Outlier Detection with Autoencoder Ensembles

Chandni Bhatia, Jill Shah, Lujia Wang, Shiwangi Prasad, Yuting Peng

1 Abstract

In this paper, we work on autoencoder ensembles for unsupervised outlier detection. One problem with neural networks is that they are sensitive to noise and often require large data sets to work robustly, while increasing data size makes them slow. As a result, there are only a few existing works in the literature on the use of neural networks in outlier detection. In this paper,we developed autoencoder which vary randomly on the connectivity architecture of the autoencoder to obtain significantly better performance. Furthermore, we combine this technique with an adaptive sampling method to make our approach more efficient and effective.

2 Introduction

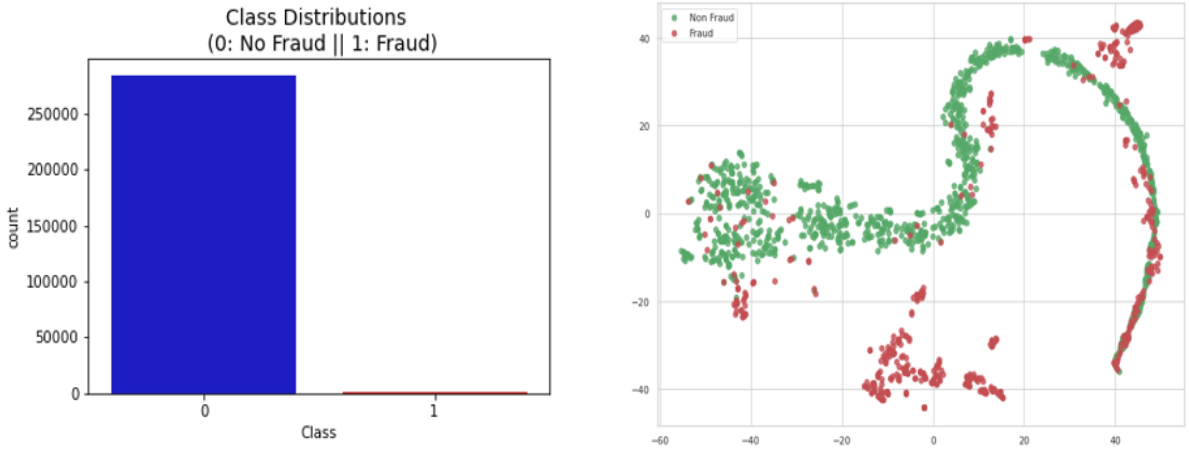
Outliers are data points that differ significantly from the remaining data. The basic approach in neural networks is to use a multi-layer symmetric neural network to reconstruct (i.e, replicate) the data. The reconstruction error is used as the outlier score. There are several problems with this approach. First, even though deep neural networks are generally considered a powerful learning tool on large data sets, the effectiveness on smaller data sets remains in doubt because of the overfitting caused by the large number of parameters. Training such neural networks often results in convergence to local optima. Increasing data size reduces overfitting but can cause computational challenges. Furthermore, there is sometimes an inherent bottleneck on data availability. Although neural networks have been explored for outlier detection [10, 13, 22], this class of approaches has not been popular in the outlier detection community because of the aforementioned drawbacks. In this work, we employ Isolation forest and local outlier factor method for outlier detection.

Next, instead of fully connected autoencoders, various randomly connected autoencoders with different structures and connection densities were implemented reducing the computational complexity. Moreover, we leverage a carefully designed adaptive sample size method within the ensemble framework to achieve the dual goals of improved diversity and training time. Therefore, our approach combines adaptive sampling with randomized model construction in order to achieve high-quality results. We refer to this model as RandNet, which stands for Randomized Neural Network for Outlier Detection. We present experimental results showing the effectiveness of the approach. Although we do not investigate the option of training the base ensemble components in parallel, a salient observation about this approach is that the training process can be easily parallelized.

3 DataSets

3.a Credit Card Data Set

This Data contains 284807 rows and 30 columns. The last column contains information regarding fraud/no fraud. All the columns have been transformed using PCA and the original information regarding the transaction is not available due to privacy concerns. Some statistics of the data set is worth noting. There's 492 cases of cases of frauds in this data set out of 284807. Which is only 0.17 percent. Thus the data set is highly imbalanced. Average transaction amount of the credit card is 88.35 dollars. And maximum fraudulent transaction amount was 2125 compared to maximum of 25691.16. Hence fraudsters try to hide their transaction amount near mean of the data. Mean of non-fraud vs fraud is 88.29 vs 122.29.



(a) Credit Card Classes

(b) Credit Card T-SNE

Figure 1: Credit Card Data Set

3.b Dummy Data Set

Besides the above data sets, a dummy data is constructed for quick sanity checks. This is a small data set with 10000 samples. Two groups of samples is generated from different distributions. Inliers are from normal distribution and outliers are from uniform distribution. The T-SNE graph of the dummy sample is shown as below:

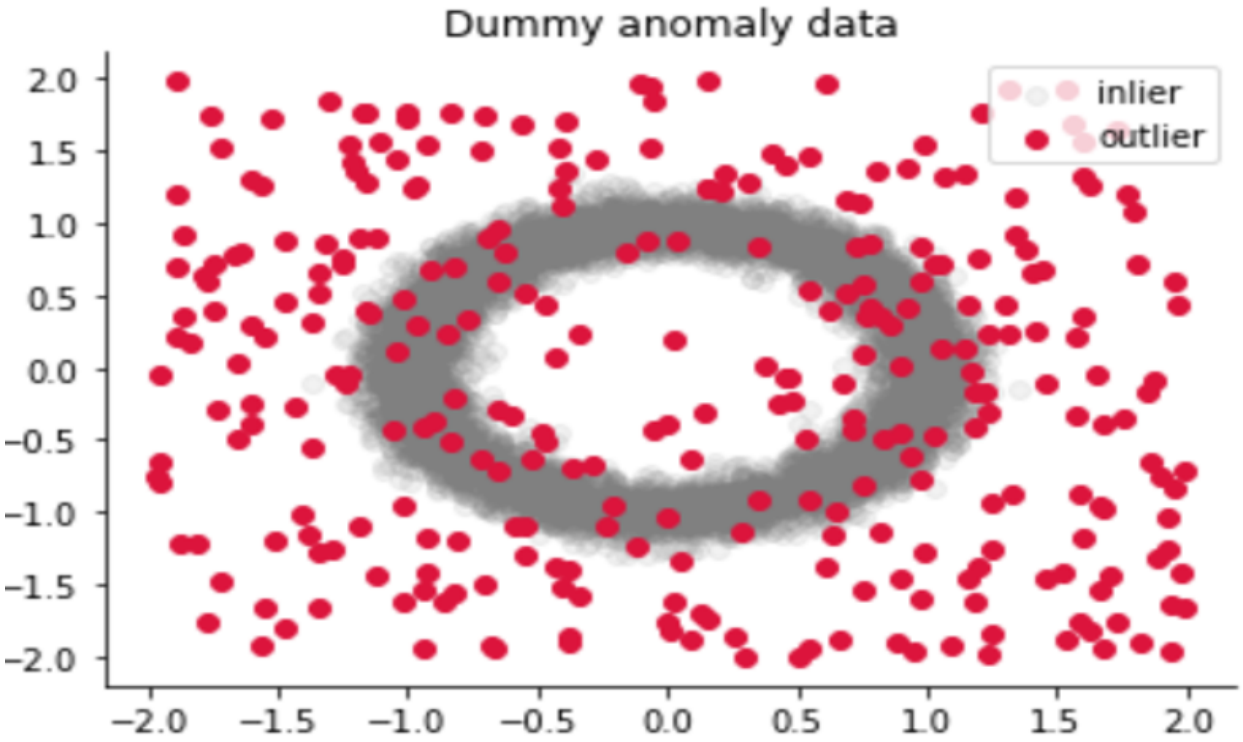


Figure 2: Dummy Data T-SNE

4 Handling imbalanced data - SMOTE

4.a Undersampling

Random under-sampling is a non-heuristic method that aims to balance class distribution through the random elimination of majority class examples. The rationale behind it is to try to balance out the dataset in an attempt to overcome the idiosyncrasies of the machine learning algorithm. The major drawback of random undersampling is that this method can discard potentially useful data that could be important for the induction process.

Below is the algorithm for undersampling technique :

1. Choose random data from the majority class. 2. If the random data's nearest neighbor is the data from the minority class , then remove the data point.

4.b Oversampling

Random over-sampling is a non-heuristic method that aims to balance class distribution through the random replication of minority class examples. Random over-sampling can increase the likelihood of occurring overfitting, since it makes exact copies of the minority class examples. SMOTE generates synthetic minority examples to over-sample the minority class. Its main idea is to form new minority class examples by interpolating between several minority class examples that lie together. For every minority example, its k nearest neighbors of the same class are calculated, then some examples are randomly selected from them according to the over-sampling rate. After that, new synthetic examples are generated along the line between

the minority example and its selected nearest neighbors. Thus, the overfitting problem is avoided and causes the decision boundaries for the minority class to spread further into the majority class space.

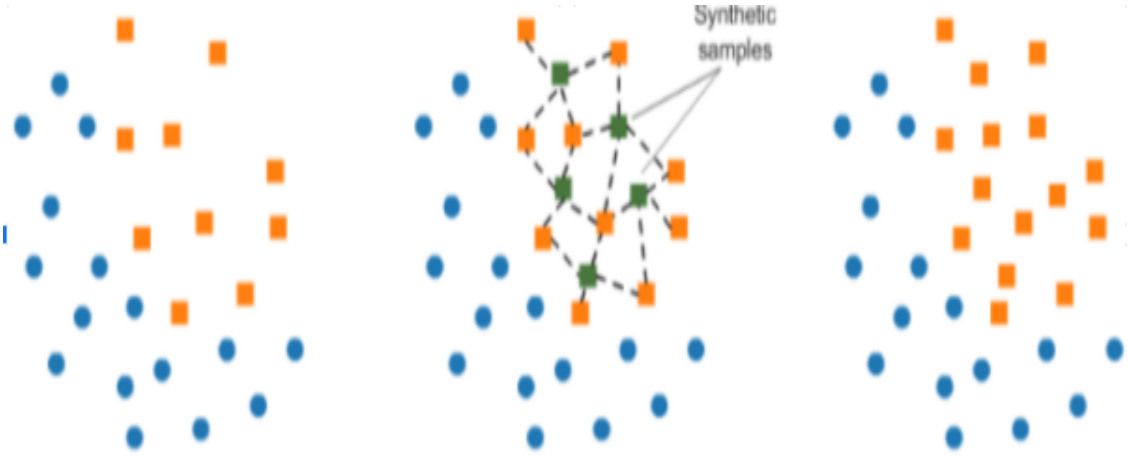


Figure 3: SMOTE technique

Below steps highlights the algorithm for oversampling technique:

1. Two hyperparameters : K - nearest neighbors, Ratio between rare and abundant sample.

2. For i in range (N):

a. Choose random minority point

b. Get K nearest neighbours

c. Choose random nearest neighbour

d. for each dimension of X and Y :

e. sample alpha between [0,1]

f. X = x + alpha(y - x)

g. similarly for Y. Add X,Y as data point

This way new data points are generated in SMOTE technique.

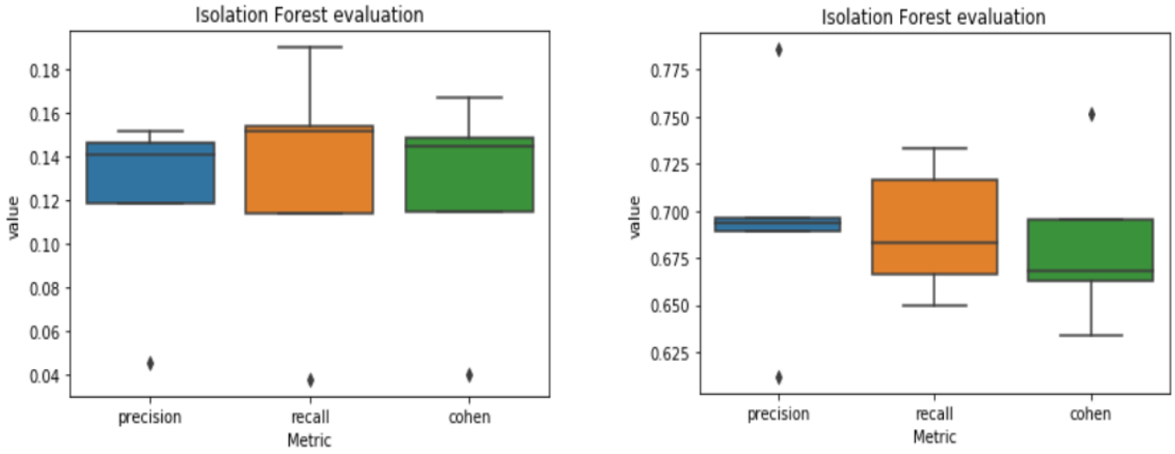
5 The Baseline model

5.a Isolation Forest

Isolation forest is non-parametric method. A decision tree can isolate outlier data points (by splitting the data among features) by much shorter branches than inliers. And for the isolation forest, it randomly select features and split values, recursively partitioning our data. The path length is a measure of inlier-ness: the longer the path, the more regular the data point is.

For Isolation Forest, we feed both credit card data set and dummy data set. Two graphs below show the predictions of two data set. From the dummy data set, the precision and recall metric are good, which

means the Isolation Forest model works for outlier detection. However, the prediction on credit card data set is poor according to the left graph.



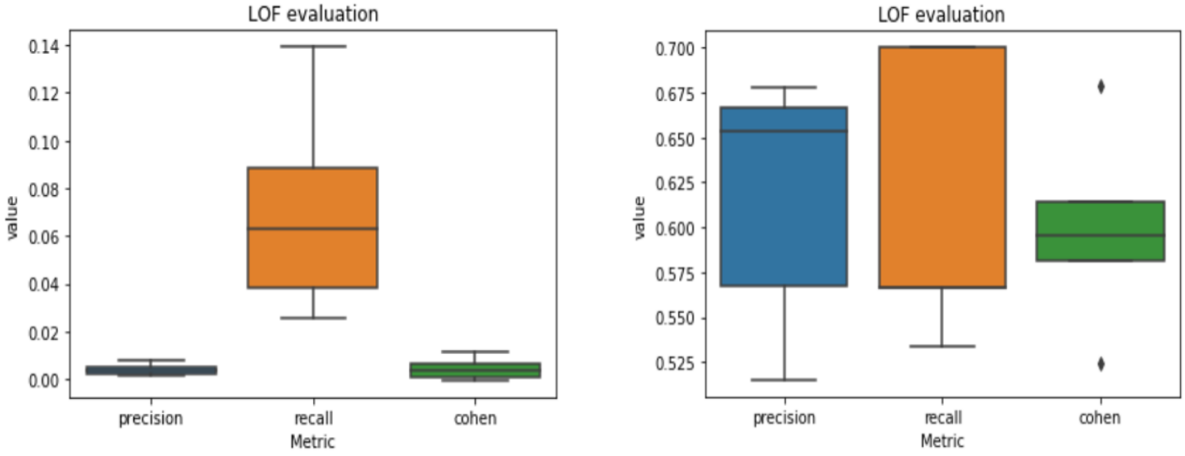
(a) Credit Card (b) Sanity Check

Figure 4: Isolation Forest

5.b Local Outlier Factor

The LOF, Local Outlier Factor, is an Unsupervised anomaly detection algorithm. It defines outliers by doing a density-based scoring and computes the local density deviation of a given data point with respect to its neighbors. Samples having substantially lower density compared to its neighbors are classified as outliers.

The predictions of both two data set are very similar to Isolation Forest. For dummy data set, the prediction is good, however, not good enough for a more complicate data set like Credit Card. The predictions of two baseline model on credit card data set is poor, which means more advanced algorithm should be introduced to detect the outliers of Credit Card data set.



(a) Credit Card (b) Sanity Check

Figure 5: Local Outlier Factor

6 The RandNet Model

A fully connected autoencoder is a special type of Neural Network which performs hierarchical and nonlinear dimensionality reduction of the data. It is layered and has symmetric architecture. The input layer has the highest number of nodes which decreases till the middle layer. The middle layer is the smallest layer. It is called the Bottleneck layer whose output is the reduced form (encoded) data. Autoencoders are typically used for:

- Dimensionality reduction
- Denoising
- Outlier detection

In our paper we will use autoencoders for detecting outliers. The idea behind outlier detection is as follows: Outliers are harder to be accurately represented in reduced form as compared to the inliers. Hence the reconstruction error for the outlier would be large. Hence, by setting an appropriate threshold (hyperparameter), we can detect the outliers. In our paper, we incorporate additional features to our autoencoders to improve performance which are listed in the subsequent subsections.

6.a Neural Network Structure

Instead of having a fully connected autoencoder architecture, we implemented randomly connected autoencoders having different densities. This helped us in two ways: reduced computational complexity and helped make each ensemble as independent as possible. Due to this sparse representation there is a tradeoff: Underfitting vs reduced variation. But we see that the benefits of reduced variation through ensembles implementation far outweighs the cost due to underfitting. it ensured that each ensemle component captured different aspects of the underlying pattern. To generate the random connections, we have used sampling with *replacement*. So if we have l1 connections in the first layer and l2 connections in the second layer, the total number of connections possible would be l1.l2. But with sampling with replacement, some connections will be picked more than once and some will not be picked even once. These 'not picked' connections will be dropped in the implementation of that particular ensemble. This method ensures, that all ensembles are as independent as possible becuase each has a random set of connections with varying densities.

6.b Outlier scoring

For identifying outliers, we have considered the reconstruction loss. Reconstruction loss is the squared sum of error in the reconstruction over the different dimensions of the data point. We have normalized the scores for each ensemble to make them comparable. The final score of a datapoint is the median of the scores across ensembles. We then classified the outliers and inliers based on a selected threshold level.

6.c Training and Adaptive Sampling

Adaptive sampling helps make the optimization procedure more faster and efficient by changing the sample size in each iteration. Initially during the training process, we do not need a very accurate value for gradient but only a general direction for which a small sample size is sufficient. Only as the candidate solution approaches is when we need more data to compute accurate gradients. Therefore we increase the data set with higher epochs. Paper describes 3 ways to increase the sampling size:

1. Linear increase: Turns out to be too slow for a neural network implementation.
2. Start with larger batch size + linear: Slows down the algorithm.
3. Exponential increase: This is the chosen method because with alpha value greater 1, the batch size increases with each epoch.

To implement this, we implemented a custom dataloader. Approach 1: Keeping the batch size constant, increase the number of steps per epoch, to include more data at later epochs. However, this approach had an issue where the fit_generator calls the len() method only once at the beginning of training and then sets the steps_per_epoch constant. To circumvent this problem we had to either:

1. Change the source code in keras to call len() method after each epoch or
2. Implement a Batch Wise Adaptation.

In the second implementation, we changed the number of datapoints in each batch during a particular epoch to accommodate more data points in later epochs.

6.d Ensemble based method

A ensemble of autoencoders were used to improve accuracy and produce more robust results. In order to make ensemble method more effective, the predictors should be independent of each other. This is achieved through creating randomly generated connections for each ensemble.

7 Results

7.a Performance Metrics

7.b SMOTE results

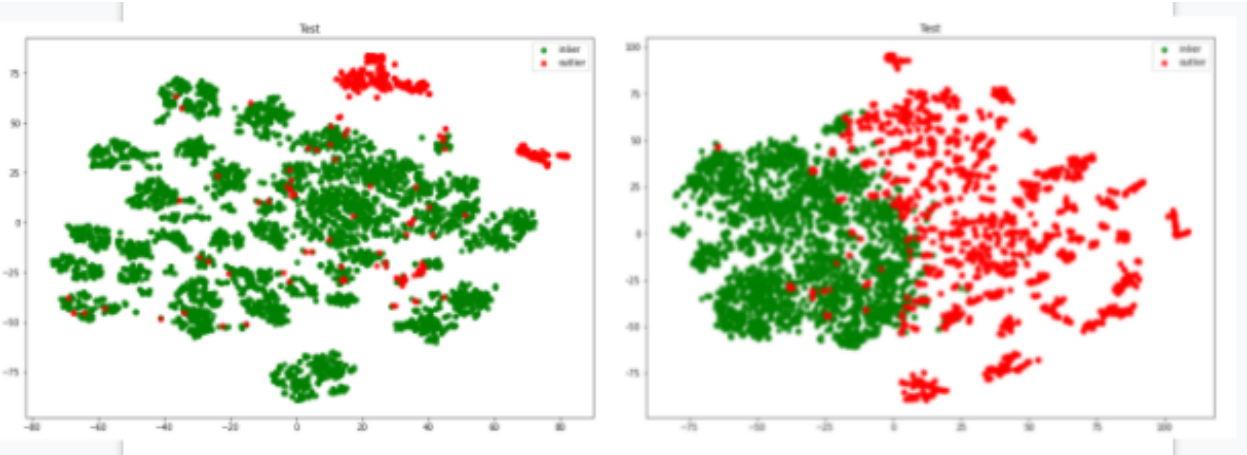
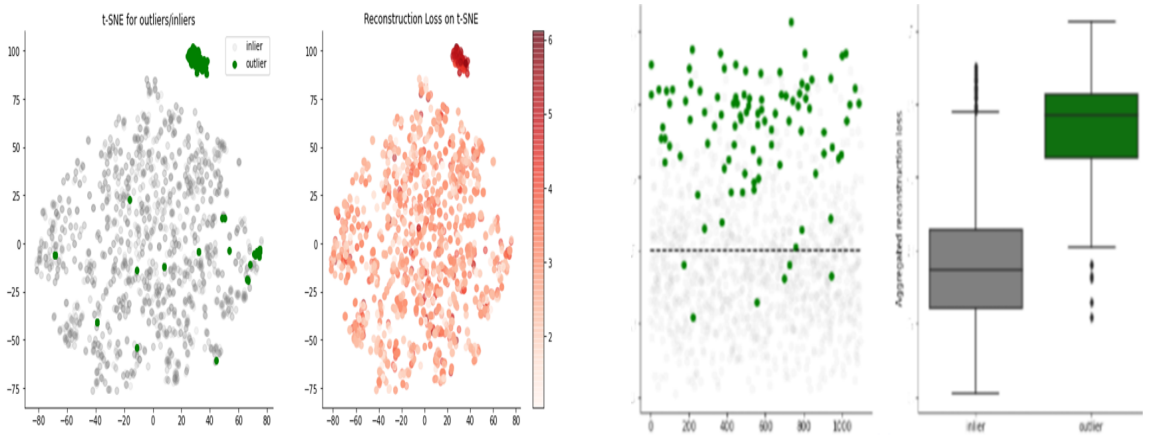


Figure 6: Before and after applying SMOTE

Figure 6 demonstrates the results from running SMOTE against the minority class. This is a very recognisable characteristic of SMOTE-based up-sampling

From the figure it can be seen that number of outlier samples have increased and become equal to inlier sample.

7.c AutoEncoder results



(a) t-SNE and Reconstruction Loss for Outliers (b) Distribution of Reconstruction loss for Inliers and Outliers

Figure 7: Reconstruction Loss Analysis

From the above pictures, we see that the outliers have consistently high reconstruction loss but the main issue seems to be that there are a fair amount of inlier data points with high median reconstruction loss.

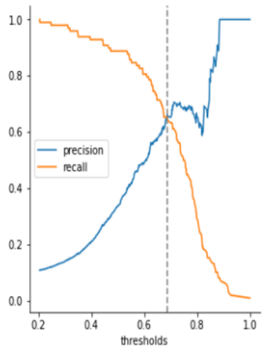


Figure 8: Precision-Recall Tradeoff

From the above graph, we see that there is a trade-off with varying the threshold: Precision vs Recall. The optimum value seems to be 0.65 Precision and 0.65 Recall at around 0.69 threshold. However, with the credit card data we are more interested in detecting fraudulent activities. As such, we will be more tolerant to let a few non fraudulent activities be classified as fraudulent. But we would not want any fraudulent activity to go undetected. Thus, we are **more concerned about Recall than Precision** and could try to improve Recall at the cost of precision.From the above graph we see that we can achieve, 90% recall with 40% precision by setting the threshold to approximately 0.55.

8 Conclusion

Paper shows how to use an ensemble of autoencoders in order to perform anomaly detection. The proposed technique works fine. It uses random edge sampling in conjunction with adaptive data sampling in order to achieve results. Unlike the earlier methods proposed for neural network based outlier detection, our approach is able to avoid overfitting and achieve robustness because of its ensemble-centric approach.

9 Notebook - code

Randomized Autoencoder Ensembles for Unsupervised Outlier Detection

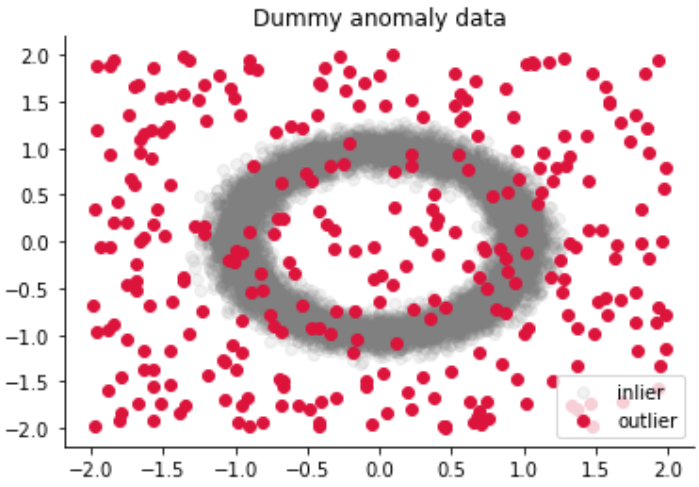
```
!pip install smote_variants
import smote_variants as sv


import numpy as np
import pandas as pd

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.preprocessing import QuantileTransformer, MinMaxScaler, StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, roc_curve, roc_auc_score, precision_rec
from sklearn.manifold import TSNE

from tqdm import tqdm
import pickle
```

Data sets

We will use some dummy data for sanity checks and some real world data too.

```
def generate_dummy(n_samples = 10000,
                   contamination = 0.03,
                   outlier_range = 2,
                   n_features = 2,
                   normalize=False):

    # normal inliers
    dummy_data = np.random.randn(n_samples, n_features)
    if normalize:
        noise = 1+np.random.randn(dummy_data.shape[0], 1)*0.1
        dummy_data = dummy_data/(np.linalg.norm(dummy_data, axis=1).reshape(-1, 1))
        dummy_data *= noise

    # uniform outliers from given range
    n_outliers = int(n_samples*contamination)
    dummy_data[:n_outliers, :] = np.random.uniform(-1*outlier_range, outlier_range, size=(
```

```
    dummy_data = pd.DataFrame(dummy_data, columns = range(n_features))
    dummy_data["Class"] = np.concatenate([np.ones((n_outliers,)), np.zeros((n_samples-n_ou

    return dummy_data

dummy_data = generate_dummy(normalize=True)


plt.figure()
plt.scatter(dummy_data.loc[dummy_data["Class"] == 0, 0],
            dummy_data.loc[dummy_data["Class"] == 0, 1], c="grey", alpha=0.1, label="inlie
plt.scatter(dummy_data.loc[dummy_data["Class"] == 1, 0],
            dummy_data.loc[dummy_data["Class"] == 1, 1], c="crimson", alpha=1, label="outl
plt.legend()
sns.despine()
plt.title("Dummy anomaly data")
plt.show()
```




```
dummy_data.Class.mean()
```

```
0.03
```

We will work with credit card data from [Wordline & ULB](#).

```
data = pd.read_csv("creditcard.csv")
data.head()
```

	Time	V1	V2	V3	V4	V5	V6	V7	
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.09
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.08
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.24
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.37
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.27

```
data.shape
```

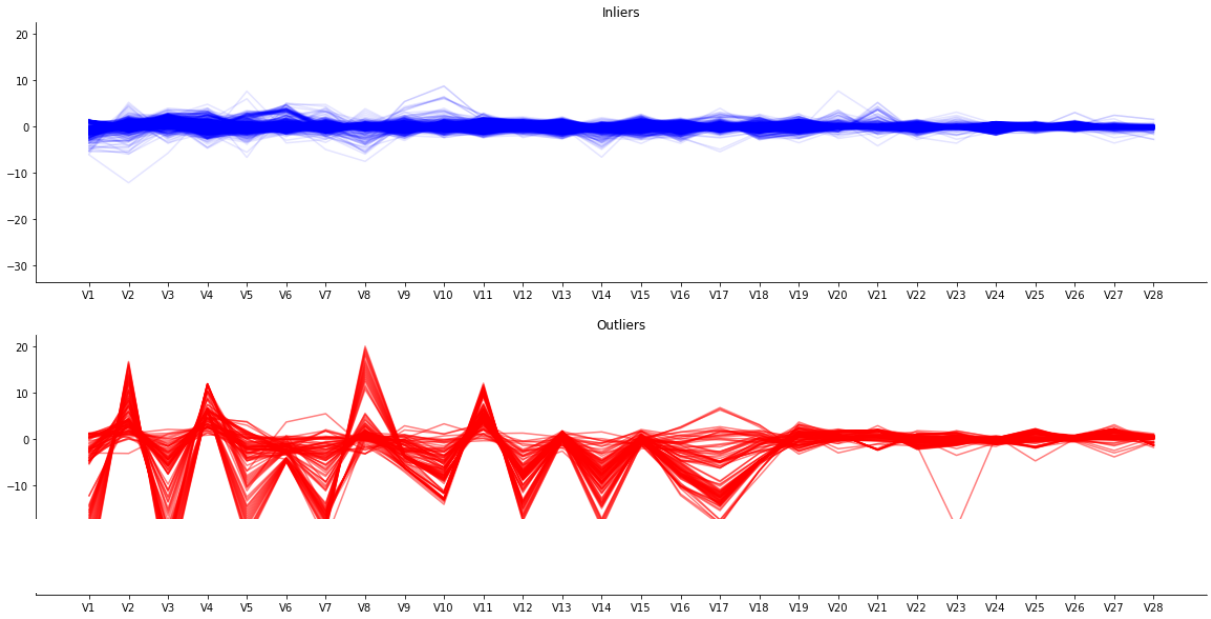
```
(284807, 31)
```

The proportion of outliers is around 0.17%:

```
data["Class"].mean()
```

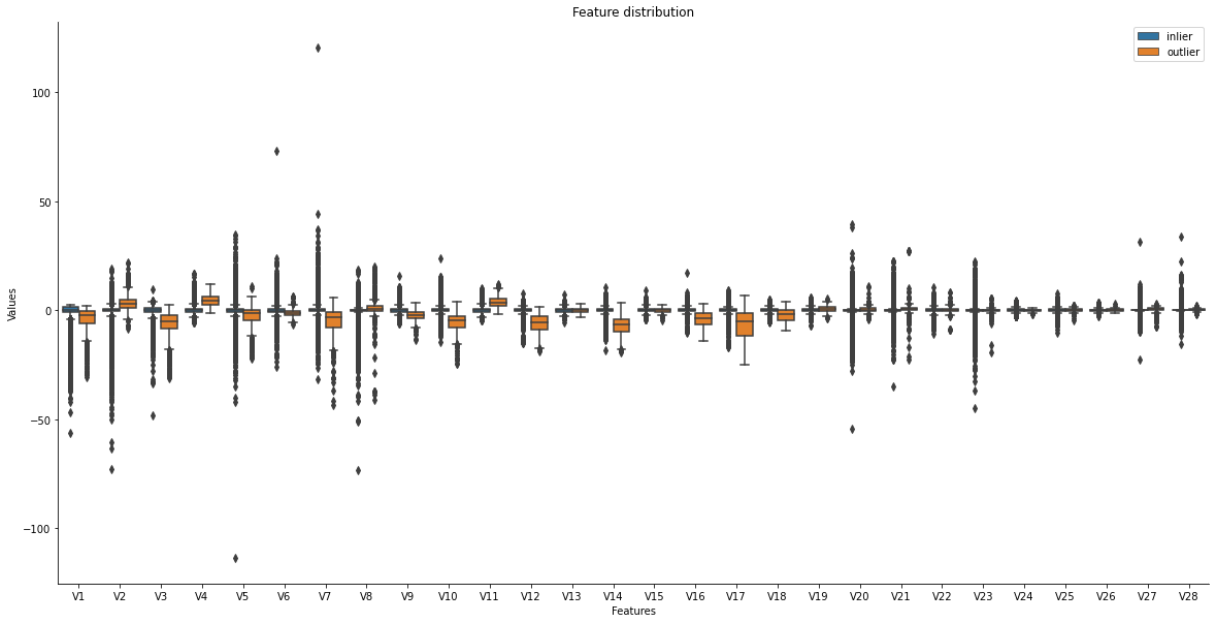
```
0.001727485630620034
```

```
feature_cols = [col for col in data.columns if col.startswith("V")]
# feature_cols = data.columns
inlier_data = data[data.Class == 0].reset_index()
outlier_data = data[data.Class == 1].reset_index()

fig, axs = plt.subplots(2, 1, figsize=(20, 10), sharey=True)
axs[0].plot(inlier_data.loc[:1000, feature_cols].T, c="blue", alpha=0.1)
axs[0].set_title("Inliers")
axs[1].plot(outlier_data.loc[:100, feature_cols].T, c="red", alpha=0.5)
axs[1].set_title("Outliers")
sns.despine()
plt.show()
```



The outliers are definitely showing different behaviour, much higher variance in many features. There also seems to some clustering among the inliers (around V5/6 there seems to be a distict group).

```
plot_data = data.melt(value_vars=feature_cols,
                      id_vars=["Class"],
                      var_name='groups',
                      value_name='vals')


fig, axs = plt.subplots(1, 1, figsize=(20, 10))
sns.boxplot(x="groups", y="vals", hue="Class", data=plot_data, ax=axs)
sns.despine()
plt.xlabel("Features")
plt.ylabel("Values")
L=plt.legend()
L.get_texts()[0].set_text('inlier')
L.get_texts()[1].set_text('outlier')
plt.title("Feature distribution")
plt.show()
```



The data is centered and the variance is pretty much on the same scale for these features (not showing `Time` and `Amount`).

```
feature_stats = data.drop("Class", axis=1).apply([np.std, np.mean]).T
feature_stats
```

	std	mean
Time	47488.145955	9.481386e+04
V1	1.958696	3.919560e-15
V2	1.651309	5.688174e-16
V3	1.516255	-8.769071e-15
V4	1.415869	2.782312e-15
V5	1.380247	-1.552563e-15
V6	1.332271	2.010663e-15
V7	1.237094	-1.694249e-15
V8	1.194353	-1.927028e-16
V9	1.098632	-3.137024e-15
V10	1.088850	1.768627e-15
V11	1.020713	9.170318e-16
V12	0.999201	-1.810658e-15
V13	0.995274	1.693438e-15
V14	0.958596	1.479045e-15
V15	0.915316	3.482336e-15
V16	0.876253	1.392007e-15
V17	0.849337	-7.528491e-16
V20	0.770925	5.085503e-16

Time and amount are on very different scales then the other (PCA) features of the data.

V22	0.725702	7.959909e-16

Data scaling for training

V24	0.605647	4.453112e-15

We will do just some basic preprocessing to standardize the data and also to map it between 0 and 1 (which is required by our autoencoder implementation with the final sigmoid layer).

V27	0.403632	3.660161e-16

```
from·sklearn.preprocessing·import·QuantileTransformer,·MinMaxScaler
from·sklearn.pipeline·import·Pipeline
from·sklearn.base·import·TransformerMixin


class·MyRobustScaler(TransformerMixin):
····def·__init__(self,·low_q=0.1,·high_q=0.9):
········self.low_q·=·low_q
```

```
        self.low_q = low_q
        self.high_q = high_q

        self.feature_low_bound = None
        self.feature_high_bound = None

    def fit(self, X):
        """
        Record per-feature quantiles.
        """
        self.feature_low_bound, self.feature_high_bound = np.quantile(X, [self.low_q, self

        return self

    def transform(self, X):
        """
        Limit the data between the quantiles.
        """
        X_transformed = np.where(X <= self.feature_low_bound, self.feature_low_bound, X)
        X_transformed = np.where(X_transformed >= self.feature_high_bound, self.feature_hi

        return X_transformed

qs_mm_scaler = Pipeline([
                    ("quantile_transform", QuantileTransformer(output_distribution="normal
                    ("minmax", MinMaxScaler())
])

mm_scaler = Pipeline([("minmax", MinMaxScaler())])

own_scaler = Pipeline([("own_robust", MyRobustScaler()),
                       ("minmax", MinMaxScaler())])
```

```
feature_cols = data.columns[:30]
inlier_sample = data.loc[data.Class == 0, feature_cols].sample(5000).values
outliers = data.loc[data.Class == 1, feature_cols].values


scaler = own_scaler
vis_data = np.concatenate([inlier_sample, outliers], axis=0)
# vis_data = scaler.transform(vis_data)

vis_labels = np.concatenate([np.zeros((inlier_sample.shape[0],)), np.ones((outliers.shape[
vis_data.shape, vis_labels.shape

from sklearn.utils import shuffle
X, y = shuffle(vis_data, vis_labels, random_state=0)


inlier_sample.shape
```

```
(5000, 30)
```

```
# inlier_sample.shape
data.columns[:30]
```

```
Index(['Time', 'V1', 'V2', 'V3', 'V4', 'V5', 'V6', 'V7', 'V8', 'V9', 'V10',
       'V11', 'V12', 'V13', 'V14', 'V15', 'V16', 'V17', 'V18', 'V19', 'V20',
       'V21', 'V22', 'V23', 'V24', 'V25', 'V26', 'V27', 'V28', 'Amount'],
      dtype='object')
```

```
data.loc[data.Class == 0, feature_cols].sample(5000).values.shape
```

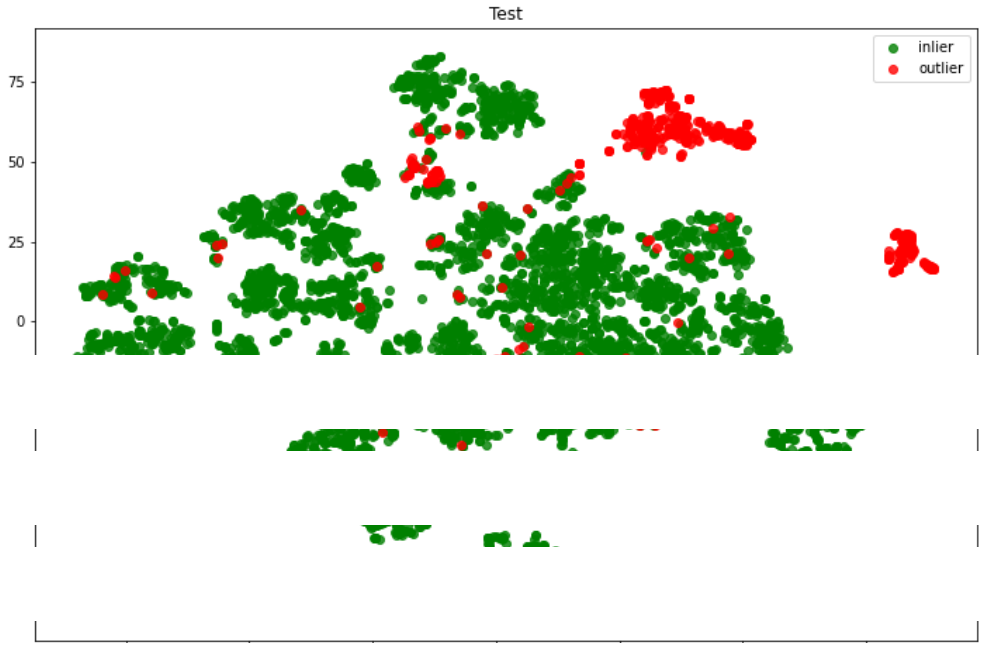
```
(5000, 30)
```

```
vis_labels
```

```
array([0., 0., 0., ..., 1., 1., 1.])
```

```
def tsne_plot(x1, y1, name="graph.png", titleStr= 'Test'):
    tsne = TSNE(n_components=2, random_state=101)
    X_t = tsne.fit_transform(x1)

    plt.figure(figsize=(12, 8))
    plt.scatter(X_t[np.where(y1 == 0), 0], X_t[np.where(y1 == 0), 1], marker='o', color='g
    plt.scatter(X_t[np.where(y1 == 1), 0], X_t[np.where(y1 == 1), 1], marker='o', color='r

    plt.legend(loc='best')

    plt.title(titleStr)
    plt.savefig(name)
    plt.show()
```

```
tsne_plot(X,y)
```




```
scaler = own_scaler
```

```
#X = data.drop("Class", axis=1).values
#y = data["Class"].values

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=123)
```

```
X.shape
```

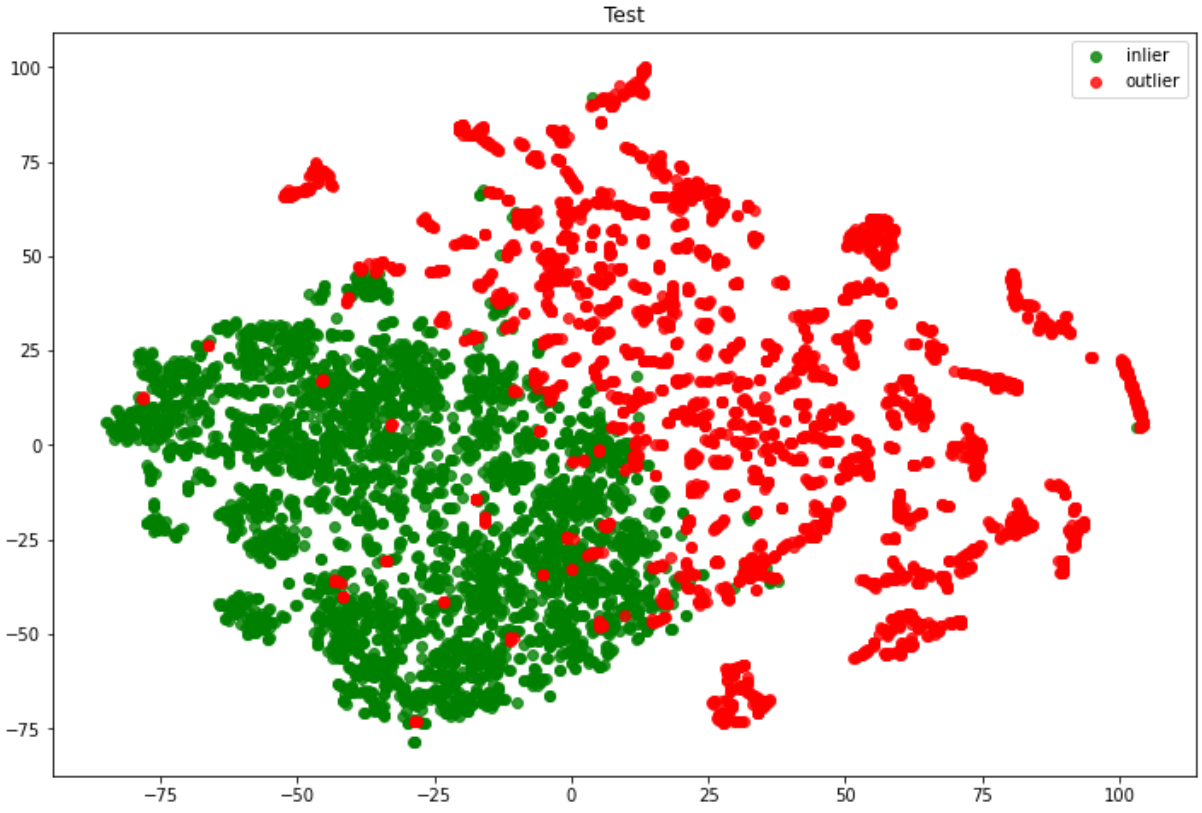
```
(5492, 30)
```

```
oversampler = sv.SMOTE(random_state=1234)
X_bal, Y_bal= oversampler.sample(X_train, y_train)
```

```
X_bal = scaler.fit_transform(X_bal)
X_test = scaler.transform(X_test)
X_train = scaler.fit_transform(X_train)

y_train = y_train.reshape(-1, 1)
Y_bal = Y_bal.reshape(-1, 1)
y_test = y_test.reshape(-1, 1)
```

```
2021-10-09 21:49:12,866:INFO:SMOTE: Running sampling via ('SMOTE', "{'proportion': 1
```

```
tsne_plot(X_bal, Y_bal)
```




```
X_train.shape, y_train.shape, X_test.shape, y_test.shape
```

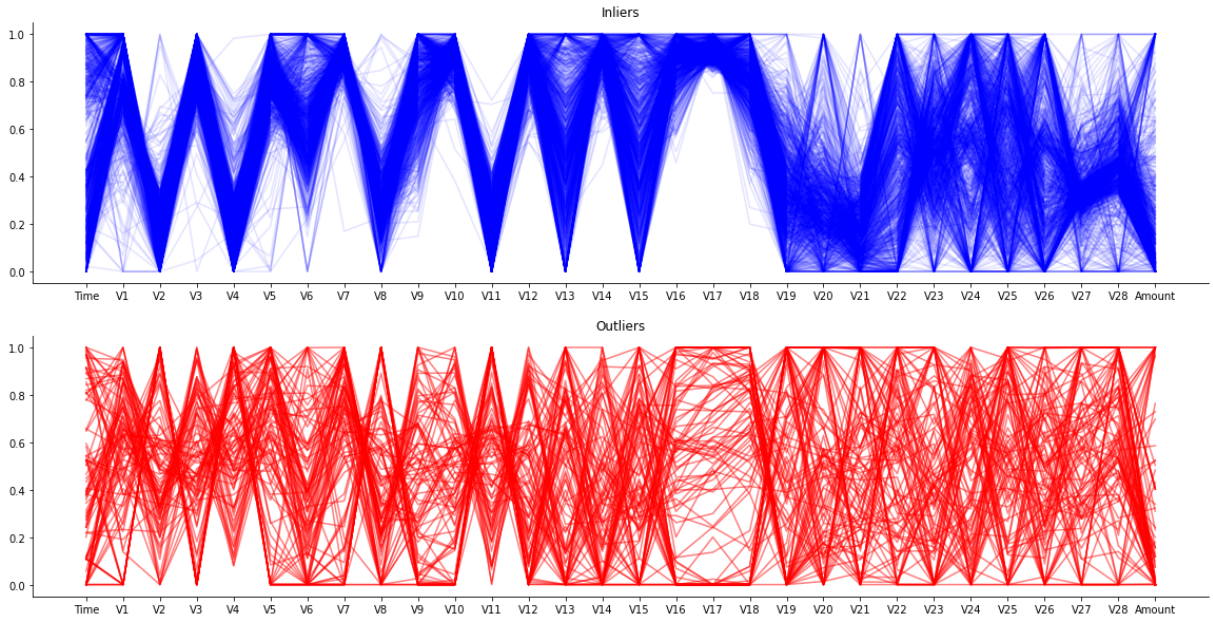
```
((4393, 30), (4393, 1), (1099, 30), (1099, 1))
```

```
y_train.sum(), y_test.sum()
```

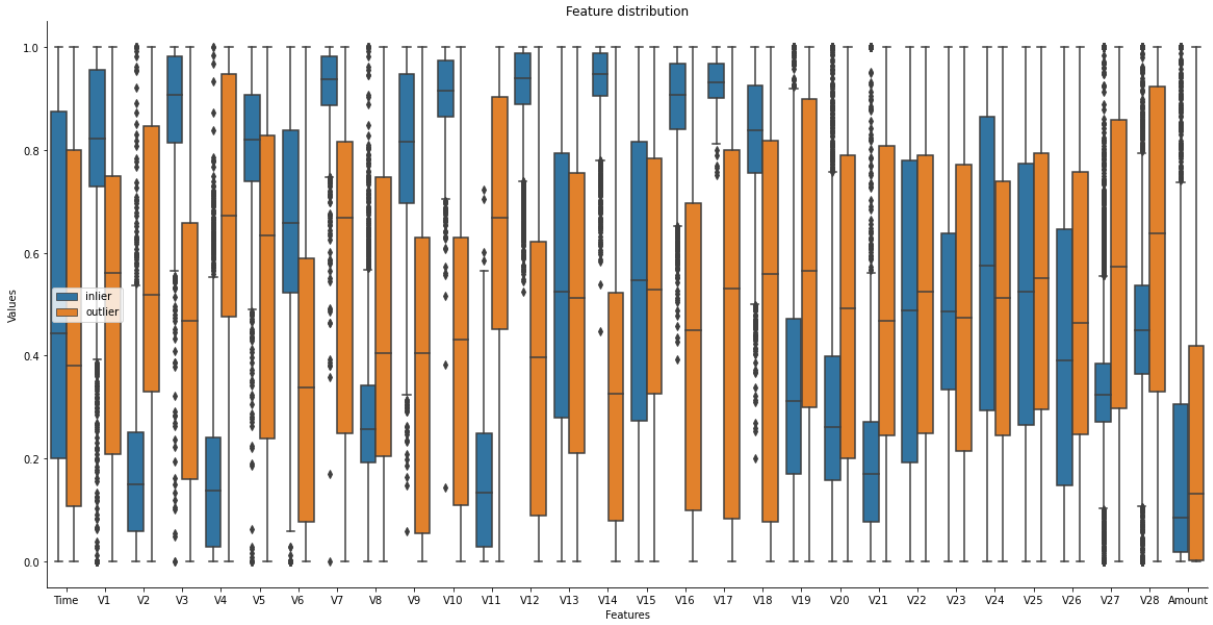
```
(383.0, 109.0)
```

```
scaled_data = pd.DataFrame(np.concatenate([X_bal, Y_bal], axis=1), columns=data.columns)
```

```
feature_cols = list(data.drop("Class", axis=1).columns)

inlier_data = scaled_data[scaled_data.Class == 0].reset_index()
outlier_data = scaled_data[scaled_data.Class == 1].reset_index()

fig, axs = plt.subplots(2, 1, figsize=(20, 10), sharey=True)
axs[0].plot(inlier_data.loc[:1000, feature_cols].T, c="blue", alpha=0.1)
axs[0].set_title("Inliers")
axs[1].plot(outlier_data.loc[:100, feature_cols].T, c="red", alpha=0.5)
axs[1].set_title("Outliers")
sns.despine()
plt.savefig("wordline_features.png")
plt.show()
```




```
plot_data = scaled_data.melt(value_vars=feature_cols,
                    id_vars=["Class"],
                    var_name='groups',
                    value_name='vals')


fig, axs = plt.subplots(1, 1, figsize=(20, 10))
sns.boxplot(x="groups", y="vals", hue="Class", data=plot_data, ax=axs)
sns.despine()
plt.xlabel("Features")
plt.ylabel("Values")
L=plt.legend()
L.get_texts()[0].set_text('inlier')
L.get_texts()[1].set_text('outlier')
plt.title("Feature distribution")
plt.show()
```



It is clear that the outliers have a very different marginal distribution on most features. This hopefully is a good sign that our scaling didn't destort too much the distinctive features of the outliers.

t-SNE visualization

Before the modeling, we can visualize some of our data. We will pick 5000 inliers and all the outliers and see if there are some patterns.

```
inlier_sample = data.loc[data.Class == 0, feature_cols].sample(5000).values
outliers = data.loc[data.Class == 1, feature_cols].values

vis_data = np.concatenate([inlier_sample, outliers], axis=0)
vis_data = scaler.transform(vis_data)

vis_labels = np.concatenate([np.zeros((inlier_sample.shape[0],)), np.ones((outliers.shape[
vis_data.shape, vis_labels.shape
```

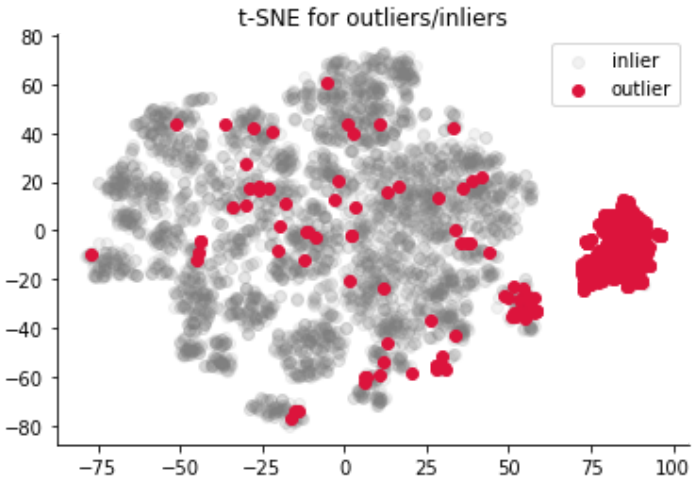
```
((5492, 30), (5492,))
```

```
vis_data
```

```
array([[0.87882408, 1.        , 0.33444285, ..., 0.31407693, 0.22882343,
        0.        ],
       [0.14769587, 0.73763343, 0.27807217, ..., 0.45005742, 0.51955761,
        0.40820528],
       [0.04173586, 0.29786989, 0.01492932, ..., 0.37062962, 0.74849076,
        1.        ],
       ...,
       [1.        , 0.39560364, 0.79423671, ..., 1.        , 0.93129777,
        0.36808847],
       [1.        , 0.        , 0.61886063, ..., 1.        , 0.        ,
        1.        ],
       [1.        , 0.99661508, 0.4801864 , ..., 0.36731881, 0.3825311 ,
        0.19881277]])
```

```
from sklearn.manifold import TSNE

tsne = TSNE(n_components=2)
tsne_data = tsne.fit_transform(vis_data)
```

```
plt.figure()
plt.scatter(tsne_data[vis_labels == 0, 0],
            tsne_data[vis_labels == 0, 1], c="grey", alpha=0.1, label="inlier")
plt.scatter(tsne_data[vis_labels == 1, 0],
            tsne_data[vis_labels == 1, 1], c="crimson", alpha=1, label="outlier")
plt.legend()
sns.despine()
plt.title("t-SNE for outliers/inliers")
plt.savefig("tsne_wordline.png")
plt.show()
```



There is definitely strong separation among some of the outliers. But keeping in mind that we would like to do **unsupervised** outlier detection, it is not clear how we would tell apart the outlier

clusters from some of the inlier blobs on the edge of the data (there is a fair amount of variation on the resulting plot here).

```
dummy_X = dummy_data.drop("Class", axis=1).values
dummy_y = dummy_data["Class"].values
```

Local Outlier Factors

This is a non-parametric, density-based method that finds outliers by looking for data points which have significantly lower local densities than their neighbouring points. The rough algorithm is as follows:

1. Find the k-nearest neighbors and use the distance to estimate a local density;
2. Compare local densities with neighbours to detect outliers.

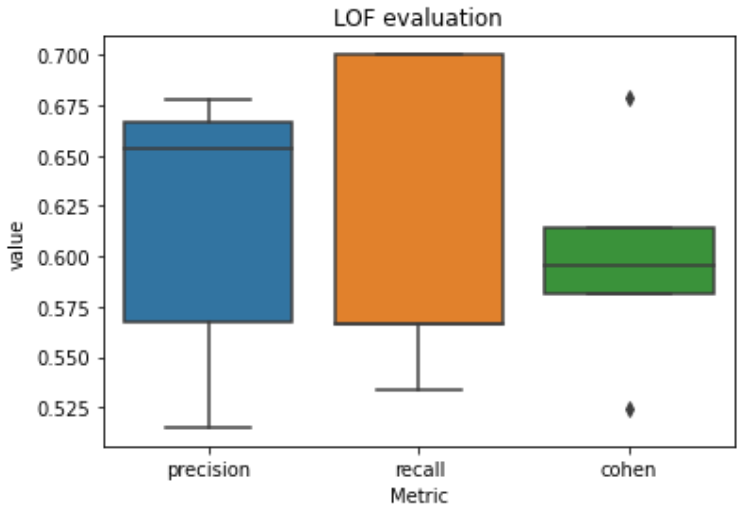
LOF being a distance-based method, it's important that our data is scaled. See more here.

Running this on the full data set takes a lot of time.

```
%%time

from sklearn.neighbors import LocalOutlierFactor

model = LocalOutlierFactor(n_neighbors=5, contamination=y.mean(), n_jobs=-1) # n_neighbour

results = eval_model(model, X_train, y_train)


plt.figure()
sns.boxplot(x="variable", y="value", data=results.melt())
plt.xlabel("Metric")
plt.title("LOF evaluation")
plt.show()
```

Again, the dummy data:

```
model = LocalOutlierFactor(n_neighbors=10, contamination=dummy_y.mean(), n_jobs=-1, novelt

dummy_results = eval_model(model, dummy_X, dummy_y)



plt.figure()
sns.boxplot(x="variable", y="value", data=dummy_results.melt())
plt.xlabel("Metric")
plt.title("LOF evaluation")
plt.show()
```



Randomized autoencoders

The basic idea of our approach is to randomly drop connections in a neural network, and yet retain a certain level of control on the connection density between various layers and also create models with different types of densities

We will build an autoencoder class that can mask connections in its layers.

```
class RandAE(tf.keras.Sequential):
    def __init__(self, input_dim, hidden_dims, drop_ratio=0.5, **kwargs):
        super(RandAE, self).__init__(**kwargs)

        self.input_dim = input_dim
        self.hidden_dims = hidden_dims
        self.drop_ratio = drop_ratio

        self.layer_masks = dict()

        self.build_model()

    def build_model(self) -> None:
        """
        Adds the layers and records masks.
        """

        self.add(layers.Input(self.input_dim, name="input"))

        for i, dim in enumerate(self.hidden_dims):
            layer_name = f"hidden_{i}"
            layer = layers.Dense(dim,
                                 activation="relu" if i > 0 else "sigmoid",
                                 name=layer_name)
            self.add(layer)
```

```
        # add layer mask
        self.layer_masks[layer_name] = self.get_mask(layer)

    layer_name = "output"
    output_layer = layers.Dense(self.input_dim, activation="sigmoid", name=layer_name)
    self.add(output_layer)
    self.layer_masks[layer_name] = self.get_mask(output_layer)

def get_mask(self, layer) -> np.ndarray:
    """
    Build mask for a layer.
    """

    shape = layer.input_shape[1], layer.output_shape[1]

    return np.random.choice([0., 1.], size=shape, p=[self.drop_ratio, 1-self.drop_rati

def load_masks(self, mask_pickle_path) -> None:
    """
    Load the masks from a pickled dictionary.
    """

    with open(mask_pickle_path, 'rb') as handle:
        self.layer_masks = pickle.load(handle)

def get_encoder(self) -> keras.Sequential:
    """
    Get the encoder from the full model.
    """

    n_layers = (len(self.hidden_dims)+1)//2
    encoder_layers = [layers.Input(self.input_dim)] + self.layers[:n_layers]

    return keras.Sequential(encoder_layers)


def mask_weights(self) -> None:
    """
    Apply the masks to each layer in the encoder and decoder.
    """

    for layer in self.layers:
        layer_name = layer.name
        if layer_name in self.layer_masks:
            masked_w = layer.weights[0].numpy()*self.layer_masks[layer_name]
            b = layer.weights[1].numpy()
            layer.set_weights((masked_w, b))

def call(self, data, training=True) -> tf.Tensor:

    # mask the weights before original forward pass
    self.mask_weights()

    return super().call(data)
```

We can preform some sanity checks on our model:

```
# test compile
model = RandAE(32, [16, 8, 4, 8, 16])
model.compile(optimizer="adam", loss="mse", run_eagerly=True)
```

```
# structure
model.summary()
```

```
Model: "rand_ae"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
hidden_0 (Dense)             (None, 16)                528
_________________________________________________________________
hidden_1 (Dense)             (None, 8)                 136
_________________________________________________________________
hidden_2 (Dense)             (None, 4)                 36
_________________________________________________________________
hidden_3 (Dense)             (None, 8)                 40
_________________________________________________________________
hidden_4 (Dense)             (None, 16)                144
_________________________________________________________________
output (Dense)               (None, 32)                544
=================================================================
Total params: 1,428
Trainable params: 1,428
Non-trainable params: 0
_________________________________________________________________
```

```
# check weights before first call
for layer in model.layers:
    pct = np.round((layer.weights[0].numpy() == 0).mean()*100, 2)
    print(f"{layer.name} has {pct}% exact-0 weights.")
```

```
 hidden_0 has 0.0% exact-0 weights.
 hidden_1 has 0.0% exact-0 weights.
 hidden_2 has 0.0% exact-0 weights.
 hidden_3 has 0.0% exact-0 weights.
 hidden_4 has 0.0% exact-0 weights.
 output has 0.0% exact-0 weights.
```

```
# test forward with sample data
data = np.random.randn(128, 32)
output = model(data)
```

```
# print 0 weight ratios again
for layer in model.layers:
    pct = np.round((layer.weights[0].numpy() == 0).mean()*100, 2)
    print(f"{layer.name} has {pct}% exact-0 weights.")
```

```
 hidden_0 has 50.78% exact-0 weights.
 hidden_1 has 46.09% exact-0 weights.
```

```
hidden_2 has 40.62% exact-0 weights.
hidden_3 has 62.5% exact-0 weights.
hidden_4 has 46.88% exact-0 weights.
output has 49.22% exact-0 weights.
```

The weights look good!

Outliers on the latent manifold

We can do a quick EDA-type test run and see if our model can learn a better latent represenation of the data. Note that

- our sigmoid activation in the last layer require the data to be scaled between 0 and 1;
- we set `drop_ratio=0` so we have a regular autoencoder.

```
model = RandAE(X_bal.shape[1], [16], drop_ratio=0) # shallow AE
model.compile(optimizer="adam", loss="mse", run_eagerly=True)

print(f"Baseline loss: {np.square(X_bal - X_bal.mean(axis=0)).mean()}")

history = model.fit(X_bal, Y_bal, epochs=5, batch_size=128)
```

```
Baseline loss: 0.10141651784080594
Epoch 1/5
63/63 [==============================] - 1s 9ms/step - loss: 0.2448
Epoch 2/5
63/63 [==============================] - 1s 10ms/step - loss: 0.2172
Epoch 3/5
63/63 [==============================] - 1s 10ms/step - loss: 0.1757
Epoch 4/5
63/63 [==============================] - 1s 10ms/step - loss: 0.1366
Epoch 5/5
63/63 [==============================] - 1s 10ms/step - loss: 0.1099
```

The baseline MSE shows that there was definitely a fair amount of learning.

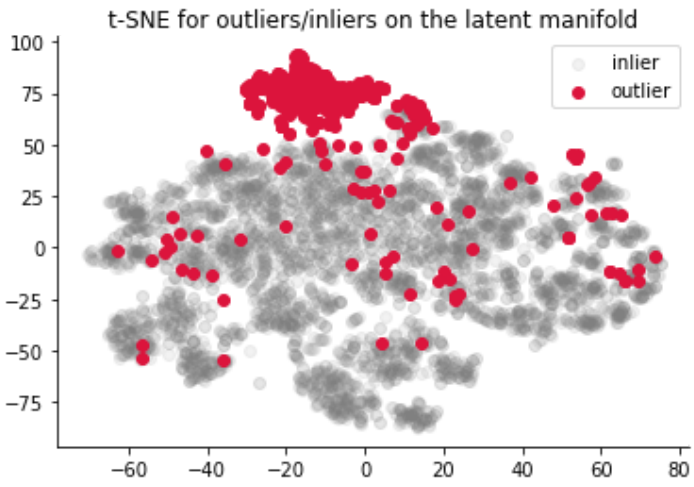
Now we can use the encoder to get the latent representation of the plot data:

```
vis_data
```

```
array([[0.87882408, 1.        , 0.33444285, ..., 0.31407693, 0.22882343,
        0.        ],
       [0.14769587, 0.73763343, 0.27807217, ..., 0.45005742, 0.51955761,
        0.40820528],
       [0.04173586, 0.29786989, 0.01492932, ..., 0.37062962, 0.74849076,
        1.        ],
       ...,
       [1.        , 0.39560364, 0.79423671, ..., 1.        , 0.93129777,
        0.36808847],
       [1.        , 0.        , 0.61886063, ..., 1.        , 0.        ,
```

```
            1.        ],
           [1.        , 0.99661508, 0.4801864 , ..., 0.36731881, 0.3825311 ,
            0.19881277]])
```

```
encoder = model.get_encoder()

vis_data_latent = encoder.predict(vis_data)

tsne = TSNE(n_components=2)
tsne_data = tsne.fit_transform(vis_data_latent)


plt.figure()
plt.scatter(tsne_data[vis_labels == 0, 0],
            tsne_data[vis_labels == 0, 1], c="grey", alpha=0.1, label="inlier")
plt.scatter(tsne_data[vis_labels == 1, 0],
            tsne_data[vis_labels == 1, 1], c="crimson", alpha=1, label="outlier")
plt.legend()
sns.despine()
plt.title("t-SNE for outliers/inliers on the latent manifold")
plt.savefig("tsne_wordline_latent_2.png")
plt.show()
```



The image changed somewhat but pretty similar overall: note that this was plotted based on the 16-dim latent representation instead the original features.

Detecting outliers based on reconstruction loss

We can look for the outliers using our autoencoder by selecting the points with the largest reconstruction loss.

```
input_data = X_test
input_labels = y_test

# make ensemble predictions
```

```
pred = model.predict(input_data)

# calculate reconstruction MSE
reconstruction_loss = np.square(pred - input_data).mean(axis=1)

# set threshold by the population contamination
threshold = np.quantile(reconstruction_loss, 1-input_labels.mean())
outlier_pred = np.where(reconstruction_loss > threshold, 1, 0)

# evaluate
cr = classification_report(input_labels, outlier_pred)
print(cr)
```

```
                   precision    recall  f1-score   support

              0.0       0.96      0.96      0.96      1001
              1.0       0.61      0.61      0.61        98

         accuracy                           0.93      1099
        macro avg       0.79      0.79      0.79      1099
     weighted avg       0.93      0.93      0.93      1099
```

A more wholesome picture comes from looking at the full Precision-Recall curve.

```
# we minmax scale our loss to resemble probabilities
min_ = reconstruction_loss.min()
max_ = reconstruction_loss.max()
scaled_loss = (reconstruction_loss - min_)/(max_ - min_)

prec, recall, thresholds = precision_recall_curve(input_labels, scaled_loss)


plt.figure()
plt.plot(recall, prec)
plt.xlabel("recall")
plt.ylabel("precision")
sns.despine()
plt.title("PR curve")
plt.show()
```

Finally, we can look at Cohen's kappa. This is a measure of agreement between two arrays that takes into account by-chance equality. The values range between 1 and -1 with the former being the best score.

```
cohen_kappa_score(input_labels, outlier_pred)
```

```
0.5742828599971457
```


Data loaders for adaptive learning

We want to change our training data as we progress during training and feed more and more samples to the model in later epochs. For this, we will write a custom data loader.

```
from tensorflow import keras
import warnings, math


## NOTE: does not work as expected - see comments below
class AdaptiveDataGenerator(keras.utils.Sequence):
    def __init__(self, x, batch_size, alpha=1.01, subsample=0.3, shuffle=True, verbose=Fal
        self.x = x
        self.subsample = subsample
        self.verbose = verbose

        if self.subsample:
            sample_idx = np.random.choice(self.x.shape[0], size=int(self.subsample*self.x.
            self.x = self.x[sample_idx]

        self.batch_size = batch_size

        # adaptive learning params
        self.alpha = alpha
        if self.alpha <= 1:
            raise warnings.warn("Alpha should be set to > 1 to increase training data.")
        self.shuffle = shuffle

        # per epoch variables
        self.epoch = 0
        self.train_ratio = 0.5
        self.current_x = None
        self.on_epoch_end()
```

```
    def __len__(self):
        return math.ceil(len(self.current_x) / self.batch_size)

    def __getitem__(self, idx):
        """
        Return a batch for autoencoder training.
        """

        batch = self.current_x[idx*self.batch_size: (idx+1)*self.batch_size]

        return batch, batch

    def on_epoch_end(self):
        """
        Called before training and after every epoch to include more and more training dat
        """

        # slice training data for the next epoch
        slice_idx = int(self.train_ratio*self.x.shape[0])
        self.current_x = self.x[:slice_idx]

        # shuffle rows to mix data in different batches
        if self.shuffle:
            rand_idx = np.arange(self.current_x.shape[0])
            np.random.shuffle(rand_idx)
            self.current_x = self.current_x[rand_idx]

        if self.verbose:
            print(f"Epoch {self.epoch+1} -- {self.current_x.shape[0]/self.x.shape[0]*100}%

        # update training params
        self.train_ratio = min(self.train_ratio*self.alpha, 1)
        self.epoch += 1
```

Some sanity checks:

```
import unittest

class TestAdaptiveDataGenerator(unittest.TestCase):
    def setUp(self):
        self.X_bal = np.random.randn(1000, 10)
        self.batch_size = 50
        self.generator = AdaptiveDataGenerator(self.X_bal, self.batch_size, subsample=0.5,
```

```
        self.model = RandAE(input_dim=10, hidden_dims=[16, 8, 16])
        self.model.compile(optimizer="adam", loss="mse", run_eagerly=True)

    def test_fit(self):
        self.model.fit_generator(generator=self.generator,
                                 epochs=50,
                                 workers=1)

unittest.main(argv=[''], verbosity=2, exit=False)
```

```
test_fit (__main__.TestAdaptiveDataGenerator) ... Epoch 1 -- 50.0% data
Epoch 1/50
5/5 [==============================] - 0s 14ms/step - loss: 1.2151
/usr/local/lib/python3.7/dist-packages/keras/engine/training.py:1972: UserWarning:
  warnings.warn('`Model.fit_generator` is deprecated and '
Epoch 2 -- 60.0% data
Epoch 2/50
5/5 [==============================] - 0s 13ms/step - loss: 1.1953
Epoch 3 -- 72.0% data
Epoch 3/50
5/5 [==============================] - 0s 13ms/step - loss: 1.2292
Epoch 4 -- 86.4% data
Epoch 4/50
5/5 [==============================] - 0s 16ms/step - loss: 1.2309
Epoch 5 -- 100.0% data
Epoch 5/50
5/5 [==============================] - 0s 12ms/step - loss: 1.1888
Epoch 6 -- 100.0% data
Epoch 6/50
5/5 [==============================] - 0s 14ms/step - loss: 1.1963
Epoch 7 -- 100.0% data
Epoch 7/50
5/5 [==============================] - 0s 14ms/step - loss: 1.1822
Epoch 8 -- 100.0% data
Epoch 8/50
5/5 [==============================] - 0s 12ms/step - loss: 1.1891
Epoch 9 -- 100.0% data
Epoch 9/50
5/5 [==============================] - 0s 16ms/step - loss: 1.1951
Epoch 10 -- 100.0% data
Epoch 10/50
5/5 [==============================] - 0s 12ms/step - loss: 1.1666
Epoch 11 -- 100.0% data
Epoch 11/50
5/5 [==============================] - 0s 14ms/step - loss: 1.1687
Epoch 12 -- 100.0% data
Epoch 12/50
5/5 [==============================] - 0s 13ms/step - loss: 1.1474
Epoch 13 -- 100.0% data
Epoch 13/50
5/5 [==============================] - 0s 12ms/step - loss: 1.1286
Epoch 14 -- 100.0% data
Epoch 14/50
5/5 [==============================] - 0s 12ms/step - loss: 1.1800
Epoch 15 -- 100.0% data
Epoch 15/50
5/5 [==============================] - 0s 13ms/step - loss: 1.1516
Epoch 16 -- 100.0% data
Epoch 16/50
5/5 [==============================] - 0s 13ms/step - loss: 1.1314
```

```
Epoch 17 -- 100.0% data
Epoch 17/50
5/5 [==============================] - 0s 12ms/step - loss: 1.1079
Epoch 18 -- 100.0% data
Epoch 18/50
5/5 [==============================] - 0s 14ms/step - loss: 1.1362
Epoch 19 -- 100.0% data
Epoch 19/50
```

There was an issue here that the number of steps per epoch is not changing, even though the batch size is fixed and we increase the data size. The problem is that `fit_generator` calls the `len` method only at the beginning of the training so the `steps_per_epoch` stays whatever is set initially.

1. We could write a custom training loop calling fit `epochs` times on different data sets. Probably the simplest work around.
2. We could increase the `batch_size` as we train to cover more of the training data; this seems an ok alternative as we start from 50% of the data already so worse case we double the batch size. There is further evidence that supports that increasing the batch size is similar (if not better) to learning rate annealing.
3. There is a dirty fix: change the Keras source code.

I will go with option 2.

```
from tensorflow import keras
import warnings, math


class BatchAdaptiveDataGenerator(keras.utils.Sequence):
    def __init__(self, x, start_batch_size, epochs, subsample=0.3, start_data_ratio=0.5, s
        self.x = x
        self.subsample = subsample
        self.verbose = verbose

        if self.subsample:
            sample_idx = np.random.choice(self.x.shape[0], size=int(self.subsample*self.x.
            self.x = self.x[sample_idx]

        # initial training params
        self.epochs = epochs
        self.start_batch_size = start_batch_size
        self.start data ratio = start data ratio
```

```
                _    _              _    _
        self.steps_per_epoch = int(self.start_data_ratio*self.x.shape[0]/self.start_batch_

        # adaptive learning param to increase batch_size after each epoch
        self.alpha = np.exp(np.log(1/self.start_data_ratio)/self.epochs)
        self.shuffle = shuffle

        # per epoch variables
        self.epoch = 0
        self.current_x = None
        self.current_batch_size = self.start_batch_size
        self.on_epoch_end()

    def __len__(self):
        return self.steps_per_epoch

    def __getitem__(self, idx):
        """
        Return a batch for autoencoder training.
        """

        batch = self.current_x[idx*self.current_batch_size: (idx+1)*self.current_batch_siz

        return batch, batch

    def on_epoch_end(self):
        """
        Called before training and after every epoch to include more and more training dat
        """

        # update training data by slicing and shuffling
        current_x_size = int(self.current_batch_size*self.steps_per_epoch)
        self.current_x = self.x[:current_x_size]

        # shuffle rows to mix data in different batches
        if self.shuffle:
            rand_idx = np.arange(self.current_x.shape[0])
            np.random.shuffle(rand_idx)
            self.current_x = self.current_x[rand_idx]

        if self.verbose:
            print(f"Epoch {self.epoch+1} -- {self.current_x.shape[0]/self.x.shape[0]*100}%

        # update batch size
        self.current_batch_size = int(self.start_batch_size * self.alpha**self.epoch)
        self.epoch += 1


class TestBatchAdaptiveDataGenerator(unittest.TestCase):
    def setUp(self):
        self.X_bal = np.random.randn(10000, 10)
        # self.X_bal = X_bal
        self.batch_size = 32
        self.epochs = 50
        self.generator = BatchAdaptiveDataGenerator(self.X_bal, self.batch_size, self.epoc
        self.model = RandAE(input_dim=10, hidden_dims=[16, 8, 16])
```

```
        self.model.compile(optimizer="adam", loss="mse", run_eagerly=True)

    def test_fit(self):
        self.model.fit_generator(generator=self.generator,
                                 epochs=10,
                                 workers=1)

unittest.main(argv=[''], verbosity=2, exit=False)
```

```
test_fit (__main__.TestAdaptiveDataGenerator) ... Epoch 1 -- 50.0% data
Epoch 1/50
5/5 [==============================] - 0s 12ms/step - loss: 1.1673
Epoch 2 -- 60.0% data
Epoch 2/50
1/5 [=====>........................] - ETA: 0s - loss: 1.0847/usr/local/lib/python
  warnings.warn('`Model.fit_generator` is deprecated and '
5/5 [==============================] - 0s 13ms/step - loss: 1.1685
Epoch 3 -- 72.0% data
Epoch 3/50
5/5 [==============================] - 0s 12ms/step - loss: 1.1775
Epoch 4 -- 86.4% data
Epoch 4/50
5/5 [==============================] - 0s 13ms/step - loss: 1.2085
Epoch 5 -- 100.0% data
Epoch 5/50
5/5 [==============================] - 0s 17ms/step - loss: 1.1999
Epoch 6 -- 100.0% data
Epoch 6/50
5/5 [==============================] - 0s 12ms/step - loss: 1.1348
Epoch 7 -- 100.0% data
Epoch 7/50
5/5 [==============================] - 0s 14ms/step - loss: 1.1237
Epoch 8 -- 100.0% data
Epoch 8/50
5/5 [==============================] - 0s 14ms/step - loss: 1.1807
Epoch 9 -- 100.0% data
Epoch 9/50
5/5 [==============================] - 0s 14ms/step - loss: 1.1625
Epoch 10 -- 100.0% data
Epoch 10/50
5/5 [==============================] - 0s 13ms/step - loss: 1.2021
Epoch 11 -- 100.0% data
Epoch 11/50
5/5 [==============================] - 0s 13ms/step - loss: 1.1402
Epoch 12 -- 100.0% data
Epoch 12/50
5/5 [==============================] - 0s 15ms/step - loss: 1.1431
Epoch 13 -- 100.0% data
Epoch 13/50
5/5 [==============================] - 0s 15ms/step - loss: 1.1134
Epoch 14 -- 100.0% data
Epoch 14/50
5/5 [==============================] - 0s 12ms/step - loss: 1.1162
Epoch 15 -- 100.0% data
Epoch 15/50
5/5 [==============================] - 0s 14ms/step - loss: 1.1568
Epoch 16 -- 100.0% data
Epoch 16/50
5/5 [==============================] - 0s 14ms/step - loss: 1.1380
Epoch 17 -- 100.0% data
```

```
Epoch 17/50
5/5 [==============================] - 0s 14ms/step - loss: 1.1194
Epoch 18 -- 100.0% data
Epoch 18/50
5/5 [==============================] - 0s 15ms/step - loss: 1.1348
Epoch 19 -- 100.0% data
Epoch 19/50
```

Training the ensemble

We would like to set the `alpha` parameter that controls the amount of data used so that at the end of training, we use the full sample.

```
# baseline loss
np.square(X_bal - X_bal.mean(axis=0)).mean()
```

```
0.10141651784080594
```

```
from datetime import datetime

MODEL_PARAMS ={"input_dim": X_bal.shape[1],
               "hidden_dims": [16],
               "drop_ratio": 0.33}

COMPILE_PARAMS = {"optimizer": keras.optimizers.Adam(learning_rate=1e-3),
                  "loss": keras.losses.MeanSquaredError(),
                  "run_eagerly": True,}

EPOCHS = 20

DATA_GEN_PARAMS = {"start_batch_size": 128,
                   "epochs": EPOCHS,
                   "subsample": 0.3,}

FIT_PARAMS = {"epochs": EPOCHS,
              "verbose": 1}

n_models = 10
ensemble = []

all_params = {
    "model_params": MODEL_PARAMS,
    "compile_params": COMPILE_PARAMS,
    "epochs": EPOCHS,
    "data_gen_params": DATA_GEN_PARAMS,
    "fit_params": FIT_PARAMS,
    "n_models": N_MODELS,
}

from tqdm import tqdm
import pickle
```

```
# fit models and save results
print("Fitting the models...")
timestamp = datetime.now().strftime("%Y-%m-%d-%H-%M-%S")

# with open(f"models/all_params_{timestamp}.pickle", "wb") as handle:
#     pickle.dump(all_params, handle, protocol=pickle.HIGHEST_PROTOCOL)

for i in tqdm(range(n_models)):
    model = RandAE(**MODEL_PARAMS)
    model.compile(**COMPILE_PARAMS)

    data_generator = BatchAdaptiveDataGenerator(X_bal, **DATA_GEN_PARAMS)
    model.fit_generator(generator=data_generator, **FIT_PARAMS)

    # save the weights and mask
    model.save_weights(f"models/randae_model_{i}")
    with open(f'model_{i}_masks.pickle', 'wb') as handle:
        pickle.dump(model.layer_masks, handle, protocol=pickle.HIGHEST_PROTOCOL)

    with open(f"models/model_{i}_history_{timestamp}.pickle", "wb") as handle:
            pickle.dump(history.history, handle, protocol=pickle.HIGHEST_PROTOCOL)

    ensemble.append(model)
```

```
 9/9 [==============================] - 0s 10ms/step - loss: 0.0753
 Epoch 19/20
 9/9 [==============================] - 0s 10ms/step - loss: 0.0741
 Epoch 20/20
 9/9 [==============================] - 0s 10ms/step - loss: 0.0731
  20%|██        | 2/10 [00:05<00:22,  2.75s/it]Epoch 1/20
 9/9 [==============================] - 0s 10ms/step - loss: 0.1167
 Epoch 2/20
 9/9 [==============================] - 0s 10ms/step - loss: 0.1062
 Epoch 3/20
 9/9 [==============================] - 0s 11ms/step - loss: 0.1002
 Epoch 4/20
 9/9 [==============================] - 0s 10ms/step - loss: 0.0966
 Epoch 5/20
 9/9 [==============================] - 0s 10ms/step - loss: 0.0939
 Epoch 6/20
 9/9 [==============================] - 0s 11ms/step - loss: 0.0917
 Epoch 7/20
 9/9 [==============================] - 0s 11ms/step - loss: 0.0895
 Epoch 8/20
 9/9 [==============================] - 0s 11ms/step - loss: 0.0875
 Epoch 9/20
 9/9 [==============================] - 0s 11ms/step - loss: 0.0852
 Epoch 10/20
 9/9 [==============================] - 0s 10ms/step - loss: 0.0826
 Epoch 11/20
 9/9 [==============================] - 0s 11ms/step - loss: 0.0804
 Epoch 12/20
 9/9 [==============================] - 0s 11ms/step - loss: 0.0781
 Epoch 13/20
 9/9 [==============================] - 0s 10ms/step - loss: 0.0758
 Epoch 14/20
```

```
9/9 [==============================] - 0s 11ms/step - loss: 0.0743
Epoch 15/20
9/9 [==============================] - 0s 10ms/step - loss: 0.0724
Epoch 16/20
9/9 [==============================] - 0s 10ms/step - loss: 0.0706
Epoch 17/20
9/9 [==============================] - 0s 10ms/step - loss: 0.0691
Epoch 18/20
9/9 [==============================] - 0s 11ms/step - loss: 0.0678
Epoch 19/20
9/9 [==============================] - 0s 10ms/step - loss: 0.0664
Epoch 20/20
9/9 [==============================] - 0s 10ms/step - loss: 0.0651
 30%|███       | 3/10 [00:08<00:19,  2.75s/it]Epoch 1/20
9/9 [==============================] - 0s 11ms/step - loss: 0.1206
Epoch 2/20
9/9 [==============================] - 0s 12ms/step - loss: 0.1088
Epoch 3/20
9/9 [==============================] - 0s 11ms/step - loss: 0.1021
Epoch 4/20
9/9 [==============================] - 0s 11ms/step - loss: 0.0978
Epoch 5/20
9/9 [==============================] - 0s 11ms/step - loss: 0.0950
Epoch 6/20
9/9 [==============================] - 0s 11ms/step - loss: 0.0925
Epoch 7/20
```

After running a few experiments, it looks that shallow AE's are learning much better in this setup. I would implement layer-wise pretraining next to address this (as in the original RandNet paper).

Evaluating the ensemble

```
def eval_ensemble(ensemble, input_data, input_labels, contamination):

    results = dict()

    # make prediction by each ensemble component
    predictions = [model.predict(input_data) for model in ensemble]

    # SSE based on reconstruction for each component
    reconstruction_loss = np.stack(
        [np.square(pred - input_data).sum(axis=1) for pred in predictions], axis=1
    ) # could be better ways to aggregate other then sum...

    results["reconstruction_loss"] = reconstruction_loss

    # scale the std to account for different levels of overfitting
    scaler = StandardScaler(with_mean=False)
    reconstruction_loss = scaler.fit_transform(reconstruction_loss)

    # find the median loss for each sample
    median_loss = np.median(reconstruction_loss, axis=1)
```

```
    median_loss = np.median(reconstruction_loss, axis=1)

    # calibrate the threshold by training contamination ratio
    threshold = np.quantile(median_loss, contamination)

    # make hard predicition
    test_outliers = np.where(median_loss > threshold, 1, 0)

    results["classification_report"] = classification_report(
        input_labels, test_outliers
    )

    # min-max scaling the reconstruction loss to calculate PR-curve
    min_ = median_loss.min()
    max_ = median_loss.max()
    scaled_loss = (median_loss - min_) / (max_ - min_)

    results["precision_recall_curve"] = precision_recall_curve(
        input_labels, scaled_loss
    )

    results["cohen_kappa"] = cohen_kappa_score(input_labels, test_outliers)

    return results


# evaluate the ensemble on the test set
print("Evaluating...")
eval_results = eval_ensemble(ensemble, X_test, y_test, contamination=y_train.mean())

with open(f"models/eval_results_{timestamp}.pickle", "wb") as handle:
    pickle.dump(eval_results, handle, protocol=pickle.HIGHEST_PROTOCOL)
```

```
 Evaluating...
```

We ran the experiment using the python scripts and can load the evaluation results back. The two main runs compare preparing the data with quantile scaler plus minmax and minmax scaler only.

```
!pip install pickle5
#import pickle5 as pickle
```

```
 Collecting pickle5
   Downloading pickle5-0.0.11.tar.gz (132 kB)
      |████████████████████████████████| 132 kB 7.0 MB/s
 Building wheels for collected packages: pickle5
   Building wheel for pickle5 (setup.py) ... done
   Created wheel for pickle5: filename=pickle5-0.0.11-cp37-cp37m-linux_x86_64.whl siz
   Stored in directory: /root/.cache/pip/wheels/7e/6a/00/67136a90d6aca437d806d1d3cedf
 Successfully built pickle5
 Installing collected packages: pickle5
 Successfully installed pickle5-0.0.11
```

```
import pickle5 as pickle
```

```
import pickle5 as pickle

# MM only, smaller subsampling with larger model
#last_experiment = "models/eval_results_2020-12-10-09-51-25.pickle"

# Own + MM: my robust scaler with minmax
#own_experiment = "models/eval_results_2020-12-10-08-47-54.pickle"

# MM: minmax scaling of the data only
#minmax_experiment = "models/eval_results_2020-12-09-22-36-15.pickle"

# QS+MM: quantile and minmax scaling -- performed worse in the end.
quantile_minmax_experiment = "models/eval_results_2021-09-24-17-34-57.pickle"

with open(quantile_minmax_experiment, "rb") as handle:
    results = pickle.load(handle)
```

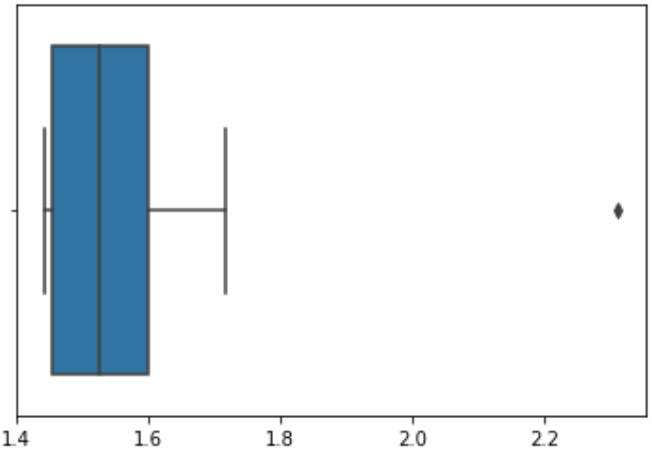
We can look more closely at the reconstruction losses.

```
reconstruction_loss = results["reconstruction_loss"]
```

Looking at the different components, the means and variances of the component losses are quite close.

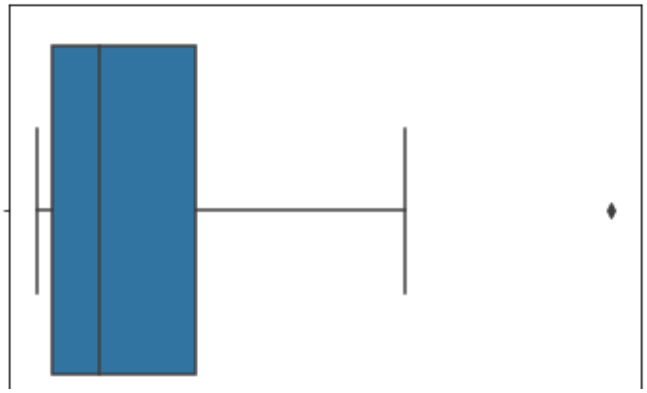
```
sns.boxplot(reconstruction_loss.mean(axis=0))
plt.show()
```

```
/usr/local/lib/python3.7/dist-packages/seaborn/_decorators.py:43: FutureWarning: Pas
  FutureWarning
```




```
sns.boxplot(reconstruction_loss.std(axis=0))
plt.show()
```

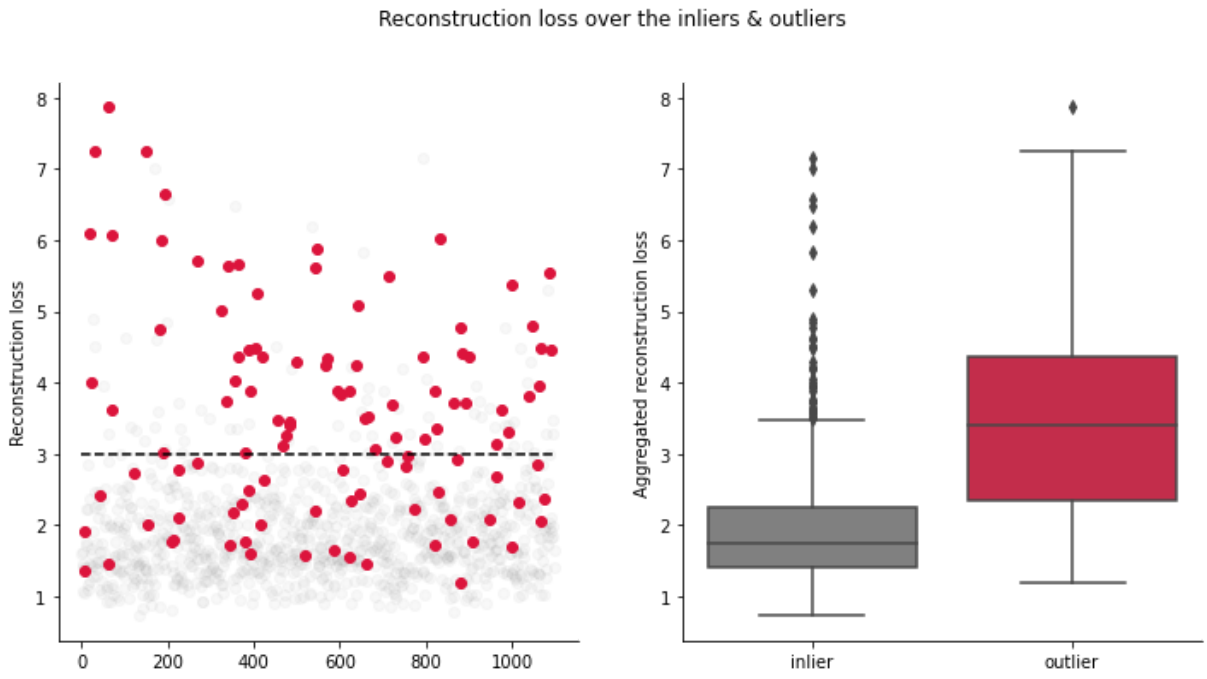
```
/usr/local/lib/python3.7/dist-packages/seaborn/_decorators.py:43: FutureWarning: Pas
  FutureWarning
```



Own: pretty even std and means with one outlier component.

MM: Interesting to see a few outliers here, there are three components (out of 25) from the ensemble that fit quite differently than the others.

```
# scale the reconstruction loss to account for those outliers.
scaler = StandardScaler(with_mean=False)
reconstruction_loss = scaler.fit_transform(reconstruction_loss, )

# aggregate loss across the ensemble components
agg_loss = np.median(reconstruction_loss, axis=1)

threshold = 3

plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.scatter(np.where(y_test==0)[0],
            agg_loss[(y_test == 0).ravel()],
            c="grey",
            alpha=0.05)
plt.scatter(np.where(y_test == 1)[0],
            agg_loss[(y_test == 1).ravel()],
            c="crimson",
            alpha=1)
plt.hlines(threshold, xmin=0, xmax=y_test.shape[0], linestyles="--", colors="black")
plt.ylabel("Reconstruction loss")
sns.despine()

ax = plt.subplot(1, 2, 2)
plot_data = pd.DataFrame(data={"agg_loss": agg_loss, "label": np.where(y_test.ravel() == 0
sns.boxplot(y="agg_loss", x="label", data=plot_data, palette=["grey", "crimson"], ax=ax)
ax.set_xlabel("")
ax.set_ylabel("Aggregated reconstruction loss")
sns.despine()

plt.suptitle("Reconstruction loss over the inliers & outliers")
plt.show()
```



MM scaling only: better results than QS+MM scaling, inlier loss is much more concentrated with.

QS+MM scaling: The outliers have consistently high reconstruction loss but the main issue seems to be that there are a fair amount of inlier data points with high median reconstruction loss. It would be interesting to see why this happens and to look at these data points specifically. I wonder if these overlap with those inlier clusters we initially observed. Threshold around 0.5 looks optimal.

```
bigloss_idx = np.where(agg_loss > threshold)[0]
smallloss_idx = np.random.choice(np.where(agg_loss <= threshold)[0], size=(2500,))
idx = np.concatenate([bigloss_idx, smallloss_idx], axis=0)
plot_X = X_test[idx]
plot_y = y_test[idx].ravel()
plot_loss = agg_loss[idx]
```

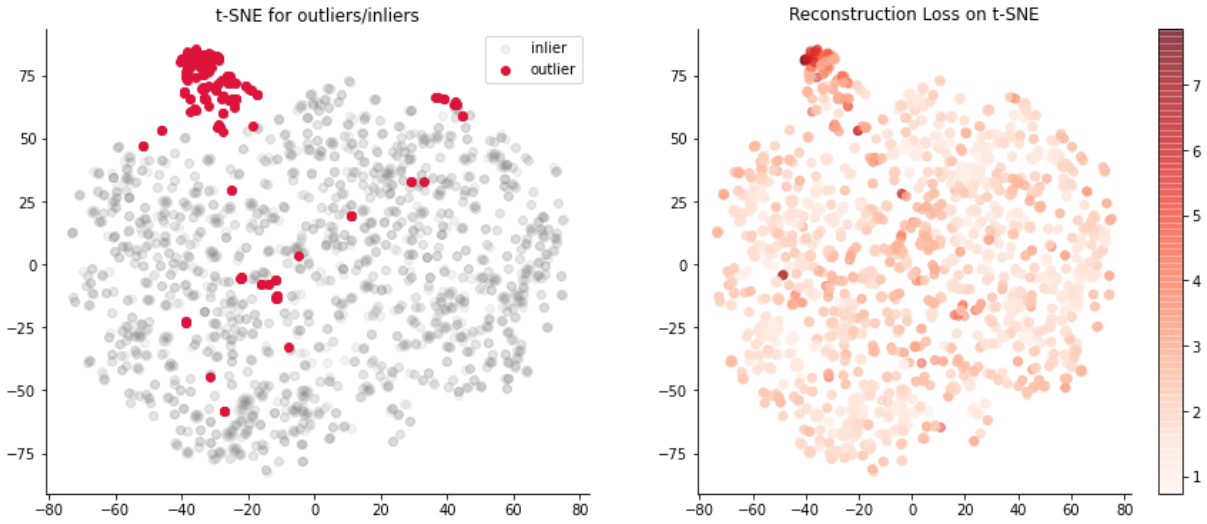
```
plot_X.shape
```

```
(2637, 30)
```

```
tsne = TSNE(n_components=2)
tsne_data = tsne.fit_transform(plot_X)
```

```
plt.figure(figsize=(15, 6))
plt.subplot(1, 2, 1)
plt.scatter(tsne_data[plot_y == 0, 0],
            tsne_data[plot_y == 0, 1], c="grey", alpha=0.1, label="inlier")
plt.scatter(tsne_data[plot_y == 1, 0],
            tsne_data[plot_y == 1, 1], c="crimson", alpha=1, label="outlier")
plt.legend()
sns.despine()
```

```
sns.despine()
plt.title("t-SNE for outliers/inliers")

plt.subplot(1, 2, 2)
plt.scatter(tsne_data[:, 0],
            tsne_data[:, 1],
            c=plot_loss, cmap="Reds", alpha=0.5)
plt.colorbar()
plt.title("Reconstruction Loss on t-SNE")
sns.despine()
#plt.savefig("assets/tsne_reconstruction_loss.png")
plt.show()
```



Own: this gives the worse reconstruction losses for inliers... clearly not the right approach :)

MM: mixed result here. This picture supports the previous observation that the reconstruction loss is much more concentrated for inliers (although there are a fair number of badly learned inlier points). t-SNE very nicely separates the outliers but surprisingly there are a few points there with small reconstruction loss.

QS+MM: What I see is that autoencoders are nicely fitted on the 'central' data regioun but struggle to reproduce plenty of the inliers around the outer regions. This relates to the observations we made early on about how some of the inliers formed similar clusters at the periphery.

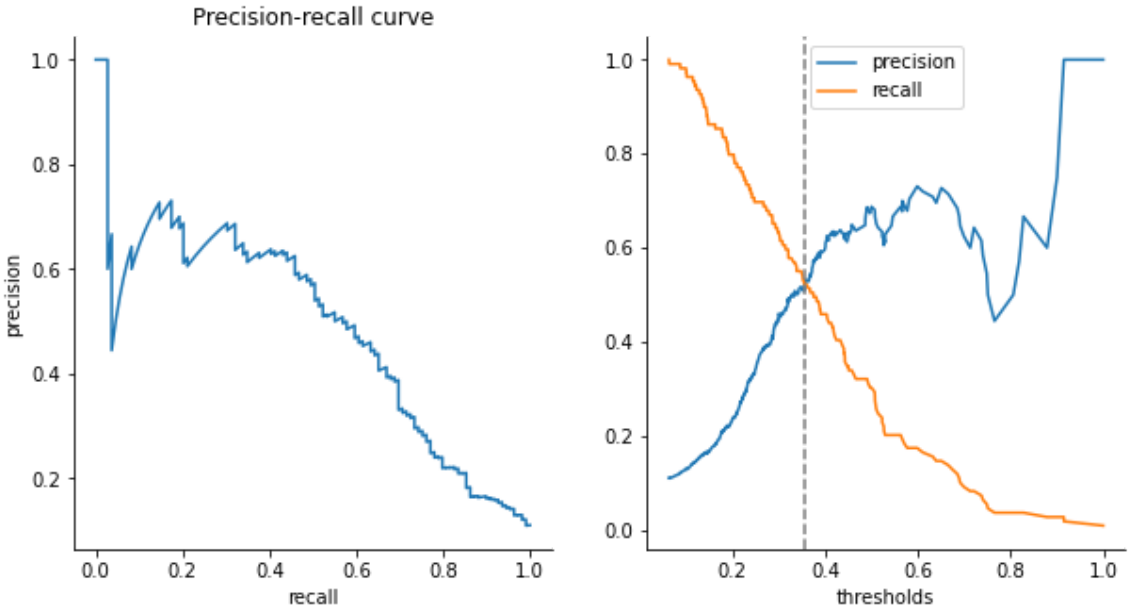
As for model accuracy with respect the true labels:

```
prec, recall, thresholds = results["precision_recall_curve"]

best_idx = np.argmin(np.abs(prec-recall))
best_prec, best_recall = prec[best_idx], recall[best_idx]
```

```
print(f"Best precision {np.round(best_prec, 2)}, "\
      f"recall: {np.round(best_recall, 2)} at {np.round(thresholds[best_idx], 2)} thresho]
```

```
Best precision 0.52, recall: 0.52 at 0.36 threshold.
```

```
fig, axs = plt.subplots(1, 2, figsize=(10, 5))
axs[0].plot(recall, prec)
axs[0].set_xlabel("recall")
axs[0].set_ylabel("precision")
axs[0].set_title("Precision-recall curve")
axs[1].plot(thresholds, prec[:-1], label="precision")
axs[1].plot(thresholds, recall[:-1], label="recall")
axs[1].axvline(thresholds[best_idx], 0, 1, c="grey", linestyle="--")
axs[1].set_xlabel("thresholds")
sns.despine()
plt.legend()
plt.show()
```



Own: looks worse than MM...

MM: looks better than QS+MM, we can get up to 47% precision/recall.

QS+MM: The precision-recall curves show that the initial results are not stellar, we get the best score with a threshold around 0.5 where both the precision and recall are around 40%, this is somewhat under the elliptic envelope scores that were around 68% recall and 66% precision.

My best bet, given how much better the elliptic envelope is on the unscaled data and how bad both the elliptic envelope and autoencoders are on the scaled, is that my scaling is not appropriate.