Thesis

August 25, 2020

- **Student:** Adam Napora (ID 18197892)
- **Supervisor:** Alessio Benavoli

- **Date:** 23 August 2020
- **Course:** MSc in Artificial Intelligence, 2019/2020
- **Faculty:** Science and Engineering
- **Title:** Enriched Camera Monitoring System with Computer Vision and Machine Learning

Enriched Camera Monitoring System

With Computer Vision and Machine Learning

UNIVERSITY OF
LIMERICK
OLLSCOIL LUIMNIGH

Abstract

This research is a study of a Smart Camera Monitoring System.

The aim is to find the most suitable algorithms, which can facilitate: robust data collection pipeline; ability to make accurate predictions for future events; and identification of anomalies.

Even though the existing literature is rich in the fields of Computer Vision, Machine Learning, and Anomaly Detection, there seems to be a gap in combining them into a coherent study, of an end to end process.

The study has highlighted several significant complexities, that need to be addressed, in order to achieve a reliable object detection pipeline. Including, but not limited to: hardware selection; network configuration; and coding data processing routine.

Low cost *IOT* devices, like Raspberry Pi, can be utilized for camera streaming. The use of *Background Subtraction* can increase efficiency, and save resources, while employing *Yolo Object Detector* can accurately detect up to 80 types of objects in the image data.

Prediction of object counts in a given hour, can be determined by training a model, which uses historical data, composed of collected images with detections, and a weather forecast. *Gaussian Process* and *Histogram-Based Gradient Boosting Regressor*, tend to be the most suitable algorithms to achieve the lowest error rates.

Search for anomalous signals in the image data, can be conducted via *Probabilistic* and *Deep Learning Auto Encoder* models. User alerts can be triggered, when thresholds are breached in the process of scanning camera frames for deviations from the norm, and keeping track of a number of objects in a given time interval.

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

Scope of the research

This research uses modern Computer Vision, Machine Learning, and Hardware, to study a Camera Monitoring System.

Three core features are explored: ability to detect objects in a video stream; prediction of a number of objects in a time interval; and anomaly detection.

Most of the publicly available research papers, target the areas listed above individually. This study however, shows how the three components can be connected, and how they can interact with one another. For example: a collection of images with detected objects, can be used as a valuable data source, to predict future object counts, or to trigger alerts.

Research questions

Below are the key questions posed in this study:

- What is the level of complexity, required to build a fast, and reliable object detection pipeline, using *IOT devices* and *Computer Vision*?
- Given the dataset with collected images, can the future object counts be accurately predicted using *Machine Learning*?
- Can *Anomaly Detection* algorithms help in recognizing anomalous patterns in the object detection data?

If the research goals are achieved, then the final product should be generic enough to apply it to other use cases, such as predicting traffic or tourist congestion, or animal behaviour.

Research importance

Based on the findings and experiments of this study, anyone with some Computer Science skills, should be able to build their own monitoring system, and potentially enhance the security of their monitored objects.

In the world of open source, it is important to share information with others, who can use it to their own advantage, but can also provide useful feedback, and contribute to the product. The source code for this research is therefore publicly available on Github.

Known Limitations

The list below covers the main limitations discovered in the study:

- It was not yet possible, to test the system in another location, which would test the true generalization characteristics of the proposed models
- The type of the camera used in the process is rather basic. The default Raspberry Pi camera, does not have the night vision capability, or waterproof casing, 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 [1], most of the burglaries occur during the day, when most of adults are at work or at school
- Forecast and anomaly detection accuracy is somewhat limited. Based on the stochasticity of the measured process, it might be difficult to improve. However, adding extra features, or collecting more data, could have a positive impact
- Loss of power translates into the loss in data, as at this point, an alternative source of power is not available

Reader's guide

Despite the fact, that this dissertation has been exported to a pdf file, it has been fully written in Jupyter Notebooks, so please, forgive the poor formatting, and image alignment in the pdf file.

Full project can be cloned from GitHub, code samples can be executed, and plots reproduced.

Below are some guidelines, to make the reading a better experience:

- Chapters contain links to the previous, and next chapters, on the top and in the bottom of each Notebook
- Citations, important concepts and terms are written in *italic*
- Some chapters provide a reference to an in-depth study Notebooks (called the *Extras*), which contain well documented code samples, additional commentary and plots
- Chapters are structured as a hierarchy, with a maximum of two levels of depth (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 is formated like `this_function`
- Some paragraphs are divided by a title in a **bold font** to improve readability

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

index | next

```
<IPython.core.display.HTML object>
```

2. Literature Review

index | prev | next

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

In 1963, Larry Roberts in his Ph.D. [2], mentions that the *pictorial data* understanding by the machines is a challenge, and he proposed the first computer program, used to process a photograph into a line drawing.

Since then, a study about the use of computers to recognize objects in images has emerged. Small steps towards the progress, were often negated, by the vast complexity in the explored field.

In a well known, *Summer Project* experiment [3], a camera has been attached to a computer, to "describe what it saw". Minsky (MIT) delegated this task, to a first year undergraduate student.

Little did they know at the time, how long it would take numerous scientists to solve this problem at a satisfactory level.

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* [4].

From year 2012 onwards, there has been an exponential progress in the fields of Object Recognition, and Object Detection. Hand crafted feature detectors, like SIFT [5], used in conjunction with prediction algorithms, like SVM [6], have been challenged by more capable Convolutional Neural Networks.

Last decade also brought a substantial innovation in the hardware space, and data availability. Starting from the usage of GPU (Graphical Processing Unit) to parallelize matrix multiplication; doubling CPU clock speeds 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, and solve challenging problems.

This chapter provides a theoretical foundation for the key ideas and algorithms, which have been explored, and utilized inside this research.

Note: Code samples used to generate plots illustrated in this chapter, can be found in the corresponding Extra Notebook 1

2.1. Data Collection

Data Collection involves a mini-computer (Raspberry Pi), which streams image data to the Central Unit (a GPU-enabled, Ubuntu-based Desktop PC). It processes camera frames in an infinite loop, with two key algorithms: - Background subtraction - Yolo Object Recognition

2.1.1. Motion Detection

To detect objects of interest in a video stream, a naive approach could be used, to send all frames into the Object Detector algorithm.

This would work, but it would be extremely inefficient. Object detection process is complex, and tends to be a bottleneck in terms of speed and hardware utilization.

An optimized variant, detects a significant change in a series of consecutive images, and only when motion is detected, frames are sent to the Object Detector.

Below is an example of a static background, without motion:

Fig. 2.1. Static background

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

Fig. 2.2. Moving object

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

Let the starting point be a static image without any moving objects (called *the background* - *BG*). Next, every consecutive frame (*the foreground* - *FG*) is compared against the background, using pixel intensity. Once the difference is above a threshold, motion is detected.

Unfortunately, there are many challenges in this optimistic approach, which can trigger false alarms:

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

A more advanced algorithm has been proposed in the *Improved Adaptive Gaussian Mixture Model for Background Subtraction* paper [7]. 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 not static, but is constantly updated. Author used 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, so it is given 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, author 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 the 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

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

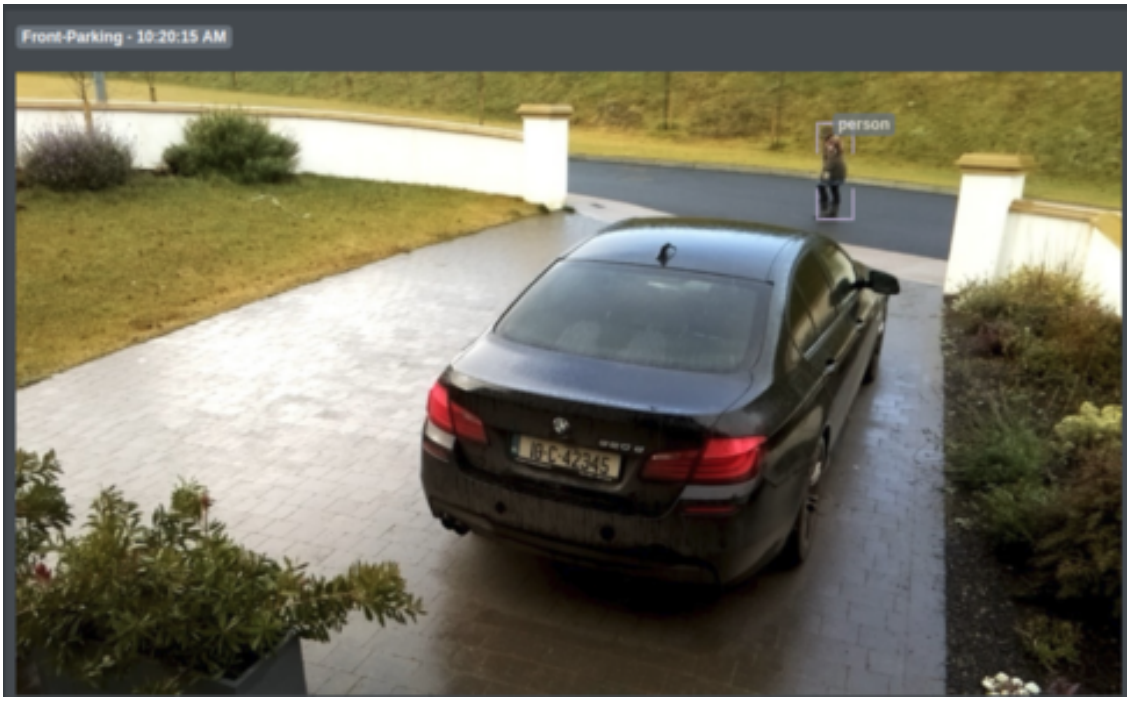
Object detection is based on two Computer Vision concepts:

- 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)

The task of object detector, is to classify multiple objects in an image, and also tell their locations.

Below is an screen-shot from a web application, built for this research. Image shows a single *Person* object, classified by object detector, after a positive response from the motion sensing routine:

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 objects of interest in the frame, they would be contained in their own bounding boxes too.

Object Detection is a dynamically evolving field, with new algorithms constantly competing for the best accuracy and speed.

Out of the two, arguably the most popular options for object detection in Python: *Yolo* and *SSD*, I have decided to use Yolo (You Only Look Once). Its ability to run in real time on a GPU at 30+ frames per second, high accuracy and wide-spread adoption rate, make it a primary choice. 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.

Yolo V1:

Even though this research uses Yolo Version 2 [8], the theory section below is dedicated to Yolo version 1 [9]. It is important to understand this version first, as the next release builds on its predecessor.

Yolo v1 has been released in 2015, as a ground breaking approach to object detection, with the extremely high real time prediction speed. This was a significant achievement in comparison to previous object detectors, like R-CNN [10], where a single image could take 20 seconds to get

processed or even in comparison to more modern Fast R-CNN [11], which still took 2 seconds to process a single image and Faster R-CNN [12] with 0.14 second per frame.

The processing speeds can be extremely important for applications, which make real time decisions based on the image data. A self driving car, robotic arm, or a camera monitoring system are some examples, where high detection speed is beneficial.

According to Yolo authors, algorithm can run at 45 frames per second, with a slight decrease in accuracy for smaller objects.

Table below visualizes the progress in the area, where the FPS column shows the number of frames, which can be processed in a second:

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 the complete re-engineering of how the object detectors can operate.

Traditional methods used previously, like: a sliding window with HOG (Histogram of Oriented Gradients); SVM (Support Vector Machine); and region proposals (seen in R-CNN's); required potentially thousands of iterations over the single frame.

In contrast, Yolo uses only a single pass, through the entire image, to generate predictions.

Then, to get rid of overlapping bounding boxes, the ones with very low probability are discarded, and Non-max suppression [13] algorithm is applied.

This approach generates an output of 1470 features, with 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 are the labels for 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:

$$Conf = Pr(\text{Object}) \cdot \text{IOU}_{\text{pred}}^{\text{truth}}$$

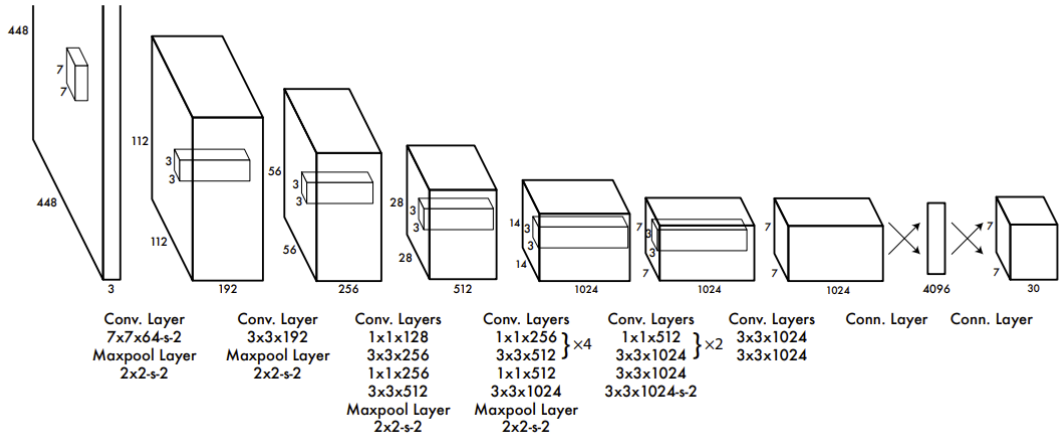
, where IOU represents an Intersection Over Union, an evaluation metric for bounding boxes:

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

If there are no objects present in the cell, then the confidence is zero, but if there is an object, the confidence is 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 the 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)

Below are the 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.

In summary, Yolo is a solid trade-off between speed and accuracy, and it is one of the most useful open source tools for object detection.

Yolo V2:

An updated 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. As authors conclude, it outperforms the other two most popular object detectors: *SSD* and *Faster RCNN with ResNet*.

The main improvements in Yolo v2 have been achieved via a number of innovative 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 to a very popular choice for feature extractor, VGG-16 [14], which requires 30.69 billion floating point operations. For comparison, Yolo V1 required 8.52 billion, as it was based on the Googlenet architecture [15].

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

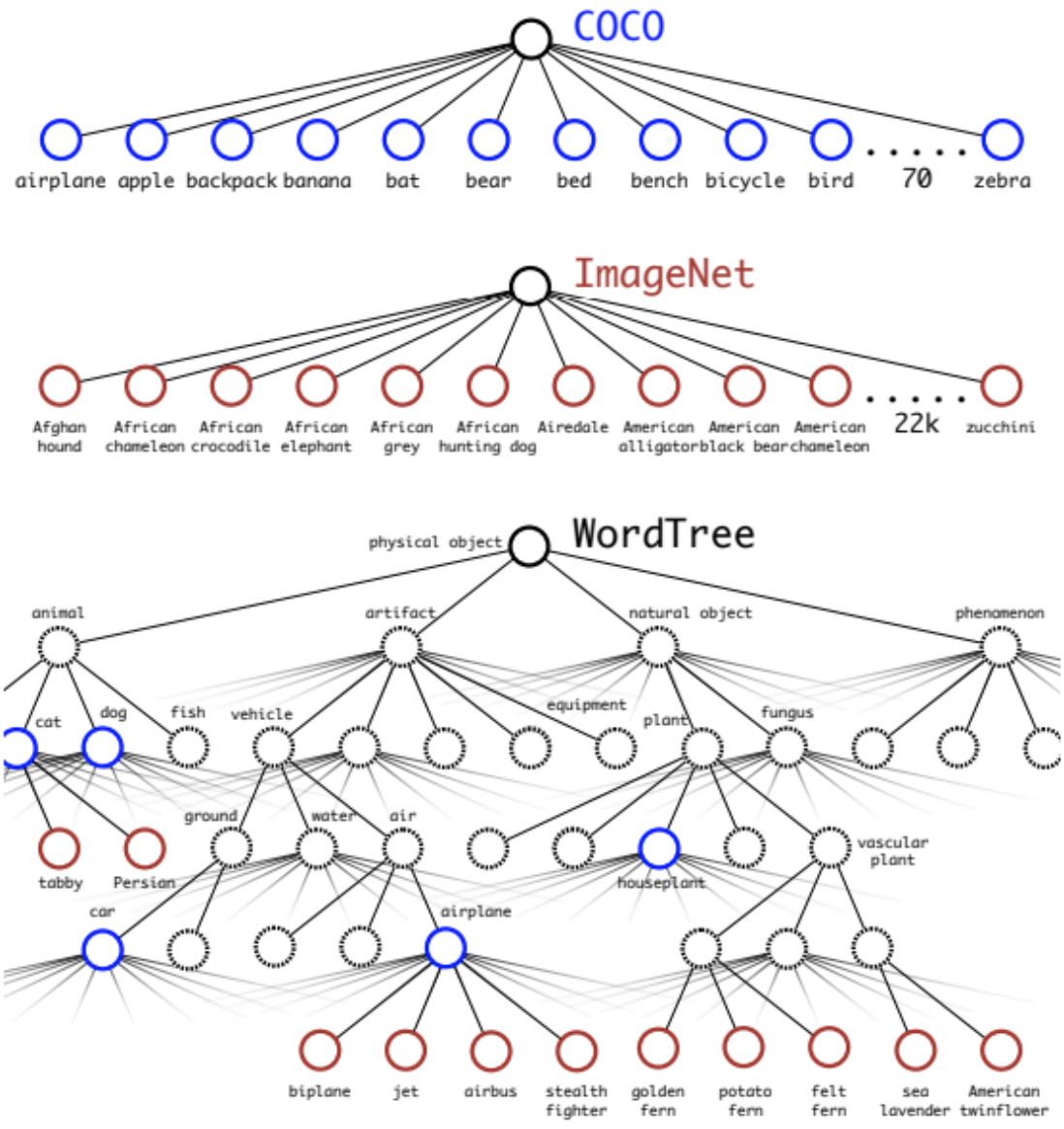
The *classification model* utilizes ImageNet dataset, with 1000 very fine-grained classes (like "Norfolk terrier"), and *detection model* uses COCO dataset, with 80 high-level class names (like "dog").

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

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

This model can been visualized as the following tree diagram:

Fig. 2.5. Yolo V2 WordTree



Yolo V3, V4:

Since 2018, two further iterations of Yolo have been released:

- version 3 [16], after which Redmon has discontinued his work in Computer Vision (tweet), due to ethical reasons (military applications and privacy concerns)
- version 4 [17]

2.2. Forecasting

We say that a prediction 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, a learner like *Decision Tree*, models relationships in the training data very well, but tends to perform poorly on the testing sets. We call this behaviour *High Variance*.

A good model has 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 [18].

Literature review in this section presents and compares, three *Machine Learning* algorithms:

- Standard Decision Tree
- Gradient Boosting Decision Tree
- Gaussian Process

Study on other Machine Learning algorithms (Linear Regression; Feed Forward Neural Network; LSTM), did not improve the results, and therefore they were omitted from the results and literature review.

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 popular choice, when decisions must be clearly understood, and explained.

The history of Decision Trees used for regression problems, is not trivial to pin down, and goes back to the research by J.N. Morgan [19], titled *Problems in the Analysis of Survey Data, and a Proposal*, when the first decision tree for regression was drawn.

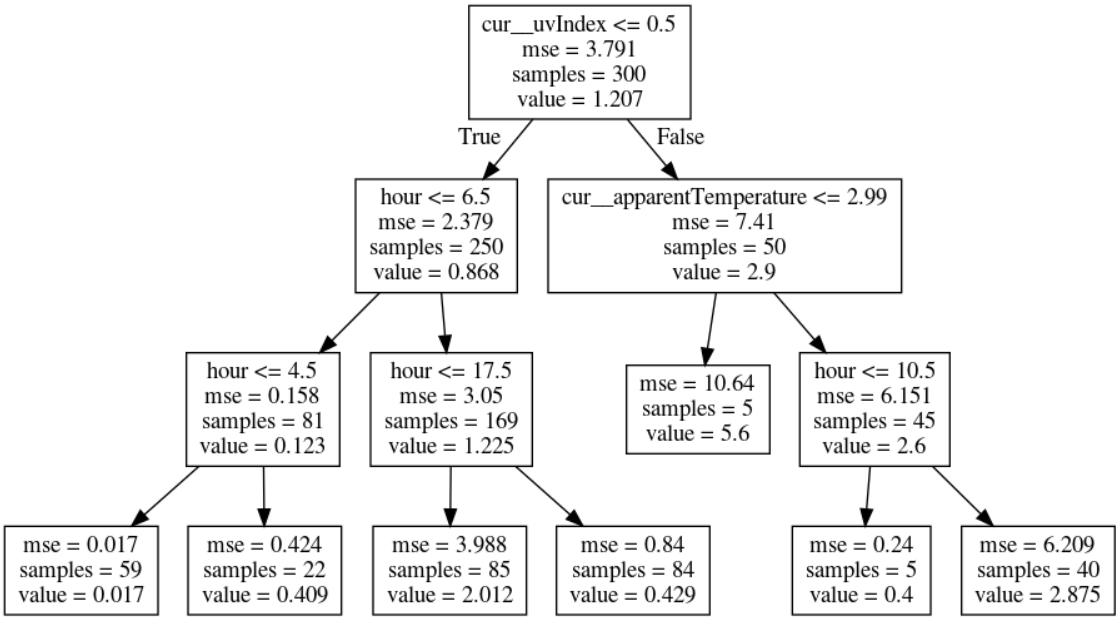
What has been a challenge back then (computational overhead), is actually not an issue any more. In fact, decision trees are one of the fastest Machine Learners available, for relatively small datasets.

However, a downside of decision trees, is their low accuracy on an out-of-sample datasets, and low robustness. Small changes in the training data may easily lead to a very different tree.

It is important to discuss their inner workings, before moving on to the more advanced algorithms.

Below is a visualization of a Regressor Tree training. For illustrative purposes, the tree uses a sample of the *Hourly Person Counts* dataset, with the shape of 100×4.

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 the *root node*
- nodes in the middle are called the *decision nodes*, and are connected by arrows creating a section called the *branch*
- the eight boxes in the bottom are the *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 prevent too high variance.

2.2.2. Gradient Boosting Regressor Tree

There are many extensions to the base Decision Tree algorithm.

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 [20]: *LightGBM: A Highly Efficient Gradient Boosting Decision Tree*, and it is a modern take on the original Gradient Boosting Machine algorithm by Jerome Friedman [21].

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 a single decision tree, is easier to interpret than many hundreds small trees. However, when the single tree grows to a very large scale, this conclusion tends to break.

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

- Start with a dataset composed of input features, 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: $F_0(x) = \text{argmin}_\gamma \sum_{i=1}^{n} L(y_i, \gamma)$. Solving this equation, finds the initial predictions. 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* version, is related to the way Decision Trees find the best value to split the data. Vanilla Gradient Boosting algorithm sorts the values, and then

for each pair of values, runs the test if the split is optimal. However, histogram-based Regressor avoids the computational problem for larger datasets, by binning the values into (typically) 256 bins, and using integer-based data structures (histograms). This way, expensive sort and tests for all continuous floating point values is eliminated.

One of the most recent improvements in the `sklearn` implementation is a *Poisson* loss function, which is 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.

As per definition [22], 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.

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, which is conditioned on the input.

This feature is utilized in the Forecasting Notebook.

2.2.3. Gaussian Process

To begin the discussion about Gaussian Process, it is important to summarize the Gaussian Distribution, and its characteristics.

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

Its origins go back to 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 [23](Springer 2008) and unique analytical properties, make Gaussian distributions a very useful tool, which can be applied to Machine Learning.

The literature review below takes a gradual approach, to understand the Gaussian Processes:

- Univariate Gaussian Distribution
- Multivariate Gaussian Distribution
- Gaussian Process

The choice of studying Gaussian Process is this research, was motivated by the property to generate predictions, as well as their uncertainty. Knowing the confidence of the predictions, allows to make more informed, and explainable decisions in the AI systems.

Univariate Gaussian

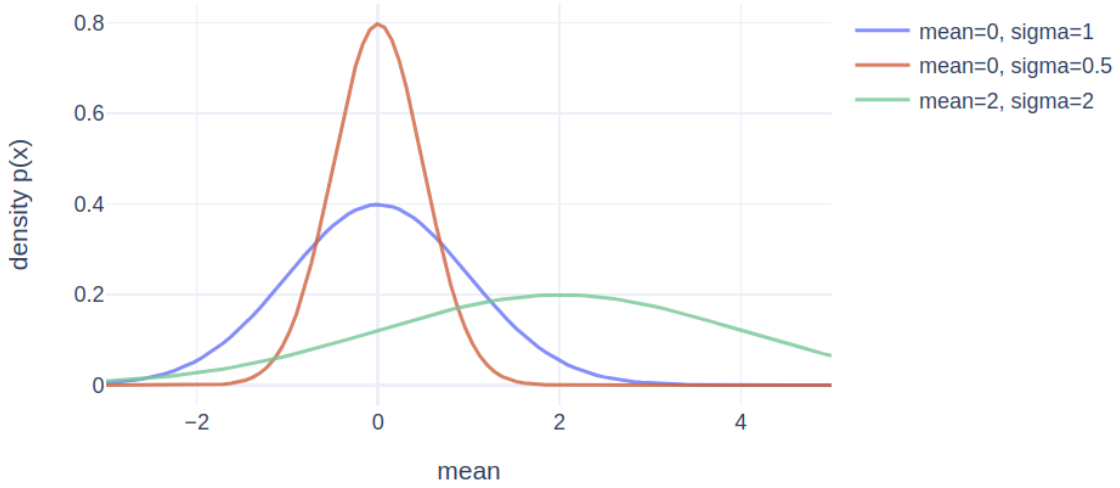
As per definition [24], Gaussian distribution is given by:$\mathcal{N}(\mu, \sigma^2)$, where μ is the expected value of a distribution, and σ corresponds to a standard deviation from μ. Sigma squared (σ^2) is also known as a variance.

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. Three sample 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 two parameters:

- mean vector μ
- covariance matrix Σ, which measures the correlation of the each pair of variables

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, $\mathcal{N}(\mu, \Sigma)$ is used, to denote this distribution.

Below are two examples of multivariate Gaussian distribution:

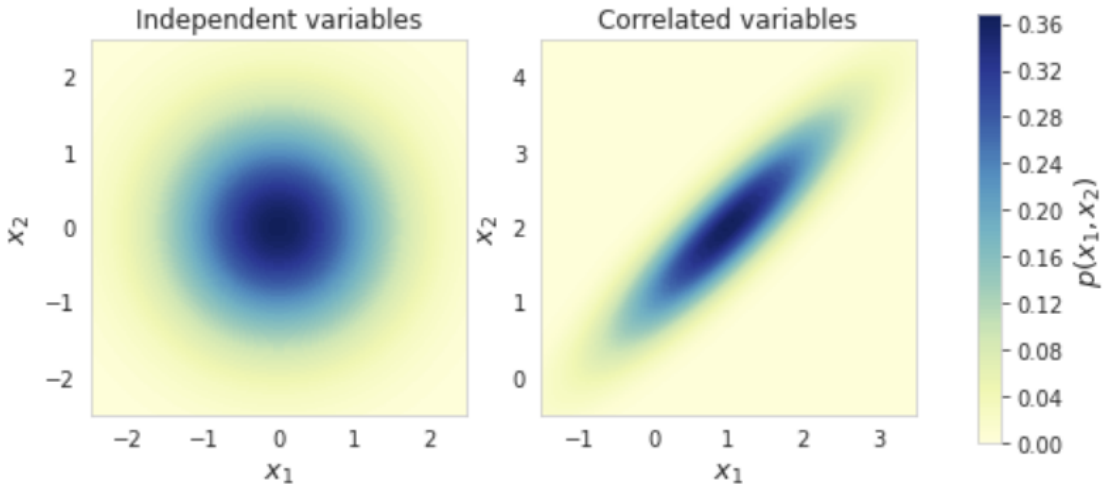
- first example shows 2 uncorrelated variables. Change in x_1 does not mean a change in x_2 ($0, 0$ diagonals in Σ):

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

- 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 via *Cholesky decomposition* [25], 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 [26], is a stochastic process, which involves 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 the 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 uncertainty.

The difference between the multivariate Gaussian, and Gaussian process, is that the latter 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.

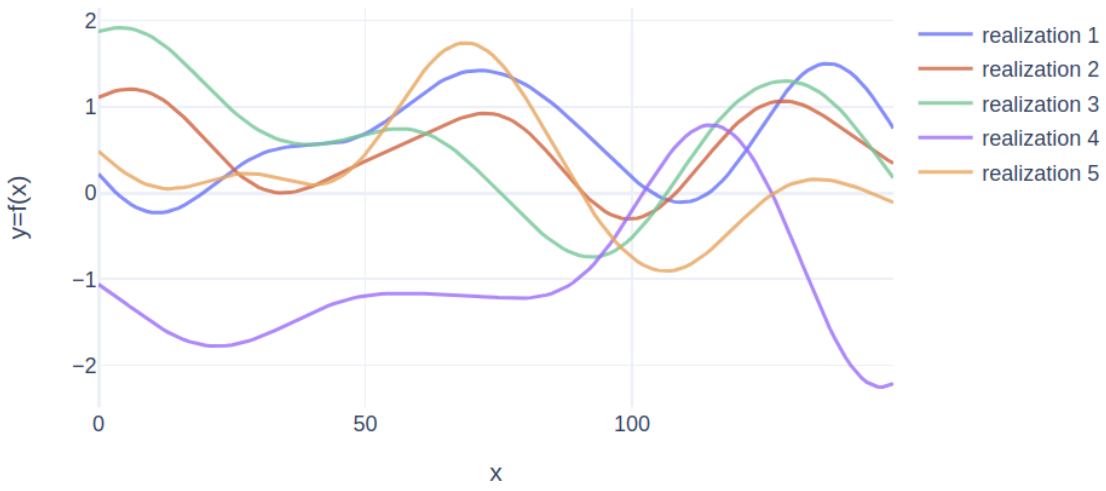
In the Bayesian language, selecting the specification for the covariance function (called the kernel function), is setting a prior information. Kernel needs to be positive-definite, to be a valid function.

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:

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

Sampling from prior, when number of points is finite results in a marginal distribution that is Gaussian.

Fig. 2.9. Sampling from RBF



When Gaussian Process is used for regression, a three-step procedure is followed:

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 $y_2 = f(X_2)$, samples are drawn 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:

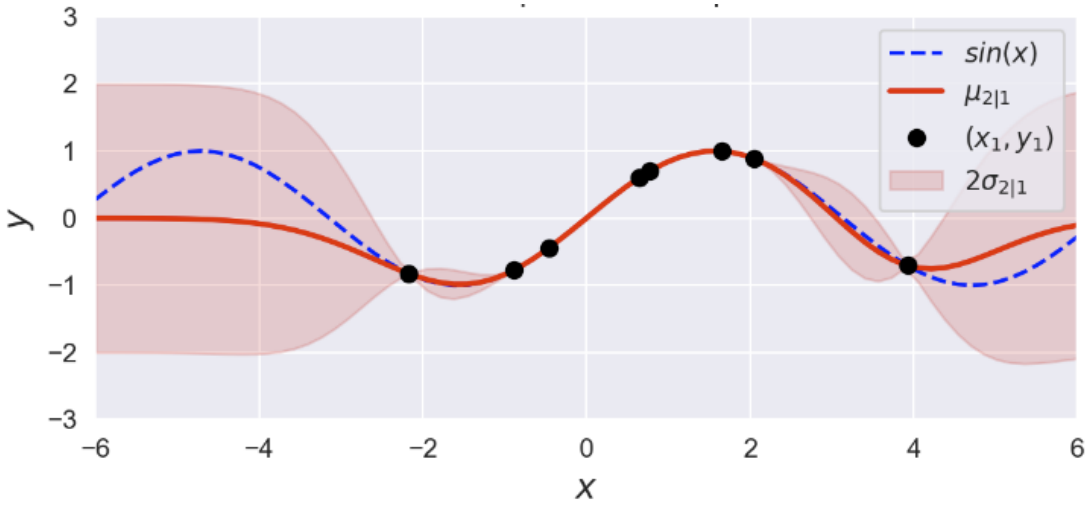
$$\mu_{2|1} = (\Sigma_{11}^{-1}\Sigma_{12})^T y_1$$

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

y_2 can be predicted, through the use of 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 increases in the sections without data points:

Fig. 2.10. Gaussian Process predictions and uncertainty



2.2.4. Conclusion

Gaussian processes are an elegant, robust and informed approach to Machine Learning. One not only generates predictions for unseen data, but also the uncertainty. This can be a 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. During experiments in this research, Gaussian process algorithms were rather slow to train (given the dataset of only < 4000 records), and they consumed a lot of memory (GPy was much more efficient in this area, than pymc3).

There are methods used to decrease the complexity to $\mathcal{O}(n^2)$, and new frameworks continuously work on the improvements (pymc developers are currently switching their 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 features in this research is *Anomaly Detection*.

For example, it could be useful from a security perspective, if an AI system could alert home owners, when something out of the ordinary, is taking place around their property.

Auto-encoders, are a Neural Network models, which are 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. Therefore it is loosely classified as an unsupervised learning algorithm.

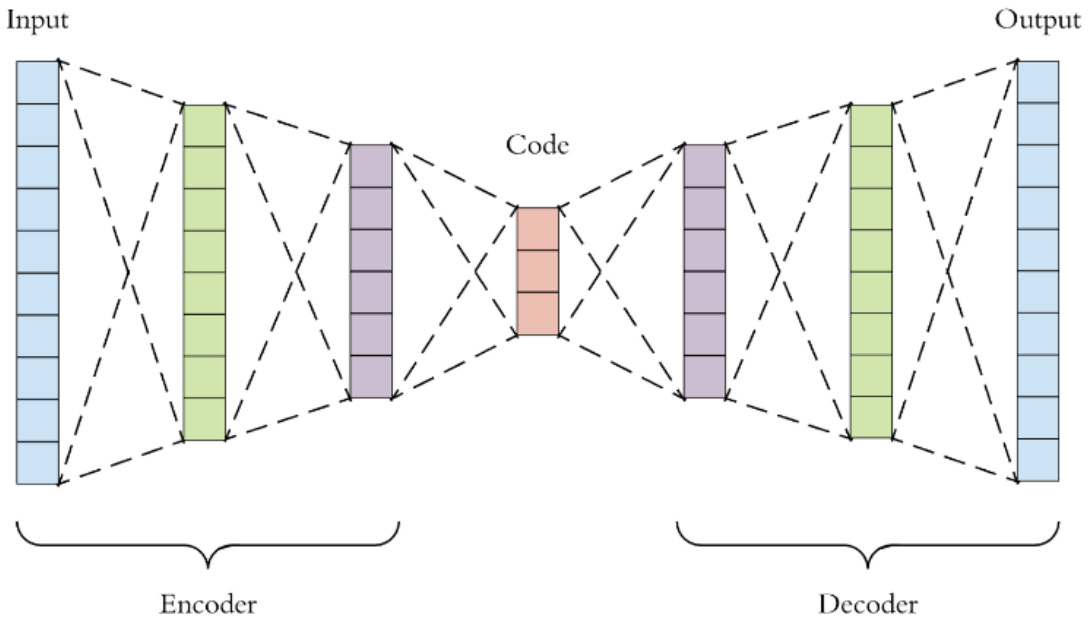
There is a weak evidence of the origins for this Neural Network architecture, which is described in an online *Deep Learning book* [27], where author dates back the method to the eighties. However, the terminology and the use cases have changed drastically over the years.

Currently, auto-encoders are used to achieve following goals:

- data compression (dimensionality reduction)
- image de-noising
- anomaly detection
- machine translation

While there are several architectures of this type of Neural Network, at a high level it can be represented in a following diagram [28]:

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 learns to reproduce the inputs perfectly, is not 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 whole 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 released at rapid pace. The overview above, provided a background on the algorithms utilized in the next chapters.

Next Chapter focuses on the **System Design**.

index | prev | next

3. High Level System Design

index | prev | next

The system design, proposed as part of this research, can be described as a two high level components:

- Real time frame processing used for object detection, and anomaly detection
- Batch processing used for forecasting, and other kind of anomaly detection

Note: This design is a proof of concept. Only individual components are developed as part of this research.

3.1. Real time frame processing

Fig. 3.1. System Design - Real Time Processing

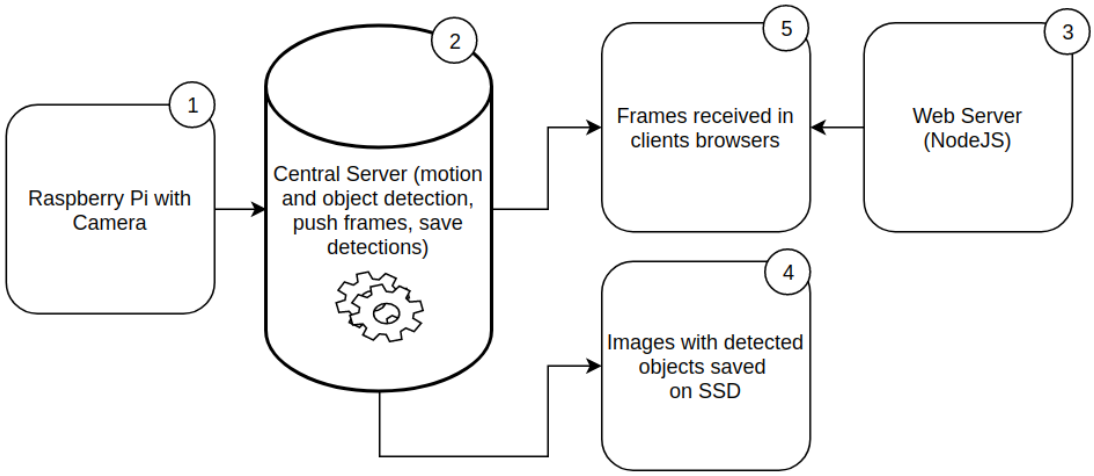


Diagram description:

- Raspberry Pi camera (1) generates frames
- Frames, as *jpeg buffers*, flow to a Desktop PC via high performance message queue - ZMQ (2)
- Processing is applied to each frame:
  - resize, apply motion sensing, detect object classes and their x, y coordinates
  - save images on SSD Drive (4), for further analysis and batch processing
  - emit all frames to connected end users (5)
- Client web browser application is served by NodeJS web server to render a UI (3)

The data flow is described in more detail in the next chapter - Data Collection.

3.2 Batch processing

Fig. 3.2. System Design - Batch Processing

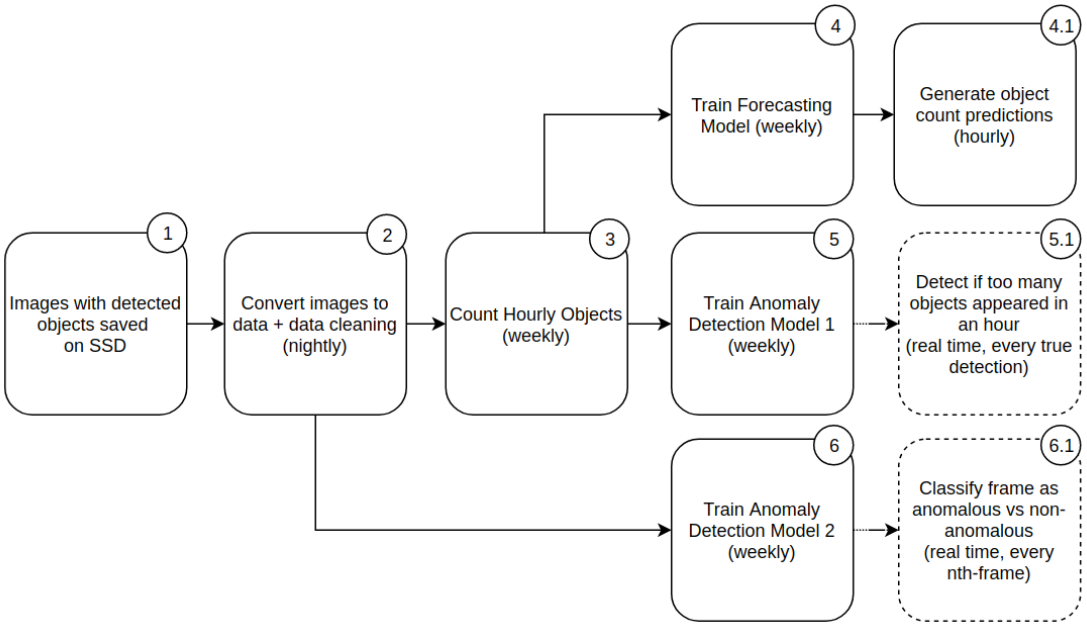


Diagram description:

- Start with collection of raw images (1)
- Every night a scheduled task is executed, where images from the whole day are converted into a 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 training of a Machine Learning model (4), used for forecasting
  - New predictions are generated every hour (4.1) for next three days

  - Weekly anomaly detection training script (6), used 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 raw image pixel intensities, is used to train an auto-encoder model, capable of classifying frames as anomalous (6) [the classification occurs in real time (box 6.1)]

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

3.3. Conclusion

This chapter served a high level overview of the system design.

Next Chapter contains more detail for the Figure 3.1. rendered above.

index | prev | next

4. Data Collection and Pre-Processing

index | prev | next

The aim of the data collection pipeline, is to use a camera, to stream video signal. Then, a central processing machine, runs a series of operations to detect objects, and stores images, when objects of interest are identified.

This chapter is an in-depth analysis, and an extension of a "Real time frame processing" diagram (Fig. 3.1.) from Chapter 3. It is broken it down into the following sections:

- physical layer (hardware):
  - hardware selection
  - connectivity
  - camera location choice
  - redundancy
- logical layer (software):
  - video stream and consumption
  - frame life-cycle
- results:
  - data representation
  - data volume

4.1. Physical layer (hardware)

4.1.1. Hardware selection

Streaming device:

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

- standalone (professional) camera unit

- mini computer with a camera module

The factors, which were prioritized as part of this research are:

- low cost
- availability of online documentation
- customization and flexibility
- ability to perform additional tasks on-device

Based on that the mini computer seems to be a better choice. Raspberry Pi, with an additional Pi-Camera module, can be selected. R-Pi has the following specification:

- 4-core ARM processor
- 1GB of RAM
- Ethernet, 4 USB and HDMI ports

Considering the small form factor of $3.54 \times 2.36 \times 0.79$ inches, it can stream the video signal, and run multiple tasks at the same time. It runs Linux, so it is flexible and customizable.

Most of the standalone cameras are more durable, support night vision mode, and protect from environmental hazards. However, these extras come at much higher cost, little flexibility in terms of configuration options, and often limited online support.

Processing device:

For the server machine, a multi-purpose Desktop PC was chosen. It might be a good solution if there is already one inside the house.

There is another tension here, between the local PC and a cloud-based solution. Below are the three characteristics of cloud, which were the deciding factors against it:

- cloud requires fast and reliable broadband connection
- data travels outside the local network, and can be attacked by the hackers
- cloud introduces additional dependency and potential complexity, which has to be managed and maintained

Even with the downsides listed above, the cloud option is much more scalable and in many cases, might be a more suitable option, if transferred data is protected well.

During this experiment, the internet connection was very limited, and therefore the cloud usage was discarded.

Below is the configuration for the PC machine utilized in the research:

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

It allows to run a smooth, real time image preprocessing, object detection, and at the same time execute scheduled tasks, used to train the models, or generate predictions.

4.1.2. Connectivity

The initial wireless setup, caused several issues. There was a severe lag in the video stream, packets were dropped, and logging in to Raspberry Pi devices remotely, was very slow, with occasional minutes of unresponsiveness.

The only choice to improve the latency, was to wire the house with ethernet cables. The downside of this approach was the additional cost, and a significant time investment.

Upon completion however, it provided near-noiseless communication between the camera and the server, and much more optimized development process.

Below is a high level diagram, of the current network topology (excluding statics IP addresses of the devices):

Fig. 4.1. Network Topology

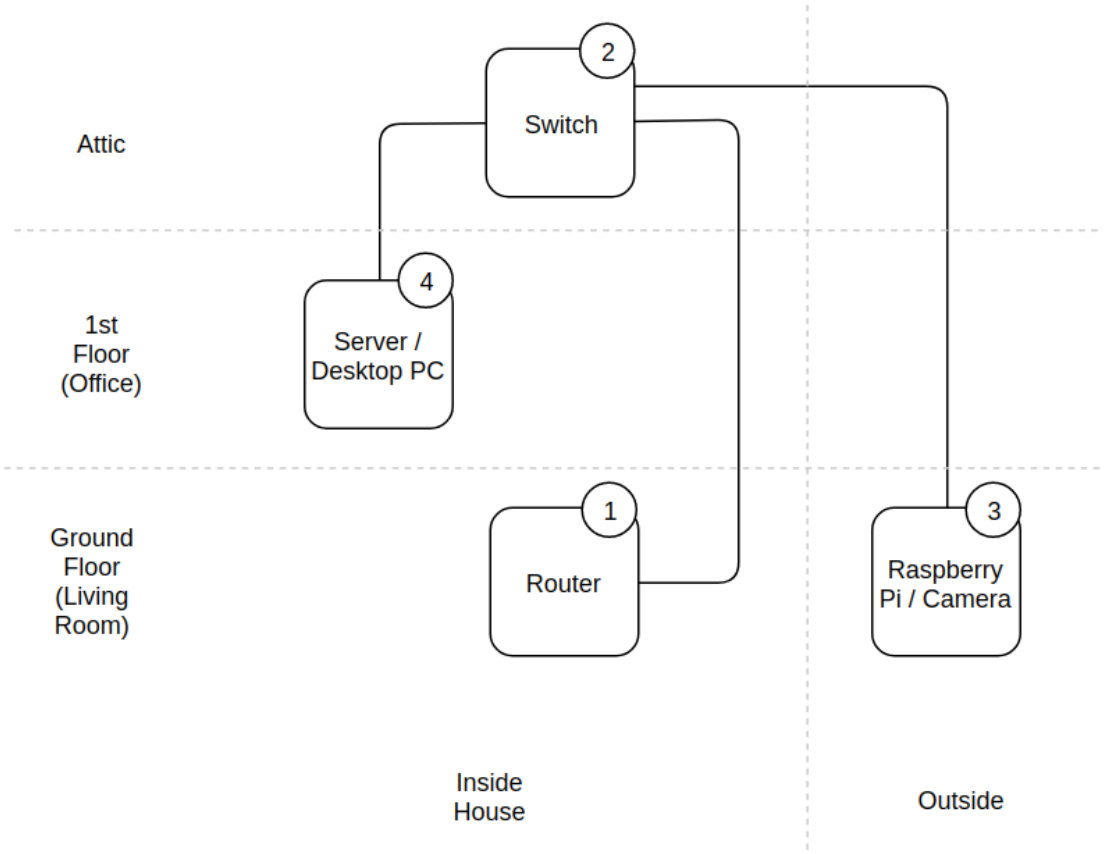


Diagram Description

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

- Signal travels from the router (1), to a 1Gb Switch (2) in the attic. Switch has 4 Power Over

Ethernet (POE) ports

- From the switch, a network cable is run to the Raspberry Pi (3), which is located outside of the house. Raspberry Pi has an additional POE module, and therefore it only requires a single wire. Pi does not support 1Gb speed, but 100Mb offered by POE, is enough to transmit video signal at high rate
- The second connection from the switch, goes from the attic into the office, where the server PC (4) is located

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

4.1.3. Camera location

Optimal camera placement is not a trivial task.

Camera was mounted in a position, which is usually chosen for the 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 camera location is certainly too low to be considered a proper security camera, but it shelters the device against the rain, wind and direct sunlight, which often cause serious issues for the vision systems.

The RaspberryPi with camera module is secured by a strong double sided tape to the roof in the porch, and ports are blocked off from the humidity:

Fig. 4.1. Raspberry Pi Camera Placement #1

Fig. 4.2. Raspberry Pi Camera Placement #2

After many attempts at getting a suitable case for R-Pi, it became obvious that the priority is its stability, and a tilt along both axis (figures 4.1 and 4.2).

With the proper casing, even during harsh storms, the camera should not move the lens.

Another challenge, which can occur at random occasion is the occlusion caused by natural events. This could be as simple as a leaf or dust particles, 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 has decided to adopt the Raspberry Pi case as its home.

Based on the findings above, the AI system should be able to detect the loss in the image quality, and send a maintenance alert to the users.

4.1.4. Redundancy

A three power outages occurred during the six months of data collection, and it resulted in a loss of data for three days.

An alternative source of power in the security systems is critical. This redundancy comes at the additional cost, and this is currently a known limitation of the system.

After a problem with power or network, it is important for the devices to 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 via a software called Supervisor.

In case of disk 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 (see Appendices), but can be further refined and improved.

4.2.1. Video streams

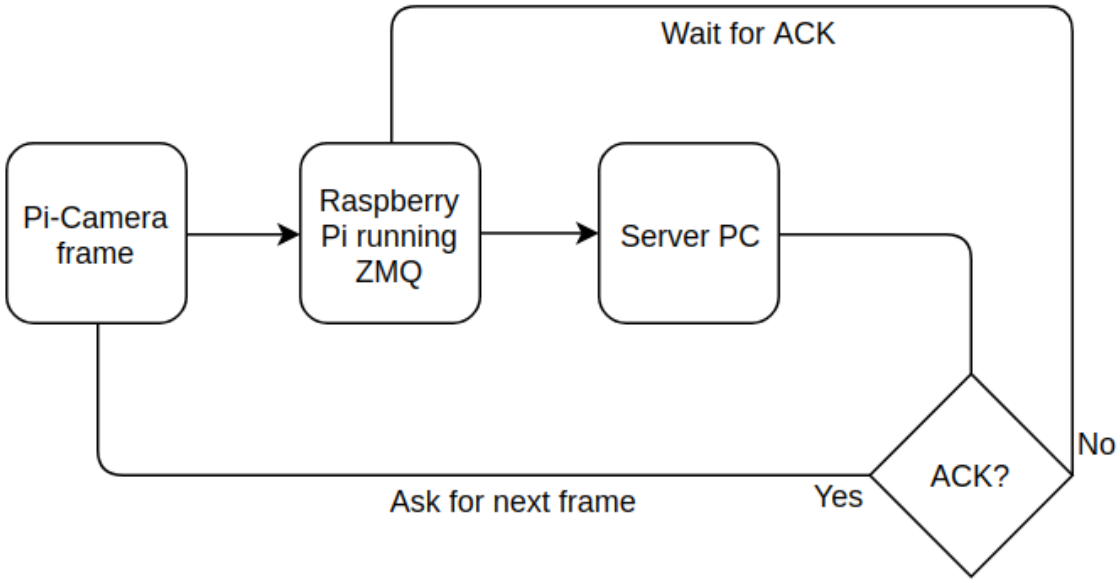
There are many ways, in which video frames can be broadcasted to other devices.

In general, people often choose an easy to setup streaming protocol, like *RTSP*, if they 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 an OpenCV and Python. In addition, RTSP streams send the video continuously, without considering if the receiving end is online, which can shorten the lifespan of the camera.

If an audio component is not required, and enhanced flexibility and customization is important, then a message queue might be a more suitable option. ImageZMQ is a convenient implementation of *ZMQ/PyZMQ*, a peer to peer message queue system, optimized for high performance, which can wait for an acknowledgment signal from the receiving clients.

Below is a diagram of a frame exchange, between the Pi-camera and Desktop 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 without a receiver. This extends the lifespan of the camera, and removes the complexity of keeping up with the incoming frames.

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

As part of this research, there are two repositories published in GitHub, with the code required to run the client, and the server. These also exist in the Appendices.

The `client.py` and `server.py` are registered as Linux service via Supervisor. It ensures 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)

Below is a diagram, which highlights the complexity of the data flow for each frame:

Fig. 4.3. Frame lifecycle

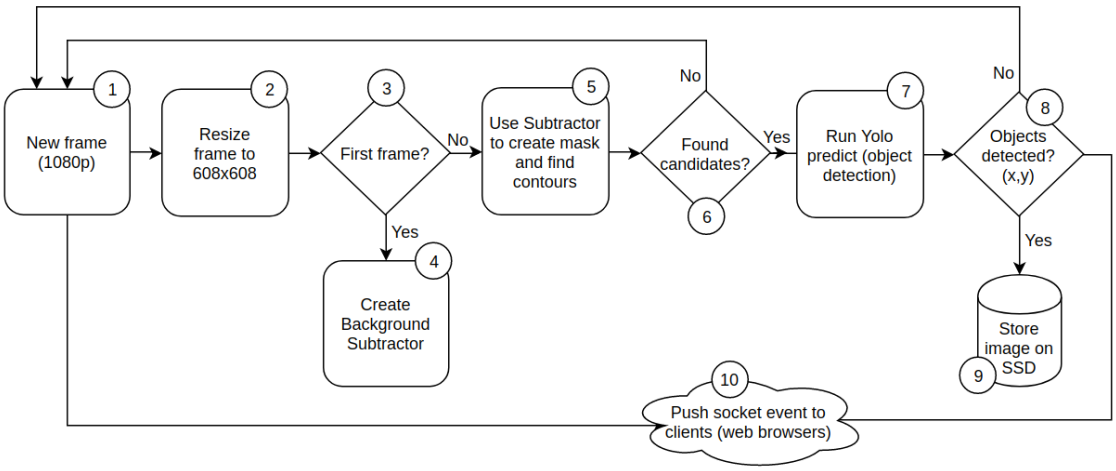


Diagram description

- Once a frame is collected from the message queue (ImageHub), it is a HD resolution (1)
- A copy of the is resized (2), otherwise next steps will be very slow. The choice of 608×608 makes sense, as this is the 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 [29], and unlike Euclidean distance, which measures distance between two points, is given by:

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

    , 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 tends to help, to record multiple videos from a particular location, for the calibration purpose. 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

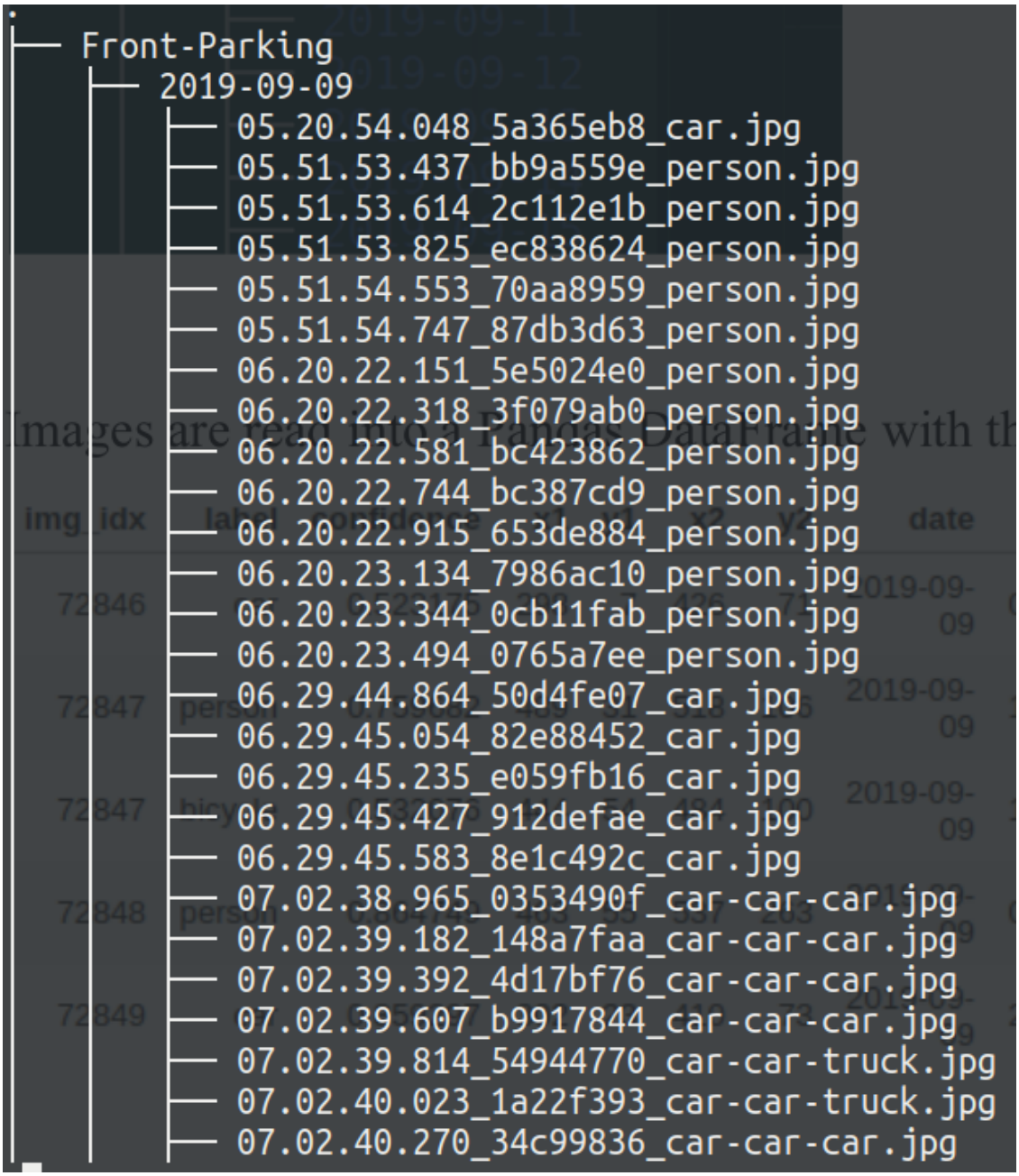
- When dealing with Person and Vehicle object categories, objects of size > 35, are good candidates for further detection. Everything below this value is not a valid object, and can be dropped. Parameter `MIN_OBJ_AREA` is another heuristic, which should be calibrated for each environment
- Once candidates are found, frame is forwarded to Yolo, to generate predictions for all objects in the 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 in the DarkNet implementation requires to provide:
  - set of labels (yolov2.cfg)
  - pre-trained weights (yolov2.weights)
  - confidence threshold for predictions (reject below (0.40))
  - 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
- Additionally, 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 (8)
- If an object of interest is detected, frame is saved as an image on the hard drive (9) in the folder, which corresponds to a date
- Then, independently of the object detection process, the original HD 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, 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 only the images, which contained valid detections.

Image data is organized as a hierarchy of folders. The top level contains a device name, and then dates. Image filenames contain objects detected, and the detection time.

Fig. 4.4. Collected Images



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

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

4.4. Conclusion

There are still a lot of details, which have been omitted from this chapter: - error handling - socket server implementation - an infinite loop in a separate thread, used to capture the stream within a *Flask* app context

However, the full implementation, can be found in the Extra Script 1 - app.py.

Considering the amount of processing for each frame, this pipeline runs at impressive speed, and provides a smooth experience for the end users.

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

Next chapter focuses on generating the prediction for the number of objects, expected to appear in a single hour.

index | prev | next

5. Forecasting

index | prev | next

Motivation:

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

Asking this question, has changed the perspective about the project, which was initially focused on the object detection pipeline only.

It can be very beneficial to predict the expected object counts. At allows for informed planning decisions, and anomaly detection.

Accurate forecast could answer questions like: "How many `cars` will appear in this location, between `9AM` and `10AM`, on a `weekend` day, when the weather is `good`", where all the parameters are dynamic.

Time interval:

It is an important decision, to choose the forecast time interval, which also represents the lowest granularity for the predictions.

During data exploration, a 15-minutes, 1-hour and 3-hour windows have been considered, but the 1 hour turned out to be the most optimal choice, given the following facts:

- 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

In a more generic sense, the time interval value will depend on the use case, and the available data, and it should be seen as a parameter to tune.

This Chapter:

This Notebook is an in-depth study of:

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

It is important to call out, that the 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 most suitable model, which will be also identified.

Notes:

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

5.1. Extract raw image data

Forecast in this research uses historical data, composed of the images with detections. These need to be pre-processed and turned into a tabular format.

This is where Python as a general purpose programming language can help. Below is a list of pre-processing steps:

- 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
- save 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 categories under the `label` feature; object detector's `confidence` and `x,y` coordinates for bounding boxes; `filename` of an 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.

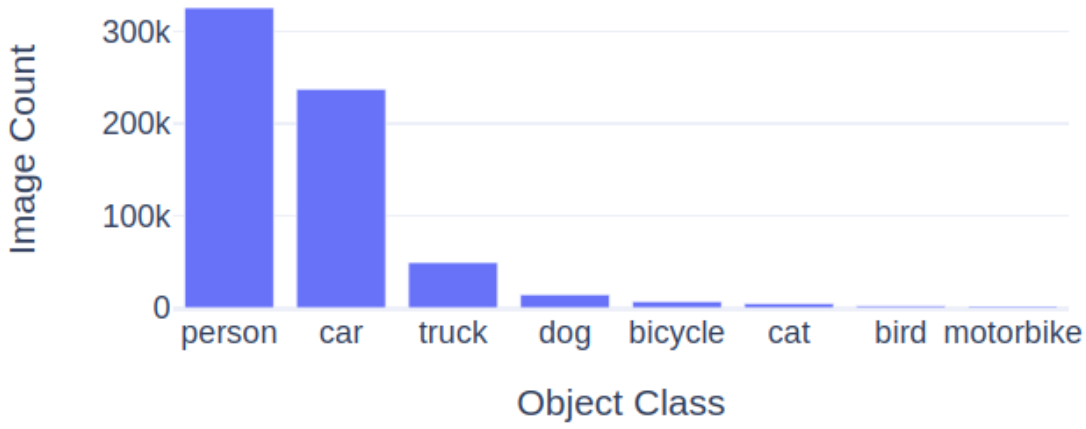
5.2. Count objects in frame sequences

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

Before counts can be calculated, a data clean up step needs to be included. Throughout the data collection process, there were three days of downtime (power outage), and any data collected during these days needs to be purged.

After the data purging, it is important to visualize the distribution of the object categories in the dataset:

Fig. 5.2. Object Category Distribution



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

Based on this distribution, a decision was made to merge *cars* and *trucks* into a *vehicle* class, and initially focus on predictions for *Person* and *Vehicle* only.

To count objects by hour, an iterative approach was taken:

- 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 a well known Pythagorean metric:

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

The algorithm above worked 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

A sample output for the *Person* category is provided below. The assumption is that each record represents a unique observation, extracted from the sequences of events:

Fig. 5.3. Unique detections

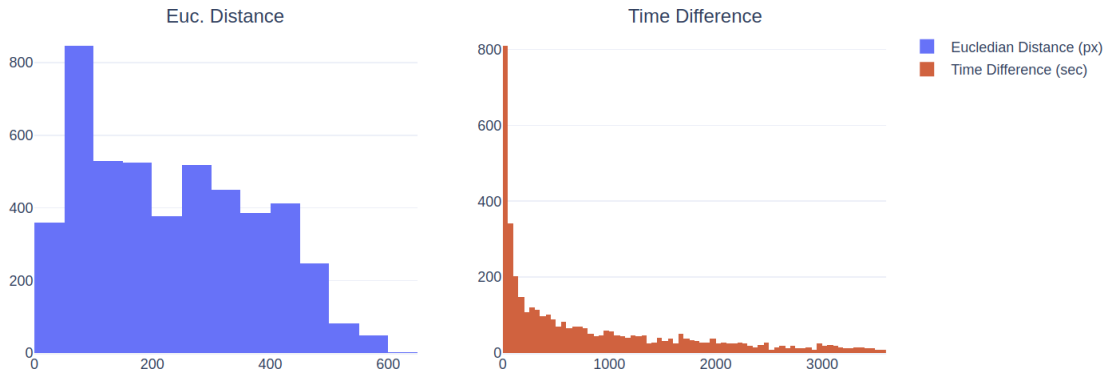
label	confidence	date	time	sec_diff	euc_distance
person	0.450496	2019-09-09	07.03.03	2560.471	279.827179
person	0.658724	2019-09-09	07.29.50	1606.475	291.185508
person	0.439394	2019-09-09	08.07.56	2270.200	247.779136
person	0.768141	2019-09-09	08.42.40	2075.153	103.711137
person	0.545153	2019-09-09	08.52.30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

This dataset only contains observations for a single class label, and adds a `time difference` and `euclidean distance` features, calculated against a previous image.

The shape of the dataset 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 intuitive sense, as the images are 608×608 px in size.

The differences in time between objects are a little bit surprising, with the majority of objects being captured in range of 0 and 50 seconds, between each other. Some tweaks to `THRESH_NEW_EVENT_MIN_DISTANCE` could improve this picture (this is left for the future work).

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 at a daily/hourly level.

Pandas is a useful tool for working with dates, and has a `resample` method, which allows to aggregate the data to the hourly level, and to fill the gaps without 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, while 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

6 months of historical weather data was pulled, to potentially improve the prediction accuracy.

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: This service does not take any new registrations, after the 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 requires a `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.

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)

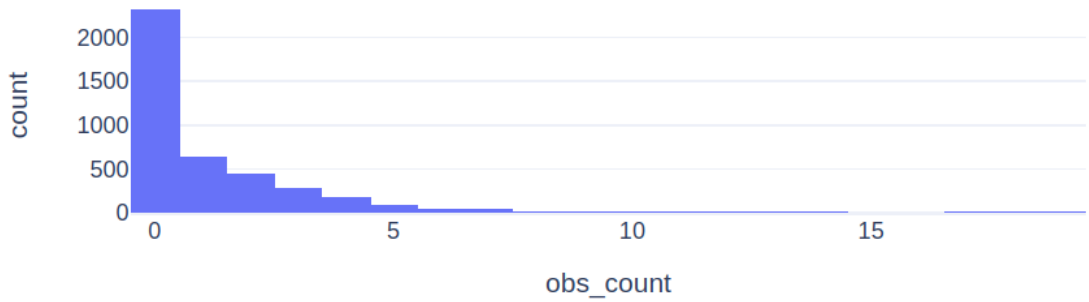
Once data with object counts is prepared, and the weather data collected, these two data sources can be combined using a common `timestamp` attribute. This merged dataset is now a basis for exploratory data analysis, and Machine Learning.

Target variable distribution

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

It is expected to be dominated by 0's, as during the night, and during the quiet periods of the day, it is common to see zero 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).

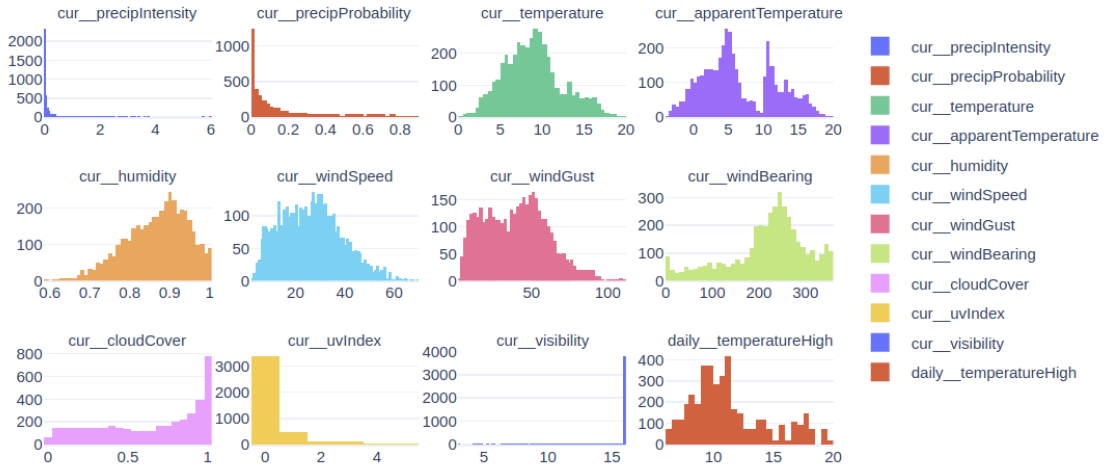
Missing values

One of the benefits of the manually collected dataset, is that there is no missing data. However, very often, 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 a multi-histogram plot, which shows numerical features related to weather data:

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 those, which assume that the features are normally distributed, or do not contain outliers.

The most common 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

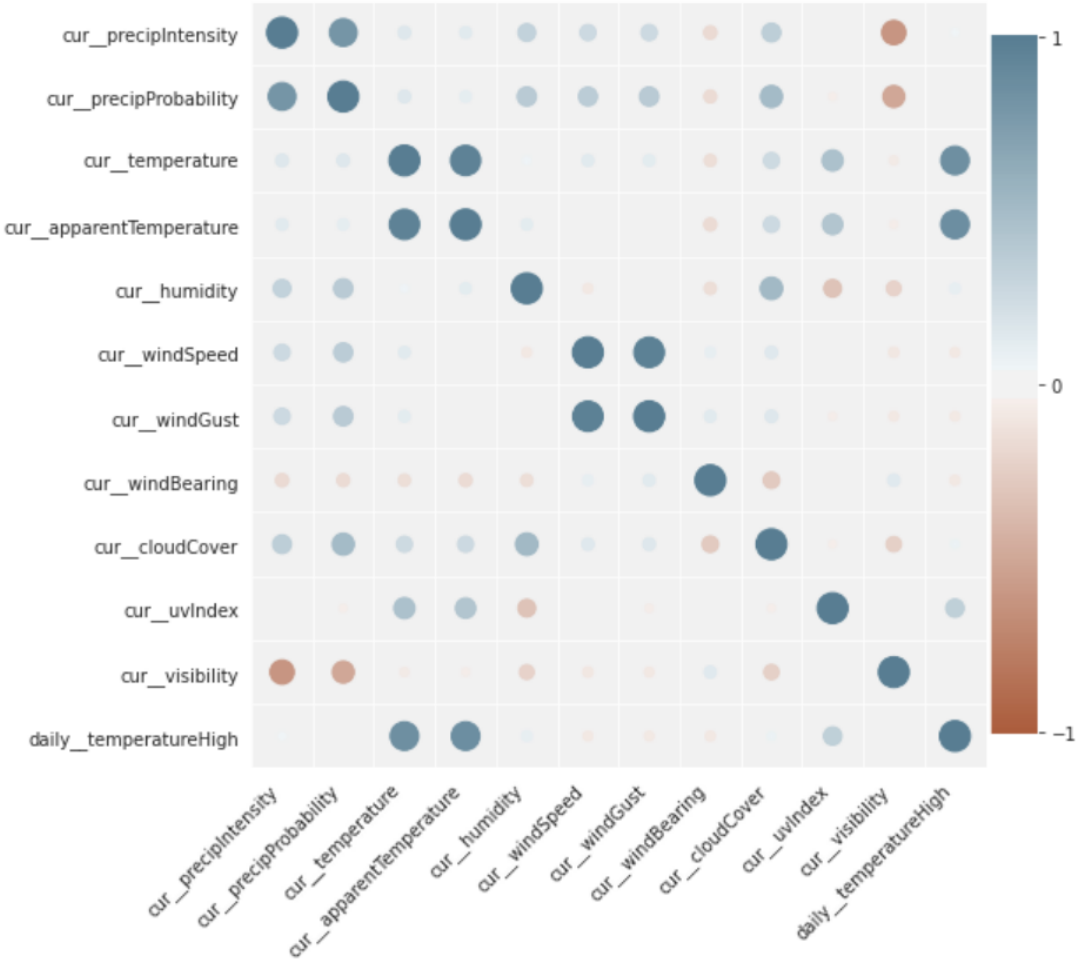
Sometimes it is also accepted to remove observations with outliers all together, however in case of forecasting it would mean a loss of the continuity in the data, and it is not recommended.

The good part of this exercise, is that tested Machine Learning algorithms, have produced the same results with, and without these transformations, and the decision was made to not transform the features.

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 this effect is to plot a correlation matrix:

Fig. 5.8. Feature Correlation



This kind of plot can be really useful, and it helps to discard features, which are too correlated with each other. Less features means improved training speed, and less complex models.

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

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

5.6. Predicting counts

Given the available dataset, it is now possible to generate a model, with a hope, that it will be able to predict future object counts with low errors.

Errors can be calculated in many different ways, but each model in this chapter will be judged based on the following metrics:

- 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:

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

5.6.1. Naive model

"All models are wrong, but some are useful" [30].

Naive method to generate predictions, is often a good starting point, 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 crude forecasting method.

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

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

Benefits:

- this model 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

Below are the values calculated for the Naive model:

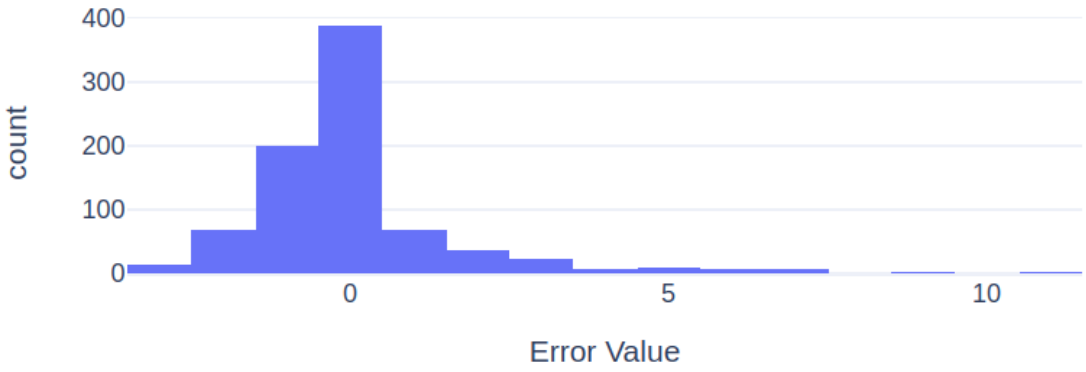
Tbl. 5.1. Error metrics - Naive model

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

In the metrics above, the errors are quite high. The R2 Score of 0.27 is considered a relatively poor indicator of the predictive power, but at the same time it is not equal to 0 (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.

Additionally, the model has overestimated a lot of values. This means that the model is biased. However, for the Forecast models in this research, there is no greater penalty for over or under-estimating values, therefore the goal should be to see low skewness in the distribution of errors.

This is of course 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.

Making Predictions:

The model can now be queried for an expected count, given an 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 expectation.

5.6.2. Machine Learning

Prediction of 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 nature of the target values is, that they are 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 via the *Poisson Process*, where one would estimate a *rate* (λ) of observations in a given time interval, and use a 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

Following models have been also tested without any tangible benefits:

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

5.6.3. Feature Selection for Machine Learning

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

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

K-Best features

The aim of this technique, is to use statistical methods, to test the linear relationships between the features, and the target variable.

For regression problem, this method uses a *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 that there is a correlation):

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

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

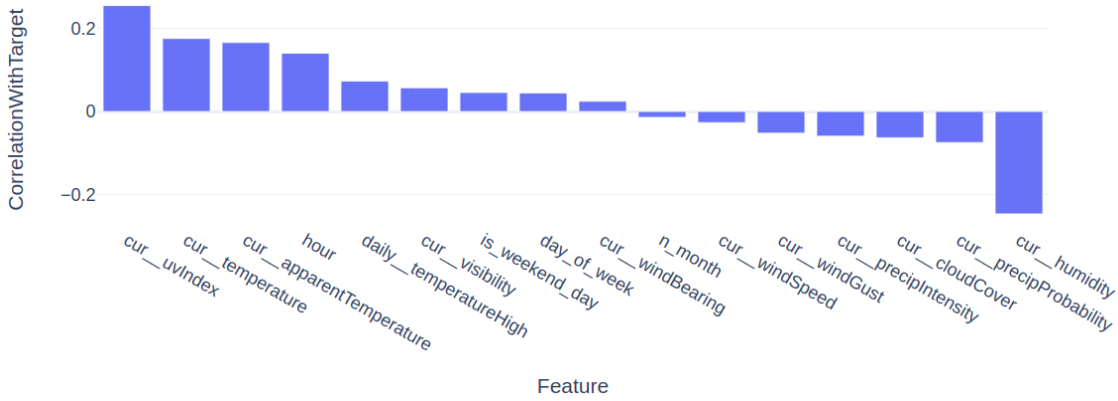
Plot below shows a jointplot between the best feature `cur__uvIndex` and `obs_count` (for 800 samples, which is 25% of the training dataset):

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.

Correlations between input features, and the target variable are shown below:

Fig. 5.11. KBest - Correlations With Target Variable



Drawbacks:

While it is advised in Machine Learning, 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 quickly.

Detail implementation, more techniques, and additional commentary for the plots above, can be found in the Extra Notebook 5.

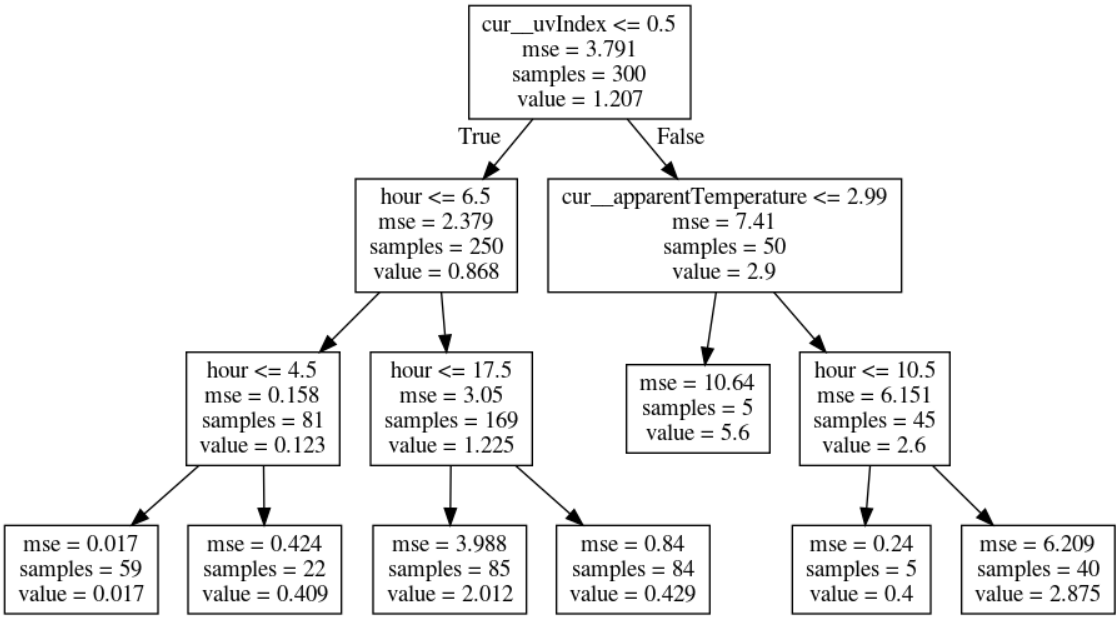
5.6.4. Decision Tree

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 (for Classification and Regression).

Decision Tree model searches for the best features and values, to split the dataset. The nodes in the tree fall into two groups, which separate the Target Variable values in the best possible way.

When trained on the Person object counts dataset, as expected, the algorithm chose to split the dataset by the *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



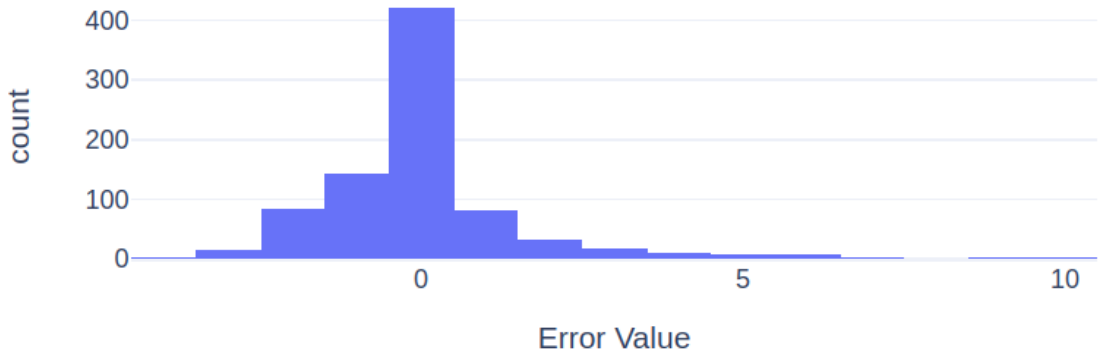
Below are the results achieved from running a 5-fold Cross Validation on this model:

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, and more efficient.

Gradient Boosting Trees are more advanced mathematically, and 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 well. It does not have to search across all feature values to find the best splits, but creates an N histogram bins for values 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

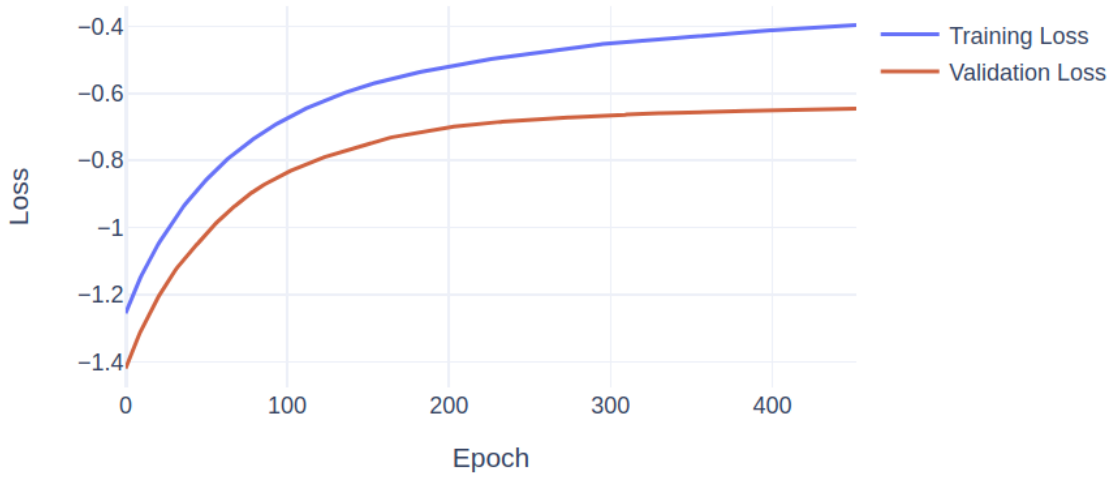
Metric	Score
MPD	1.25
MSE	2.47
MAE	0.87
R2	0.36
ACC	0.54

Looking at the R2 Score, this model achieves a significantly higher goodness of fit 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 the amount of computation 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 the 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

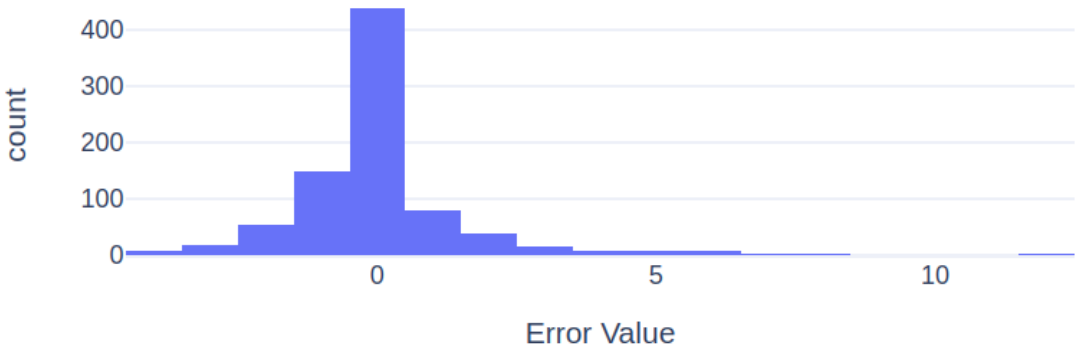


Drawbacks:

- 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. This also a benefit, as due to this characteristics, it is also more accurate
- the predictions from the model are point estimate, which means that this model does not return the uncertainty about the results. It is therefore difficult to reason about the predictions with low confidence

Error Distribution:

Fig. 5.15. Error Distribution - Gradient Based Decision Tree



Similar to vanilla Decision Tree model, the errors are distributed quite evenly.

5.6.6. Gaussian Process

The last, but not least model, tested in this chapter, is the Probabilistic Model, which uses Bayesian inference method, to update beliefs after observing the data.

RBF kernel is utilized as a covariance function, with 1.0 for `variance` and `lengthscale`, and a Poisson family function is used as a likelihood.

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

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

The number of optimized parameters is $N\_Features + 1$. It is composed of 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 can be a problem. The most popular choice, pymc3, consumed over $20GB$ of RAM on a Gaussian Processes with only single feature, and therefore GPy was used as an alternative method
- 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 observe numerical instabilities
- 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 mean squared error metric is the lowest out of all tested models
- 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 a possibility 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:

Below is the process of generating predictions using GPy:

- 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 of the predictions

Poisson process allows to use values for λ, to obtain integer counts from the predictions (and 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 the range of most probable counts.

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

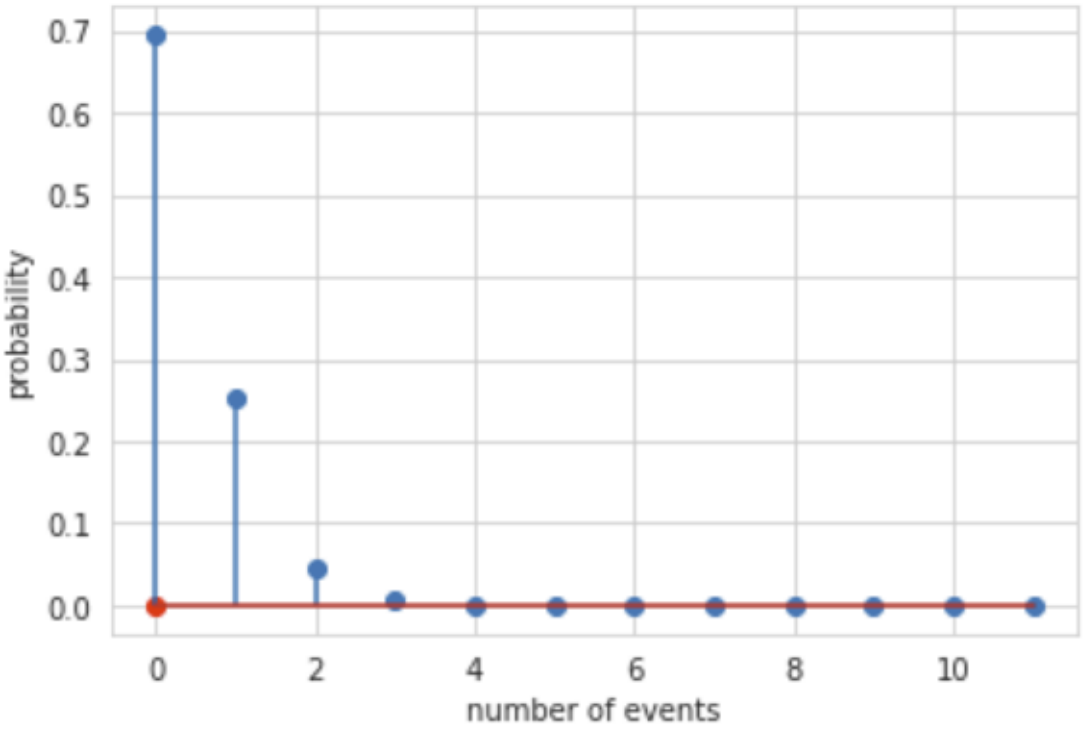
- choose a random predicted rate λ
- then Poisson probability mass function (pmf) can be used to generate probabilities for each number of events. This is calculated using the following equation:

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

- 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.

To actually 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 precipitation intensity of 0.8 (inches of of liquid water per hour, DarkSky), there is a 69% 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 model selection so much.

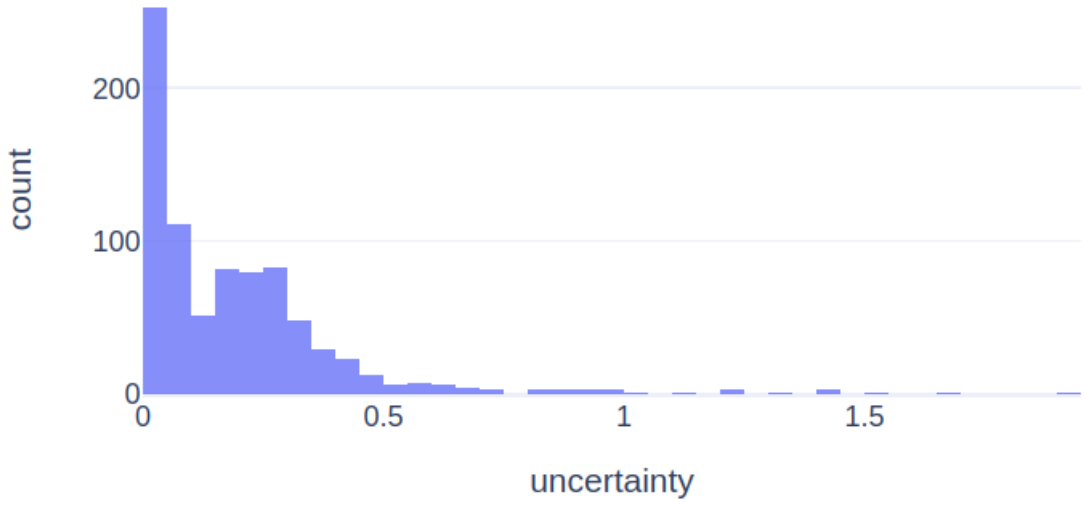
The error distribution is very similar to the Gradient Boosted Decision Tree model.

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 clear, that there is almost zero uncertainty around some predictions, and a little more in others. Overall the average uncertainty is 0.20.

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

Fig. 5.19. Low Uncertainty Predictions

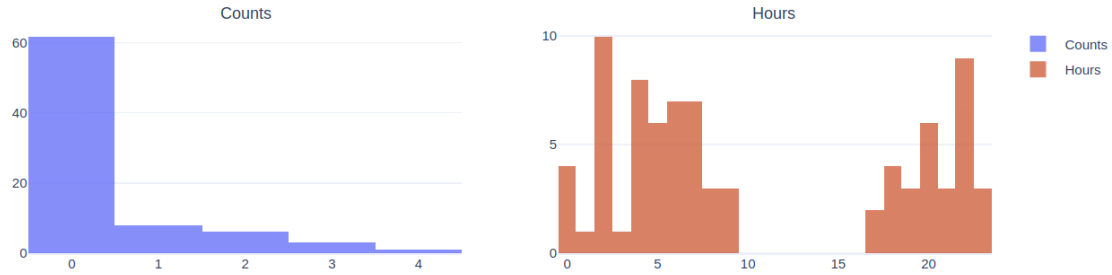
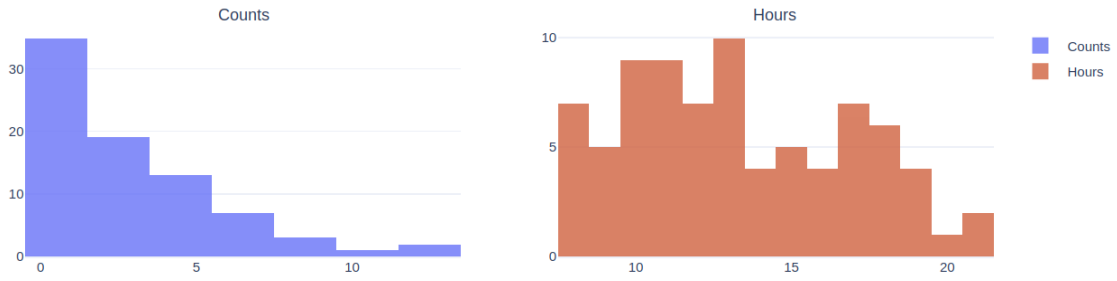


Fig. 5.20. High Uncertainty Predictions



Both groups above use 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 significant advantage of probabilistic models, versus the point estimate models found in 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]

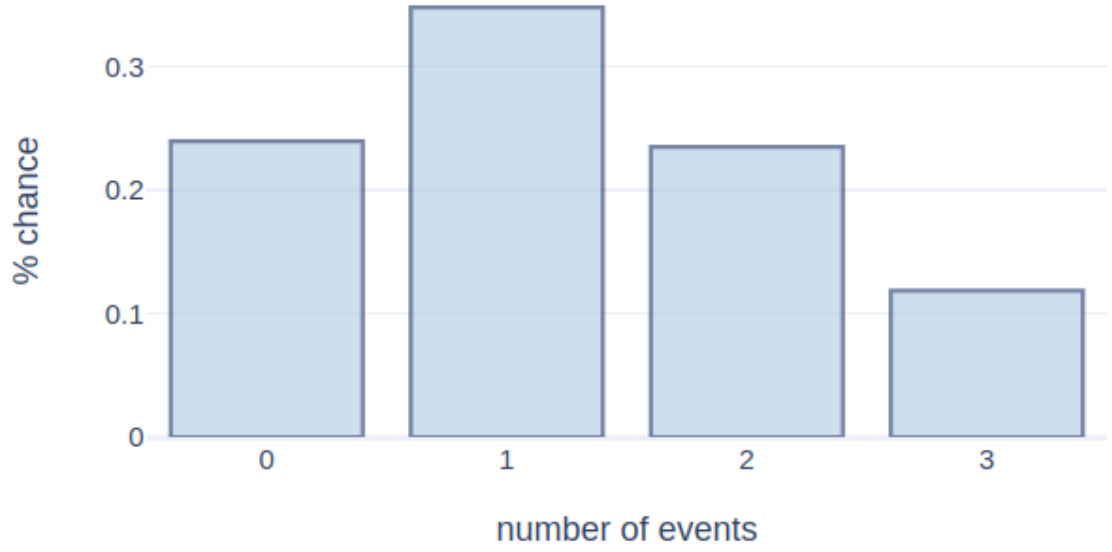
Then, another function is used to visualize this data to the user. Below are the bar charts for three different magnitudes of uncertainty:

Visualization for low σ (0.005):

Fig. 5.21. Low uncertainty predictions

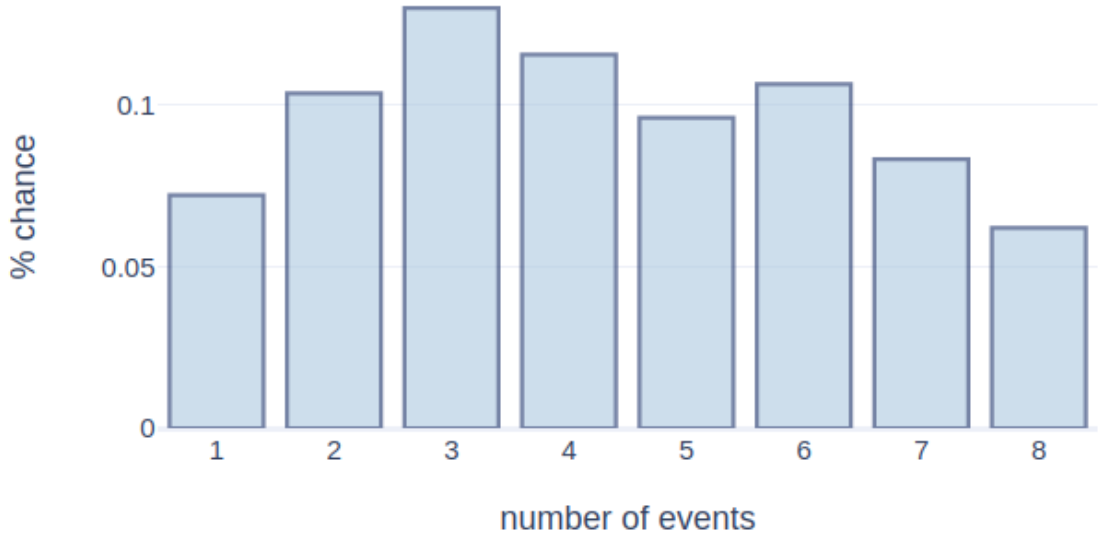
Visualization for medium σ (0.149):

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?

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

To obtain a 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?

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

To obtain a 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 (this is outside of scope of this research).

5.7. Conclusion

By use of object detections, Machine Learning can be used to make future predictions.

Evidence above shows that even though the accuracy of the models is far from being always correct, it is quite significantly better than using a naive approach of average counts per hour.

Model Summary:

In summary, all models described in this Chapter are quite robust, all have their benefits and trade-offs, and all ended up showing 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
- – suffers from poor results on the *Vehicle* object category

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 degrades other metrics.

However, since this is a Poisson process, perhaps Mean Poisson Deviance should be the most reflective of the model's performance, and in this case Gaussian Process achieves the lowest (best) score.

Recommendation:

Gaussian Process provided the most useful tools to generate meaningful predictions. It is clear what the model is returning, and the error rate for the *Person* object category is satisfactory.

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

However, at this point - after a limited number of attempts - Gaussian Process achieves quite poor results on the *Vehicle* object category.

This makes the *Gradient Boosting Regressor Tree* a strong competitor, as it is an order of magnitude quicker to train, allows to generate learning curves, and it is more robust to object category selection.

Next Chapter focuses on the Anomaly Detection problem.

index | prev | next

```
<IPython.core.display.HTML object>
```

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 were identified as the most useful, given the available dataset:

- based on unusually high count of objects in a given hour
- based on the raw content of the images, which are too different from the norm

6.1. Anomalies estimated from event counts

Object detections can be flagged as anomalies, by an unusually high number of events in a given hour, for an object category (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 would compare a number of already registered objects in an hour, versus a threshold, to determine if the next object should be classified as anomalous. If that limit is breached, a notification could be triggered to the users.

These minima and maxima thresholds, could also be displayed in the forecast as a vertical bars, to make the alerts more explainable.

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

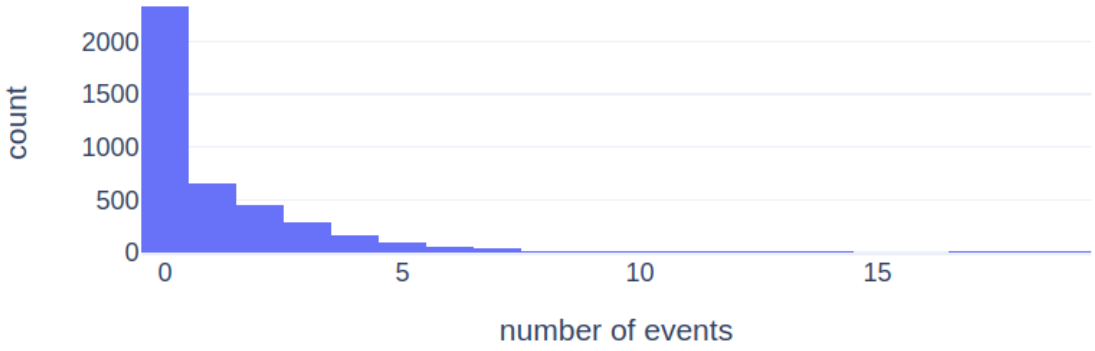
This section is focused around three ways of solving this problem:

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

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

Below is the count distribution for the *Person* class:

Fig. 6.1. Person Count Distribution - All Hours

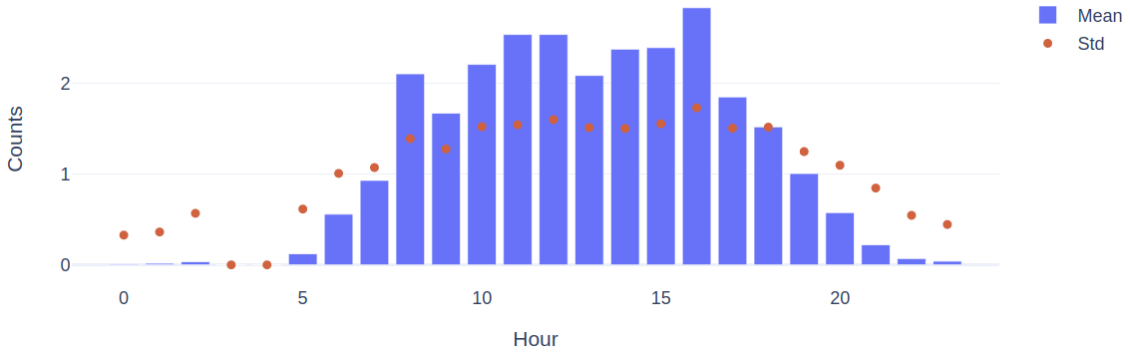


Overall, this data is heavily skewed towards 0's, but this is expected. During the night or when it is dark, the number of objects is 0, as the camera does not have the night-vision capability. At other time intervals, there is just little activity taking place.

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.

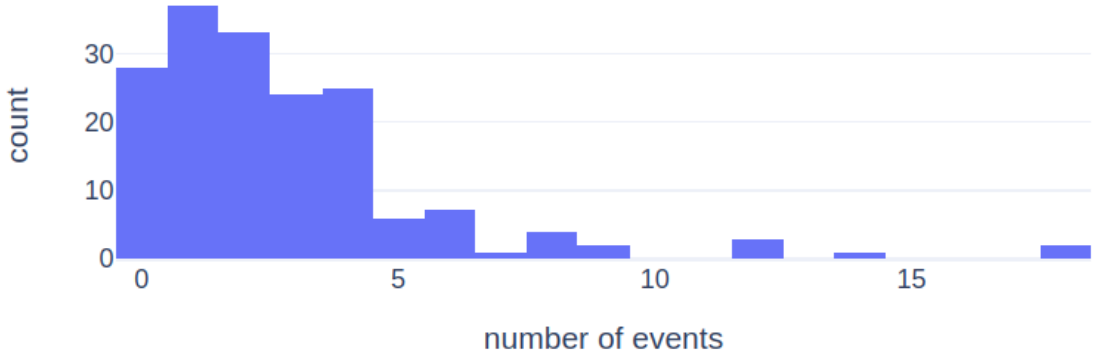
Fig. 6.2. Person Count Distribution By Hour



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

For example, 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 commonly utilized is IQR (Interquartile Range).

This method is used in the boxplots [31]. It is simple to explain, well understood and often produces satisfactory results.

First, one would calculate an *Interquartile Range (IQR)* by the use 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:

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

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

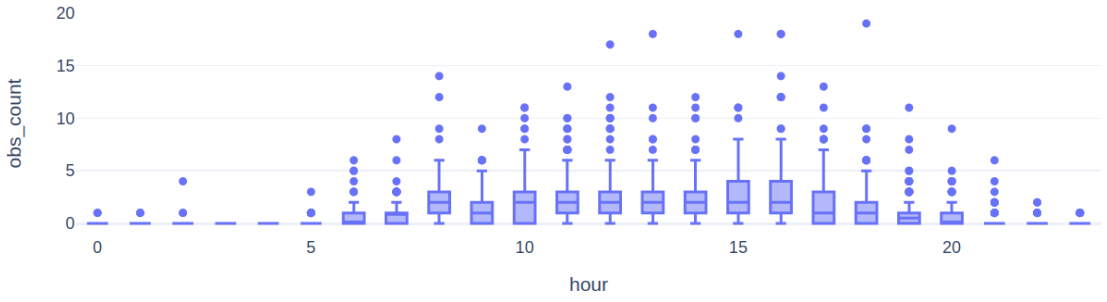
And finally, values below lower or above upper bound are classified is outliers:

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

, 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* [32], boxplots suffer from False Positives in heavily skewed datasets.

6.1.2. Adjusted boxplot for skewed distributions

In Hubert's research, 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.

These calculations, applied to the count dataset, classified 148 observations as outliers, which represents a large percentage of the dataset - 3.5%.

6.1.2. Z-Score

Z-Score determines, how many standard deviations σ, the points X, are 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, and the uncertainty

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. Then it performs an analysis of the posterior, which contains N-samples drawn from the distribution.

In relation to the count dataset, a two suitable distributions have been identified, 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.

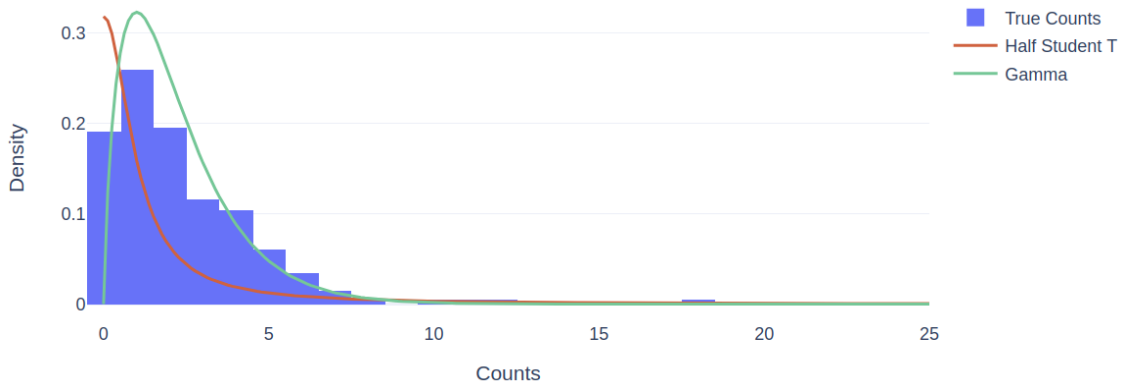
- Gamma distribution, with parameters $\alpha = 1.5$ (shape), and $\beta = 0.5$ (rate), and density function:

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

, 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. Prior 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 component 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 a STAT 504 course from Penn State [33], 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 trivial with the 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 the results are analyzed, below is the intuition around sampling 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 [34], 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:

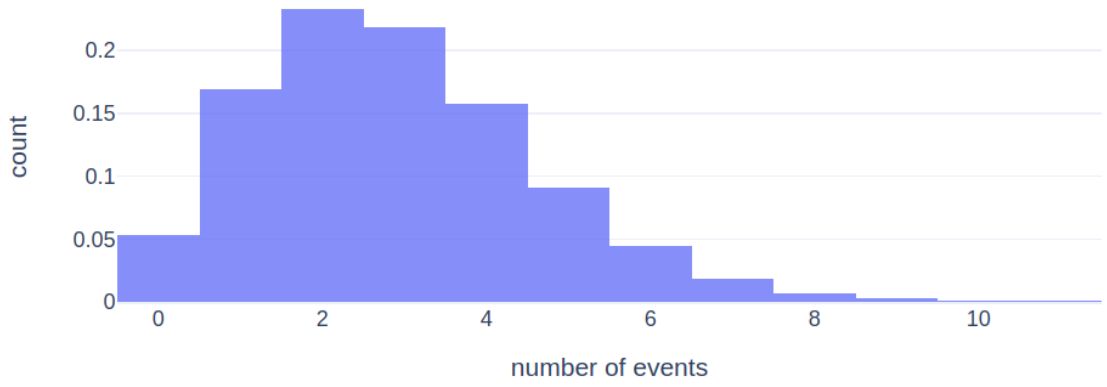
Fig. 6.6. PyMC3 - Posterior Stats

	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



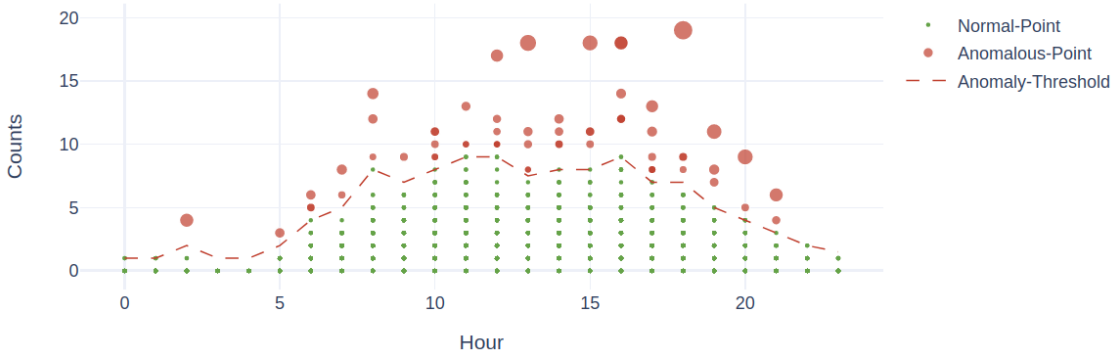
For example, if $K = 8$, then there is 1% chance to see an 8 or more objects at 4PM.

The final step 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, a plot has been generated, 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.

In the future work, it would be an interesting experiment, to calculate the fences using the rates estimated by the prediction model from the Forecasting Chapter. In this method, the fences 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 [35], 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*.

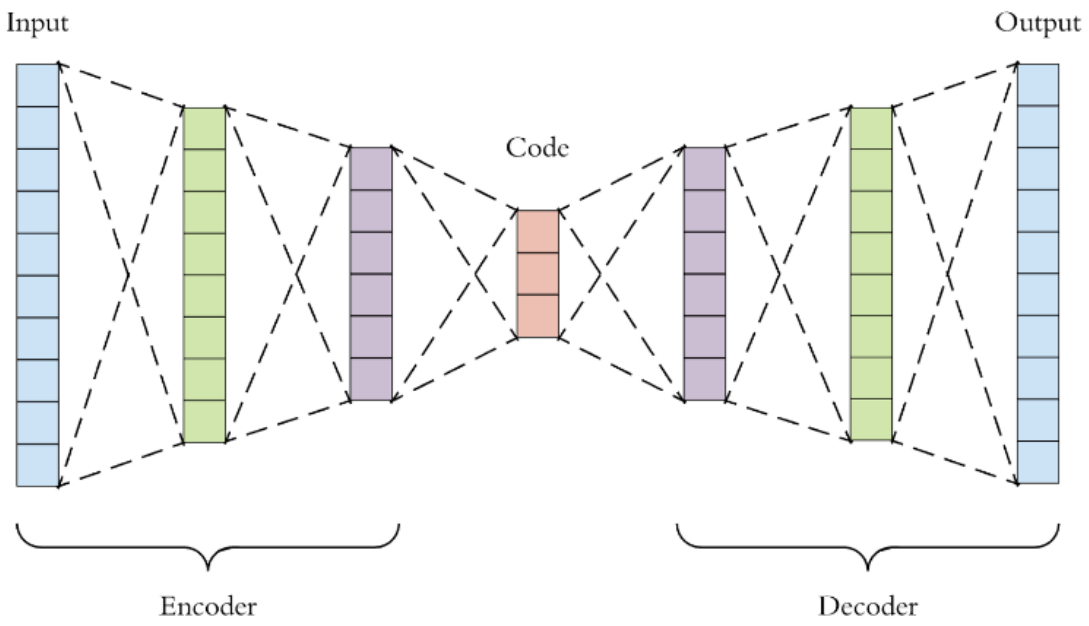
Using unsupervised learning has been chosen mainly for two reasons: - there are over $600K$ images collected in the process and they do not contain any anomaly-related labels - anomaly detection usually deals with highly imbalanced datasets (where potentially less than 1% represents anomalous images),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

A Convolutional Auto Encoder is used in this section, as CNN is often selected for image data (Yolo already used CNN for object detections in Chapter 4).

Fig. 6.9. Auto-encoder Diagram



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

In contract to section 6.1. above, auto encoders will use the data for all object categories 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.

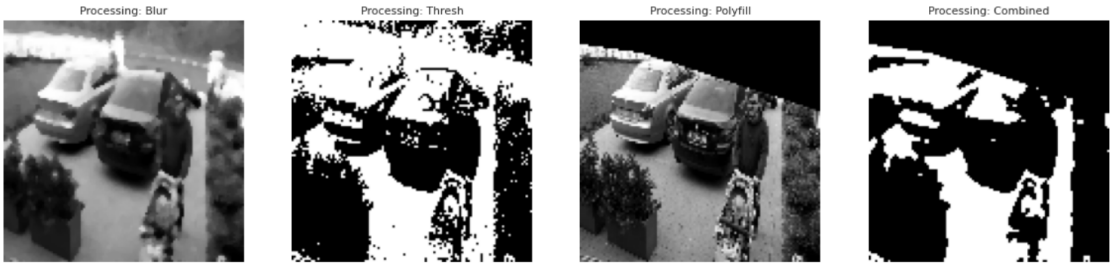
A function has been defined called `process_frame`, which allows to perform morphological operations on images through flags as function arguments. It was partially inspired by an article on the popular Computer Vision blog - PyImageSearch [36]:

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

Below is an example of an original image (6.11.), and the morphological operations applied to it (6.12.):

Fig. 6.11. Original Image

Fig. 6.12. Pre-processed Image



The dataframe at the start, only contains a list of all images stored on the disk. Images need to be extracted into a raw pixel 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

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 the conducted experiments, an increase in 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 the 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, only image size 56×56 is considered in the rest of this Notebook. It is a good trade-off between too small and too large.

In addition, $10K$ images are used for training, as it is the lowest number of images, which produces best final results.

The statistics from the use of different numbers of images, and resolutions are provided below, 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.

The data normalization step to the $0 - 1$ range, 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 the image data (data type becomes a `float32` type):

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

The last step 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. The 0.8/0.2 split is chosen, with a `random_state` parameter set to a constant value for reproducibility.

The shape of the training data, which is ready for auto encoder training, is $(8000 \times 56 \times 56 \times 1)$.

6.2.2. Training with auto encoder

The architecture of the convolutional auto-encoder below, is inspired by the PyImageSearch blog entry [36]. 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

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

After many experiments with different model parameters, the conclusion has been made, that this architecture is quite optimal in its original shape:

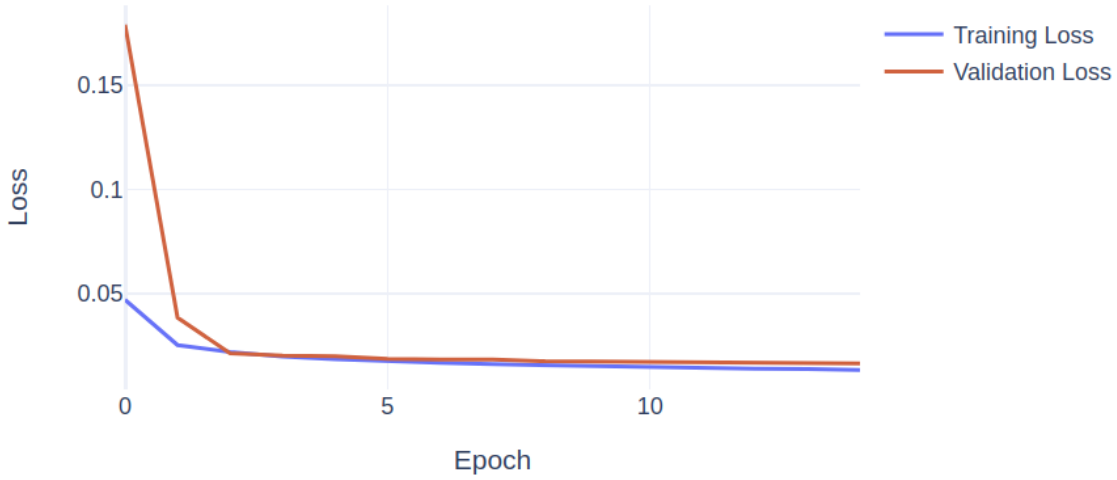
Fig. 6.13. Auto Encoder Summary

```
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 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 that model was discarded at this stage)
- 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).

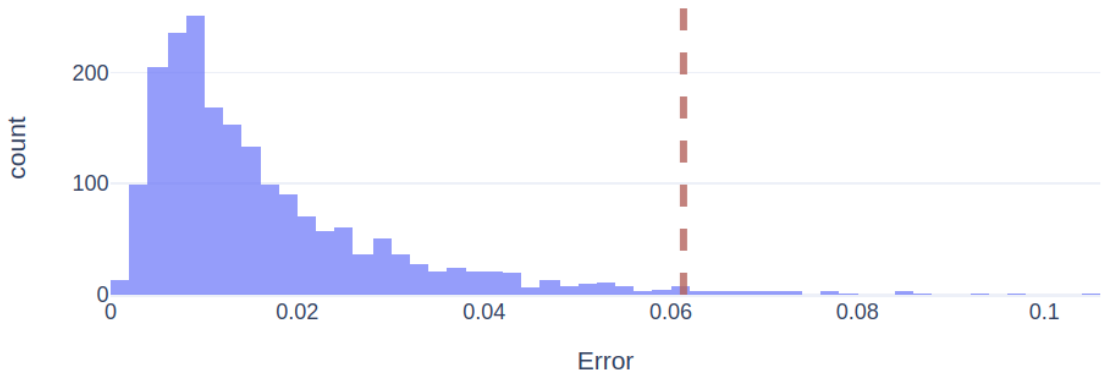
To make predictions for all 2000 test images, and to calculate the errors, only 0.25 second is required.

The next step is a decision point, to determine the percentage of observations, which should be classified as anomalous. For example let this percentage be 0.99.

Is possible to calculate a threshold for the mean squared error, using the 99th quantile, above which, points are classified as anomalies. This threshold value is 0.0651.

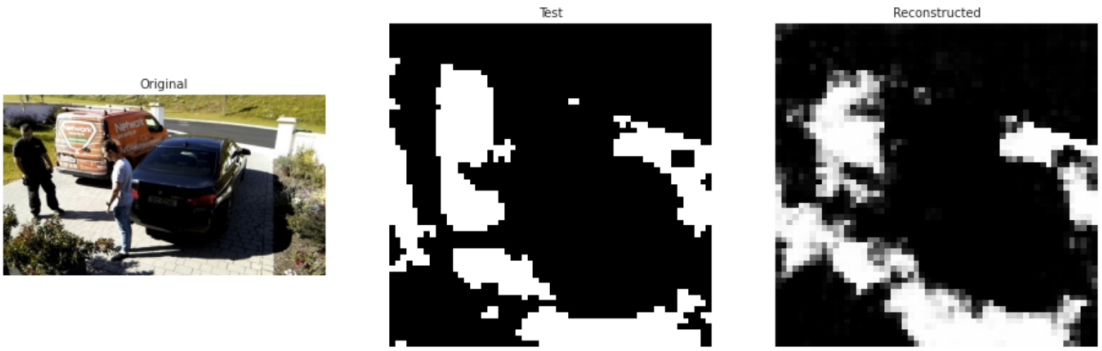
Below is a histogram, which shows the distribution of errors with a red dashed line, which represents the calculated threshold:

Fig. 6.15. Auto-encoder - Error Threshold



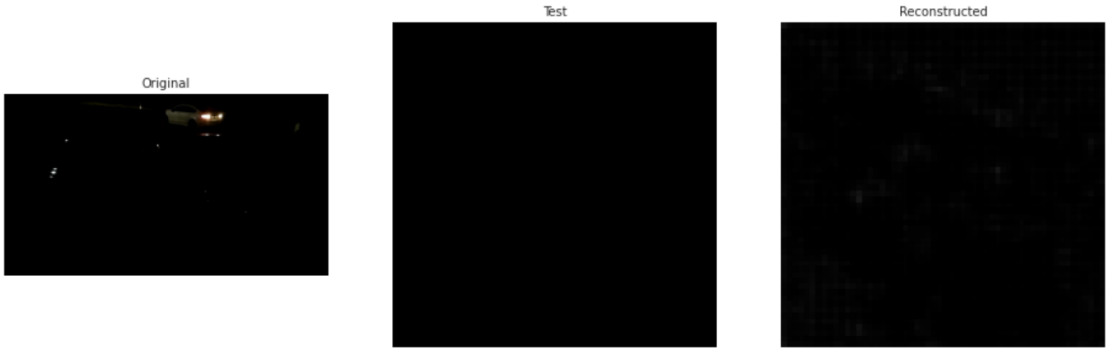
To make sure the method is working, it can be very helpful to look at the images with the largest error (they would be classified as anomalous), an example is below, which shows the original image (left), with the preprocessed (middle) and reconstructed one by auto encoder (right):

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



It appears to be working for the anomalous image (as there is a lot of busy pixels in the 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 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.

What can be said about the output of this procedure?

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 choice would be to train on a few rolling months. This would help if the region of interest changed 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 30 images have been manually annotated:

- 15 as non-anomalous
- 15 as anomalous

Procedure:

The process is almost the same as error calculation in the previous step (a helper function called `test_anomalies` was defined 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:

The labels are now available, and the classification metrics to evaluate model's performance can be utilized [37]:

- 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, which shows the previous error distribution, where the red dashed line represents the anomaly threshold, and a set of short red and green lines, which show the 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.

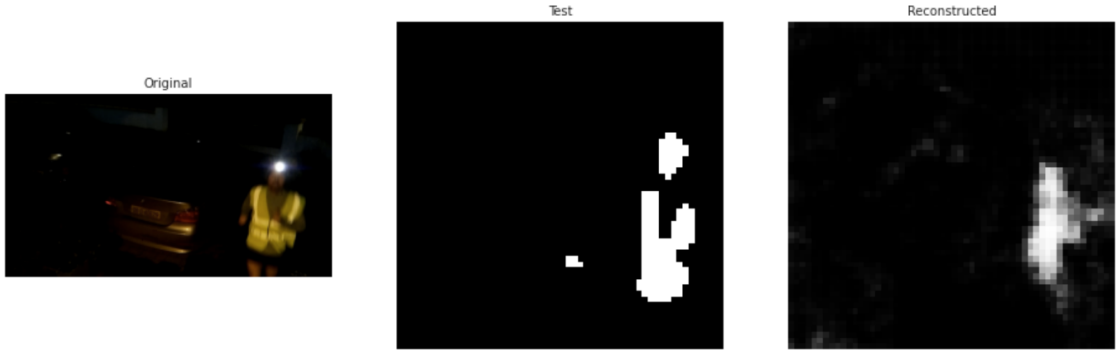
Below are the metrics for using the current model, trained previously:

- 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 the darkness level, and increase error sensitivity during the nightly hours. Below is an example of an 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 the 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

The evidence shows that the 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 the object detection data, and more opportunities are certainly available.

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, which used probabilistic programming and Poisson distribution characteristics.

Then, paragraph 6.2. showed promising capabilities for the real time image scoring, via auto-encoders.

There are many other techniques, which would be worth exploring in the future. For example, to utilize the actual forecast from chapter 5 in section 6.1., and to switch the auto-encoder type to a Variational Auto Encoder in section 6.2, for improved predictions.

The next chapter contains the final conclusions, which summarize this research as a whole, and more call-outs for the future opportunities and improvements.

index | prev | next

7. Conclusions and Future Considerations

index | prev | next

Below are the key conclusions to the three research questions in this study:

1. What is the level of complexity, required to build a fast, and reliable object detection pipeline, using *IOT devices* and *Computer Vision*?

A reliable data collection stage manifested itself with a high complexity. 6 months of image capture posed various challenges, and led to the following insights:

- It is crucial to place the camera in the right location. It may require wiring the house with the ethernet cables, and an investment in Power Over Ethernet adapters. Camera units (and *IOT* devices) placed outside of the house, need to be monitored against environmental effects: direct exposure to sunlight, humidity, dust, dirt, insects and even birds. All of them can have a negative impact on the picture quality
- Multiple tasks performed on each frame from the camera, may introduce processing latency. Motion sensing (*Background Subtraction*) with suitable parameters, and fast object detector (*Yolo v2*), can eliminate this problem
- Smooth transmission of *High Definition* images to a web browser, can be achieved by using *web sockets* in a separate Python thread
- Software services need to start automatically when devices are rebooted, or when network connections are broken. Utilizing *Supervisor* Linux utility, and a proper network setup can minimize the loss in data

2. Given the dataset with collected images, can the future object counts be accurately predicted using *Machine Learning*?

Object counts for a given category (*Person* or *Vehicle*), can be predicted with relatively low error rates using Machine Learning models.

This process requires a significant amount of image data extraction, cleaning and pre-processing. Numerous models of different type and complexity, have been tested (ranging from *Linear Regression* through *Bi-Directional LTSM Neural Networks*).

Given the evidence gathered in Chapter 5, there are two types of models, which can be successfully applied to make predictions: a probabilistic model (*Gaussian Process*), and a point estimate model (*Histogram-Based Gradient Boosting Regressor*).

While Gaussian Processes have an advantage of providing uncertainty about the predictions, Gradient Boosting models are faster to train, and more robust to the object category selection.

3. Can *Anomaly Detection* algorithms assist in recognizing anomalous patterns in the object detection data?

Applying anomaly detection algorithms to the collected image data, can generate useful results.

Hourly threshold estimation Estimating a maximum number of objects per hour, allows to flag anomalies above that threshold. Each object category, like Person or Vehicle, is analyzed individually.

Probabilistic approach, which utilizes *gamma* distribution and *Poisson* likelihood function, produces an optimal result and classifies 61 out of 4140 observations as anomalous.

Raw image classification The second methodology applies an Auto Encoder Neural Network directly to raw image data. This technique is categorized as *Unsupervised Machine Learning*, as the historical images are not labeled. In contrast with *hourly threshold estimation*, multiple object classes are considered inside a single model.

The inner workings of this method, is to search for images, which differ the most from the others, using raw pixel data. This technique presents two opportunities:

- An alert can be triggered, if an incoming image deviates outside of a threshold (calculated using *mean squared error*). In an experiment, a gathering of people outside of the house, was successfully flagged as an anomalous event
- Time spent of manual image analysis, can be significantly reduced, by sorting an image collection using the anomaly threshold, in a descending order. Additionally, this approach should lower the risk of missing an important event

In a model evaluation stage, a hand-labeled dataset with 30 images was used. The best model model was able to classify 9 out of 15 anomalies correctly. It obtained a *Recall score* of 0.6, while not sacrificing the *F1 score* of 0.72.

Recommendations for future work

By developing a Minimum Viable Product, incorrect assumptions and potential weaknesses, can be quickly identified in the core features. The *MVP* should also include a basic user interface, with a good representation of forecast and anomaly data.

Further future recommendations are summarized below:

- Modern AI systems should emphasize ethics and protect privacy. Privacy mode should at least blur people's faces, or even full silhouettes, if required
- To prove that the system is truly generalizable, it should ideally be deployed in another household
- Anomaly detection based on hourly threshold estimation, can be significantly enhanced, by incorporating forecast data. Threshold estimated via forecast predictions, would carry additional information, like day of the week, and weather conditions
- Portability might potentially be strengthened, by allowing to consume an *RTSP* stream, instead of only *Message Queues*
- Security can be enhanced by an addition of waterproof casing, a camera with night vision mode, or even another camera looking at the same scene, but from a different angle
- Current strategy for counting objects is rather basic, and uses *Euclidean Distance*. To allow for more advanced object tracking, *Kalman Filter* could be utilized
- In the raw image classification, *Variational auto encoder* could replace the vanilla version. It would prevent overfitting, and ensure that the properties of latent space, optimize generative process
- New versions of Python libraries could improve performance, and reduce resource consumption
- Overall cost of the hardware, could potentially be significantly lowered, assuming that the *on-device learning* alone can achieve accurate results, and high performance
- Higher volume of collected data, would open up the possibility, to test other forecasting models, which can use periodicity and seasonality components
- After AI is deployed in production, it should be able to adopt itself, to the changes in the environment. This can be achieved by utilizing the most recent subset of detections for training data

Final remark

Use of AI in the Home Monitoring setting, is still quite underutilized. However, there is a potential for further adoption, due to relatively low hardware costs, and exponential progress in the fields of Computer Vision and Machine Learning.

While building modern AI systems, it is Engineers' responsibility to prioritize ethics, transparency and explainability. These factors will future-proof the design, against potential changes in law.

The proposed system can play an important role in enhancing the security of monitored objects, by utilizing valuable insights drawn from the collected data.

index | prev | next

8. Acknowledgements

index | prev | next

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, who made it all possible.

I would like to thank my supervisor, Dr. Alessio Benavoli, for all his guidance, his very accurate feedback, and his availability.

Additionally, a credit to be given to my friends from work, who have read the thesis (or even parts of it!), and pointed out the corrections and improvements (Phil O'Mahony, Brian Condon, Deepak Mehta, Shamsul Hassan).

Especially, a big thank you to Phil O'Mahony, for all our Jitsi calls, and numerous tips in Data Analysis, Forecasting and Anomaly Detection.

I would like to also thank my closest family, for understanding, when I was locked in the study room for weeks (or maybe it was two years actually?).

And finally a very special, warm thank you, to my wife, Anna, for all her support during the writing of this thesis, and for her help to transform my thousands of thoughts, into understandable English.

10. Appendices

index | prev

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

- LiteratureReview
- ExtractRawImageData
- ObjectCount
- FetchWeatherData
- Person-EDA-Forecast
- Person-AnomalyDetectionForHourlyCounts
- RawImagesAnomalyDetectionTraining
- DataCollectionPipeline
- ForecastTrainingScript
- GeneratePredictionsScript

References

[1] Derek Crippin. 12 telltale signs burglars are casing your home, 2020.

[2] Lawrence Roberts. *Machine Perception of Three-Dimensional Solids*. 01 1963.

[3] Marvin Minsky. *The Summer Vision Project*. 07 1966.

[4] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[5] David G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60(2):91–110, November 2004.

[6] C. Cortes and V. Vapnik. Support vector networks. *Machine Learning*, 20:273–297, 1995.

[7] Zoran Zivkovic. Improved adaptive gaussian mixture model for background subtraction. In *Proceedings of the Pattern Recognition, 17th International Conference on (ICPR'04) Volume 2 - Volume 02*, ICPR '04, page 28–31, USA, 2004. IEEE Computer Society.

[8] J. Redmon and A. Farhadi. Yolo9000: Better, faster, stronger. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6517–6525, 2017.

[9] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. pages 779–788, 07 2016.

[10] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. *arXiv e-prints*, page arXiv:1311.2524, November 2013.

[11] R. Girshick. Fast r-cnn. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1440–1448, 2015.

[12] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 91–99. Curran Associates, Inc., 2015.

[13] Jan Hosang, Rodrigo Benenson, and Bernt Schiele. Learning non-maximum suppression. 05 2017.

[14] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[15] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions, 2014. cite arxiv:1409.4842.

[16] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement, 2018. cite arxiv:1804.02767Comment: Tech Report.

[17] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. YOLOv4: Optimal Speed and Accuracy of Object Detection. *arXiv e-prints*, page arXiv:2004.10934, April 2020.

[18] Brady Neal, Sarthak Mittal, Aristide Baratin, Vinayak Tantia, Matthew Scicluna, Simon Lacoste-Julien, and Ioannis Mitliagkas. A Modern Take on the Bias-Variance Tradeoff in Neural Networks. *arXiv e-prints*, page arXiv:1810.08591, October 2018.

[19] James N. Morgan and John A. Sonquist. Problems in the analysis of survey data, and a proposal. *Journal of the American Statistical Association*, 58(302):415–434, 1963.

[20] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 3146–3154. Curran Associates, Inc., 2017.

[21] Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *Ann. Statist.*, 29(5):1189–1232, 10 2001.

[22] Wikipedia. Poisson distribution.

[23] Springer. *Central Limit Theorem*, pages 66–68. Springer New York, New York, NY, 2008.

[24] Peter Roelants. Multivariate normal distribution, 2020.

[25] Wikipedia. Cholesky decomposition, 2002.

[26] Wikipedia. Gaussian process.

[27] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[28] Arden Dertat. Applied deep learning - part 3: Autoencoders, 2017.

[29] Selva Prabhakaran. Mahalonobis distance – understanding the math with examples, 2019.

[30] George E. P. Box. Science and statistics. *Journal of the American Statistical Association*, 71(356):791–799, 1976.

[31] John W. Tukey. *Exploratory Data Analysis*. Addison-Wesley, 1977.

[32] M. Hubert and E. Vandervieren. An adjusted boxplot for skewed distributions. *Comput. Stat. Data Anal.*, 52(12):5186–5201, August 2008.

[33] Penn State Eberly College of Science. Analysis of discrete data.

[34] R. (Roger) Fletcher. *Practical methods of optimization*. Chichester ; New York : Wiley, 1987.

[35] Raghavendra Chalapathy and Sanjay Chawla. Deep learning for anomaly detection: A survey, 2019. cite arxiv:1901.03407.

[36] Adrian Rosebrock. Anomaly detection with keras, tensorflow, and deep learning, 2020.

[37] Guest Contributor. Understanding roc curves with python, 2019.