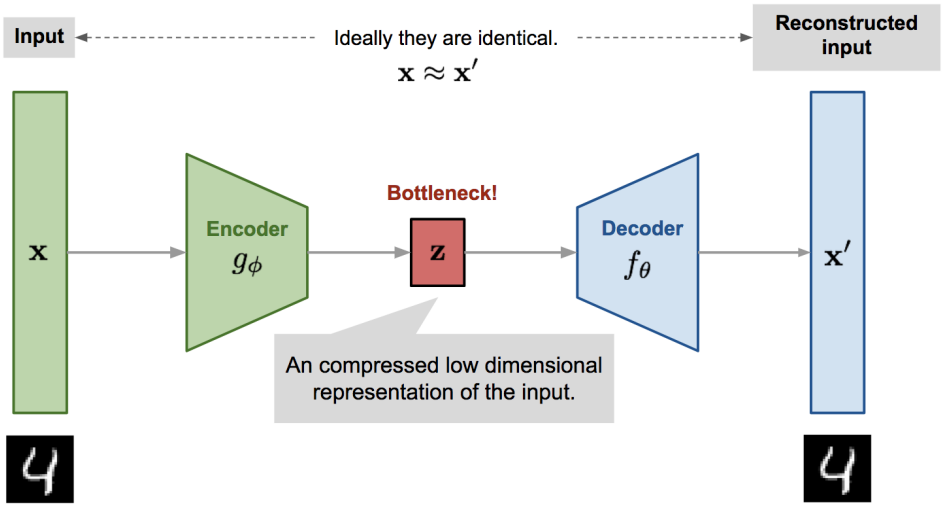
Table of Contents

Unsupervised Learning with Auto-Encoders

If you are interested in an introduction to auto-encoders, head over to Julien Despois' article (https://hackernoon.com/latent-space-visualization-deep-learning-bits-2-bd09a46920df). If a more technical breakdown is what you are looking for, check out Lilian Weng's blog post (https://lilianweng.github.io/lil-log/2018/08/12/from-autoencoder-to-beta-vae.html) from which the below image is sourced. It illustrates the functioning of an auto-encoder for MNIST images, but the concept is the same.



The idea is quite straightforward:

1. Due to the **bottleneck architecture** of the neural network, it is forced to learn a **condensed representation** from which to reproduce the original input.
2. We feed it **only normal transactions**, which it will learn to reproduce with high fidelity.
3. As a consequence, if a **fraud transaction is sufficiently distinct** from normal transactions, the auto-encoder will have trouble reproducing it with its learned

weights, and the subsequent **reconstruction loss will be high**.
4. Anything above a specific loss (treshold) will be **flagged as anomalous** and thus labeled as fraud.

Preprocessing

Import Libraries & set Random Seeds

In [1]:
```
# read & manipulate data
import pandas as pd
import numpy as np
import tensorflow as tf

# visualisations
import matplotlib.pyplot as plt
import seaborn as sns
sns.set(style='whitegrid', context='notebook')
%matplotlib notebook

# misc
import random as rn

# load the dataset
df = pd.read_csv('creditcard.csv')

# manual parameters
RANDOM_SEED = 42
TRAINING_SAMPLE = 200000
VALIDATE_SIZE = 0.2

# setting random seeds for libraries to ensure reproducibility
np.random.seed(RANDOM_SEED)
rn.seed(RANDOM_SEED)
tf.set_random_seed(RANDOM_SEED)
```

```
2022-09-19 10:13:36.801026: W tensorflow/stream_executor/platform
/default/dso_loader.cc:64] Could not load dynamic library 'libcud
art.so.11.0'; dlerror: libcudart.so.11.0: cannot open shared obje
ct file: No such file or directory
2022-09-19 10:13:36.801046: I tensorflow/stream_executor/cuda/cud
art_stub.cc:29] Ignore above cudart dlerror if you do not have a
GPU set up on your machine.

---------------------------------------------------------------
----------
AttributeError                            Traceback (most recent
call last)
/tmp/ipykernel_5459/1038738862.py in <module>
     24 np.random.seed(RANDOM_SEED)
     25 rn.seed(RANDOM_SEED)
---> 26 tf.set_random_seed(RANDOM_SEED)

AttributeError: module 'tensorflow' has no attribute 'set_random_
seed'
```

Renaming columns

In [2]:
```
# let's quickly convert the columns to lower case and rename the Cl
# so as to not cause syntax errors
df.columns = map(str.lower, df.columns)
df.rename(columns={'class': 'label'}, inplace=True)

# print first 5 rows to get an initial impression of the data we're
df.head()
```

Out[2]:

	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	0.0986
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.0851
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.2476
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.3774
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.2705

5 rows × 31 columns

Calculated field: log10(amount)

Turn the amount feature into a normally distributed log equivalent.

In [3]:
```
# add a negligible amount to avoid taking the log of 0
df['log10_amount'] = np.log10(df.amount + 0.00001)
```

In [4]:
```
# keep the label field at the back
df = df[
    [col for col in df if col not in ['label', 'log10_amount']] +
    ['log10_amount', 'label']
]
```

Visualising clusters with t-SNE

t-Distributed Stochastic Neighbor Embedding (t-SNE)

From the sklearn documentation (https://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html):

> t-SNE [1] is a tool to visualize high-dimensional data. It converts similarities between data points to joint probabilities and tries to minimize the Kullback-Leibler divergence between the joint probabilities of the low-dimensional embedding and the high-dimensional data. t-SNE has a cost function that is not convex, i.e. with different initializations we can get different results.

In plain English, most certainly oversimplifying matters: **t-SNE is a dimensionality**

reduction technique used for visualisations of complex datasets. It **maps clusters in high-dimensional data** to **a two- or three dimensional plane** so we can get an idea of how easy it will be to **discriminate between classes**. It does this by trying to keep the distance between data points in lower dimensions proportional to the probability that these data points are neighbours in the higher dimensions.

A more elaborate introduction (https://www.datacamp.com/community/tutorials/introduction-t-sne) is available on DataCamp.

Undersampling the non-fraud

To keep the computation time low, let's feed t-SNE only a small subsample (undersampling the clean transactions).

In [5]:
```
# manual parameter
RATIO_TO_FRAUD = 15

# dropping redundant columns
df = df.drop(['time', 'amount'], axis=1)

# splitting by class
fraud = df[df.label == 1]
clean = df[df.label == 0]

# undersample clean transactions
clean_undersampled = clean.sample(
    int(len(fraud) * RATIO_TO_FRAUD),
    random_state=RANDOM_SEED
)

# concatenate with fraud transactions into a single dataframe
visualisation_initial = pd.concat([fraud, clean_undersampled])
column_names = list(visualisation_initial.drop('label', axis=1).col

# isolate features from labels
features, labels = visualisation_initial.drop('label', axis=1).valu
                   visualisation_initial.label.values
```

In [6]:
```
print(f"""The non-fraud dataset has been undersampled from {len(cle
This represents a ratio of {RATIO_TO_FRAUD}:1 to fraud.""")
```

```
The non-fraud dataset has been undersampled from 284,315 to 7,38
0.
This represents a ratio of 15:1 to fraud.
```

t-SNE output

In [7]:
```
from sklearn.manifold import TSNE
from mpl_toolkits.mplot3d import Axes3D

def tsne_scatter(features, labels, dimensions=2, save_as='graph.png
    if dimensions not in (2, 3):
        raise ValueError('tsne_scatter can only plot in 2d or 3d (W

    # t-SNE dimensionality reduction
    features_embedded = TSNE(n_components=dimensions, random_state=
```

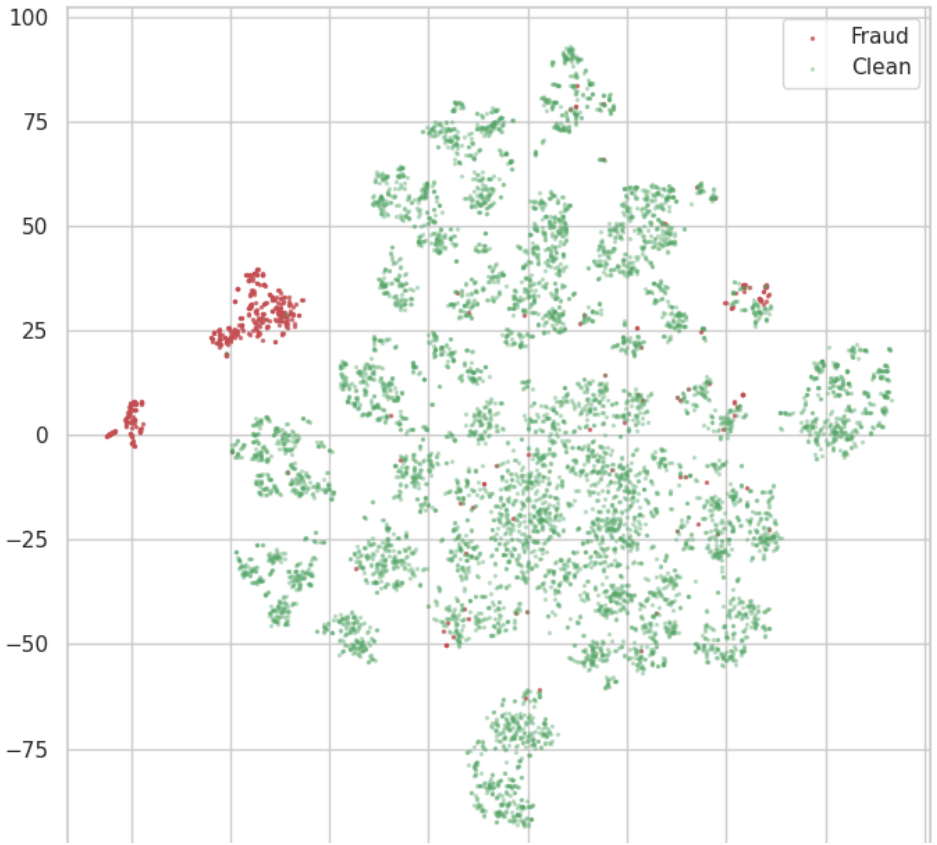
```
    # initialising the plot
    fig, ax = plt.subplots(figsize=(8,8))

    # counting dimensions
    if dimensions == 3: ax = fig.add_subplot(111, projection='3d')

    # plotting data
    ax.scatter(
        *zip(*features_embedded[np.where(labels==1)]),
        marker='o',
        color='r',
        s=2,
        alpha=0.7,
        label='Fraud'
    )
    ax.scatter(
        *zip(*features_embedded[np.where(labels==0)]),
        marker='o',
        color='g',
        s=2,
        alpha=0.3,
        label='Clean'
    )

    # storing it to be displayed later
    plt.legend(loc='best')
    plt.savefig(save_as);
    plt.show;
```

In [8]:
```
tsne_scatter(features, labels, dimensions=2, save_as='tsne_initial
```

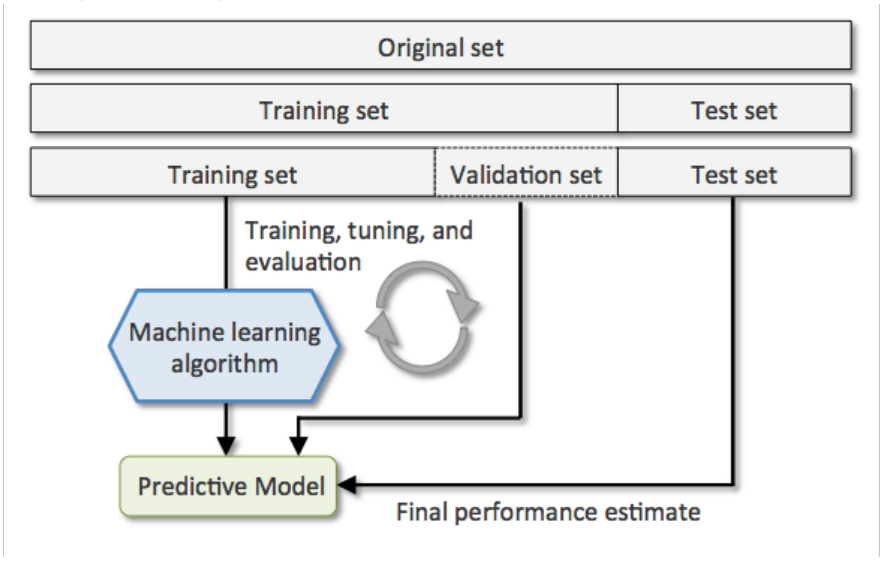


Some clusters are apparent, but a minority of fraud transactions remains sneaky, sneaky.

Train/Validate/Test split

Our auto-encoder will **only train on transactions that were normal**. What's left over will be combined with the fraud set to form our test sample.

We will be doing something akin to the below:



1. Training: only non-fraud
   - Split into:
     A. Actual training of our autoencoder
     B. Validation of the neural network's ability to generalize
2. Testing : mix of fraud and non-fraud
   - Treated like new data
   - Attempt to locate outliers
     A. Compute reconstruction loss
     B. Apply threshold

In [9]:
```
print(f"""Shape of the datasets:
    clean (rows, cols) = {clean.shape}
    fraud (rows, cols) = {fraud.shape}""")
```

```
Shape of the datasets:
    clean (rows, cols) = (284315, 30)
    fraud (rows, cols) = (492, 30)
```

In [10]:
```
# shuffle our training set
clean = clean.sample(frac=1).reset_index(drop=True)

# training set: exlusively non-fraud transactions
X_train = clean.iloc[:TRAINING_SAMPLE].drop('label', axis=1)
```

```
# testing  set: the remaining non-fraud + all the fraud
X_test = clean.iloc[TRAINING_SAMPLE:].append(fraud).sample(frac=1)
```

In [11]:
```
print(f"""Our testing set is composed as follows:

{X_test.label.value_counts()}""")
```

```
Our testing set is composed as follows:

0    84315
1      492
Name: label, dtype: int64
```

In [14]:
```
creditcard.csv from sklearn.model_selection import train_test_split

# train // validate - no labels since they're all clean anyway
X_train, X_validate = train_test_split(X_train,
                                       test_size=VALIDATE_SIZE,
                                       random_state=RANDOM_SEED)

# manually splitting the labels from the test df
X_test, y_test = X_test.drop('label', axis=1).values, X_test.label.
```

```
  File "/tmp/ipykernel_5459/2916041613.py", line 1
    creditcard.csv from sklearn.model_selection import train_test
_split
                   ^
SyntaxError: invalid syntax
```

Summary

In [15]:
```
print(f"""Shape of the datasets:
    training (rows, cols) = {X_train.shape}
    validate (rows, cols) = {X_validate.shape}
    holdout  (rows, cols) = {X_test.shape}""")
```

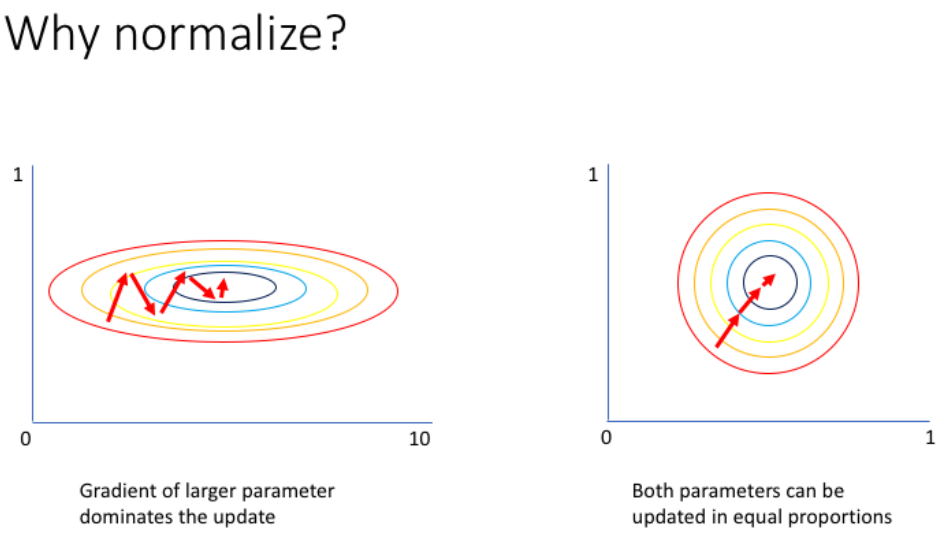
```
---------------------------------------------------------------------
----------
NameError                                 Traceback (most recent
call last)
/tmp/ipykernel_5459/2934506048.py in <module>
      1 print(f"""Shape of the datasets:
      2     training (rows, cols) = {X_train.shape}
----> 3     validate (rows, cols) = {X_validate.shape}
      4     holdout  (rows, cols) = {X_test.shape}""")

NameError: name 'X_validate' is not defined
```

Normalising & Standardising

Why

In an excellent article by Jeremy Jordan (https://www.jeremyjordan.me/batch-normalization/), it is explained why making sure your data is normally distributed can **help stochastic gradient descent converge** more effectively. In a nutshell:



When

At what point in the data processing do we apply standardisation/normalisation? An excellent answer was provided on StackOverflow (https://stackoverflow.com/questions/49444262/normalize-data-before-or-after-split-of-training-and-testing-data).

> Don't forget that **testing data points represent real-world data**. Feature normalization (or data standardization) of the explanatory (or predictor) variables is a technique used to center and normalise the data by subtracting the mean and dividing by the variance. **If you take the mean and variance of the whole dataset you'll be introducing future information into the training explanatory variables** (i.e. the mean and variance).
>
> Therefore, you should **perform feature normalisation over the training data**. Then **perform normalisation on testing *_instances as well, but this time_* using the mean and variance of training** explanatory variables. In this way, we can test and evaluate whether our model can generalize well to new, unseen data points.
>
> [Answer by Giorgos Myrianthous (https://stackoverflow.com/users/7131757/giorgos-myrianthous)]

Building our pipeline

In [16]:
```
from sklearn.preprocessing import Normalizer, MinMaxScaler
from sklearn.pipeline import Pipeline

# configure our pipeline
pipeline = Pipeline([('normalizer', Normalizer()),
```

```
                    ('scaler', MinMaxScaler())])
```

Fitting the pipeline

In [17]:
```
# get normalization parameters by fitting to the training data
pipeline.fit(X_train);
```

Applying transformations with acquired parameters

In [18]:
```
creditcard.csv# transform the training and validation data with the
X_train_transformed = pipeline.transform(X_train)
X_validate_transformed = pipeline.transform(X_validate)
```

```
---------------------------------------------------------------------
----------
NameError                                 Traceback (most recent
call last)
/tmp/ipykernel_5459/810551744.py in <module>
----> 1 creditcard.csv# transform the training and validation dat
a with these parameters
      2 X_train_transformed = pipeline.transform(X_train)
      3 X_validate_transformed = pipeline.transform(X_validate)

NameError: name 'creditcard' is not defined
```

Before & After

In [ ]:
```
g = sns.PairGrid(X_train.iloc[:,:3].sample(600, random_state=RANDOM
plt.subplots_adjust(top=0.9)
g.fig.suptitle('Before:')
g.map_diag(sns.kdeplot)
g.map_offdiag(sns.kdeplot);
```

In [ ]:
```
g = sns.PairGrid(pd.DataFrame(X_train_transformed, columns=column_r
plt.subplots_adjust(top=0.9)
g.fig.suptitle('After:')
g.map_diag(sns.kdeplot)
g.map_offdiag(sns.kdeplot);
```

We can tell the data is slightly more **uniform and proportionally distributed**.
The ranges were also shrunk to fit **between 0 and 1**.

Training the auto-encoder

TensorBoard

As documented in [this kernel by Aurelio Agundez (https://www.kaggle.com/aagundez/using-tensorboard-in-kaggle-kernels)](), TensorBoard requires a running kernel, so its

output will only be available in an editor session. Fork this notebook if you wish to interact with it.

In [ ]:
```
# Load the extension and start TensorBoard
%load_ext tensorboard.notebook
%tensorboard --logdir logs
```

Architecture of our model

Keras has become the standard high-level API within Tensorflow. No surprise, it's awesome. Check out their blog post on the topic of autoencoders (https://blog.keras.io/building-autoencoders-in-keras.html).

In [ ]:
```
# data dimensions // hyperparameters
input_dim = X_train_transformed.shape[1]
BATCH_SIZE = 256
EPOCHS = 100

# https://keras.io/layers/core/
autoencoder = tf.keras.models.Sequential([

    # deconstruct / encode
    tf.keras.layers.Dense(input_dim, activation='elu', input_shape=
    tf.keras.layers.Dense(16, activation='elu'),
    tf.keras.layers.Dense(8, activation='elu'),
    tf.keras.layers.Dense(4, activation='elu'),
    tf.keras.layers.Dense(2, activation='elu'),

    # reconstruction / decode
    tf.keras.layers.Dense(4, activation='elu'),
    tf.keras.layers.Dense(8, activation='elu'),
    tf.keras.layers.Dense(16, activation='elu'),
    tf.keras.layers.Dense(input_dim, activation='elu')

])

# https://keras.io/api/models/model_training_apis/
autoencoder.compile(optimizer="adam",
                    loss="mse",
                    metrics=["acc"])

# print an overview of our model
autoencoder.summary();
```

Callbacks

- Continue as long as the model is reducing the training loss.
- Save only the weights for the model with the lowest validation loss, though.
- Get graphical insights with Tensorboard.

In [ ]:
```
from datetime import datetime

# current date and time
yyyymmddHHMM = datetime.now().strftime('%Y%m%d%H%M')
```

```
# new folder for a new run
log_subdir = f'{yyyymmddHHMM}_batch{BATCH_SIZE}_layers{len(autoencc

# define our early stopping
early_stop = tf.keras.callbacks.EarlyStopping(
    monitor='val_loss',
    min_delta=0.0001,
    patience=10,
    verbose=1,
    mode='min',
    restore_best_weights=True
)

save_model = tf.keras.callbacks.ModelCheckpoint(
    filepath='autoencoder_best_weights.hdf5',
    save_best_only=True,
    monitor='val_loss',
    verbose=0,
    mode='min'
)

tensorboard = tf.keras.callbacks.TensorBoard(
    f'logs/{log_subdir}',
    batch_size=BATCH_SIZE,
    update_freq='batch'
)

# callbacks argument only takes a list
cb = [early_stop, save_model, tensorboard]
```

Training

In [ ]:
```
history = autoencoder.fit(
    X_train_transformed, X_train_transformed,
    shuffle=True,
    epochs=EPOCHS,
    batch_size=BATCH_SIZE,
    callbacks=cb,
    validation_data=(X_validate_transformed, X_validate_transformec
);
```

Reconstructions

We **apply the transformation pipeline to our test set**.
Then, we **pass the data through the trained autoencoder**.

In [ ]:
```
# transform the test set with the pipeline fitted to the training s
X_test_transformed = pipeline.transform(X_test)

# pass the transformed test set through the autoencoder to get the
reconstructions = autoencoder.predict(X_test_transformed)
```

Calculate the reconstruction loss for every transaction and draw a sample.

In [ ]:
```
# calculating the mean squared error reconstruction loss per row in
```

```
mse = np.mean(np.power(X_test_transformed - reconstructions, 2), ax
```

In [ ]:
```
clean = mse[y_test==0]
fraud = mse[y_test==1]

fig, ax = plt.subplots(figsize=(6,6))

ax.hist(clean, bins=50, density=True, label="clean", alpha=.6, colo
ax.hist(fraud, bins=50, density=True, label="fraud", alpha=.6, colo

plt.title("(Normalized) Distribution of the Reconstruction Loss")
plt.legend()
plt.show()
```

Very promising! Although some transactions seem to fool the autoencoder, the fraudulent transactions clearly have a distinguishing element in their data that sets them apart from clean ones.

Setting a threshold for classification

Unsupervised

Normally, in an unsupervised solution, this is where the story would end. We would **set a threshold that limits the amount of false positives** to a manageable degree, **and captures the most anomalous data points**.

Percentiles

We could set this threshold by taking the top x% of the dataset and considering it anomalous.

MAD

We could also use a **modified Z-score using the Median Absolute Deviation to define outliers** on our reconstruction data. Here is a [good blog post on the topic (https://medium.com/james-blogs/outliers-make-us-go-mad-univariate-outlier-detection-b3a72f1ea8c7)](https://medium.com/james-blogs/outliers-make-us-go-mad-univariate-outlier-detection-b3a72f1ea8c7) by João Rodrigues, illustrating why this algorithm is more robust and scalable than the percentiles method.

In [ ]:
```
THRESHOLD = 3

def mad_score(points):
    """https://www.itl.nist.gov/div898/handbook/eda/section3/eda35h
    m = np.median(points)
    ad = np.abs(points - m)
    mad = np.median(ad)

    return 0.6745 * ad / mad

z_scores = mad_score(mse)
outliers = z_scores > THRESHOLD
```

In [ ]:
```
print(f"Detected {np.sum(outliers):,} outliers in a total of {np.si
```

Supervised

We know the labels, so we can verify our results.

Classification Matrix on MAD outliers

A closer look:

In [ ]:
```
from sklearn.metrics import (confusion_matrix,
                             precision_recall_curve)

# get (mis)classification
cm = confusion_matrix(y_test, outliers)

# true/false positives/negatives
(tn, fp,
 fn, tp) = cm.flatten()
```

In [ ]:
```
print(f"""The classifications using the MAD method with threshold={
{cm}

% of transactions labeled as fraud that were correct (precision): {
% of fraudulent transactions were caught succesfully (recall):    {
```

Asymmetric error cost

In the real world, we can expect **different costs associated with reporting a false positive versus reporting a false negative**. Missing a fraud case is likely to be much more costly than wrongly flagging a transaction as one. In another kernel (https://www.kaggle.com/robinteuwens/fraud-detection-as-a-cost-optimization-problem/comments), I discuss an approach to determining these costs for this dataset in depth.

Recall & Precision

Generally speaking, you will have to prioritise what you find more important. This dilemma is commonly called the **"recall vs precision" trade-off**. If you want to increase recall, **adjust the MAD's Z-Score threshold** downwards, if you want recover precision, increase it.

In [ ]:
```
clean = z_scores[y_test==0]
fraud = z_scores[y_test==1]

fig, ax = plt.subplots(figsize=(6,6))

ax.hist(clean, bins=50, density=True, label="clean", alpha=.6, colo
ax.hist(fraud, bins=50, density=True, label="fraud", alpha=.6, colo

plt.title("Distribution of the modified z-scores")
plt.legend()
plt.show()
```

Latent Space

It is always interesting to look at the **compressed representation** our neural network devised.

Encoder

Let's build the encoder that gets us to the bottleneck. We take the layers from our autoencoder.

In [ ]:
```
encoder = tf.keras.models.Sequential(autoencoder.layers[:5])
encoder.summary()
```

Undersampling

Consistent with the previous t-sne visualisation, let's undersample the clean transactions.

In [ ]:
```
# taking all the fraud, undersampling clean
fraud = X_test_transformed[y_test==1]
clean = X_test_transformed[y_test==0][:len(fraud) * RATIO_TO_FRAUD,

# combining arrays & building labels
features = np.append(fraud, clean, axis=0)
labels = np.append(np.ones(len(fraud)),
                   np.zeros(len(clean)))

# getting latent space representation
latent_representation = encoder.predict(features)

print(f'Clean transactions downsampled from {len(X_test_transformed
print('Shape of latent representation:', latent_representation.shap
```

Visualising the Latent Space

In [ ]:
```
X = latent_representation[:,0]
y = latent_representation[:,1]

# plotting
plt.subplots(figsize=(8, 8))
plt.scatter(X[labels==0], y[labels==0], s=1, c='g', alpha=0.3, labe
plt.scatter(X[labels==1], y[labels==1], s=2, c='r', alpha=0.7, labe

# labeling
plt.legend(loc='best')
plt.title('Latent Space Representation')

# saving & displaying
plt.savefig('latent_representation_2d');
plt.show()
```

Although there is no perfectly distinct cluster, **most of the fradulent transactions**

appear to be neatly grouped together. This is in line with the hope/idea that both **classes would occupy distinct areas in latent space**, due to the **encoder's weights not being calibrated to cope with fraudulent transactions**.

Conclusion

We could already tell from our misclassifications that the network was not able to generalize perfectly. However, we must not forget that **our model was trained never having seen a single fraud case!** In that regard, its performance is decent. It illustrates the power of **autoencoders as anomaly detection tools**.

To improve its performance, perhaps we need to:

- improve the model architecture
- diversify the training data more, with a broader sample of clean transactions
- augment the data with different, additional features - the data itself might not be good enough to distinguish between classes perfectly (i.e. fraudsters are disguising themselves well enough to always go undetected using these data points, no matter the algorithm).

In [ ]:

In [ ]:

In [ ]: