UCL Candidate Code: WPJP8

Module Title: Machine Learning in Smart Buildings

Module Code: BENV0119

This coursework comprises of three tasks using supervised learning, unsupervised learning and reinforcement learning.

For task 1 and task 2, we'll be using the publicly available REFIT smart home dataset. Full details and access to the data can be found here: https://repository.lboro.ac.uk/articles/dataset/REFIT_Smart_Home_dataset/2070091/1. The dataset contains sub-hourly readings from a variety of sensors within 20 smart homes close to Loughborough University in the East Midlands (UK).Climate data is also provided over the monitoring period May 2012 to October 2015. For this coursework, the data of Building 01 will be selected. Detail data are contained in the TimeSeriesVariable and REFIT_TIME_SERIES_VALUES files, which will be extracted. For task 1, both classification and regression tasks will be carried out.For task 2, we will be using k-means clustering to investigate the thermal conditions of a space in the building.

For task 3, we'll be working with OpenAI's gym, specifically with the "MountainCar-v0" environment. We will be exploring how reinforcement learning may be applied to smart buildings, and how to implemente the Q-learning algorithm.

1. Supervised Learning

1.1. Introduction and Aims

Supervised learning is an approach to creating artificial intelligence (AI), where a computer algorithm is trained on input data that has been labeled for a particular output. The model is trained until it can detect the underlying patterns and relationships between the input data and the output labels, enabling it to yield accurate labeling results when presented with never-before-seen data.

Supervised learning is good at classification and regression problems, and in this coursework both of them will be investigated. For our REFIT dataset, we will be primarily exploring gas volumes and investigating factors might have influenced gas consumption. This will comprise of two subtasks:

- subtask A: Compare the performance of two or more ML classification algorithms for predicting whether gas is being used or not. Potential algorithms include logistic regression, SVM, random forest, decision trees, KNN and Naive Bayes.

- subtask B: Compare the performance of two or more ML regression algorithms for predicting gas consumption in real smart home data. Potential algorithms include multiple linear regression, SVM, neural networks, decision tree regression, and lasso regression.

The selection of algorithms will be discussed in the following sections. When comparing the performances of algorithms, both accuracy and run time will be considered. In summary, the aims and objectives of task 1 are to:

- train the classification/regression models using the training dataset to predict the hourly energy consumption of homes using input features
- evaluate the accuracy and run time of regression algorithms applied using the testing and validation datasets
- conduct hyper parameter tuning and establish the best performing algorithms for this data

1.2. Theory

In this task, the performance of SVM, decision trees and random forest will be compared in terms of the pridiction of classification of gas consumption, and the performance of multiple linear regression and decision trees will be compared in terms of the prediction of regressino of gas consumption. Here, the theory behind these algorthims is described.

1.2.1 Feature Selection

Feature selection is crucial for model training. In this coursework, the features (air temperature, surface temperature, electrical power, etc) that may influence gas consumption are unknown. It is thus important to understand the correlations between data, so that the most relevant features can be selected for training. There are many methods to investigate the correlations between data, including pandas.Dataframe.corrwith, mutual information, etc. In this task we will be using corrwith, as the datasets are pretty straightforward. We will also be understanding these correlations through scatter plots.

1.2.2 Subtask A: SVM, decision tree and random forest

The algorithms of this task are chosen based on their appropriateness. Detailed mechanisms, principles and pros & cons of these algorithms can all be found in the scikit-learn documentation (https://scikit-learn.org/stable/).

We will be first exploring SVM algorithms. Support vector machines (SVMs) are a set of supervised learning methods used for classification, regression and outliers detection.At first approximation what SVMs do is to find a separating line(or hyperplane) between data of two classes. SVM is an algorithm that takes the data as an input and outputs a line that separates those classes if possible.

After an initial training of the SVM model, a model score is calculated, and hyperparameters of the model are tuned with cross validation algorithms. Cross-Validation is essentially a technique used to assess how well a model performs on a new independent dataset.The

simplest example of cross-validation is when we split your data into three groups: training data, validation data, and testing data, where we see the training data to build the model, the validation data to tune the hyperparameters, and the testing data to evaluate your final model.

When evaluating different settings (“hyperparameters”) for estimators, such as the C setting that must be manually set for an SVM, there is still a risk of overfitting on the test set because the parameters can be tweaked until the estimator performs optimally. This way, knowledge about the test set can “leak” into the model and evaluation metrics no longer report on generalization performance. To solve this problem, yet another part of the dataset can be held out as a so-called “validation set”: training proceeds on the training set, after which evaluation is done on the validation set, and when the experiment seems to be successful, final evaluation can be done on the test set.

However, by partitioning the available data into three sets, we drastically reduce the number of samples which can be used for learning the model, and the results can depend on a particular random choice for the pair of (train, validation) sets.A solution to this problem is a procedure called cross-validation (CV for short). A test set should still be held out for final evaluation, but the validation set is no longer needed when doing CV. In the basic approach, called k-fold CV, the training set is split into k smaller sets (other approaches are described below, but generally follow the same principles).

In this task, we will have high dimensional data sets and the number of features is not greater than the number of samples, and therefore the adoption of SVM seems appropriate. After implementing SVMs, we will then explore decision trees. Decision Trees (DTs) are a non-parametric supervised learning method used for classification and regression. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. A tree can be seen as a piecewise constant approximation.

The benefits of decicision trees are that they are easy to understand, perform classification without requiring much computation, and provide a clear indication of which fields are most important for prediction or classification. However, decision trees are prone to errors in classification problems with many class and relatively small number of training examples.

While decision trees are common supervised learning algorithms, they can be prone to problems, such as bias and overfitting. However, when multiple decision trees form an ensemble in the random forest algorithm, they predict more accurate results, particularly when the individual trees are uncorrelated with each other.

Random forest algorithms have three main hyperparameters, which need to be set before training. These include node size, the number of trees, and the number of features sampled. From there, the random forest classifier can be used to solve for regression or classification problems.

1.2.3 Subtask B: Multiple Linear Regression and Decision tree regression

We first explore multiple linear regression.Multiple linear regression (MLR) is a statistical technique that uses several explanatory variables to predict the outcome of a response variable. The goal of multiple linear regression is to model the linear relationship between the explanatory (independent) variables and response (dependent) variables. In essence, multiple regression is the extension of ordinary least-squares (OLS) regression because it involves more than one explanatory variable.

Formula of multiple linear regression

$$y(\theta) = \theta_{(0)} + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \ldots + \theta_n x_n + e$$

where $y(\theta)$ is independent variable, *x{n} is explanatory variable, θ*{(0)} is y-intercept,θ_{n} is slope coefficient for each explanatory variable, and e is the models' error term.

The multiple regression model is based on the following assumptions:

- There is a linear relationship between the dependent variables and the independent variables
- The independent variables are not too highly correlated with each other
- $y(\theta)$ observations are selected independently and randomly from the population
- Residuals should be normally distributed with a mean of 0 and variance σ
- The coefficient of determination (R-squared) is a statistical metric that is used to measure how much of the variation in outcome can be explained by the variation in the independent variables. R2 always increases as more predictors are added to the MLR model, even though the predictors may not be related to the outcome variable.

R2 by itself can't thus be used to identify which predictors should be included in a model and which should be excluded. R2 can only be between 0 and 1, where 0 indicates that the outcome cannot be predicted by any of the independent variables and 1 indicates that the outcome can be predicted without error from the independent variables.

When interpreting the results of multiple regression, beta coefficients are valid while holding all other variables constant ("all else equal"). The output from a multiple regression can be displayed horizontally as an equation, or vertically in table form.

We then explore decision tree regression. To construct a decision tree, we start by specifying a feature that will become the root node. Typically, no single feature can perfectly predict the final classes; this is called impurity. Methods such as Gini, entropy, and information gain are used to measure this impurity and identify how well a feature classifies the given data. The feature with the least impurity is selected as the node at any level. To calculate Gini impurity for a feature with numerical values, first sort the data in ascending order and calculate the averages of the adjoining values. Then, calculate the Gini impurity at each selected average value by arranging the data points based on whether the feature values are less than or greater than the selected value and whether that selection correctly classifies the data. The Gini impurity is then calculated using the equation below, where K is the number of classification categories and p is the proportion of instances of those categories.

Lasso regression is then adopted. The Least Absolute Shrinkage and Selection Operator is abbreviated as “LASSO.” Lasso regression is a type of regularisation. It is preferred over

regression methods for more precise prediction. This model makes use of shrinkage which is the process by which data values are shrunk towards a central point known as the mean. L1 regularisation is used in Lasso Regression. It is used when there are many features because it performs feature selection automatically. The main purpose of Lasso Regression is to find the coefficients that minimize the error sum of squares by applying a penalty to these coefficients.

sse = np.sum ((y-b1x1-b2x2-...-bo) *2) + (alpha (|b1|+|b2|+|b3|+...+|bo|))

1.3. Methods and procedures

For each of the task basic steps for machine learning will be carried out:

1. problem/ data identification
2. data preprocessing and feature selection
3. selection of algorithm/algorithms
4. split data,select & train models
5. model testing and evaluation
6. hyperparameter tuning
7. model optimization

1.3.1. Import the packages and modules required for this task

In [12]:
```
# import packages related to data processing
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
import seaborn as sns
from scipy import stats

%matplotlib inline

import warnings
warnings.simplefilter(action = 'ignore')
```

1.3.2. Read and extract the REFIT smart home and TimeSeriesVariable data

In [13]:
```
# read in the smart home data
data = 'REFIT_TIME_SERIES_VALUES.csv'
df = pd.read_csv(data)

# read in the time series data
data_ts = 'TimeSeriesVariable.csv'
df_ts = pd.read_csv(data_ts)

# select key Building01 data: building id, timeseries id & variable type
df_ts_KeyData = df_ts.loc[11:146,['Build_id','id','SensorFK','intervalLength','var

# display key data
df_ts_KeyData.head()
```

Out[13]:

	index	Build_id	id	SensorFK	intervalLength	variableType
0	11	Building01	TimeSeriesVariable1554	11.0	30.0	Gas volume
1	12	Building01	TimeSeriesVariable1584	12.0	30.0	Electrical power
2	13	Building01	TimeSeriesVariable41	13.0	15.0	Air temperature
3	14	Building01	TimeSeriesVariable42	14.0	15.0	Air temperature
4	15	Building01	TimeSeriesVariable43	15.0	15.0	Air temperature

We can see that there are 136 datasets of Building01, and the first one is Gas volume (see in variableType). Other features include elctrical power, air temperature, open/closed, on/off, alarm, etc. The detailed data of these variables are stored in 'REFIT_TIME_SERIES_VALUES.csv', and the indexes of them can be read from "id" (TimeSeriesVariable1554, etc). The meanings of these data, incluing where and how the data are monitored, can be accessed from other documents. The interval lengths of the features also vary, with some being 30 min, some being 15 min, and some being varied. We first define a function to extract data series from 'REFIT_TIME_SERIES_VALUES.csv' that correspond to the TimeSeriesVariable id.

In [14]:
```
#define a function to extract a particular data series from the data

def extract_data(var_id = 'TimeSeriesVariable15', col_name = 'Temp'):

    #select the data to use
    some_values = [var_id]
    df_sel = df.loc[df['TimeSeriesVariable/@id'].isin(some_values)]

    #set the DateTime as a date time object and the index
    format = '%Y-%m-%d %H:%M:%S'
    df_sel['dateTime_ix'] = pd.to_datetime(df_sel['dateTime'], format=format)
    df_sel = df_sel.set_index(pd.DatetimeIndex(df_sel['dateTime']))

    #sel only useful vars
    df_sel = df_sel[['data']]
    #set the col name
    df_sel.rename(columns={'data': col_name}, inplace=True)

    return df_sel

# Suppress the SettingWithCopy warning
pd.set_option("mode.chained_assignment", None)
```

We then extract lists of variables and their indexes, and use a dictionary to write them into the format of "Parameter_{}", so that the data can be more conveniently accessed.

In [15]:
```
## Extract all relevant data
# create lists of timeSeries and variableType
timeSeries =[]
variableType = []

# apply timeSeries and variableType data
for i in range(len(df_ts_KeyData.index)):
    timeSeries.append(df_ts_KeyData.loc[i,'id'])
    variableType.append(df_ts_KeyData.loc[i,'variableType'])

# extract all the data and put them in a dictionary
```

```
df_params = {}

for i in range(len(df_ts_KeyData)):
    df_param = extract_data(var_id =timeSeries[i], col_name = variableType[i])
    df_params['Parameter_'+str(i)]=df_param

locals().update(df_params)

# test and see what the data is like
print(Parameter_0)
```

```
                           Gas volume
dateTime
2013-09-14 00:00:00+00:00        0.00
2013-09-14 00:30:00+00:00        0.00
2013-09-14 01:00:00+00:00        0.00
2013-09-14 01:30:00+00:00        0.00
2013-09-14 02:00:00+00:00        0.00
...                               ...
2015-05-06 21:30:00+00:00        0.14
2015-05-06 22:00:00+00:00        0.05
2015-05-06 22:30:00+00:00        0.10
2015-05-06 23:00:00+00:00        0.00
2015-05-06 23:30:00+00:00        0.00

[28704 rows x 1 columns]
```

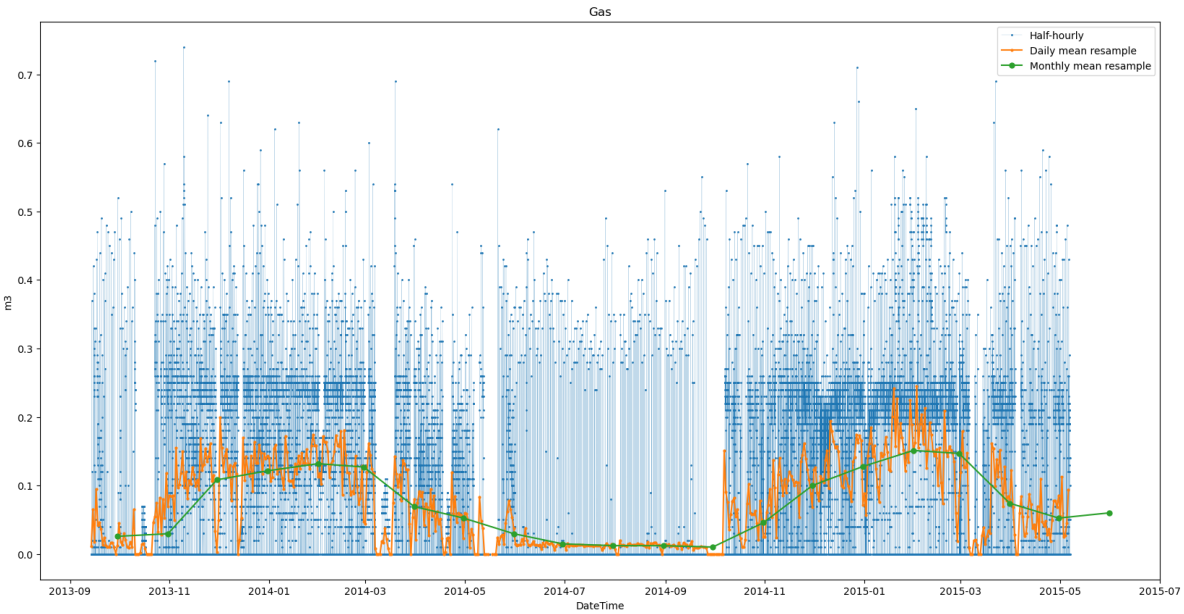
1.3.3. Initial intepretation and visualization of data

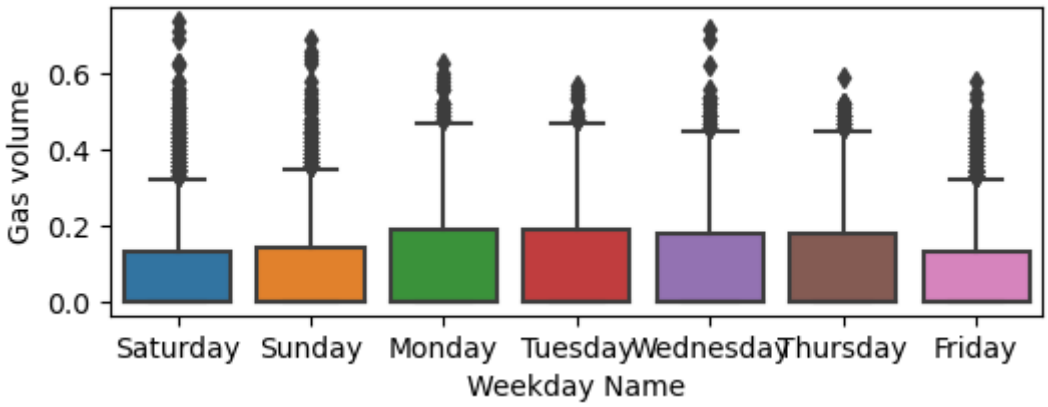
To understand the patterns of the gas volume data, we first plot the dataset and observe if there are any patterns in seasons, months or weeks. For the visualisation, we use the `matplotlib` package and the `pandas` inbuilt plotting functionalities. Alternatively, advanced libraries for plotting exist (e.g. `seaborn`), but in this tutorial we will as much as possible stick to panda's built-in funcionalities.

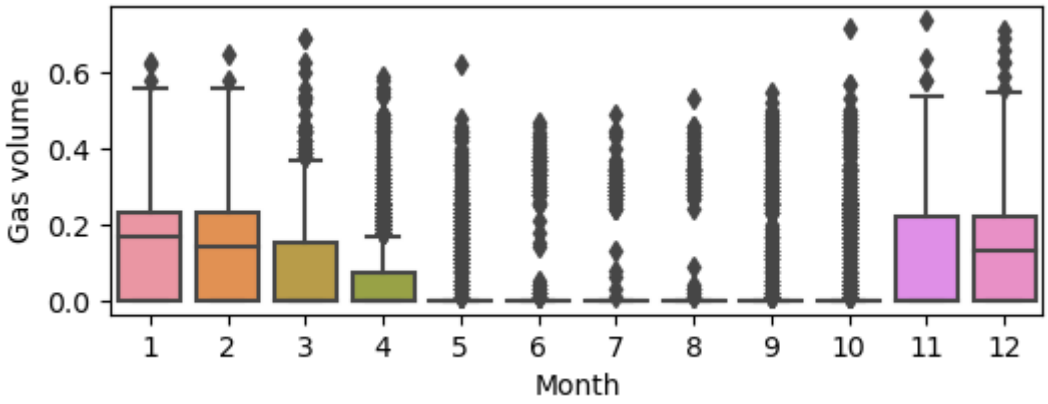
In [16]:
```
# We know that gas volume is the first row
df_gas = Parameter_0

## get daily and monthly gas consumption
data_columns = ['Gas volume']
# Resample to weekly frequency, aggregating with mean
df_gas_daily = df_gas[data_columns].resample('D').mean()
df_gas_month = df_gas[data_columns].resample('M').mean()

# plot half-hourly, daily and monthly gas consumption on one graph
start,end = '2013-09', '2015-05'
fig,ax = plt.subplots(figsize = (20,10))
ax.plot(df_gas.loc[start:end,'Gas volume'],marker='o', markersize=1, linestyle='-'
ax.plot(df_gas_daily.loc[start:end,'Gas volume'],marker='o', markersize=2, linesty
ax.plot(df_gas_month.loc[start:end,'Gas volume'],marker='o', markersize=5, linesty

ax.set_ylabel('Gas consumption (m3)')
ax.set_xlabel('DateTime')
ax.legend();

# Add columns with year, month, and weekday name
df_gas['Year'] = df_gas.index.year
df_gas['Month'] = df_gas.index.month
df_gas['Weekday Name'] = df_gas.index.day_name()

# plot weekly patterns
```

```
fig, axes = plt.subplots(figsize=(6,2 ), sharex=True)
sns.boxplot(data=df_gas, x='Weekday Name', y='Gas volume')

# plot monthly patterns
fig, axes = plt.subplots(figsize=(6, 2), sharex=True)
sns.boxplot(data=df_gas, x='Month', y='Gas volume')
ax.set_ylabel('m3')
ax.set_title('Gas')
```

Out[16]: Text(0.5, 1.0, 'Gas')







It can be observed that the monitored period span between Sep 2013 to May 2015. The gas consumption demonstrates a pattern of seasonality (peaks in winter and plunges in summer), which makes sense. The data demonstrates clear patterns in months, and less clear patterns within week days (however we can still see that gas consumption is lower on Friday and weekends). Most data points of the half-hourly gas consumption are between 0

to about 0.5 cubic meters, with some extreme cases reaching 0.7 cubic meters. The quality of data is also pretty decent, with no apparant missing data or outliers.

Which data sets are most correlated to gas volume? We then explore how to select best features to train the machine learning models.

1.3.4. Subtask A： feature selection

SubTaskA is a classification task which aims at predicting if the gas is being used. It will be using two machine leaning algorithms: decision tree and SVM (support vector machine), to achieve this goal, and the accuracy and run time of these two algorithms will be compared and discussed.

There are many monitored features that might influence the usage of gas (for example air temperature, surface temperature, relative humidity, brightness, etc.) and we will be selecting the features that are most relevant. The start & end dates and time intervals of features vary (for example some features are monitored across 2013 and 2015 and some 2014 to 2015, and some features are monitored every 30 min and some are monitored every 15 min). We already know that gas volume is monitored every 30 min, and the monitored date is 2013-09 to 2015-05. For this task, we will first select the features that also have a 30-min time interval.

In [17]:
```
## create a list gas_onOff to record the state of on/off of gas (0 and 1)
df_gas_onOff=df_gas.copy()

# replace values of gas consumption in the dataframe with 0(off) or 1(on)
for i in range(len(df_gas_onOff.index)):
    if df_gas_onOff['Gas volume'][i]>0:
        df_gas_onOff['Gas volume'][i]=1
    elif df_gas_onOff['Gas volume'][i]==0:
        df_gas_onOff['Gas volume'][i]=0
    else:
        print('error')

# check the time frames of data. At this stage we only consider and extract half-ho
half_hourly_index=[]
half_hourly_list =[]
for i in range(len(df_ts_KeyData)):
    if df_ts_KeyData['intervalLength'][i]==30:
        half_hourly_index.append(i)
        half_hourly_list.append((str(i)+'_'+df_ts_KeyData['variableType'][i]))

print(half_hourly_index)
```

```
[0, 1, 16, 17, 18, 19, 20, 21, 22, 23, 24, 37, 38, 39, 40, 41, 42, 43, 44, 45, 54,
55, 56, 57, 58, 59, 60, 61, 62, 72, 73, 74, 75, 76, 77, 95, 96, 97, 98, 99, 100, 1
01, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 117, 118, 119, 12
0, 121, 122, 123, 124, 125, 126, 127, 128]
```

We then select the features that have overlapping date with gas volume.

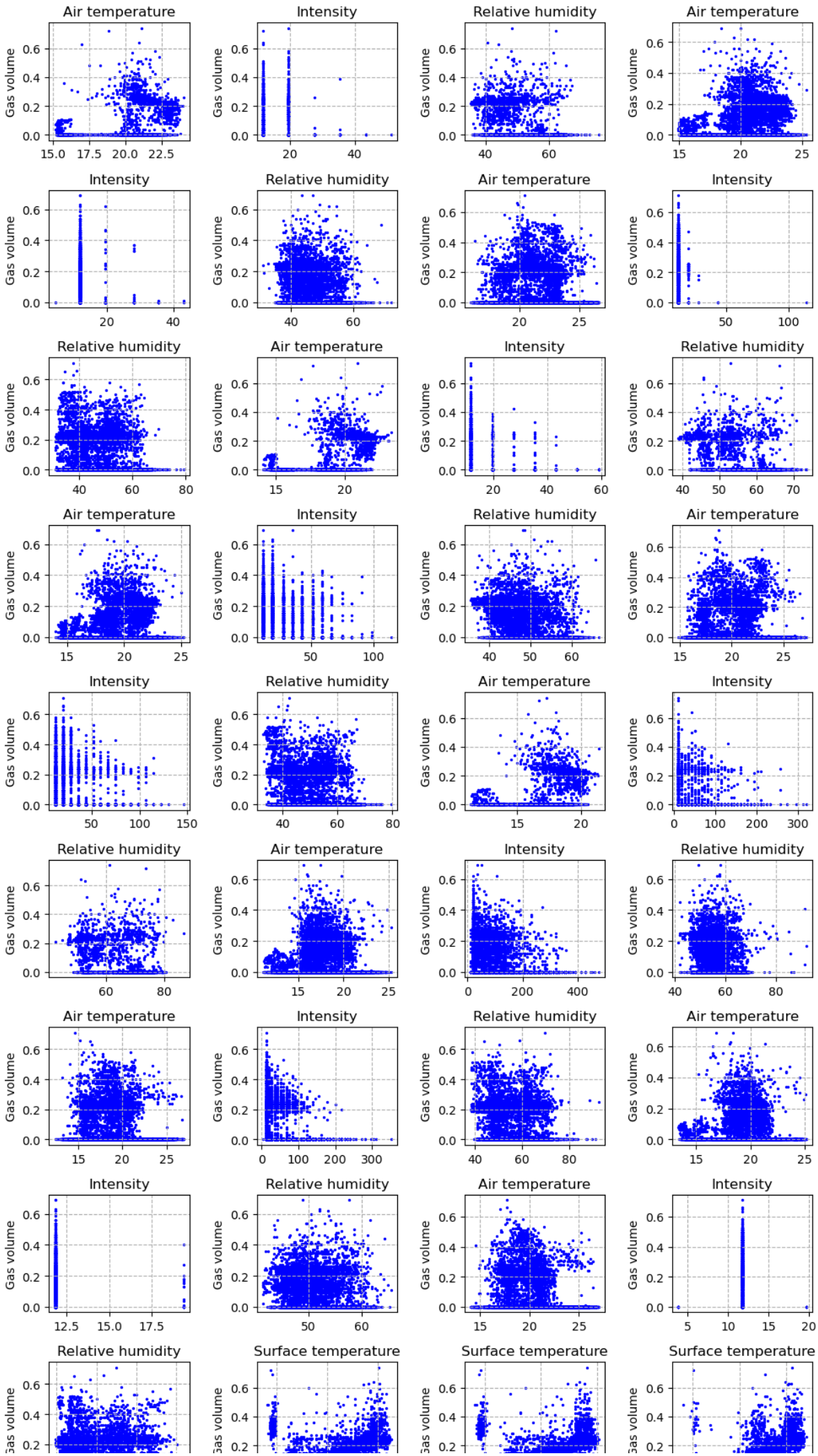
In [18]:
```
## remove empty data
empty_list_index = []
for i in range(len(df_ts_KeyData)):
    if df_params['Parameter_'+str(i)].empty:
        empty_list_index.append(i)
empty_list_index
```

```
# remove elements of the empty_list_index in the half_hourly_index
half_hourly_index= [i for i in half_hourly_index if i not in empty_list_index]

## check if the dataframes of the parameters overlap.
# This is done by merging them first,and see if the data is empty inside
no_overlap_index = []
for i in half_hourly_index:
    df_param = df_params['Parameter_'+str(i)]
    df_merge=pd.merge(df_gas,df_param,how='inner', left_index=True, right_index=Tru
    if df_merge.empty:
        no_overlap_index.append(i)
half_hourly_index= [i for i in half_hourly_index if i not in no_overlap_index]

# remove the gas volume data itself
half_hourly_index.remove(0)
print(half_hourly_index)
print(len(half_hourly_index))
```

```
[16, 17, 18, 19, 20, 21, 22, 23, 24, 37, 38, 39, 40, 41, 42, 43, 44, 45, 54, 55, 5
6, 57, 58, 59, 60, 61, 62, 72, 73, 74, 75, 76, 77, 95, 99, 106, 110, 117, 121, 12
5]
40
```

The result shows that there are the 40 features that have a 30 min monitored intervals, and that have overlapping period with gas consumption. We will then explore the correlation between these features and gas volume to select the most relevant ones. This is first done by scatter plots that help us intuitively understand the data.

In [19]:
```
# Create figure
fig, axes = plt.subplots(10,4, figsize = (10,20))
ax = axes.flatten()

for num,index in enumerate(half_hourly_index):
    df_merge = pd.merge(df_gas,df_params['Parameter_'+str(index)],how='inner', lef
    df_merge.plot.scatter(x = str(variableType[index]), y ='Gas volume', ax = ax[nu
    # Format figure:
    ax[num].set_xlabel('')
    ax[num].set_title(str(variableType[index]))
    ax[num].grid(ls = '--')

for j in range(i+1, len(ax)): ax[j].axis('off') # deactivate unused sub-axes
plt.tight_layout()
```


Air temperature
Intensity
Relative humidity
Air temperature
Intensity
Relative humidity
Air temperature
Intensity
Relative humidity
Air temperature
Intensity
Relative humidity
Air temperature
Intensity
Relative humidity
Air temperature
Intensity
Relative humidity
Air temperature
Intensity
Relative humidity
Air temperature
Intensity
Relative humidity
Air temperature
Intensity
Relative humidity
Air temperature
Intensity
Relative humidity
Air temperature
Intensity
Relative humidity
Surface temperature
Surface temperature
Surface temperature
Gas volume

It can be observed that patterns of linearality exist in the plots of 'surface temperature', but no clear patterns exist in air temperature and relative humidity. The plots of 'Intensity' are different from other ones being that they are discrete, and so they will not be used for our feature selections. We then use pd.DataFrame.corr to numerically calculate the correlations between gas volume and other data sets.

In [20]:
```
# create a list of correlation data
corre = []
for i in half_hourly_index:
    df_param = df_params['Parameter_'+str(i)]
    df_merge=pd.merge(df_gas_onOff,df_param,how='inner', left_index=True, right_in
    corr =df_merge['Gas volume'].corr(df_merge[variableType[i]])
    corre.append(corr)
#print(corre)

# sort the data from small to large
sort_index = [i for i, x in sorted(enumerate(corre), key=lambda x: x[1])]

# select 10 most correlated features
selected_index = sort_index[len(sort_index)-10:len(sort_index)]
params_selected_index = []
for i in selected_index:
    params_index = half_hourly_index[i]
    params_selected_index.append(params_index)

# merge the most relavant features into one table
df_merge_taskA = pd.merge(df_gas_onOff,df_params['Parameter_'+str(params_selected_
for i in params_selected_index[1:len(params_selected_index)]:
    df_merge_taskA = pd.merge(df_merge_taskA,df_params['Parameter_'+str(i)],how='i

# remove redundant information
df_merge_taskA = df_merge_taskA.drop(columns = ['Year','Month','Weekday Name'])
df_merge_taskA
```

Out[20]:

dateTime	Gas volume	Air temperature_x	Air temperature_y	Air temperature	Surface temperature_x	Surfa temperature_
2013-10-02 06:00:00+00:00	1.0	21.151	18.176	19.793	20.620	21.13
2013-10-02 06:30:00+00:00	0.0	21.199	18.176	19.793	20.620	21.13
2013-10-02 07:00:00+00:00	0.0	21.199	18.105	19.674	20.120	20.63
2013-10-02 07:30:00+00:00	0.0	21.151	18.010	19.698	20.120	20.63
2013-10-02 08:00:00+00:00	0.0	21.246	17.891	20.126	20.120	20.63
...	...	...	...	...	...	
2013-12-03 13:30:00+00:00	1.0	22.154	19.817	20.960	57.004	41.1(
2013-12-03 14:00:00+00:00	1.0	22.298	20.388	21.103	57.004	41.1(
2013-12-03 14:30:00+00:00	1.0	19.270	18.913	18.247	57.004	41.6(
2013-12-03 15:00:00+00:00	1.0	18.652	18.652	17.891	57.004	41.6(
2013-12-03 15:30:00+00:00	1.0	21.270	21.056	20.936	56.508	41.6(

3250 rows × 11 columns

The correlation between gas volume and the 40 features are 0.336,0.260,-0.389, etc. By sorting the data, 10 features that are most relevant with gas volume are: Parameter_16, Parameter_54, Parameter_37, etc. These features are then put in one table, and we can see that the most correlated features are air temperature and surface temperature. These data may also be correlated, in which case we will later explore if 10 features are necessary, or that we can achieve similar performances with fewer features.

We can also see that the merged dataframe is 2013-10-02 to 2013-12-03, and this is the period we will be analyzing. This is beneficial, because Oct to Dec is also the period where gas volume rapidly grows up and shows clear patterns, and by machine learning we can better understand this trend.

1.3.5. Subtask A

We first explore a very popular classification algorith, support vector machine. Support Vector Machines (SVM) is a supervised machine learning algorithm which can be used for classification or regression problems. It uses a technique called the kernel trick to transform our data and then based on these transformations it finds an optimal boundary between the

possible outputs. The goal of SVM is to identify an optimal separating hyperplane which maximizes the margin between different classes of the training data.The advantages of support vector machines are: (https://scikit-learn.org/stable/modules/svm.html#:~:text=Support%20vector%20machines%20(SVMs)%20are,

- Effective in high dimensional spaces.
- Still effective in cases where number of dimensions is greater than the number of samples.
- Uses a subset of training points in the decision function (called support vectors), so it is also memory efficient.
- Versatile: different Kernel functions can be specified for the decision function. Common kernels are provided, but it is also possible to specify custom kernels.

The disadvantages of support vector machines include:

- If the number of features is much greater than the number of samples, avoid over-fitting in choosing Kernel functions and regularization term is crucial.
- SVMs do not directly provide probability estimates, these are calculated using an expensive five-fold cross-validation.

In [77]:
```
# import related to ML models
from sklearn.model_selection import train_test_split
from sklearn import metrics
from sklearn.metrics import f1_score
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
from sklearn.metrics import plot_confusion_matrix
from sklearn.metrics import classification_report
from sklearn.utils import resample
# import related to hyperparameter tuning
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import cross_val_score
import timeit
# import related to SVM
from sklearn import svm
from sklearn.preprocessing import scale
from sklearn.svm import SVC
```

For the training of any Machine Learning algorithm, we separate the data into a ***training*** set and a ***validation*** set. The aim of this separation is to keep one part of the data "clean" in order to validate the performance of our model. This allows to check, for example, if the model is overfitting the data. Typically, the size of the validation set is between 20-33% of the data.

To split the data (randomly) into training and validation set, we can use a built-in function `train_test_split` of our Machine-Learning package `scikit-learn`, which is imported as `sklearn`.For this coursework, all data will be split into 70% training and 30% testing.

In [78]:
```
# application of support vector machine
# separate target variables
X = df_merge_taskA.values[:,4:9]
Y = df_merge_taskA.values[:,0]
Y=Y.astype('int')
```

```
# splitting dataset to training dataset and test dataset
X_train, X_test,y_train, y_test = train_test_split(X,Y,test_size = 0.3,random_stat
```

In [79]:
```
# quickly train the SVM model
clf_svm = SVC().fit(X_train, y_train)
clf_svm.score(X_test, y_test)
```

Out[79]:
```
0.9138461538461539
```

Hyper-parameters are parameters that are not directly learnt within estimators. In scikit-learn, they are passed as arguments to the constructor of the estimator classes. Grid search is commonly used as an approach to hyper-parameter tuning that will methodically build and evaluate a model for each combination of algorithm parameters specified in a grid.

In [80]:
```
# Record how long the hyperparameter tuning takes
start = timeit.default_timer()

# Set up grid search parameters
param_grid = [
    {'C': [0.1,0.5,1,10,100],
    'gamma':['scale',1,0.1,0.01,0.001,0.0001],
    'kernel': ['rbf']},
]

# Optimize hyperparameters with cross validation and GridSearchCV()
optimal_params=GridSearchCV(SVC(),param_grid,cv=5,scoring='accuracy')
optimal_params.fit(X_train,y_train)
print(optimal_params.best_params_)

# Record runtime
stop = timeit.default_timer()
svm_cv_time = stop-start
print('Subtask A Decision tree run time: ', svm_cv_time)
```

```
{'C': 1, 'gamma': 0.1, 'kernel': 'rbf'}
Subtask A Decision tree run time:  11.823475000001054
```

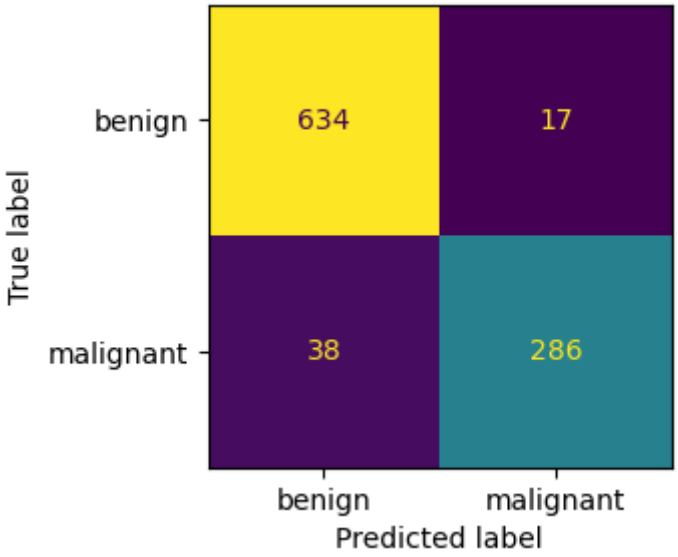
In [81]:
```
# quickly train the SVM model
clf_svm = SVC(random_state = 42,C=1,gamma = 0.1,kernel = 'rbf').fit(X_train, y_tra
clf_svm.score(X_test, y_test)
```

Out[81]:
```
0.9435897435897436
```

We can see this result is a lot better than before (0.91). Now we execute the model and see how it performs by calculating accuracy, f1 score and plotting confusion matrix.

In [82]:
```
# apply svm model and check accuracy and f1 score
predict_svm = clf_svm.predict(X_test)
svm_accuracy = metrics.accuracy_score(y_test,predict_svm)
svm_f1 = metrics.f1_score(y_test,predict_svm)
print("Accuracy:",svm_accuracy,"F1_score:",svm_f1)

# plot confusion matrix
svm_cm = confusion_matrix(y_test,predict_svm)

fig, ax = plt.subplots(figsize=(3, 3))
svm_conf = plot_confusion_matrix(clf_svm,X_test, y_test, values_format = 'd', ax=a
```

```
Accuracy: 0.9435897435897436 F1_score: 0.912280701754386
```



Results:TP = 637, TN = 254, FP = 14, FN = 70. This is a pretty good result, with accuracy achieveing 93% and F1 score being 0.88. SVM works really well with a clear margin of separation, but it fails to perform well when the data set has more noise, which is potentially why the f1 score hasn't achieved 0.9.

We then explore decision tree algorithms. The strengths of decision tree methods are (https://scikit-learn.org/stable/modules/tree.html):

- Decision trees are easy to understand. Its results are in a set of rules
- It performs classification without requiring much computation.
- It is capable to handle both continuous and categorical variables.
- It provides a clear indication of which fields are most important for prediction or classification.

The weaknesses of decision tree methods:

- Decision trees are less appropriate for estimation tasks where the goal is to predict the value of a continuous attribute.
- They are prone to errors in classification problems with many class and relatively small number of training examples.
- There is a high probability of overfitting in Decision Tree.

In [83]:
```
# imports related to decision trees
from sklearn.tree import DecisionTreeClassifier
from sklearn import tree
```

In [84]:
```
# quickly train the SVM model and see how it performs
clf_dt = DecisionTreeClassifier().fit(X_train, y_train)
clf_dt.score(X_test, y_test)
```

Out[84]:
```
0.9271794871794872
```

In [85]:
```
# Optimize hyperparameters with cross validation and GridSearchCV()
start = timeit.default_timer()
param_grid = [
    { 'max_features': ['auto', 'sqrt', 'log2'],
    'max_depth' : [4,5,6,7,8,9,10,11,12],
    'min_samples_split' : [1,2,3,4,5,6,7,8,9,10],
```

```
    'min_samples_leaf' : [1,2,3,4,5,6,7,8,9,10],
    'criterion' :['gini', 'entropy']},
]
dt = DecisionTreeClassifier()
grid_search = GridSearchCV(estimator = dt,param_grid = param_grid,cv = 5,scoring='
grid_search.fit(X_train,y_train)
print(grid_search.best_params_)

# Print out run time
stop = timeit.default_timer()
dt_cv_time = stop-start
print('Subtask A Decision tree run time: ', dt_cv_time)
```

```
{'criterion': 'entropy', 'max_depth': 12, 'max_features': 'auto', 'min_samples_lea
f': 1, 'min_samples_split': 9}
Subtask A Decision tree run time:  40.08520630000021
```

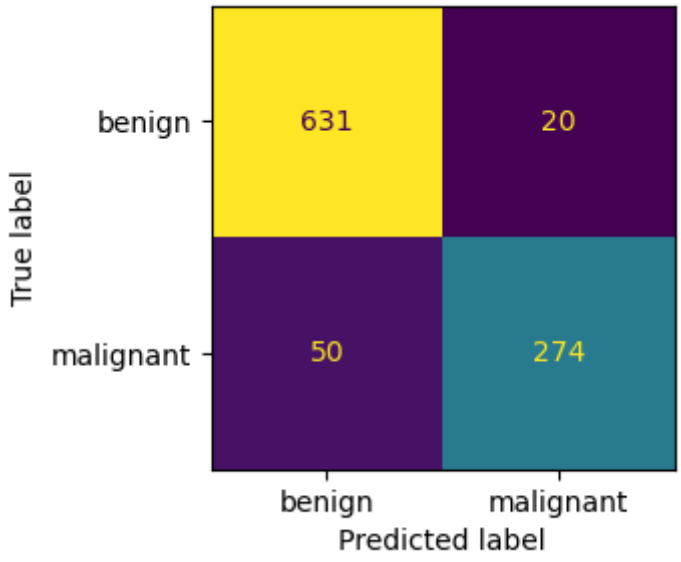
In [86]:
```
# function to perform training with the hyperparameters
clf_dt = DecisionTreeClassifier(criterion = 'entropy',max_depth =9, max_features='
clf_dt = clf_dt.fit(X_train,y_train)
clf_dt.score(X_test, y_test)
```

Out[86]:
```
0.9282051282051282
```

We can see that by hyperparameter tuning we have improved the socre of the algorithm. Now we execute the model and see how it performs by calculating accuracy, f1 score and plotting confusion matrix.

In [87]:
```
predict_dt = clf_dt.predict(X_test)

# calculate accuracy
dt_accuracy = metrics.accuracy_score(y_test,predict_dt)
dt_f1 = metrics.f1_score(y_test,predict_dt)
print("Accuracy:",dt_accuracy,"F1 score:",dt_f1)

# plot confusion matrix
fig, ax = plt.subplots(figsize=(3, 3))
dt_conf=plot_confusion_matrix(clf_dt,X_test, y_test, values_format = 'd', ax=ax, c
```

```
Accuracy: 0.9282051282051282 F1 score: 0.8867313915857604
```



While decision trees are common supervised learning algorithms, they can be prone to problems, such as bias and overfitting. However, when multiple decision trees form an

ensemble in the random forest algorithm, they predict more accurate results, particularly when the individual trees are uncorrelated with each other.

Random forest algorithms have three main hyperparameters, which need to be set before training. These include node size, the number of trees, and the number of features sampled. From there, the random forest classifier can be used to solve for regression or classification problems.

In [88]:
```
from sklearn.ensemble import RandomForestClassifier
# quickly train the SVM model and see how it performs
clf_rf = DecisionTreeClassifier().fit(X_train, y_train)
clf_rf.score(X_test, y_test)
```

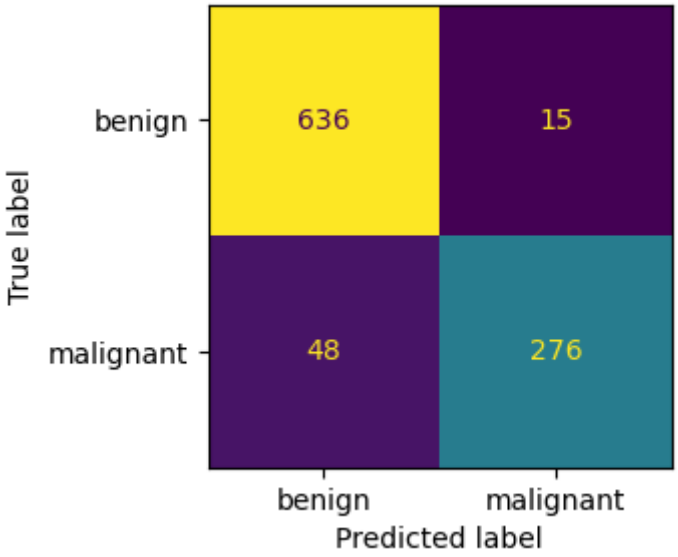
Out[88]:
```
0.9302564102564103
```

In [89]:
```
# Optimize hyperparameters with cross validation and GridSearchCV()
start = timeit.default_timer()
param_grid = {
    'n_estimators': [200, 500],
    'max_features': ['auto', 'sqrt', 'log2'],
    'max_depth' : [4,5,6,7,8],
    'criterion' :['gini', 'entropy']
}

rf = RandomForestClassifier()
grid_search = GridSearchCV(estimator = rf,param_grid = param_grid,cv = 5)
grid_search.fit(X_train,y_train)
print(grid_search.best_params_)

# Print out run time
stop = timeit.default_timer()
rf_cv_time = stop-start
print('Subtask A Random forest run time: ', rf_cv_time)
```

```
{'criterion': 'entropy', 'max_depth': 8, 'max_features': 'log2', 'n_estimators': 2
00}
Subtask A Decision tree run time:  135.95030009999937
```

In [90]:
```
#clt_rf = RandomForestRegressor(n_estimators = 1800, min_samples_split = 5, min_sa
clt_rf = RandomForestClassifier(criterion = 'entropy',n_estimators = 200,max_depth
clt_rf.fit(X_train,y_train)
#rf.score(X_test,y_test)
predict_rf = clt_rf.predict(X_test)

# calculate accuracy
rf_accuracy = metrics.accuracy_score(y_test,predict_dt)
rf_f1 = metrics.f1_score(y_test,predict_dt)
print("Accuracy:",rf_accuracy,"F1 score:",rf_f1)

# plot confusion matrix
fig, ax = plt.subplots(figsize=(3, 3))
rf_conf=plot_confusion_matrix(clt_rf,X_test, y_test, values_format = 'd', ax=ax, c
```

```
Accuracy: 0.9282051282051282 F1 score: 0.8867313915857604
```



1.3.6. Subtask B

In task B we investigate the appropriate ML algorithms to predict gas consumptions. We first explore multiple linear regression.

In [113…

```
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import r2_score

# merge the most relavant features into one table
df_merge_taskB = pd.merge(df_gas,df_params['Parameter_'+str(params_selected_index[(
for i in params_selected_index[1:len(params_selected_index)]:
    df_merge_taskB = pd.merge(df_merge_taskB,df_params['Parameter_'+str(i)],how='i

# remove redundant information
df_merge_taskB = df_merge_taskB.drop(columns = ['Year','Month','Weekday Name'])
# have a look at what's inside
df_merge_taskB.head()
```

Out[113]:

dateTime	Gas volume	Air temperature_x	Air temperature_y	Air temperature	Surface temperature_x	Surfa temperature
2013-10-02 06:00:00+00:00	0.49	21.151	18.176	19.793	20.62	21.1
2013-10-02 06:30:00+00:00	0.00	21.199	18.176	19.793	20.62	21.1
2013-10-02 07:00:00+00:00	0.00	21.199	18.105	19.674	20.12	20.6
2013-10-02 07:30:00+00:00	0.00	21.151	18.010	19.698	20.12	20.6
2013-10-02 08:00:00+00:00	0.00	21.246	17.891	20.126	20.12	20.6

In [116…
```
# application of multiple linear regressions
# separate target variables
X = df_merge_taskB.values[:,4:9]
Y = df_merge_taskB.values[:,0]
Y=Y.astype('double')

# splitting dataset to training dataset and test dataset
X_train, X_test,y_train, y_test = train_test_split(X,Y,test_size = 0.3,random_stat
```

In [117…
```
from sklearn.ensemble import RandomForestClassifier
# quickly train the SVM model and see how it performs
lr = LinearRegression().fit(X_train,y_train)
lr.score(X_test, y_test)
#C = lr.intercept_
#m = lr.coef_
#print(C,m)
```

Out[117]:
```
0.5991279385685371
```

In [124…
```
# Optimize hyperparameters with cross validation and GridSearchCV()
start = timeit.default_timer()
param_grid = {
    'n_jobs': [-1,1],
    'fit_intercept':[True, False],
}

lr = LinearRegression()
grid_search = GridSearchCV(estimator = lr,param_grid = param_grid,cv = 5)
grid_search.fit(X_train,y_train)
print(grid_search.best_params_)

# Print out run time
stop = timeit.default_timer()
rf_cv_time = stop-start
print('Subtask A Decision tree run time: ', rf_cv_time)
```

```
{'fit_intercept': True, 'n_jobs': -1}
Subtask A Decision tree run time:  0.018353200001001824
```

In [126…
```
lr = LinearRegression(fit_intercept = True, n_jobs = -1).fit(X_train,y_train)
predict_lr=lr.predict(X_train)

# Calculate scpres: route mean square, MAE and r2
mlr_RMSE= np.sqrt(mean_squared_error(y_train,predict_lr ))
mlr_MAE = mean_absolute_error(y_train,predict_lr)
mlr_R2=r2_score(y_train,predict_lr)

print(mlr_RMSE,mlr_MAE,mlr_R2)
```

```
0.073989526084052 0.037991980551304226 0.5695802855269485
```

R2 is not very close to 1, which means there's room for improvement. However there's limited room for hyperparamter tuning for multiple linear regressions.

We then use decision tree regressor.Decision Tree is one of the most commonly used, practical approaches for supervised learning.It is a tree-structured classifier with three types of nodes. The Root Node is the initial node which represents the entire sample and may get split further into further nodes. The Interior Nodes represent the features of a data set and the branches represent the decision rules. Finally, the Leaf Nodes represent the outcome. This algorithm is very useful for solving decision-related problems.With a particular data

point, it is run completely through the entirely tree by answering True/False questions till it reaches the leaf node. The final prediction is the average of the value of the dependent variable in that particular leaf node. Through multiple iterations, the Tree is able to predict a proper value for the data point.

In [129…
```
clf_dtr = DecisionTreeRegressor().fit(X_train,y_train)
clf_dtr.score(X_test, y_test)
```

Out[129]:
```
0.2519463873640718
```

In [135…
```
# Optimize hyperparameters with cross validation and GridSearchCV()
start = timeit.default_timer()
param_grid={"splitter":["best","random"],
            "max_depth" : [1,3,5,7,9,11,12],
           "min_samples_leaf":[1,2,3,4,5,6,7,8,9,10],
           "min_weight_fraction_leaf":[0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9],
           "max_features":["auto","log2","sqrt",None],
           "max_leaf_nodes":[None,10,20,30,40,50,60,70,80,90] }

clf_dtr = DecisionTreeRegressor()
grid_search = GridSearchCV(estimator = clf_dtr,param_grid = param_grid,cv = 5,scor
grid_search.fit(X_train,y_train)
print(grid_search.best_params_)

# Print out run time
stop = timeit.default_timer()
dt_dtr_time = stop-start
print('Subtask A Decision tree regressor run time: ', dt_dtr_time)
```

```
{'max_depth': 1, 'max_features': 'auto', 'max_leaf_nodes': None, 'min_samples_lea
f': 1, 'min_weight_fraction_leaf': 0.1, 'splitter': 'best'}
Subtask A Decision tree regressor run time:  244.8324008999989
```

In [140…
```
# application of decision tree
# function to perform training with entropy
clf_dtr = DecisionTreeRegressor( max_depth = 1, min_samples_split = 10, max_featur
                                min_weight_fraction_leaf = 0.1,splitter = 'best',r
clf_dtr.fit(X_train,y_train)
clf_dtr.score(X_test, y_test)
```

Out[140]:
```
0.5606189099994543
```

In [173…
```
predict_dt_2 = clf_dtr.predict(X_test)

dtr_RMSE= np.sqrt(mean_squared_error(y_test,predict_dt_2 ))
dtr_MAE = mean_absolute_error(y_test,predict_dt_2)
dtr_R2=r2_score(y_test,predict_dt_2)
print(dtr_RMSE,dtr_MAE,dtr_R2)
```

```
0.07473029792580287 0.03967398304128204 0.5606189099994543
```

Cross validation allows us to compare different machine learning methods and get a sense of how well they work in practice. Ten-fold cross validation is a pretty common practice.

In [144…
```
cross_val_score(clf_dtr, X_train, y_train, cv=10)
```

Out[144]:
```
array([0.4827296 , 0.60721084, 0.57024679, 0.73555946, 0.4916232 ,
       0.50003605, 0.46571414, 0.476289  , 0.54650703, 0.68536425])
```

In [165…
```
from sklearn import linear_model
from sklearn.linear_model import Lasso
```

```
clf_lasso = Lasso().fit(X_train,y_train)
clf_lasso.score(X_test, y_test)
```

Out[165]:
```
0.2372432182330496
```

In [169…
```
# Optimize hyperparameters with cross validation and GridSearchCV()
start = timeit.default_timer()

alphas = np.array([5, 0.5, 0.05, 0.005, 0.0005, 1, 0.1, 0.01,0.001,0.0001])

clf_lasso = Lasso()
grid_search = GridSearchCV(estimator=clf_lasso, param_grid=dict(alpha=alphas),cv=5
grid_search.fit(X_train,y_train)
print(grid_search.best_params_)

# Print out run time
stop = timeit.default_timer()
dt_lassp_time = stop-start
print('Subtask A Lasso regressor run time: ', dt_lassp_time)
```

```
{'alpha': 0.01}
Subtask A Lasso regressor run time:  0.09879860000000917
```

In [170…
```
clf_lasso = Lasso( alpha = 0.01,random_state = 42)
clf_lasso.fit(X_train,y_train)
clf_lasso.score(X_test, y_test)
```

Out[170]:
```
0.5990324074332636
```

It can be seen that the score of the model has been significantly improved (from 0.23 to 0.59).

In [171…
```
predict_lasso = clf_lasso.predict(X_test)

dtr_RMSE= np.sqrt(mean_squared_error(y_test,predict_lasso ))
dtr_MAE = mean_absolute_error(y_test,predict_lasso)
dtr_R2=r2_score(y_test,predict_lasso)
print(dtr_RMSE,dtr_MAE,dtr_R2)
```

```
0.07138889655697903 0.03644612084339877 0.5990324074332636
```

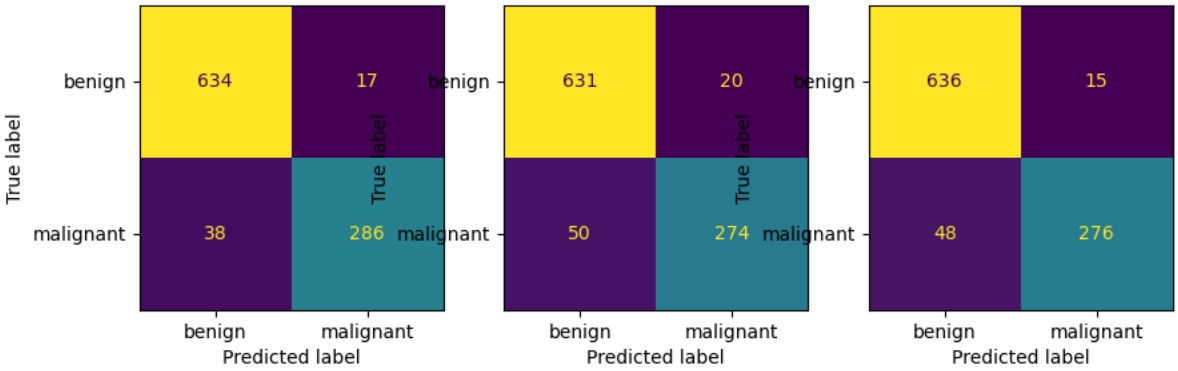
1.3.7 Results and Summary

1.3.7.1 Subtask A: most appropriate model is SVM

Subtask A aims at predicting the gas consumption and evaluating the performances of different machine learning algorithms. The confusion matrices of SVM, decision tree and random forest algorithms are plotted below. Choosing a threshold that allows misclassifications is an example of the Bias/Variance tradeoff that plagues all of machine learning.

In [172…
```
fig, axes = plt.subplots(1,3,figsize = (10,10))
ax = axes.flatten()

svm_conf.plot(ax=ax[0],colorbar = False)
dt_conf.plot(ax=ax[1],colorbar = False)
rf_conf.plot(ax=ax[2],colorbar = False)
```

Out[172]: <sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x2120a12c8e0>



The accuracy, f1 score and training times of these algorithms are also compared.

Algorithm	Accuracy	F_1 socre	Training time
SVM	0.943	0.912	11.823
Decision tree	0.928	0.886	40.085
Random forest	0.928	0.886	135.950

The evaluation of these models should take into account of all parameters. We often use accuracy when the classes are balanced and there is no major downside to predicting false negatives.We often use F1 score when the classes are imbalanced and there is a serious downside to predicting false negatives.

SVM turns out to be the most appropriate algorithm, with accuracy and f1 score being the highest among the three, and training time being the lowest. However, SVM might have a tendancy of overfitting. Overfitting is easy to diagnose with the accuracy visualizations we have available. If "Accuracy" (measured against the training set) is very good and "Validation Accuracy" (measured against a validation set) is not as good, then the model is overfitting. In future analyses the overfitting problem can be investigated with validation accuracy calculations.

1.3.7.2 Subtask B: most appropriate model is Lasso regression

Subtask B aims at predicting the gas consumption and evaluating the performances of different machine learning algorithms. The performance matrix of these algorithms are listed below.

Algorithm	RMSE	MAE	R^2	Training time
Multiple linear regression	0.073	0.037	0.569	0.018
Decision tree regressor	0.074	0.039	0.560	244.832
Lasso regression	0.071	0.036	0.599	0.098

The evaluation of models should take into account of all parameters. R2 can only be between 0 and 1, where 0 indicates that the outcome cannot be predicted by any of the independent variables and 1 indicates that the outcome can be predicted without error from the independent variables.

The most appropriate algorithm turns out to be lasso regression, with R2 being the highest and training time relatively short.

2. Unsupervised Learning

2.1. Introduction and Aims

Unsupervised learning is when we only have input data (X) without a corresponding target variable(y) to predict. The aim is to model the underlying structure of the dtaa in ofrder to learn from data and identify groups of data with similar characterstics/ behaviours.

In this task we want to create a segmentation with thermal conditions. To do this, we will be extracting the 'air temperature' and 'relative humidity' data from one space of the building, and use K means clustering algorithm to categorize the thermal conditions of this space. The sensors chosen for this task are Sensor 37,38,39 & 40, which monitor the air temperature and relative humidity of Space 8 (an unconditioned Conservatory space) over different time frames. We want to combine these data, categorize, and see if there're any patterns. The hypothesis is that these conditions correspond to different months.

In [152…
```
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
```

In [153…
```
# sensor 37,38,39,40 are what we want
sensor = [37,38,39,40]
sensor_index = []

# iterate through keydata list and see what indexes correspond to these sensors
for i in sensor:
    sensor_index_list = df_ts_KeyData[df_ts_KeyData['SensorFK'] == i].index.tolist
    for j in range(len(sensor_index_list)):
        sensor_index.append(sensor_index_list[j])
print(sensor_index)

#when looking into the data, index 38,41 and 44 are 'Intensity', which is not what
remove_list = [38,41,44]
for i in remove_list:
    sensor_index.remove(i)

# We also look into the list and see which indexes are 'Air temperature' and which
sensorindex_temp = [37,40,43,46]
sensorindex_rh = [39,42,45,47]
```

```
[37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47]
```

In [154…
```
# The monitored data are of different timestamps, and some of them don't overlap.
# Here we don't care so much about dateTime, but the contents of data.
# We do this by resetting index. We will also rename the columns to avoid repetitiv
reindex_37 = df_params['Parameter_'+str(37)].reset_index().drop(columns = ['dateTi
reindex_40 = df_params['Parameter_'+str(40)].reset_index().drop(columns = ['dateTi

# merge temperatures
df_merge_clustering = pd.merge(reindex_37,reindex_40,how='inner', left_index=True,
for index,i in enumerate(sensorindex_temp[2:4]):
    reindex = df_params['Parameter_'+str(i)].reset_index().drop(columns = ['dateTi
    df_merge_clustering = pd.merge(df_merge_clustering,reindex,how='inner', left_i
```

```
# merge relative humidity
for index,i in enumerate(sensorindex_rh):
    reindex = df_params['Parameter_'+str(i)].reset_index().drop(columns = ['dateTi
    df_merge_clustering = pd.merge(df_merge_clustering,reindex,how='inner', left_i

df_merge_clustering.head()
```

Out[154]:

	temp_1	temp_2	temp_3	temp_4	rh_1	rh_2	rh_3	rh_4
0	19.865	21.818889	22.657	18.652	59.728	42.281	52.579	42.307
1	19.817	21.867222	22.753	18.628	59.781	42.933	53.188	42.176
2	19.793	21.891111	23.112	18.580	59.747	42.806	52.956	42.043
3	19.793	21.795000	23.352	18.557	60.899	42.860	52.676	42.040
4	19.674	21.747778	23.400	18.580	60.910	42.693	52.305	42.267

There are a number of ways to identify the optimal number of clusters, and the method we use here is known as the Elbow method. The Elbow plot allows us to plot the inertia, which is a measure of how well the data was clustered by the K-Means algorithm, against the number of clusters. From this plot, we are looking for a point where the inertia begins to slow down.

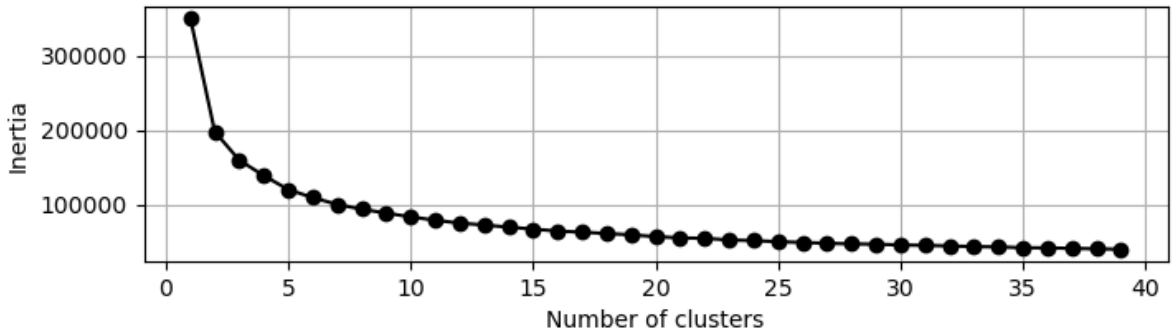
In [155…

```
# We define a function to get the inertia of different number of categories
def optimise_k_means(data,max_k):
    means = []
    inertias = []

    # iterate through data and append inertia
    for k in range(1,max_k):
        kmeans=KMeans(n_clusters = k)
        kmeans.fit(data)
        means.append(k)
        inertias.append(kmeans.inertia_)

    # Generate the elbow plot
    fig = plt.subplots(figsize = (8,2))
    plt.plot(means,inertias,'o-',color = 'black')
    plt.xlabel('Number of clusters')
    plt.ylabel('Inertia')
    plt.grid(True)
    plt.show()
```

In [156…

```
# Apply the function to our datasets
optimise_k_means(df_merge_clustering,40)
```

```
C:\Users\16337\anaconda3\lib\site-packages\sklearn\cluster\_kmeans.py:1036: UserWa
rning: KMeans is known to have a memory leak on Windows with MKL, when there are l
ess chunks than available threads. You can avoid it by setting the environment var
iable OMP_NUM_THREADS=12.
  warnings.warn(
```



The turning point of this plot is not very obvious, and it seems the line gradually flattens after k=5. Here we take an initial value of 5, as the greater the value of k is, the longer it takes to run. We will try out different values of k later, for example k = 15, k = 30.

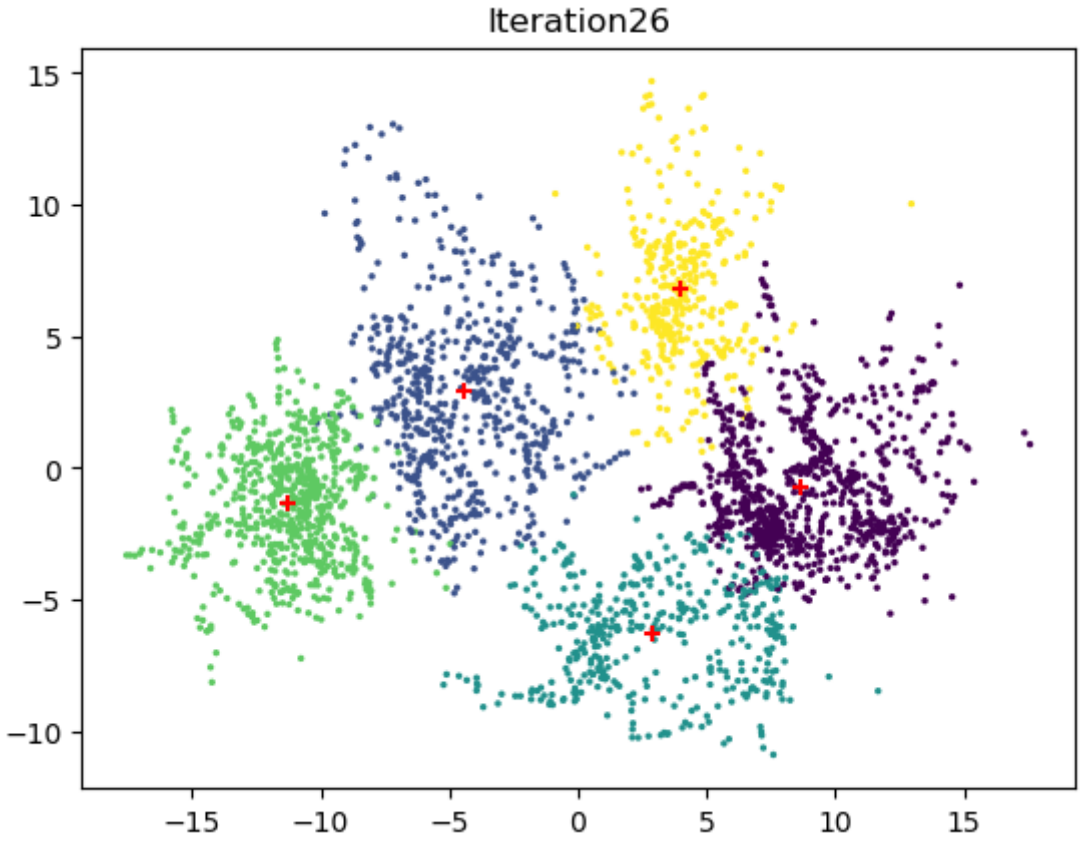
In [157…

```
from sklearn.decomposition import PCA
from IPython.display import clear_output

## Generate random centroids to start with
def random_centroids(data,k):
    centroids = []
    for i in range(k):
        centroid = data.apply(lambda x:float(x.sample()))
        centroids.append(centroid)
    return pd.concat(centroids,axis = 1)

## Generate labels for data
def get_labels(data,centroids):
    distances = centroids.apply(lambda x: np.sqrt(((data-x)**2).sum(axis = 1)))
    return distances.idxmin(axis = 1)

## New centroids are generated from the distance between data sets and centroids
def new_centroids(data,labels,k):
    return data.groupby(labels).apply(lambda x: np.exp(np.log(x).mean())).T

## The model is multi-dimensional, we use PCA to change data into two-dimentional
def plot_clusters(data,labels,centroids,iteration):
    pca = PCA(n_components = 2)
    data_2d = pca.fit_transform(data)
    centroids_2d = pca.transform(centroids.T)
    clear_output (wait = True)
    plt.title(f'Iteration{iteration}')
    plt.scatter(x=data_2d[:,0],
            y=data_2d[:,1],
            c=labels,
            s=2
           )
    plt.scatter(x=centroids_2d[:,0],y=centroids_2d[:,1],marker = '+',color = 'red'
    plt.show()
```

In [158…

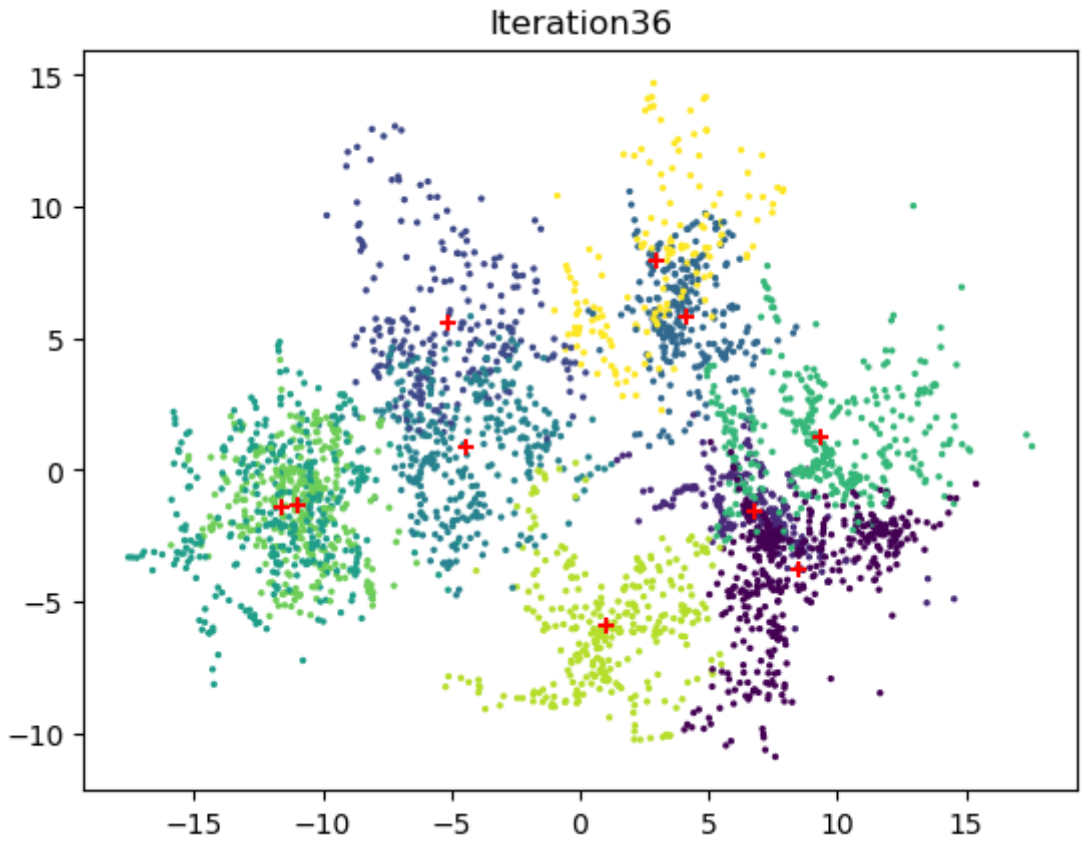
```
## We start with maximum iteration of 100 and k = 5
max_iterations = 100
k = 5
centroids = random_centroids(df_merge_clustering,k)
old_centroids = pd.DataFrame()
iteration = 1

## Iterate through clustering until centroids stop updating
while iteration<max_iterations and not centroids.equals(old_centroids):
    old_centroids = centroids
    labels = get_labels(df_merge_clustering,centroids)
```

```
    centroids = new_centroids(df_merge_clustering,labels,k)
    plot_clusters(df_merge_clustering,labels,centroids,iteration)
    iteration +=1

#Get the data for centroids
centroids_k_5 = centroids
```



In [119…

```
## see how it performs with k=10
k = 10
centroids = random_centroids(df_merge_clustering,k)
old_centroids = pd.DataFrame()
iteration = 1
while iteration<max_iterations and not centroids.equals(old_centroids):
    old_centroids = centroids
    labels = get_labels(df_merge_clustering,centroids)
    centroids = new_centroids(df_merge_clustering,labels,k)
    plot_clusters(df_merge_clustering,labels,centroids,iteration)
    iteration +=1
```



We can see that the categorization of plots is more chaotic when k = 10 (plot above), where several data points of the same place are clustered in different groups. The performance of k-means clustering when k = 5 is pretty good, and we will go with this model. Then we have a look at the data of the centroids and try to explore what it means.

In [115…
```
centroids_k_5
```

Out[115]:

	0	1	2	3	4
temp_1	17.544085	19.836918	19.186609	16.940903	19.702614
temp_2	19.948565	19.286712	18.872158	18.551524	20.073765
temp_3	23.257373	23.450225	22.841177	22.583479	24.643073
temp_4	13.120457	19.870768	19.511089	13.761253	12.931950
rh_1	64.944265	59.112016	47.250101	62.431318	55.527401
rh_2	48.266689	46.140212	43.932394	50.394917	47.966586
rh_3	50.948393	52.341220	53.836293	57.163580	56.168565
rh_4	46.741723	40.509361	43.829305	45.993362	48.858886

It's difficult to intuitively see what the data mean in a multi-dimensional unsupervised learning model. The temperature and relative humidity sensors are of different timestamps, and by re-indexing and merging them we have lost the timeseries information. Here we take the average of temperature and rh data of each cluster to see if there's any pattern.

In [161…
```
# average of temperature & relative humidity
temps_avrg = []
rhs_avrg= []
for i in range(5):
```

```
    temp_avrg = centroids_k_5[i][0:4].mean()
    temps_avrg.append(temp_avrg)
    rh_avrg = centroids_k_5[i][5:8].mean()
    rhs_avrg.append(rh_avrg)

# create a new dataframe and sort the data according to average temperature
d = {'average temp':temps_avrg,'average rh':rhs_avrg}
df = pd.DataFrame(d)
df = df.sort_values(by=['average temp'])
df.head()
```

Out[161]:

	average temp	average rh
0	18.017934	49.945335
2	19.227926	51.506898
3	19.942438	47.309291
1	20.539373	46.806839
4	20.608480	46.363754

It can be observed that in general, the increase of temperature brings a decrease of relative humidity in this space. This is in accordance with our everyday experience.

3. Reinforcement Learning

3.1. Introduction and Aims

Reinforcement learning is a sub-field of machine learning which uses a reward (or cost) to learn a good policy for solving a problem. A policy is a mapping from states of the environment to the actions available to the agent. RL differs from supervised learning in that the system does not know what the correct action for any given state is, only that some actions are better than others. A reinforcement learning agent is a system that can figure out which action is the best for each situation through a process of trial and error. By using the information gained from each trial, an RL agent improves its policy iteratively.

1. test random environment with OpenAI Gym
2. Create a deep learning model with Keras
3. Build Agent with Keras-RL
4. Reloading Agent from Memory

The Q-learning algorithm is guaranteed to converge if each state-action pair is visited a sufficiently large number of times. In this situation, however, we are dealing with a continuous state space, which means that there are infinitely many state-action pairs, making it impossible to satisfy this condition.

A solution is to just discretize the state space. One simple way in which this can be done is to round the first element of the state vector to the nearest 0.1 and the second element to the nearest 0.01, and then (for convenience) multiply the first element by 10 and the second by 100.

This reduces the number of state-action pairs down to 855, which now makes it possible to satisfy the condition required for Q-learning to converge.

In [1]:
```
pip install gym
```

```
Requirement already satisfied: gym in c:\users\16337\anaconda3\lib\site-packages
(0.26.1)
Requirement already satisfied: gym-notices>=0.0.4 in c:\users\16337\anaconda3\lib
\site-packages (from gym) (0.0.8)
Requirement already satisfied: numpy>=1.18.0 in c:\users\16337\anaconda3\lib\site-
packages (from gym) (1.21.5)
Requirement already satisfied: cloudpickle>=1.2.0 in c:\users\16337\anaconda3\lib
\site-packages (from gym) (2.0.0)
Requirement already satisfied: importlib-metadata>=4.8.0 in c:\users\16337\anacond
a3\lib\site-packages (from gym) (4.11.3)
Requirement already satisfied: zipp>=0.5 in c:\users\16337\anaconda3\lib\site-pack
ages (from importlib-metadata>=4.8.0->gym) (3.8.0)
Note: you may need to restart the kernel to use updated packages.
```

In [174…
```
import gym

# Import and initialize Mountain Car Environment
env = gym.make('MountainCar-v0')
env.reset()

# Define Q-learning function
def QLearning(env, learning, discount, epsilon, min_eps, episodes):
    # Determine size of discretized state space
    num_states = (env.observation_space.high - env.observation_space.low)*\
                    np.array([10, 100])
    num_states = np.round(num_states, 0).astype(int) + 1

    # Initialize Q table
    Q = np.random.uniform(low = -1, high = 1,
                          size = (num_states[0], num_states[1],
                                  env.action_space.n))

    # Initialize variables to track rewards
    reward_list = []
    ave_reward_list = []

    # Calculate episodic reduction in epsilon
    reduction = (epsilon - min_eps)/episodes

    # Run Q learning algorithm
    for i in range(episodes):
        # Initialize parameters
        done = False
        tot_reward, reward = 0,0
        state = env.reset()

        # Discretize state
        state_adj = (state[0] - env.observation_space.low)*np.array([10, 100])
        state_adj = np.round(state_adj, 0).astype(int)

        while done != True:
            # Render environment for last five episodes
            if i >= (episodes - 20):
                env.render()

            # Determine next action - epsilon greedy strategy
            if np.random.random() < 1 - epsilon:
                action = np.argmax(Q[state_adj[0], state_adj[1]])
```

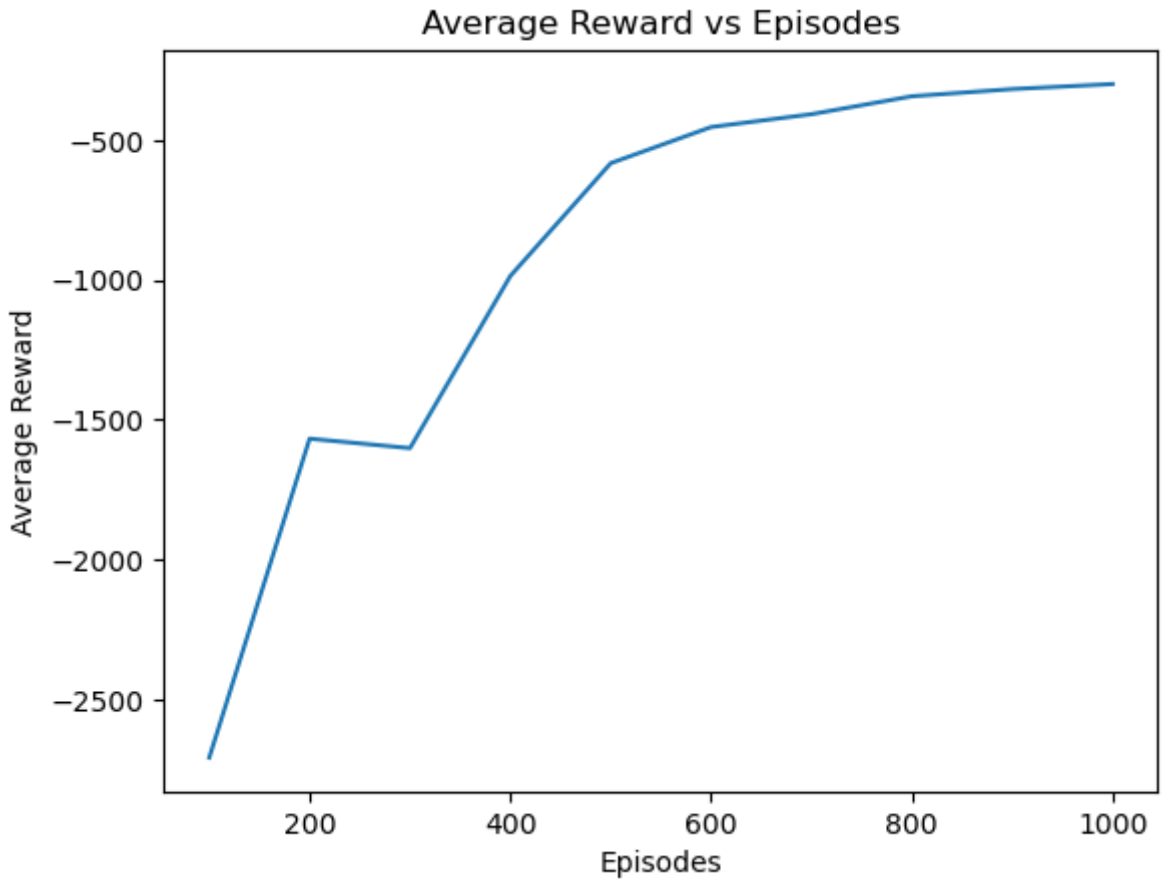
```
            else:
                action = np.random.randint(0, env.action_space.n)

            # Get next state and reward
            state2, reward, done, *info = env.step(action)

            # Discretize state2
            state2_adj = (state2 - env.observation_space.low)*np.array([10, 100])
            state2_adj = np.round(state2_adj, 0).astype(int)

            #Allow for terminal states
            if done and state2[0] >= 0.5:
                Q[state_adj[0], state_adj[1], action] = reward

            # Adjust Q value for current state
            else:
                delta = learning*(reward +
                                  discount*np.max(Q[state2_adj[0],
                                                    state2_adj[1]]) -
                                  Q[state_adj[0], state_adj[1],action])
                Q[state_adj[0], state_adj[1],action] += delta

            # Update variables
            tot_reward += reward
            state_adj = state2_adj

        # Decay epsilon
        if epsilon > min_eps:
            epsilon -= reduction

        # Track rewards
        reward_list.append(tot_reward)

        if (i+1) % 100 == 0:
            ave_reward = np.mean(reward_list)
            ave_reward_list.append(ave_reward)
            reward_list = []



    return ave_reward_list
```

In [175…

```
# Run Q-learning algorithm for 1000 episodes
# QLearning(env, learning, discount, epsilon, min_eps, episodes)
rewards = QLearning(env, 0.2, 0.9, 0.8, 0, 1000)
plt.plot(100*(np.arange(len(rewards)) + 1), rewards)
plt.xlabel('Episodes')
plt.ylabel('Average Reward')
plt.title('Average Reward vs Episodes')
```

Out[175]:

```
Text(0.5, 1.0, 'Average Reward vs Episodes')
```

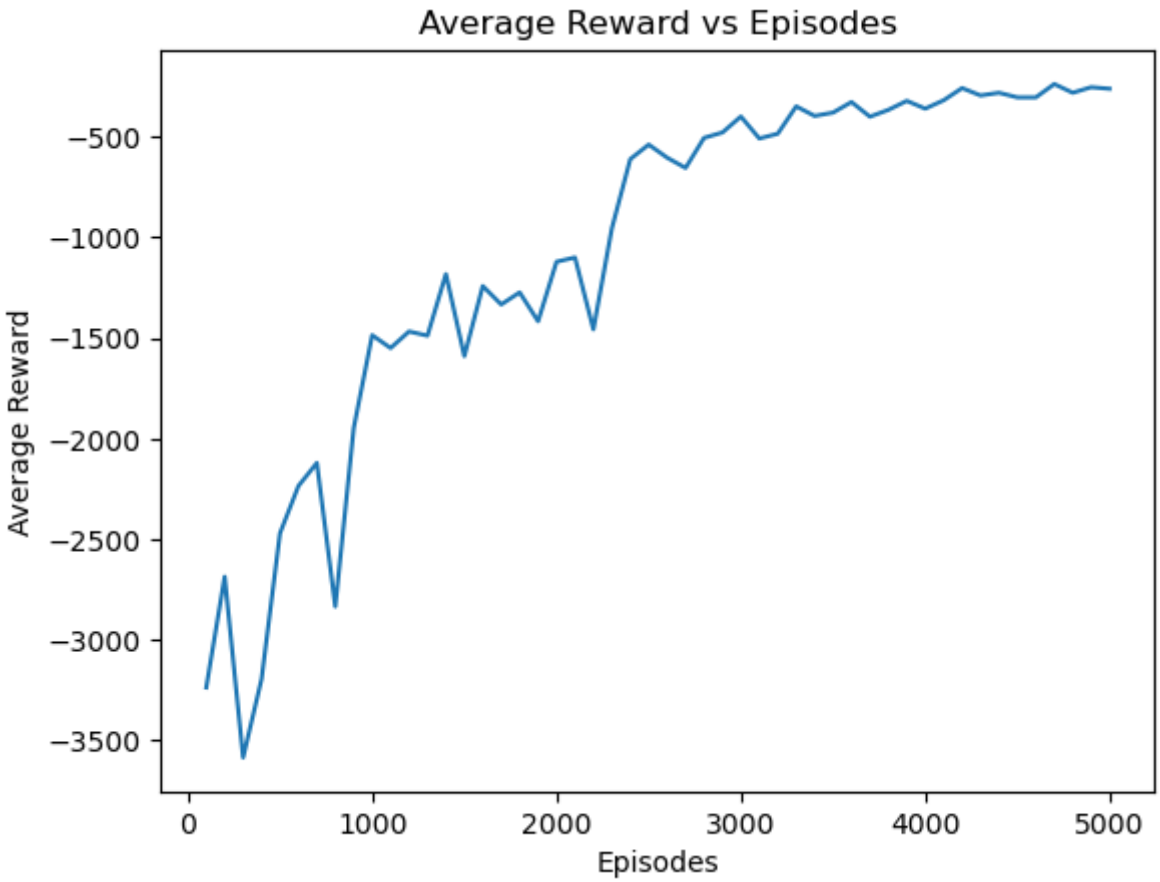


In [176…

```
# Run Q-learning algorithm for 5000 episodes
# QLearning(env, learning, discount, epsilon, min_eps, episodes)
rewards = QLearning(env, 0.2, 0.9, 0.8, 0, 5000)
plt.plot(100*(np.arange(len(rewards)) + 1), rewards)
plt.xlabel('Episodes')
plt.ylabel('Average Reward')
plt.title('Average Reward vs Episodes')
```

Out[176]:

```
Text(0.5, 1.0, 'Average Reward vs Episodes')
```



Plotting the average reward vs the episode number for the 5000 episodes, we can see that, initially, the average reward is fairly flat, with each run terminating once the maximum 200 movements is reached. This is the exploration phase of the algorithm. Nevertheless, in the final 1000 episodes, the algorithm takes what it's learned through exploration and exploits it in order to increase the average reward, with the episodes now ending in less than 200 movements, as the car learns to ascend the mountain. This exploitation phase is only possible because the algorithm was given sufficient time to explore the environment, which is why the car was unable to climb the mountain when the algorithm was only run for 500 episodes.

In [ ]: