Thesis

August 15, 2020

Enriched Camera Monitoring System

With Computer Vision and Machine Learning

UNIVERSITY of LIMERICK

OLLSCOIL LUIMNIGH

Adam Napora (Student 18197892)

15 July 2020

Abstract

The objective of this research is to find the most useful algorithms in a Camera Monitoring System, which facilitate robust data collection pipeline, ability to make predictions for future events, and

identification of anomalous situations.

There are countless papers and articles about the low level components, like Computer Vision, Machine Learning and Anomaly Detection algorithms, however it is very challenging to find a coherent study about a reliable, scalable and generalizable end-to-end process to link all these techniques in a Real Time Video Stream scenario.

Following the data flow, this study is composed of three main domains:

- data collection
- predictions for future events
- anomaly detection

Even though the final results gathered as part of this study are not perfect, they show that low cost IOT (Internet Of Things) devices, like Raspberry Pi, can be utilized for reliable data collection, probabilistic Machine Learning models can be developed to predict future events, and Deep Learning and probabilistic methods can be deployed to find anomalous signals in the image data.

The recommendation for the reader is to review a Table of Contents below, an Introduction and the Conclusion Chapters, and then dive into the Methodology chapters (3,4,5,6).

Often code snippets are easier to relate to than mathematical, statistical, or computer science terminology, and therefore there is a suite of accompanying *Extra Notebooks* with well documented Python source code, which are referenced through out the text.

Table of Contents

Keywords

Computer Vision, Deep Learning, Supervised Machine Learning, Video Streaming, Unsupervised Machine Learning, Internet of Things (IOT), Neural Networks, Probabilistic Programming, Anomaly Detection, Forecasting, Data Processing, Message Queues, Jupyter Notebooks, Poisson Distributions, Python, Exploratory Data Analysis, Feature Selection.

1. Introduction

index | next

1.1. Background

Computers and Vision have been already linked together since the sixties.

In 1963, Larry Roberts in his Ph.D. (Roberts 1963) mentions that the *pictorial data* understanding by the machines has been a challenge for quite a while.

Since then, research in the area had its ups and downs, but the most recent major break-through, and what is currently seen as the beginning of the modern era, can be credited to the paper by Alex Krizhevsky: *ImageNet Classification with Deep Convolutional Neural Networks* (Krizhevsky 2012).

From year 2012 onwards there has been an exponential progress in the area of Object Recognition and Detection. Hand crafted feature detectors, like SIFT (Lowe, 2004), and prediction algorithms, like SVM (Vapnik et al., 1995), have been challenged by automated feature detection and prediction capabilities offered by Convolutional Neural Networks.

Modern Computer Vision and prediction algorithms can be leveraged by using flexible and abstracted libraries, online articles, books and video tutorials. Seeing this progress would be like something from an alien civilization for Larry Roberts.

These "intelligent software" ideas have been accompanied by the substantial innovation in the hardware space and data availability. Starting from usage of GPU (Graphical Processing Unit) to significantly boost matrix computations, doubling CPU clocks every year, and a spike in highly affordable small form factor IOT devices (like Raspberry Pi), it is now possible to efficiently operate on large volumes of image data to solve interesting problems.

1.2. This Research

My research uses modern Computer Vision, Machine Learning and hardware to create a Camera Monitoring System, capable of showing a live video stream with real time object detection and recognition.

The questions I will try to answer are: - How complex is it to build a fast and reliable object detection pipeline using *Computer Vision*? - Given collected data with object detections, can future object counts be predicted using *Machine Learning*?

- Does object detections data contain anomalous signals, which can be recognized with *Anomaly Detection* algorithms and used for alerts to the users?

If the research goals are achieved, then the final product should be generic enough to apply it in other households and to other use cases, like predicting traffic, tourist congestion or animal behaviour, and finding unusual events or even security threats from the video stream.

From a data protection and security perspective, the aim is to keep the data local, which means that the internet connection should not be required and data breach is much less probable.

And lastly, I would like this thesis to be distributed as an open source project, so anyone curious can see how all these pieces are glued together, make their own improvements and build their own datasets and algorithms.

1.3. Limitations

Like any software in the real world, the system I am proposing here has its limitations.

The type of the camera used in the process of video capture is very basic. The default Raspberry Pi camera (PiCam), does not have the Night Vision capability, which somewhat limits its usage as a security device. However, according to FBI, and as reported by many home alarm companies in the online sources (alarmnewengland 2020), most of burglaries occur between 10AM and 3PM, when most of adults are at work or school.

The next limitation is the prediction accuracy. Due to heavily stochastic nature of the world around us and potential gaps in data collection process, it is not flawless. But the main objective of this research is less about accuracy and metrics, but more about usefulness.

1.4. Guidelines for reader

Even though this dissertation has been exported to a pdf file, it has been written in Jupyter Notebooks.

This means that the project can be cloned from GitHub, code samples can be executed and plots reproduced.

Here are more guidelines to make reading a more enjoyable experience:

- Chapters contain links to the previous and next chapters on top and in the bottom of each Notebook
- Key highlights will be highlighted with a yellow background (only in in Jupyter format)
- Important concepts, areas or terminology will be written as *italic*
- Some chapters provide a reference to an in-depth study (called *Extras*) with well documented code samples, additional commentary and plots
- Chapters are structured as a hierarchy with maximum two levels of depth (for example 6. -> 6.1. -> 6.1.1.)
- There are often clickable links to create a better flow
- All mathematical notations are written in LaTex
- Each reference to a code or function will be formated like `this_function`
- Some paragraphs will be divided by a title in **bold font** to improve text spacing

Next Chapter contains a Literature Review, which is a study of theoretical framework related to this research.

index | next

2. Literature Review

index | prev | next

This chapter provides a theoretical foundation for the key ideas and algorithms, which can be found inside this research.

Note: Code samples used to generate included plots can be found in the corresponding Extra Notebook 1

2.1. Data Collection

Data Collection involves a mini-computer (Raspberry Pi) which streams the data to the central unit (a Ubuntu-based Desktop PC with a GPU), which runs an infinite loop with the two key algorithms: - Background subtraction - Yolo Object Recognition

Both of these algorithms are extremely useful in the image processing applications, and they represent a foundation of how data is collected in the system.

2.1.1. Motion Detection

Considering how objects of interest could be detected in a 30 frames per second video stream, a naive approach would be to send all frames into an object detector.

In theory this could work, but it would be extremely inefficient and resource intensive. Can a significant change in a series of images be detected first, and only then object detector utilized in the process?

Here is an example of a static background:

Fig. 2.1. Static background

And now a moving object (a person running) in a 7 consecutive frames:

Fig. 2.2. Moving object

It turns out that there is already a set well established algorithms in the Computer Vision domain designed precisely for that purpose. They are not 100% bulletproof, but they do not need to be. If they can help to reject over 90% of static frames, GPU can be allocated to other tasks or simply kept idle to extend its lifespan. The bonus of not abusing GPU is a quiet machine and less heat generation.

One of the most popular and successful methods for motion detection in images is the *Background Subtraction*.

At a very high level the concept is very simple: there is a starting point with a static image without any moving objects (called *background* - BG). Then, every consecutive frame will be compared against the background to detect any changes in the *foreground* - FG.

Unfortunately, there are many challenges in this optimistic approach: - the initial background might already contain moving objects - next frames actually do not contain any moving objects, but only changes in light illumination - shadows appear and disappear - camera is in-door and light is turned on and off - tree branches move in the background - changing weather conditions (rain, snow, hail) - small objects constantly show in the camera lens

These not so rare anomalies demand a more sophisticated approach than just a simple subtraction of foreground from the background.

One very popular improvement over the vanilla algorithm has been proposed in the *Improved Adaptive Gaussian Mixture Model for Background Subtraction* paper (Zivkivoc 2004). The aim of Zivkivoc's work was to overcome some of the challenges above and achieve efficiency by reducing the processing time.

In the *MOG2* model, the background is constantly updated and not static. As author describes it, it uses recursive equations to constantly update parameters and also select appropriate number of components per each pixel. At a high level author describes a metric R (using a Bayesian decision), which follows the formula:

$$R = \frac{p(\text{BG}|\mathbf{x}^{(t)})}{p(\text{FG}|\mathbf{x}^{(t)})} = \frac{p(\mathbf{x}^{(t)}|\text{BG})p(\text{BG})}{p(\mathbf{x}^{(t)}|\text{FG})p(\text{FG})}$$

, where the aim is to determine the ratio between the probability of new pixel at time t being a foreground or a background.

In general prior information about FG is unknown, it is a uniform distribution. Then, a decision is made if object is a BG if the probability of x at time t, given BG is greater than some threshold value (c_{thr}):

$$p(\mathbf{x}^{(t)}|BG) > (= Rc_{FG})$$

The left side of the equation is referred to as a background model. It depends on the training set denoted as X.

In order to eliminate the problem of suddenly changing lighting factor, authors proposed to keep updating the training set by dropping old values, appending new ones and re-estimating the background model (using Gaussian mixture model with M components):

$$\hat{p}(\mathbf{x}|X_T, BG + FG) = \sum_{m=1}^{M} \hat{\pi}_m \mathcal{N}(\mathbf{x}; \hat{\boldsymbol{\mu}}_m, \hat{\sigma}_m^2 I)$$

Where means and variances, which describe the Gaussian components are added. Covariance matrices are diagonal and identity matrix has proper dimensions. The weights are non-negative and add up to 1.

For each new data samples, equations are updated recursively, as follows:

$$\hat{\pi}_m \leftarrow \hat{\pi}_m + \alpha(o_m^{(t)} - \hat{\pi}_m)$$

$$\hat{\boldsymbol{\mu}}_m \leftarrow \hat{\boldsymbol{\mu}}_m + o_m^{(t)}(\alpha/\hat{\pi}_m)\boldsymbol{\delta}_m$$

$$\hat{\sigma}_m^2 \leftarrow \hat{\sigma}_m^2 + o_m^{(t)}(\alpha/\hat{\pi}_m)(\boldsymbol{\delta}_m^T\boldsymbol{\delta}_m - \hat{\sigma}_m^2)$$

There is an introduction of an α - alpha parameter here, which is exponentially decaying, meaning that the older data samples will be given less importance:

$$\alpha = 1/T$$

In the new sample the $o_m^{(t)}$ value is set to 1 for the component with a largest weight and 0 in other components.

The following formula denotes the squared distance from m-th component:

$$\boldsymbol{\delta}_m^T \boldsymbol{\delta}_m / \hat{\sigma}_m^2$$

If the maximum number of components is reached, the component with the lowest weight is removed. Hence the algorithm has been defined by the author to be an "online clustering algorithm".

Author also describes how model deals with foreground objects, which remain static for a longer duration of time: for the FG object to be considered a BG, it needs to be static for approximately some number of frames:

$$log(1 - c_f)/log(1 - \alpha)$$

c_f stands for the maximum portion of data, which belongs to FG objects without influencing the BG model. For sample values for c_f and α, author has calculated 105 frames for the FG object to be considered a BG.

Weights define the underlying multinomial distribution. After additional derivations, author rewrites the first equation to the following form (this is after including the *Dirichlet* prior for multinomial distribution):

$$\hat{\pi}_m \leftarrow \hat{\pi}_m + \alpha(o_m^{(t)} - \hat{\pi}_m) - \alpha c_T$$

Although the algorithm is very accurate, it has some trade offs and prerequisites need to be met: - frame needs to be stationary - it is highly recommended to resize images (in theory, this is not a strong requirement, but this model is very slow when used with the 1080p or even 720p image resolution)

The OpenCV implementation for Python can be found under `cv2.createBackgroundSubtractorMOG2` function, which I have used to detect motion between consecutive image frames.

The parameters used for detection will be explained in the later Chapter: Data Collection.

2.1.2. Object Detection

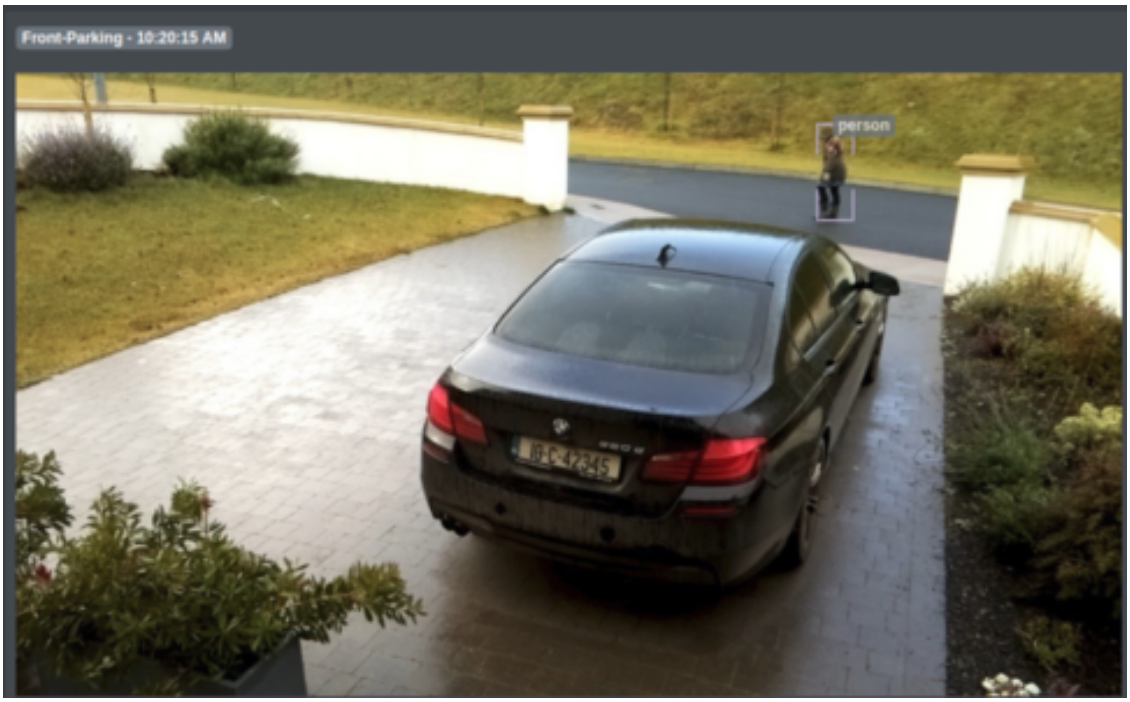
Once motion is detected, resized frames can be sent to an Object Detector to analyze the content of an image.

Object detection comes from two fundamental ideas in Computer Vision: - Image Classification (look at an image and classify a single class: Car, Person, Dog etc.) - Object Localization (where an object is located inside an image)

Object Detection's task is to classify multiple objects in an image and tell their locations.

Here is an example screen-shot from a web application, which I have built for this purpose. It detected motion and a single *Person* object in a real time camera stream:

Fig. 2.3. Real time detection in Web App



In *Fig. 2.3* there is a person walking by in front of the parking area. There is a purple rectangle around the person, called the *bounding box*. If there were more people in the frame, they would be contained in their own bounding boxes too.

Object Detection is a dynamically evolving field with new algorithms and applications developed at high frequency. This is an advantage, as it creates new opportunities, but it is also a challenge for practitioners to keep up with the fast pace of change.

Out of two arguably most popular options for object detection in Python: *Yolo* and *SSD*, I have decided to use Yolo (You Only Look Once) due to the fact that it can be run in real time on a GPU at 30+ frames per second with good online documentation and wide-spread adoption rate. In comparison - I have not found a GPU implementation for the SSD (Single Shot Detector) algorithm, and the amount of web based knowledge about it is lacking.

One could attribute Yolo's popularity to its catchy name, but it is nevertheless a very useful and fast algorithm for object detection.

Yolo V1:

Even though in my research I have used Yolo Version 2, the section below is dedicated to Yolo version 1, as it is the foundation for object detection, which utilizes multiple Computer Vision and Deep Learning paradigms.

Yolo v1 (Redmon et al., 2015) has been released in 2015 as a new approach to object detection, which promised extreme speed and making real time object detection a reality. This was a significant achievement in comparison to previous object detectors, like R-CNN (Girshick et al. 2013) where a single image could take 20 seconds to get processed or even in comparison to more modern Fast R-CNN (Girshick 2015), which still took 2 seconds to process a single image and Faster R-CNN (Ren et al., 2015) with 0.14 second per frame.

The processing time can be extremely important for certain applications, like self driving car or camera monitoring system. According to Yolo authors, algorithm can run at 45 frames per second with a slight decrease in accuracy for smaller objects.

To visualize progress in the area, I have put together a table below:

Detector	FPS
R-CNN	0.05
Fast R-CNN	0.5
Faster R-CNN	7
Yolo v1	45

Yolo owes the gain in speed to a complete re-think of how the object detectors can operate. Unlike a traditional approach, like sliding window with HOG (Histogram of Oriented Gradients), SVM (Support Vector Machine) and region proposals (seen in R-CNN's), Yolo uses a single pass through the entire image to generate predictions (with the sliding window it might have been event thousands passes through a single image of different size),

Then, to get rid of overlapping bounding boxes, the ones with very low probability are discarded, and Non-max suppression (Hosang et al., 2017) algorithm is applied.

This approach generates an output of 1470 features, containing all the data needed to understand the content of an image. Assuming 20 object classes, the calculation is as follows:

$$7 \times 7 \times (2 \times 5 + 20) = 7 \times 7 \times 30\text{tensor} = 1470\text{features}$$

, where 7×7 is related to a grid size, which image is divided by, 2×5 stands for 2 bounding boxes inside each grid cell, and 20 is a number of predicted classes in a One Hot Encoded notation.

Each grid cell predicts two boxes and can only have a single class.

Below is the name and description for each of the 5 nodes found in each bounding box: - $Conf$ - confidence - x - x coordinate of center of the box (relative to grid cell) - y - y coordinate - w - width of the box (relative to whole image) - h - height

The confidence if object is present in the grid is calculated as: $Pr(\text{Object}) \cdot \text{IOU}_{\text{pred}}^{\text{truth}}$

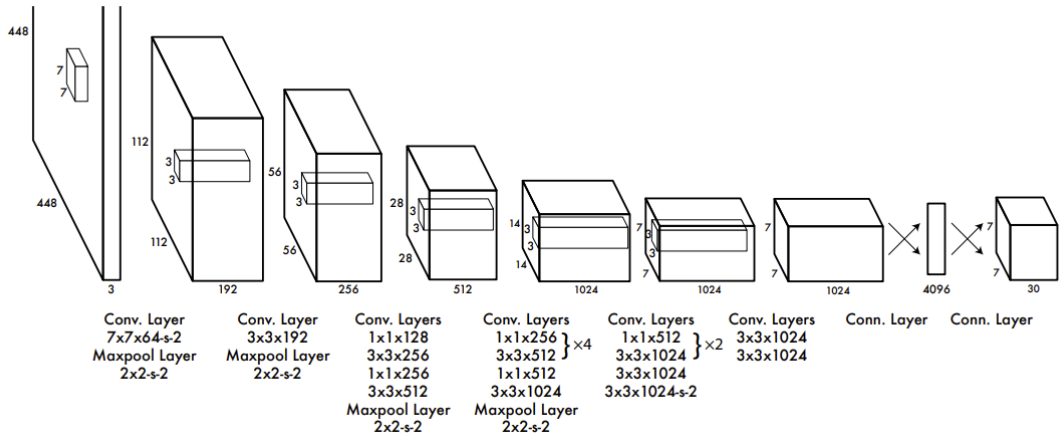
, where IOU represents an Intersection Over Union, an evaluation metric for bounding boxes:

$$\text{IOU} = \frac{\text{areaOfOverlap}}{\text{areaOfUnion}}$$

If there is no object present in the cell, then the confidence will be zero, but if there is an object, it will equal to the *IOU* metric.

For training, a following Convolutional Neural network is used:

Fig. 2.4. Yolo v1 architecture



, with 24 convolutional layers followed by 2 fully connected layers. Convolutional layers are pre-trained on ImageNet classification (with 1000 classes), and final output is, as expected, a $7 \times 7 \times 30$ tensor.

This type of complex network was trained for a week using the freely available *Darknet* framework, which is also used for the real time inference in this research.

The network is trained with 224×224 image resolution and then extended to 448×448 at detection stage.

As activation function, authors have used a leaky rectified linear activation (Leaky ReLU):

$$f(x) = \begin{cases} x, & \text{if } x > 0 \\ 0.1x, & \text{otherwise} \end{cases}$$

The loss function to optimize is a highly customized SumSquared Error with the specific characteristics: - to avoid issues with gradients for cells without any objects - to distinguish errors in large boxes versus small boxes (errors in large boxes matter less)

Here are other significant training parameters: - epochs: 135 - batch size: 64 - momentum: 0.9 - decay: 0.0005 - custom learning rate schedule

To prevent the overfit, dropout layers are introduced, and to increase image variability data augmentation is used.

Overall, Yolo is a good trade-off between speed and accuracy and at present is one of the most useful tools, which is free of cost and provides plethora of online documentation and examples.

Yolo V2:

An upgraded version of Yolo (V2) was released in 2016 as a state of the art real time object detector capable of detecting over 9000 object categories.

Capable of achieving a 76.8 mAP (mean average precision) on VOC 2007 (The PASCAL Visual Object Classes Challenge 2007) while maintaining 40 FPS (frames per second), which, as authors conclude, outperforms the other two most populate object detectors: *SSD* and *Faster RCNN with ResNet*.

The main improvements in Yolo v2 have been achieved through a number of core ideas: - Batch Normalization: Added to all convolutional layers to stabilize training, speed up convergence and add regularization - High Resolution Classifier: End to end fully trained on 448×448 image resolution, so more details can be detected - Convolutional With Anchor Boxes: Diving an image into N-overlapping boxes of WxH size - helpful to detect smaller objects, like multiple people faces - Dimension Clusters: Instead of hand picked anchor box dimensions, Yolo V2 uses k-means clustering with a custom distance metric $d(box, centroid) = 1 - IOU(box, centroid)$ - Direct location prediction: Increase model stability during early training iterations by introducing logistic activation to constrain network predictions or coordinates relative to the location of the cell grid - Multi-Scale Training: Aim is to make the model robust to varied image resolutions, which is achieved by randomly choosing a new image dimension every 10 batches during the training (size must be divisible by 32, from 320×320 to 608×608. When Yolo is run at 288×288, it achieves much better performance, which might be useful for multiple video streams (for example one camera inside and two outside the house)

The architecture is composed of 19 convolutional layers and 5 max-pooling layers. To process an image 5.58 billion operations is required. This might seem very high, but it is much lower in comparison to a very popular choice for feature extractor VGG-16 (Simonyan et al. 2014), which require 30.69 billion floating point operations. Yolo V1 required 8.52 billion, as it was based on the Googlenet architecture (Shegedy et al. 2014).

Yolo V2 uses its own classification model called *Darknet-19*, which is trained for classification and for detection using slightly different architecture and hyper-parameters from V1 and similar data augmentation techniques to V1.

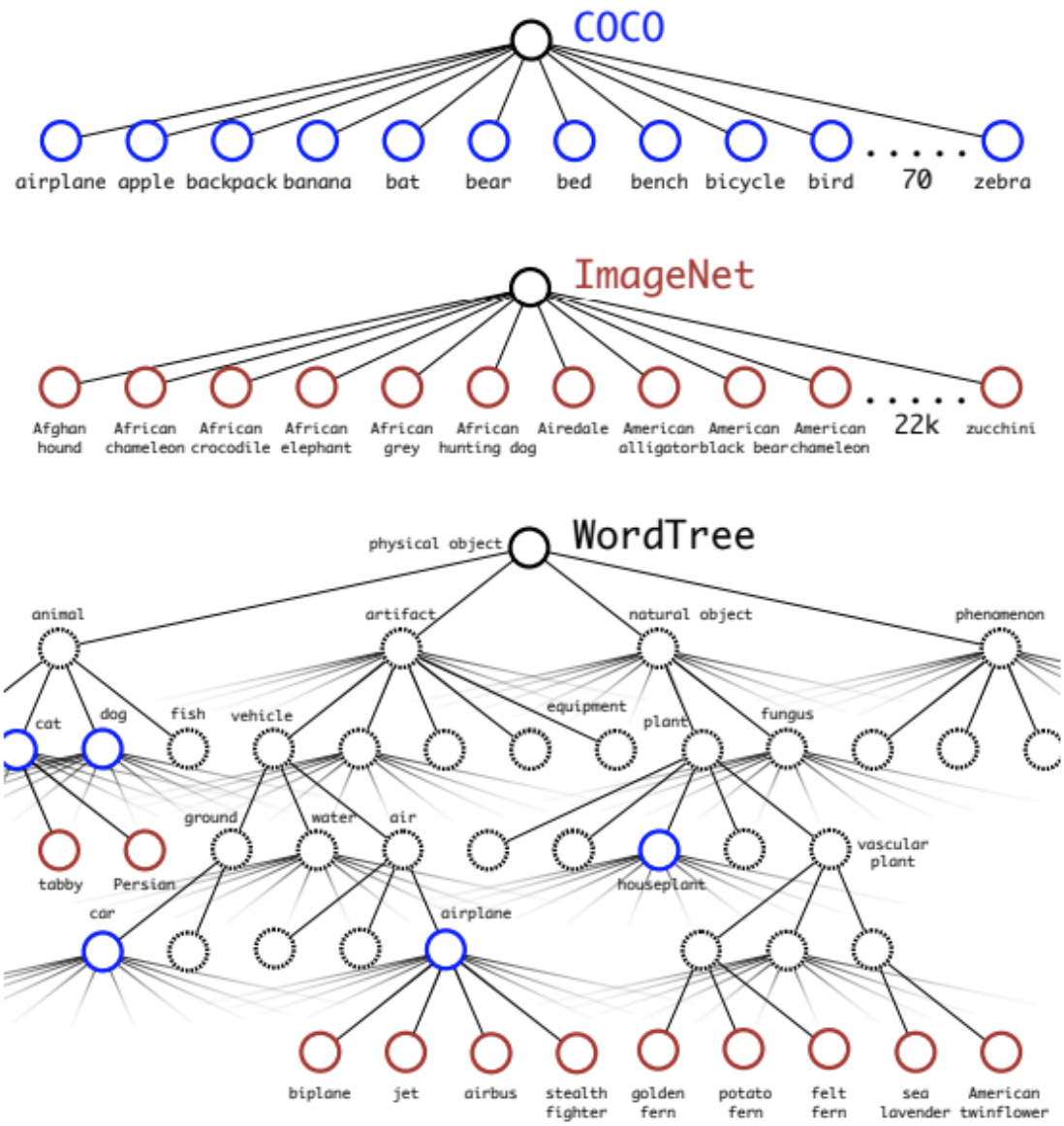
The *classification model* uses ImageNet dataset with 1000 very fine-grained classes (like "Norfolk terrier") and *detection model* uses COCO dataset with 80 high-level class names (like "dog").

Since authors wanted to jointly train on classification and detection data, a *hierarchical classification* was used, where the final soft-max layer with the flat encoding of mutually exclusive labels is not assumed. This has been achieved using an approach borrowed from Natural Language Processing called *WordTree*, where for each lower level class, a probability is calculated if this class belongs to a broader category, example:

$$Pr(Norfolk - terrier|terrier)$$

This model can been visualized as the following tree diagram:

Fig. 2.5. Yolo V2 WordTree



Yolo V3, V4:

Since 2018 there have been already two iterations for Yolo object detector: - version 3 (Redmon et al. 2018) - version 4 (Bochkovskiy et al. 2020)

They both look very promising in terms of further accuracy and performance boosts, but I am leaving this work for the future increments of this project.

2.1.3. Conclusion

Yolo V2 represents a good trade off between accuracy and performance, and has proven to work well in case of detecting people and vehicles from the Raspberry Pi camera frames, which I discuss

further in the Data Collection chapter.

2.2. Forecasting

In advance of discussions about Machine Learning models, I would like to point out a fundamental concept: *Bias and Variance trade-off*.

We say that a model is good if it fits the training and testing data well. Models like Linear Regression create a straight line through the data points, and often do not represent the relationships very well. This is called *High Bias*. On the other hand, learners like Decision Trees model relationships in the training data very well, but tend to perform poorly on the testing sets. We call this behaviour *High Variance*. Ideally we always look for a model with relatively low bias and low variance. In practice, it is a matter of finding a good trade-off.

Although the above statement tends to hold for non-Neural Network models, it does not always apply to Neural Networks, which can generalize well, even with their complexity and High Variance (Neal et al., 2018).

The Forecasting Chapter includes a study on a single *Naive* and three *Machine Learning* algorithms, which are compared next in the Literature Review: - Standard Decision Tree - Gradient Boosting Decision Tree - Gaussian Process

As a side note, I have also explored many other algorithms, like: - Linear Regression - Random Forest - Multiple variations of Feed Forward Neural Network - Long Short Term Memory Recurrent Neural Network (LSTM)

Training so many Machine Learning models was a very valuable experience, but the additional algorithms listed above were not beneficial to the research results.

2.2.1. Decision Tree Regressor

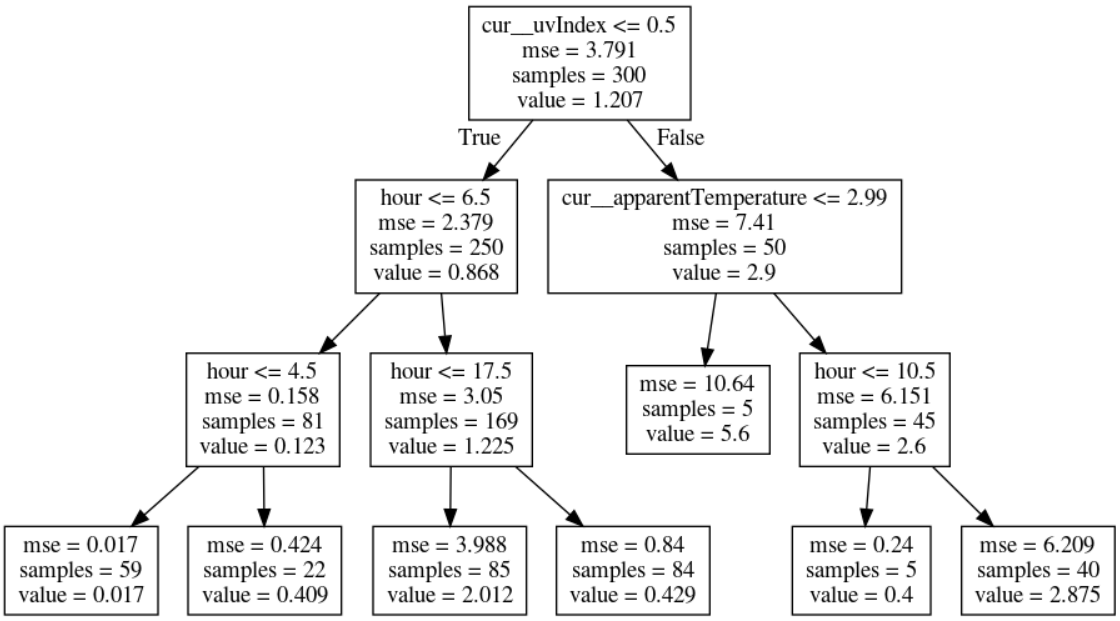
Decision Trees are a building block for many more sophisticated Machine Learning algorithms. Their simplicity and interpretability make them a very popular choice, when decisions must be clearly understood and explained.

The history of Decision Trees used for regression problems is not very easy to track, and goes back to a research by J.N. Morgan (Morgan et al., 1963, p. 430) titled *Problems in the Analysis of Survey Data, and a Proposal*, where most likely first decision tree for regression was drawn. What has been a challenge back then (computational overhead) is actually not an issue in 2020, which makes Decision Trees one of the fastest Machine Learners available for relatively condensed datasets.

The problem with decision trees is their low accuracy on an out-of-sample datasets and low robustness (small changes in training data may easily lead to a very different tree). It is however useful to discuss their inner workings before moving on to the more advanced algorithms.

Here is an example output from a Tree Regressor algorithm run on a small subset of 100 observations (for illustrative purpose) from the dataset with *People* detections:

Fig. 2.6. Decision Tree



This tree can be interpreted as a series of sub-decisions to reach a decision goal. Following the *right hand side path*, the model predicts a value of 2.875 by making the following decisions: - `uvIndex` is greater than 0.5 - `temperature` is greater than 2.99 - `hour` is greater than 10

It intuitively makes sense, as during the day, when temperature is not very low and after 10AM, expectation of approximately 3 objects is correct.

Below is the basic terminology related to the hierarchy above: - the single box on top of the diagram is called a *root node* - nodes in the middle are called the *decision nodes* and are connected by arrows creating a section called a *branch* - the eight boxes in the bottom are *leaf nodes*

The best split for Regression Trees is usually calculated using *mean squared error*, however other metrics (like *mean absolute error*) can be utilized as well.

The top-down procedure to generate a tree is the same for each node: - iterate through candidate features - for each feature: - sort values - find average between each pair of values and use as a candidate split value - calculate average for values the left and right nodes - calculate squared residuals for each node - sum all residuals or average those - split the data by the feature and value, which produces the lowest squared error - keep doing this until: - reached maximum depth of a tree allowed - there is not enough samples to create a split - all samples contain the same value - leaf nodes will eventually contain an average value for the target variable

Hyper-parameters `max_depth` and `min_samples_split` are used as a regularization term to avoid creating a model with *too high variance*.

2.2.2. Gradient Boosting Regressor Tree

There are many extensions to the base Decision Tree algorithm. I have found the Gradient Boosting Regressor to perform the best with the object detections dataset.

The *Histogram Based Gradient Boosting Regressor* is a relatively new estimator added in 2019 into the Sci-Kit Learn library, developed in *Cython* to optimize speed. It is very efficient and capable of handling large datasets and missing values.

The implementation in sklearn was inspired by the 2017 paper by Guolin Ke et al. (Ke et al., 2017), *LightGBM: A Highly Efficient Gradient Boosting Decision Tree* and it is a modern take on the original Gradient Boosting Machine algorithm by Jerome Friedman (Friedman, 1999).

The original algorithm developed by Friedman puts robustness as one of the most favorable characteristics. The paper also mentions that the TreeBoost removes the need for feature transformations, and chooses only important features, while ignoring irrelevant input variables. It handles missing data, and enhances stability through the use of many small trees (instead of a single large one). Author admits that single Decision Tree is easier to interpret than many (even hundreds) small trees, however when one tree grows to a very large scale, this conclusion does not hold any more.

Friedman's algorithm can be described in a sequence of steps:

- Start with a dataset consisting of input features and a target variable $\{(x_j, y_i)\}_{i=1}^{n}$ and a Loss Function, which is *differentiable* $L(y, F(x))$ (like *least squares*)
- Initialize a model with a constant value: $F_0(x) = \text{argmin}_\gamma \sum_{i=1}^{n} L(y_i, \gamma)$, solve this equation find the initial predictions by taking the derivatives for the losses for each target, summing them up and setting the sum to 0. In the Decision Tree terminology, this step creates a leaf, which predicts the initial values
- Next section is an iteration for $m = 1$ to M which produces M small trees (where M is a hyper-parameter to tune), and contains following steps
  - Compute $r_{im} = -[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}]_{F(x} = F_{m-1}(x)$ for $i = 1, ..., n$
  - Fit regression tree to the r_{im} values and create terminal regions R_{jm} for $j = 1...J_m$
  - For $j = 1, ..., J_m$ compute $\gamma_{jm} = \text{argmin}_\gamma \sum_{x_i \in R_{ij}} L(y_j, F_{m-1}(x_i) + \gamma)$
  - Update $F_m(x) = F_{m-1}(x) + \nu \sum_{j=1}^{J_m} \gamma_m I(x \in R_{jm})$, where ν is another hyper-parameter to tune, the learning rate
- The iteration stops when either all steps were exhausted or when there is no significant change in the errors

The key idea in the *Histogram Based Gradient Boosting* is related to the way Decision Trees find the best value to split the data. Where vanilla Gradient Boosting algorithm sorts the values and then for each pair of values runs a test if the split is optimal, histogram based Tree avoids the computational problem for larger datasets by binning the values into (typically) 256 bins and use integer-based data structures (histograms). This way expensive sorting and testing all continuous floating point values is avoided.

One of the most recent improvements in this algorithm in `sklearn` is a *Poisson* loss function, which is more suitable when data is believed to come from a Poisson distribution (adequate for count data):

$$L(y, \hat{y}) = \frac{1}{N} \sum_{i=0}^{N} (\hat{y}_i - y_i \log \hat{y}_i)$$

, where $\hat{y}$ is the predicted expected value and y is the ground truth value.

Poisson is a discrete probability distribution, which expresses the probability of a given number of events occurring in a fixed interval of time or space if these events occur with a known constant mean rate and independently of the time since the last event wiki-link

Probability mass function of X for $k = 0, 1, 2, 3, ...$ is given by:

$$f(k; \lambda) = Pr(X = k) = \frac{\lambda^k e^{-\lambda}}{k!})$$

, where $\lambda > 0$, *expected value* and *variance* are both equal to λ, e is Euler's number ($e = 2.718...$) and $k!$ is the factorial of k.

Minimizing the Poisson loss is equivalent of maximizing the likelihood of the data under the assumption that the target comes from a Poisson distribution, conditioned on the input (peltarion.com, Poisson loss).

This feature is utilized in the Forecasting Notebook.

2.2.3. Gaussian Process

Gaussian Distribution, also known as Normal Distribution is the cornerstone of statistical learning.

Its origins go back to the theory with Abrhama de Moivre (1667-1754) and Carl Friedrich Gauss (1777-1855), and it is a fundamental concept used to model real-valued, random and continuous variables, which can be observed vastly in the nature, social studies, mathematics and engineering.

The Central Limit Theorem (Laplace 1810) and unique analytical properties make Gaussian distributions a very useful tool, which can be also applied to Machine Learning.

The literature review below takes a gradual approach in order to understand the Gaussian Processes (peterroelants blog): - Univariate Gaussian Distribution - Multivariate Gaussian Distribution - Gaussian Process

It is interesting for this research as Gaussian Process can be used to make future predictions given the historical data, and it can also generate an uncertainty about these predictions, which allows for more informed decisions.

Univariate Gaussian

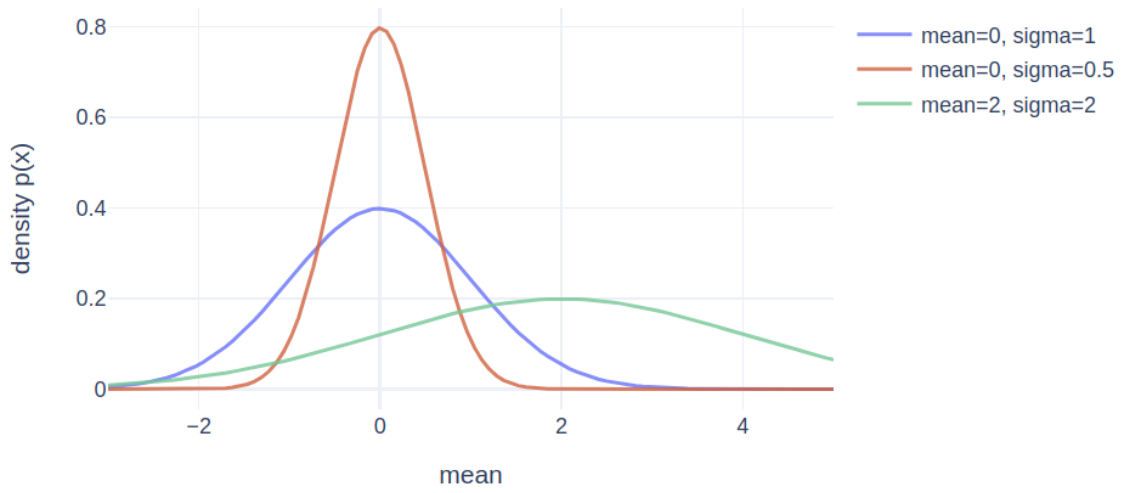
Gaussian distribution is given by:$\mathcal{N}(\mu, \sigma^2)$, where μ is the expected value of a distribution, and σ corresponds to a standard deviation from μ. Sigma squared (σ^2) is also known as a variance.

The *pdf* (probability density function) for a normal distribution is given by:

$$p(x|\mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} exp(-\frac{(x-\mu)^2}{2\sigma^2})$$

The word "univariate" relates to a single random variable (x) in the equation above. Several values for μ and σ can be plotted to observe the familiar *bell curves*:

Fig. 2.7. Univariate Gaussian



Multivariate Gaussian

Multivariate normal distribution is used for analysis of multiple random variables (for example x_1 and x_2). Similarly to the univariate case, it is defined by a two parameters: - mean vector μ - covariance matrix Σ, which measures how correlated each pair of variables is

The equation below describes a join probability for the multivariate normal with d variables (i.e. the dimension of the dataset):

$$p(x|\mu, \Sigma) = \frac{1}{\sqrt{(2\pi)^2|\Sigma|}} exp(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu))$$

, where x is this time a vector of values (of size d), Σ is the symmetric and positive definite covariance matrix (of size dxd), and $|\Sigma|$ is its determinant.

As a shorthand, we use $\mathcal{N}(\mu, \Sigma)$ to denote this distribution.

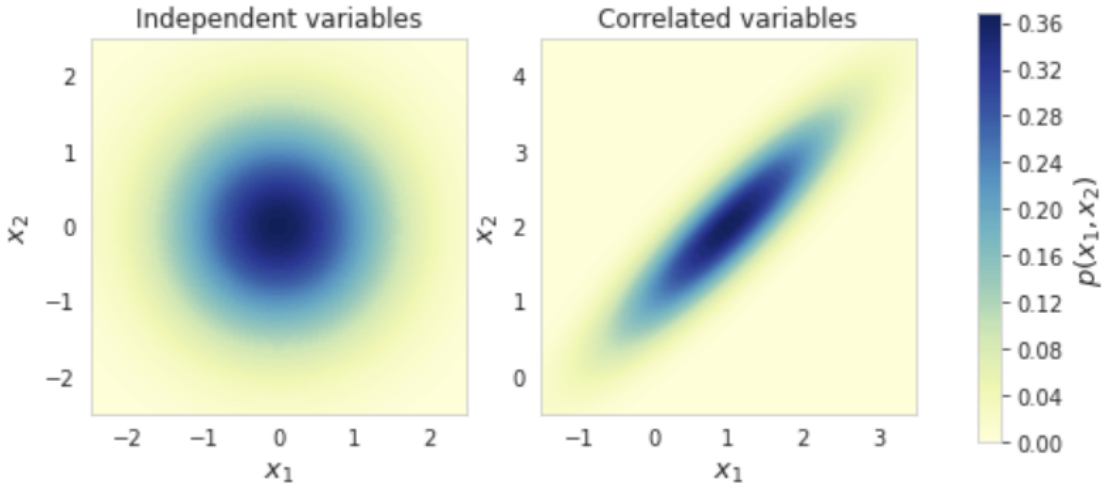
Below are a two examples of multivariate Gaussian distribution: - first example shows 2 uncorrelated variables. Change in x_1 does not mean a change in x_2 ($0, 0$ diagonals in Σ):

$$\mathcal{N}\left(\begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}\right)$$

- second example shows 2 highly correlated variables. When x_1 increases, x_2 will increase also ($0.9, 0.9$ diagonals in Σ):

$$\mathcal{N}\left(\begin{bmatrix}1\\2\end{bmatrix}, \begin{bmatrix}1 & 0.9\\0.9 & 1\end{bmatrix}\right)$$

Fig. 2.8. Bivariate Gaussian



Sampling from a multivariate distribution can be done by sampling from the standard normal X $\mathcal{N}(0, I_d)$, where $\mu = 0$ and covariance is the identity matrix I_d.

Affine transformation is applied to X, where $Y = LX + \mu$ and covariance $\Sigma_y = LL^T$ (we can omit the Σ from the affine transform, as it is an identity matrix.

The next step is to find L and this is done using a technique called the *Cholesky decomposition* (Cholesky 1910), which allows for efficient numerical solutions.

A pseudo-code below to sample from the Correlated examples is included in the Extra Notebook here:

The conditional distribution for x given y is defined as $p(x|y) = \mathcal{N}(\mu_{x|y}, \Sigma_{x|y})$, with:

$$\mu_{x|y} = \mu_x + CB^{-1}(y - \mu_y)$$

$$\Sigma_{x|y} = A - CB^{-1}C^T = \tilde{A}^{-1}$$

, with the symbols explained below:

$$\begin{bmatrix} x \\ y \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \mu_x \\ \mu_y \end{bmatrix}, \begin{bmatrix} A & C \\ C^T & B \end{bmatrix} \right) = \mathcal{N}(\mu, \Sigma)$$

Pseudo-code to find means and covariances is also included in the Extra Notebook.

Gaussian Process

Gaussian process is a stochastic process (Wikipedia, 2020) involving random variables, represented by a multivariate normal distribution. It is a joint distribution over infinitely many random variables, and as such, it is a distribution over functions $f(x)$ with a continuous domain.

When used in Machine Learning context, kernel function, which measures similarity between points is used to predict values for unseen observations.

A practical benefit from such an approach is that the result is not only a point estimate, but also a range of standard deviations σ, which can be interpreted as an uncertainty.

Difference between the multivariate Gaussian and Gaussian process is that Gaussian processes operate on mu and Σ defined as a function, which removes the limitation of the finite number of jointly distributed Gaussians.

Gaussian process is defined as:

$$f(x) \sim \mathcal{GP}(m(x), k(x, x'))$$

, where $m(x)$ is a mean function and $k(x, x')$ is a covariance function.

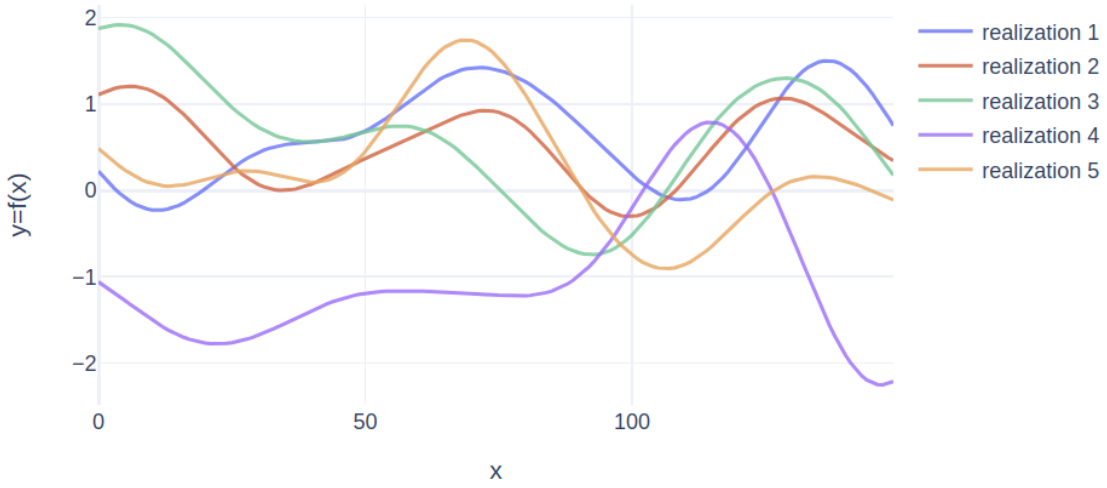
In the Bayesian language, selecting the specification for the covariance function (called the kernel function), is setting a prior information. Kernel needs to be positive-definite to be a valid function.

The most commonly seen kernel function is the *Exponential Quadratic* (*RBF* kernel), which produces a smooth function (see figure 2.9. below), and this is in fact the function used in this research. It is given by:

$$k(x_a, x_b) = exp(-\frac{1}{2\sigma^2}\|x_a - x_b\|^2)$$

One can sample from prior using a finite number of points and this results in a marginal distribution that is Gaussian.

Fig. 2.9. Sampling from RBF



Using Gaussian Process for regression is a three step process:

1. Define a prior kernel function

2. Create a posterior distribution, given some data and likelihood function
3. Generate predictions (y) for the input variables (X)

To make predictions $y_2 = f(X_2)$, one would draw samples from the posterior distribution $p(y_2|y_1, X_1, X_2)$:

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}, \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix} \right)$$

and then use the conditional distribution:

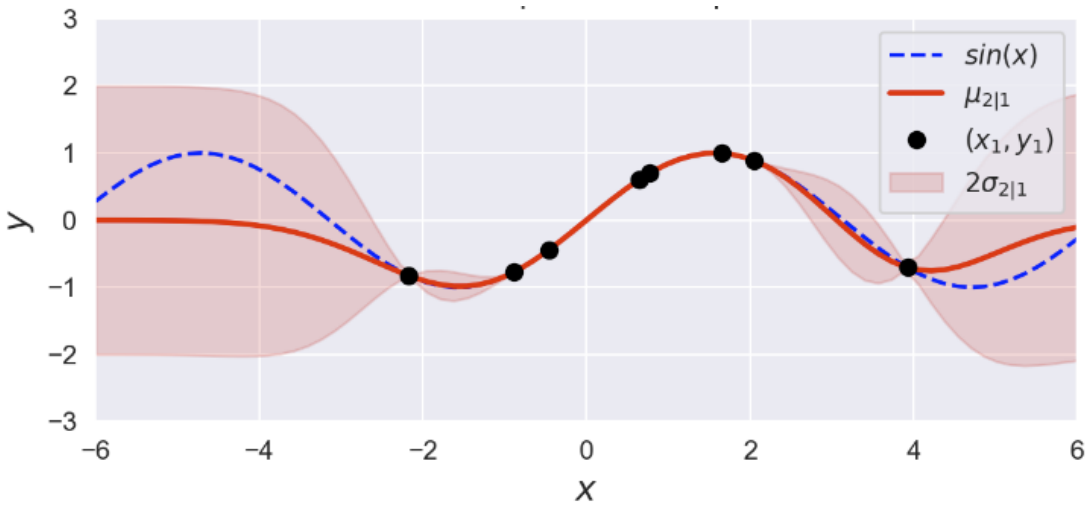
$$\mu_{2|1} = (\Sigma_{11}^{-1}\Sigma_{12})^T y_1$$

$$\Sigma_{2|1} = \Sigma_{22} - (\Sigma_{12}^{-1}\Sigma_{12})^T \Sigma_{12}$$

y_2 can be predicted by using mean $\mu_{2|1}$.

The visualization for predictions for a noiseless distribution shows that uncertainty (a salmon-color fill around the sine wave) in the points with data (black dots below) is minimal, but it grows in the sections without any data points:

Fig. 2.10. Gaussian Process predictions and uncertainty



2.2.4. Conclusion

Gaussian processes are a very elegant, robust and informed approach to Machine Learning. One not only generates predictions for unseen data, but also the uncertainty. This can be a very useful tool in the decision making.

However it is important to keep in mind that working with larger datasets can be a **challenge** due to $\mathcal{O}(n^3)$ complexity. In my eyes, certainly Gaussian process algorithms were very slow to train (given that I have only used them with < 4000 records) and they have consumed a lot of memory (GPy was much more efficient than pymc3).

There are tricks used to decrease the complexity to $\mathcal{O}(n^2)$ and new frameworks continuously work on the improvements (pymc3 library is currently switching back-end to TensorFlow), but training Gaussian Process-based models can be a challenging and time consuming task.

2.3. Auto-encoders for Anomaly Detection

One of the core three ideas in this research is *Anomaly Detection*. For example, it could very useful to alert home owners when something out of the ordinary is taking place around their property.

Below is a Literature Review about *auto-encoders*, which are Neural Network models, often used for anomaly detection in the large scale datasets (like image or text data).

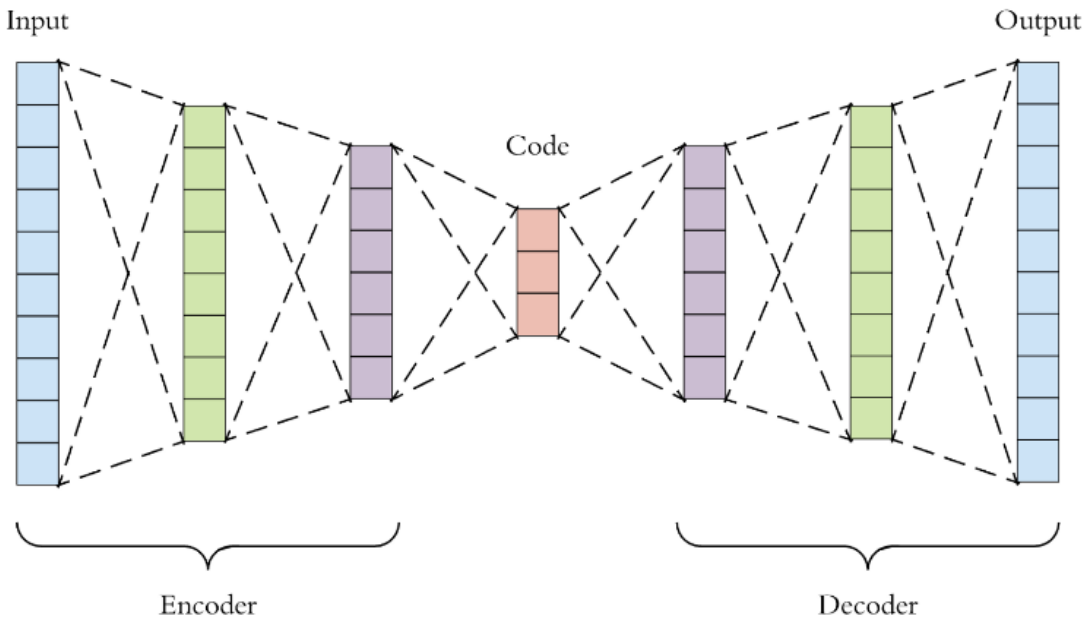
Auto-encoder learns to predict (reconstruct) its own inputs, without any prior knowledge of target outputs or labels. It is loosely classified as an unsupervised learning algorithm.

There is a weak evidence of the origins for this Neural Network architecture, which is described in an online Deep Learning book (Goodfellow-et-al-2016), where author dates the method back all the way to 80's, even though the terminology and use cases have changed drastically over the years.

In the modern are, auto-encoders are used to achieve several goals, like: - data compression (dimensionality reduction) - image de-noising - anomaly detection - machine translation

While there are several variants of this type of Neural Network, at a high level it can be represented in a following diagram from towardsdatascience 2017 article:

Fig. 2.11. Auto-encoder diagram



If the Encoder part is $h = f(x)$, then the Decoder is $r = g(h)$.

The main idea behind this design is to use a Feed-Forward Network to learn to copy the Input, but due to the size-constrained bottleneck layer in the middle, only the most salient characteristics of the data are learned (an auto-encoder, which can learn to reproduce the inputs perfectly would not be very useful).

The learning process is fairly standard and aims to minimize a loss function:

$$\mathcal{L}(x, g(f(x)))$$

, where $\mathcal{L}$ can be any differentiable function like *mean squared error*, penalizing $g(f(x))$ from being dissimilar from x.

When MSE is used, auto-encoder can be compared to PCA (Principal Component Analysis), but with a non-linear choice for functions f and g, it becomes a more powerful non-linear generalization of PCA.

There are trade-offs to such a powerful model. As authors of the Deep Learning book conclude, when this model is given too much capacity, it fails to learn anything useful.

Given this challenge, a family of auto-encoder type of models have been developed, with *Variational Auto-encoders* being the most popular one.

2.4. Conclusion

Auto-encoders conclude the Literature Review in this research. The knowledge base in the area of object detection, forecasting and anomaly detection is vast with new research papers and articles landing every day, but I hope that the overview above sets the tone on what is to come in the next chapters.

And the next Chapter focuses on the overall **System Design**.

index | prev | next

3. High Level System Design

index | prev | next

The overall system can be described as a two high level components:

- Real time frame processing used for object and anomaly detection
- Batch processing used for forecasting and anomaly detection

3.1. Real time frame processing

Fig. 3.1. System Design - Real Time Processing

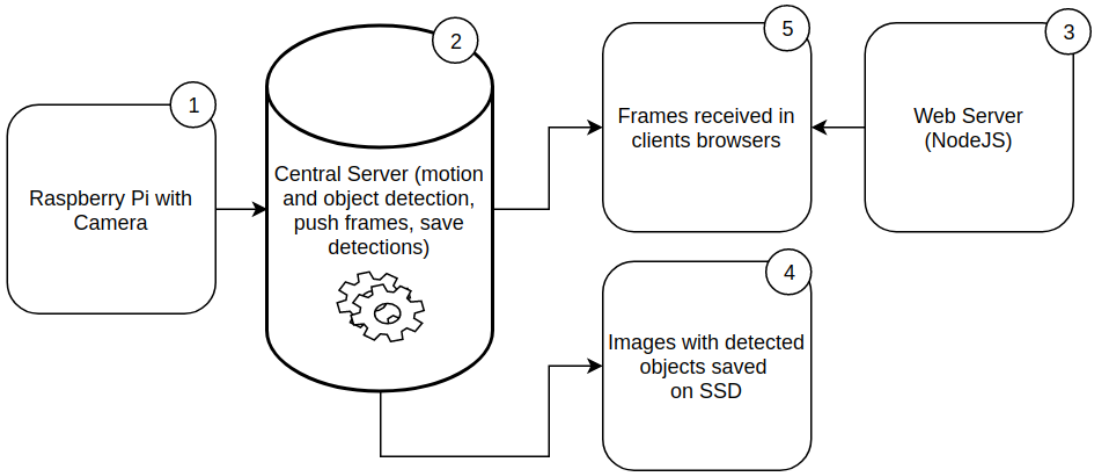


Diagram description:

- First step is to generate frames from the Raspberry Pi camera (1)
- Then frames (as Numpy arrays) flow to a Desktop PC using high performance message queue - ZMQ (2). PC then:
  - detects motion
  - detects objects and their locations in an image
  - sends every frame to connect clients (using web browsers) (5)
  - saves images on an SSD Drive (4) for further analysis and batch processing
- Client web browser application is served by NodeJS web server (3)

The data flow and the steps in data processing are described in more detail in the next chapter - Data Collection.

3.2 Batch processing

Fig. 3.2. System Design - Batch Processing

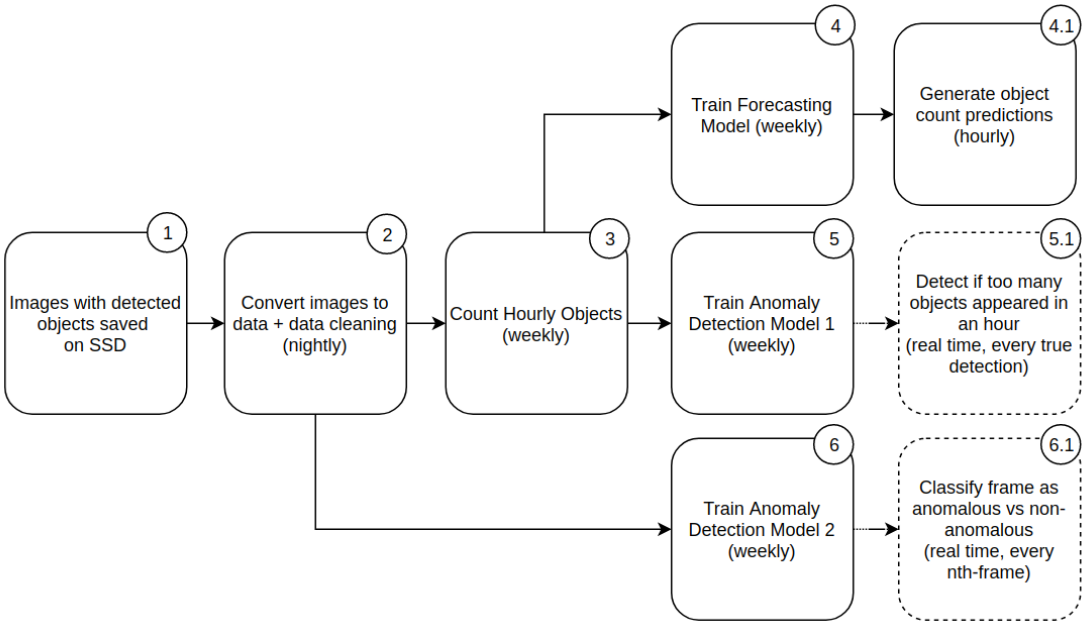


Diagram description:

- The starting point contains raw images collected on the hard drive (1)
- Every night a scheduled task is executed, where images from the whole day are converted into tabular dataset (2)
- Then in step 3, another scheduled task runs once every week and counts unique observations into hourly buckets and this dataset is used by two processes:
  - Weekly forecast training (4) of probabilistic model, which predicts count of objects to show in a given hour, predictions are updated every hour (4.1)
  - Weekly anomaly detection training script (6) which uses a probabilistic approach to set up hourly thresholds, above which new objects are treated as anomalies. This anomaly verification runs in real time (5.1)
- Tabular dataset with images is used to train an auto-encoder model capable of classifying frames as anomalous vs non-anomalous (6). The classification happens in real time (box 6.1)

Details for each process are be covered in chapters Forecasting and Anomaly Detection.

3.3. Conclusion

This chapter has covered system capabilities at a high level. Next Chapter - Data Collection - dives into more detail detail of the figure 3.1. seen above.

index | prev | next

4. Data Collection and Pre-Processing

The aim of the data collection pipeline is to use camera to stream video signal to the central processing machine, which runs a series of operations to detect objects and store images if objects of interest were identified.

This chapter is an in-depth analysis and an extension of a "Real time frames processing" diagram (Fig. 3.1.) from Chapter 3. The subject turned out to be quite complex, and I have broken it down into following sections:

- physical layer (hardware):
  - choosing hardware
  - connectivity
  - picking location for a camera
  - redundancy
- logical layer (software):
  - streaming video (called the client) and consuming video (called the server)
  - frame life-cycle (pre-processing image data, detecting motion and objects, forwarding video to other devices)
- results:
  - data representation
  - data volumes

4.1. Physical layer (hardware)

4.1.1. Choosing hardware

Streaming device:

Choosing the right hardware can be challenging and time consuming. When picking the camera system, there appears to be a two fundamentally different approaches to it:

- standalone (professional) camera unit
- mini computer with a camera module

My decision was motivated by the following factors:

- low cost
- popularity and online documentation
- flexibility in configuration
- ability to perform additional tasks on-device

Based on that I have decided that option 2 (mini computer) is a better choice. Raspberry PI with additional Pi-Camera module was an obvious choice here due to my own experience with the device and high availability of online materials for it. Raspberry PI 3 has a 4-core ARM processor and 1GB of RAM, which is quite powerful, considering the small form factor (only $3.54x2.36x0.79$ inches).

The trade-off here is that most of the standalone cameras are more durable and secure and often support Night Vision mode. All these extras come at much higher cost, little flexibility in terms of configuration options and limited online support.

Processing device:

For the server machine, I have decided to use a powerful, multi-purpose Desktop PC.

I have considered setting it up in the cloud (AWS, Google Cloud or Azure), which tends to be a popular option in the modern camera systems, however I have identified two problems with that option:

- my broadband at the time of setting it up was very slow
- there is a worry that data goes outside of the local network and can be attacked by the hackers

Below is the configuration for the PC:

- Intel i5 6-core CPU
- 32 GB RAM
- Nvidia GeForce 11 GB GPU
- Storage:
  - 256 GB NVME drive
  - 1 TB SSD drive

This kind of system allows to run a silky smooth, real time image preprocessing and object detection at high enough FPS and hourly/nightly/weekly scheduled tasks.

4.1.2. Connectivity

Initially I have set up the communication between devices as wireless, but it caused several issues. There was visible delay in the real time video stream, packets were dropping and logging in to Raspberry Pi remotely was very slow with occasional minutes of unresponsiveness.

I have chosen to wire the house, which allowed for near noiseless communication between the camera and the server, and sped up the development process significantly. The downside of this approach was that it was a timely and costly maneuver.

Below is a high level network topology diagram:

Fig. 4.1. Network Topology

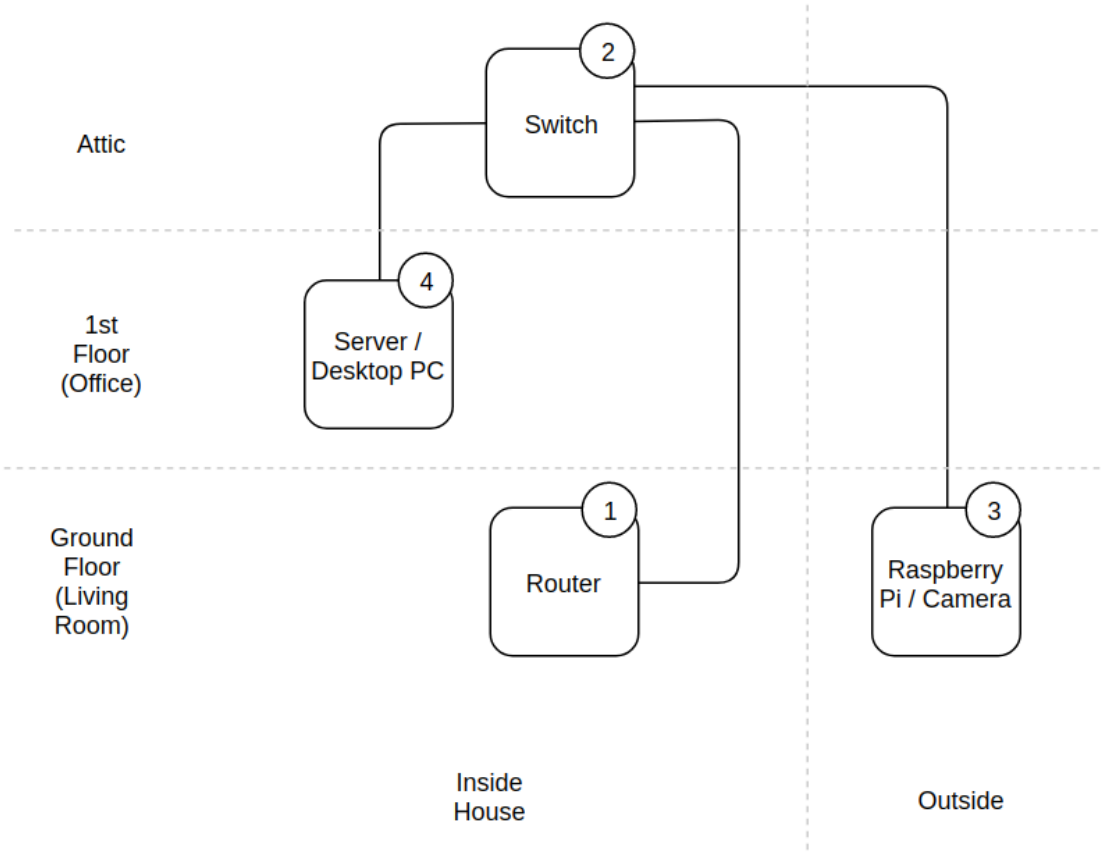


Diagram Description

In the diagram above, continuous lines represent wires, and dashed lines are either dividing floors, or the areas inside and outside the house:

- Starting from the router (1), signal goes to the attic to a 1Gb Switch (2), which has 4 Power Over Ethernet (POE) ports
- From the switch, there is a network cable going into the Raspberry Pi (3), which is located outside of the house. Raspberry Pi has an additional POE module next to it, so it only requires a single wire to receive power and network. Pi does not support 1Gb connections through its native RJ45 port, but 100Mb is more than enough to transmit video signal at high rate
- The second connection from the switch goes down through the wall from attic into the office on the first floor, where the server PC (4) is located

Static IP addresses were assigned to the devices along with the hostnames, so they can be easily accessed from within the network.

4.1.3. Camera location

Optimal camera placement is not a trivial task.

I have decided to place a camera in a position, which is usually chosen as a primary security camera: in front of the house, above the main door. This decision was made after learnings from multiple experiments with other locations (for example inside the house, behind the window).

The location chosen for the camera is certainly too low to be considered a proper security camera, but it shelters the device against the rain, wind and direct sunlight, which are often a serious issue for the vision systems.

The RaspberryPi with camera module is glued using a strong double sided tape onto the roof in the porch:

Fig. 4.1. Raspberry Pi Camera Placement #1

Fig. 4.2. Raspberry Pi Camera Placement #2

After many attempts at getting a good and flexible case for Pi, I finally found one which allows for a full 360 camera rotation along both axis (this can be seen above in figures 4.1 and 4.2).

Surprisingly enough, even during harsh storm, wind and rain, the camera mount does not shift at

all.

Another challenge, which can occur at random occasion is occlusion caused by a natural event. This could be as simple as a leaf or dust, but it can prevent camera from registering a clear picture for a long period of time. During the six months of data collection, only one such incident occurred when a spider decided to adopt the Raspberry Pi case as a house.

A good improvement for the future would be to detect the loss in image quality and send a maintenance alert to the users.

4.1.4. Redundancy

I have experienced three power outages, which meant a loss in data for three days (as I was at work during that time). Having an alternative source for the power is critical if the system must be always online. This redundancy comes at additional cost, and this is currently a limitation of this system.

What is also important is that when device comes online again, it should have a mechanism to resume streaming (or collecting) data. This will be discussed later in this chapter but is handled on both: client and server using an open source software called Supervisor.

In case of hardware failures on the server, there is a back up script running every night to synchronize images into another machine. And in case of a hardware failure on the client, a spare Raspberry Pi exists (which is currently used as a test/development box).

4.2. Logical layer (software)

Below are the key software ingredients used in this project. Each of them already exists in a working system, but can also be further refined and improved from the innovation, scalability, reliability, security and performance point of view.

4.2.1. Video streams

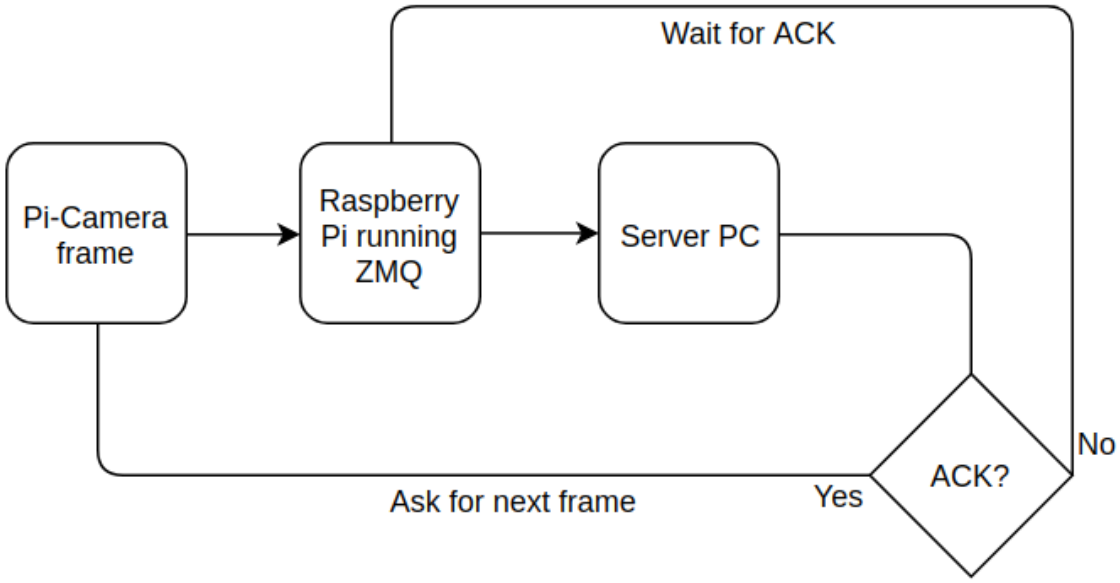
There are many ways video frames can be broadcasted to other devices.

In general, people often choose easy to setup streaming protocols, like *RTSP* if they just want to display a video in a video player like VLC, however RTSP signal can be quite troublesome to capture if further image processing is required through Open CV and Python. Also, RTSP streams the video without considering if the receiving end is online.

If the audio component is not required and full flexibility and customization is important, then a message queue might be a better option. ImageZMQ is a convenient implementation of *ZMQ/PyZMQ* - a peer to peer message queue system optimized for high performance, with an option to receive acknowledgment signal from the receiving clients.

Here is a diagram of a frame exchange between Pi-camera and PC when *ImageZMQ* is used:

Fig. 4.2. ZMQ Message Queue



In this type or architecture, there is no need to keep sending the frames when there is no-one receiving them as ACK signal is required before the next frame can be sent.

The drawback of this approach is that when receiver stops receiving, the streaming device sending script must be restarted (or connection re-initiated).

I have published two GitHub repositories with the code required to run a client, and a server.

The `client.py` and `server.py` scripts are registered in the Linux software called Supervisor, which makes sure that the processes are always ON (when system restarts or when scripts get terminated for any reason).

4.2.2. Frame lifecycle (including object detection)

This is the most complex part of Data Collection. It took me over one year to get this end-to-end process right.

Below is a data flow for each frame:

Fig. 4.3. Frame lifecycle

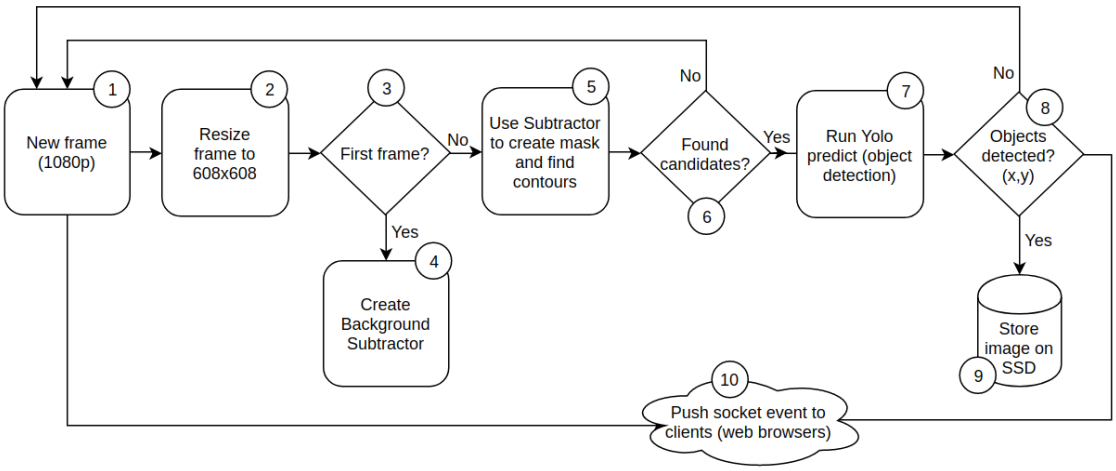


Diagram description

Picking up from previous pseudo-code for capturing the stream (server):

- Once the frame is collected from the ImageHub, it is a full HD resolution: 1920×1080 px (1)
- The frame needs to be resized (2), otherwise next steps will be very slow. The choice of 608×608 makes sense, as this is a resolution needed by the Yolo algorithm (7)
- The next step is to use a static background to extract the moving foreground. This is a *Background Subtractor's* task (discussed in detail in the Literature Review chaper)
- Before this technique can be applied, there is a check (3) if it has been already created
- Below are the arguments for the createBackgroundSubtractorMOG2 class described at opencv website:
  - `history` - Length of the history
  - `varThreshold` - Threshold on the squared Mahalanobis distance between the pixel and the model to decide whether a pixel is well described by the background model. This parameter does not affect the background update
  - `detectShadows` - If true, the algorithm will detect shadows and mark them. It decreases the speed a bit, so if you do not need this feature, set the parameter to false
  - *Mahalanobis distance* is a multivariate distance metric that measures the distance between a point and a distribution (unlike Euclidean distance, which measures a distance between two points) and it is given by (machinelearningplus 2019):

$$D^2 = (x - m)^T \cdot C^{-1} \cdot (x - m)$$

    , where x is a vector of data points, m is a vector of means for each feature, and C^{-1} is an inverse covariance matrix of independent variable
  - The parameters above are mostly heuristics and they depend on the location of the camera, the size of the objects and the type of movement to detect. It usually helps to record multiple videos from a particular location for calibration purposes. Below are the values, which worked well in this scenario:
    * `BG_SUB_HISTORY = 20` - Use rolling 20 images
    * `BG_SUB_THRESH = 30` - Ignore changes below threshold
    * `BG_SUB_SHADOWS = True` - Detect shadows

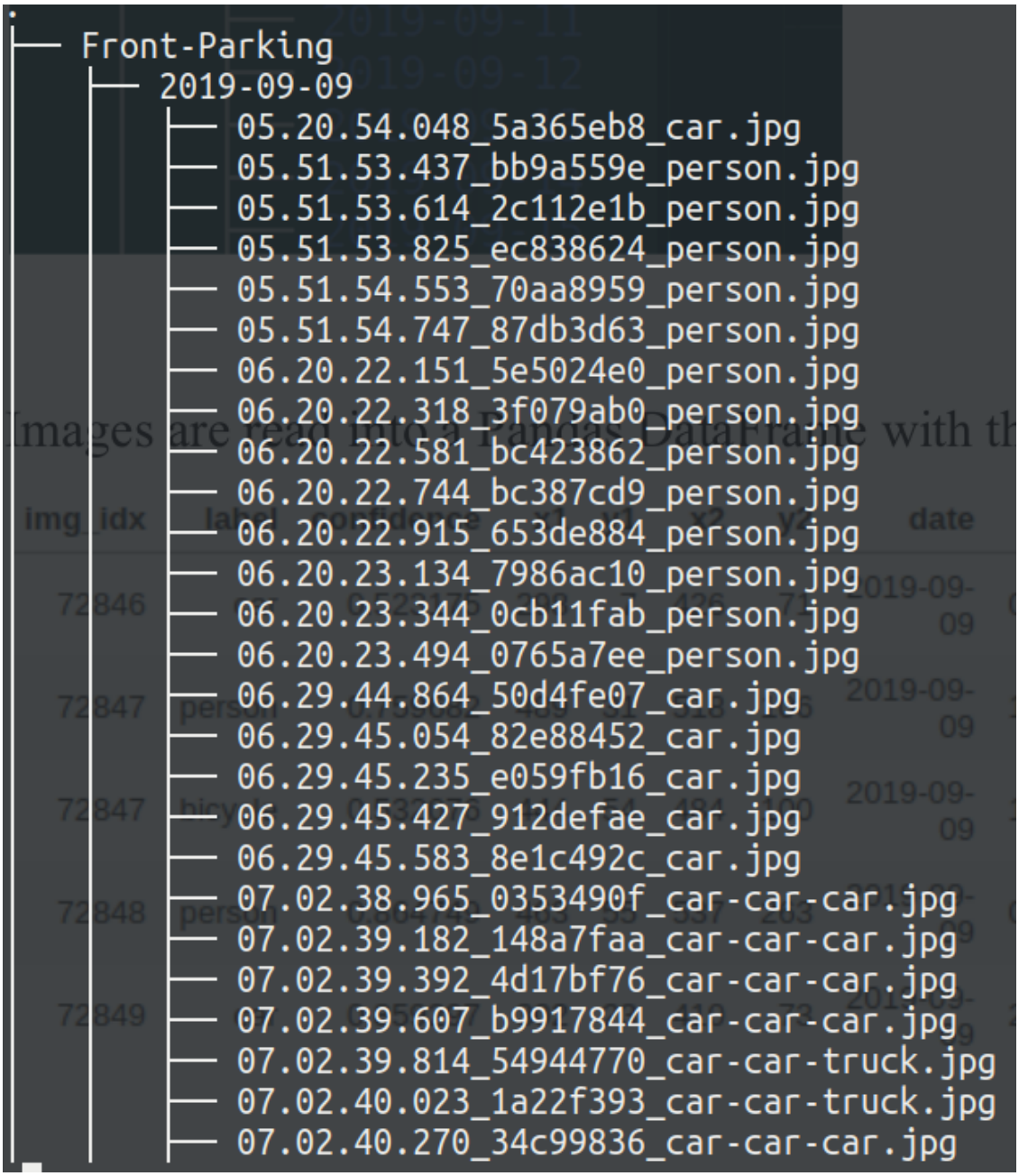
- The background subtraction algorithm is then applied (5) and the generated `mask` verified if the detected contours meet the criteria to become an object candidate
- According to my study, objects of size > 35 are good candidates for further detection. Everything below this value is not a valid object and should be dropped. Parameter `MIN_OBJ_AREA` is another heuristic, which should be calibrated for each environment
- Once candidates are found, frame is forwarded to Object Detector (Yolo V2) to generate predictions for all objects in a Coco dataset (like Person,Car,Dog,Bike etc.). This activity happens in step 7, while Yolo itself is instantiated before the main loop, using following parameters:
  - `THRESHOLD = 0.40` - Reject detections with confidence lower than 0.4
  - `GPU = 0.5` - Use only 50% of GPU for computation
- Yolo requires to provide the set of labels (yolov2.cfg), pre-trained weights (yolov2.weights), confidence threshold for predictions (reject below (0.40)) and optionally how much GPU power can be used by this process (0.5 runs smoothly on 11Gb GPU with 608×608 images). Predictions are generated with `tfnet.return_predict(frame)` code, which returns a list of objects, with the confidence and x,y coordinates
- Also within step 7, script needs to filter out all objects, which are not tracked (like lamp, monitor, phone etc.) and only then final predictions are obtained and can be checked (8)
- If an object of interest is detected, frame is saved as an image on the hard drive (9) in a folder corresponding to a date (along with the filename, representing time and name of all objects detected in a frame)
- Then, independently of the object detection process, the original $1080p$ frame, along with the predictions, are pushed through the socket server (10) to the outside world (a web application can connect to this socket and receive a real time stream with object detections)
- The values sent over socket will help to draw the bounding boxes around objects for any screen size (and multiple devices) in the UI layer

4.3. Results

The result of the data collection are images with valid detections.

Image data is organized as a hierarchy of folders. The top level contains the streaming device name, which then contains directories corresponding to dates with image files inside. Image filenames contain objects detected and time.

Fig. 4.4. Collected Images



The dataset collected for this research contains over $600K$ images between 09th of September 2019 and 02nd Match 2020. The size of a single $1080p$ image compressed to jpg is approx. $300kB$. The total size of the dataset is then approx. $180GB$.

An average number of raw images captured per day is approx. 2000.

4.4. Conclusion

This Notebook presented a bird eye view over the valley of Data Collection. There are a lot of details, which have been omitted, like error handling, socket server implementation and running an infinite loop in a separate thread to capture the stream within a *Flask* app context. However all these nuggets can be found in the Extra Script 1 - app.py, which was used to gather the dataset in this research.

Considering the amount of processing for each frame, this pipeline runs at impressive 30 frames per second, and provides a smooth experience for the end users, who can observe real time object detections in a web browser.

There are a number of improvements already identified in this process, which are left for the future iterations:

- include *privacy mode*, where bounding boxes with detected people are blurred
- switch object detector to Yolo V4 for increased detection accuracy and speed
- test image segmentation techniques to improve the bounding boxes approach
- add hourly forecast of expected objects for a day into the UI
- trigger alerts when anomalies are detected:
  - number of objects in a current hour is suspiciously high
  - raw image content looks like it has come from a different population than the images registered in the history

Next chapter focuses on generating Forecast for the number of objects expected to appear in an hour.

index | prev | next

5. Forecasting

index | prev | next

Can detected objects be used as a dataset to predict future object counts?

Asking this question has changed my perspective about this project. Initially it was solely about Object Detection pipeline, however when that was done, I began to look at images as a valuable data source with diverse applications.

Forecasting expected object counts can be very beneficial. At allows for informed planning decisions and anomaly detection. For example, if the predicted number of people at 3PM on a weekend-day is 5, and the observed count is 15, then something unusual might be taking place, and perhaps a user would like to be notified about it.

Time interval:

It is important to decide on the time interval for the forecast.

I have explored several intervals, including 15-minutes, 1-hour and 3-hour windows, but the 1 hour turned out to be the best choice for the following reasons:

- 15 minute forecast is too noisy and it is too difficult to predict object counts with high accuracy
- 3 hour window is too large and reduces the data size too much
- 1 hour is a good trade-off

This value will depend on the use case and the available data and it should be made customizable.

This Chapter:

This Notebook is an in-depth study of the steps taken to generate such a forecast:

- data extraction and clean up of raw images
- counting objects paradigm
- more data preparation
- additional data sources
- exploratory data analysis (EDA)
- forecasting

Forecast needs to be generated for a specific object type (like a *Person* or *Vehicle*).

The Section section in the bottom will summarize the findings, where the best model, which should be used going forward will be identified.

Notes:

- Code samples with comments and more plots are available in Extra Notebook 5
- To keep this chapter well organized and concise, I have only included a study on *Person* class (but the approach is generic and should work for all object classes)
- For model comparisons I have used following metrics:
  - MPD - Mean Poisson Deviance
  - MSE - Mean Squared Error
  - MAE - Mean Absolute Error
  - R2 - R2 Score
  - ACC - Accuracy

5.1. Extract raw image data

In order to perform forecasting, historical data is needed. In case of this research, the historical data are the images with detections, which need to be pre-processed and turned into a tabular format, which will allow further analysis.

This is where Python as a general purpose programming language can help. Below is a list of steps used to accomplish this goal:

- start with images, which are stored on the hard drive
- scan through all directories representing dates and find all images within these directories
- resize images to 608×608 px and use Yolo to detect objects
- keep saving all detections for each day in a csv file (just in case processing fails)
- load all images from csv files into a single, large Pandas DataFrame
- save dataframe with the whole dataset into an efficient .parquet format

A sample of the processed dataset is presented below:

Fig. 5.1. Detections tabular data

label	confidence	x1	y1	x2	x2	filename	date_time
car	0.523175	298	7	426	426	07.02.40.270_34c99836_car-car-car.jpg	2019-09-09 07:02:40.270
person	0.759682	489	31	518	518	12.02.42.921_ea6c9143_person-bicycle.jpg	2019-09-09 12:02:42.921
bicycle	0.532076	444	54	484	484	12.02.42.921_ea6c9143_person-bicycle.jpg	2019-09-09 12:02:42.921
person	0.864749	463	55	537	537	07.30.02.409_c5662b14_person-car-car.jpg	2019-09-09 07:30:02.409
car	0.859297	302	23	410	410	20.26.56.841_4ba2f42d_car.jpg	2019-09-09 20:26:56.841

The dataset contains object classes under `label` feature, object detector's `confidence`, `x,y` coordinates for bounding boxes around detected objects, `filename` of the image, `date and time` of the detection, and a few more less interesting properties.

There are 643,471 records with detections, which came from 222,195 unique images, which yields 2.9 objects per image on average.

The implementation details and more commentary for this transformation can be found in the Extra Notebook 2.

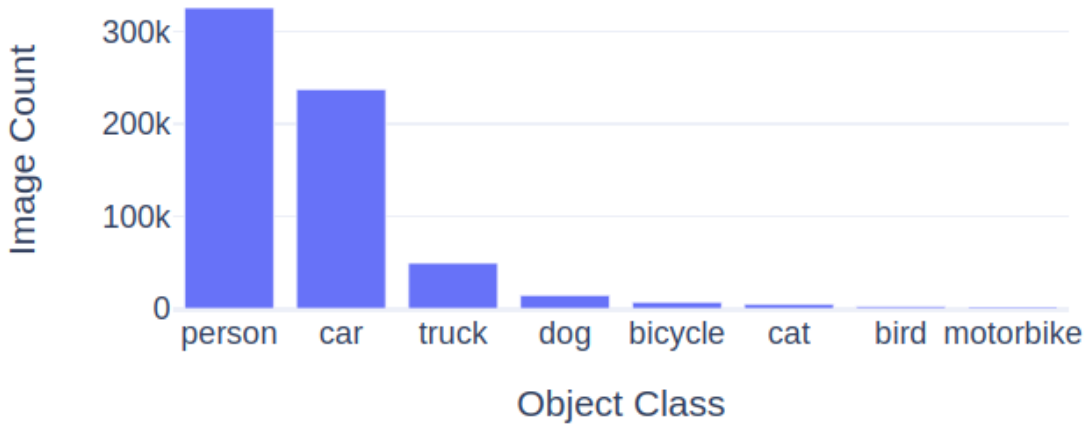
5.2. Count objects in frame sequences

The next step in the data pre-processing is to determine a method to count objects in a given hour.

First, some cleaning was required. Throughout the data collection process, there were three days of downtime, so any data collected randomly for these days needed to be purged.

Now it is beneficial to observe a distribution of classes captured by the object detection pipeline:

Fig. 5.2. Class Distribution



This picture shows a significant disproportion between the first two classes and the rest.

Based on this distribution I have decided to merge *cars* and *trucks* into a *vehicle* class and focus on predictions for *Person* and *Vehicle* only.

The iterative approach I took to counting objects in an hour is quite simple and works well for a dataset with a low number of objects/events:

- sort data by time
- split dataset by object type (label) and perform following tasks for each:
  - iterate through all detections
  - calculate difference in time between consecutive observations
  - calculate centroid for the detected boxes: $x_{center} = (x_{left} + x_{right})/2$, $y_{center} = (y_{top} + y_{bottom})/2$
  - use x_{center}, y_{center} centroid coordinates to calculate an Euclidean Distance between object centroids in consecutive frames, if it is the same observation in a sequence, then the center will be close to the previous center
  - keep only objects where the difference in time and distance are greater than the predefined thresholds (these have been initially set using heuristics and will change depending on the camera and its location, please see plots below for more details)

The formula to calculate the distance is the well known Pythagorean metric:

$$d = \sqrt{(x1 - x2)^2 + (y1 - y2)^2}$$

The algorithm above works well with the following threshold parameters:

- `THRESH_NEW_EVENT_SECS = 10` - Time after which we treat another observation as unique count
- `THRESH_NEW_EVENT_MIN_DISTANCE = 30` - Distance in pixels between centroids

There are multiple results from this pre-processing step (one for each object class), but a sample from the *Person* dataset is provided below and it represents unique observations extracted from sequences:

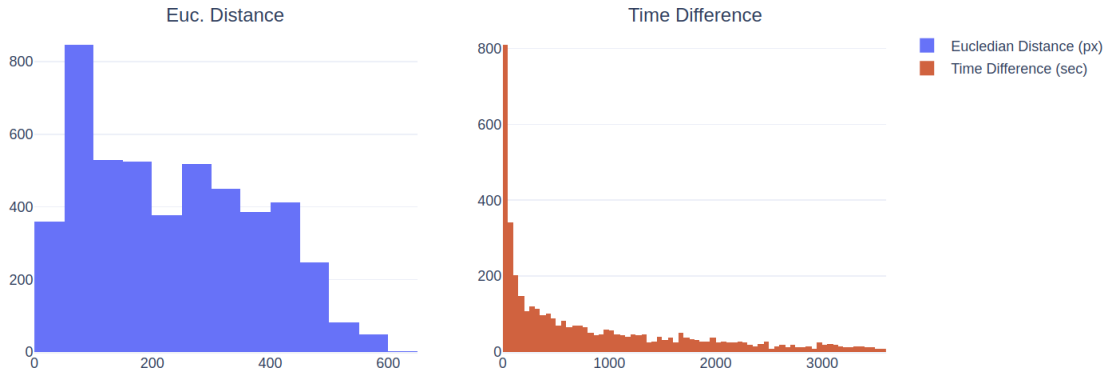
Fig. 5.3. Unique detections

label	confidence	date	time	sec_diff	euc_distance
person	0.450496	2019-09-09	07.03.03	2560.471	279.827179
person	0.658724	2019-09-09	07.29.50	1606.475	291.185508
person	0.439394	2019-09-09	08.07.56	2270.200	247.779136
person	0.768141	2019-09-09	08.42.40	2075.153	103.711137
person	0.545153	2019-09-09	08.52.30	585.361	195.661059

This dataset only contains observations for a single class label and adds a `time difference` and `euclidean distance` versus a previous image. The shape is 4790×26.

To confirm the validity of the method, below are two distributions: left one showing Euclidean Distance calculated in pixels, and right one showing difference in seconds between observations:

Fig. 5.4. Distance metrics



Plots interpretation:

It can be observed that there tends to be a quite wide distribution of pixel differences between objects with 50 to 100 pixels being the most probable range. The x-values in the graph make sense as the images are 608×608 px in size.

The differences in time between objects are a little bit surprising with majority of objects being captured between 0 and 50 seconds between each other. Perhaps increasing the time between objects could be an interesting tweak to explore (I am leaving this for the future work on this project).

The implementation details and more commentary for this data transformation can be found in the Extra Notebook 2 - ObjectCount:

5.3. Further data preparation

One can very quickly notice that the majority of work in the real-life data driven/Machine Learning projects is cleaning and preparing the data.

Following this trend, the next step is to roll up the dataset to daily/hourly level.

Luckily Pandas is a very useful tool for working with dates and has a `resample` method, which allows to aggregate the data to hourly level and fill the gaps with no observations as 0's.

When this is done, more time-related features can be created, like:

- `hour`
- `n_month`
- `is_weekend_day`

Below are a sample two records of the dataset after this step of data preparation with the full code is included in an Extra Notebook 5:

Fig. 5.5. Object counts data

date	hour	obs_count	n_month	n_week_in_month	day_of_week	day_of_week_name	is_weekend_day	day_of_week_name_short
2019-09-09	7	2	9	2	0	Monday	0	WeekDay
2019-09-09	8	3	9	2	0	Monday	0	WeekDay

5.4. Weather data

I have also pulled 6 months of historical weather data to analyze if merging this data with the historical object counts can increase the prediction accuracy.

It turns out that DarkSky - one of the most popular weather applications, has an API, which can be used for free up to $1,000$ requests per day (note that it does not take any new registrations after recent acquisition by Apple).

Pandas `date_range` function can be used to generate a range of DateTime objects with an hourly interval, which need to be converted to Unix Timestamps.

Making an API request is a simple `GET` HTTP request to *https://api.darksky.net/forecast* with an `API_KEY` (received during registration to the service) and a `latitude`,`longitude` parameters for the desired location. For my house it is:

$$lat = 51.802931199999996, long = -8.302591999999999$$

Pulling this data for six months at hourly interval generates 4224 data points with 24 features.

The questions that are possible to answer now are:

- did it rain at 8AM on Monday?
- was there a storm last Friday at 4PM?
- what was the temperature yesterday at midday?

Full details and code to generate this dataset is located in the Notebooks folder as Extra Notebook 4.

5.5. Exploratory data analysis (EDA)

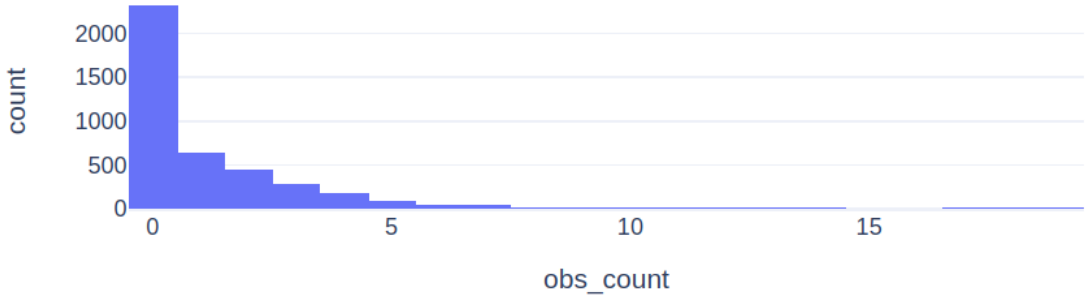
Once the object counts data is prepared and weather data fetched, these two data sources can be combined using a common `timestamp` attribute. This merged dataset is now a basis for data analysis and forecasting.

Target variable distribution

The variable to predict is the number of object counts within an hour.

It is expected that it is dominated by 0's as during the night, and often during the day, there would be many hours without any observations:

Fig. 5.6. Object counts frequency / all hours combined



The plot above shows some very high counts on the right hand side (above *count* = 5). It will be very challenging to predict these numbers and only a model with a very *high variance* would be able to do that. However, as mentioned in the Literature Review, such a model is not a good choice, as it will often perform poorly on the test-data (due to memorizing the data instead of pattern learning).

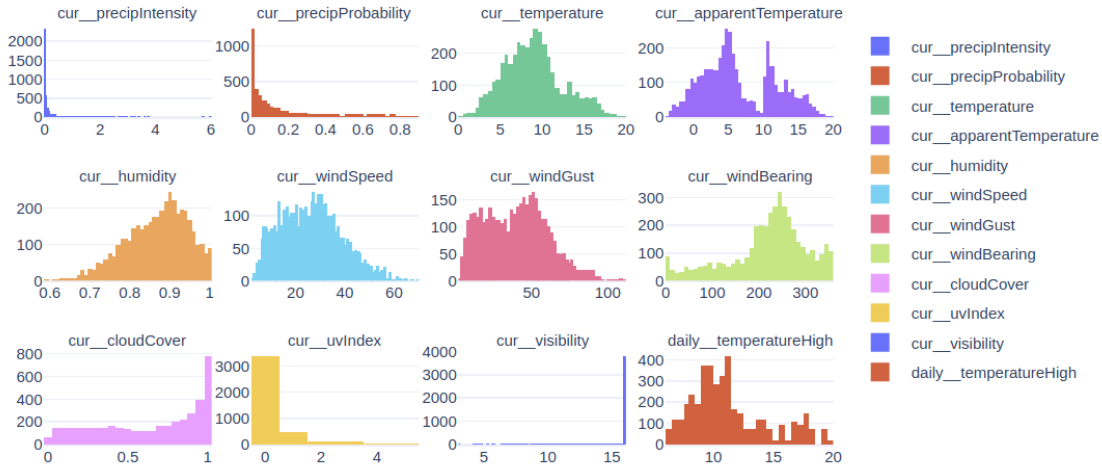
Missing values

One of the benefits of manually collected dataset is that there is no missing data. However very often when one is given a dataset a decision needs to be made about the treatment of missing data (remove records, try to impute the values or mark as missing).

Outliers

Below is the plot of weather related numerical features, where outliers will be visible:

Fig. 5.7. Feature Histograms



It is clearly visible that *precipitation*, *uvIndex* and *visibility* are heavily skewed and contain serious outliers.

This can often negatively affect the predictive power of the models, especially if they assume that the features are normally distributed or does not contain outliers.

The most popular method for dealing with outliers is to transform the features using one of the following transformations:

- square root - $\sqrt{x}$
- natural logarithm - $\updownarrow\wr\}(x)$
- reciprocal transformation -$1/x$

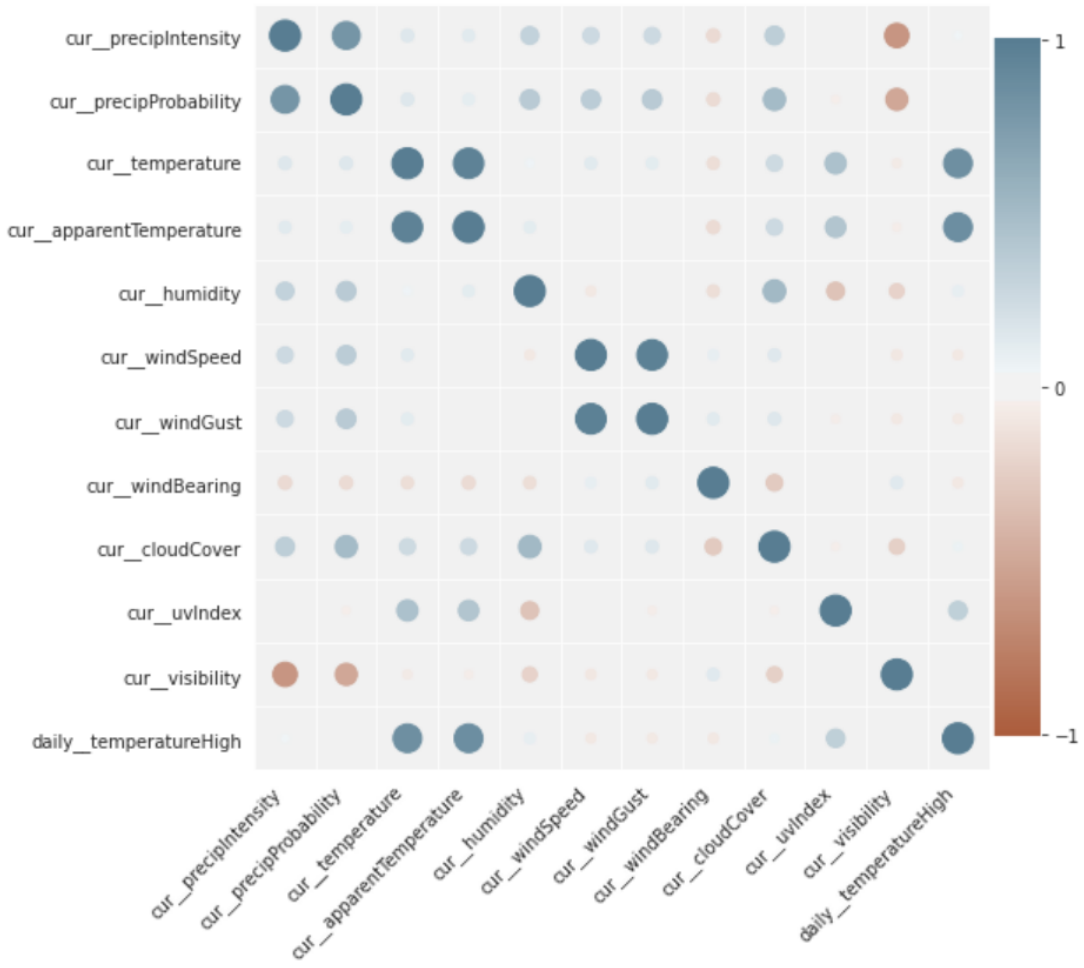
Sometimes it is also accepted to remove observations with outliers all together, however in case of forecasting it would mean a loss of continuity in the data.

The good part of this exercise is that the Machine Learning algorithms I have tried, have produced the same results with and without these transformations, so I have decided to not transform the features, as if there are not transformed - they are easier to understand and interpret.

Feature co-linearity

If independent variables (x) are correlated with each other, it can be a problem for the statistical models (like Linear Regression). One way to analyze the effect of features on each other is to plot a correlation matrix:

Fig. 5.8. Feature correlation



This kind of plot can be really useful as it helps to discard features, which are too correlated with each other, influence the speed of training and (in terms of statistical models) decrease coefficients.

Colors show the direction of correlation (blue is negative and red positive), and the size plus opacity of the circles show the strength of the relationship.

It is clear from the graph (and makes logical sense) that `apparentTemperature` will be highly correlated with `dailyTemperature` and that `humidity` is somewhat correlated with the `cloudCover`.

5.6. Predicting counts

This section covers the following activities in order to predict object counts, given the available dataset:

- create a naive model (without Machine Learning) and calculate error rate
- use feature selection techniques to see which features should be included in the ML model
- train several Machine Learning models and calculate error rate
- conclude with a recommendation of the method to use going forward

5.6.1. Naive model

"All models are wrong, but some are useful" (Box 1976).

Starting with a naive method to generate predictions is often a good idea, as the result is a baseline model to beat with more sophisticated methods.

Idea:

Grouping the dataset by hour and calculating a mean of object counts forms a very basic forecast method.

$$forecast(X_h) = roundInt(\frac{1}{n}\sum_{i=1}^{n} x_i)$$

, where X_h is the training dataset containing all observations for a given hour h, and n is a number of observations in that training set.

Benefits:

- it is easy to understand and explain
- it is also fast to compute and requires low resources
- it works for each object class without tweaking features or parameters

Downsides:

- it has low accuracy
- it does not take into account other factors (like *weather-type* or *day-of-week*)
- it is skewed by outliers in the target variable
- it does not provide the uncertainty about the results

Implementation:

- split dataset into training and test sets 5 times (5-Fold *Cross Validation*) and for each fold:
- calculate mean averages for each hour
- calculate metrics against the test-set:
- Mean Poisson Deviance:

$$MPD = \frac{1}{n_{samples}} \sum_{i=0}^{n_{samples}-1} 2(y_i log(y/\hat{y}_i) + \hat{y}_i - y_i)$$

, where $\hat{y}_i$ is the i-th predicted value, and y_i is the i-th true value (source)

- Mean Absolute Error:

$$MAE = \frac{1}{n}\sum_{i=1}^{n} |y_i - \hat{y}_i|$$

- Mean Squared Error:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

- R2 Squared:

$$r2 = 1 - \frac{SS_{res}}{SS_{tot}}$$

, where SS_{res} is the sum of squares of residuals and SS_{tot} is the total sum of squares (source)

- Accuracy:

$$acc = \begin{cases} 1, & \text{if } predicted = true \\ 0, & \text{otherwise} \end{cases}$$

Below are the values calculated for the Naive model:

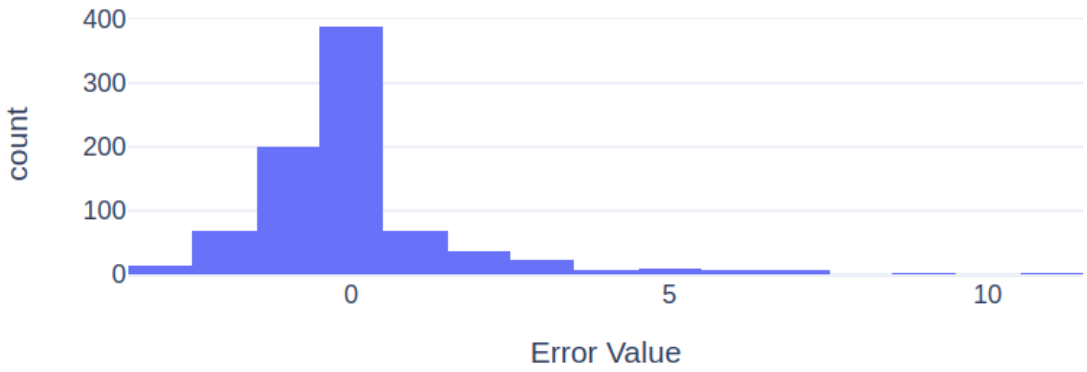
Tbl. 5.1. Error metrics - Naive model

Metric	Score
MPD	1.42
MSE	2.81
MAE	0.94
R2	0.27
ACC	0.49

Looking at the metrics above, the errors are quite high considering that the count values are low. The R2 Score of 0.27 is considered a relatively poor indicator of the predictive power, but at the same time it is not Zero (which would mean that model is not able to predict any variability in the response).

Distribution of errors made by this model:

Fig. 5.9. Error distribution - Naive model



As expected, the high counts on the right hand side is where the model made bigger errors.

It has also overestimated a lot of values. This means that the model is biased, and in general, bias in Machine Learning is not a good symptom. For this model there is no greater penalty for over or under-estimating values, so the goal should be as little skew as possible in the distribution of errors (close to *Normal*).

This is not always the case. For example in *Business Analytics* forecasting, it might be better to overestimate the demand and build extra stock, than not being able to sell products to the customers due to over-constrained supply.

Even though there are clear indicators of problems in this model, it is still very useful.

It can now be queried for an expected count at a given hour:

- 4AM - 0
- 6AM - 0.61, which can be rounded up to 1 (or Poisson probability density can be used to estimate probabilities for each count, this will be explored later)
- 4PM - 2.99, which can be rounded up to 3

These predictions are actually not very wrong, and they are quite inline with the general expectations, given the three scenarios above.

5.6.2. Machine Learning

Predicting object counts can be framed as a Supervised, Machine Learning problem. The historical counts are the target values (y), and other factors, like `hour`, `day-of-week`, `temperature`, `precipitation` are the input features (X).

The tricky nature of the target values is that even though they are numeric, they are not continuous, but rather a special case of a Binomial Distribution, where the number of trials goes to infinity.

The counts are non-negative integers $\{0, 1, 2, 3, 4...\}$and can be modeled using *Poisson Process*, where one would estimate rate (λ) of observations in a given time interval, and use a simple set of equations to answer questions like:

- Given λ and a time interval, what is the probability of seeing next observation in the next 15 minutes?
- Given λ, what is the probability of seeing 6 (or any number) observations in a time interval?

This chapter includes results for the following Machine Learning models:

- Decision tree regressor (and Vanilla Decision Tree as warm up to a more complex model)
- Gradient boosted decision tree regressor
- Gaussian Process

I have also tested many other models:

- Linear Regression
- Support Vector Machines
- Feed Forward Neural Networks
- Long-Short Term Memory Recurrent Neural Networks

However, apart from a good experience in training a broad number of Machine Learning models, I have not found them to be beneficial to this research:

- some were too simple to capture complex, non-linear relationships (Linear Regression)
- some were too complex or unstable for the data volumes and much slower to train and iterate on (LSTM Neural Networks)

5.6.3. Feature Selection for Machine Learning

Before applying Machine Learning to the problem, the correct features to use should be identified. Below are the 3 methods, which I have applied in this research:

- select K-Best using statistical test
- feature importances
- correlation matrix

K-Best features

The aim of this technique is to test using statistical methods the linear relationships between features and a target variable. For regression problem it uses Pearson correlation:

$$corr = ((x - \mu_x) \cdot (y - \mu_y))/(\sigma_x \cdot \sigma_y)$$

Correlation is then transformed to an F Score, and then p-value sklearn docs.

Below are some useful statistics for the sample best 8 features:

Tbl. 5.2. KBest feature selection

	feature	k_best_score	p_value	pearson_corr
0	cur__uvIndex	229.220	0.000	0.254
1	cur__humidity	212.884	0.000	-0.246
2	cur__temperature	105.037	0.000	0.175
3	cur__apparentTemperature	93.344	0.000	0.166
4	hour	66.064	0.000	0.140
5	cur__precipProbability	18.472	0.000	-0.074
6	daily__temperatureHigh	17.716	0.000	0.073
7	cur__cloudCover	13.205	0.000	-0.063

The statistical tests often focus on the notion of the *null hypothesis*, which is an assumption that a feature does not have a significant relationship with a target variable.

The *p-value* is a probability used to determine if the null hypothesis can be rejected (meaning there is a correlation):

$$X = \begin{cases} 0, & \text{if } a = 1 \\ 1, & \text{otherwise} \end{cases}$$

Looking again at the table Tbl. 5.2., all features have $p < 0.05$, so they do influence the count of objects, where the last column (`pearson_corr`) determines the strength and direction of the correlation.

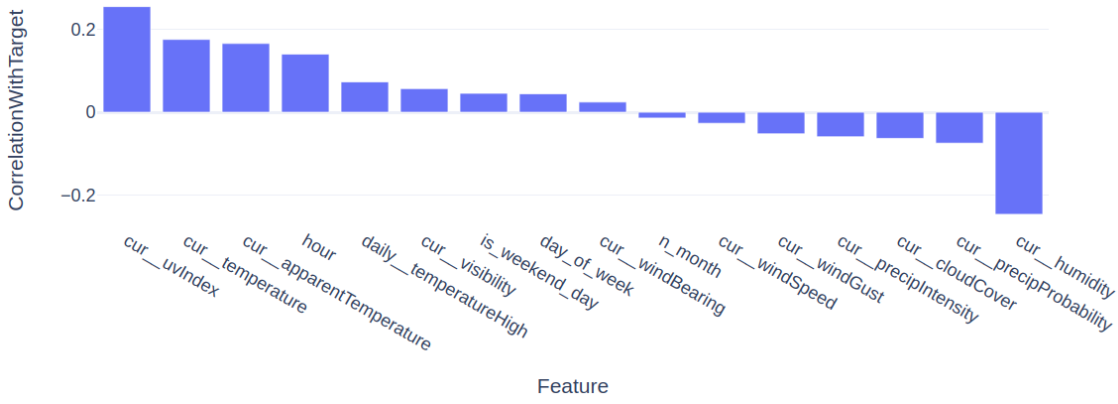
Plot below shows a jointplot between the best feature `cur__uvIndex` and `obs_count` (for 800 samples, which is 25% of the training dataset):

Fig. 5.10. KBest jointplot

Since the Pearson correlation between the UV-Index and count of observations is positive (0.331), then it means that the count of observations increases linearly when UV-Index increases.

Looking at all the correlations between all features and the target variable shows a following picture:

Fig. 5.11. KBest - correlations with Target Variable



Drawbacks:

While it is always advised to verify the relationships between variables, the approach above assumes that they are linear. In the real-life however, this assumption very often crumbles and there might be a weak dependency between the features as well, which may be difficult to deal with. As a rule of thumb, features should not be discarded too easily.

As usual, detail implementation, more techniques and additional commentary for the plots can be found in the Extra Notebook 5.

5.6.4. Decision Tree

Decision Trees are one of the most basic and easy to interpret ML models. They do not require data scaling, and can be used for both: Data Analysis and Machine Learning (classification and regression).

The theoretical aspect of Decision Trees has been covered in the Literature Review Chapter, but in a nutshell model searches for the best features and values to split the dataset, so the nodes in the tree fall into two groups separating the Target Variable values.

When trained on the Person object counts dataset, as expected, the algorithm chose to split the dataset by *UV-Index* in the root node, and then by *hour* and *temperature*. This can be observed in the visualization below:

Fig. 5.12. Decision tree - nodes

Below are the results achieved from running a 5-fold Cross Validation on this model:

Tbl. 5.3. Error metrics - Decision Tree

Metric	Person-Score
MPD	1.39
MSE	2.73
MAE	0.92
R2	0.29
ACC	0.53

This is already a an improvement over the naive model, which had the $R2\ Score = 2.24$.

Fig. 5.13. Decision tree - error distribution

The error distribution looks more evenly spread, which means that the model is less biased than the Naive model.

5.6.5. Gradient Boosting Regressor Tree

There are many extensions of a Decision Tree model to make it more generalizable to unseen data.

Gradient Boosting Trees use a more advanced mathematically technique to calculate the gradient of a loss function (like mean squared error) to find optimal parameters to reduce the errors.

The implementation in sklearn is designed to work well with larger datasets and generalizes better, as it does not have to search across all feature values to find the best split value, but creates $N-bins$ instead.

This model, cross validated across 5 folds with hyper-parameters estimated using a GridSearch across $10K$ models is capable of achieving the following results:

Tbl. 5.4. Error metrics - Gradient Boosted DT

Metric	Score
MPD	1.25
MSE	2.47
MAE	0.87
R2	0.36
ACC	0.54

Looking at the R2 Score, this model achieves a significantly higher goodness of fit $R2\ Score = 0.36$.

Benefits:

- due to the usage of gradients and multiple training passes, there is an opportunity to see how the error decreases over time on the training and testing sets. This is a common strategy in Neural Networks, but not so often in the Sci-kit Learn framework models
- the model runs very fast, considering how much computation is happening behind the scenes
- this regressor also allows to use a Poisson loss function, which is the most suitable loss for the count datasets (please refer to the Literature Review Chapter for the theory and application of the Poisson distribution
- the plot below shows a good convergence after 400 iterations (validation curve is flattening) and there are no signs of overfitting (validation curve is not increasing):

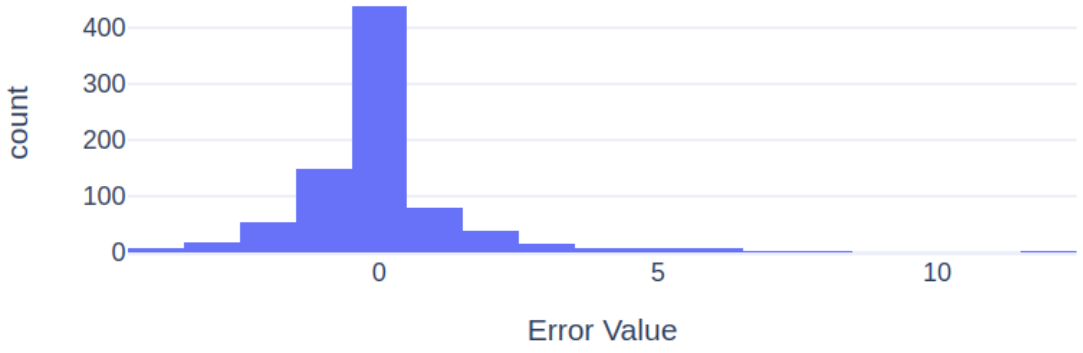
Fig. 5.14. Loss curve - Gradient Based Decision Tree

Drawbacks:

- the drawback to this model is that it is not easy to interpret the multiple trees generated inside the algorithm and that it is much more complex than the simple Decision Tree model. Obviously this also a benefit, as it is also much more capable
- the predictions from the model are point estimate, which means that this model does not return the uncertainty about the results, so it is difficult to reason about where model was and was not confident about the generated outputs

Error Distribution:

Fig. 5.15. Error distribution - Gradient Based Decision Tree



Similar to vanilla Decision Tree model, the errors are distributed quite evenly, which is a good sign of a weakly-biased model.

5.6.6. Gaussian Process

The last model tested in this chapter is the Probabilistic Model using Bayesian inference method to update beliefs after observing the data.

Model uses an *RBF kernel* as a covariance function with 1.0 for `variance` and `lengthscale` and a Poisson family function is used as a likelihood.

Training and optimizing parameters converges after 18 iterations, in under 4 minutes.

The default optimizer in the *GPy* Python Gaussian Process framework utilized is gradient based (using *L-BFGS-B* optimization algorithm by default). This can be displayed by using a verbose output in the model's `optimize` step.

The count of optimized parameters is $N\_Features + 1$, as it is a lengthscale for each feature and one parameter for the variance. All values can be inspected by accessing them as the properties of kernel object (`model.kernel`).

Below are some findings about using Gaussian Process for Machine Learning:

Drawbacks:

- choosing a library was a problem. The most popular choice - pymc3 - consumed over $20GB$ of RAM on a Gaussian Processes with only single feature, and I have switched to another package - GPy
- some functions in GPy use deprecated features from other libraries, and warnings need to be explicitly silenced
- speed of training can be a problem as well. In comparison to Gradient Boosting Decision Trees it is very time consuming (4 minutes at minimum versus 10 seconds)
- online documentation is very limited and often outdated
- it is quite common to come across numerical instability issues
- model, which works well for a *Person* class does not work so well for other classes any more, and additional search for good features/parameters is required

Benefits:

- the errors produced by this model are the lowest from all the models tested
- there is a clear interpretation of the results in the context of count data
- due to the access to full covariance and mean functions, there is an opportunity to sample from the posterior and treat the standard deviation of sampled functions as a measure of uncertainty, which can be used to improve the credibility of predictions

Generating Predictions:

The process of generating predictions using GPy is quite interesting:

- First objective (like in most of probabilistic frameworks) is to draw N samples from the Posterior. Usually, a large enough number, like 500 is sufficient
  - The `predict_noiseless` function gives the mean functions of the Gaussian Process and the full covariance matrix

  - These two objects are then used to sample from a multivariate normal distribution (this can be actually called sampling from the Posterior in Bayesian terminology), which generates the data of shape ($N\_predictions\ x\ N\_samples$), for example if there are 800 predictions and the chosen sample size is 500, then the shape of the sampled data is 800×500
- The sampled functions need to be transformed through $exp(x)$, which also ensures no negative intensities (there can not be a negative count)
- The mean rates (λ) can be estimated by averaging the exponentiated sampled functions
- Calculating standard deviation (σ) from exponentiated sampled functions can be interpreted as a measure of uncertainty around predictions

With values for λ, it is now possible to obtain counts for the predictions (or even more granular - a probability for each count, like $\{0, 1, 2...20\}$).

It is also possible to answer questions like: how many objects to expect in the first 12 minutes of the current hour?

With values for σ, uncertainty can be exploited to influence predictions, for example if uncertainty is low it could be sufficient to show a single number, but if the uncertainty is high, perhaps a better idea would be to show a range of most probable answers.

Below is an example for how a count of events be obtained from a mean rate:

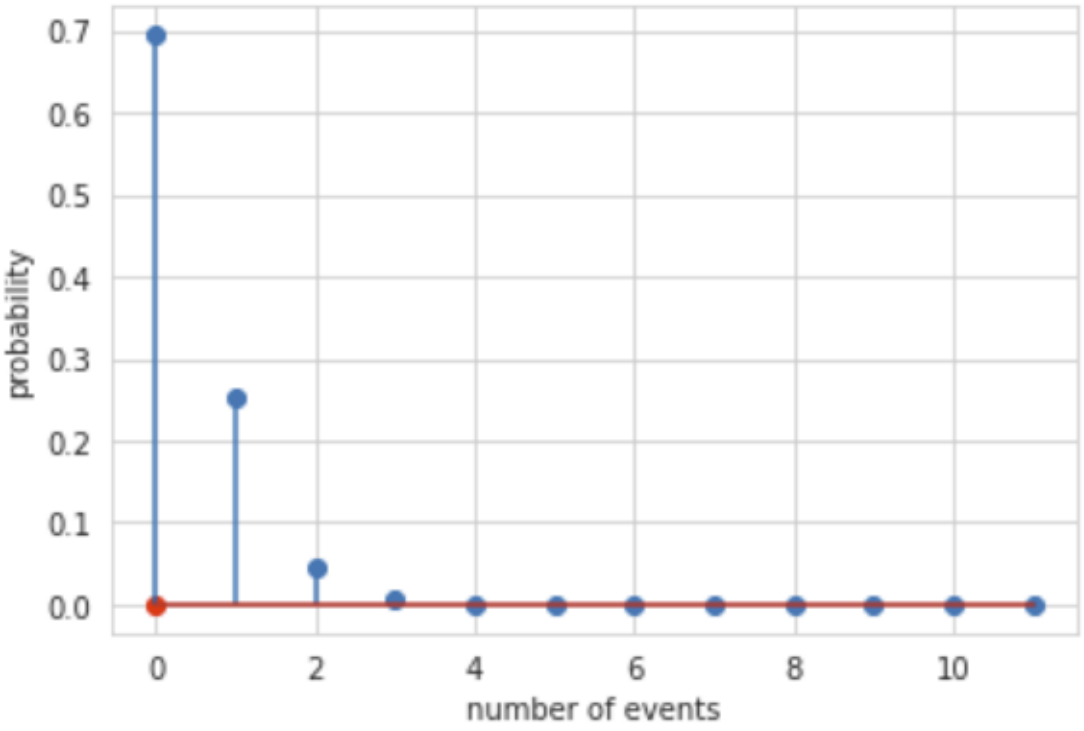
- choose a random predicted rate λ
- then Poisson probability mass function (pmf) can be used to generate probabilities for each number of events. This is calculated using the following equation:

$$p(X = K) = \frac{\lambda^K}{K!} e^{-\lambda}$$

- then an array index at the highest probability is the the most probable count for a given time interval

Below is a graph, which shows probabilities (y axis) for each count of events (axis x), and marks the true observation as a red dot. The predicted λ is 0.365.

Fig. 5.16. Poisson PMF



In the above plot, the highest probability is at the 0 number of events, and the true value is also 0. The probability for obtaining 0 is estimated at 0.694.

Next, to explain this prediction, it is possible to find a corresponding observation in the test data:

Fig. 5.17. Poisson PMF - Test observation

hour	20.00000
cur__precipIntensity	0.79540
cur__apparentTemperature	11.69000
cur__uvIndex	0.00000
is_weekend_day	1.00000

Interpretation:

For 8PM on a weekend, with relatively mild temperature (~12 degrees C) and 0.8 inches of of liquid water per hour (DarkSky), there is a 70% of probability that there will be no objects within an hour, with 25% chance that there will be a single object.

Based on the information gathered so far, error metrics can be already calculated for the estimated mean rates:

Tbl. 5.5. Error metrics - Gaussian Process

Metric	Person-Score
MPD	1.21
MSE	2.35
MAE	0.87
R2	0.33
ACC	0.50

This is interesting as the mean Poisson deviance and mean squared error were further reduced, but the rest of metrics did not benefit from this approach.

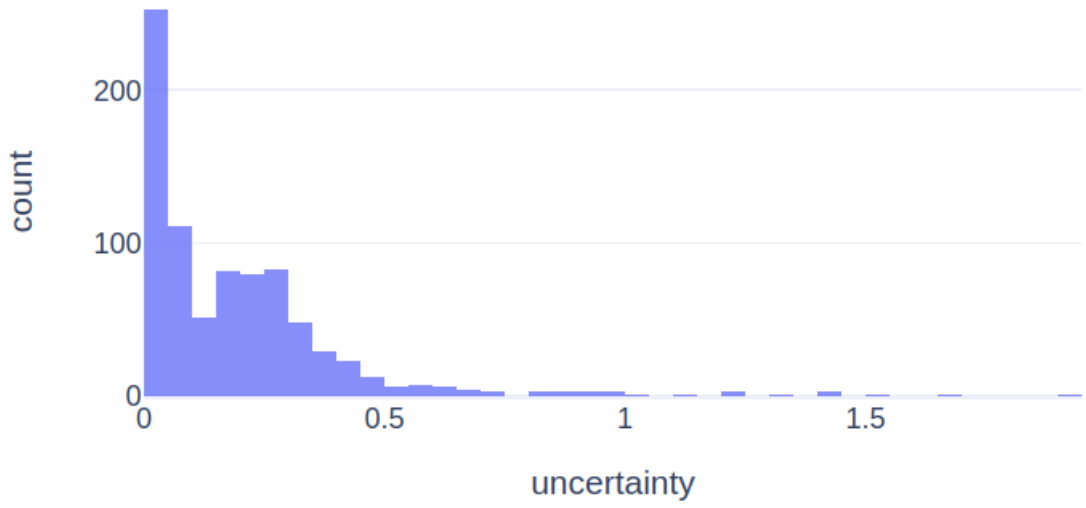
The error distribution is actually very similar to the Gradient Boosted Decision Tree model.

Uncertainty analysis

By calculating Standard Deviation σ across the sampled exponentiated functions, it can be used to measure uncertainty.

The plot below shows the distribution of σ across all predictions:

Fig. 5.18. Uncertainty Histogram



It is clearly visible that there is almost zero uncertainty around some predictions and a little more in others. Overall the average of uncertainty is 0.20.

Plots below show counts and hours in low and high uncertainty scenarios:

Fig. 5.19. Low Uncertainty predictions

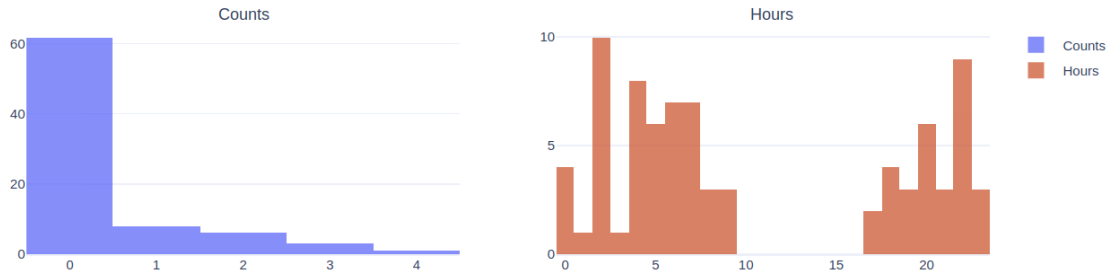
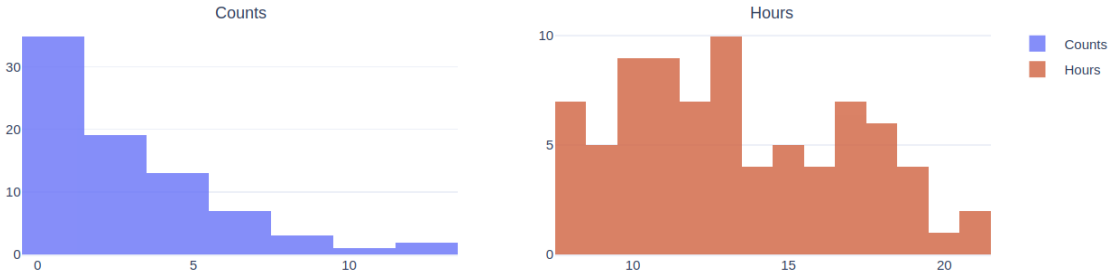


Fig. 5.20. High Uncertainty predictions



Both groups above are using the same number of samples (80) for a meaningful and fair comparison.

It is easy to notice that the predictions during the nightly hours have higher confidence and during the day it is more difficult for the model to be confident.

This is a big advantage of using probabilistic models versus the point estimates generated with sklearn. This idea is explored more below, where uncertainty is embedded in the predictions.

Let the goal be to show to the user a distribution of probabilities for each count ($\{0, 1, 2, ..., 8\}$) in a form of a bar chart.

This distribution can be generated from all sampled rates (500 floating point rates for each record in X_test), and it can be interpreted as embedding uncertainty in the probabilities.

The list of steps to generate such probabilities are listed below:

- choose $N - counts$ for all sampled rates in a single prediction set (where N equals the length of sampled means) using a random choice from a Poisson distribution
- count unique values for each count value
- calculate probability for obtaining individual counts

This *sampling* function is fast and generates 828 predictions in under 0.2 of a second.

These steps are wrapped in the `gen_fcst_probas` function, which returns:

- the most probable count
- numbers, for which probability was at least 0.05
- counts for the numbers, for which probability was at least 0.05

- probabilities > 0.05

Note: Number 0.05 has been chosen arbitrarily to discard very low predictions.

To make it clear, below is an example of a call and response from this function:

```
# let sampled_rates be a 500 sampled rates for a single prediction
expected_count, numbers, counts, probas = gen_fcst_probas(sampled_rates)
```

Output:

- expected_count: 2
- numbers: [0 1 2 3 4 5]
- counts: [246 556 649 520 301 150]
- probas: [0.0984 0.2224 0.2596 0.208 0.1204 0.06]

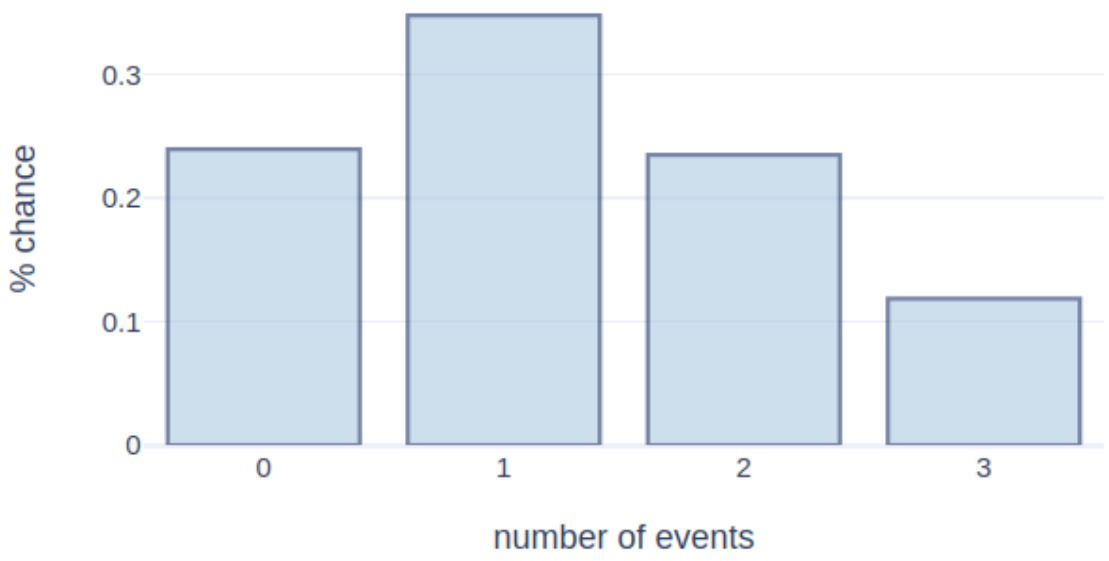
Then, another function is used to visualize this data to the user. Below are the bar charts for three different magnitudes of uncertainty:

Visualization for low σ (0.005):

Fig. 5.21. Low uncertainty predictions

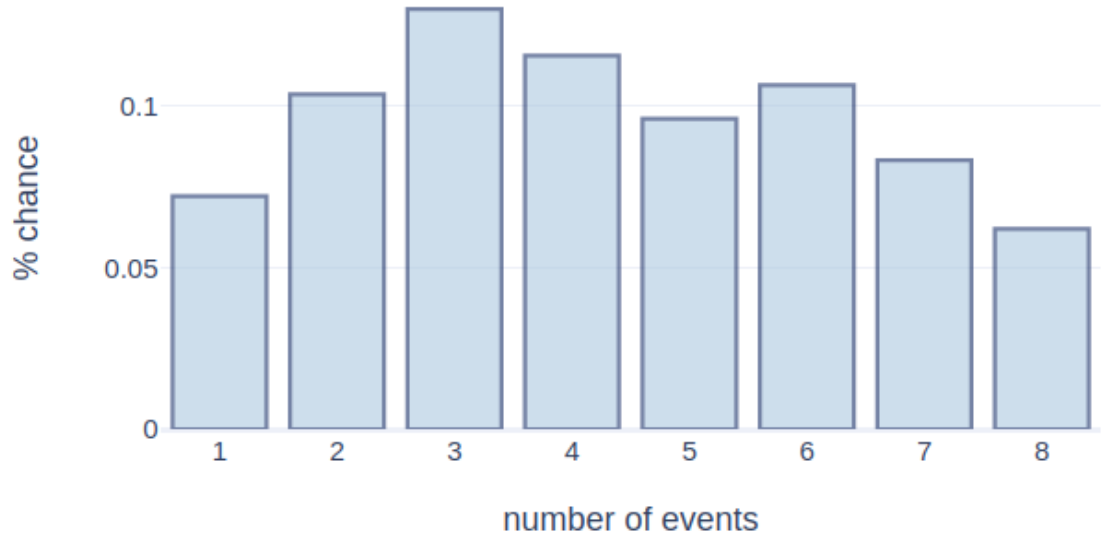
Visualization for medium σ (0.149):

Fig. 5.22. Medium uncertainty predictions



Visualization for high σ (3.295):

Fig. 5.23. High uncertainty predictions



Looking at all the estimations, below is the distribution of number of bars, which would be visible to the users for the predictions:

Tbl. 5.6. Forecast UI - # of bars for predictions

Bar count	# of predictions
1	188
2	141
3	83
4	102
5	140
6	152
7-9	22

Please note that all functions, which were required to calculate count ranges, estimate probabilities, and create plots above, are defined in the Extra Notebook 5.

Other opportunities

Given the mean intensities, it is also possible to answer very useful questions, like:

- What is the probability to see an object at time greater than t given the rate λ?

$$p(t) = exp(-\lambda * t)$$

So if one wants to know probability of a count after 48 minutes past the hour and λ is 1.89, then the result is 22%.

- What is the probability to see an object at time less or equal t given the rate λ?

$$p(t) = 1 - exp(-\lambda * t)$$

So if one wants to know probability of a count in less than 12 minutes and λ is 1.89, then the result is 31%.

This could be very useful in a full blown AI speech-enabled system, where users could ask these questions and get instant answers from the AI, however I am leaving this idea for a future exercise.

5.7. Conclusion

Using object detections, Machine Learning can be used to make future predictions. Evidence above shows that even though the models' performance is far from being always correct, it is significantly better than using a naive approach of average counts per hour.

Model Summary:

In summary, all models described in this Chapter are very robust, all have their benefits and trade-offs, and all ended up with different error rates. Below is a quick comparison of Pros and Cons for each model inside this chapter:

Naive model:

- + simple to understand
- + very robust
- + fast, little training required
- − highest error rate

- − unable to incorporate additional knowledge
- − can not tell uncertainty for predictions

Decision Tree:

- + very explainable ML model
- + fast to train and use
- − does not generalize well for unseen data
- − can not tell uncertainty for predictions

Gradient Boosted Tree:

- + low error rate
- + can handle large data volumes
- + fast to train
- + ability to visualize training progress and potential overfit
- − difficult to understand decisions made
- − can not tell uncertainty for predictions

Gaussian Process:

- + lowest mean squared error and Poisson deviance out of all models
- + easy to explain for statistical-savvy people
- + can tell uncertainty for predictions
- − slow and challenging to train
- − lack of good, modern libraries and limited documentation
- − does not scale well for larger datasets

Metrics:

Below are the error rates for all models in a single table generated for the Person class:

Tbl. 5.4. Error metrics - All models for Person class

Metric	Naive	Decision Tree	Gradient DT	GP
MPD	1.42	1.39	1.25	1.21
MSE	2.81	2.73	2.47	2.35
MAE	0.94	0.92	0.87	0.86
R2	0.27	0.29	0.36	0.33
ACC	0.49	0.53	0.54	0.51

Looking at the metrics, it seems like starting from the Naive model, the metrics have improved until the Gaussian Process, which improves some and reduces other metrics, however since this is a Poisson process, perhaps Mean Poisson Deviance should be the most reflective of the model's performance.

Recommendation:

I think that *Gaussian Process* provided the most useful tools to generate meaningful predictions. It is clear what the model is returning, and error rate is good enough.

Being able to embed uncertainty in the predictions gives this model a distinct advantage.

The next step would be to test the models with another object class (so far the same features have not proven to produce same quality of metrics for the *GP* model) and ultimately decide about the best model to use in the real system.

Next Chapter focuses on the Anomaly Detection problem.

index | prev | next

6. Anomaly Detection

index | prev | next

Motivation:

While initially the goal for this system was set around Object Detection, and then Forecasting, these two concepts were followed by another idea: Can object detections be used to detect anomalies and trigger useful alerts to the users?

This Chapter is an analysis of two methods for detecting anomalies, which I have identified as the most useful:

- based on unusually high count of objects in a given hour
- based on the content of the frames, which look very different to others

6.1. Anomalies estimated from event counts

Flagging object detections as anomalies can be determined by unusually high number of events in a given hour for an object class (like Person, Car, Cat, etc.).

This kind of anomaly detection routine could be run in real time when objects of a specific class are detected in the video stream. System compares a number of already registered objects in an hour (for example 3) versus a threshold (for example 9) to determine if the next object should be classified as anomalous. If that limit is breached, a notification could be triggered to home owner.

These min and max thresholds could be also displayed in the forecast as two bars: one on left and one on the right hand side of the counts to make the alerts easily explainable.

This task forms a univariate outlier detection problem, where system learns the thresholds from the historical counts.

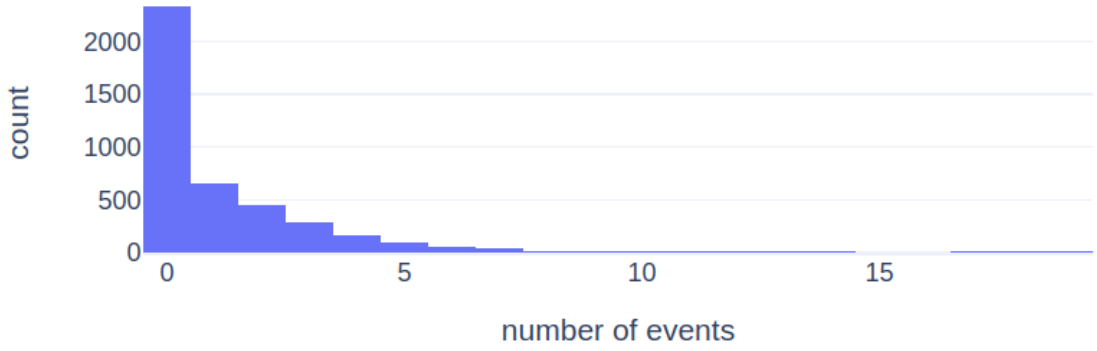
This section is focused around three individual ways of solving this problem:

- IQR (and improved version of IQR)
- Z-Score
- Probabilistic method

Note: Results and plots in this section is based on the *Person* object, but the same method can be applied to any object class.

Below is the count distribution for *Person*:

Fig. 6.1. Count distribution - all hours

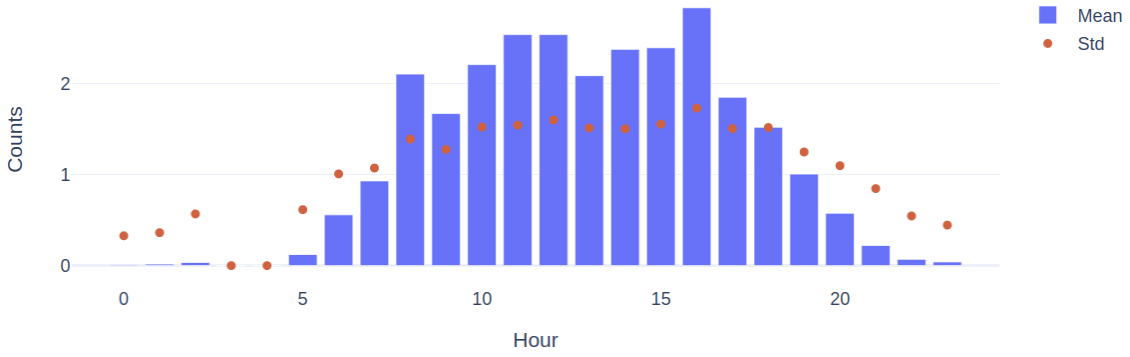


Overall this data is heavily skewed towards 0's, but this is expected. During the night or when it is dark, the number of objects is 0, as the camera does not have the night-vision capability. In other time intervals there is just little activity taking place.

The mean μ of this population is 1.16 and standard deviation σ is 1.95. Taking a square root of σ gives 1.08, which is close to μ and it is one of the main characteristics of the *Poisson distribution*.

Next, looking at the distribution of μ's for each hour shows a more detailed picture, where bars represent mean averages and red dots are a square root of σ.

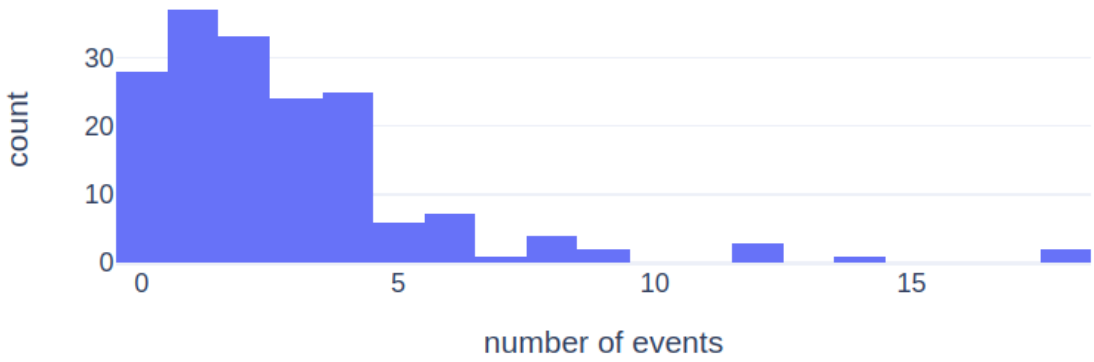
Fig. 6.2. Count distribution by hour



This picture shows quite a large spread of means by hour, and it is sufficiently convincing to focus on anomaly detection for individual hours separately.

Looking at the frequency of counts for 4PM shows quite heavy skewness towards the left side:

Fig. 6.3. Count distribution at 4PM



Note: All code snippets, more in-depth commentary and code for plots created in this section can be found in the corresponding Extra Notebook 6.

6.1.1. IQR

There are many statistical tools to deal with this kind of problem, but arguably the most popular and easy to understand is IQR (Interquartile Range).

This method is used in a very popular plotting technique for outlier identification, the boxplot (Tukey 1977). It is simple to explain, well understood and very often produces satisfactory results.

First, one would calculate an *Interquartile Range (IQR)* using the difference between the third and first quartile, where first ($Q1$) and third quartiles ($Q3$) are the medians of lower and upper halves of the data respectively:

$$IQR = Q3 - Q1$$

Then the lower and upper bounds are calculated by subtracting and adding $IQR * 1.5$ from $Q1$ and $Q3$ respectively:

$$lowerBound = Q1 - (IQR * 1.5)$$

$$upperBound = Q3 + (IQR * 1.5)$$

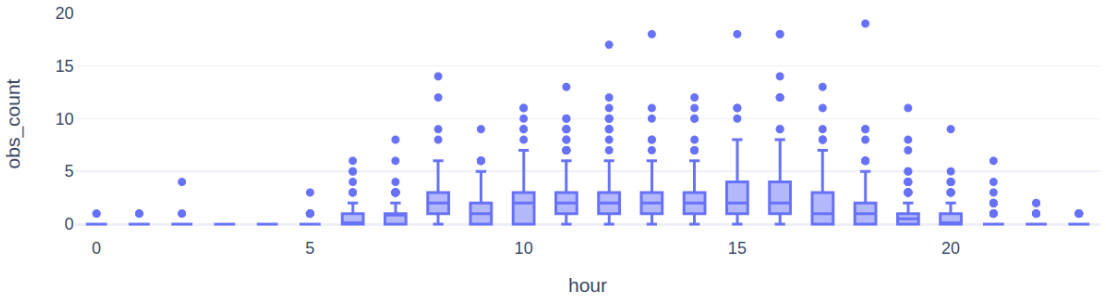
And finally, values below lower or above upper bound are classified is outliers:

$$f(x) = \begin{cases} anomaly, & \text{if } x < lowerBound \text{ or } x > upperBound \\ not - anomaly, & \text{otherwise} \end{cases}$$

, where x is a count of objects in a single hour.

Below is a boxplot for the count dataset by hour:

Fig. 6.4. IQR analysis - boxplot



Looking at the graph, it is flagging a lot of points. After calculating the percentage above and below the bounds, IQR method classifies 5% of observations as anomalous.

The high percentage of anomalies is related to the fact that counts for each hour are heavily skewed, and as mentioned in the *Adjusted boxplot* work by Hubert at al., 2008, boxplots suffer from False Positives in heavily skewed datasets.

6.1.2. Adjusted boxplot for skewed distributions

In the 2008's paper by M. Hubert et al., an alternative method has been proposed to IQR: *An Adjusted Boxplot for Skewed Distributions*.

The procedure is quite similar to IQR, but introduces additional steps:

- calculation of data skewness (called medcouple - MC):

$$MC = \frac{(Q3-Q2)-(Q2-Q1)}{Q3-Q1}$$

For the count dataset and Person object class, this measure is 0.33.

- Then, the lower and upper bounds are calculated as follows:

$$lowerBound = Q1 - h_l(MC)IQR$$
$$upperBound = Q3 + h_u(MC)IQR$$

, where:

$$h_l(MC) = 1.5e^{aMC}$$
$$h_u(MC) = 1.5e^{bMC}$$

The authors of the paper have optimized the values for the constants a and b as -4 and 3, in a way that fences mark 0.7% observations as outliers.

Applying these calculations to the count dataset classifies 148 observations as outliers, which represents 3.5% of the dataset and still generates too many False Positives.

6.1.2. Z-Score

Z-Score determines how many standard deviations σ are points X away from the mean μ.

$$zScore_i = (x_i - \mu) \div \sigma$$

, where x_i is the i-th data point, μ and σ are a sample arithmetic mean and standard deviation respectively.

Z-Score has quite interesting properties when applied to Normal distributions, where 99.7% of the data points lie between +/- 3 σ's.

In case if the skewed count dataset it also performs quite well and identifies 75 outliers, which represents 2% of the dataset, but this is not always the case for skewed datasets.

6.1.3. Probabilistic method

Probabilistic models utilize *Bayesian Theorem* to derive the following formula from the Conditional Probability theory:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Where:

- $P(A|B)$ is the posterior, meaning conditional probability of event A given that B is true
- $P(B|A)$ is the likelihood, also conditional probability of event B occurring given A is true
- $P(A)$ is the prior (information we already know about the data)
- $P(B)$ is the marginal probability of observing event B

There are many benefits from using probabilistic modeling. Some of them are included below:

- no assumptions made about the distribution of the data
- it allows to provide prior information to the model about distributions
- it does not require a lot of data
- it can generate predictions with the uncertainty about them

Prior

Probabilistic programming uses prior information already known (like a distribution of an outcome random variable), then it explicitly calls out the likelihood, which defines how to sample the probability space given the data, and then it performs an analysis of the posterior, which contains N-samples drawn from the distribution.

In relation to the count dataset, I have identified a 2 candidate distributions, which can be used as a prior in the model:

- Half Student T distribution with parameters $\sigma = 1.0$ and $\nu = 1.0$ and density function:

$$f(t) = \frac{\gamma(\frac{\nu+1}{2})}{\sqrt{\nu\pi}\Gamma(\frac{\nu}{2})}(1 + \frac{t^2}{\nu})^{-\frac{\nu+1}{2}}$$

, where ν is the number of degrees of freedom and Γ is the gamma function.

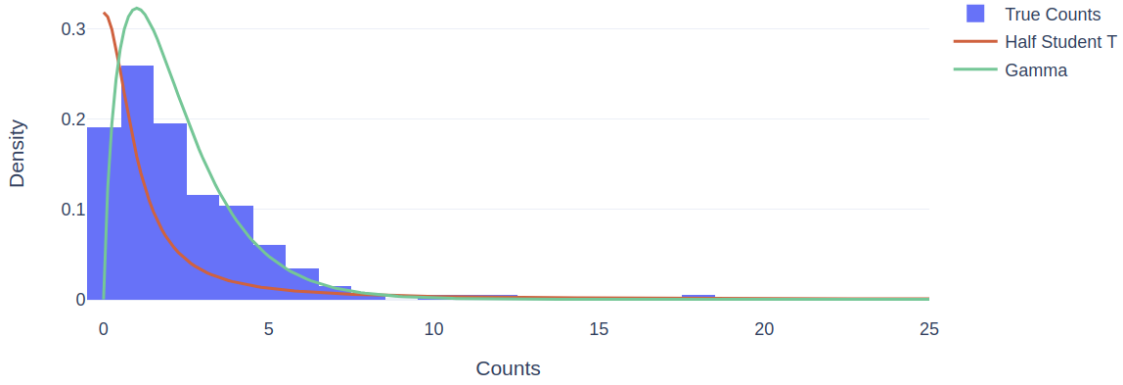
- Gamma distribution with parameters $\alpha = 1.5$ (shape) and $\beta = 0.5$ (rate) and density function:

$$f(x; \alpha; \beta) = \frac{\beta^\alpha x^{\alpha-1} e^{-\beta x}}{\Gamma(\alpha)}$$

, where $x > 0$, $\alpha, \beta > 0$ and $\Gamma(\alpha)$ is the gamma function

Below is the multi-plot with both distributions and the true dataset with counts between 1PM and 3PM:

Fig. 6.5. Priors selection



Based on the graph above, Gamma distribution with $\alpha = 1.8$ and $\beta = 0.8$ seems to be more suitable to this distribution.

Likelihood

The next item needed is the likelihood function, which is used to estimate the counts for every hour, given the data X.

A suitable likelihood function in case of the count data is Poisson, which is given by:

$$L(\lambda; x_1, ..., x_n) = \prod_{j=1}^{n} exp(-\lambda)\frac{1}{x_j!}\lambda^{x_j}$$

As highlighted in the online.stat.psu.edu article, likelihood is a tool for summarizing the data's evidence about unknown parameters, and often (due to computational convenience), it is transformed into log-likelihood.

The log-likelihood for the Poisson process is given by:

$$l(\lambda; x_1, ..., x_n) = -n\lambda - \sum_{j=1}^{n} ln(x_j!) + ln(\lambda) \sum_{j=1}^{n} x_j$$

Now coding up the solution is made very simple with libraries for Probabilistic Programming, like `PyMC3`:

- first define a Gamma prior (it is possible to have a list of 24 priors - one for each hour)
- then define a list Poisson likelihood functions (again, 1 for each hour)
- finally, sample from the posterior and analyze results

Before going into the results, I would like to explain why sampling is used by PyMC3 and which sampling method is appropriate to which kind of data.

In order to compute the optimized values for the model's parameters (also called maximum a posteriori, or MAP), there are two paths to take:

- numerical optimization methods, which are usually fast and easy (`find_map` function in PyMC3)

Default optimization algorithm is BFGS (Broyden–Fletcher–Goldfarb–Shanno, Fletcher, Roger, 1987), but other functions from `scipy.optimize` are acceptable. The downside of this approach is that it often finds only local optima, and as advised in PyMC3 documentation, this method only exists for historical reasons. The second limitation is a lack of uncertainty measure, as only a single value for each parameter is returned.

- sampling based optimization used for more complex scenarios (`sample` function in PyMC3)

This method is a recommended, simulation-based approach with a few algorithms suitable for different problems:

- Binary variables will be assigned to BinaryMetropolis
- Discrete variables will be assigned to Metropolis
- Continuous variables will be assigned to NUTS (No-U-Turn Sampler)

The `sample` function return a `trace` object, which can be queried to obtain the samples for individual parameter. A standard deviation of these values can be interpreted as an uncertainty.

Sampling process for the count data dataset takes less than 30 seconds.

Below are some of the most useful statistics (like mean, standard deviation, median and quantiles) for each hour, which can be easily generated by PyMC3 with the use of `pm.summary` method:

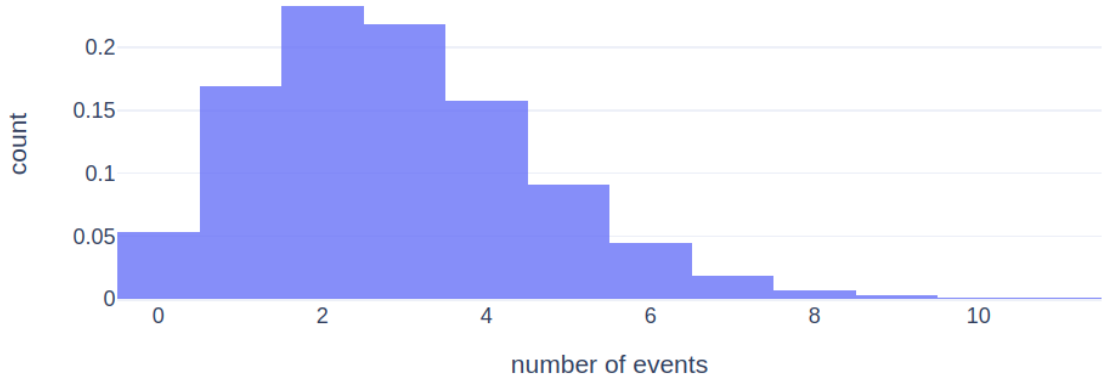
Fig. 6.6. PyMC3 - Posterior stats

	mean	sd	hpd_3%	hpd_97%	mcse_mean	mcse_sd	ess_mean	ess_sd	ess_bulk	ess_tail	r_hat	median_sd	1%	median	99%
lambda_hour_6	0.566	0.058	0.459	0.677	0.000	0.000	16446.0	16187.0	16149.0	5347.0	1.0	0.058	0.439	0.563	0.708
lambda_hour_8	2.105	0.107	1.902	2.303	0.001	0.001	18863.0	18523.0	18984.0	5712.0	1.0	0.107	1.861	2.103	2.366
lambda_hour_10	2.208	0.111	2.012	2.428	0.001	0.001	18257.0	18096.0	18223.0	5601.0	1.0	0.111	1.957	2.207	2.476
lambda_hour_12	2.535	0.121	2.321	2.775	0.001	0.001	15633.0	15567.0	15552.0	5600.0	1.0	0.121	2.259	2.532	2.824
lambda_hour_14	2.375	0.115	2.162	2.594	0.001	0.001	18240.0	17758.0	18434.0	5442.0	1.0	0.115	2.115	2.374	2.653
lambda_hour_16	2.831	0.127	2.590	3.068	0.001	0.001	18484.0	18115.0	18665.0	5830.0	1.0	0.127	2.545	2.829	3.131
lambda_hour_18	1.525	0.092	1.357	1.703	0.001	0.001	16952.0	16512.0	17063.0	5052.0	1.0	0.092	1.322	1.523	1.745
lambda_hour_20	0.583	0.058	0.479	0.693	0.000	0.000	17644.0	17399.0	17306.0	5520.0	1.0	0.058	0.460	0.581	0.726

Next, one can take an advantage of having multiple samples for the estimated rate λ, and generate N counts for all these rates (using all sampled rates for a single hour embeds the uncertainty into the generated counts).

Probability density for the 4PM then can be plotted and questions asked about the probability of obtaining a count K:

Fig. 6.7. Probability density for 4PM



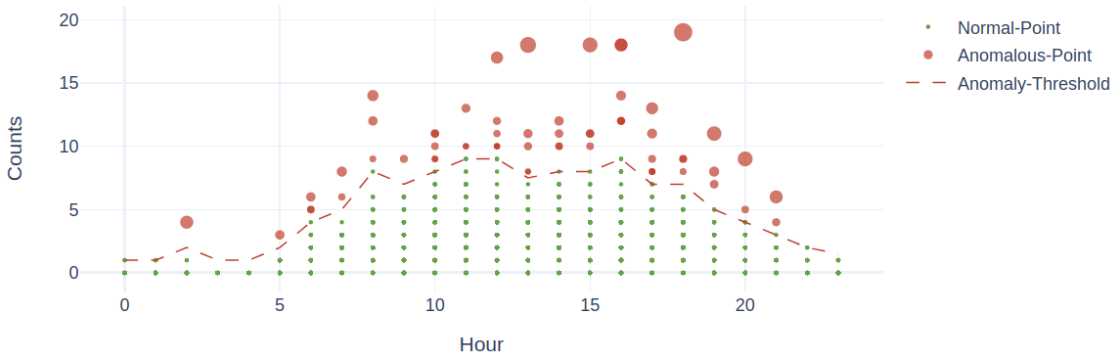
For example, if $K = 8$, then there is 1% chance to see an 8 or more objects at 4PM.

Now, the final idea here is to define a percentage of observations, which need to be classified as anomalies, and use the approach from above, but this time apply it to all hours.

Once this is set to 0.001, 61 anomalous observations are detected, which represents 1.5% of the dataset. The result is the max count fence for each hour, above which counts become anomalies. For example the fence for 4PM has been determined as 9.0.

To visualize the results for all hours I have generated a plot, with a dashed line representing the threshold for anomalies. Red dots are anomalies, and their size corresponds to their magnitude:

Fig. 6.8. Anomaly thresholds by hour



6.1.4. Summary

Probabilistic Model is the most flexible and gives the most interesting opportunities based on the generated samples. The percentage of anomalies is fully under control, and can be made as strict, or as relaxed as required.

The next step in this section would be to actually embed the max-count fences in the hourly forecast graph, so it is clear for the users of the system when to expect an alert for each hour, and why an alert has been triggered.

It would be also interesting to calculate the fences using the rates estimated by Gaussian Process in the Forecasting Chapter. This way the fence would be tailored to the specific scenario (like weekend, rainy day, morning time for example).

6.2. Anomalies estimated from raw images' content

The aim of this section is to investigate if patterns encoded in raw images can identify anomalies.

Analysis below will be conducted with two goals in mind: - There could be a process running in real time, which would tell the difference between the normal and unusual images, and based on that, it could send notifications to the users about suspicious activities - On average, object detector identifies roughly 2000 images each day. It is very tedious to scan through all of them every day. If there was a score, which could be used to sort images by the "most different ones", the manual process could be somewhat eliminated

In the *Deep Learning for Anomaly Detection: A Survey* paper (Chalapathy 2019), the author mentions that even though the fields of Anomaly Detection and Deep Learning have been well exploited by the researchers individually, these two areas are not linked enough and there are a lot of opportunities to explore.

Finding anomalies based on the camera frames can be framed as an *unsupervised Machine Learning* problem, where a model is trained to learn patterns encoded in the images through noisy reconstruction of the inputs. Then when a new (original) image is passed through this model, an error between the reconstructed and original image can be used to classify images as *normal* or *anomalous*.

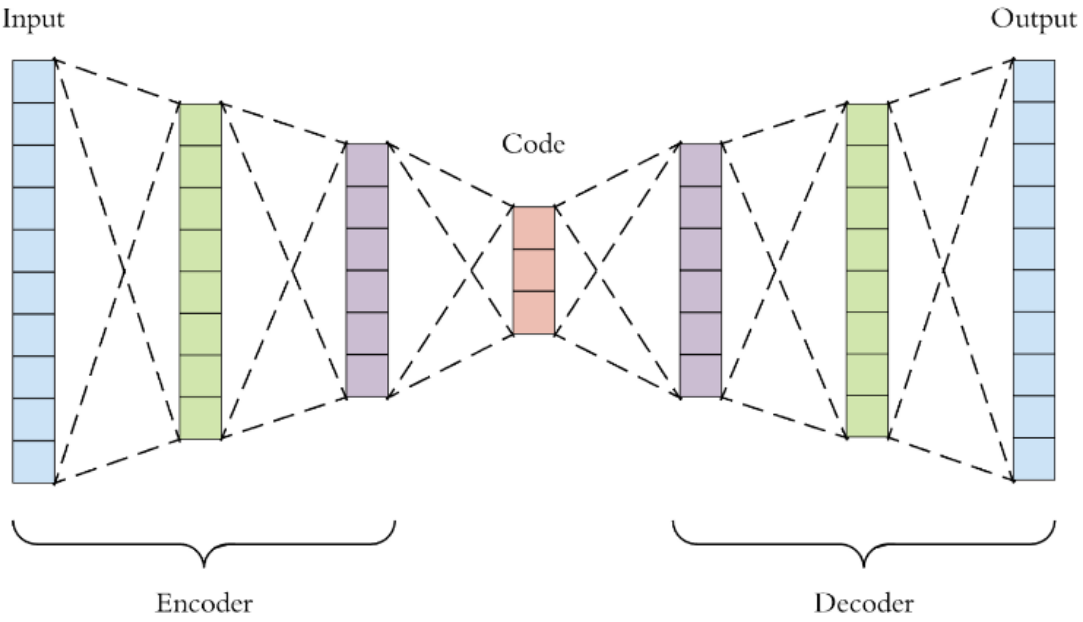
Using unsupervised learning has been chosen mainly for two reasons: - there are over $600K$ images collected in the process and they do not contain any anomaly-related labels - anomaly detection usually deals with highly imbalanced datasets (where potentially less than 1% represents anomalous images), but supervised models tend to prefer balanced datasets

The usage of *Deep Learning* is mostly motivated by the fact, that Deep Neural Networks can efficiently deal with large scale datasets (i.e. image data) and they can use GPU to significantly decrease parameter optimization time.

The Neural Network models, which learn how to reconstruct their own inputs are called *Auto Encoders*. Outside of anomaly detection, they are also used for data compression and noise removal.

I am using a Convolutional Auto Encoder in this section as CNNs are often selected for as complex datasets as image data (they have been previously used in Chapter 4 for Yolo object detections).

Fig. 6.9. Auto-encoder diagram



Please refer to the Literature Review chapter for more theoretical aspects around auto-encoders.

In contract to sections 6.1. above, auto encoders will use the data for all object classes combined, as there is no need to do it separately for different object classes.

Note: All code snippets, more in-depth commentary and code for plots created in this section can be found in the corresponding Extra Notebook 7.

6.2.1. Computer Vision and image pre-processing

Starting point for this section are all object detections created in Forecasting Chapter, Section 5.1.:

Fig. 6.10. Detections tabular data

label	confidence	x1	y1	x2	x2	filename	date_time
car	0.523175	298	7	426	426	07.02.40.270_34c99836_car-car-car.jpg	2019-09-09 07:02:40.270
person	0.759682	489	31	518	518	12.02.42.921_ea6c9143_person-bicycle.jpg	2019-09-09 12:02:42.921
bicycle	0.532076	444	54	484	484	12.02.42.921_ea6c9143_person-bicycle.jpg	2019-09-09 12:02:42.921
person	0.864749	463	55	537	537	07.30.02.409_c5662b14_person-car-car.jpg	2019-09-09 07:30:02.409
car	0.859297	302	23	410	410	20.26.56.841_4ba2f42d_car.jpg	2019-09-09 20:26:56.841

This dataset contains the folders and filenames of the collected images.

Raw images can be very useful for many purposes (like forensic investigations, automated alerts, or simply historical value), but they need some form of pre-processing before they can be used for Machine Learning.

Computer Vision is a vast area of Artificial Intelligence, which provides plethora of guidelines and solutions for image processing and image content analysis.

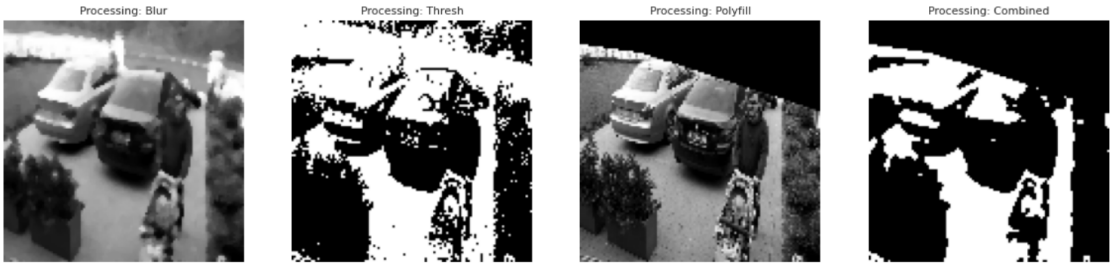
I have defined a function called `process_frame`, which allows to perform morphological operations on images using flags as function arguments. It was partially inspired by a post on PyImageSearch (PyImageSearch 2017):

- convert to gray scale
- apply Median blur
- apply Thresholding
- crop ROI using polyfill

Here is an example of an original image, and below are the morphological operations applied to it:

Fig. 6.11. Original image

Fig. 6.12. Pre-processed image



As the dataframe at the start only contains a list of all images stored on the disk, they need to be extracted into a data.

The procedure for extracting the raw image content consists of the following steps:

- pre-allocate memory in two numpy arrays: one for the image data and one for corresponding filenames
- take a sample of images (folders and filenames) from the dataframe
- iterate through the sampled dataframe and for each record:
  - open an image from the disk
  - process image using `process_frame` function by using provided pre-processing parameters
  - add image data (as a numpy array) and corresponding filename to the numpy arrays

Image preprocessing steps above took 1 minute and 40 seconds.

In terms of the resolution of the images, the higher it is, the more detail is available for the Deep Learning model, but it comes at a cost: A single gray-scale 28×28 pixel image generates a record with 784 features and an image with the size 112×112 results in $12,544$ features respectively.

Since the original frames are captured in $Full - HD$ $(720 \times 1280 \times 3)$, without resizing they would contain $2,764,800$ features per image.

Based on my experiments, increasing size to more than 608×608 tends to cause issues with the GPU memory and it actually degrades the performance of the model, as it learns too much noise.

Another issue with the high image quality is the speed of image pre-processing and model training: it can be a difference between minutes and hours on a single model training.

For all the above reasons I am only considering image size 56×56 in the rest of this Notebook. It is a good trade-off between too small and too large. I am also choosing to use $10K$ images for training, as it is the lowest number of images, which produces best final results.

The statistics from using different numbers of images and resolutions are provided in the Section **??**.

Next step is to add an extra dimension to the dataset, as Neural Network will expect the shape of (height, width and depth). The depth is needed in case if color images were used.

Normalizing the data on a $0 - 1$ scale helps with training stability. Since image data is a `uint8` type, it can take values only between 0 and 255, so dividing all values by 255.0 is the standard normalization step for image data (data type becomes a `float32` type):

$$normalize(X) = X/255.0$$

And the last step here is to split the dataset into train and test splits. Unfortunately due to slow training times it is not advised to run cross validation splits for Deep Neural Networks. I am choosing a 0.8/0.2 split with a `random_state` parameter set to a constant value for reproducibility.

The shape of the training data, which is now ready to be used in an auto encoder Neural Network is $(8000 \times 56 \times 56 \times 1)$.

6.2.2. Training with auto encoder

The architecture of a convolutional auto encoder below is inspired by a blog entry on PyImageSearch (PyImageSearch 2020). The model is built on top of Keras functional API.

$$autoEncoder = decoder(encoder(X))$$

- Encoder:
  - Input: X_train
  - Convolutional layer: 32 and 64 filters, each followed by Leaky ReLU and Batch Normalization:
    - changing the size of filters or adding/removing filters have decreased the performance
    - the difference between ReLU and Leaky ReLU activations is very small, but Leaky ReLU seems to be a little more robust. I have concluded that letting some weights to be slightly negative makes a difference
  - Output: Latent (bottleneck) layer with 16 nodes by default. Experiments with different sizes (8 and 32) did not make any improvements, where 8 nodes has decreased the performance by around 10%
- Decoder:
  - Input: Latent layer
  - Convolutional transpose layer: 32 and 64 filters, each followed by Leaky ReLU and Batch Normalization
  - Single CNN layer: this layer is used to recover the original depth
  - Activation: Sigmoid is used to make sure output values match the input range (between 0 and 1)
- Optimizer: Adam with learning rate α and decay $decay = \alpha \div nEpochs$
  - Switching optimizers to Stochastic Gradient Descent or RMS-Prop did not improve the model's performance
- Loss function: mean squared error is used as a simple and well understood error function, which will be used below to identify anomalies

I am leaving out the theory behind the Neural Network layers, as it would inflate this research significantly. It is constantly repeated across many papers and articles and would not make this section more useful. Hopefully the intuition, which is provided above is more valuable.

After many experiments with different model parameters, I have concluded that this architecture is quite optimal in its original shape:

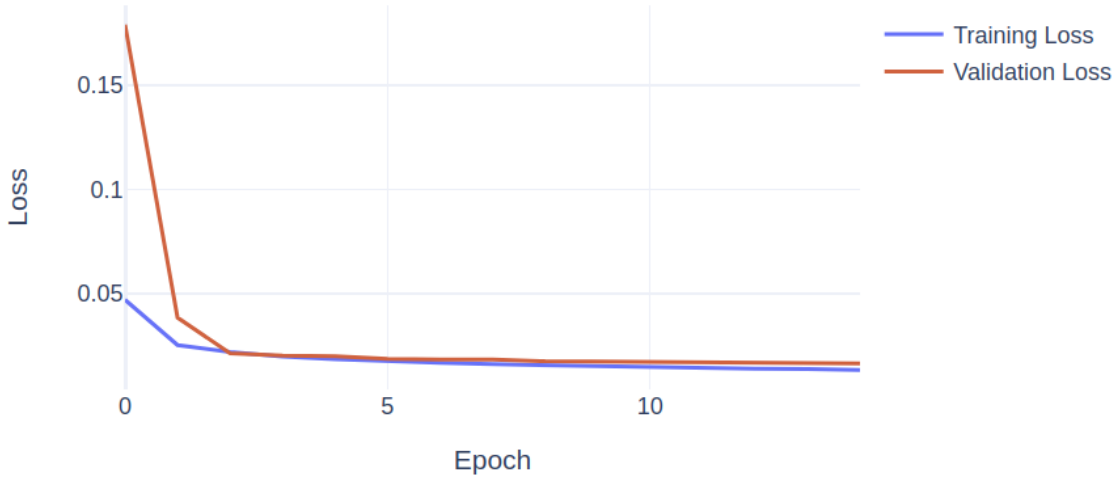
Fig. 6.13. Auto Encoder summary

```
Model: "autoencoder"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_19 (InputLayer)        [(None, 56, 56, 1)]       0
_________________________________________________________________
encoder (Model)              (None, 16)                219920
_________________________________________________________________
decoder (Model)              (None, 56, 56, 1)         269313
=================================================================
Total params: 489,233
Trainable params: 488,849
Non-trainable params: 384
```

Given the CNN-based architecture above and 56×56 image size, the model has almost $500K$ trainable parameters. This number goes up to $1.7M$ for 112×112 images.

Model converges well without any symptoms of overfitting, and only requires 10 epochs of training with $2s$ per epoch and ends with training loss 0.0135 and validation loss 0.0167:

Fig. 6.14. Auto Encoder loss



Notes:

- when $25K$ images are used for training, instead of $10K$, the curves are much smoother and model converges after 30 epochs (however it does not improve the final results, so I have discarded that model)
- it is important that the model generates some error and does not memorize the whole training set, as this kind of model would not be useful at all to detect anomalies

Below is a table, which builds an intuition around the number of epochs to converge and time it takes for each kind of sample size and image resolution:

Tbl. 6.1. Auto encoder - convergence statistics

Sample Size	Res.	Sec. Per Epoch	Epochs to Converge
10,000	56 x 56	2	10
25,000	28 x 28	3	20
25,000	56 x 56	6	30
25,000	112 x 112	15	50
50,000	28 x 28	5	50
50,000	56 x 56	10	50
50,000	112 x 112	28	50

As per expectation, the more samples and the higher the resolution, the longer it takes to train each epoch and more epochs is required to converge.

6.2.3. Model evaluation on test-set

The test-set contains $2K$ 56×56 preprocessed, grayscale images, normalized to $0 - 1$ range.

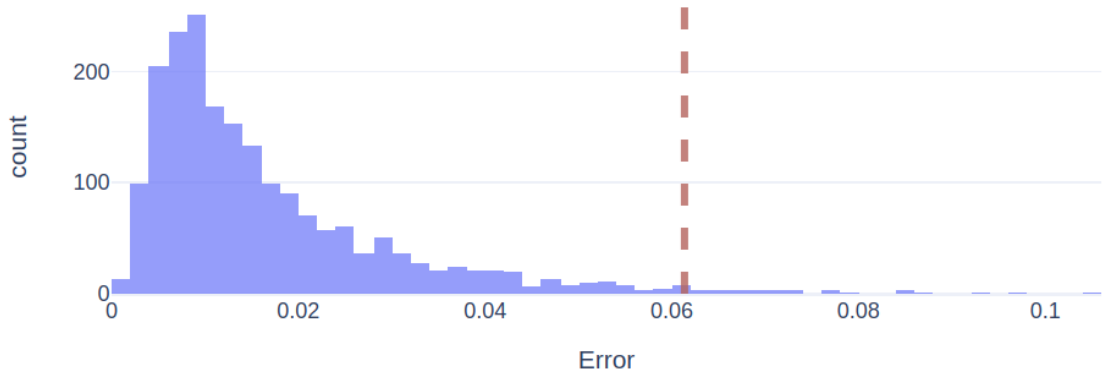
When predictions are generated using Keras `predict` method, their values are compared against the original images and errors are calculated using *mean squared error* statistic (any suitable error measurement can be actually utilised here).

Making predictions for all 2000 test images and calculating errors takes only 0.25 of a second.

Then a decision needs to be made to determine a percentage of the dataset, which should be classified as anomalous. For example let this percentage be 0.99.

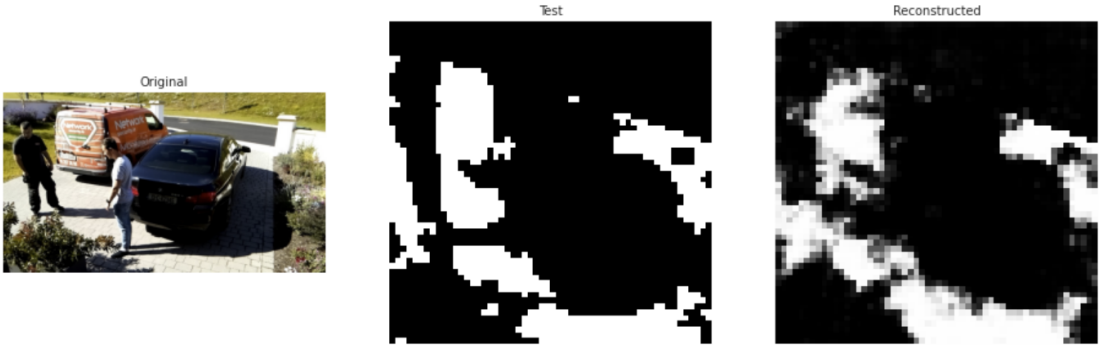
Then it is possible to calculate a threshold for the error using 99th quantile, above which points are classified as anomalies. This threshold value is 0.0651. Below is a histogram, which shows the distribution of errors with a red dashed line, representing the calculated threshold:

Fig. 6.15. Auto Encoder loss



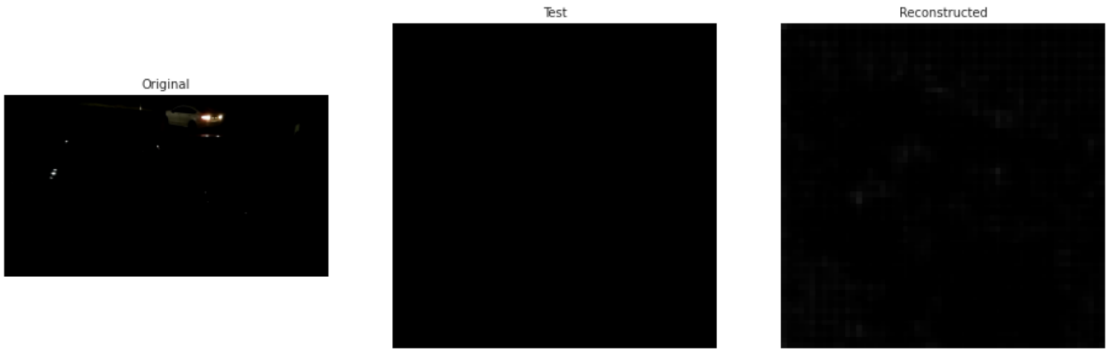
To make sure the method is working, it can be very helpful to look at the images with the largest error (they would be classified as anomalous), an example is below, which shows the original image (left), with the preprocessed (middle) and reconstructed one by auto encoder (right):

Fig. 6.16. Auto encoder - anomalous image



And looking at an image with the lowest error should display a normal frame:

Fig. 6.17. Auto encoder - normal image



Seems like it is working for the anomalous image (as there is a lot going on in that frame and this kind of event should be flagged as an anomaly).

But it is somewhat surprising in case of the normal image. The first 8 normal images are very dark and contain a lot of pixels with a value of zero, so reconstructing those is much easier for the model and therefore the error will be much lower as well.

So, what can be said about the output of this procedure?

I think it has the potential, but it most likely needs some improvements in the data collection stage, and perhaps more computer vision preprocessing steps (for example one of the images with highest error was flagged due to the dry patches of otherwise wet surface, and perhaps more sensitivity is required when it is dark).

Finally the training procedure needs to be carefully crafted. Instead of training the model on the

whole data, a better idea would be to train on a few rolling months. This would help if the region of interest changes over time (people change their cars or plant new trees, or even move the camera to another location).

6.2.4. Model evaluation on hand-labeled data

The very last step in terms of auto-encoder analysis is a test with labeled images. This is very helpful, as it allows to generate metrics to compare several various models analytically.

For this purpose I have manually annotated 30 images:

- 15 as non-anomalous
- 15 as anomalous

Procedure:

The process is almost the same as error calculation in the previous step (I have defined a helper function called `test_anomalies` for this task):

- load images from the disk (non-anomalous and anomalous samples reside in their respective 0 and 1 folders)
- pre-process images and reshape data
- run prediction through auto-encoder Keras model
- calculate mean squared errors
- establish if anomalies are found based on the previously calculated threshold (0.0651)
- show images (optionally)
- plot errors from auto-encoder (optionally)
- return anomaly boolean flags for each image

Evaluation metrics:

Now the labels are available and standard classification metrics to evaluate model's performance can be utilized (Stackabuse, 2020):

- Accuracy

Accuracy measures a percentage of correct predictions out of all predictions:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

where TP are True Positives, TN are True Negatives, FP are False Positives and FN are False Negatives

- Precision

Precision tends to be a good metric when the cost of False Positives is high. Out of all the predicted positive instances, how many were predicted correctly:

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

- Recall

Contrary to Precision, Recall is a useful metric when the cost of False Negatives is high. Out of all the positive classes, how many instances were identified correctly:

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

- F1 Score

F1 Score is a mixture of Precision and Recall in a single metric (when classifying 0's and 1's correctly are both equally important). F1 Measure is also called a harmonic mean of Precision and Recall:

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

In the case of anomaly detection, accuracy tends to be a poor measure due to the dominance of the *Normal* observations. Precision is also not the most useful metric, as the cost of falsely classified observations as an anomalies tends to be less detrimental than not catching them at all.

As a result of the above statements, *Recall* is the most important metric to observe and optimize for, while keeping an eye on the F1 Score to not sacrifice too much on the other type of errors.

Below is the plot showing previous error distribution with the red dashed line representing the anomaly threshold, and a set of short red and green lines showing anomalous and normal predictions respectively:

Fig. 6.18. Auto encoder - hand crafted images classification

The perfect score would put all green lines on the left side of the threshold (red dashed) line and all red short lines on the right hand side.

The plot shows 6 anomalous images misclassified out of 15.

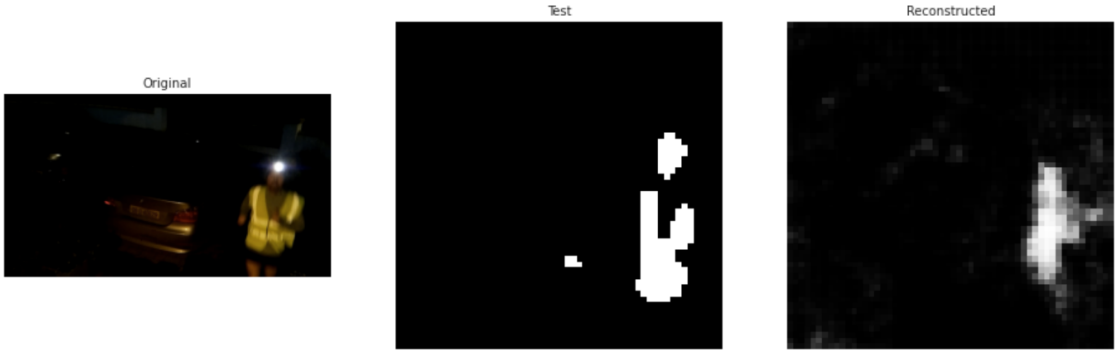
Below are the metrics for using the current model, trained previously:

- acc: 0.77
- prec: 0.9
- rec: 0.6
- f1: 0.72

Most misclassified images can be explained by not enough variety against what is seen as a normal frame. A potentially good improvement would be to apply some computer vision to detect when

it is dark, and increase error sensitivity during nightly hours. Below is an example of anomalous image, which has been classified as normal:

Fig. 6.19. Auto encoder - misclassification



6.2.5. Summary

For the reference, below is a table with the statistics collected for all types of models trained as part of this exercise (with the model names representing the number of training samples, resolution and pre-processing parameters), sorted by highest Recall value:

Fig. 6.20. Auto encoder - model selection - metrics

model_name	acc	prec	rec	f1
nsamples25000_res56_blurTrue_threshTrue_polyTrue	0.766667	0.900000	0.600000	0.720000
nsamples10000_res56_blurTrue_threshTrue_polyTrue	0.766667	0.900000	0.600000	0.720000
nsamples10000_res56_blurTrue_threshTrue_polyTrue_RELU	0.766667	0.900000	0.600000	0.720000
nsamples25000_res56_blurTrue_threshFalse_polyTrue	0.500000	0.500000	1.000000	0.666667
nsamples10000_res56_blurTrue_threshTrue_polyTrue_RELU_LESS_CONV_FILTERS	0.733333	0.888889	0.533333	0.666667
nsamples50000_res112_blurTrue_threshTrue_polyTrue	0.700000	0.800000	0.533333	0.640000
nsamples50000_res28_blurTrue_threshTrue_polyTrue	0.700000	0.875000	0.466667	0.608696
nsamples2000_res56_blurTrue_threshTrue_polyTrue	0.700000	0.875000	0.466667	0.608696
nsamples10000_res56_blurTrue_threshTrue_polyTrue_RELU_NO_LR_DECAY	0.700000	0.875000	0.466667	0.608696
nsamples25000_res56_blurTrue_threshTrue_polyFalse	0.466667	0.478261	0.733333	0.578947
nsamples100000_res28_blurTrue_threshTrue_polyTrue	0.666667	0.857143	0.400000	0.545455
nsamples5000_res56_blurTrue_threshTrue_polyTrue	0.633333	0.833333	0.333333	0.476190

6.3. Conclusion

It turned that that collected data does contain some anomalous signals, which can be exploited by statistics, machine learning and computer vision.

Both methods from sections 6.1. and 6.2. are only some examples of what can be done with object detection data from anomaly detection perspective and readers are certainly encouraged to think about their use cases.

In section 6.1. it was very easy to fall into a pitfall of *IQR* method, but since it is not suitable for skewed datasets - a much more flexible approach has been developed using probabilistic programming and Poisson distribution characteristics.

Then paragraph 6.2. showed promising capabilities for real time image scoring using auto-encoders.

There are many other techniques, which I am planning to explore in the future, like using the actual forecast from chapter 5 in section 6.1., and using a Variational Auto Encoder in 6.2.

The next chapter contains the final conclusion surrounding this research as a whole and more call-outs for future opportunities and improvements.

index | prev | next

7. Conclusion and Future Considerations

index | prev | next

This is the final conclusion section for this project.

I believe that all three research questions from the Introduction have been answered:

- How complex is it to build a fast and reliable object detection pipeline using Computer Vision?

A reliable data collection stage was very complex. Collecting images 24/7 for 6 months straight has challenged my problem solving skills *a little more* than initially anticipated. Placing the camera in a right location alone required wiring the house and getting a smooth experience in terms of frames per second, while performing multiple tasks on each frame required many iterations.

- Given collected data with object detections, can future object counts be predicted using Machine Learning?

Future object counts can be predicted using Machine Learning, but not without a significant amount of data cleaning and processing, and thousands of models have been tested in the process. The benefit for me is the expertise in training various types of models (for example Bi-Directional Long-Short Term Memory Recurrent Neural Networks), but ultimately I have chosen the most useful model, which not only gives the predictions but also their uncertainty.

- Does object detections data contain anomalous signals, which can be recognized with Anomaly Detection algorithms and used for alerts to the users?

It is possible to apply anomaly detection algorithms to the collected data and find out how many objects of certain type within an hour is too many, and which frames are too different from others and should be flagged as anomalies.

Below are some recommendations for future improvements:

- Putting it all together into a working product could lead to other ideas and opportunities and could expose potential gaps to tweak.
- Using a professional camera with a wider field of view, night vision capability or waterproof case would make it possible to mount it in a more inaccessible location while improving object detections.
- Exploring Object Detection on-device instead of streaming frames to a central PC could significantly reduce the cost of the setup. While Raspberry Pi itself is not powerful enough for this task, there are advances in the area worth an exploration (for example NVidia Jetson Nano or Google Coral).
- Testing the system in another house. It is an interesting research question to assess if the models developed in this research are truly generalizable.
- Privacy mode should be incorporated. People in the detected images should be blurred to hide their identity. This is an easy Computer Vision task and an important subject in terms of ethics and how modern AI systems should work.
- More data. Collecting data for a year or two years of data creates new possibilities to use more suitable forecasting models with periodicity component and ability to observe spikes and dips over the full year.
- Anomaly detection can be also improved. Using forecast predictions to find hourly anomalies, and employing a more advanced versions of the Convolutional auto encoder would be worth to try too. Also, perhaps there should be an alternative method for dealing with the nightly observations.

All the notes above show plethora of opportunities for future research. That, with constant releases of new libraries, like *Yolo v4* for object detection, *pymc4* (in beta as of July 2020 finally using *TensorFlow*) for probabilistic programming or *RTSP* streams for standardized, smoother FPS, the system could be made more accurate, fast, modern and ultimately useful.

8. References

index | prev | next

Machine Perception (Roberts 1963)

https://dspace.mit.edu/bitstream/handle/1721.1/11589/33959125-MIT.pdf?sequence=2&isAllowed=y

Alexnet 2012 (Krizhevsky 2012)

https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

Distinctive Image Features from Scale-Invariant Keypoints (Lowe 2004)

https://www.cs.ubc.ca/~lowe/papers/ijcv04.pdf

Support Vector Machines (Vapnik et al., 1995)

http://image.diku.dk/imagecanon/material/cortes_vapnik95.pdf

Burglary report (alarmnewengland 2020)

https://www.alarmnewengland.com/blog/how-to-tell-if-burglars-are-casing-your-home

Improved Adaptive Gaussian Mixture Model for Background Subtraction (Zivkivoc 2004)

https://ieeexplore.ieee.org/document/1333992

Yolo v1 (Redmon et.al. 2015)

https://arxiv.org/pdf/1506.02640.pdf

Yolo v2 (Redmon et.al. 2016)

https://arxiv.org/abs/1612.08242

R-CNN (Girshick et al. 2013)

https://arxiv.org/abs/1311.2524

Fast R-CNN (Girshick 2015)

https://arxiv.org/abs/1504.08083

Faster R-CNN (Ren et al., 2015)

https://arxiv.org/pdf/1506.01497.pdf

VGG-16 (Simonyan et al. 2014)

https://arxiv.org/abs/1409.1556

Learning non-maximum suppression (Hosang et al., 2017)

https://arxiv.org/abs/1705.02950

Googlenet (Shegedy et al. 2014)

https://arxiv.org/abs/1409.4842

Yolo V3 (Redmon et al. 2018)

https://pjreddie.com/media/files/papers/YOLOv3.pdf

Yolo V4 (Bochkovskiy et al. 2020)

https://arxiv.org/abs/2004.10934

Regression Trees (Morgan et al., 1963, p. 430)

http://cda.psych.uiuc.edu/statistical_learning_course/morgan_sonquist.pdf

Modern Bias-Variance Tradeoff (Neal et al., 2018)

https://arxiv.org/abs/1810.08591

LightGBM (Ke et al., 2017)

https://papers.nips.cc/paper/6907-lightgbm-a-highly-efficient-gradient-boosting-decision-tree

Greedy Function Approximation: A Gradient Boosting Machine (Friedman 1999)

https://statweb.stanford.edu/~jhf/ftp/trebst.pdf

Stat-Quest Gradient Boosting Tree tutorial (Starmen 2019)

https://statquest.org/video-index/

Central Limit Theorem (Laplace 1810, Fischer 2011)

http://www.medicine.mcgill.ca/epidemiology/hanley/bios601/GaussianModel/HistoryCentralLimitTheorem.pdf

On the numerical resolution of systems of linear equations - Cholesky decomposition (Cholesky 1910)

http://bibnum.education.fr/mathematiques/algebre/sur-la-resolution-numerique-des-systemes-d-equations-lineaires

Gaussian Distributions (peterroelants.github.io Blog)

https://peterroelants.github.io/posts/multivariate-normal-primer/

Gaussian Process (Wikipedia, 2020)

https://en.wikipedia.org/wiki/Gaussian_process

AutoEncoders (towardsdatascience 2017)

https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798

Deep Learning Book, AutoEncoders (Goodfellow-et-al-2016)

http://www.deeplearningbook.org/

@book{Goodfellow-et-al-2016, title={Deep Learning}, author={Ian Goodfellow and Yoshua Bengio and Aaron Courville}, publisher={MIT Press}, note={http://www.deeplearningbook.org}, year={2016} }

Mahalonobis Distance (machinelearningplus 2019)

https://www.machinelearningplus.com/statistics/mahalanobis-distance/

All models are wrong (Box 1976)

https://en.wikipedia.org/wiki/Journal_of_the_American_Statistical_Association

Adjusted boxplot (Hubert et al., 2008)

https://wis.kuleuven.be/stat/robust/papers/2008/adjboxplot-revision.pdf

Exploratory data analysis (Tukey 1977)

https://archive.org/details/exploratorydataa00tuke_0

BFGS (Fletcher, Roger, 1987)

https://archive.org/details/practicalmethods0000flet

Deep Learning for Anomaly Detection: A Survey (Chalapathy 2019)

https://arxiv.org/abs/1901.03407

PyImageSearch - OCR Blog-post (PyImageSearch 2017)

https://www.pyimagesearch.com/2017/07/10/using-tesseract-ocr-python

PyImageSearch - Anomaly detection (PyImageSearch 2020)

https://www.pyimagesearch.com/2020/03/02/anomaly-detection-with-keras-tensorflow-and-deep-learning/

Classification metrics (Stackabuse, 2020)

https://stackabuse.com/understanding-roc-curves-with-python/

9. Acknowledgements

I would like to thank all the people involved in this work. This research is a compilation of my own work and experiments, indispensable feedback from my university supervisor and my peers. It stands on the shoulders of Machine Learning and Computer Vision giants, which made it all possible.

I would like to thank Alessio Benavoli for all his guidance, my friends from work (especially Phil O'Mahony) for priceless tips in Data Analysis, Forecasting and Anomaly Detection, and my closest family for understanding when I was locked in the study room for weeks.

10. Appendices

Below are the links to all Extra Notebooks and Scripts referenced in this research:

- LiteratureReview
- ExtractRawImageData
- ObjectCount
- FetchWeatherData
- Person-EDA-Forecast
- Person-AnomalyDetectionForHourlyCounts
- RawImagesAnomalyDetectionTraining
- app.py