POLITECNICO DI TORINO

Master Course in Computer Engineering

Master Thesis

Machine Learning based Classification System to detect tampered Big Data in the context of a Simulated Aqueduct

Advisor
Prof. Paolo Ernesto Prinetto

Candidate
Marco Zanobini

ACADEMIC YEAR 2018/2019

Table Of Contents

Abstract

In order to secure data of an Aqueduct, a simulated environment has been developed inside the Cyber Range in “Superior Institute Mario Boella” located in Turin. The object of this thesis is to create and compare Classification algorithms based on Machine Learning which best fits the proposed model. This Classifier, trained only using "true" class data, should recognize tampered data caused by anomalies or malevolent attackers.

To be perfectly suitable with the simulation environment all the code has been developed in Python taking advantage of the most widely used libraries for Machine Learning. To maintain a high level of abstraction, usability and portability Keras has been adopted as a framework to interact to Tensorflow backend.

Tensorflow is an open source library created by Google and designed to provide different toolkits at different levels of abstraction, from Estimators (High level, object-oriented API) to Python Tensorflow which wraps C++ Kernel (lower level API).

The most common alternative is Scikit-learn which provides a smaller amount of possibilities regarding the number of different models to adopt, in particular it lacks all the part of deep learning but it has some algorithms that perfectly fit the study case.

Both supervised and unsupervised approaches offer good performances after a correct parameters’ tuning. Due to the fact that all the data used for training belong to the same class the problem could be located under the labels of “One Class Classification” and “Anomaly/Outlier/Novelty Detection”.

The first attempt to develop the correct Machine Learning algorithm uses one of the most intuitive models, the Logistic Regressor, which is the common choice for a Classifier. Obviously this model does not fit perfectly the study case background and it’s not considered as more than a first draft by this study. For that reason, its structure is maintained with the following model but due to the inconsistency of its prediction it is not considered as a real model and suitable for a comparison with others.

After the comparison both the SVM and the simple Autoencoder register exactly the same result, even if the first one is much faster, correctly classifying nearly the 86% of the examples. Better performance for the Local Outlier Factor algorithm, with just 1 misprediction its result is quite impressive considering the speed for training. Despite the absence of particular drawbacks in these models another algorithm takes the lead with 100% accuracy and F1 score equal to 1.

The AEKDE model outstands the others, correctly recognizing all the samples and having the lowest need for the user to interacts with them fulfilling the basic goals of a Machine Learning system.

Setup

Cyber Range Simulated Aqueduct

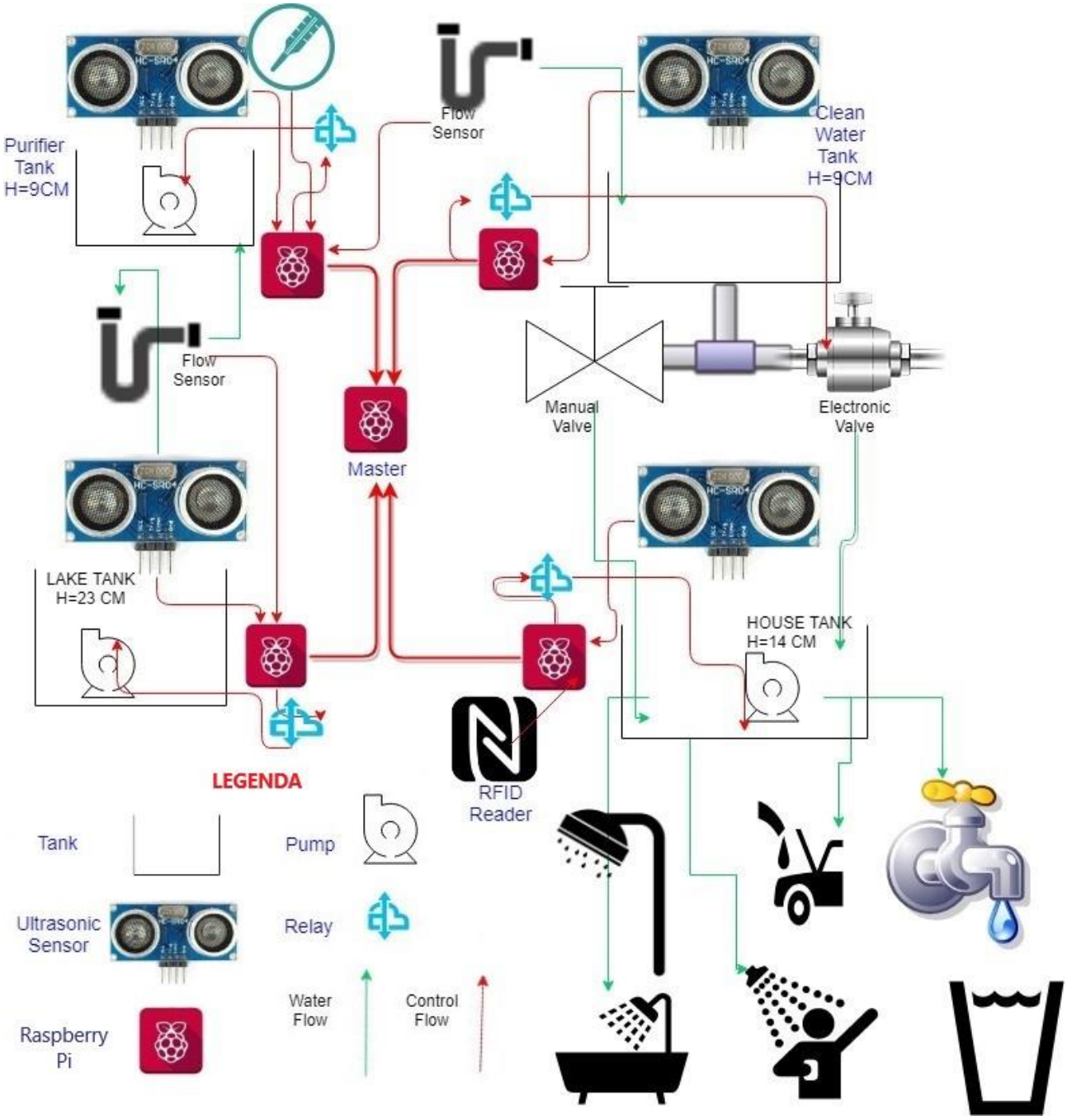

Figure 1. Scheme of the Aqueduct Simulated Model

The Tanks

There are 4 water tanks equipped with a pump to let the water flow to the next destination and an ultrasonic sensor to measure the distance expressed in centimetres from the top of the tank to the water level. Moreover, the “Purifier Tank” has another sensor plunged in the water to measure temperature in Celsius degrees.

Figure 2. Temperature Sensor

Pipes

The gum pipes which connect the tanks have a flow binary sensor to measures if the water is flowing through the pipes

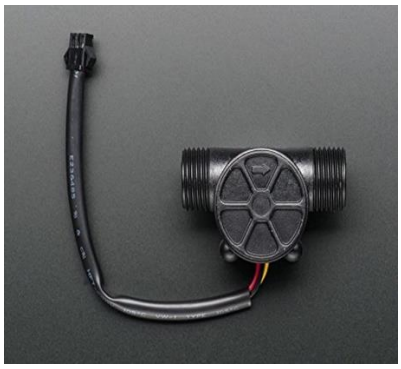

Figure 3. Flow Sensor

Valves

Located between “Clean Water Tank” and “House Tank” there are 2 valves: 1 manual and the other controlled electronically. The electronic valve can be unlocked by an electronic input sent by the Raspberry anytime a precise magnetic card is read by a RFID Card Reader.

Figure 4. Electronic Valve

Rasperry Pi 3

Raspberry Pi 3 is a low-cost, single-board mini computer developed for teaching purpose which is spreading for its great capability and versatility at a very cheap price.

This system is controlled by a network composed by 5 Raspberry Pis: 1 master and 4 slaves. The master will act as a central server on which our solution will be developed while the slaves will send all the data recorded by the sensors to the master where the database is stored.

Figure 5. Raspberry Pi 3 Model B

Specifications

- Quad Core 1.2GHz Broadcom BCM2837 64bit CPU
- 1GB RAM
- BCM43438 wireless LAN and Bluetooth Low Energy (BLE) on board
- 100 Base Ethernet
- 40-pin extended GPIO
- 4 USB2 ports
- 4 Pole stereo output and composite video port
- Full size HDMI
- CSI camera port for connecting a Raspberry Pi camera
- DSI display port for connecting a Raspberry Pi touchscreen display
- Micro SD port for loading your operating system and storing data
- Upgraded switched Micro USB power source up to 2.5A

Software Requirements

- Python 3.5.2
- Virtualenv 16.1.0
- Pip 18.1.0
- Tensorflow 1.12.0
- Keras 2.2.4
- Scikit learn 0.20.1

SciKit-Learn Overview

SciKit-Learn was initially developed for Google Summer of Code project in 2007 but it was largely adopted for its simplicity in implementing Machine Learning inside a system. Based upon the SciPy stack, composed by several libraries for scientific and mathematical manipulation such as NumPy, Pandas and Matplotlib this library offers both supervised and unsupervised learning algorithms via

a consistent interface in Python. One of the main drawbacks is that none of the models use a Neural Network making the library incomplete to be used exclusively for this project.

Scikit-Learn is characterized by a clean, uniform, and streamlined API, as well as by very useful and complete online documentation which are the main reasons for its fast and wide adoption.

Tensorflow Overview

Tensorflow is a free and open-source library created by the Google Brain team to support the production of machine learning algorithm at first internally at Google and in a second time release to the public. TensorFlow is available on 64-bit Linux, macOS, Windows, and mobile computing platforms including Android and iOS. Its front-end API provides a user-friendly interface using Python to build applications with the framework which will be then executed in high-performance C++.

TensorFlow allows developers to create dataflow graphs, that are structures which describe the movement of the data through the graph, or a series of processing nodes. Each node in the graph represents a mathematical operation, and each connection or edge between nodes is a multidimensional data array, or tensor. While all the nodes and tensors are Python objects, provided with high-level of programming abstraction, actual math operations are performed through libraries of transformation written in C++ in the lowest level.

One interesting benefit of this framework is the TensorBoard visualization suite, it lets the developer inspect and profile the way graphs run by way of an interactive, web-based dashboard. That offers a full view of your model, showing graphs and stats with all the customized metrics defined in addition to the complete dataflow representation.

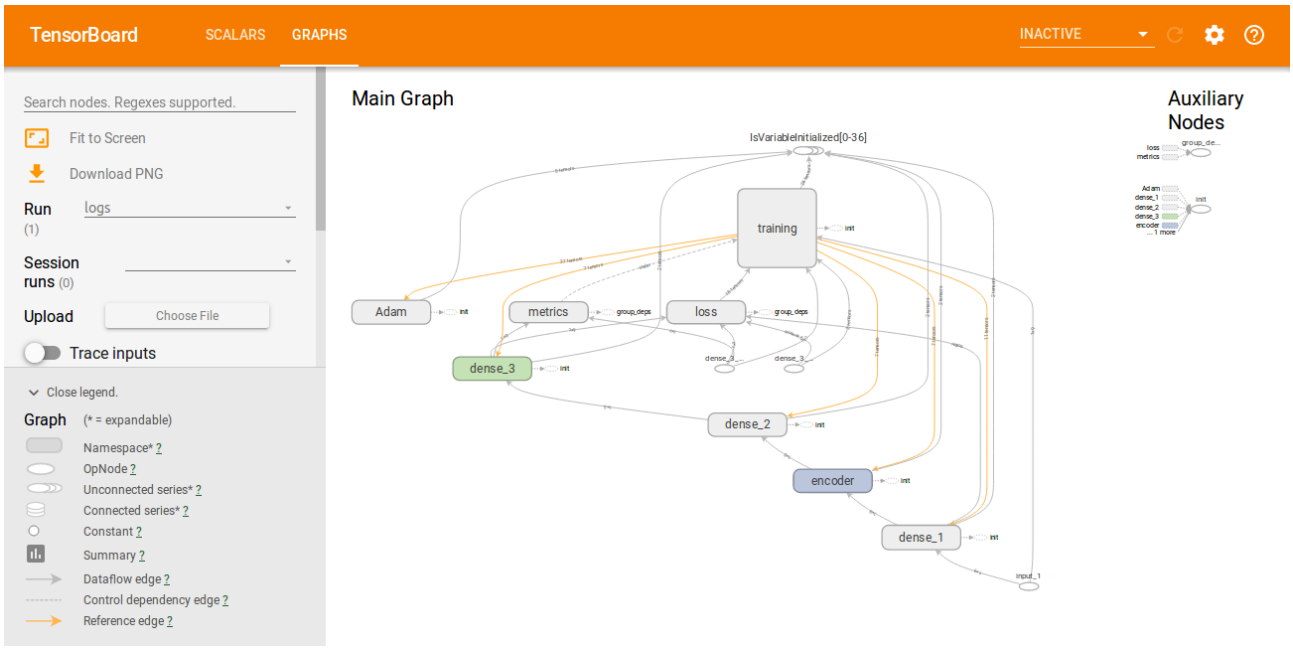

Figure 6. Tensorboard

Keras Overview

Keras is a high-level neural networks API written in Python and supporting multiple back-end neural network computation engines such as Tensorflow and Theano. It was created to be user

friendly, modular, easy to extend, and to work with Python. The API was “designed for human beings, not machines,” and “follows best practices for reducing cognitive load.”

Since the interface is backed primarily by Google, Tensorflow can be used at different levels of abstraction integrating the model provided by the API to the fully customizable of the lowest level creating an algorithm totally adaptive.

Preliminary Analysis

Developing a Machine Learning model starting from the creation of the system in a supervised environment grants the truthfulness of the data, all of it is real and no malfunctions or attacks should be registered in it. Nevertheless, robustness should be adopted by the model to avoid improper training and compromise the entire decision process.

Talking about feature engineering no evident correlation between the features could be revealed.

Despite that, some logic constraints could be introduced to check data trustworthiness prior inference phase, increasing speed and granting reliability to the prediction. Even if many features, as for example temperature, could be bounded following common sense, in this research they will not restricted due to their “too specific” nature. A limit on binary numbers would be a solid choice due to its great versatility and to the fact that it will straightforward to configure it and check it.

Anyway, this improvement will be implemented in a second time to not influence the comparative among all the models studied.

Data Collection

The data from all the different sensor is collected and merged on the base of the timestamp in a unique csv file named *Database.csv* which will be used in the following machine learning models.

This file will contain the following fields:

- *timestamp*: time of data insertion in the db in the format yyyy-dd-mm hh:mm:ss
- *lake_dist*: distance of the water level from the sensor (positioned on the top of the lake tank)
- *purifier_dist*: distance of the water level from the sensor (positioned on the top of the purifier tank)
- *clean_dist*: distance of the water level from the sensor (positioned on the top of the clean water tank)
- *house_dist*: distance of the water level from the sensor (positioned on the top of the house tank)
- *lake_pump*: binary describing if the lake pump is working (1) or not (0)
- *purifier_pump*: binary describing if the purifier pump is working (1) or not (0)
- *house_pump*: binary describing if the house pump is working (1) or not (0)
- *purifier_temp*: temperature expressed in Celsius degrees of the water in the purifier tank
- *clean_valve*: binary describing if the clean valve is open (1) or not (0)

Code

```
---------------------------------------- dbfetch.py ---------------------------
------------------
#!/usr/bin/python3
import PyMySQL
import csv

# Open database connection
db =
PyMySQL.connect("localhost","cpswminf_pandey1","CINI2017","cpswminf_project" )

# prepare a cursor object using cursor() method
cursor = db.cursor()

tables = ["DISTANCE", "PUMPSTATUS", "TEMPERATURE", "VALVESTATUS"]
headers = [["timestamp", "lake_dist", "purifier_dist", "clean_dist",
"house_dist"], ["timestamp", "lake", "purifier", "house"], ["timestamp",
"purifier"], ["timestamp", "clean"]]

for i in range(len(tables)):
      print ("Downloading "+tables[i]+" database...")
      # Prepare SQL query to INSERT a record into the database.
      sql = "SELECT * FROM '%s'" % (tables[i])
      try:
            # Execute the SQL command
            cursor.execute(sql)
            # Fetch all the rows in a list of lists.
            results = cursor.fetchall()

            with open(tables[i]+'.csv', 'wb') as csvfile:
                  filewriter = csv.writer(csvfile, delimiter=',',
                                          quotechar='|',
quoting=csv.QUOTE_MINIMAL)
                  if len(headers[i]) == 1:
                        filewriter.writeheaders[i](headers[i][0])
                  elif len(headers[i]) == 2:
```

```
                        filewriter.writeheaders[i](headers[i][0],
headers[i][1])
                  elif len(headers[i]) == 3:
                        filewriter.writeheaders[i](headers[i][0],
headers[i][1], headers[i][2])
                  elif len(headers[i]) == 4:
                        filewriter.writeheaders[i](headers[i][0],
headers[i][1], headers[i][2], headers[i][3])
                  elif len(headers[i]) == 5:
                        filewriter.writeheaders[i](headers[i][0],
headers[i][1], headers[i][2], headers[i][3], headers[i][4])
                  else:
                        print("Headers uncorrectly defined in "+tables[i])
                  for row in results:
                        if len(row) == 1:
                              filewriter.writerow(row[0])
                        elif len(row) == 2:
                              filewriter.writerow(row[0], row[1])
                        elif len(row) == 3:
                              filewriter.writerow(row[0], row[1], row[2])
                        elif len(row) == 4:
                              filewriter.writerow(row[0], row[1], row[2],
row[3])
                        elif len(row) == 5:
                              filewriter.writerow(row[0], row[1], row[2],
row[3], row[4])
                        else:
                              print("No Data Found in "+tables[i])
      except:
            print ("Error: unable to fetch data")

# disconnect from server
db.close()
```

Test Client

To test our system a client is implemented for testing purpose, it will request an input vector or a csv file composed by rows containing a vector each (excluding the first row, expected to be the header), it opens a connection through a socket with the server and sends this vector as a simple string.

Then it loops until the user press the termination command (the ‘q’ key).

Code

```
import socket, os.path, datetime, sys, argparse, csv

parser = argparse.ArgumentParser(description='Linear Classifier Client: for testing purpose.')
parser.add_argument('-hn','--host', dest='host', action='store', metavar='HOSTNAME',
                    default='127.0.0.1',help='server hostname (default: 127.0.0.1)')
parser.add_argument('-p','--port', dest='port', action='store', metavar='PORT', type=int,
                    default=50001,help='server port number (default: 50001)')

args = parser.parse_args()

host = args.host
port = args.port

def socket_exchange(msg):
  s = socket.socket()
  s.connect((host, port))
  s.send(msg.encode('utf-8'))
  s.shutdown(socket.SHUT_WR)
  data = s.recv(1024).decode('utf-8')
  print(data)
  s.close()

print("\nCommands:\n  _ data vector (Syntax: n,n,n,n,n,n,n,n,n)\n  _ 'xxxx.csv' file containing
inputs\n  _ 'h' to print this help\n  _ 'q' to quit\n")
#Prediction Phase
while True:
    i = input("Enter command: ")
    if not i:
        print("No input. Please retry\n")
        continue
    elif len(str(i)) == 1:
      if ord(str(i)) == 113:
        break
      elif ord(str(i)) == ord('h'):
        print("\nCommands:\n  _ data vector (Syntax: n,n,n,n,n,n,n,n,n)\n  _ 'xxxx.csv' file containing
inputs\n  _ 'h' to print this help\n  _ 'q' to quit\n")
    elif i[-4:] == ".csv":
      with open(i, "r") as f:
        for row in list(csv.reader(f, delimiter=','))[1:]:
          msg = ""
          first = True
          for field in row[:-1]:
```

```
            if not first:
              msg += ","
            msg += str(field)
            first = False
          socket_exchange(msg)
      else:
        socket_exchange(i)
print("Client shutdown correctly. Bye Bye")
```

Best Practices

All the models created are provided with different functionalities common for anyone to improve performances of the model granting more usability, speed and accuracy. Most notable are:

- Binary Check
- Model Checkpoint
- Additional Arguments

Binary Check

Adding a check for binary value could prevent malicious attacks and anomalies very dangerous for the integrity of the system and easy to create. This issue is easily resolved applying a binary constraint on the value of selected features using an array initialized at the top of the program. Valves are typical examples of values on which this check could be applied, binary perfectly describe their state “on/off”. Implementing this check will prevent uses of the trained model intercepting the request upstream of it, reducing overhead.

Model Checkpoint

Some models could have long training time and that should be repeated every time the program stops running. That can happen voluntarily or not, which is the worst case due to its unpredictability. Training many times in a short period could be terrible for performance especially in a real-time background where responses should be given instantly. To solve this issue granting durability and speed a checkpoint is created right after the training to avoid the restart of this process on every start up. Checkpoint could be overwritten by the user, which can be a good option to train on more examples giving more precision and accuracy at the price of a little time waste.

Additional Arguments

A lot of extra options can be specified by the use of arguments while executing the Python code. This is realized using the argparse library, which intuitively implements a parser for the extra arguments of the function invocation. Among the multiple options some are common to every piece of code: the help (*-h*) to show every option, the warning (*-w*) to activate extra messages and graphs invisible in normal execution and the overwrite (*-ow*) for the model checkpoint. Other remarkable options are the ones to specify server ip/port address useful in test phase to run multiple instances of the server.

Machine Learning

Introduction

"**Machine learning** (ML) is the scientific study of algorithms and statistical models that computer systems use to effectively perform a specific task without using explicit instructions, relying on models and inference instead." (Wikipedia, the free enciclopedia, n.d.)

Machine Learning is a subfield of Artificial Intelligence used to fit data inside a mathematical model which will, in a second moment, predict the output or making a decision on a set of data "autonomously" applying a label on it. The main difference from traditional computational approach used in computer science is the ability to generate the algorithm in a second time using data already collected, this offers the possibility to make easier life to programmers by creating the algorithm and recognizing patterns and correlation among data not always clear.

By convention in machine learning, you'll write the equation for a model slightly differently but most of the time it can be reduced to:

$$y' = b + w_n x_n$$

where:

- y' is the predicted **label** (a desired output).
- b is the **bias** (the y-intercept), sometimes referred to as w_0.
- w_n is a n-dimension weight vector of features. Weight is the same concept as the "slope" m in the traditional equation of a line.
- x_n is the vector of the **features** (a known input).

Let's highlight two phases of a model's life:

- **Training** a model simply means learning (determining) good values for all the weights and the bias from labelled examples.
- **Inference** means applying the trained model to unlabelled examples. That is, you use the trained model to make useful predictions (y').

Two of the most widely adopted machine learning methods are **supervised learning** which trains algorithms based on example input and output data that is labelled by humans, and **unsupervised learning** which provides the algorithm with no labelled data in order to allow it to find structure within its input data.

A deep dive into the Training Phase

In supervised learning, a machine learning algorithm builds a model by examining many examples and attempting to find a model that minimizes loss; this process is called **empirical risk minimization**.

Loss is the penalty for a bad prediction. That is, **loss** is a number indicating how bad the model's prediction was on a single example. If the model's prediction is perfect, the loss is zero; otherwise,

the loss is greater. The goal of training a model is to find a set of weights and biases that have *low* loss, on average, across all examples.

One of the most popular and widely used algorithms for machine learning models is **gradient descent**. This is an iterative approach, the purpose of which is to find the minimum of the loss by proceeding step by step inspecting the gradient (or derivative) of different points and taking the next step in the direction of the negative gradient. More precisely the gradient vector has both magnitude and direction, to find the next point the algorithm multiplies a scalar factor known as **learning rate** (or **step size**) by the gradient.

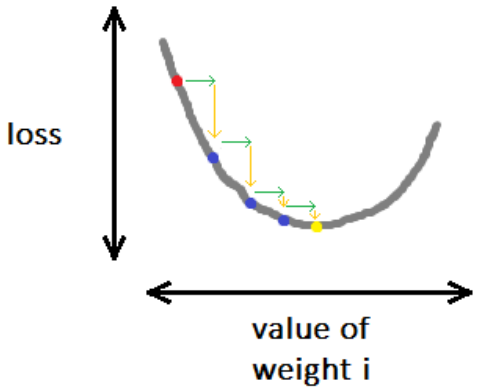

Figure 7. Gradient Descent Algorithm

The total number of examples used to calculate the gradient is called a **batch.** Obviously using all the examples contained in the database (**full-batch**) could bring the computational time for each iteration to a very long time. The **stochastic gradient descent (SGD)** resolves it in a very extreme way reducing the batch to just 1 example chosen at random.

One less drastic way to resolve the time issue is to adopt the **mini-batch SGD** which is a compromise between the previous methods, using generally from 10 to 1000 examples taken randomly this method reduces noise as in full batch but in a much lower time.

Therefore, good performances of the model are not always related to how much training is done and that's why even training should be correctly balanced. Stepping further too much with training will cause **overfitting**, which will occur when the model fits too well to the training data. When that happens, the model loses its ability to generalize to new examples and starts mispredicting lowering severely its accuracy.

Nevertheless, decreasing under a certain limit training will cause the opposite, the model without enough data to train on will not be able to create a function to associate correctly input to targets remaining too generic and nearly useless. This phenomenon is known as **underfitting**.

Data Sets

The data should be correctly split into different sets to guarantee the best result.
Usually data is divided in 2: **training set** and **test set**. Training set obviously is the largest and it's used to fit data inside the model while test set is used to verify the performances according to different metrics useful to tweak the model and reach its goals after some loops.

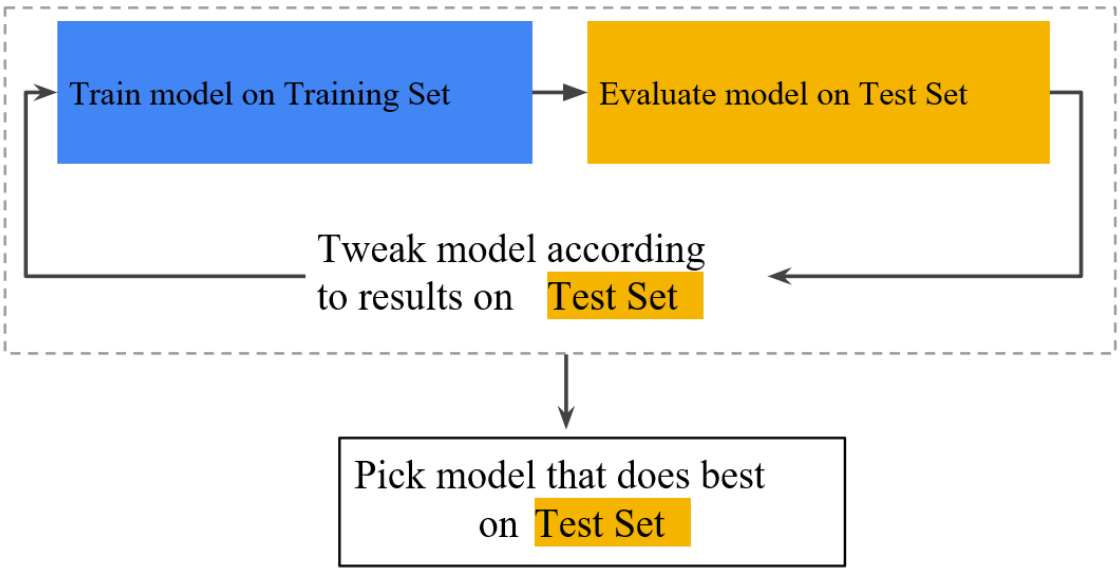

Figure 8. Training with two sets (Google, n.d.)

It's good practice to add a third set of data called **validation set** with the same function of the test set, leaving to the latter the task to confirm the model and avoiding overfitting.

The new training lifecycle will resemble to this:

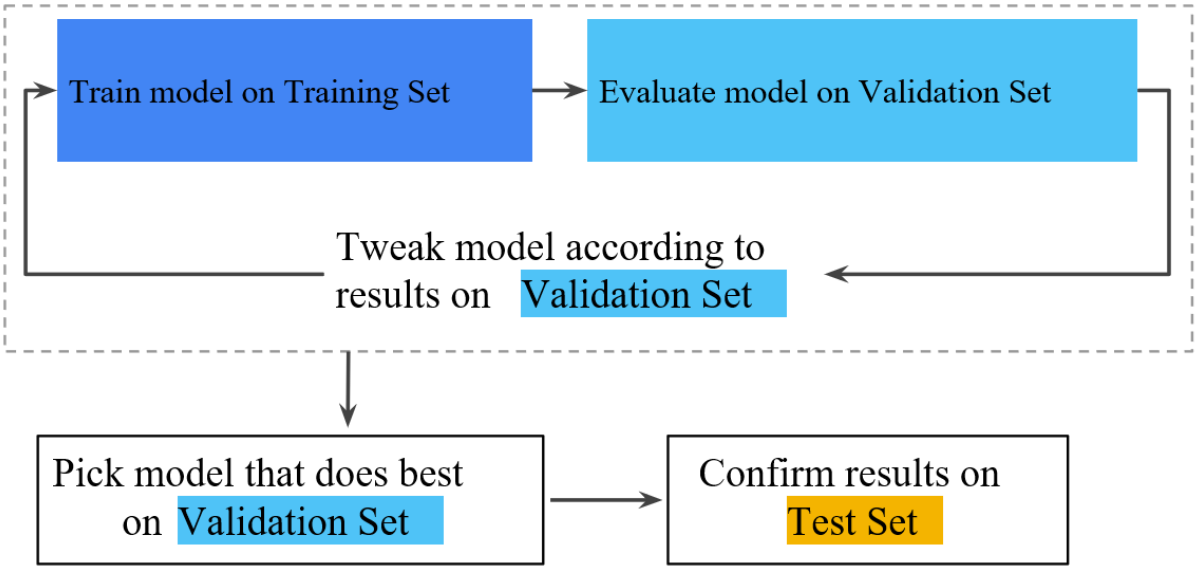

Figure 9. Training with 3 sets (Google, n.d.)

Obviously in our study case both the train and the validation set will contain data belonging only to the "ham"/"not spam" class while the test set will contain both classes. Since many instruments we are going to use will just consider the approach with just train and test set, in the code the validation set could be referred as test set and vice versa.

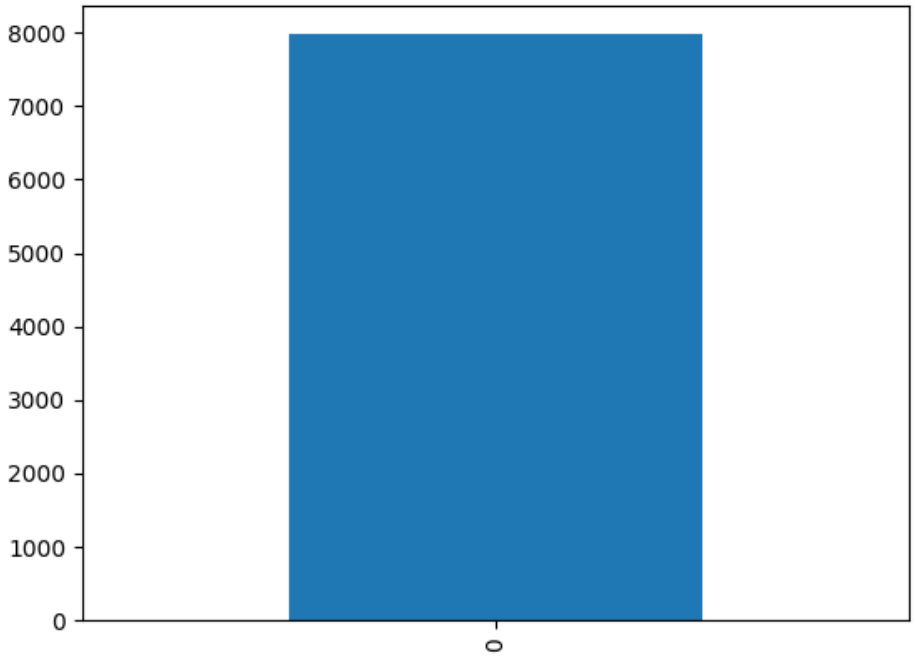

Figure 10. Dataset containing only class 0 ("no spam")

Classification

A Classification problem subsists when there is a bunch of classes and each input should be located into one or more of this classes. Considering our study case as a binary problem there are only 4 possible outcomes:

A **true positive (TP)** is an outcome where the model *correctly* predicts the *positive* class. Similarly, a **true negative (TN)** is an outcome where the model *correctly* predicts the *negative* class.

A **false positive (FP)** is an outcome where the model *incorrectly* predicts the *positive* class. And a **false negative (FN)** is an outcome where the model *incorrectly* predicts the *negative* class.

These results are commonly displayed in a table layout called **confusion matrix**, a 2x2 matrix with one axis containing the predicted results ("True" and "False") and the expected results in the other axis ("True" and "False"). The cells belonging to the table exactly represent the 4 outcomes previously presented, more precisely the number of true positives/negatives and false positives/negatives.

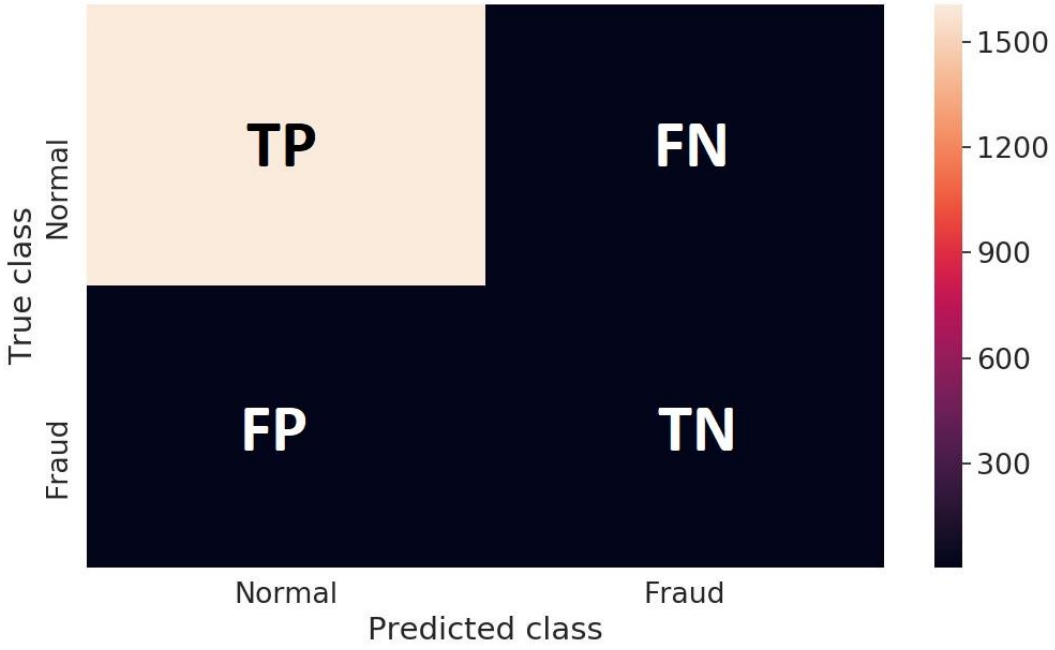

Figure 11. Confusion Matrix

One of the main metrics to evaluate model "goodness" is accuracy, that is defined as follow:

$$Accuracy = \frac{\#\ of\ correct\ predictions}{Total\ \#\ of\ predictions} = \frac{TP + TN}{TP + TN + FP + FN}$$

Accuracy alone doesn't tell the full story when you're working with a **class-imbalanced data set** where there is a significant disparity between the number of positive and negative labels.

Other useful measurements are:

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

F1 Score

Our model would be easier to judge if we have one single scalar instead of Precision and Recall, It's given by the following formula:

$$F1 = 2 \times \frac{Precision * Recall}{Precision + Recall}$$

F1 Score keeps a **balance** between Precision and Recall. We use it if there is uneven class distribution, as precision and recall may give misleading results!
So we use F1 Score as a comparison indicator between Precision and Recall Numbers!

AUC (Area Under Curve)

The area measured under the ROC curve gives a scale-invariant and classification-threshold-invariant measure of how good a model is. This because the ROC plots 2 parameters at every possible decision threshold: True Positive Rate (TPR) vs. False Positive Rate (FPR).

$$TPR = \frac{TP}{TP+FN} \qquad FPR = \frac{FP}{FP+TN}$$

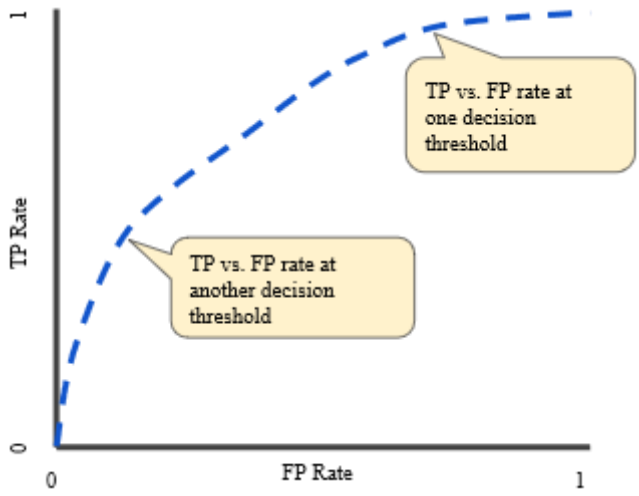

Figure 12. ROC Curve (Google, n.d.)

The evident drawback of this metric is given by its threshold-invariance, in fact the AUC results to be an average among all possible threshold and it will not give a specific judgement of our case.

First Model: Logistic Regression

Many problems require a probability estimate as output. Logistic regression is an extremely efficient mechanism for calculating probabilities. Practically speaking, you can use the returned probability in either of the following two ways:

- "As is"
- Converted to a binary category.

In order to map a logistic regression value to a binary category, you must define a **classification threshold** (also called the **decision threshold**). A value above that threshold indicates "spam"; a value below indicates "not spam" or “ham”.

You might be wondering how a logistic regression model can ensure output that always falls between 0 and 1. As it happens, a **sigmoid function**, defined as follows, produces output having those same characteristics:

$$y' = \frac{1}{1 + e^{-z}}$$

where:

- y' is the output of the logistic regression model for a particular example.
- z is $b + w_1x_1 + w_2x_2 + ... w_Nx_N$ (output of the linear layer of a model trained with logistic regression)

Loss function for Logistic Regression

The loss function for linear regression is squared loss. The loss function for logistic regression is **Log Loss**, which is defined as follows:

$$Log\ Loss = \sum_{(x,y)\in D} -y\ log(y') - (1-y)\ log(1-y')$$

where:

- $(x, y) \in D$ is the data set containing many labelled examples, which are (x, y) pairs.
- y is the label in a labeled example. Since this is logistic regression, every value of y must either be 0 or 1.
- y' is the predicted value (somewhere between 0 and 1), given the set of features in x.

Code

```
from __future__ import print_function
import math
```

```
from IPython import display
from matplotlib import cm
from matplotlib import gridspec
from matplotlib import pyplot as plt
import numpy as np
import pandas as pd
from sklearn import metrics
import tensorflow as tf
from tensorflow.python.data import Dataset
import csv
import time
import os
from sklearn.decomposition import PCA
from sklearn.metrics import f1_score
from sklearn import preprocessing
import argparse
import socket
import sys
import itertools

parser = argparse.ArgumentParser(description='Activate a Linear Classifier.')
parser.add_argument('-d','--delayed', dest='delayed', action='store_true',
                    help='activate delayed mode')
parser.add_argument('-c','--check', dest='check_mode', action='store_true',
                    help='start check mode')
parser.add_argument('-w','--warn', dest='warning', action='store_true',
                    help='activate warnings')
parser.add_argument('--chk_file', dest='chk_fp', action='store',
metavar='NAME',
                    default='class_checkpoint',help='used to specify
checkpoint class name')
parser.add_argument('-hn','--host', dest='host', action='store',
metavar='HOSTNAME',
                    default='127.0.0.1',help='server hostname (default:
127.0.0.1)')
parser.add_argument('-p','--port', dest='port', action='store',
metavar='PORT', type=int,
                    default=50001,help='server port number (default: 50001)')
```

```
args = parser.parse_args()


tf.logging.set_verbosity(tf.logging.ERROR)
pd.options.display.max_rows = 10
pd.options.display.float_format = '{:.1f}'.format
columns = ["lake_dist",
     "purifier_dist",
     "clean_dist",
     "house_dist",
     "lake_pump",
     "purifier_pump",
     "house_pump",
     "purifier_temp",
     "clean_valve"]

delayed = args.delayed
check_mode = args.check_mode
show_warnings = args.warning

# Server Address
host = args.host
port = args.port

# Checkpoint folder
filepath = args.chk_fp

# Delayed mode checklist file
chk_fname = "delayed_check.csv"

aqueduct_dataframe = pd.read_csv("Database.csv", sep=",")

aqueduct_dataframe = aqueduct_dataframe.reindex(
    np.random.permutation(aqueduct_dataframe.index))

def preprocess_features(aqueduct_dataframe):
  """Prepares input features from California housing data set.
```

```
  Args:
    aqueduct_dataframe: A Pandas DataFrame expected to contain data
      from the California housing data set.
  Returns:
    A DataFrame that contains the features to be used for the model, including
    synthetic features.
  """
  selected_features = aqueduct_dataframe[columns]
  processed_features = selected_features.copy()
  return processed_features

def preprocess_targets(aqueduct_dataframe):
  """Prepares target features (i.e., labels) from California housing data set.

  Args:
    aqueduct_dataframe: A Pandas DataFrame expected to contain data
      from the California housing data set.
  Returns:
    A DataFrame that contains the target feature.
  """
  output_targets = pd.DataFrame()
  # Create a boolean categorical feature representing whether the
  # median_house_value is above a set threshold.
  output_targets["is_it_spam"] = aqueduct_dataframe["is_it_spam"]
  return output_targets

# Count total row number
row_count = len(aqueduct_dataframe)
training_set_count = (int)(row_count * 7 / 100) #real value is '/10'

# Choose the first 7% examples for training.
training_examples =
preprocess_features(aqueduct_dataframe.head(training_set_count))
training_targets =
preprocess_targets(aqueduct_dataframe.head(training_set_count))

# Choose the last 3% examples for validation.
```

```
#validation_examples = preprocess_features(aqueduct_dataframe.tail(row_count -
training_set_count))
#validation_targets = preprocess_targets(aqueduct_dataframe.tail(row_count -
training_set_count))

validation_examples =
preprocess_features(aqueduct_dataframe.tail(int(row_count/10) -
training_set_count))
validation_targets =
preprocess_targets(aqueduct_dataframe.tail(int(row_count/10) -
training_set_count))

if show_warnings:
  # Double-check that we've done the right thing.
  print("Training examples summary:")
  display.display(training_examples.describe())
  print("Validation examples summary:")
  display.display(validation_examples.describe())

  print("Training targets summary:")
  display.display(training_targets.describe())
  print("Validation targets summary:")
  display.display(validation_targets.describe())

################################################
################### PCA ########################

training_examples_scaled =
preprocessing.scale(training_examples.astype(float))
if show_warnings:
  pca = PCA(2)
  pca.fit(training_examples_scaled)
  X_pca = pca.transform(training_examples_scaled)
  plt.scatter(X_pca[:,0], X_pca[:,1])
  plt.show()
training_examples_scaled = pd.DataFrame(training_examples_scaled,
columns=columns)
```

```
validation_examples_scaled =
preprocessing.scale(validation_examples.astype(float))
if show_warnings:
  pca.fit(training_examples_scaled)
  X_pca = pca.transform(validation_examples_scaled)
  plt.scatter(X_pca[:,0], X_pca[:,1])
  plt.show()
validation_examples_scaled = pd.DataFrame(validation_examples_scaled,
columns=columns)

################################################
################################################2

def construct_feature_columns(input_features):
  """Construct the TensorFlow Feature Columns.

  Args:
    input_features: The names of the numerical input features to use.
  Returns:
    A set of feature columns
  """
  return set([tf.feature_column.numeric_column(my_feature)
              for my_feature in input_features])

def my_input_fn(features, targets, batch_size=1, shuffle=True,
num_epochs=None):
    """Trains a linear regression model.

    Args:
      features: pandas DataFrame of features
      targets: pandas DataFrame of targets
      batch_size: Size of batches to be passed to the model
      shuffle: True or False. Whether to shuffle the data.
      num_epochs: Number of epochs for which data should be repeated. None =
repeat indefinitely
    Returns:
      Tuple of (features, labels) for next data batch
    """
```

```
    # Convert pandas data into a dict of np arrays.
    features = {key:np.array(value) for key,value in dict(features).items()}

    # Construct a dataset, and configure batching/repeating.
    ds = Dataset.from_tensor_slices((features,targets)) # warning: 2GB limit
    ds = ds.batch(batch_size).repeat(num_epochs)

    # Shuffle the data, if specified.
    if shuffle:
      ds = ds.shuffle(10000)

    # Return the next batch of data.
    features, labels = ds.make_one_shot_iterator().get_next()
    return features, labels

def train_linear_classifier_model(
    learning_rate,
    steps,
    batch_size,
    training_examples,
    training_targets,
    validation_examples,
    validation_targets):
  """Trains a linear regression model.

  In addition to training, this function also prints training progress
information,
  as well as a plot of the training and validation loss over time.

  Args:
    learning_rate: A `float`, the learning rate.
    steps: A non-zero `int`, the total number of training steps. A training
step
      consists of a forward and backward pass using a single batch.
    batch_size: A non-zero `int`, the batch size.
    training_examples: A `DataFrame` containing one or more columns from
      `aqueduct_dataframe` to use as input features for training.
```

```
    training_targets: A `DataFrame` containing exactly one column from
      `aqueduct_dataframe` to use as target for training.
    validation_examples: A `DataFrame` containing one or more columns from
      `aqueduct_dataframe` to use as input features for validation.
    validation_targets: A `DataFrame` containing exactly one column from
      `aqueduct_dataframe` to use as target for validation.

  Returns:
    A `LinearRegressor` object trained on the training data.
  """

  periods = 10
  steps_per_period = steps / periods

  # Checkpoint Strategy configuration
  run_config = tf.contrib.learn.RunConfig(
    model_dir=filepath,
    keep_checkpoint_max=1)

  # Create a linear classifier object.
  my_optimizer =
tf.train.GradientDescentOptimizer(learning_rate=learning_rate)#,
l1_regularization_strength=0.1)
  my_optimizer = tf.contrib.estimator.clip_gradients_by_norm(my_optimizer,
5.0)
  linear_classifier = tf.estimator.LinearClassifier(
      feature_columns=construct_feature_columns(training_examples),
      optimizer=my_optimizer,
      config=run_config,
      #metric_ops=
  )
  if not tf.train.checkpoint_exists(filepath):
    # Create input functions.
    training_input_fn = lambda: my_input_fn(training_examples,
                                            training_targets["is_it_spam"],
                                            batch_size=batch_size)
    predict_training_input_fn = lambda: my_input_fn(training_examples,
```

```
training_targets["is_it_spam"],
                                                    num_epochs=1,
                                                    shuffle=False)
    predict_validation_input_fn = lambda: my_input_fn(validation_examples,

validation_targets["is_it_spam"],
                                                      num_epochs=1,
                                                      shuffle=False)

    # Train the model, but do so inside a loop so that we can periodically
assess
    # loss metrics.
    print("Training model...")
    print("LogLoss (on training data):")
    training_log_losses = []
    validation_log_losses = []
    for period in range (0, periods):
      # Train the model, starting from the prior state.
      linear_classifier.train(
          input_fn=training_input_fn,
          steps=steps_per_period
      )
      # Take a break and compute predictions.
      training_probabilities =
linear_classifier.predict(input_fn=predict_training_input_fn)
      training_probabilities, training_predictions =
itertools.tee(training_probabilities)
      training_predictions = np.array([item['classes'][0] for item in
training_predictions], dtype=int)
      training_probabilities = np.array([item['probabilities'] for item in
training_probabilities])

      validation_probabilities =
linear_classifier.predict(input_fn=predict_validation_input_fn)
      validation_probabilities = np.array([item['probabilities'] for item in
validation_probabilities])
```

```
      training_log_loss = metrics.log_loss(training_targets,
training_probabilities, labels=[0,1])
      validation_log_loss = metrics.log_loss(validation_targets,
validation_probabilities, labels=[0,1])
#      training_f1_score, train_update_op =
tf.contrib.metrics.f1_score(training_targets, np.array([item[1] for item in
training_probabilities]))
      # Occasionally print the current loss.
      print("  period %02d : LogLoss=%0.2f f1=%0.2f" % (period,
training_log_loss, f1_score(training_targets, training_predictions,
labels=[0,1])))
      # Add the loss metrics from this period to our list.
      training_log_losses.append(training_log_loss)
      validation_log_losses.append(validation_log_loss)
    print("Model training finished.")

    # Output a graph of loss metrics over periods.
    plt.ylabel("LogLoss")
    plt.xlabel("Periods")
    plt.title("LogLoss vs. Periods")
    plt.tight_layout()
    plt.plot(training_log_losses, label="training")
    plt.plot(validation_log_losses, label="validation")
    plt.legend()

  return linear_classifier

def is_it_binary(df_col):
  for elem in df_col:
#    if not(math.isclose(elem,float(0)) or math.isclose(elem,float(1))): #for
floats
    if elem!=0 and elem!=1:
      return False
  return True

# Returns an array of positions of the suspicious values (evaluating
local/global boundaries)
def new_vulnerability_test(timestamp, input_record):
```

```
  clean_df = aqueduct_dataframe[aqueduct_dataframe["is_it_spam"]==0]
  past_years = clean_df[clean_df["timestamp"].str[5:7]==str(timestamp)[7:9]]
  # if there is data of the same month (even in different years)
  if len(past_years)!=0:
    examined_df = past_years
  else:
    examined_df = clean_df

  bad_data = list()
  # First we check for non binary values
  for i in range(len(columns)):
    if is_it_binary(examined_df[columns[i]]) and (input_record[i]!=0 and
input_record[i]!=1):
      if len(bad_data)==0:
        bad_data.append([-1])
      bad_data.append([i])

  if len(bad_data)==0:
    for i in range(len(columns)):
      col_df = examined_df[columns[i]]
#      print(columns[i]+" max:"+str(col_df.max())+" min:"+str(col_df.min())+"
avg:"+str(col_df.mean())+" std:"+str(col_df.std())+"
bin:"+str(is_it_binary(col_df)))
      if input_record[i]>round(col_df.mean()+col_df.std()) or
input_record[i]<round(col_df.mean()-col_df.std()):
        if len(bad_data)==0:
          bad_data.append([1])
        bad_data.append([i, col_df.max(), col_df.min()])
  return bad_data

def check_new_data():
  if os.path.isfile(chk_fname):
    with open(chk_fname, "r") as f:
      for row in list(csv.reader(f, delimiter=','))[1:]:
        timestamp =
row[0].replace("[","").replace("]","").replace("'","").strip()
```

```
        cols =
row[1].replace("[","").replace("]","").replace("'","").replace("
","").split(",")
        vals =
row[2].replace("[","").replace("]","").replace("'","").replace("
","").split(",")
        maxs =
row[3].replace("[","").replace("]","").replace("'","").replace("
","").split(",")
        mins =
row[4].replace("[","").replace("]","").replace("'","").replace("
","").split(",")
        new_record =
np.asarray(np.array(row[5].replace("[","").replace("]","").replace("'","").rep
lace(" ","").split(",")), np.int)
        print("ALERT! "+str(len(cols))+" suspicious data detected!\n")
        for i in range(len(cols)):
          print(cols[i]+"   curr_val: "+vals[i]+" [max:"+maxs[i]+",
min:"+mins[i]+"]")
        i = input("\nDigit (A)ccept to mark data as good or anything else to
mark it as spam: ")
        if str(i)=="Accept" or str(i)=="accept" or (len(str(i))==1 and
(ord(str(i)) == ord("A") or ord(str(i)) == ord("a"))):
          spam_detected = 0
          print("Data saved as NO SPAM\n")
        else:
          spam_detected = 1
          print("Data saved as SPAM\n")
        new_train(new_record, spam_detected, timestamp)
    os.remove(chk_fname)
  else:
    print("No data to examine\n")

def new_train(new_record, spam_detected, timestamp):
  global aqueduct_dataframe
  global linear_classifier
  ts = [timestamp]
  df1 = preprocess_features(pd.DataFrame([new_record], columns=columns))[:1]
```

```
  df1.loc[0,"timestamp"] = timestamp
  df1.loc[0,"is_it_spam"] = spam_detected
  aqueduct_dataframe = aqueduct_dataframe.append(df1, sort=True)
  training_examples = preprocess_features(aqueduct_dataframe.tail(20))
  training_targets = preprocess_targets(aqueduct_dataframe.tail(20))


  if show_warnings:
    print(df)
    print(spam_detected)
  training_input_fn = lambda: my_input_fn(training_examples,
                                          training_targets,
                                          batch_size=20)
  linear_classifier = linear_classifier.train(
        input_fn=training_input_fn,
        steps=20
  )

  new_record = new_record.tolist()
  new_record.append(spam_detected)
  ts.extend(new_record)
  with open('Database.csv', 'a') as csvfile:
    filewriter = csv.writer(csvfile, delimiter=',', quotechar='|',
quoting=csv.QUOTE_NONE)
    filewriter.writerow(ts)

def insert_record (new_record):

  global linear_classifier
  global aqueduct_dataframe

  test_input = preprocess_features(pd.DataFrame([new_record],
columns=columns))
  df = test_input[:1]
  predict_input_fn = tf.estimator.inputs.pandas_input_fn(x=df, shuffle=False)

  predict_results = linear_classifier.predict(input_fn=predict_input_fn)
  # Print the prediction results.
```

```
print("\nPrediction results:")
for i, prediction in enumerate(predict_results):
  if show_warnings:
    print(prediction)
  print(str(int(prediction['classes'][0]))+"\n")
spam_detected = int(prediction['classes'][0])

# Add the prediction to the csv file and train on it
# Create the timestamp
ts = time.gmtime()
row = [time.strftime("%Y-%m-%d %H:%M:%S", ts)]
bad_data = []

if spam_detected == 0:
  bad_data = new_vulnerability_test(row, new_record)
  if show_warnings:
    print(bad_data)
  if len(bad_data)!=0:
    spam_detected = bad_data[0][0]

if spam_detected == -1:
  # if value not binary in binary field
  spam_detected = 1
  print("Binary value not respected. Data saved as SPAM\n")

elif spam_detected == 1:
  if len(bad_data) == 0:
    print("Data saved as SPAM")
  else:
    print("ALERT! "+str(len(bad_data)-1)+" suspicious data detected!\n")
    if delayed:
      # Save new suspicious data to file
      exists = os.path.isfile(chk_fname)
      with open(chk_fname, "a") as csv_file:
        filewriter = csv.writer(csv_file, delimiter=',')
        cols = []
        vals = []
        maxs = []
```

```
          mins = []
          if not exists:
            filewriter.writerow(["timestamp", "column", "curr_val", "max",
"min", "record"])
          for pos in bad_data[1:]:
            cols.append(columns[pos[0]])
            vals.append(new_record[pos[0]])
            maxs.append(pos[1])
            mins.append(pos[2])

          check_row = [row, cols, vals, maxs, mins, new_record.tolist()]
          filewriter.writerow(check_row)

        print("Data saved in "+chk_fname+" for delayed check\n")
        return "Suspicious data, further analysis requested\n"
      for pos in bad_data[1:]:
        print(str(columns[pos[0]])+"   curr_val: "+str(new_record[pos[0]])+"
[max:"+str(pos[1])+", min:"+str(pos[2])+"]")
      i = input("\nDigit (A)ccept to mark data as good or anything else to
mark it as spam: ")
      if str(i)=="Accept" or str(i)=="accept" or (len(str(i))==1 and
(ord(str(i)) == ord("A") or ord(str(i)) == ord("a"))):
        spam_detected = 0
        print("Data saved as NO SPAM\n")
      else:
        print("Data saved as SPAM\n")

  new_train(new_record, spam_detected, row)
  if spam_detected == 1:
    return "Data saved as SPAM\n"
  else:
    return "Data saved as NO SPAM\n"

# Create classifier and start training
linear_classifier = train_linear_classifier_model(
    learning_rate=0.01,
    steps=200,
    batch_size=20,
```

```
    training_examples=training_examples_scaled,
    training_targets=training_targets,
    validation_examples=validation_examples_scaled,
    validation_targets=validation_targets)

# Check suspicious data delayed
if(check_mode):
  check_new_data()
# Else start the server
else:
  s = socket.socket()
  s.bind((host,port))
  print("Server Started")
  s.listen(1)
  while True:
    print("Waiting for incoming connections...")
    c, addr = s.accept()
    print("Connection from: " + str(addr))
    array = ''
    msg = ''
    while True:
      data = c.recv(1024).decode('utf-8')
      if not data:
        break
      array += data
    try:
      new_record = np.asarray(np.array(array.split(",")), np.int)
      print(new_record)
      msg = insert_record(new_record)
    except ValueError:
      msg = "Incorrect Syntax. Retry\n"
      print(msg)
      c.send(msg.encode('utf-8'))
      c.shutdown(socket.SHUT_WR)
      c.close()
      continue
    if new_record.size != len(columns):
      msg = "Wrong # of values. Retry\n"
```

```
        print(msg)
    c.send(msg.encode('utf-8'))
    c.shutdown(socket.SHUT_WR)
    c.close()
```

Results

As expected, training the Linear Classifier on just one class brings the model to predict always 0 (“no spam”) on every input. To overcome this problem, a sort of filter is implemented to recognize “abnormal” data even when it is not recognized by our system. This filter simply checks if the features of the input lie in the range between maximum and minimum recorded with a margin of error equal to the standard deviation, if not, suspicious results are analysed by a supervisor in real time or saved in a separate file for further analysis, leaving the decision to the human. In that way is possible to start collecting some tampered data on which the model can be trained resolving the problem at the price of involving the assistance of a supervisor, losing some of the advantages gained by a machine learning approach. Moreover, in not-delayed mode, since the system needs a human supervisor to check real time data this inevitably results in a slow down or even in a deadlock. Condition which can be resolved using a multi-threading environment keeping an eye on concurrency.

Second Model: SVM

In machine learning, support-vector machines (SVMs, also support-vector networks) are supervised learning models with associated learning algorithms that analyse data used for classification and regression analysis. Given a set of training examples, each marked as belonging to one or the other of two categories, an SVM training algorithm builds a model that assigns new examples to one category or the other, making it a non-probabilistic binary linear. An SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall.

It is easy to visualize it in 2 or 3 dimensions, obviously each dimension represents a feature (it will be a 9-dimensions space in the studied case). Objective of the SVM is to find a hyperplane between the 2 classes to maximize the gap between them

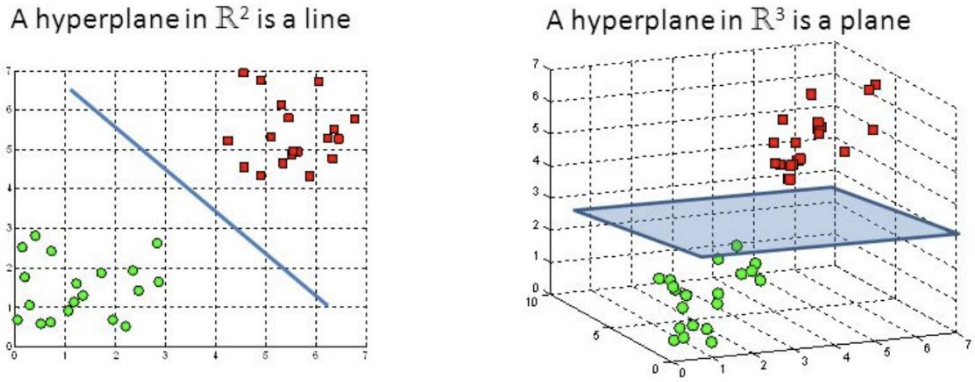

Figure 13. Representation of a Hyperplane in different dimensions (Gandhi, 2018)

Most essential parameters to tune are the *kernel* which is left to the default value "rbf" (good also for non-linear hyperplanes) and *gamma* which controls how exactly the model fits the training data. Obviously values too high of gamma could reflect in model overfitting.

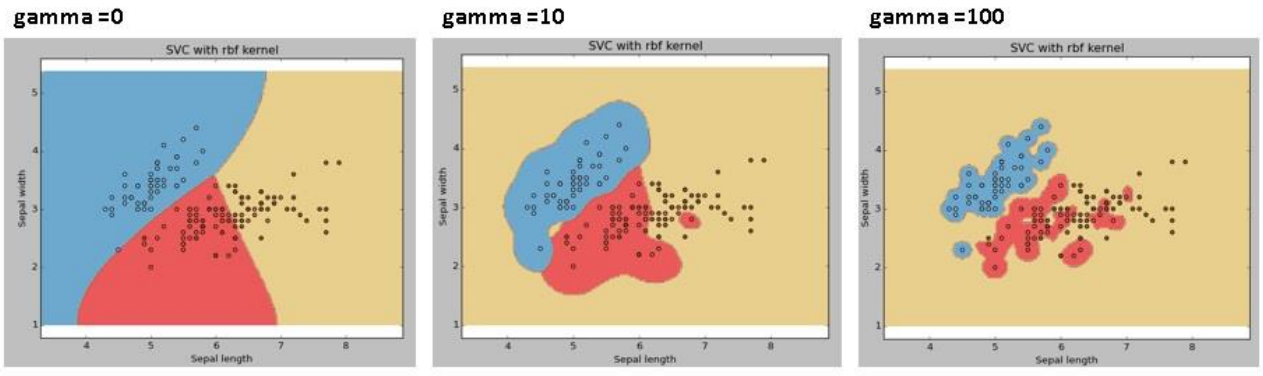

Figure 14. Gamma parameter influence (Ray, 2017)

This could be practically helpful in this case where we need a One-Class Classifier (***OCC***), a classifier that will be trained on just one class and recognizing data as "spam" or "not spam".

Code

```
import numpy as np
import pandas as pd
from sklearn import utils
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn import svm
from sklearn import metrics
import os, csv, argparse, socket, sys, time
from sklearn.externals import joblib

parser = argparse.ArgumentParser(description='Activate a Linear Classifier.')
parser.add_argument('-w','--warn', dest='warning', action='store_true',
                    help='activate warnings')
parser.add_argument('--chk_file', dest='chk_fp', action='store',
metavar='NAME',
                    default='occ_svm.model',help='used to specify checkpoint
model name')
parser.add_argument('-ow','--overwrite', dest='ow', action='store_true',
                    help='train again and overwrite the model')
parser.add_argument('-hn','--host', dest='host', action='store',
metavar='HOSTNAME',
                    default='127.0.0.1',help='server hostname (default:
127.0.0.1)')
parser.add_argument('-p','--port', dest='port', action='store',
metavar='PORT', type=int,
                    default=50001,help='server port number (default: 50001)')

args = parser.parse_args()


pd.options.display.max_rows = 10
pd.options.display.float_format = '{:.1f}'.format
columns = ["lake_dist",
     "purifier_dist",
     "clean_dist",
     "house_dist",
     "lake_pump",
     "purifier_pump",
```

```
     "house_pump",
     "purifier_temp",
     "clean_valve"]

show_warnings = args.warning

# Server Address
host = args.host
port = args.port

# Checkpoint folder
filepath = args.chk_fp
overwrite = args.ow

# import the CSV from Database.csv file
# this will return a pandas dataframe.
aqueduct_dataframe = pd.read_csv("Database.csv", sep=",", low_memory=False)

# let's take a look at the types of attack labels are present in the data.
aqueduct_dataframe["is_it_spam"].value_counts().plot(kind='bar')
plt.show()

# we're using a one-class SVM, so we need.. a single class. the dataset
'label'
# column contains multiple different categories of attacks, so to make use of
# this data in a one-class system we need to convert the attacks into
# class 1 (no_spam) and class -1 (spam)
aqueduct_dataframe.loc[aqueduct_dataframe['is_it_spam'] != 0, "is_it_spam"] =
-1
aqueduct_dataframe.loc[aqueduct_dataframe['is_it_spam'] == 0, "is_it_spam"] =
1



def preprocess_features(aqueduct_dataframe):
  """Prepares input features from personal data set.

  Args:
    aqueduct_dataframe: A Pandas DataFrame expected to contain data
```

```
      from the personal aqueduct data set.
  Returns:
    A DataFrame that contains the features to be used for the model, including
    synthetic features.
  """
  selected_features = aqueduct_dataframe[columns]
  processed_features = selected_features.copy()
  return processed_features

def preprocess_targets(aqueduct_dataframe):
  """Prepares target features (i.e., labels) from Aqueduct data set.

  Args:
    aqueduct_dataframe: A Pandas DataFrame expected to contain data
      from the California housing data set.
  Returns:
    A DataFrame that contains the target feature.
  """
  output_targets = pd.DataFrame()
  # Create a boolean categorical feature representing whether the
  # median_house_value is above a set threshold.
  output_targets["is_it_spam"] = aqueduct_dataframe["is_it_spam"]
  return output_targets

def new_train(new_record, prediction, timestamp):
  global features
  global target
  global model
  ts = timestamp
  df1 = preprocess_features(pd.DataFrame([new_record], columns=columns))[:1]
  df1.loc[0,"timestamp"] = timestamp
  df1.loc[0,"is_it_spam"] = prediction
  training_ex = preprocess_features(df1)
  features = features.append(training_ex, sort=True)
  training_target = preprocess_targets(df1)
  target = target.append(training_target, sort=True)

  if show_warnings:
```

```
    print(df1)
    print(prediction)

  model = model.fit(training_ex)

  spam_detected = 0
  if prediction == -1:
    spam_detected = 1
  new_record = new_record.tolist()
  new_record.append(spam_detected)
  ts.extend(new_record)
  print(ts)
  with open('Database.csv', 'a') as csvfile:
    filewriter = csv.writer(csvfile, delimiter=',', quotechar='|',
quoting=csv.QUOTE_NONE)
    filewriter.writerow(ts)
  joblib.dump(model, filepath, compress=9)

def insert_record (new_record):

  global model
  global aqueduct_dataframe

  test_input = preprocess_features(pd.DataFrame([new_record],
columns=columns))
  df = test_input[:1]

  pred = model.predict(df)
  # Print the prediction results.
  print("\nPrediction result: "+str(pred))

  # Add the prediction to the csv file and train on it
  # Create the timestamp
  ts = time.gmtime()
  row = [time.strftime("%Y-%m-%d %H:%M:%S", ts)]

  new_train(new_record, pred, row)
  if pred == 1:
```

```
    return "Data saved as NO SPAM\n"
  else:
    return "Data saved as SPAM\n"

# grab out the is_it_spam value as the target for training and testing. since
we're
# only selecting a single column from the `aqueduct_dataframe` dataframe,
we'll just get a
# series, not a new dataframe
target = preprocess_targets(aqueduct_dataframe)

# find the proportion of outliers we expect (aka where `is_it_spam == -1`).
because
# target is a series, we just compare against itself rather than a column.
outliers = target[target['is_it_spam'] == -1]
if show_warnings:
  print("outliers.shape", outliers.shape)
  print("outlier fraction", outliers.shape[0]/target.shape[0])

# drop label columns from the dataframe. we're doing this so we can do
# unsupervised training with unlabelled data. we've already copied the label
# out into the target series so we can compare against it later.
#aqueduct_dataframe.drop("timestamp", axis=1, inplace=True)
#aqueduct_dataframe.drop("is_it_spam", axis=1, inplace=True)
features = preprocess_features(aqueduct_dataframe)

# check the shape for sanity checking.
if show_warnings:
  print(features.shape)

#Divide train and test data
#train_data, test_data, train_target, test_target = train_test_split(features,
target, train_size = 0.8)
train_data = features
train_target = target

# set nu (which should be the proportion of outliers in our dataset)
if outliers.shape[0] != 0:
```

```
  nu = outliers.shape[0] / target.shape[0]
else:
  nu = 0.05
if show_warnings:
  print("nu", nu, "\n")

if os.path.isfile(filepath) and not overwrite:
  model = joblib.load(filepath)
else:
  model = svm.OneClassSVM(nu=nu, kernel='rbf', gamma=0.00005)
  model.fit(train_data)

  joblib.dump(model, filepath, compress=9)

if show_warnings:
  preds = model.predict(train_data)
  targs = train_target

  print("Training:")
  print("accuracy: ", metrics.accuracy_score(targs, preds))
  print("precision: ", metrics.precision_score(targs, preds))
  print("recall: ", metrics.recall_score(targs, preds))
  print("f1: ", metrics.f1_score(targs, preds))
#  print("area under curve (auc): ", metrics.roc_auc_score(targs, preds))
  print()

  with open('test_input.csv', 'r') as csvfile:
    readCSV = csv.reader(csvfile, delimiter=',')
    test_data = list(readCSV)
  new_cols = columns.copy()
  new_cols.append("is_it_spam")
  test = pd.DataFrame(test_data[1:], columns=new_cols)
  # Converting all objects to float and dropping incomplete rows
  test = test.replace(r'^\s*$', np.nan, regex=True).apply(lambda x:
pd.to_numeric(x, errors='coerce')).dropna()
  test.loc[test['is_it_spam'] != 0, "is_it_spam"] = -1
  test.loc[test['is_it_spam'] == 0, "is_it_spam"] = 1
  preds = model.predict(preprocess_features(test))
```

```
  targs = preprocess_targets(test)

  missing = set(targs['is_it_spam']) - set(preds)
  print("Testing:")
  if len(missing) > 0:
    print("Value "+str(missing)+" not present. Can not calculate metrics.")
  else:
    print("accuracy: ", metrics.accuracy_score(targs, preds))
    print("precision: ", metrics.precision_score(targs, preds))
    print("recall: ", metrics.recall_score(targs, preds))
    print("f1: ", metrics.f1_score(targs, preds))
    print("area under curve (auc): ", metrics.roc_auc_score(targs, preds))


# Start the server
s = socket.socket()
s.bind((host,port))
print("Server Started (CTRL+C to exit)")
s.listen(1)
try:
  while True:
    print("Waiting for incoming connections...")
    c, addr = s.accept()
    print("Connection from: " + str(addr))
    array = ''
    msg = ''
    while True:
      data = c.recv(1024).decode('utf-8')
      if not data:
        break
      array += data
    try:
      new_record = np.asarray(np.array(array.split(",")), np.int)
      print(new_record)
      if new_record.size != len(columns):
        msg = "Wrong # of values. Retry\n"
        print(msg)
      else:
```

```
            msg = insert_record(new_record)
        except ValueError:
          msg = "Incorrect Syntax. Retry\n"
          print(msg)
          c.send(msg.encode('utf-8'))
          c.shutdown(socket.SHUT_WR)
          c.close()
          continue
        c.send(msg.encode('utf-8'))
        c.shutdown(socket.SHUT_WR)
        c.close()
    except KeyboardInterrupt:
      s.shutdown(socket.SHUT_RDWR)
      s.close()
      print ("\nServer shutdown correctly. Bye Bye")
```

Results

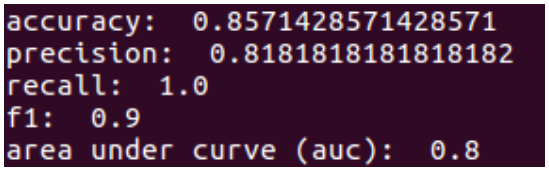

Figure 15. SVM Results

Tweaking the hyperparameters of the model the best result obtained has an f1 score of 0.9 on the data set. It has discovered 4 out of 5 spam and misclassified just 1 of the “good” data.

After the results obtained with Logistic Regression, SVM gives pretty good advantages correctly classifying a wide percentage of data without human intervention. This algorithm confirms the expectations as the most used for One Class Classification among supervised methods.

Third Model: Local Outlier Factor

Exploring the unsupervised algorithms this option seems the first natural step to follow, it is still intuitive because of its classification which is based on a multidimensional representation of the input as points in this space as the SVM. The main difference involves how the outliers are identified, while the SVM uses distance this uses density.

The **Local Outlier Factor (LOF)** algorithm is an anomaly detection method which computes the local density of each point based on its nearest neighbours.

All the algorithm can be schematized in 3 points:

- Defined k as the number of neighbours specified by parameter and *k-distance* as the distance between the chosen point and its k^{th} nearest neighbour calculate, for each point, the *reachability distance* with any other point. This is defined as the maximum between the *k-distance* of the point and the distance to the point selected.

$$reach\_dist(a,b) = \max(k\_distance(b), dist(a,b))$$

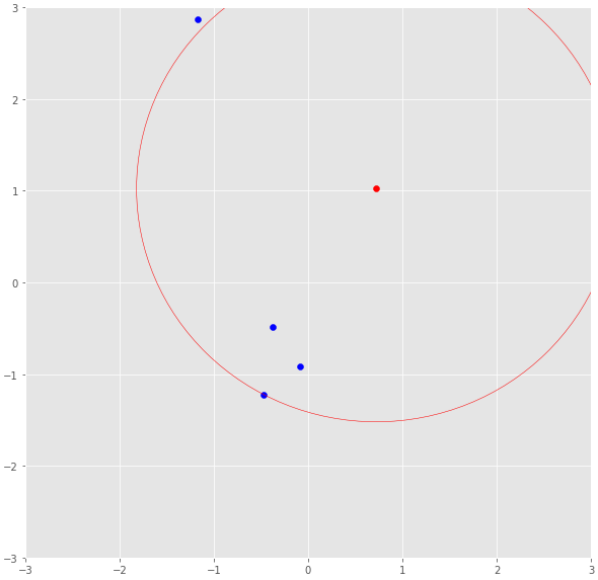

Figure 16. K-distance of a point with k=3 (Wenig, 2018)

- To get the *local reachability distance* for a point *a*, we will first calculate the reachability distance of *a* to all its k nearest neighbors and take the average of that number. The *lrd* is then simply the inverse of that average.

$$lrd(a) = \frac{k}{\sum_{n=0}^{k} reach\_dist(a,n)}$$

- Finally, the LOF is calculated as an average among the *lrd* of the point and the *lrd* of its k neighbours. If the density of a point is much smaller than the densities of its neighbours (LOF $\gg$1), the point is far from dense areas and, hence, an outlier.

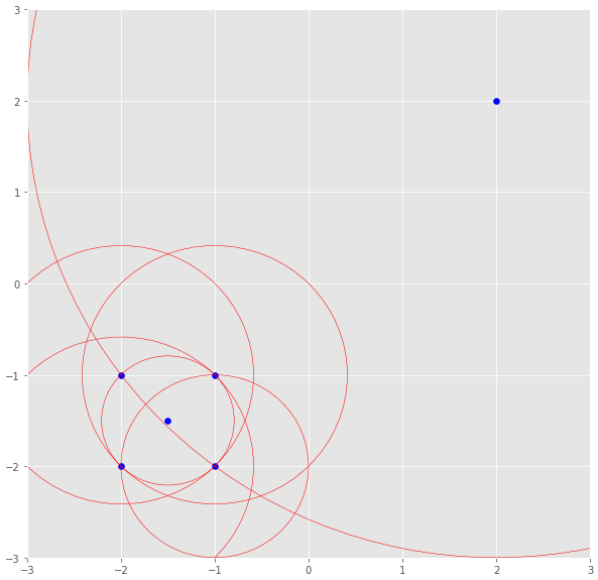

Figure 17. Outlier has big distance compared to the neighbors (k=4) (Wenig, 2018)

Code

```
import numpy as np
import pandas as pd
from sklearn import utils
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.neighbors import LocalOutlierFactor
from sklearn import metrics
import os, csv, argparse, socket, sys, time
from sklearn.externals import joblib

parser = argparse.ArgumentParser(description='Activate a Linear Classifier.')
parser.add_argument('-w','--warn', dest='warning', action='store_true',
                    help='activate warnings')
parser.add_argument('--chk_file', dest='chk_fp', action='store',
metavar='NAME',
                    default='occ_iso.model',help='used to specify checkpoint
model name')
parser.add_argument('-ow','--overwrite', dest='ow', action='store_true',
                    help='train again and overwrite the model')
parser.add_argument('-hn','--host', dest='host', action='store',
metavar='HOSTNAME',
```

```
                    default='127.0.0.1',help='server hostname (default:
127.0.0.1)')
parser.add_argument('-p','--port', dest='port', action='store',
metavar='PORT', type=int,
                    default=50001,help='server port number (default: 50001)')

args = parser.parse_args()


pd.options.display.max_rows = 10
pd.options.display.float_format = '{:.1f}'.format
columns = ["lake_dist",
     "purifier_dist",
     "clean_dist",
     "house_dist",
     "lake_pump",
     "purifier_pump",
     "house_pump",
     "purifier_temp",
     "clean_valve"]

show_warnings = args.warning

# Server Address
host = args.host
port = args.port

# Checkpoint folder
filepath = args.chk_fp
overwrite = args.ow

# import the CSV from Database.csv file
# this will return a pandas dataframe.
aqueduct_dataframe = pd.read_csv("Database.csv", sep=",", low_memory=False)

# let's take a look at the types of attack labels are present in the data.
aqueduct_dataframe["is_it_spam"].value_counts().plot(kind='bar')
plt.show()
```

```
# we're using a one-class SVM, so we need.. a single class. the dataset
'label'
# column contains multiple different categories of attacks, so to make use of
# this data in a one-class system we need to convert the attacks into
# class 1 (no_spam) and class -1 (spam)
aqueduct_dataframe.loc[aqueduct_dataframe['is_it_spam'] != 0, "is_it_spam"] =
-1
aqueduct_dataframe.loc[aqueduct_dataframe['is_it_spam'] == 0, "is_it_spam"] =
1


def preprocess_features(aqueduct_dataframe):
  """Prepares input features from personal data set.

  Args:
    aqueduct_dataframe: A Pandas DataFrame expected to contain data
      from the personal aqueduct data set.
  Returns:
    A DataFrame that contains the features to be used for the model, including
    synthetic features.
  """
  selected_features = aqueduct_dataframe[columns]
  processed_features = selected_features.copy()
  return processed_features

def preprocess_targets(aqueduct_dataframe):
  """Prepares target features (i.e., labels) from Aqueduct data set.

  Args:
    aqueduct_dataframe: A Pandas DataFrame expected to contain data
      from the California housing data set.
  Returns:
    A DataFrame that contains the target feature.
  """
  output_targets = pd.DataFrame()
  # Create a boolean categorical feature representing whether the
  # median_house_value is above a set threshold.
```

```
  output_targets["is_it_spam"] = aqueduct_dataframe["is_it_spam"]
  return output_targets

def new_train(new_record, prediction, timestamp):
  global features
  global target
  global model
  ts = timestamp
  df1 = preprocess_features(pd.DataFrame([new_record], columns=columns))[:1]
  df1.loc[0,"timestamp"] = timestamp
  df1.loc[0,"is_it_spam"] = prediction
  training_ex = preprocess_features(df1)
  features = features.append(training_ex, sort=True)
  training_target = preprocess_targets(df1)
  target = target.append(training_target, sort=True)

  if show_warnings:
    print(df1)
    print(prediction)

  model = model.fit(training_ex)

  spam_detected = 0
  if prediction == -1:
    spam_detected = 1
  new_record = new_record.tolist()
  new_record.append(spam_detected)
  ts.extend(new_record)
  print(ts)
  with open('Database.csv', 'a') as csvfile:
    filewriter = csv.writer(csvfile, delimiter=',', quotechar='|',
quoting=csv.QUOTE_NONE)
    filewriter.writerow(ts)
  joblib.dump(model, filepath, compress=9)

def insert_record (new_record):

  global model
```

```
  global aqueduct_dataframe

  test_input = preprocess_features(pd.DataFrame([new_record],
columns=columns))
  df = test_input[:1]

  pred = model.predict(df)
  # Print the prediction results.
  print("\nPrediction result: "+str(pred))

  # Add the prediction to the csv file and train on it
  # Create the timestamp
  ts = time.gmtime()
  row = [time.strftime("%Y-%m-%d %H:%M:%S", ts)]

  new_train(new_record, pred, row)
  if pred == 1:
    return "Data saved as NO SPAM\n"
  else:
    return "Data saved as SPAM\n"

# grab out the is_it_spam value as the target for training and testing. since
we're
# only selecting a single column from the `aqueduct_dataframe` dataframe,
we'll just get a
# series, not a new dataframe
target = preprocess_targets(aqueduct_dataframe)

# find the proportion of outliers we expect (aka where `is_it_spam == -1`).
because
# target is a series, we just compare against itself rather than a column.
outliers = target[target['is_it_spam'] == -1]
if show_warnings:
  print("outliers.shape", outliers.shape)
  print("outlier fraction", outliers.shape[0]/target.shape[0])

# drop label columns from the dataframe. we're doing this so we can do
# unsupervised training with unlabelled data. we've already copied the label
```

```
# out into the target series so we can compare against it later.
#aqueduct_dataframe.drop("timestamp", axis=1, inplace=True)
#aqueduct_dataframe.drop("is_it_spam", axis=1, inplace=True)
features = preprocess_features(aqueduct_dataframe)

# check the shape for sanity checking.
if show_warnings:
  print(features.shape)

#Divide train and test data
train_data, test_data, train_target, test_target = train_test_split(features,
target, train_size = 0.8)
#train_data = features
#train_target = target

if os.path.isfile(filepath) and not overwrite:
  model = joblib.load(filepath)
else:
  model = LocalOutlierFactor(n_neighbors=10, novelty=True, contamination=0.1)
  model.fit(train_data)

  joblib.dump(model, filepath, compress=9)

if show_warnings:
  preds = model.predict(train_data)
  targs = train_target


  with open('test_input.csv', 'r') as csvfile:
    readCSV = csv.reader(csvfile, delimiter=',')
    test_data = list(readCSV)
  new_cols = columns.copy()
  new_cols.append("is_it_spam")
  test = pd.DataFrame(test_data[1:], columns=new_cols)
  # Converting all objects to float and dropping incomplete rows
  test = test.replace(r'^\s*$', np.nan, regex=True).apply(lambda x:
pd.to_numeric(x, errors='coerce')).dropna()
  test.loc[test['is_it_spam'] != 0, "is_it_spam"] = -1
```

```
  test.loc[test['is_it_spam'] == 0, "is_it_spam"] = 1
  preds = model.predict(preprocess_features(test))
  targs = preprocess_targets(test)

  missing = set(targs['is_it_spam']) - set(preds)
  print("Validation:")
  if len(missing) > 0:
    print("Value "+str(missing)+" not present. Can not calculate metrics.")
  else:
    print("TARGS: ",np.asarray(targs["is_it_spam"], np.int))
    print("PREDS: ",preds)

    print("accuracy: ", metrics.accuracy_score(targs, preds))
    print("precision: ", metrics.precision_score(targs, preds))
    print("recall: ", metrics.recall_score(targs, preds))
    print("f1: ", metrics.f1_score(targs, preds))
    print("area under curve (auc): ", metrics.roc_auc_score(targs, preds))


# Start the server
s = socket.socket()
s.bind((host,port))
print("Server Started (CTRL+C to exit)")
s.listen(1)
try:
  while True:
    print("Waiting for incoming connections...")
    c, addr = s.accept()
    print("Connection from: " + str(addr))
    array = ''
    msg = ''
    while True:
      data = c.recv(1024).decode('utf-8')
      if not data:
        break
      array += data
    try:
      new_record = np.asarray(np.array(array.split(",")), np.int)
```

```
      print(new_record)
      if new_record.size != len(columns):
        msg = "Wrong # of values. Retry\n"
        print(msg)
      else:
        msg = insert_record(new_record)
    except ValueError:
      msg = "Incorrect Syntax. Retry\n"
      print(msg)
      c.send(msg.encode('utf-8'))
      c.shutdown(socket.SHUT_WR)
      c.close()
      continue
    c.send(msg.encode('utf-8'))
    c.shutdown(socket.SHUT_WR)
    c.close()
except KeyboardInterrupt:
  s.shutdown(socket.SHUT_RDWR)
  s.close()
  print ("\nServer shutdown correctly. Bye Bye")
```

Results

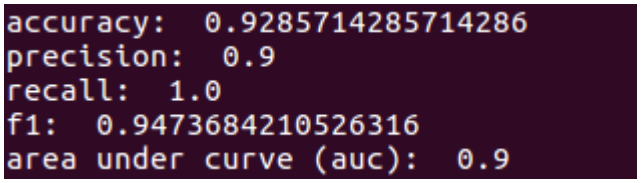

Figure 18. LOF Results

This algorithm scores incredibly well recognizing correctly all the examples except for one. The tuning of the parameters could be trivial, in particular for the *contamination* parameter which is used to set the decision threshold during the training phase. The *n_neighbors* specifies the k parameter already discussed and could be relevant for a proper work of the algorithm: too small and could be erroneous in a noisy environment, too big and it can miss local outliers.

Fourth Model: AutoEncoder

Despite models seen since this point, the following will be the first attempt to use a Neural Network which is an unsupervised approach inspired to the human body and considering the fast diffusion of these in all sort of environment will be worth to analyse this advanced kind of technology for our purpose.

Basics on Neural Networks

"An Artificial Neural Network (ANN) is an information processing paradigm that is inspired by the way biological nervous systems, such as the brain, process information. The key element of this paradigm is the novel structure of the information processing system. It is composed of a large number of highly interconnected processing elements (neurones) working in unison to solve specific problems. ANNs, like people, learn by example. An ANN is configured for a specific application, such as pattern recognition or data classification, through a learning process. Learning in biological systems involves adjustments to the synaptic connections that exist between the neurones. This is true of ANNs as well." (Stergiou & Siganos, n.d.)

Exactly as in biology, neurons fire or activate depending on some factors, for the Artificial Neural Networks that is based on a function called **activation function** which map the input to an output in a range proper of the function. This could be linear or non-linear but we only consider the second type because of the many problems of the first like the propagation of the linearity through the layers reducing all the network to a single-layer perceptron.

In the following a brief introduction to the most widely used activation functions is presented.

Sigmoid Function

Already presented in the Logistic Regression section **the Sigmoid function** is often used in model where the output is a probability because of the output included in the interval $[0,1]$.

$$A = \frac{1}{1+e^{-x}}$$

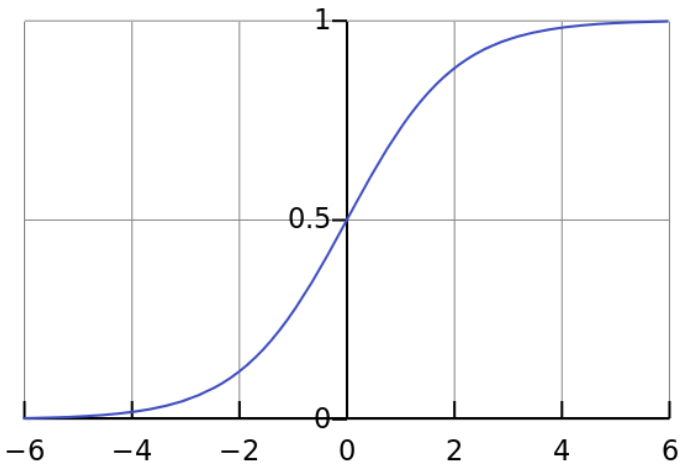

Figure 19. Sigmoid Function (Avinash, 2017)

Tanh Function

Another activation function that is vastly used is the **tanh function**.

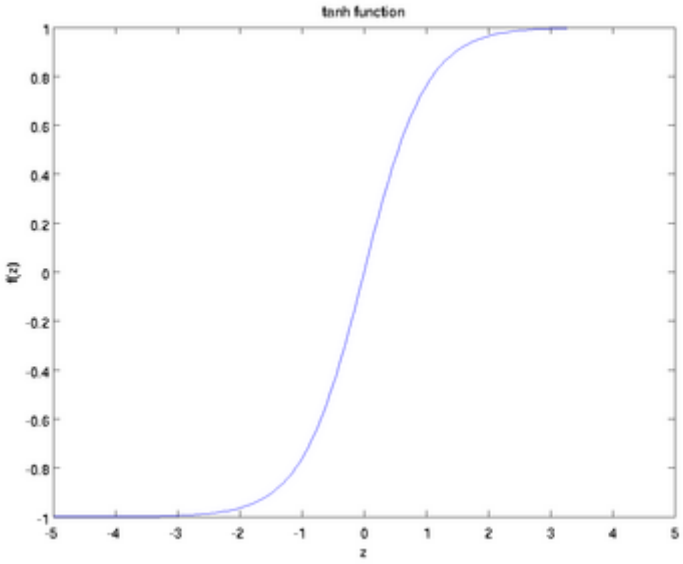

Figure 20. Tanh Function (Avinash, 2017)

$$f(x) = tanh(x) = \frac{2}{1+e^{-2x}} - 1$$

Hm. This looks very similar to sigmoid. In fact, it is a scaled sigmoid function!

$$tanh(x) = 2\, sigmoid(2x) - 1$$

Like sigmoid it is nonlinear in nature, so it resolves the problem of stacking layers. It is bound to range (-1, 1) so no worries of activations blowing up. One point to mention is that the gradient is stronger for tanh than sigmoid (derivatives are steeper).

Tanh is a common choice for binary classification, the fact it stands between -1 and 1 makes easier the classification identifying one class with negative values and the other with positive values.

ReLu

The following **rectified linear unit** activation function (or **ReLU**, for short) often works a little better than a smooth function like the sigmoid, while also being significantly easier to compute.

$$F(x) = max(0, x)$$

The superiority of ReLU is based on empirical findings, probably driven by ReLU having a more useful range of responsiveness. A sigmoid's responsiveness falls off relatively quickly on both sides.

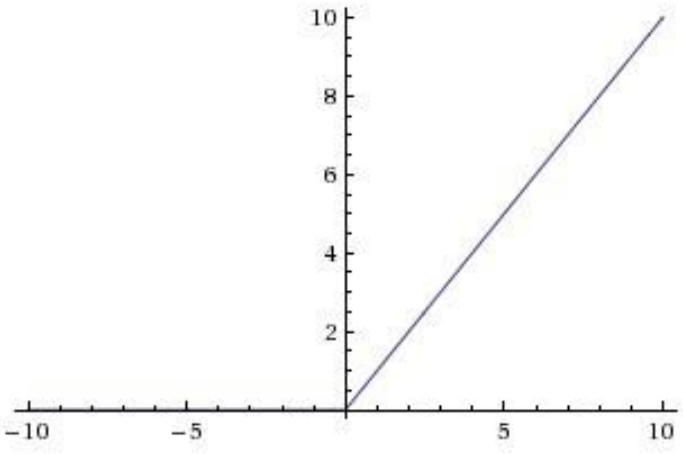

Figure 21. ReLU Function (Avinash, 2017)

AutoEncoder

The final attempt was to pass from a supervised model to an unsupervised one through an ANN, the most appropriate and widely accepted choice seems to be using an autoencoder.

An autoencoder is a neural network with the same number of inputs and output and one or more hidden layer between them with less neurons that will create a sort of bottleneck. This model is constituted by an Encoder with the purpose of including the same information in less data with a minimal loss and a Decoder with the purpose of reconstructing the input data from the compressed one with a minimal loss.

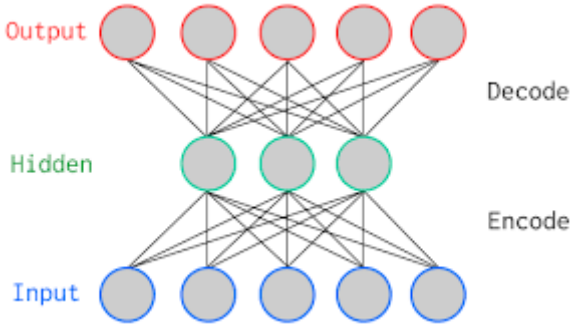

Figure 22. AutoEncoder Model

Training the autoencoder with just “ham” data will create a model minimizing the reconstruction error for that class. During inference the classification will be based on a threshold applied on the reconstruction error, a big value could be a symptom of a “spam” data, different from anyone seen during the training phase. Most of the time choose the right threshold is very trivial, especially in this case where we train on just one class and the reconstructed error of the other class is not known a priori.

In the proposed network both the encoder and the decoder are formed by two layers: the first one with a Tanh as activation function while the second one (hidden layer) with a ReLu. The

external layers have full dimension (equal to the number of features), the internal ones have half dimension.

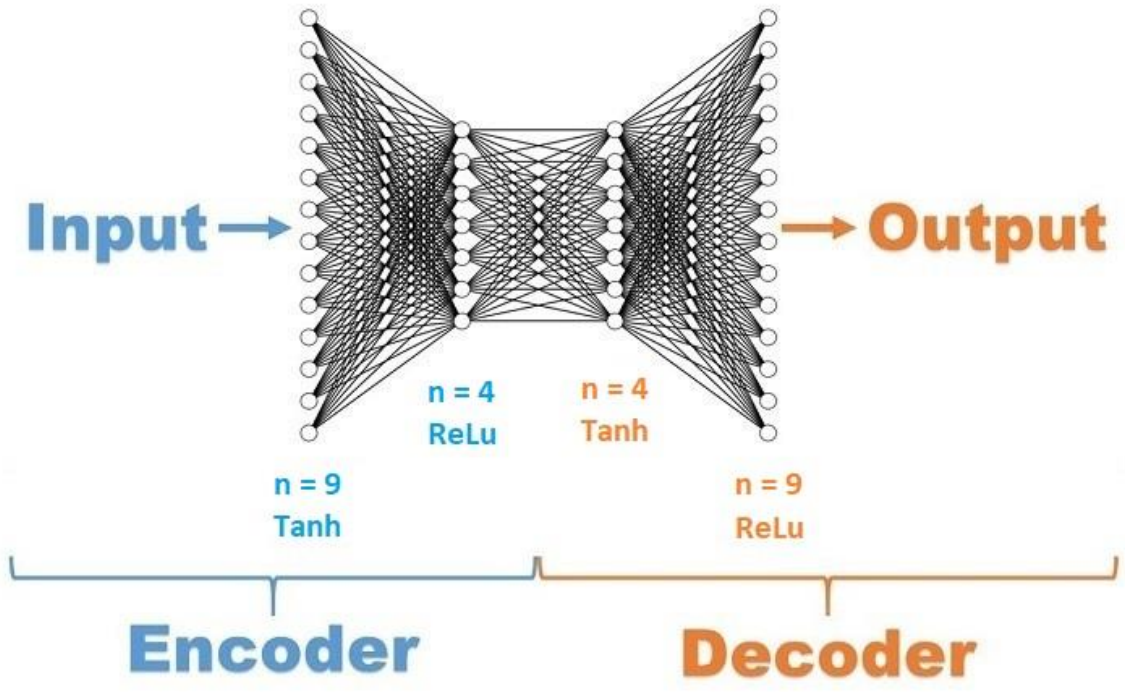

Figure 23. AutoEncoder internal structure (Ellison, 2018)

Code

```
# import packages
# matplotlib inline
import pandas as pd
import numpy as np
from scipy import stats
import tensorflow as tf
import matplotlib.pyplot as plt
import seaborn as sns
import pickle
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, precision_recall_curve
from sklearn.metrics import recall_score, classification_report, auc,
roc_curve
from sklearn.metrics import precision_recall_fscore_support, f1_score
from sklearn.preprocessing import StandardScaler
from pylab import rcParams
from keras.models import Model, load_model
from keras.layers import Input, Dense
from keras.callbacks import ModelCheckpoint, TensorBoard
```

```
from keras import regularizers
import keras.backend as K

from sklearn.preprocessing import MinMaxScaler
from sklearn.neighbors.kde import KernelDensity

#set random seed and percentage of test data
RANDOM_SEED = 314 #used to help randomly select the data points
TEST_PCT = 0.2 # 20% of the data

#set up graphic style in this case I am using the color scheme from xkcd.com
rcParams['figure.figsize'] = 14, 8.7 # Golden Mean
LABELS = ["Normal","Fraud"]
col_list = ["cerulean","scarlet"]# https://xkcd.com/color/rgb/
sns.set(style='white', font_scale=1.75, palette=sns.xkcd_palette(col_list),
color_codes=False)

df = pd.read_csv("Database.csv") #unzip and read in data downloaded to the
local directory
df.head(n=5) #just to check you imported the dataset properly

columns = ["lake_dist",
     "purifier_dist",
     "clean_dist",
     "house_dist",
     "lake_pump",
     "purifier_pump",
     "house_pump",
     "purifier_temp",
     "clean_valve"]

def preprocess_features(aqueduct_dataframe):
  """Prepares input features from personal data set.

  Args:
    aqueduct_dataframe: A Pandas DataFrame expected to contain data
      from the personal aqueduct data set.
  Returns:
```

```
    A DataFrame that contains the features to be used for the model, including
    synthetic features.
  """
  selected_features = aqueduct_dataframe[columns]
  processed_features = selected_features.copy()
  return processed_features

def preprocess_targets(aqueduct_dataframe):
  """Prepares target features (i.e., labels) from Aqueduct data set.

  Args:
    aqueduct_dataframe: A Pandas DataFrame expected to contain data
      from the California housing data set.
  Returns:
    A DataFrame that contains the target feature.
  """
  output_targets = pd.DataFrame()
  # Create a boolean categorical feature representing whether the
  # median_house_value is above a set threshold.
  output_targets["is_it_spam"] = aqueduct_dataframe["is_it_spam"]
  return output_targets

def print_metrics(y_true, y_pred):
    tot_spam = np.sum(np.array(y_true)==1, axis=0) #true neg
    indices = [i for i in range(len(y_pred)) if y_pred[i]==1] #predicted neg
    pos = len(y_pred)-len(indices) #predicted pos
    tn = len([i for i in indices if np.asarray(y_true, np.int)[i] == 1])
    fn = len(indices)-tn
    fp = tot_spam-tn
    tp = pos-fp
    acc = (tn+tp)/len(y_pred)
    precision = tp/(tp+fp)
    recall = tp/(tp+fn)
    f1_score = 2*tp/(2*tp+(len(indices)-tn)+fp)
    print("Accuracy: {0: .3f}".format(acc))
    print("Precision: {0: .3f}".format(precision))
    print("Recall: {0: .3f}".format(recall))
    print("F1-score: {0: .3f}".format(f1_score))
```

```
print(pd.value_counts(df['is_it_spam'], sort = True)) #class comparison
0=Normal 1=Fraud


#if you don't have an intuitive sense of how imbalanced these two classes are,
let's go visual
count_classes = pd.value_counts(df['is_it_spam'], sort = True)
count_classes.plot(kind = 'bar', rot=0)
plt.xticks(range(2), LABELS)
plt.title("Frequency by observation number")
plt.xlabel("Class")
plt.ylabel("Number of Observations");
plt.show()

normal_df = df[df.is_it_spam == 0] #save normal_df observations into a
separate df
fraud_df = df[df.is_it_spam == 1] #do the same for frauds
df_norm = df
df_norm = df_norm.drop(['timestamp'], axis=1)


train_x, test_x = train_test_split(df_norm, test_size=TEST_PCT,
random_state=RANDOM_SEED)
train_x = train_x[train_x.is_it_spam == 0] #where normal transactions
train_x = train_x.drop(['is_it_spam'], axis=1) #drop the class column


test_y = test_x['is_it_spam'] #save the class column for the test set
test_x = test_x.drop(['is_it_spam'], axis=1) #drop the class column

train_x = train_x.values #transform to ndarray
test_x = test_x.values

# Normalize Data (Experimental)
#scaler = MinMaxScaler()
#train_x_scaled = scaler.fit_transform(train_x)
#test_x_scaled  = scaler.transform(test_x)
```

```
train_x_scaled = train_x
test_x_scaled  = test_x


nb_epoch = 150
batch_size = 50   #128
input_dim = train_x.shape[1] #num of columns, 9
encoding_dim = 9
hidden_dim = int(encoding_dim / 2)+2 #i.e. 4
learning_rate = 1e-3

input_layer = Input(shape=(input_dim, ))
encoder = Dense(encoding_dim, activation="tanh",
activity_regularizer=regularizers.l1(learning_rate),
name="encoder")(input_layer)
encoder = Dense(hidden_dim, activation="relu")(encoder)
decoder = Dense(hidden_dim, activation='tanh')(encoder)
decoder = Dense(input_dim, activation='relu')(decoder)
autoencoder = Model(inputs=input_layer, outputs=decoder)

autoencoder.compile(metrics=['accuracy'],
                    loss='mean_squared_error',
                    optimizer='adam')

cp = ModelCheckpoint(filepath="autoencoder_fraud.h5",
                                save_best_only=True,
                                verbose=0)

tb = TensorBoard(log_dir='./logs',
                 histogram_freq=0,
                 write_graph=True,
                 write_images=True)

history = autoencoder.fit(train_x_scaled, train_x_scaled,
                    epochs=nb_epoch,
                    batch_size=batch_size,
                    shuffle=True,
                    validation_data=(test_x_scaled, test_x_scaled),
```

```
                    verbose=1,
                    callbacks=[cp, tb]).history


plt.plot(history['loss'], linewidth=2, label='Train')
plt.plot(history['val_loss'], linewidth=2, label='Test')
plt.legend(loc='upper right')
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
#plt.ylim(ymin=0.70,ymax=1)
plt.show()

pred = autoencoder.predict(test_x_scaled)

mse = np.mean(np.power(test_x_scaled - pred, 2), axis=1)
error_df = pd.DataFrame({'Reconstruction_error': mse,
                         'True_class': test_y})

threshold_fixed = 5 # default is 0
#step = 1
#requested_pool = 97 # es. 80%
#ex_true = 0 # examples gotten
#print(error_df['Reconstruction_error'])
#while ex_true < requested_pool:
#    threshold_fixed += step
#    trues = [1 for elem in error_df['Reconstruction_error'] if
elem<threshold_fixed]
#    ex_true = 100*sum(trues)/len(error_df['Reconstruction_error'])
print("Threshold: "+str(threshold_fixed))

groups = error_df.groupby('True_class')
fig, ax = plt.subplots()

for name, group in groups:
    ax.plot(group.index, group.Reconstruction_error, marker='o', ms=3.5,
linestyle='',
            label= "Fraud" if name == 1 else "Normal")
```

```
ax.hlines(threshold_fixed, ax.get_xlim()[0], ax.get_xlim()[1], colors="r",
zorder=100, label='Threshold')
ax.legend()
plt.title("Reconstruction error for different classes")
plt.ylabel("Reconstruction error")
plt.xlabel("Data point index")
plt.show();

validation_df = pd.read_csv("test_input.csv")
new_cols = columns.copy()
new_cols.append("is_it_spam")
test_input = pd.DataFrame(validation_df, columns=new_cols)
test_input = test_input.replace(r'^\s*$', np.nan, regex=True).apply(lambda x:
pd.to_numeric(x, errors='coerce')).dropna()

feat = preprocess_features(test_input)
pred = autoencoder.predict(feat)
targ = preprocess_targets(test_input)



mse = np.mean(np.power(feat - pred, 2), axis=1)
error_df = pd.DataFrame({'Reconstruction_error': mse,
                         'True_class': targ['is_it_spam']})
print(error_df.describe())

false_pos_rate, true_pos_rate, thresholds = roc_curve(error_df.True_class,
error_df.Reconstruction_error)
roc_auc = auc(false_pos_rate, true_pos_rate,)

plt.plot(false_pos_rate, true_pos_rate, linewidth=5, label='AUC = %0.3f'%
roc_auc)
plt.plot([0,1],[0,1], linewidth=5)

plt.xlim([-0.01, 1])
plt.ylim([0, 1.01])
plt.legend(loc='lower right')
plt.title('Receiver operating characteristic curve (ROC)')
plt.ylabel('True Positive Rate')
```

```
plt.xlabel('False Positive Rate')
plt.show()

precision_rt, recall_rt, threshold_rt =
precision_recall_curve(error_df.True_class, error_df.Reconstruction_error)
plt.plot(recall_rt, precision_rt, linewidth=5, label='Precision-Recall curve')
plt.title('Recall vs Precision')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.show()

plt.plot(threshold_rt, precision_rt[1:], label="Precision",linewidth=5)
plt.plot(threshold_rt, recall_rt[1:], label="Recall",linewidth=5)
plt.title('Precision and recall for different threshold values')
plt.xlabel('Threshold')
plt.ylabel('Precision/Recall')
plt.legend()
plt.show()


pred_y = [1 if e > threshold_fixed else 0 for e in
error_df.Reconstruction_error.values]
conf_matrix = confusion_matrix(error_df.True_class, pred_y)

plt.figure(figsize=(12, 12))
sns.heatmap(conf_matrix, xticklabels=LABELS, yticklabels=LABELS, annot=True,
fmt="d");
plt.title("Confusion matrix")
plt.ylabel('True class')
plt.xlabel('Predicted class')
plt.show()

print_metrics(error_df.True_class, pred_y)
```

Results

```
Accuracy:  0.857
Precision:  0.818
Recall:  1.000
F1-score:  0.900
```

Figure 24. AutoEncoder Results

The autoencoder has a performance nearly identical to the SVM on the test set measured by the F1 score. Unfortunately, it seems much more slower in training probably due to the encode/decode process and the consequently map of neural connections during each step.

Analysing the ROC we can notice that this model has an overall better performance for every possible threshold. That data, compared to the outcome shows some incongruity: due to the fact that the AUC tells an overall value created among all the threshold and in this model is so high that means that the chosen threshold was not an optimal one. This was proved by other runs, even if it results really hard to choose arbitrarily a threshold and apply it to every iteration, selecting it manually shows great outcomes near to the ones of the LOF. However, due to the unpredictability of the settings and the strong automatic component of the model we are searching for, the adoption of this algorithm is discouraged as too dependant of user supervision.

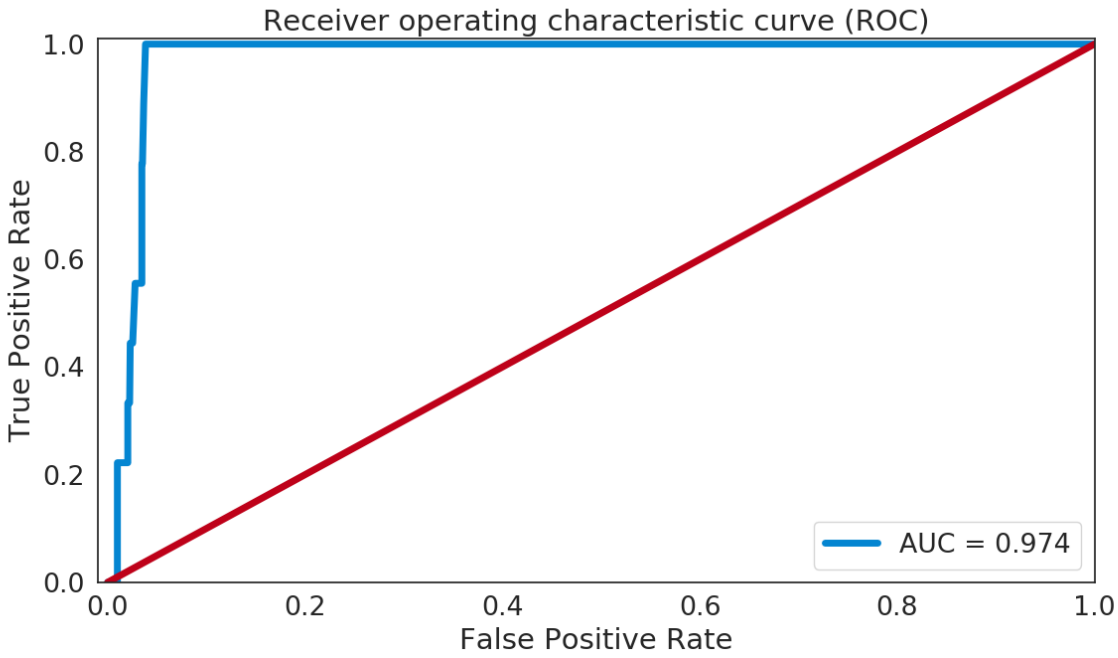

Figure 25. AutoEncoder AUC

Nevertheless, the autoencoder does not stands out of the crowd on average, not bringing useful advantages compared to the LOF model but opening the path to a new kind of models. Neural Networks, anyway, prove to be very adaptable to different situations and it's worth to better investigate with other models of this class searching for one which better fits our case.

Hybrid Autoencoder with KDE

After reading a paper entitled *"A Hybrid Autoencoder and Density Estimation Model for Anomaly Detection"* I decided to take a step further and compare this new model to the classical autoencoder to improve my results.

This system take advantage of an autoencoder during the train phase as previous but it uses its compressed data, right after encoding, applying on them a kernel density estimation (KDE).

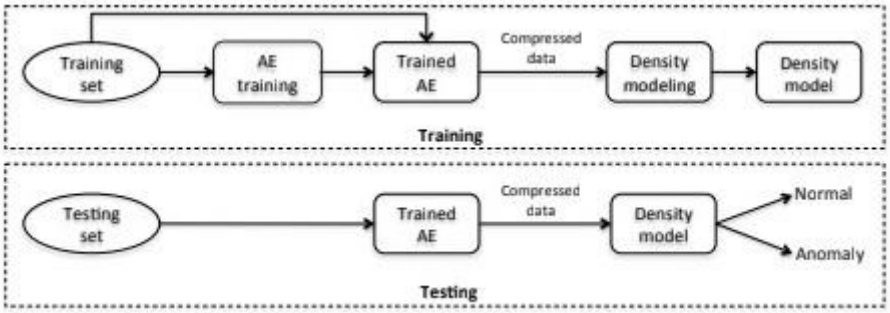

Figure 26. AEKDE Model

KDE is a non-parametric method of estimating probability density given a sample. Let $x1, x2, \ldots, xn$ be a set of d-dimensional samples in $\mathbb{R}^d$ drawn from an unknown distribution with density function $p(x)$. An estimate $\hat{p}(x)$ of the density at x can be calculated using

$$\hat{p}(x) = \frac{1}{n}\sum_{i=1}^{n} K_h\,(x - x_i)$$

where $K_h : \mathbb{R}^d \to \mathbb{R}$ is a kernel function with a parameter h called *bandwidth*.

The Gaussian kernel is common in applications and it is the one used in this model. As illustrated in Fig. 1(b) in KDE each point contributes a small "bump" to the overall density, with its shape controlled by the kernel and bandwidth. The bandwidth parameter h controls the trade-off between bias of the estimator and its variance.

$$K_h(x) = exp(-\frac{x^2}{2h^2})$$

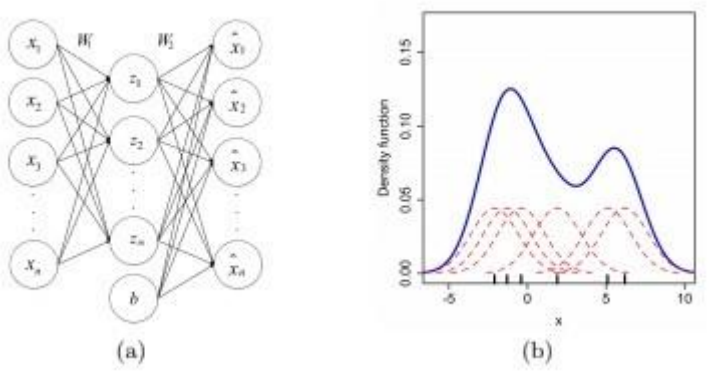

Figure 27. (a) An Autoencoder. (b) Density Estimated with KDE

KDE could be explained in a simpler way thinking at each value in input as a point on the x axis, for each occurrence a new point will be placed on the top of the other creating a graph that will look like this:

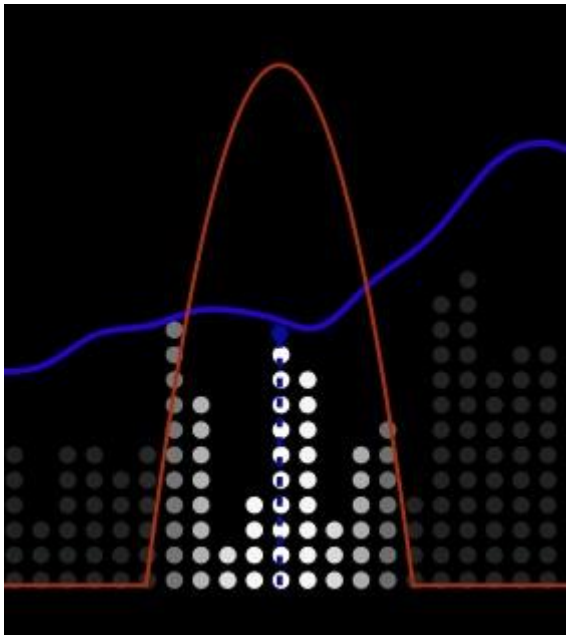

Figure 28. KDE

The blue dot represents the value of the Kernel Density function (blue line) for a specified input, this measure is defined applying a weight function to an interval of neighbours of the point. This function is represented by the red line, also called **kernel function**, which establish the contribute of every input to the current value. How much the dot influences the KDE is represented by its brightness, the more is lit the more will be the contribute to the function and that's why dots out of the interval of the kernel functions are completely dark.

The chosen type of kernel and bandwidth value influence the shape and the width/amplitude of the curve as it can be clear from the following figures.

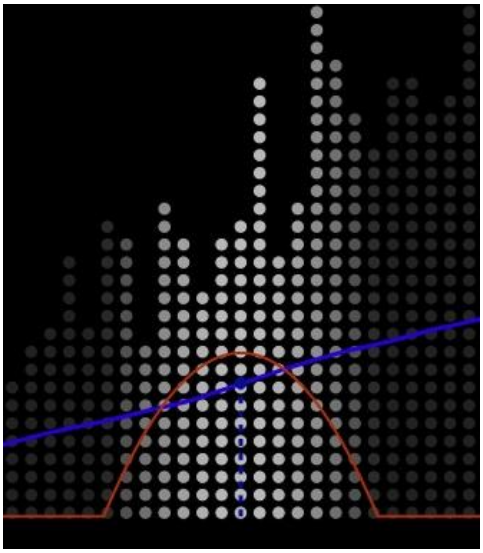

Figure 29. KDE with High Bandwidth

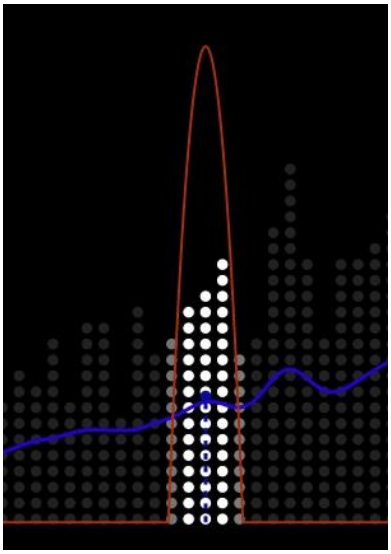

Figure 30. KDE with Low Bandwidth

Code

```
# import packages
# matplotlib inline
import pandas as pd
import numpy as np
from scipy import stats
import tensorflow as tf
import matplotlib.pyplot as plt
import seaborn as sns
import pickle
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, precision_recall_curve
from sklearn.metrics import recall_score, classification_report, auc,
roc_curve
from sklearn.metrics import precision_recall_fscore_support, f1_score
from sklearn.preprocessing import StandardScaler
from pylab import rcParams
from keras.models import Model, load_model
from keras.layers import Input, Dense
from keras.callbacks import ModelCheckpoint, TensorBoard
from keras import regularizers
```

```
import keras.backend as K
from sklearn.neighbors.kde import KernelDensity
import argparse
import socket
import time
import csv
import os

#set random seed and percentage of test data
RANDOM_SEED = 314 #used to help randomly select the data points
TEST_PCT = 0.2 # 20% of the data

#set up graphic style in this case I am using the color scheme from xkcd.com
rcParams['figure.figsize'] = 14, 8.7 # Golden Mean
LABELS = ["Normal","Fraud"]
col_list = ["cerulean","scarlet"]# https://xkcd.com/color/rgb/
sns.set(style='white', font_scale=1.75, palette=sns.xkcd_palette(col_list),
color_codes=False)

parser = argparse.ArgumentParser(description='Activate a Linear Classifier.')
parser.add_argument('-w','--warn', dest='warning', action='store_true',
                    help='activate warnings')
parser.add_argument('--ae_chk', dest='ae_chk_fp', action='store',
metavar='NAME',
                    default='autoencoder_fraud.h5',help='used to specify
autoencoder checkpoint name')
parser.add_argument('-ow','--overwrite', dest='ow', action='store_true',
                    help='train again and overwrite the model')
parser.add_argument('-hn','--host', dest='host', action='store',
metavar='HOSTNAME',
                    default='127.0.0.1',help='server hostname (default:
127.0.0.1)')
parser.add_argument('-p','--port', dest='port', action='store',
metavar='PORT', type=int,
                    default=50001,help='server port number (default: 50001)')

args = parser.parse_args()
```

```
pd.options.display.max_rows = 10
pd.options.display.float_format = '{:.1f}'.format
columns = ["lake_dist",
     "purifier_dist",
     "clean_dist",
     "house_dist",
     "lake_pump",
     "purifier_pump",
     "house_pump",
     "purifier_temp",
     "clean_valve"]
known_binary_cols = ["lake_pump","purifier_pump","house_pump","clean_valve"]

show_warnings = args.warning

# Server Address
host = args.host
port = args.port

# Checkpoint folder
ae_filepath = args.ae_chk_fp
overwrite = args.ow

df = pd.read_csv("Database.csv") #unzip and read in data downloaded to the
local directory

if show_warnings:
  print(pd.value_counts(df['is_it_spam'], sort = True)) #class comparison
0=Normal 1=Fraud

  #if you don't have an intuitive sense of how imbalanced these two classes
are, let's go visual
  count_classes = pd.value_counts(df['is_it_spam'], sort = True)
  count_classes.plot(kind = 'bar', rot=0)
  plt.xticks(range(2), LABELS)
  plt.title("Frequency by observation number")
  plt.xlabel("Class")
```

```
  plt.ylabel("Number of Observations");
  plt.show()


def preprocess_features(aqueduct_dataframe):
  """Prepares input features from personal data set.

  Args:
    aqueduct_dataframe: A Pandas DataFrame expected to contain data
      from the personal aqueduct data set.
  Returns:
    A DataFrame that contains the features to be used for the model, including
    synthetic features.
  """
  selected_features = aqueduct_dataframe[columns]
  processed_features = selected_features.copy()
  return processed_features

def preprocess_targets(aqueduct_dataframe):
  """Prepares target features (i.e., labels) from Aqueduct data set.

  Args:
    aqueduct_dataframe: A Pandas DataFrame expected to contain data
      from the California housing data set.
  Returns:
    A DataFrame that contains the target feature.
  """
  output_targets = pd.DataFrame()
  # Create a boolean categorical feature representing whether the
  # median_house_value is above a set threshold.
  output_targets["is_it_spam"] = aqueduct_dataframe["is_it_spam"]
  return output_targets

def binary_check(data):
  # Returns True if at least one binary constraint is violated
  for col in known_binary_cols:
    if data[col]!=0 and data[col]!=1:
      return True
```

```
  return False

def insert_record (new_record):

  global kde
  global cpencoder

  test_input = preprocess_features(pd.DataFrame([new_record],
columns=columns))
  data = test_input.squeeze()
  # Preliminary binary check
  if binary_check(data):
    pred = 1
  else:
    # Prediction using trained model
    log = kde.score_samples(cpencoder.predict(test_input))
    pred = int(log<0)
  # Print the prediction results.
  print("\nPrediction result: "+str(pred))

  # Add the prediction to the csv file and train on it
  # Create the timestamp
  ts = time.gmtime()
  row = [time.strftime("%Y-%m-%d %H:%M:%S", ts)]

  new_record = new_record.tolist()
  new_record.append(pred)
  row.extend(new_record)
  with open('Database.csv', 'a') as csvfile:
    filewriter = csv.writer(csvfile, delimiter=',', quotechar='|',
quoting=csv.QUOTE_NONE)
    filewriter.writerow(row)
  if pred == 1:
    return "Data saved as SPAM\n"
  else:
    return "Data saved as NO SPAM\n"

def print_metrics(y_true, y_pred):
```

```
    tot_spam = np.sum(np.array(y_true)==1, axis=0) #true neg
    indices = [i for i in range(len(y_pred)) if y_pred[i]==1] #predicted neg
    pos = len(y_pred)-len(indices) #predicted pos
    tn = len([i for i in indices if np.asarray(y_true, np.int)[i] == 1])
    fn = len(indices)-tn
    fp = tot_spam-tn
    tp = pos-fp
    acc = (tn+tp)/len(y_pred)
    precision = tp/(tp+fp)
    recall = tp/(tp+fn)
    f1_score = 2*tp/(2*tp+(len(indices)-tn)+fp)
    print("Accuracy: {0: .3f}".format(acc))
    print("Precision: {0: .3f}".format(precision))
    print("Recall: {0: .3f}".format(recall))
    print("F1-score: {0: .3f}".format(f1_score))

#data = df.drop(['Time'], axis=1) #if you think the var is unimportant
df_norm = df
#df_norm['Time'] =
StandardScaler().fit_transform(df_norm['Time'].values.reshape(-1, 1))
#df_norm['Amount'] =
StandardScaler().fit_transform(df_norm['Amount'].values.reshape(-1, 1))
df_norm = df_norm.drop(['timestamp'], axis=1)


train_x, test_x = train_test_split(df_norm, test_size=TEST_PCT,
random_state=RANDOM_SEED)
train_x = train_x[train_x.is_it_spam == 0] #where normal transactions
train_x = train_x.drop(['is_it_spam'], axis=1) #drop the class column


test_y = test_x['is_it_spam'] #save the class column for the test set
test_x = test_x.drop(['is_it_spam'], axis=1) #drop the class column

train_x = train_x.values #transform to ndarray
test_x = test_x.values

if not os.path.isfile(ae_filepath) or overwrite:
```

```
  # Training Specification
  nb_epoch = 200
  batch_size = 50   #128
  input_dim = train_x.shape[1] #num of columns, 9
  encoding_dim = input_dim
  hidden_dim = int(encoding_dim / 2)+1 #i.e. 4
  learning_rate = 1e-3

  # AutoEncoder Shape
  input_layer = Input(shape=(input_dim, ))
  encoder = Dense(encoding_dim, activation="tanh",
activity_regularizer=regularizers.l1(learning_rate))(input_layer)
  encoder = Dense(hidden_dim, activation="relu", name="encoder")(encoder)
  decoder = Dense(hidden_dim, activation='tanh')(encoder)
  decoder = Dense(input_dim, activation='relu')(decoder)
  autoencoder = Model(inputs=input_layer, outputs=decoder)

  autoencoder.compile(metrics=['accuracy'],
                      loss='mean_squared_error',
                      optimizer='adam')

  cp = ModelCheckpoint(filepath=ae_filepath,
                               save_best_only=True,
                               verbose=0)

  tb = TensorBoard(log_dir='./logs',
                  histogram_freq=0,
                  write_graph=True,
                  write_images=True)

  history = autoencoder.fit(train_x, train_x,
                      epochs=nb_epoch,
                      batch_size=batch_size,
                      shuffle=True,
                      validation_data=(test_x, test_x),
                      verbose=1,
                      callbacks=[cp, tb]).history
```

```
else:
  autoencoder = load_model(ae_filepath)

cpautoencoder = autoencoder

cpencoder = Model(inputs=cpautoencoder.input,
                          outputs=cpautoencoder.get_layer('encoder').output)

encoded_train_preds = cpencoder.predict(train_x)

kde = KernelDensity(kernel='gaussian', bandwidth=0.01)
kde.fit(encoded_train_preds)

validation_df = pd.read_csv("test_input.csv")
new_cols = columns.copy()
new_cols.append("is_it_spam")
test_input = pd.DataFrame(validation_df, columns=new_cols)
test_input = test_input.replace(r'^\s*$', np.nan, regex=True).apply(lambda x:
pd.to_numeric(x, errors='coerce')).dropna()
feat = preprocess_features(test_input)
encoded_preds = cpencoder.predict(feat)
log = kde.score_samples(encoded_preds)
if show_warnings:
  print("Kernel Density Estimation:")
  print(np.exp(log))
  plt.bar(np.arange(1,len(log)+1),np.exp(log),align='center')
  plt.title("Kernel Density Estimation")
  plt.xlabel("Test Case")
  plt.xticks(np.arange(1,len(log)+1))
  plt.show()
pred = np.asarray((log<0), np.int)
i = 0
for index,row in feat.iterrows():
    if binary_check(row):
        pred[i] = 1
    i+=1
targ = preprocess_targets(test_input)
true = np.asarray(targ['is_it_spam'], np.int)
```

```
print("Pred: ",pred)
print("TRUTH: ",true)

#pred_y = [1 if den<0 else 0 for den in logs]
if show_warnings:
    print_metrics(true, pred)

# Start the server
s = socket.socket()
s.bind((host,port))
print("Server Started (CTRL+C to exit)")
s.listen(1)
try:
  while True:
    print("Waiting for incoming connections...")
    c, addr = s.accept()
    print("Connection from: " + str(addr))
    array = ''
    msg = ''
    while True:
      data = c.recv(1024).decode('utf-8')
      if not data:
        break
      array += data
    try:
      new_record = np.asarray(np.array(array.split(",")), np.int)
      print(new_record)
      if new_record.size != len(columns):
        msg = "Wrong # of values. Retry\n"
        print(msg)
      else:
        msg = insert_record(new_record)
    except ValueError:
      msg = "Incorrect Syntax. Retry\n"
      print(msg)
      c.send(msg.encode('utf-8'))
      c.shutdown(socket.SHUT_WR)
      c.close()
```

```
            continue
        c.send(msg.encode('utf-8'))
        c.shutdown(socket.SHUT_WR)
        c.close()
except KeyboardInterrupt:
    s.shutdown(socket.SHUT_RDWR)
    s.close()
    print ("\nServer shutdown correctly. Bye Bye")
```

Results

```
Accuracy:  1.000
Precision:  1.000
Recall:  1.000
F1-score:  1.000
```

Figure 31. AEKDE Results

This model has the best possible results predicting all the test data correctly. The decision threshold was set to 0 on the $log(kde\_prediction)$, marking all the data with density between 0 and e as "spam" ($log(kde\_prediction) \leq 0 \; \forall kde\_prediction \in [0, e]$). Using a low-value bandwidth the following chart shows the test density estimated:

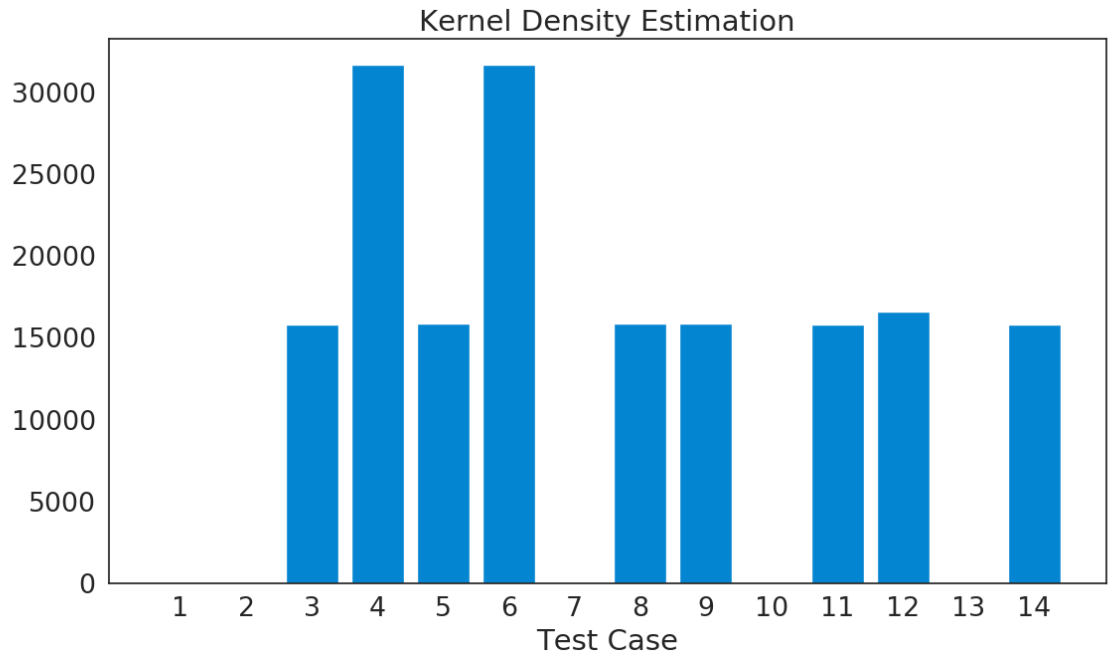

Figure 32. Density Estimation on Test Set

Compared to the simple autoencoder there is a great divergence between ham and spam data respect to the classification metric facilitating the threshold choice.

Conclusions

Due to the supervised nature of this algorithm, which grants it a labelled pool of input, the SVM was naturally seen as the most efficient algorithm among the ones taken in consideration. Surprisingly the best recorded results are the ones of the hybrid model between an Autoencoder and a Kernel Density Estimator, which is a Neural Network, therefore, provided with unsupervised learning.

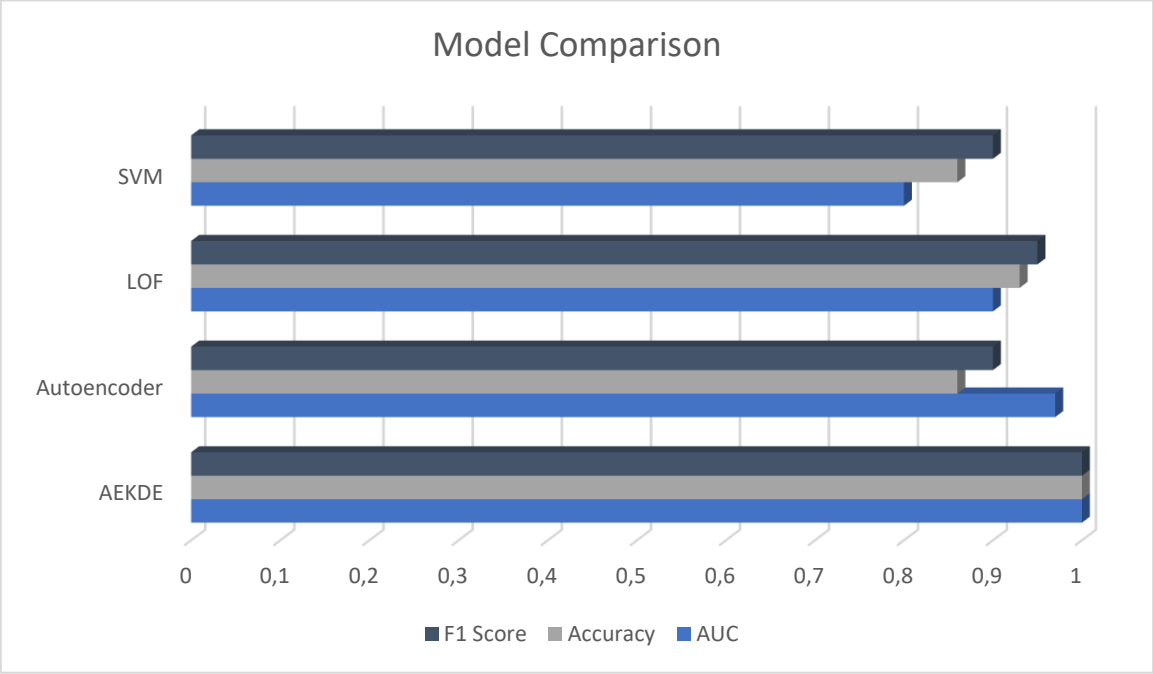

Figure 33. Model Comparison

As shown in the previous graph the AEKDE obtains the best result for each metric registering perfection for this study case. To score the others we give precedence to F1 score, as previously described, then we consider accuracy and in the end the AUC. The LOF takes the second place both for the overall scores registered and for the speed necessary for training which should be considered anyway. It is essential to concentrate for a moment on the autoencoder, that seems to have lowest performance but it is important to remember that this depends on the threshold choice and as the AUC shows it has potential to have slightly better results. Anyway, the autoencoder has been declassed due to the nature, too variable, of the threshold choice.

The victory of an unsupervised learning algorithm over supervised ones, as much astonishing it can appear, could make sense due to the nature of the initial dataset and its belonging to the same class. In fact, even if unsupervised learning appears to be more incomplete because of its lack of labels, in this circumstance the label can be taken for granted due to the nature of OCC problems flattening the differences between these classes.

Much more than that could be essential the method used to create the test set which is composed by randomly chosen data of the training set with variations on some of them to create “spam/anomalies”.

Using this approach rises some interrogatives on the model, as for example “is the model good or it is really strict so that each result not yet encountered is considered a threat?”. Unfortunately, at the moment real malicious data in the simulated environment is not yet generated but it could in a near future confirm our thesis.

Maintaining a high grade of generalization should grant our system the ability to adapt to nearly any context where there is a small number of features and a binary One-Class Classification. The increasing use of machine learning in automation and in a wide range of smart objects could be the perfect background to use the algorithm for anomaly detection.

Table of Figures

References

Avinash, S. V. (2017, March 30). *Understanding Activation Functions in Neural Networks*. Retrieved from https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0

Ellison, D. (2018, August 8). *Fraud Detection Using Autoencoders in Keras with a TensorFlow Backend*. Retrieved from https://www.datascience.com/blog/fraud-detection-with-tensorflow

Gandhi, R. (2018, June 7). *Support Vector Machine — Introduction to Machine Learning Algorithms*. Retrieved from https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47

Google. (n.d.). *Machine Learning Crash Course*. Retrieved from https://developers.google.com/machine-learning/crash-course/

Ray, S. (2017, September 13). *Understanding Support Vector Machine algorithm from examples*. Retrieved from https://www.analyticsvidhya.com/blog/2017/09/understaing-support-vector-machine-example-code/

Stergiou, C., & Siganos, D. (n.d.). *Neural Networks*. Retrieved from https://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol4/cs11/report.html

Wikipedia, the free enciclopedia. (n.d.). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Machine_learning

Van Loi Cao, Nicolau M., McDermott J. (2016) *A Hybrid Autoencoder and Density Estimation Model for Anomaly Detection*

Ringraziamenti

Dopo più di 3 anni a Torino sento il bisogno di ringraziare le persone che sono stati importanti per me e che mi hanno aiutato a raggiungere questo traguardo che segna un punto di partenza per una nuova fase della mia vita.

Ringrazio prima di tutto il professor Prinetto e Giuseppe che mi hanno aiutato nella realizzazione di questa tesi, cercando di supportarmi al meglio con grande premura e interesse.

Ovviamente questa esperienza torinese non sarebbe stata possibile senza la mia famiglia che ha deciso di assecondare questa mia scelta dandomi pieno supporto andando incontro ai sacrifici che ne sono conseguiti. Grazie per la fiducia riposta in me, è stato importante.

Voglio poi passare a tutte le persone che ho incontrato a Torino, partendo dal mio primo coinquilino Renato che poi mi ha fatto conoscere tutto il resto della mia nuova “famiglia” torinese, grazie ragazzi per tutti i momenti passati insieme. Senza di voi sarebbe stata insostenibile, quelli fra di voi che studiano al Politecnico lo sapranno.

Ovviamente voglio ringraziare tutti quelli conosciuti fra le pareti dell’università e fuori, che con me hanno condiviso i momenti più disparati fra i banchi di scuola e chi come Teuz e Dado ne hanno creati anche al di fuori.

Ringrazio gli amici di una vita che purtroppo erano lontani ma che ogni volta che li rivedevo mi facevano sentire di nuovo a casa e dimenticare delle ansie e frustrazioni, per voi le parole non servono tanto lo sapete già.

Infine a chiunque non si sia sentito citato fra nessuno di questi, vi ringrazio perché anche se non me la sento di citarvi uno ad uno mi avete accompagnato per un periodo piccolo o grande che fosse in questo percorso che mi ha condotto in questo punto.

Per concludere grazie a tutti! Grazie grazie grazie!