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

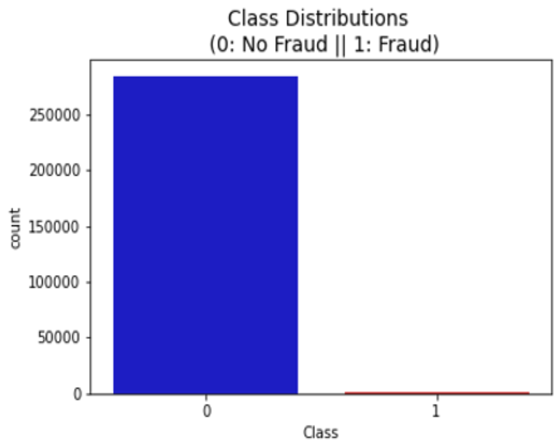
Outliers are data points that differ significantly from the remaining data. The basic approach in neural networks is to use a multi-layer symmetric neural network to reconstruct (i.e, replicate) the data. The reconstruction error is used as the outlier score. There are several problems with this approach. 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.

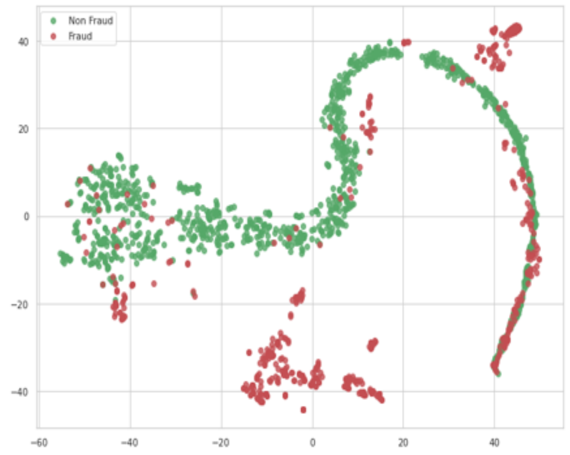
3 DataSets

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:

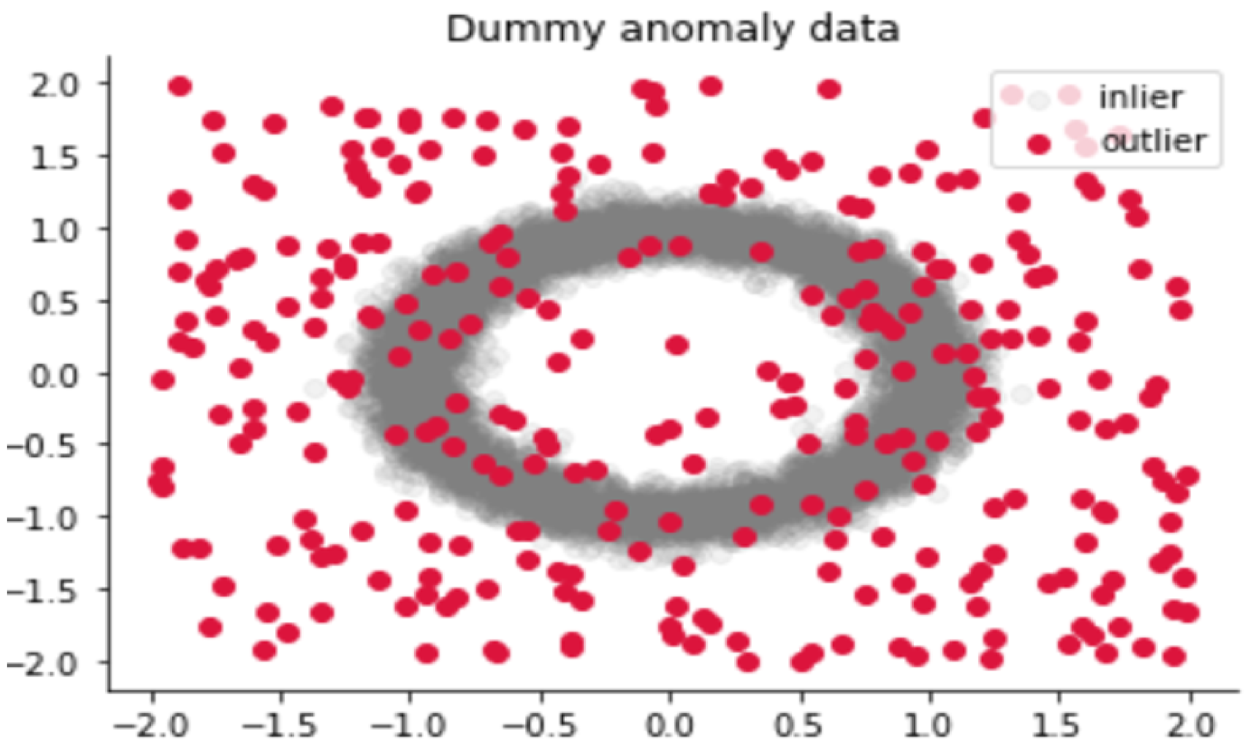


Figure 2: Dummy Data T-SNE

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.

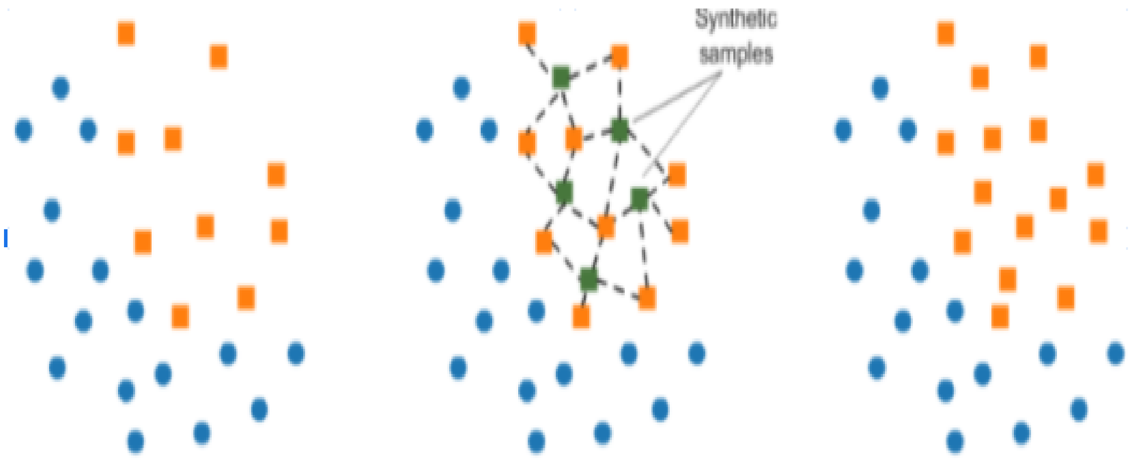


Figure 3: SMOTE technique

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

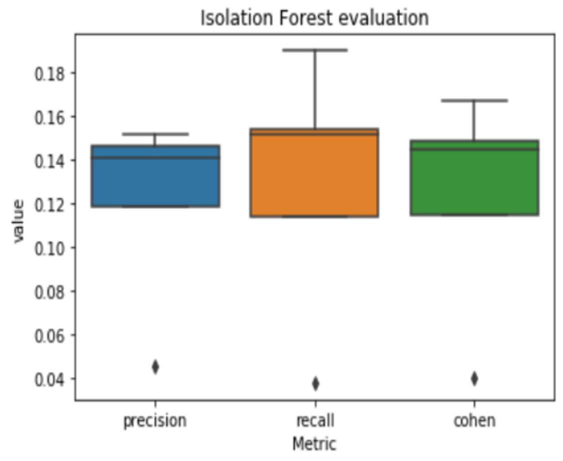
This way new data points are generated in SMOTE technique.

5 The Baseline model

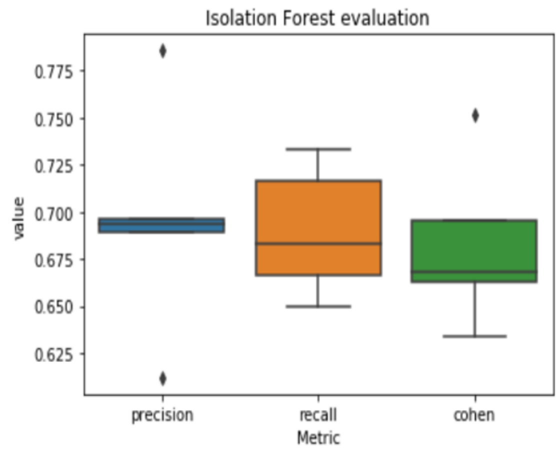
5.a Isolation Forest

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



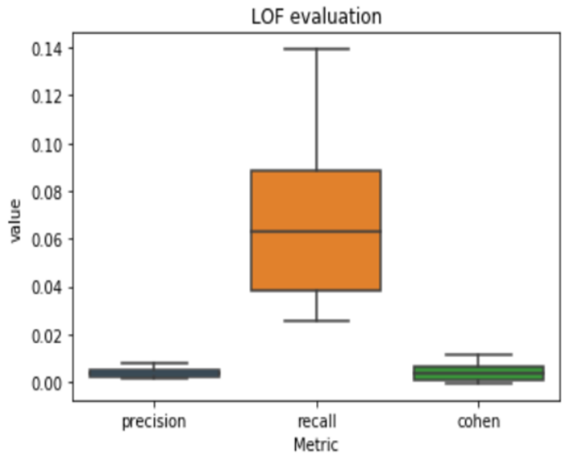
(b) Sanity Check

Figure 4: Isolation Forest

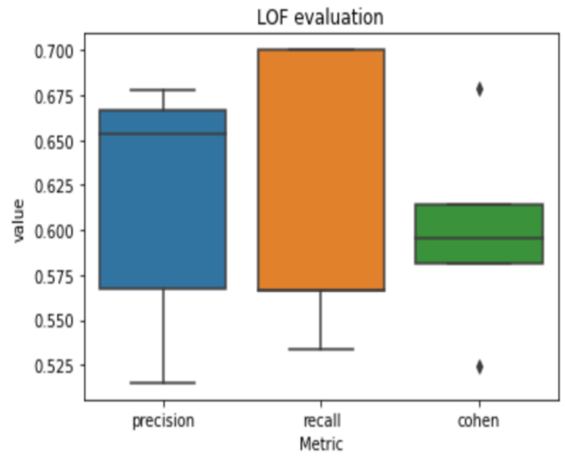
5.b Local Outlier Factor

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

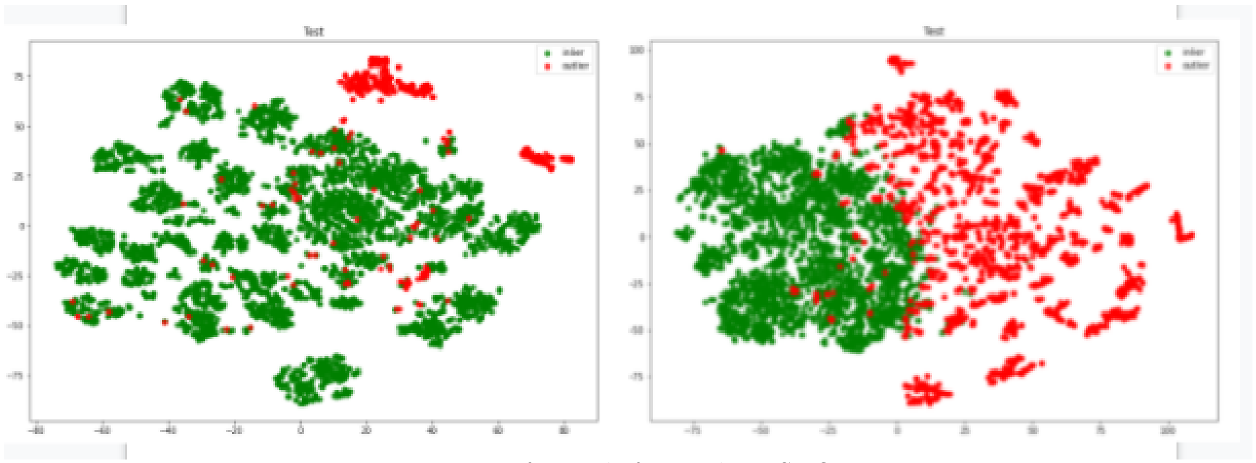
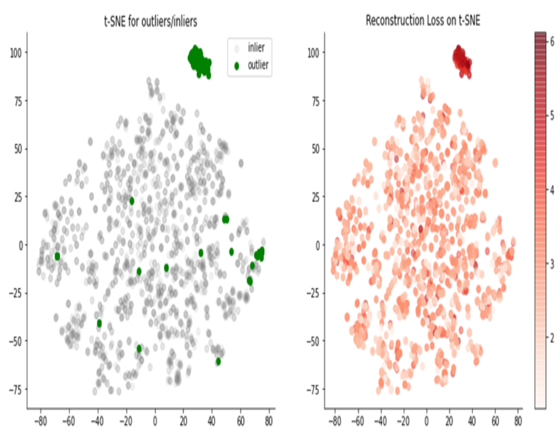


Figure 6: Before and after applying SMOTE

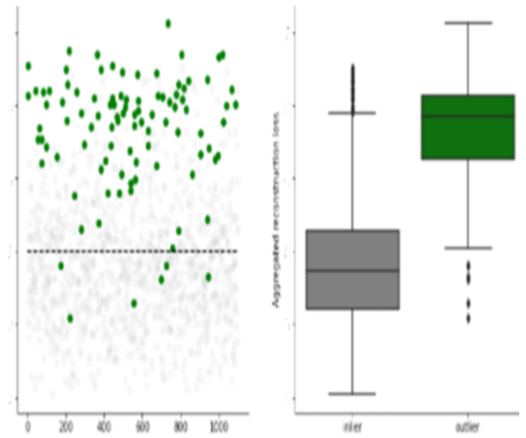
Figure 6 demonstrates the results from running SMOTE against the minority class. This is a very recognisable characteristic of SMOTE-based up-sampling

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.

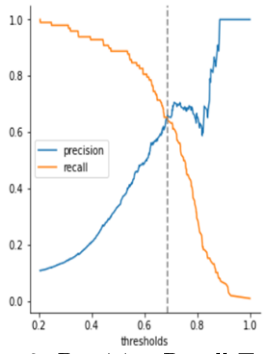


Figure 8: Precision-Recall Tradeoff

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.

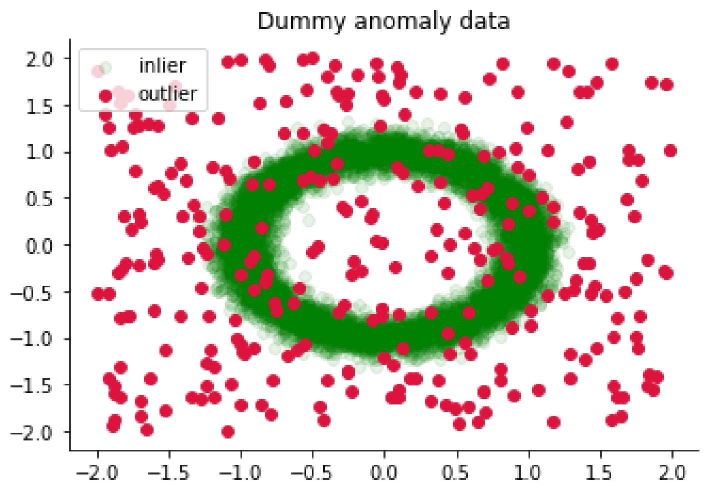
9 Notebook - code

```
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
```

Dummy data generation

```
def create_dummy_data(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 = create_dummy_data(normalize=True)


plt.figure()
plt.scatter(dummy_data.loc[dummy_data["Class"] == 0, 0],
            dummy_data.loc[dummy_data["Class"] == 0, 1], c="green", alpha=0.1, label="inli
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()
```



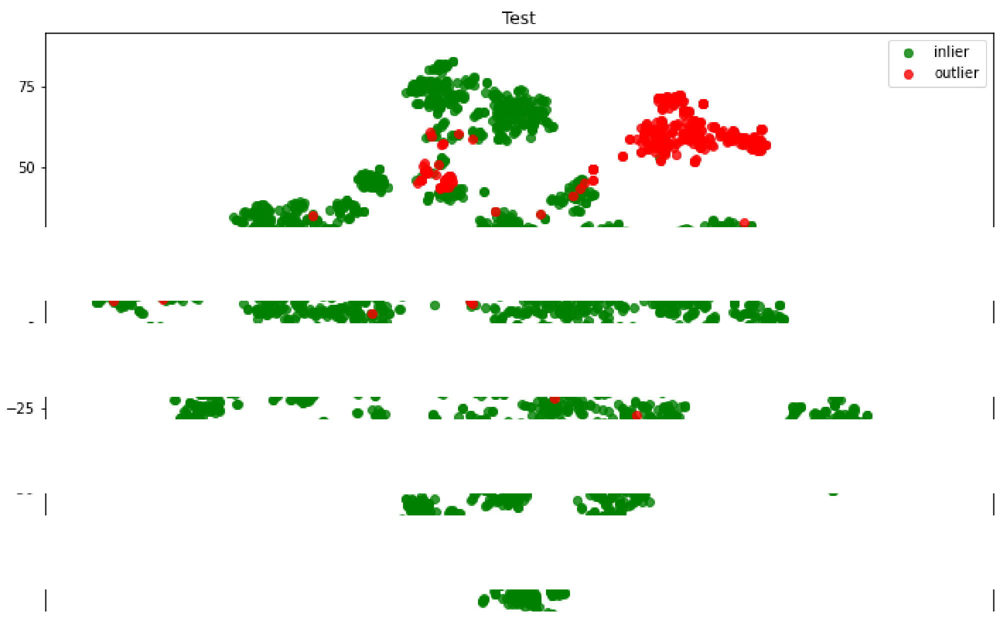
```
# Loading the credit card data set for outlier detection
# All the columns have been transformed using PCA
# and the original information regarding the trans-action  is  not  available  due  to  pr
data = pd.read_csv("/data/workspace_files/creditcard.csv")
data.head()
```

	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

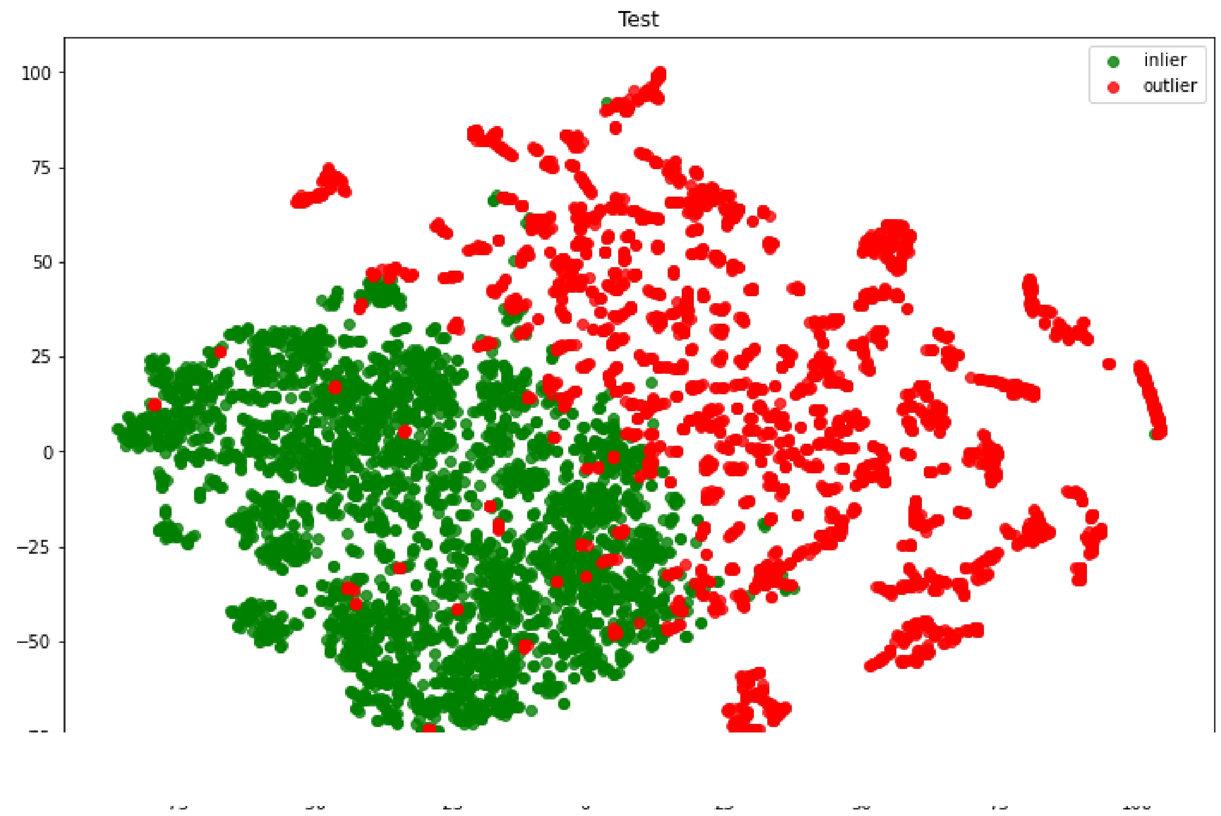
5 rows × 31 columns

```
# Scaling the dataset
from sklearn.preprocessing import MinMaxScaler
from sklearn.pipeline import Pipeline

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


scaler = mm_scaler

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

inlier_sample = inlier_sample.drop("Class", axis =1).values
outliers = outliers.drop("Class", axis =1).values

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


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)
```



```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y, rando

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

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



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)





tsne_plot(X_bal,Y_bal)
```



AutoEncoder with Ensembles Implementation

```
#This class implements an autoencoder with the option to mask layers randomly such that ea

class RandomAutoEncoder(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:
        # Randomly masks layers

        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)


            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:

        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:
        # From a pickle file

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

    def get_encoder(self) -> keras.Sequential:

        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:

        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:

               self.mask_weights()

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

Implementation of Adaptive Learning

```
from tensorflow import keras
import warnings, math

# The class below implements adaptive sampling in the autoencoder implementation
# The initial epochs are trained with lesser data. As we progress with training, more data
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]


        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_


        self.alpha = np.exp(np.log(1/self.start_data_ratio)/self.epochs)
        self.shuffle = shuffle


        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):

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

        return batch, batch

    def on_epoch_end(self):


        current_x_size = int(self.current_batch_size*self.steps_per_epoch)
        self.current_x = self.x[:current_x_size]


        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}%


        self.current_batch_size = int(self.start_batch_size * self.alpha**self.epoch)
        self.epoch += 1
```

Ensemble training

```
from datetime import datetime

MODEL_PARAMS ={"input_dim": X_train.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 = n_models = 15
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")


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

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

    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)

    ensemble.append(model)
```

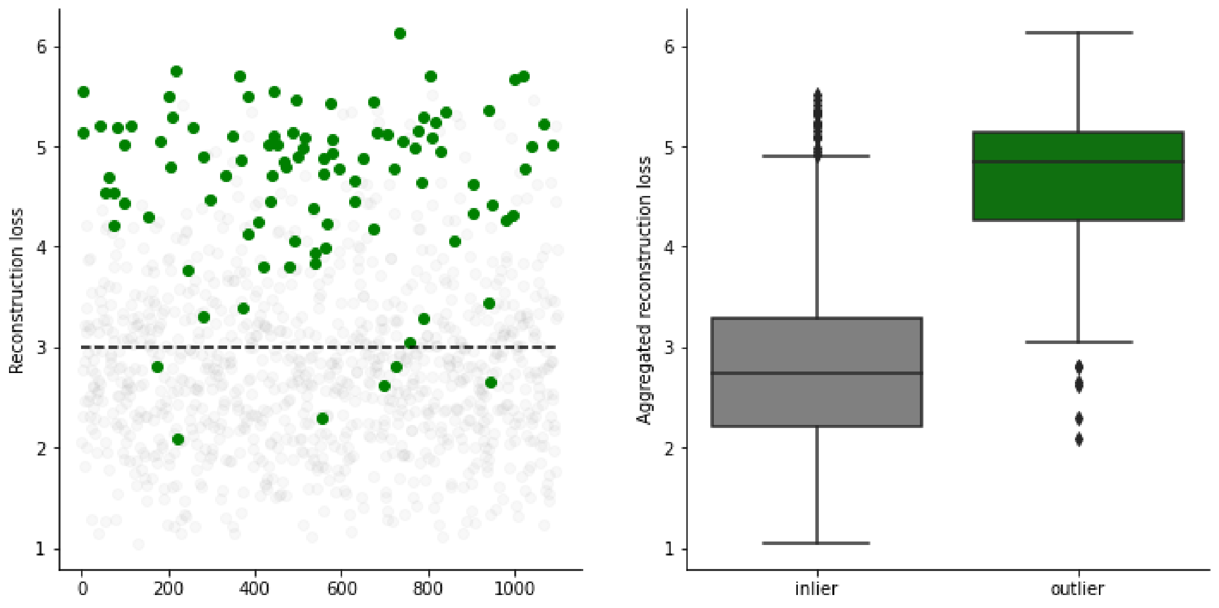
```
Fitting the models...
```

```
Epoch 1/20
5/5 [==============================] - 0s 12ms/step - loss: 0.0639
Epoch 2/20
5/5 [==============================] - 0s 11ms/step - loss: 0.0605
Epoch 3/20
5/5 [==============================] - 0s 11ms/step - loss: 0.0575
Epoch 4/20
5/5 [==============================] - 0s 12ms/step - loss: 0.0540
Epoch 5/20
5/5 [==============================] - 0s 12ms/step - loss: 0.0514
Epoch 6/20
5/5 [==============================] - 0s 12ms/step - loss: 0.0487
Epoch 7/20
5/5 [==============================] - 0s 11ms/step - loss: 0.0461
Epoch 8/20
5/5 [==============================] - 0s 11ms/step - loss: 0.0441
Epoch 9/20
5/5 [==============================] - 0s 11ms/step - loss: 0.0421
Epoch 10/20
5/5 [==============================] - 0s 12ms/step - loss: 0.0398
Epoch 11/20
5/5 [==============================] - 0s 11ms/step - loss: 0.0379
Epoch 12/20
5/5 [==============================] - 0s 11ms/step - loss: 0.0360
Epoch 13/20
5/5 [==============================] - 0s 12ms/step - loss: 0.0346
Epoch 14/20
5/5 [==============================] - 0s 11ms/step - loss: 0.0329
Epoch 15/20
5/5 [==============================] - 0s 11ms/step - loss: 0.0315
Epoch 16/20
5/5 [==============================] - 0s 11ms/step - loss: 0.0305
Epoch 17/20
5/5 [==============================] - 0s 12ms/step - loss: 0.0291
Epoch 18/20
5/5 [==============================] - 0s 11ms/step - loss: 0.0276
Epoch 19/20
5/5 [==============================] - 0s 11ms/step - loss: 0.0269
Epoch 20/20
5/5 [==============================] - 0s 11ms/step - loss: 0.0260
Epoch 1/20
5/5 [==============================] - 0s 11ms/step - loss: 0.0650
Epoch 2/20
5/5 [==============================] - 0s 11ms/step - loss: 0.0589
Epoch 3/20
5/5 [==============================] - 0s 11ms/step - loss: 0.0523
Epoch 4/20
5/5 [==============================] - 0s 11ms/step - loss: 0.0465
Epoch 5/20
5/5 [==============================] - 0s 11ms/step - loss: 0.0413
Epoch 6/20
5/5 [==============================] - 0s 12ms/step - loss: 0.0372
Epoch 7/20
5/5 [==============================] - 0s 12ms/step - loss: 0.0337
Epoch 8/20
5/5 [==============================] - 0s 12ms/step - loss: 0.0308
Epoch 9/20
```

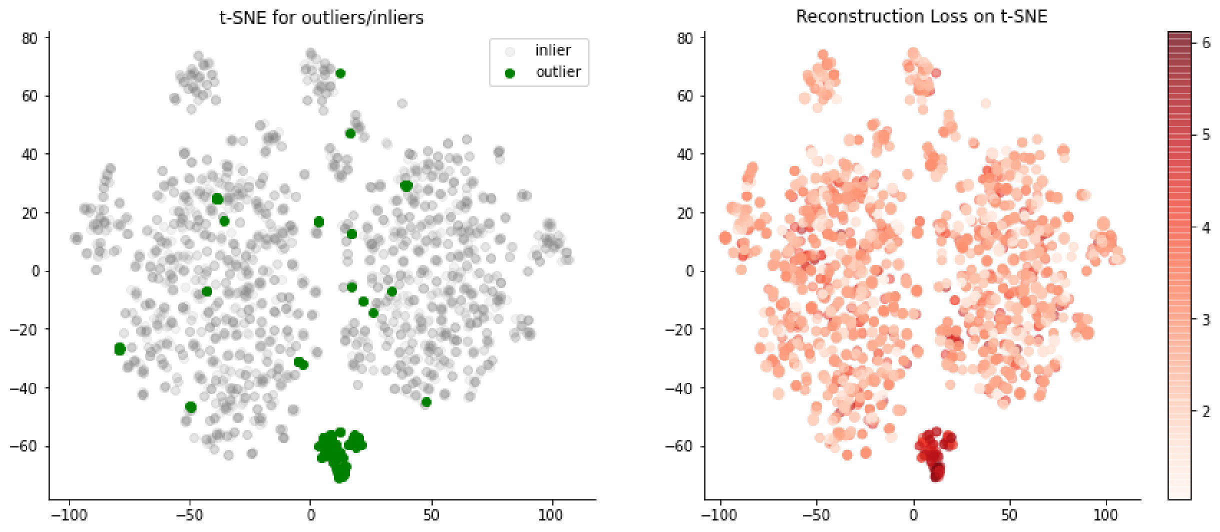
Evaluating the ensemble

```python
def ensemble_run(ensemble, input_data, input_labels, contamination):

    results = dict()

    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
    )

    results["reconstruction_loss"] = reconstruction_loss

    scaler = StandardScaler(with_mean=False)
    reconstruction_loss = scaler.fit_transform(reconstruction_loss)

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

    # threshold calibration
    threshold = np.quantile(median_loss, contamination)

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

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

    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


# evaluating the ensemble on the test set

eval_results = ensemble_run(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)


minmax_experiment = "/data/workspace_files/eval_results_2021-09-23-18-34-31.pickle"
with open(minmax_experiment, "rb") as handle:
    results = pickle.load(handle)
```

```
reconstruction_loss = results["reconstruction_loss"]


# 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="green",
            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", "green"], ax=ax)
ax.set_xlabel("")
ax.set_ylabel("Aggregated reconstruction loss")
sns.despine()

plt.show()
```



```
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]


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="GREEN", alpha=1, label="outlier")
plt.legend()
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.show()
```

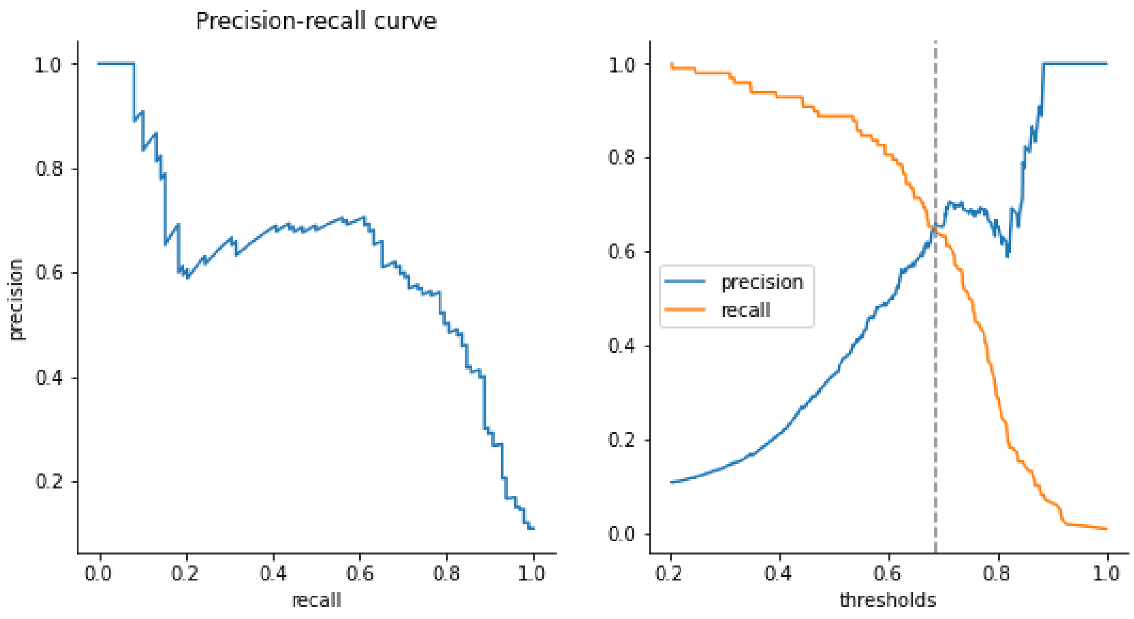


```
prec, recall, thresholds = results["precision_recall_curve"]
```

```
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)} threshol
```

```
     Best precision 0.65, recall: 0.65 at 0.69 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()
```



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. Tradeoff with varying the threshold: Precision vs Recall - 0.65 Precision and 0.65 Recall at around 0.69 threshold