\x00
")
            size = len(lbl) + 1 + 4  # because \x00
            lbls += struct.pack("<ii", size, i + 1)
            lbls += label

        fid.write(b"LIST")
        size = len(lbls) + 4
        fid.write(struct.pack("<i", size))
        fid.write(
            b"adtl"
        )  # https://web.archive.org/web/20141226210234/http://www.
sonicspot.com/guide/wavefiles.html#list
        fid.write(lbls)

    # smpl chunk
    if loops or pitch:
        if not loops:
            loops = []
        if pitch:
            midiunitynote = 12 * numpy.log2(pitch * 1.0 / 440.0) + 69
            midipitchfraction = int(
                (midiunitynote - int(midiunitynote)) * (2 ** 32 - 1)
            )
            midiunitynote = int(midiunitynote)
```

```
            # print(midipitchfraction, midiunitynote)
        else:
            midiunitynote = 0
            midipitchfraction = 0
        fid.write(b"smpl")
        size = 36 + len(loops) * 24
        sampleperiod = int(1000000000.0 / rate)

        fid.write(
            struct.pack(
                "<iiiiiIiiii",
                size,
                0,
                0,
                sampleperiod,
                midiunitynote,
                midipitchfraction,
                0,
                0,
                len(loops),
                0,
            )
        )
        for i, loop in enumerate(loops):
            fid.write(
                struct.pack(
                    "<iiiiii",
                    loop["cuepointid"],
                    loop["datatype"],
                    loop["start"],
                    loop["end"],
                    loop["fraction"],
                    loop["playcount"],
                )
            )

    # data chunks
    fid.write(b"data")
    fid.write(struct.pack("<i", data.nbytes))
    import sys

    if data.dtype.byteorder == ">" or (
        data.dtype.byteorder == "=" and sys.byteorder == "big"
    ):
        data = data.byteswap()

    data.tofile(fid)

    if (
```

```
        data.nbytes % 2 == 1
    ):  # add an extra padding byte if data.nbytes is odd: https://web.archive.org/web/20141226210234/http://www.sonicspot.com/guide/wavefiles.html#data
        fid.write("\x00")

    # This need to be made modular !
    if infos:
        info = b""
        for key, val in infos.items():
            key = bytes(key, "UTF-8")
            val = bytes(val, "UTF-8")
            # val += b'\x00' # Note: Fix windows display error. Is this valid ?
            size = len(val)  # because \x00
            if len(val) % 2 == 1:
                val += b"\x00"
            info += key
            info += struct.pack("<i", size)
            info += val
        # info += b'\x00'
        if len(info) % 2 == 1:
            info += b"\x00"
        fid.write(b"LIST")
        size = len(info) + 4
        fid.write(struct.pack("<i", size))
        fid.write(
            b"INFO"
        )  # https://web.archive.org/web/20141226210234/http://www.sonicspot.com/guide/wavefiles.html#list
        fid.write(info)

    # Determine file size and place it in correct
    #  position at start of the file.
    size = fid.tell()
    fid.seek(4)
    fid.write(struct.pack("<i", size - 8))
    fid.close()
    return "success"
```

```
import pandas as pd
import numpy as np
import holoviews as hv
import hvplot.pandas
from typing import Dict
from toolz.curried import *
from sklearn.model_selection import train_test_split
from sklearn.metrics import silhouette_score
from sklearn.base import TransformerMixin
```

```


def get_latent_space(
    model, X_train, X_test, y_train, y_test, tag, labels, feature_names
):

    latent = pipe(
        X_test,
        model.transform,
        partial(pd.DataFrame, columns=["Component 1", "Component 2"]),
    )

    return (
        pd.concat([latent], axis=1)
        .assign(label=y_test)
        .assign(label=lambda d: d.label.replace(labels))
        .assign(tag=tag)
    )


def reconstruction_benchmark(
    dataset: Dict[str, np.ndarray], models: Dict[str, TransformerMixin
    ], label: str
):
    """
    """
    data = dataset

    if hasattr(data, "target_names"):
        labels = dict(enumerate(data.target_names))
    else:
        labels = pipe(data.target, np.unique, map(str), list, np.array)
    .astype(str)

    if hasattr(data, "feature_names"):
        names = dict(enumerate(data.feature_names))
    else:
        names = pipe(range(data.data.shape[1]), map(str), list, np.
    array).astype(str)

    X_train, X_test, y_train, y_test = train_test_split(
        data.data, data.target, test_size=0.33, random_state=42
    )

    for m in models.values():
        m.fit(X_train)

    latent = pd.concat(
        [
```

```
            get_latent_space(m, X_train, X_test, y_train, y_test, t,
    labels, names)
            for t, m in models.items()
        ]
    )

    reconstruction_loss = pd.DataFrame(
        {
            t: pipe(
                X_train,
                m.transform,
                m.inverse_transform,
                lambda x: np.subtract(x, X_train),
                lambda x: np.power(x, 2),
                lambda x: np.array(x).flatten(),
                np.mean,
            )
            for t, m in models.items()
        },
        index=["Reconstruction Loss"],
    ).T

    silhouette = pd.DataFrame(
        {
            t: [silhouette_score(X=pipe(X_test, m.transform), labels=
    y_test)]
            for t, m in models.items()
        },
        index=["Silhoutte"],
    ).T

    return (
        latent,
        reconstruction_loss.hvplot.bar(title=f"{label}: Reconstruction
    Loss"),
        silhouette.hvplot.bar(title=f"{label}: Silhouette Scores"),
        latent.hvplot.scatter(
            x="Component 1", y="Component 2", color="label", groupby="
    tag", label=label
        )
        .layout()
        .cols(2),
    )
```

```
from sklearn.base import BaseEstimator, TransformerMixin
import numpy as np
from toolz.curried import *
from keras.layers import Lambda, Input, Dense
from keras.models import Model
```

```
from keras.datasets import mnist
from keras.losses import mse, binary_crossentropy
from keras.utils import plot_model
from keras import backend as K
from keras.regularizers import l2


def sampling(args):
    """Reparameterization trick by sampling from an isotropic unit
    Gaussian.

    # Arguments
        args (tensor): mean and log of variance of Q(z|X)

    # Returns
        z (tensor): sampled latent vector
    """

    z_mean, z_log_var = args
    batch = K.shape(z_mean)[0]
    dim = K.int_shape(z_mean)[1]
    # by default, random_normal has mean = 0 and std = 1.0
    epsilon = K.random_normal(shape=(batch, dim))
    return z_mean + K.exp(0.5 * z_log_var) * epsilon


class VAE(BaseEstimator, TransformerMixin):
    """Transform data using vae"""

    def __init__(
        self,
        hidden_layer_sizes: tuple = (25, 2),
        activation: str = "relu",
        solver: str = "adam",
        divergence_weight: float = 1,
        alpha: float = 0.0001,
        batch_size: str = "auto",
        learning_rate: str = "constant",
        learning_rate_init: float = 0.001,
        power_t: float = 0.5,
        max_iter: int = 200,
        shuffle: bool = True,
        random_state=None,
        tol: float = 0.0001,
        verbose: bool = False,
        warm_start: bool = False,
        momentum: float = 0.9,
        nesterovs_momentum: bool = True,
        early_stopping: bool = False,
```

```
        validation_fraction: float = 0.1,
        beta_1: float = 0.9,
        beta_2: float = 0.999,
        epsilon: float = 1e-08,
        n_iter_no_change=10,
    ):
        """
        """
        self.alpha = alpha
        self.regularizer = l2(alpha)

        self.hidden_layer_sizes = hidden_layer_sizes
        self.activation = activation
        self.max_iter = max_iter
        if solver == "auto":
            self.solver = "adam"
        else:
            self.solver = solver
        self.divergence_weight = divergence_weight
        self.model = None
        self.validation_fraction = validation_fraction

    def build_encoder_(self, layers: tuple):
        # VAE model = encoder + decoder
        # build encoder model
        input_shape, *encoder_shape, latent_dim = layers

        inputs = Input(shape=(input_shape,), name="encoder_input")
        transformations = pipe(
            encoder_shape,
            map(
                lambda d: Dense(
                    units=d,
                    kernel_regularizer=self.regularizer,
                    activation=self.activation,
                )
            ),
            lambda f: compose_left(*f),
        )

        x = pipe(inputs, transformations)

        z_mean = Dense(latent_dim, name="z_mean")(x)
        z_log_var = Dense(latent_dim, name="z_log_var")(x)

        # use reparameterization trick to push the sampling out as
    input
        z = Lambda(sampling, output_shape=(latent_dim,), name="z")([
    z_mean, z_log_var])
```

```
        # note that "output_shape" isn't necessary with the TensorFlow backend
        # instantiate encoder model
        encoder = Model(inputs, [z_mean, z_log_var, z], name="encoder")

        return inputs, encoder, z_mean, z_log_var

    def build_decoder_(self, layers: tuple):
        # build decoder model
        latent_shape, *decoder_layers, original_dim = layers

        latent_inputs = Input(shape=(latent_shape,), name="z_sampling")

        transformations = pipe(
            decoder_layers,
            map(
                lambda d: Dense(
                    units=d,
                    kernel_regularizer=self.regularizer,
                    activation=self.activation,
                )
            ),
            lambda f: compose_left(*f),
        )
        final_layer = Dense(original_dim, name="original_dim")

        outputs = pipe(latent_inputs, transformations, final_layer)

        # instantiate decoder model
        decoder = Model(latent_inputs, outputs, name="decoder")
        return decoder

    def build_model_(self, layers: tuple):
        """"""
        inputs, self.encoder, z_mean, z_log_var = self.build_encoder_(layers)
        self.decoder = self.build_decoder_(reversed(layers))

        outputs = pipe(inputs, self.encoder, get(2), self.decoder)
        vae = Model(inputs, outputs, name="vae_mlp")

        #         if args.mse:
        reconstruction_loss = mse(inputs, outputs)
        #         else:
        #             reconstruction_loss = binary_crossentropy(inputs,
        #                                                        outputs
)

```

```
        reconstruction_loss *= layers[0]
        kl_loss = 1 + z_log_var - K.square(z_mean) - K.exp(z_log_var)
        kl_loss = K.sum(kl_loss, axis=-1)
        kl_loss *= -0.5
        vae_loss = K.mean(reconstruction_loss + self.divergence_weight
    * kl_loss)

        vae.add_loss(vae_loss)
        vae.compile(optimizer=self.solver)

        return vae

    def fit(self, x: np.ndarray, y: np.ndarray = None):
        """
        """
        layers = x.shape[1], *self.hidden_layer_sizes

        #         if self.model is None:
        self.model = self.build_model_(layers)

        n_samples = x.shape[0]
        if self.solver == self.solver:
            self.batch_size = n_samples
        elif self.batch_size == self.solver:
            self.batch_size = min(200, n_samples)
        else:
            if self.batch_size < 1 or self.batch_size > n_samples:
                warnings.warn(
                    "Got `batch_size` less than 1 or larger than "
                    "sample size. It is going to be clipped"
                )
            self.batch_size = np.clip(self.batch_size, 1, n_samples)

        self.model.fit(
            x,
            epochs=self.max_iter,
            batch_size=self.batch_size,
            validation_split=self.validation_fraction,
        )

        #         vae.save_weights('vae_mlp_mnist.h5')

        return self

    def transform(self, X: np.ndarray):
        """
        """
        return pipe(self.encoder.predict(X), get(0))
```

```
    def sample(self, n: int):
        """
        """
        dim = self.hidden_layer_sizes[-1]
        N = np.random.multivariate_normal(np.zeros(dim), np.diag(np.ones(dim)), size=n)
        return self.decoder.predict(N)

    def inverse_transform(self, Xt: np.ndarray):
        """
        """
        return self.decoder.predict(Xt)

    def predict(self, X: np.ndarray):
        """
        """
        return self.model.predict(X)

    def score(self, X: np.ndarray, y: np.ndarray = None):
        """
        """
        return self.model.evaluate(X)
```

## Environment


```
name: super-spirals
channels:
  - bioconda
  - conda-forge
  - intel
  - defaults
dependencies:
  - _libgcc_mutex=0.1=main
  - absl-py=0.7.1=py36_0
  - appdirs=1.4.3=py_1
  - asn1crypto=0.24.0=py36_3
  - astor=0.8.0=py36_0
  - attrs=19.1.0=py_0
  - backcall=0.1.0=py36_2
  - backports=1.0=py36_9
  - black=19.3b0=py_0
  - bleach=2.1.3=py36_2
  - bokeh=1.3.4=py36_0
  - c-ares=1.15.0=h7b6447c_1001
  - ca-certificates=2019.9.11=hecc5488_0
  - certifi=2019.9.11=py36_0
  - cffi=1.11.5=py36_3
  - chardet=3.0.4=py36_3
  - click=7.0=py_0
  - cryptography=2.3=py36_2
  - cycler=0.10.0=py36_7
  - daal=2019.5=intel_281
  - daal4py=2019.5=py36ha68da19_2
  - decorator=4.3.0=py36_3
  - entrypoints=0.2.3=py36_2
  - fastdtw=0.2.0=py_1
  - fontconfig=2.13.1=h86ecdb6_1001
  - freetype=2.9.1=h8a8886c_1
  - gast=0.2.2=py36_0
  - get_terminal_size=1.0.0=py36_7
  - glob2=0.7=py_0
  - google-pasta=0.1.7=py_0
  - grpcio=1.23.0=py36he9ae1f9_0
  - h5py=2.8.0=py36h989c5e5_3
  - hdf5=1.10.2=2
  - holoviews=1.12.3=py_2
  - html5lib=1.0.1=py36_4
  - hvplot=0.4.0=py_1
  - icc_rt=2019.5=intel_281
  - icu=64.2=he1b5a44_1
  - idna=2.6=py36_3
  - impi_rt=2019.5=intel_281
```

```
- intel-openmp=2019.5=intel_281
- intelpython=2019.5=0
- ipykernel=4.6.1=py36_2
- ipython=6.3.1=py36_3
- ipython_genutils=0.2.0=py36_2
- jedi=0.12.0=py36_2
- jinja2=2.10.1=py_0
- joblib=0.13.2=py36_1
- jpeg=9b=h024ee3a_2
- json5=0.8.5=py_0
- jsonschema=2.6.0=py36_2
- jupyter_client=5.1.0=py36_5
- jupyter_core=4.4.0=py36_6
- jupyterlab=1.1.4=py_0
- jupyterlab_server=1.0.0=py_0
- jupytext=1.2.4=0
- kaggle=1.5.6=py36_0
- keras=2.2.4=0
- keras-applications=1.0.8=py_0
- keras-base=2.2.4=py36_0
- keras-preprocessing=1.1.0=py_1
- kiwisolver=1.0.1=py36_2
- libffi=3.2.1=11
- libgcc-ng=9.1.0=hdf63c60_0
- libiconv=1.15=h516909a_1005
- libpng=1.6.36=2
- libprotobuf=3.8.0=hd408876_0
- libsodium=1.0.16=3
- libstdcxx-ng=9.1.0=hdf63c60_0
- libtiff=4.0.10=h2733197_2
- libuuid=2.32.1=h14c3975_1000
- libxml2=2.9.9=hee79883_5
- markdown=3.1.1=py36_0
- markupsafe=1.0=py36_3
- matplotlib=3.1.1=py36_2
- mistune=0.8.3=py36_2
- mkl=2019.5=intel_281
- mkl-service=2.3.0=py36_0
- mkl_fft=1.0.14=py36ha68da19_1
- mkl_random=1.0.4=py36ha68da19_2
- nbconvert=5.2.1=py36_2
- nbformat=4.4.0=py36_2
- ncurses=6.1=he6710b0_1
- nodejs=10.13.0=he6710b0_0
- notebook=5.2.2=py36_1
- numexpr=2.6.9=py36_0
- numpy=1.17.0=py36ha68da19_13
- numpy-base=1.17.0=py36_13
- olefile=0.46=py36_0
```

```
- openssl=1.1.1c=h516909a_0
- packaging=19.1=py36_0
- pandas=0.25.0=py36_5
- pandocfilters=1.4.1=py36_2
- panel=0.6.2=h39e3cac_0
- param=1.9.1=py_0
- parso=0.2.0=py36_2
- path.py=11.0.1=py36_2
- pexpect=4.2.1=py36_4
- phantomjs=2.1.1=1
- pickleshare=0.7.4=py36_3
- pillow=6.1.0=py36h34e0f95_0
- pip=19.1.1=py36_0
- prompt_toolkit=1.0.15=py36_2
- protobuf=3.8.0=py36he6710b0_0
- ptyprocess=0.5.2=py36_2
- pycparser=2.18=py36_2
- pyct=0.4.6=py36_0
- pygments=2.2.0=py36_5
- pyopenssl=17.5.0=py36_2
- pyparsing=2.2.0=py36_2
- pysocks=1.6.7=py36_1
- python=3.6.8=h0371630_0
- python-dateutil=2.8.0=py36_0
- python-slugify=3.0.3=py_0
- pytz=2019.1=py36_0
- pyviz_comms=0.7.2=py_0
- pyyaml=5.1.1=py36_0
- pyzmq=16.0.2=py36_6
- readline=7.0=h7b6447c_5
- requests=2.20.1=py36_1
- scikit-learn=0.21.3=py36ha68da19_4
- scipy=1.3.1=py36ha68da19_2
- selenium=3.141.0=py36h7b6447c_0
- setuptools=41.0.1=py36_0
- simplegeneric=0.8.1=py36_7
- six=1.12.0=py36_0
- sqlite=3.28.0=0
- tbb=2019.8=intel_281
- tbb4py=2019.8=py36_intel_0
- tcl=8.6.4=24
- tensorboard=1.14.0=py36hf484d3e_0
- tensorflow=1.14.0=py36_0
- tensorflow-base=1.14.0=0
- tensorflow-estimator=1.14.0=py_0
- termcolor=1.1.0=py36_1
- terminado=0.8.1=py36_2
- testpath=0.3.1=py36_2
- text-unidecode=1.2=py_0
```

```
  - tk=8.6.8=hbc83047_0
  - toml=0.10.0=py_0
  - toolz=0.10.0=py_0
  - tornado=4.5.2=py36_5
  - tqdm=4.36.1=py_0
  - traitlets=4.3.2=py36_3
  - unidecode=1.1.1=py_0
  - urllib3=1.24.1=py36_2
  - wcwidth=0.1.7=py36_6
  - webencodings=0.5.1=py36_0
  - werkzeug=0.14.1=py36_0
  - wheel=0.31.0=py36_3
  - wrapt=1.11.2=py36h7b6447c_0
  - xz=5.2.4=5
  - yaml=0.1.7=2
  - zeromq=4.2.3=2
  - zip=3.0=0
  - zlib=1.2.11=5
  - zstd=1.3.7=h0b5b093_0
  - pip:
    - intel-tensorflow==1.14.0
prefix: /home/marcusskky/.conda/envs/super-spirals
```

# References

Aeberhard, Stefan, Danny Coomans, and Olivier De Vel (1994), "Comparative analysis of statistical pattern recognition methods in high dimensional settings." *Pattern Recognition*, 27, 1065–1077.

An, Jinwon and Sungzoon Cho (2015), "SNU Data Mining Center 2015-2 Special Lecture on IE Variational Autoencoder based Anomaly Detection using Reconstruction Probability."

Bengio, Yoshua, Aaron Courville, and Pascal Vincent (2013), "Representation learning: A review and new perspectives." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35, 1798–1828.

Bengio, Yoshua, Pascal Lamblin, Dan Popovici, and Hugo Larochelle (2007), "Greedy Layer-Wise Training of Deep Networks Yoshua." *In Advances in neural information processing system.*

Bentley, P., G. Nordehn, M. Coimbra, and S. Mannor (????), "The PASCAL Classifying Heart Sounds Challenge 2011 (CHSC2011) Results." http://www.peterjbentley.com/heartchallenge/index.html.

Fisher, R. A. (1936), "The use of multiple measurements in taxonomic problems." *Annals of Eugenics*, 7, 179–188.

Goodfellow, Ian, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio (2014), "Generative Adversarial Nets." iii, URL `https://papers.nips.cc/paper/5423-generative-adversarial-nets{\%}0Ahttp://doi.wiley.com/10.1002/9781118472507.fmatter{\%}0Ahttp://linkinghub.elsevier.com/retrieve/pii/B9780408001090500018`.

Kingma, Diederik P and Max Welling (2013), "Auto-Encoding Variational Bayes." *arXiv e-prints*, arXiv:1312.6114.

Street, W. N., W. H. Wolberg, and O. L. Mangasarian (1993). *Biomedical Image Processing and Biomedical Visualization.*

Tolstikhin, Ilya, Olivier Bousquet, Sylvain Gelly, and Bernhard Schoelkopf (2017), "Wasserstein auto-encoders." *arXiv preprint arXiv:1711.01558.*

Zhao, Shengjia, Jiaming Song, and Stefano Ermon (2017), "Infovae: Information maximizing variational autoencoders." *arXiv preprint arXiv:1706.02262.*