# Domain agnostic feature engineering using deep learning and Bayesian inference: Automating the automation

* Hamaad Shah
* Jeremy Achin

We will explore the use of autoencoders for automatic feature engineering. The idea is to automatically learn a set of features from raw data that can be useful in supervised learning tasks such as in computer vision and insurance.

## Computer Vision

We will use the MNIST dataset for this purpose where the raw data is a 2 dimensional tensor of pixel intensities per image. The image is our unit of analysis: We will predict the probability of each class for each image. This is a multiclass classification task and we will use the one against all AUROC score and accuracy score to assess model performance on the test fold.

## Insurance

We will use a synthetic dataset where the raw data is a 2 dimensional tensor of historical policy level information per policy-period combination: Per unit this will be , i.e., 4 historical time periods and 3 transactions types. The policy-period combination is our unit of analysis: We will predict the probability of loss for time period 5 in the future - think of this as a potential renewal of the policy for which we need to predict whether it would make a loss for us or not hence affecting whether we decided to renew the policy and / or adjust the renewal premium to take into account the additional risk. This is a binary class classification task and we will use the AUROC score and accuracy score to assess model performance.

# Author: Hamaad Shah.  
# Code repo: https://github.com/hamaadshah/autoencoders  
  
import os  
import math  
import sys  
import importlib  
  
import numpy as np  
  
import pandas as pd  
  
from sklearn import linear\_model  
from sklearn.pipeline import Pipeline  
from sklearn.preprocessing import MinMaxScaler, LabelBinarizer, RobustScaler, StandardScaler  
from sklearn.metrics import roc\_auc\_score  
from sklearn.model\_selection import train\_test\_split  
from sklearn.decomposition import PCA  
  
from scipy.stats import norm  
  
import keras  
from keras import backend as bkend  
from keras.datasets import cifar10, mnist  
from keras.layers import Dense, BatchNormalization, Dropout, Flatten, convolutional, pooling  
from keras import metrics  
  
from autoencoders\_keras.get\_session import get\_session  
import keras.backend.tensorflow\_backend as KTF  
KTF.set\_session(get\_session(gpu\_fraction=0.75, allow\_soft\_placement=True, log\_device\_placement=False))  
  
import tensorflow as tf  
from tensorflow.python.client import device\_lib  
  
from plotnine import \*  
  
import matplotlib.pyplot as plt  
  
from autoencoders\_keras.vanilla\_autoencoder import VanillaAutoencoder  
from autoencoders\_keras.convolutional\_autoencoder import ConvolutionalAutoencoder  
from autoencoders\_keras.convolutional2D\_autoencoder import Convolutional2DAutoencoder  
from autoencoders\_keras.seq2seq\_autoencoder import Seq2SeqAutoencoder  
from autoencoders\_keras.variational\_autoencoder import VariationalAutoencoder  
  
%matplotlib inline  
  
np.set\_printoptions(suppress=True)  
  
os.environ["KERAS\_BACKEND"] = "tensorflow"  
importlib.reload(bkend)  
  
print(device\_lib.list\_local\_devices())  
  
mnist = mnist.load\_data()  
(X\_train, y\_train), (X\_test, y\_test) = mnist  
X\_train = np.reshape(X\_train, [X\_train.shape[0], X\_train.shape[1] \* X\_train.shape[1]])  
X\_test = np.reshape(X\_test, [X\_test.shape[0], X\_test.shape[1] \* X\_test.shape[1]])  
y\_train = y\_train.ravel()  
y\_test = y\_test.ravel()  
X\_train = X\_train.astype("float32")  
X\_test = X\_test.astype("float32")  
X\_train /= 255.0  
X\_test /= 255.0

## Scikit-learn

We will use the Python machine learning library scikit-learn for data transformation and the classification task. Note that we will code the autoencoders as scikit-learn transformers such that they can be readily used by scikit-learn pipelines.

scaler\_classifier = MinMaxScaler(feature\_range=(0.0, 1.0))  
logistic = linear\_model.LogisticRegression(random\_state=666)  
lb = LabelBinarizer()  
lb = lb.fit(y\_train.reshape(y\_train.shape[0], 1))

## MNIST: No Autoencoders

We run the MNIST dataset without using an autoencoder. The 2 dimensional tensor of pixel intensities per image for MNIST images are of dimension . We reshape them as a 1 dimensional tensor of dimension per image. Therefore we have 784, i.e., , features for this supervised learning task per image.

### Results

* The AUROC score for the MNIST classification task without autoencoders: 99.002648%.
* The accuracy score for the MNIST classification task without autoencoders: 92.000000%.

pipe\_base = Pipeline(steps=[("scaler\_classifier", scaler\_classifier),  
 ("classifier", logistic)])  
pipe\_base = pipe\_base.fit(X\_train, y\_train)  
  
auroc\_base = roc\_auc\_score(lb.transform(y\_test.reshape(y\_test.shape[0], 1)),  
 pipe\_base.predict\_proba(X\_test),   
 average="weighted")  
  
acc\_base = pipe\_base.score(X\_test, y\_test)  
  
print("The AUROC score for the MNIST classification task without autoencoders: %.6f%%." % (auroc\_base \* 100))  
print("The accuracy score for the MNIST classification task without autoencoders: %.6f%%." % (acc\_base \* 100))

## MNIST: PCA

We use a PCA filter that picks the number of components that explain of the variation.

### Results

* The AUROC score for the MNIST classification task with PCA: 98.966705%.
* The accuracy score for the MNIST classification task with PCA: 91.430000%.

pipe\_pca = Pipeline(steps=[("PCA", PCA(n\_components=0.99)),  
 ("scaler\_classifier", scaler\_classifier),  
 ("classifier", logistic)])  
pipe\_pca = pipe\_base.fit(X\_train, y\_train)  
  
auroc\_pca = roc\_auc\_score(lb.transform(y\_test.reshape(y\_test.shape[0], 1)),  
 pipe\_pca.predict\_proba(X\_test),   
 average="weighted")  
  
acc\_pca = pipe\_pca.score(X\_test, y\_test)  
  
print("The AUROC score for the MNIST classification task with PCA: %.6f%%." % (auroc\_pca \* 100))  
print("The accuracy score for the MNIST classification task with PCA: %.6f%%." % (acc\_pca \* 100))

## MNIST: Vanilla Autoencoders

An autoencoder is an unsupervised learning technique where the objective is to learn a set of features that can be used to reconstruct the input data.

Our input data is . An encoder function maps this to a set of features such that . A decoder function uses the set of features to reconstruct the input data such that .

Lets denote the reconstructed data as . The goal is to learn the encoding and decoding functions such that we minimize the difference between the input data and the reconstructed data. An example for an objective function for this task can be the Mean Squared Error (MSE) such that .

We learn the encoding and decoding functions by minimizing the MSE using the parameters that define the encoding and decoding functions: The gradient of the MSE with respect to the parameters are calculated using the chain rule, i.e., backpropagation, and used to update the parameters via an optimization algorithm such as Stochastic Gradient Descent (SGD).

Lets assume we have a single layer autoencoder using the Exponential Linear Unit (ELU) activation function, batch normalization, dropout and the Adaptive Moment (Adam) optimization algorithm. is the batch size, is the number of features.

* **Exponential Linear Unit:** The activation function is smooth everywhere and avoids the vanishing gradient problem as the output takes on negative values when the input is negative. is taken to be .
* **Batch Normalization:** The idea is to transform the inputs into a hidden layer's activation functions. We standardize or normalize first using the mean and variance parameters on a per feature basis and then learn a set of scaling and shifting parameters on a per feature basis that transforms the data. The following equations describe this layer succintly: The parameters we learn in this layer are .
* **Dropout:** This regularization technique simply drops the outputs from input and hidden units with a certain probability say .
* **Adam Optimization Algorithm:** This adaptive algorithm combines ideas from the Momentum and RMSProp optimization algorithms. The goal is to have some memory of past gradients which can guide future parameters updates. The following equations for the algorithm succintly describe this method assuming is our set of parameters to be learnt and is the learning rate.

### Results

* The AUROC score for the MNIST classification task with an autoencoder: 99.859478%.
* The accuracy score for the MNIST classification task with an autoencoder: 96.940000%.

autoencoder = VanillaAutoencoder(n\_feat=X\_train.shape[1],  
 n\_epoch=50,  
 batch\_size=100,  
 encoder\_layers=3,  
 decoder\_layers=3,  
 n\_hidden\_units=1000,  
 encoding\_dim=500,  
 denoising=None)  
  
print(autoencoder.autoencoder.summary())  
  
pipe\_autoencoder = Pipeline(steps=[("autoencoder", autoencoder),  
 ("scaler\_classifier", scaler\_classifier),  
 ("classifier", logistic)])  
  
pipe\_autoencoder = pipe\_autoencoder.fit(X\_train, y\_train)  
  
auroc\_autoencoder = roc\_auc\_score(lb.transform(y\_test.reshape(y\_test.shape[0], 1)),   
 pipe\_autoencoder.predict\_proba(X\_test),   
 average="weighted")  
   
acc\_autoencoder = pipe\_autoencoder.score(X\_test, y\_test)  
  
print("The AUROC score for the MNIST classification task with an autoencoder: %.6f%%." % (auroc\_autoencoder \* 100))  
print("The accuracy score for the MNIST classification task with an autoencoder: %.6f%%." % (acc\_autoencoder \* 100))

## MNIST: Denoising Autoencoders

The idea here is to add some noise to the data and try to learn a set of robust features that can reconstruct the non-noisy data from the noisy data. The MSE objective functions is as follows, , where is some noise term.

### Results

* The AUROC score for the MNIST classification task with a denoising autoencoder: 99.832232%.
* The accuracy score for the MNIST classification task with a denoising autoencoder: 96.930000%.

noise = 0.10 \* np.reshape(np.random.uniform(low=0.0,   
 high=1.0,   
 size=X\_train.shape[0] \* X\_train.shape[1]),   
 [X\_train.shape[0], X\_train.shape[1]])  
  
denoising\_autoencoder = VanillaAutoencoder(n\_feat=X\_train.shape[1],  
 n\_epoch=50,  
 batch\_size=100,  
 encoder\_layers=3,  
 decoder\_layers=3,  
 n\_hidden\_units=1000,  
 encoding\_dim=500,  
 denoising=noise)  
  
print(denoising\_autoencoder.autoencoder.summary())  
  
pipe\_denoising\_autoencoder = Pipeline(steps=[("autoencoder", denoising\_autoencoder),  
 ("scaler\_classifier", scaler\_classifier),  
 ("classifier", logistic)])  
  
pipe\_denoising\_autoencoder = pipe\_denoising\_autoencoder.fit(X\_train, y\_train)  
  
auroc\_denoising\_autoencoder = roc\_auc\_score(lb.transform(y\_test.reshape(y\_test.shape[0], 1)),   
 pipe\_denoising\_autoencoder.predict\_proba(X\_test),   
 average="weighted")  
  
acc\_denoising\_autoencoder = pipe\_denoising\_autoencoder.score(X\_test, y\_test)  
  
print("The AUROC score for the MNIST classification task with a denoising autoencoder: %.6f%%." % (auroc\_denoising\_autoencoder \* 100))  
print("The accuracy score for the MNIST classification task with a denoising autoencoder: %.6f%%." % (acc\_denoising\_autoencoder \* 100))

## MNIST: 1 Dimensional Convolutional Autoencoders

So far we have used flattened or reshaped raw data. Such a 1 dimensional tensor of pixel intensities per image, , might not take into account useful spatial features that the 2 dimensional tensor, , might contain. To overcome this problem, we introduce the concept of convolution filters, considering first their 1 dimensional version and then their 2 dimensional version.

The ideas behind convolution filters are closely related to handcrafted feature engineering: One can view the handcrafted features as simply the result of a predefined convolution filter, i.e., a convolution filter that has not been learnt based on the raw data at hand.

Suppose we have raw transactions data per some unit of analysis, i.e., mortgages, that will potentially help us in classifying a unit as either defaulted or not defaulted. We will keep this example simple by only allowing the transaction values to be either $100 or $0. The raw data per unit spans 5 time periods while the defaulted label is for the next period, i.e., period 6. Here is an example of a raw data for a particular unit:

Suppose further that if the average transaction value is $20 then we will see a default in period 6 for this particular mortgage unit. Otherwise we do not see a default in period 6. The average transaction value is an example of a handcrafted feature: A predefined handcrafted feature that has not been learnt in any manner. It has been arrived at via domain knowledge of credit risk. Denote this as .

The idea of learning such a feature is an example of a 1 dimensional convolution filter. As follows:

Assuming that is the correct representation of the raw data for this supervised learning task then the optimal set of parameters learnt via supervised learning, or perhaps unsupervised learning and then transferred to the supervised learning task, i.e., transfer learning, for is as follows where is :

This is a simple example however this clearly illusrates the principle behind using deep learning for automatic feature engineering or representation learning. One of the main benefits of learning such a representation in an unsupervised manner is that the same representation can then be used for multiple supervised learning tasks: Transfer learning. This is a principled manner of learning a representation from raw data: The client only needs to provide us with the raw 2 or 3 or more dimensional tensors and DataRobot will learn the optimal representation for potentially multiple supervised learning tasks at hand.

To summarize the 1 dimensional convolution filter for our simple example is defined as:

* is the input.
* is the kernel.
* The output is called a feature map and is the convolution operator or filter. This is the main difference between a vanilla neural network and a convolution neural network: We replace the matrix multiplication operator by the convolution operator.
* Depending on the task at hand we can have different types of convolution filters.
* Kernel size can be altered. In our example the kernel size is 5.
* Stride size can be altered. In our example we had no stride size however suppose that stride size was 1 and kernel size was 2, i.e., , then we would apply the kernel at the start of the input, i.e., , and move the kernel over the next area of the input, i.e., , and so on and so forth until we arrive at a feature map that consists of 4 real values. This is called a valid convolution while a padded, i.e., say padded with zero values, convolution would give us a feature map that is the same size as the input, i.e., 5 real values in our example.
* We can apply an activation function to the feature maps such as ELU mentioned earlier.
* Finally we can summarize the information contained in feature maps by taking a maximum or average value over a defined portion of the feature map. For instance, if after using a valid convolution we arrive at a feature map of size 4 and then apply a max pooling operation with size 4 then we will be taking the maximum value of this feature map. The result is another feature map.

This automates feature engineering however introduces architecture engineering where different architectures consisting of various convolution filters, activation functions, batch normalization layers, dropout layers and pooling operators can be stacked together in a pipeline in order to learn a good representation of the raw data. One usually creates an ensemble of such architectures.

The goal behind convolutional autoencoders is to use convolution filters, activation functions, batch normalization layers, dropout layers and pooling operators to create an encoder function which will learn a good representation of our raw data. The decoder will also use a similar set of layers as the encoder to reconstruct the raw data with one exception: Instead of using a pooling operator it will use an upsampling operator. The basic idea behind the upsampling operator is to repeat an element a certain number of times say size 4: One can view this as the inverse operator to the pooling operator. The pooling operator is essentially a downsampling operator and the upsampling operator is simply the inverse of that in some sense.

### Results

* The AUROC score for the MNIST classification task with a 1 dimensional convolutional autoencoder: 99.912944%.
* The accuracy score for the MNIST classification task with a 1 dimensional convolutional autoencoder: 97.570000%.

convolutional\_autoencoder = ConvolutionalAutoencoder(input\_shape=(int(math.pow(X\_train.shape[1], 0.5)), int(math.pow(X\_train.shape[1], 0.5))),  
 n\_epoch=50,  
 batch\_size=100,  
 encoder\_layers=3,  
 decoder\_layers=3,  
 filters=100,  
 kernel\_size=8,  
 strides=1,  
 pool\_size=4,  
 denoising=None)  
  
print(convolutional\_autoencoder.autoencoder.summary())  
  
pipe\_convolutional\_autoencoder = Pipeline(steps=[("autoencoder", convolutional\_autoencoder),  
 ("scaler\_classifier", scaler\_classifier),  
 ("classifier", logistic)])  
  
pipe\_convolutional\_autoencoder = pipe\_convolutional\_autoencoder.fit(np.reshape(X\_train, [X\_train.shape[0], int(math.pow(X\_train.shape[1], 0.5)), int(math.pow(X\_train.shape[1], 0.5))]),   
 y\_train)  
  
auroc\_convolutional\_autoencoder = roc\_auc\_score(lb.transform(y\_test.reshape(y\_test.shape[0], 1)),   
 pipe\_convolutional\_autoencoder.predict\_proba(np.reshape(X\_test, [X\_test.shape[0], int(math.pow(X\_train.shape[1], 0.5)), int(math.pow(X\_train.shape[1], 0.5))])),  
 average="weighted")  
  
acc\_convolutional\_autoencoder = pipe\_convolutional\_autoencoder.score(np.reshape(X\_test, [X\_test.shape[0], int(math.pow(X\_train.shape[1], 0.5)), int(math.pow(X\_train.shape[1], 0.5))]), y\_test)  
  
print("The AUROC score for the MNIST classification task with a 1 dimensional convolutional autoencoder: %.6f%%." % (auroc\_convolutional\_autoencoder \* 100))  
print("The accuracy score for the MNIST classification task with a 1 dimensional convolutional autoencoder: %.6f%%." % (acc\_convolutional\_autoencoder \* 100))

## MNIST: Sequence to Sequence Autoencoders

Given our mortgage default example a potentially more useful deep learning architecture might be the Recurrent Neural Network (RNN), specifically their state of the art variant the Long Short Term Memory (LSTM) network. The goal is to explicitly take into account the sequential nature of the raw data.

The gradients in a RNN depend on the parameter matrices defined for the model. Simply put these parameter matrices can end up being multiplied many times over and hence cause two major problems for learning: Exploding and vanishing gradients. If the spectral radius of the parameter matrices, i.e., the maximum absolute value of the eigenvalues of a matrix, is more than 1 then gradients can become large enough, i.e., explode in value, such that learning diverges and similarly if the spectral radius is less than 1 then gradients can become small, i.e., vanish in value, such that the next best transition for the parameters cannot be reliably calculated. Appropriate calculation of the gradient is important for estimating the optimal set of parameters that define a machine learning method and the LSTM network overcomes these problems in a vanilla RNN. We now define the LSTM network for 1 time step, i.e., 1 memory cell.

We calculate the value of the input gate, the value of the memory cell state at time period where is some activation function and the value of the forget gate:

The forget gate controls the amount the LSTM remembers, i.e., the value of the memory cell state at time period where is the hadamard product:

With the updated state of the memory cell we calculate the value of the outputs gate and finally the output value itself:

We can have a wide variety of LSTM architectures such as the convolutional LSTM where note that we replace the matrix multiplication operators in the input gate, the initial estimate of the memory cell state, the forget gate and the output gate by the convolution operator :

Another popular variant is the peephole LSTM where the gates are allowed to peep at the memory cell state:

The goal for the sequence to sequence autoencoder is to create a representation of the raw data using a LSTM as an encoder. This representation will be a sequence of vectors say, , learnt from a sequence of raw data vectors say, . The final vector of the representation, , is our encoded representation, also called a context vector. This context vector is repeated as many times as the length of the sequence such that it can be used as an input to a decoder which is yet another LSTM. The decoder LSTM will use this context vector to recontruct the sequence of raw data vectors, . If the context vector is useful in the recontruction task then it can be further used for other tasks such as predicting default risk as given in our example.

### Results

* The AUROC score for the MNIST classification task with a sequence to sequence autoencoder: 99.889814%.
* The accuracy score for the MNIST classification task with a sequence to sequence autoencoder: 97.600000%.

seq2seq\_autoencoder = Seq2SeqAutoencoder(input\_shape=(int(math.pow(X\_train.shape[1], 0.5)), int(math.pow(X\_train.shape[1], 0.5))),  
 n\_epoch=50,  
 batch\_size=100,  
 encoder\_layers=3,  
 decoder\_layers=3,  
 n\_hidden\_units=100,  
 encoding\_dim=100,  
 stateful=False,  
 denoising=None)  
  
print(seq2seq\_autoencoder.autoencoder.summary())  
  
pipe\_seq2seq\_autoencoder = Pipeline(steps=[("autoencoder", seq2seq\_autoencoder),  
 ("scaler\_classifier", scaler\_classifier),  
 ("classifier", logistic)])  
  
pipe\_seq2seq\_autoencoder = pipe\_seq2seq\_autoencoder.fit(np.reshape(X\_train, [X\_train.shape[0], int(math.pow(X\_train.shape[1], 0.5)), int(math.pow(X\_train.shape[1], 0.5))]),  
 y\_train)  
  
auroc\_seq2seq\_autoencoder = roc\_auc\_score(lb.transform(y\_test.reshape(y\_test.shape[0], 1)),   
 pipe\_seq2seq\_autoencoder.predict\_proba(np.reshape(X\_test, [X\_test.shape[0], int(math.pow(X\_train.shape[1], 0.5)), int(math.pow(X\_train.shape[1], 0.5))])),  
 average="weighted")  
  
acc\_seq2seq\_autoencoder = pipe\_seq2seq\_autoencoder.score(np.reshape(X\_test, [X\_test.shape[0], int(math.pow(X\_train.shape[1], 0.5)), int(math.pow(X\_train.shape[1], 0.5))]), y\_test)  
  
print("The AUROC score for the MNIST classification task with a sequence to sequence autoencoder: %.6f%%." % (auroc\_seq2seq\_autoencoder \* 100))  
print("The accuracy score for the MNIST classification task with a sequence to sequence autoencoder: %.6f%%." % (acc\_seq2seq\_autoencoder \* 100))

## MNIST: Variational Autoencoders

We now combine Bayesian inference with deep learning by using variational inference to train a vanilla autoencoder. This moves us towards generative modelling which can have further use cases in semi-supervised learning. The other benefit of training using Bayesian inference is that we can be more robust to higher capacity deep learners, i.e., avoid overfitting.

* Assume is our raw data while is our learnt representation.
* We have a prior belief on our learnt representation:
* The posterior distribution for our learnt representation is:
* The marginal likelihood, , is often intractable causing the posterior distribution, , to be intractable:
* We therefore need an approximate posterior distribution via variational inference that can deal with the intractability. This additionally also provides the benefit of dealing with large scale datasets as generally Markov Chain Monte Carlo (MCMC) methods are not well suited for large scale datasets. One might also consider Laplace approximation for the approximate posterior distribution however we will stick with variational inference as it allows a richer set of approximations compared to Laplace approximation. Laplace approximation simply amounts to finding the Maximum A Posteriori (MAP) estimate to an augmented likelihood optimization, taking the negative of the inverse of the Hessian at the MAP estimate to estimate the variance-covariance matrix and finally use the variance-covariance matrix with a multivariate Gaussian distribution or some other appropriate multivariate distribution.
* Assume that our approximate posterior distribution, which is also our probabilistic encoder, is given as:
* Our probabilistic decoder is given by:
* Given our setup above with regards to an encoder and a decoder let us now write down the optimization problem where are the generative model parameters while are the variational parameters:
* Note that is non-negative therefore that makes the ELBO a lower bound on :
* Therefore we can alter our optimization problem to look only at the ELBO:
* The above integration problem can be solved via Monte Carlo integration as is not intractable. Assuming that the probabilistic encoder is a multivariate Gaussian with a diagonal variance-covariance matrix we use the reparameterization trick to sample from this distribution say times in order to calculate the expectation term in the ELBO optimization problem. The reparameterization trick in this particular case amounts to sampling times from the standard Gaussian distribution, multiplying the samples by and adding to the samples.
* is our learnt representation used for the reconstruction of the raw data. If the learnt representation is useful it can then be used for other tasks as well.
* This is a powerful manner of combining Bayesian inference with deep learning. Variational inference used in this manner can be applied to various deep learning architectures and has further links with the Generative Adversarial Network (GAN). We explore the use of adversarial learning in representation learning in another repo/paper.

### Results

* The AUROC score for the MNIST classification task with a variational autoencoder: 99.814369%.
* The accuracy score for the MNIST classification task with a variational autoencoder: 96.520000%.

encoding\_dim = 500  
  
variational\_autoencoder = VariationalAutoencoder(n\_feat=X\_train.shape[1],  
 n\_epoch=50,  
 batch\_size=100,  
 encoder\_layers=3,  
 decoder\_layers=3,  
 n\_hidden\_units=1000,  
 encoding\_dim=encoding\_dim,  
 denoising=None)  
  
print(variational\_autoencoder.autoencoder.summary())  
  
pipe\_variational\_autoencoder = Pipeline(steps=[("autoencoder", variational\_autoencoder),  
 ("scaler\_classifier", scaler\_classifier),  
 ("classifier", logistic)])  
  
pipe\_variational\_autoencoder = pipe\_variational\_autoencoder.fit(X\_train, y\_train)  
  
auroc\_variational\_autoencoder = roc\_auc\_score(lb.transform(y\_test.reshape(y\_test.shape[0], 1)),   
 pipe\_variational\_autoencoder.predict\_proba(X\_test),   
 average="weighted")  
  
acc\_variational\_autoencoder = pipe\_variational\_autoencoder.score(X\_test, y\_test)  
  
print("The AUROC score for the MNIST classification task with a variational autoencoder: %.6f%%." % (auroc\_variational\_autoencoder \* 100))  
print("The accuracy score for the MNIST classification task with a variational autoencoder: %.6f%%." % (acc\_variational\_autoencoder \* 100))  
  
if encoding\_dim == 2:  
 test\_encoded\_df = pd.DataFrame(pipe\_variational\_autoencoder.named\_steps["autoencoder"].encoder.predict(X\_test))  
 test\_encoded\_df["Target"] = y\_test  
 test\_encoded\_df.columns.values[0:2] = ["Encoding\_1", "Encoding\_2"]  
  
 scaler\_plot = MinMaxScaler(feature\_range=(0.25, 0.75))  
 scaler\_plot = scaler\_plot.fit(test\_encoded\_df[["Encoding\_1", "Encoding\_2"]])  
 test\_encoded\_df[["Encoding\_1", "Encoding\_2"]] = scaler\_plot.transform(test\_encoded\_df[["Encoding\_1", "Encoding\_2"]])  
  
 cluster\_plot = ggplot(test\_encoded\_df) + \  
 geom\_point(aes(x="Encoding\_1",   
 y="Encoding\_2",   
 fill="factor(Target)"),  
 size=1,  
 color = "black") + \  
 xlab("Encoding dimension 1") + \  
 ylab("Encoding dimension 2") + \  
 ggtitle("Variational autoencoder with 2-dimensional encoding") + \  
 theme\_matplotlib()  
 print(cluster\_plot)  
  
 n = 15  
 digit\_size = 28  
 figure = np.zeros((digit\_size \* n, digit\_size \* n))  
 grid\_x = norm.ppf(np.linspace(0.05, 0.95, n))  
 grid\_y = norm.ppf(np.linspace(0.05, 0.95, n))  
  
 for i, xi in enumerate(grid\_x):  
 for j, yi in enumerate(grid\_y):  
 z\_sample = np.array([[xi, yi]])  
 x\_decoded = pipe\_variational\_autoencoder.named\_steps["autoencoder"].generator.predict(z\_sample)  
 digit = x\_decoded[0].reshape(digit\_size, digit\_size)  
 figure[i \* digit\_size: (i + 1) \* digit\_size, j \* digit\_size: (j + 1) \* digit\_size] = digit  
  
 plt.figure(figsize=(10, 10))  
 plt.imshow(figure, cmap="Greys\_r")  
 plt.title("Variational autoencoder with 2-dimensional encoding\nGenerating new images")  
 plt.xlabel("")  
 plt.ylabel("")  
 plt.show()

## MNIST: 2 Dimensional Convolutional Autoencoders

For 2 dimensional convolution filters the idea is similar as for the 1 dimensional convolution filters. We will stick to our previously mentioned banking example to illustrate this point.

In the 2 dimensional tensor of raw transactions data now we have 5 historical time periods, i.e., the rows, and 3 different transaction types, i.e., the columns. We will use a kernel, , to extract useful features from the raw data. The choice of such a kernel means that we are interested in finding a feature map across all 3 transaction types and 2 historical time periods. We will use a stride length of 1 and a valid convolution to extract features over different patches of the raw data. The following will illustrate this point where :

The principles and ideas apply to 2 dimensional convolution filters as they do for their 1 dimensional counterparts there we will not repeat them here.

### Results

* The AUROC score for the MNIST classification task with a 2 dimensional convolutional autoencoder: 99.987741%.
* The accuracy score for the MNIST classification task with a 2 dimensional convolutional autoencoder: 98.860000%.

convolutional2D\_autoencoder = Convolutional2DAutoencoder(input\_shape=(int(math.pow(X\_train.shape[1], 0.5)), int(math.pow(X\_train.shape[1], 0.5)), 1),  
 n\_epoch=5,  
 batch\_size=100,  
 encoder\_layers=3,  
 decoder\_layers=3,  
 filters=100,  
 kernel\_size=(8, 8),  
 strides=(1, 1),  
 pool\_size=(4, 4),  
 denoising=None)  
  
print(convolutional2D\_autoencoder.autoencoder.summary())  
  
pipe\_convolutional2D\_autoencoder = Pipeline(steps=[("autoencoder", convolutional2D\_autoencoder),  
 ("scaler\_classifier", scaler\_classifier),  
 ("classifier", logistic)])  
  
pipe\_convolutional2D\_autoencoder = pipe\_convolutional2D\_autoencoder.fit(np.reshape(X\_train, [X\_train.shape[0], int(math.pow(X\_train.shape[1], 0.5)), int(math.pow(X\_train.shape[1], 0.5)), 1]),  
 y\_train)  
  
auroc\_convolutional2D\_autoencoder = roc\_auc\_score(lb.transform(y\_test.reshape(y\_test.shape[0], 1)),   
 pipe\_convolutional2D\_autoencoder.predict\_proba(np.reshape(X\_test, [X\_test.shape[0], int(math.pow(X\_train.shape[1], 0.5)), int(math.pow(X\_train.shape[1], 0.5)), 1])),  
 average="weighted")  
  
acc\_convolutional2D\_autoencoder = pipe\_convolutional2D\_autoencoder.score(np.reshape(X\_test, [X\_test.shape[0], int(math.pow(X\_test.shape[1], 0.5)), int(math.pow(X\_test.shape[1], 0.5)), 1]), y\_test)  
  
print("The AUROC score for the MNIST classification task with a 2 dimensional convolutional autoencoder: %.6f%%." % (auroc\_convolutional2D\_autoencoder \* 100))  
print("The accuracy score for the MNIST classification task with a 2 dimensional convolutional autoencoder: %.6f%%." % (acc\_convolutional2D\_autoencoder \* 100))

## Insurance: No Autoencoders

We now proceed to run the insurance model without any handcrafted or deep learning based feature engineering.

### Results

* The AUROC score for the insurance classification task without autoencoders: 92.206261%.
* The accuracy score for the insurance classification task without autoencoders: 82.333333%.

claim\_risk = pd.read\_csv(filepath\_or\_buffer="../R/data/claim\_risk.csv")  
claim\_risk.drop(columns="policy.id", axis=1, inplace=True)  
claim\_risk = np.asarray(claim\_risk).ravel()  
  
transactions = pd.read\_csv(filepath\_or\_buffer="../R/data/transactions.csv")  
transactions.drop(columns="policy.id", axis=1, inplace=True)  
  
n\_policies = 1000  
n\_transaction\_types = 3  
n\_time\_periods = 4  
  
transactions = np.reshape(np.asarray(transactions), (n\_policies, n\_time\_periods \* n\_transaction\_types))  
  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(transactions, claim\_risk, test\_size=0.3, random\_state=666)  
  
min\_X\_train = np.apply\_along\_axis(func1d=np.min, axis=0, arr=X\_train)  
max\_X\_train = np.apply\_along\_axis(func1d=np.max, axis=0, arr=X\_train)   
range\_X\_train = max\_X\_train - min\_X\_train + sys.float\_info.epsilon  
X\_train = (X\_train - min\_X\_train) / range\_X\_train  
X\_test = (X\_test - min\_X\_train) / range\_X\_train  
  
pipe\_base = Pipeline(steps=[("scaler\_classifier", scaler\_classifier),  
 ("classifier", logistic)])  
  
pipe\_base = pipe\_base.fit(X\_train, y\_train)  
  
auroc\_base = roc\_auc\_score(y\_true=y\_test,  
 y\_score=pipe\_base.predict\_proba(X\_test)[:, 1],   
 average="weighted")  
   
acc\_base = pipe\_base.score(X\_test, y\_test)  
  
print("The AUROC score for the insurance classification task without autoencoders: %.6f%%." % (auroc\_base \* 100))  
print("The accuracy score for the insurance classification task without autoencoders: %.6f%%." % (acc\_base \* 100))

## Insurance: PCA

We now proceed to run the insurance model without any handcrafted or deep learning based feature engineering however with a PCA filter that picks the number of components that explain of the variation.

### Results

* The AUROC score for the insurance classification task with PCA: 91.128859%.
* The accuracy score for the insurance classification task with PCA: 82.000000%.

pipe\_pca = Pipeline(steps=[("PCA", PCA(n\_components=0.99)),  
 ("scaler\_classifier", scaler\_classifier),  
 ("classifier", logistic)])  
  
pipe\_pca = pipe\_pca.fit(X\_train, y\_train)  
  
auroc\_pca = roc\_auc\_score(y\_true=y\_test,  
 y\_score=pipe\_pca.predict\_proba(X\_test)[:, 1],   
 average="weighted")  
   
acc\_pca = pipe\_pca.score(X\_test, y\_test)  
  
print("The AUROC score for the insurance classification task with PCA: %.6f%%." % (auroc\_pca \* 100))  
print("The accuracy score for the insurance classification task with PCA: %.6f%%." % (acc\_pca \* 100))

## Insurance: Handcrafted Features

In this case we have created some handcrafted features which we believe provide a useful representation of the raw data for the insurance model.

### Results

* The AUROC score for the insurance classification task with handcrafted features: 93.610635%.
* The accuracy score for the insurance classification task with handcrafted features: 85.000000%.

claim\_risk = pd.read\_csv(filepath\_or\_buffer="../R/data/claim\_risk.csv")  
claim\_risk.drop(columns="policy.id", axis=1, inplace=True)  
claim\_risk = np.asarray(claim\_risk).ravel()  
  
handcrafted\_features = pd.read\_csv(filepath\_or\_buffer="../R/data/handcrafted\_features.csv")  
handcrafted\_features = np.asarray(handcrafted\_features)  
  
n\_policies = 1000  
n\_feat = 12  
  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(handcrafted\_features, claim\_risk, test\_size=0.3, random\_state=666)  
  
min\_X\_train = np.apply\_along\_axis(func1d=np.min, axis=0, arr=X\_train)  
max\_X\_train = np.apply\_along\_axis(func1d=np.max, axis=0, arr=X\_train)   
range\_X\_train = max\_X\_train - min\_X\_train + sys.float\_info.epsilon  
X\_train = (X\_train - min\_X\_train) / range\_X\_train  
X\_test = (X\_test - min\_X\_train) / range\_X\_train  
  
pipe\_hcfe = Pipeline(steps=[("scaler\_classifier", scaler\_classifier),  
 ("classifier", logistic)])  
  
pipe\_hcfe = pipe\_hcfe.fit(X\_train, y\_train)  
  
auroc\_hcfe = roc\_auc\_score(y\_true=y\_test,  
 y\_score=pipe\_hcfe.predict\_proba(X\_test)[:, 1],   
 average="weighted")  
   
acc\_hcfe = pipe\_hcfe.score(X\_test, y\_test)  
  
print("The AUROC score for the insurance classification task with handcrafted features: %.6f%%." % (auroc\_hcfe \* 100))  
print("The accuracy score for the insurance classification task with handcrafted features: %.6f%%." % (acc\_hcfe \* 100))

## Insurance: Handcrafted Features and PCA

In this case we have created some handcrafted features which we believe provide a useful representation of the raw data for the insurance model. We also use a PCA filter.

### Results

* The AUROC score for the insurance classification task with handcrafted features and PCA: 93.160377%.
* The accuracy score for the insurance classification task with handcrafted features and PCA: 83.333333%.

pipe\_hcfe\_pca = Pipeline(steps=[("PCA", PCA(n\_components=0.99)),  
 ("scaler\_classifier", scaler\_classifier),  
 ("classifier", logistic)])  
   
pipe\_hcfe\_pca = pipe\_hcfe\_pca.fit(X\_train, y\_train)  
  
auroc\_hcfe\_pca = roc\_auc\_score(y\_true=y\_test,  
 y\_score=pipe\_hcfe\_pca.predict\_proba(X\_test)[:, 1],   
 average="weighted")  
   
acc\_hcfe\_pca = pipe\_hcfe\_pca.score(X\_test, y\_test)  
  
print("The AUROC score for the insurance classification task with handcrafted features and PCA: %.6f%%." % (auroc\_hcfe\_pca \* 100))  
print("The accuracy score for the insurance classification task with handcrafted features and PCA: %.6f%%." % (acc\_hcfe\_pca \* 100))

## Insurance: Vanilla Autoencoders

In this case we use vanilla autoencoders to learn a good representation of the raw data such that we can obtain an uplift, primarily in terms of AUROC, for the supervised learning task.

### Results

* The AUROC score for the insurance classification task with an autoencoder: 93.932247%.
* The accuracy score for the insurance classification task with an autoencoder: 85.000000%.

autoencoder = VanillaAutoencoder(n\_feat=X\_train.shape[1],  
 n\_epoch=100,  
 batch\_size=50,  
 encoder\_layers=3,  
 decoder\_layers=3,  
 n\_hidden\_units=100,  
 encoding\_dim=50,  
 denoising=None)  
  
print(autoencoder.autoencoder.summary())  
  
pipe\_autoencoder = Pipeline(steps=[("autoencoder", autoencoder),  
 ("scaler\_classifier", scaler\_classifier),  
 ("classifier", logistic)])  
  
pipe\_autoencoder = pipe\_autoencoder.fit(X\_train, y\_train)  
  
auroc\_autoencoder = roc\_auc\_score(y\_true=y\_test,  
 y\_score=pipe\_autoencoder.predict\_proba(X\_test)[:, 1],   
 average="weighted")  
   
acc\_autoencoder = pipe\_autoencoder.score(X\_test, y\_test)  
  
print("The AUROC score for the insurance classification task with an autoencoder: %.6f%%." % (auroc\_autoencoder \* 100))  
print("The accuracy score for the insurance classification task with an autoencoder: %.6f%%." % (acc\_autoencoder \* 100))

## Insurance: Denoising Autoencoders

In this case we use denoising autoencoders to learn a good representation of the raw data such that we can obtain an uplift, primarily in terms of AUROC, for the supervised learning task.

### Results

* The AUROC score for the insurance classification task with a denoising autoencoder: 93.712479%.
* The accuracy score for the insurance classification task with a denoising autoencoder: 85.333333%.

noise = 0.10 \* np.reshape(np.random.uniform(low=0.0,   
 high=1.0,   
 size=X\_train.shape[0] \* X\_train.shape[1]),   
 [X\_train.shape[0], X\_train.shape[1]])  
  
denoising\_autoencoder = VanillaAutoencoder(n\_feat=X\_train.shape[1],  
 n\_epoch=100,  
 batch\_size=50,  
 encoder\_layers=3,  
 decoder\_layers=3,  
 n\_hidden\_units=100,  
 encoding\_dim=50,  
 denoising=noise)  
  
print(denoising\_autoencoder.autoencoder.summary())  
  
pipe\_denoising\_autoencoder = Pipeline(steps=[("autoencoder", denoising\_autoencoder),  
 ("scaler\_classifier", scaler\_classifier),  
 ("classifier", logistic)])  
  
pipe\_denoising\_autoencoder = pipe\_denoising\_autoencoder.fit(X\_train, y\_train)  
  
auroc\_denoising\_autoencoder = roc\_auc\_score(y\_true=y\_test,  
 y\_score=pipe\_denoising\_autoencoder.predict\_proba(X\_test)[:, 1],   
 average="weighted")  
   
acc\_denoising\_autoencoder = pipe\_denoising\_autoencoder.score(X\_test, y\_test)  
  
print("The AUROC score for the insurance classification task with a denoising autoencoder: %.6f%%." % (auroc\_denoising\_autoencoder \* 100))  
print("The accuracy score for the insurance classification task with a denoising autoencoder: %.6f%%." % (acc\_denoising\_autoencoder \* 100))

## Insurance: Sequence to Sequence Autoencoders

In this case we use sequence to sequence autoencoders, taking into account the time series nature, i.e., sequential nature, of the raw transactions data, to learn a good representation of the raw data such that we can obtain an uplift, primarily in terms of AUROC, for the supervised learning task.

### Results

* The AUROC score for the insurance classification task with a sequence to sequence autoencoder: 91.418310%.
* The accuracy score for the insurance classification task with a sequence to sequence autoencoder: 84.000000%.

transactions = np.reshape(np.asarray(transactions), (n\_policies, n\_time\_periods, n\_transaction\_types))  
  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(transactions, claim\_risk, test\_size=0.3, random\_state=666)  
  
min\_X\_train = np.apply\_along\_axis(func1d=np.min, axis=0, arr=X\_train)  
max\_X\_train = np.apply\_along\_axis(func1d=np.max, axis=0, arr=X\_train)   
range\_X\_train = max\_X\_train - min\_X\_train + sys.float\_info.epsilon  
X\_train = (X\_train - min\_X\_train) / range\_X\_train  
X\_test = (X\_test - min\_X\_train) / range\_X\_train  
  
seq2seq\_autoencoder = Seq2SeqAutoencoder(input\_shape=(X\_train.shape[1], X\_train.shape[2]),  
 n\_epoch=100,  
 batch\_size=50,  
 encoder\_layers=3,  
 decoder\_layers=3,  
 n\_hidden\_units=50,  
 encoding\_dim=50,  
 stateful=False,  
 denoising=None)  
  
print(seq2seq\_autoencoder.autoencoder.summary())  
  
pipe\_seq2seq\_autoencoder = Pipeline(steps=[("autoencoder", seq2seq\_autoencoder),  
 ("scaler\_classifier", scaler\_classifier),  
 ("classifier", logistic)])  
  
pipe\_seq2seq\_autoencoder = pipe\_seq2seq\_autoencoder.fit(X\_train, y\_train)  
  
auroc\_seq2seq\_autoencoder = roc\_auc\_score(y\_test,   
 pipe\_seq2seq\_autoencoder.predict\_proba(X\_test)[:, 1],  
 average="weighted")  
  
acc\_seq2seq\_autoencoder = pipe\_seq2seq\_autoencoder.score(X\_test, y\_test)  
  
print("The AUROC score for the insurance classification task with a sequence to sequence autoencoder: %.6f%%." % (auroc\_seq2seq\_autoencoder \* 100))  
print("The accuracy score for the insurance classification task with a sequence to sequence autoencoder: %.6f%%." % (acc\_seq2seq\_autoencoder \* 100))

## Insurance: 1 Dimensional Convolutional Autoencoders

In this case we use 1 dimensional convolutional autoencoders to learn a good representation of the raw data such that we can obtain an uplift, primarily in terms of AUROC, for the supervised learning task.

### Results

* The AUROC score for the insurance classification task with a 1 dimensional convolutional autoencoder: 91.509434%.
* The accuracy score for the insurance classification task with a 1 dimensional convolutional autoencoder: 84.666667%.

transactions = np.reshape(np.asarray(transactions), (n\_policies, n\_time\_periods, n\_transaction\_types))  
  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(transactions, claim\_risk, test\_size=0.3, random\_state=666)  
min\_X\_train = np.apply\_along\_axis(func1d=np.min, axis=0, arr=X\_train)  
max\_X\_train = np.apply\_along\_axis(func1d=np.max, axis=0, arr=X\_train)   
range\_X\_train = max\_X\_train - min\_X\_train + sys.float\_info.epsilon  
X\_train = (X\_train - min\_X\_train) / range\_X\_train  
X\_test = (X\_test - min\_X\_train) / range\_X\_train  
  
convolutional\_autoencoder = ConvolutionalAutoencoder(input\_shape=(X\_train.shape[1], X\_train.shape[2]),  
 n\_epoch=100,  
 batch\_size=50,  
 encoder\_layers=3,  
 decoder\_layers=3,  
 filters=50,  
 kernel\_size=2,  
 strides=1,  
 pool\_size=2,  
 denoising=None)  
  
print(convolutional\_autoencoder.autoencoder.summary())  
  
pipe\_convolutional\_autoencoder = Pipeline(steps=[("autoencoder", convolutional\_autoencoder),  
 ("scaler\_classifier", scaler\_classifier),  
 ("classifier", logistic)])  
  
pipe\_convolutional\_autoencoder = pipe\_convolutional\_autoencoder.fit(X\_train, y\_train)  
  
auroc\_convolutional\_autoencoder = roc\_auc\_score(y\_test,   
 pipe\_convolutional\_autoencoder.predict\_proba(X\_test)[:, 1],  
 average="weighted")  
  
acc\_convolutional\_autoencoder = pipe\_convolutional\_autoencoder.score(X\_test, y\_test)  
  
print("The AUROC score for the insurance classification task with a 1 dimensional convolutional autoencoder: %.6f%%." % (auroc\_convolutional\_autoencoder \* 100))  
print("The accuracy score for the insurance classification task with a 1 dimensional convolutional autoencoder: %.6f%%." % (acc\_convolutional\_autoencoder \* 100))

## Insurance: 2 Dimensional Convolutional Autoencoders

In this case we use 2 dimensional convolutional autoencoders to learn a good representation of the raw data such that we can obtain an uplift, primarily in terms of AUROC, for the supervised learning task.

### Results

* The AUROC score for the insurance classification task with a 2 dimensional convolutional autoencoder: 92.645798%.
* The accuracy score for the insurance classification task with a 2 dimensional convolutional autoencoder: 86.000000%.

transactions = np.reshape(np.asarray(transactions), (n\_policies, n\_time\_periods, n\_transaction\_types, 1))  
  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(transactions, claim\_risk, test\_size=0.3, random\_state=666)  
min\_X\_train = np.apply\_along\_axis(func1d=np.min, axis=0, arr=X\_train)  
max\_X\_train = np.apply\_along\_axis(func1d=np.max, axis=0, arr=X\_train)   
range\_X\_train = max\_X\_train - min\_X\_train + sys.float\_info.epsilon  
X\_train = (X\_train - min\_X\_train) / range\_X\_train  
X\_test = (X\_test - min\_X\_train) / range\_X\_train  
  
convolutional2D\_autoencoder = Convolutional2DAutoencoder(input\_shape=(X\_train.shape[1], X\_train.shape[2], 1),  
 n\_epoch=100,  
 batch\_size=50,  
 encoder\_layers=3,  
 decoder\_layers=3,  
 filters=50,  
 kernel\_size=(2, 3),  
 strides=(1, 1),  
 pool\_size=(2, 1),  
 denoising=None)  
  
print(convolutional2D\_autoencoder.autoencoder.summary())  
  
pipe\_convolutional2D\_autoencoder = Pipeline(steps=[("autoencoder", convolutional2D\_autoencoder),  
 ("scaler\_classifier", scaler\_classifier),  
 ("classifier", logistic)])  
  
pipe\_convolutional2D\_autoencoder = pipe\_convolutional2D\_autoencoder.fit(X\_train, y\_train)  
  
auroc\_convolutional2D\_autoencoder = roc\_auc\_score(y\_test,   
 pipe\_convolutional2D\_autoencoder.predict\_proba(X\_test)[:, 1],  
 average="weighted")  
  
acc\_convolutional2D\_autoencoder = pipe\_convolutional2D\_autoencoder.score(X\_test, y\_test)  
  
print("The AUROC score for the insurance classification task with a 2 dimensional convolutional autoencoder: %.6f%%." % (auroc\_convolutional2D\_autoencoder \* 100))  
print("The accuracy score for the insurance classification task with a 2 dimensional convolutional autoencoder: %.6f%%." % (acc\_convolutional2D\_autoencoder \* 100))

## Insurance: Variational Autoencoders

In this case we use variational autoencoders to learn a good representation of the raw data such that we can obtain an uplift, primarily in terms of AUROC, for the supervised learning task.

### Results

* The AUROC score for the insurance classification task with a variational autoencoder: 90.871569%.
* The accuracy score for the insurance classification task with a variational autoencoder: 83.666667%.

transactions = np.reshape(np.asarray(transactions), (n\_policies, n\_time\_periods \* n\_transaction\_types))  
  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(transactions, claim\_risk, test\_size=0.3, random\_state=666)  
  
min\_X\_train = np.apply\_along\_axis(func1d=np.min, axis=0, arr=X\_train)  
max\_X\_train = np.apply\_along\_axis(func1d=np.max, axis=0, arr=X\_train)   
range\_X\_train = max\_X\_train - min\_X\_train + sys.float\_info.epsilon  
X\_train = (X\_train - min\_X\_train) / range\_X\_train  
X\_test = (X\_test - min\_X\_train) / range\_X\_train  
  
variational\_autoencoder = VariationalAutoencoder(n\_feat=X\_train.shape[1],  
 n\_epoch=100,  
 batch\_size=50,  
 encoder\_layers=3,  
 decoder\_layers=3,  
 n\_hidden\_units=100,  
 encoding\_dim=50,  
 denoising=None)  
  
print(variational\_autoencoder.autoencoder.summary())  
  
pipe\_variational\_autoencoder = Pipeline(steps=[("autoencoder", variational\_autoencoder),  
 ("scaler\_classifier", scaler\_classifier),  
 ("classifier", logistic)])  
  
pipe\_variational\_autoencoder = pipe\_variational\_autoencoder.fit(X\_train, y\_train)  
  
auroc\_variational\_autoencoder = roc\_auc\_score(y\_true=y\_test,  
 y\_score=pipe\_variational\_autoencoder.predict\_proba(X\_test)[:, 1],   
 average="weighted")  
   
acc\_variational\_autoencoder = pipe\_variational\_autoencoder.score(X\_test, y\_test)  
  
print("The AUROC score for the insurance classification task with a variational autoencoder: %.6f%%." % (auroc\_variational\_autoencoder \* 100))  
print("The accuracy score for the insurance classification task with a variational autoencoder: %.6f%%." % (acc\_variational\_autoencoder \* 100))

## References

1. Goodfellow, I., Bengio, Y. and Courville A. (2016). Deep Learning (MIT Press).
2. Geron, A. (2017). Hands-On Machine Learning with Scikit-Learn & Tensorflow (O'Reilly).
3. <http://scikit-learn.org/stable/#>
4. <https://towardsdatascience.com/learning-rate-schedules-and-adaptive-learning-rate-methods-for-deep-learning-2c8f433990d1>
5. <https://stackoverflow.com/questions/42177658/how-to-switch-backend-with-keras-from-tensorflow-to-theano>
6. <https://blog.keras.io/building-autoencoders-in-keras.html>
7. <https://keras.io>
8. <https://www.cs.cornell.edu/courses/cs1114/2013sp/sections/S06_convolution.pdf>
9. <http://deeplearning.net/tutorial/lstm.html>
10. Kingma, D. P., and Welling M. (2014). Auto-Encoding Variational Bayes (<https://arxiv.org/abs/1312.6114>).