Configuration Manual

MSc Research Project
MSCDATOP

Tom MacNamara
Student ID: x19144768

School of Computing
National College of Ireland

Supervisor: Dr Catherine Mulwa

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Tom MacNamara		
Student ID:	x19144768		
Programme:	MSCDATOP	**Year:**	2022
Module:	MSC Research Project		
Lecturer:	Dr Catherine Mulwa		
Submission Due Date:	19/09/2022		
Project Title:	MSC Research Project - Configuration Document		
Word Count:	3492	**Page Count:**	20

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: [illegible]

Date: 18/09/2022

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Document

Tom MacNamara

Student Number: x19144768

1 Contents

1 Hardware and Software

Analysis was carried out on a desktop PC using Windows 10 and a laptop computer running Windows 11. The majority of programming was carried out on the desktop. The desktop features an Intel 6600k processor, an Nvidia 1060 6GB graphics card, and 16GB of DDR4 RAM. All of the models were trained and all predictions were made using this hardware.

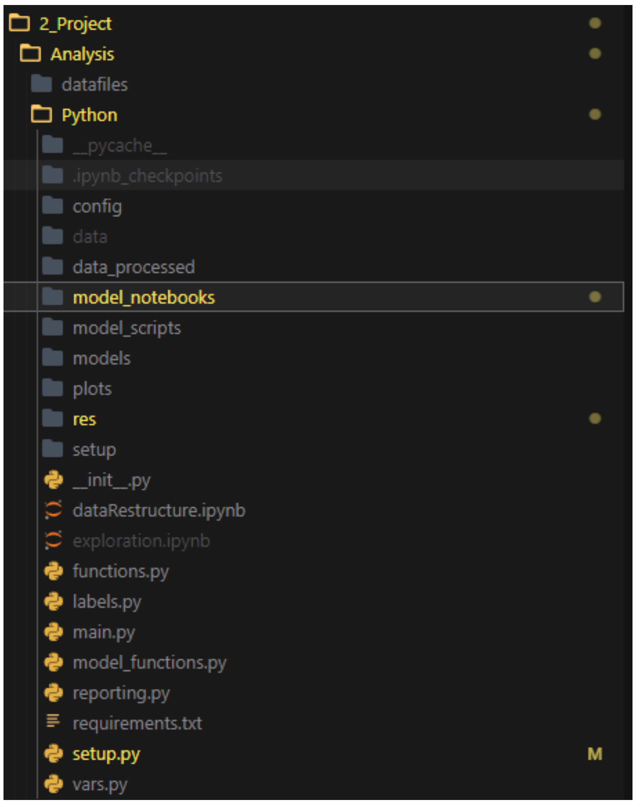
Analysis was conducted in the Python programming language using the Microsoft VSCode Integrated Development Environment (IDE). The analysis was conducted using both python files and Jupyter notebooks. Notebooks were hosted in VSCode. Various versions of Python 3.10 were used in development. Versions of some of the key Python modules used were:

NumPy	1.22.3	**TensorFlow**[1]	2.8.0
Pandas	1.4.2	**Scikit-learn**	1.1.1

Full details of the versions of all of the modules used can be found in the *requirements.txt* file.

2 Technical files

The file structure used in carrying out analysis can be seen in Figure 1. Each file and folder pertains to one specific purpose, as outlined in the following sections.



```
2_Project
  Analysis
    datafiles
    Python
      __pycache__
      .ipynb_checkpoints
      config
      data
      data_processed
      model_notebooks
      model_scripts
      models
      plots
      res
      setup
      __init__.py
      dataRestructure.ipynb
      exploration.ipynb
      functions.py
      labels.py
      main.py
      model_functions.py
      reporting.py
      requirements.txt
      setup.py                                M
      vars.py
```

Figure 1: File structure

2.1 __pycache__ and .ipynb_checkpoints folders

System generated folders. Not used in analysis

2.2 config folder

Contains one file: *config.py.*

2.2.1 config.py

config.py contains functions and variables that are useful in the configuration of models and images. Such variables include the names of the classes, the batch size to use in the tensor model, and some file paths to data locations. Functions to rescale and retrieve the data are contained in config.py. This file is further examined in section 5.1 below.

2.3 data folder

Contains eight subfolders: *bright_dune; crater; dark_dune; impact_ejecta; other; slope_streak; spider; swiss_cheese*. Each of these folders represents one of the classes of the images, and contains all of the images of that class.

[1] All modules were installed through pip. This means TensorFlow did not access CUDA GPU cores, as this requires a conda installation.

2.4 data_processed folder

Contains 26 files: PCA.tsv and 25 data files.

2.4.1 PCA.tsv

A tab separated file that contains three columns to identify some aspects of various versions of Principal Component Analysis that was conducted. The columns identify the number of components, the total variance explained, and the variance explained by each component.

2.4.2 Data files

Multiple versions of Principal Component Analysis were tested. Each version generated five files; four csv files and one pickle file. The files were named as seen in Table 1.

File Name	**File Extension**	**File Role**
PCA15c_200px_51pct_test	.csv	Contained the test data
PCA15c_200px_51pct_train	.csv	Contained the training data
PCA15c_200px_51pct_test_y	.csv	Contained the labels for the test data
PCA15c_200px_51pct_train_y	.csv	Contained the labels for the training data
PCA15c_200px_51pct	.pkl	Contained the PCA model in case it was required to re-run.

Each file is named with the number of components, the number of pixels in the image (if it was resized before PCA applied) and the proportion of the data used in PCA.

2.5 model_notebooks folder

Contains seven files, each containing the code for one model type: *knn.ipynb; lightgbm.ipynb; logistic_regression.ipynb; naïve_bayes.ipynb; random_forest.ipynb; SVM.ipynb; tensorflowCNN.ipynb.* **This is a key folder.**

The files in each folder contain the code where the models were trained and fit, and results saved. This folder is further examined in section 4.1 below.

2.6 model_scripts folder

Contains two files: *SVM.py; tensorCNN.py*

This folder was created to contain a streamlined version of each notebook in the model_notebooks folder in the form of a .py file. This method was scrapped during development in lieu of a main file. This is outlined further in Section 2.15 – *main.py.*

2.7 models folder

Artifact of a previous file structure. Not used in final project.

2.8 plots folder

Contains a folder for each model where graphics were generated to visualise metrics.

2.9 res folder

Results folder. Contains eight folders: *Collated; KNN, lightGBM; logisticRegression; NB; RF; SVM; tensorflowCNN.* **This is a key folder.**

Each folder contains the pickle files for various versions of each created model. The *Collated* folder contains one file with information of each of the models. *Collated.tsv* contains six columns: Version Number; Model Name; f1 Score; Recall; Kappa; Accuracy.

2.10 setup folder

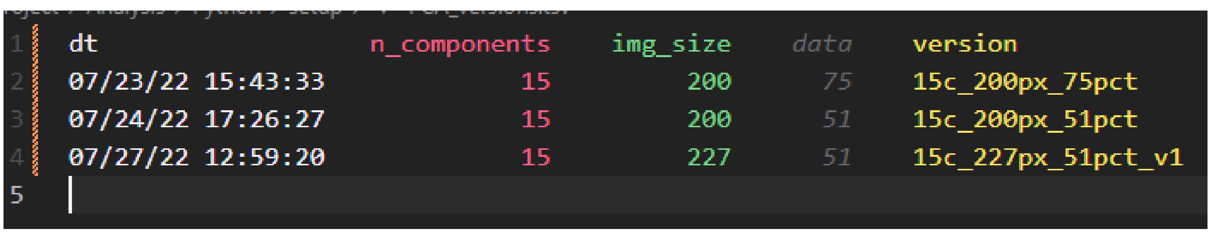
Contains two files: *PCA_versions.tsv; PCA.ipynb*

2.10.1 PCA_versions.tsv

Contains information on each of the iterations of the principal component analysis. Created after multiple iterations so is not a complete history of all PCA attempts. Shown in Figure 2.

2.10.2 PCA.ipynb

Contains the code used to transform the image data into a dataframe through Principal Component Analysis. PCA was conducted using the PCA function from the *sklearn.decomposition* module. **<u>This is a key file.</u>**



```
dt                  n_components   img_size   data   version
07/23/22 15:43:33             15        200     75   15c_200px_75pct
07/24/22 17:26:27             15        200     51   15c_200px_51pct
07/27/22 12:59:20             15        227     51   15c_227px_51pct_v1
```

Figure 2: PCA_versions.tsv

This file is further examined in section 5.2 below.

2.11 __init__.py

File which python uses to identify the folder as a module, allowing file imports.

2.12 dataRestructure.ipynb

Contains the code used to separate each class of image into its own folder at the beginning of analysis. This script was used to create the folders in Section 2.3 above.

2.13 functions.py

Contains some general use functions useful in analysis, such as for saving models with a patterned file name, loading models with context managers, and pretty printing steps of the modelling process. This file is further examined in section 5.3 below.

2.14 labels.py

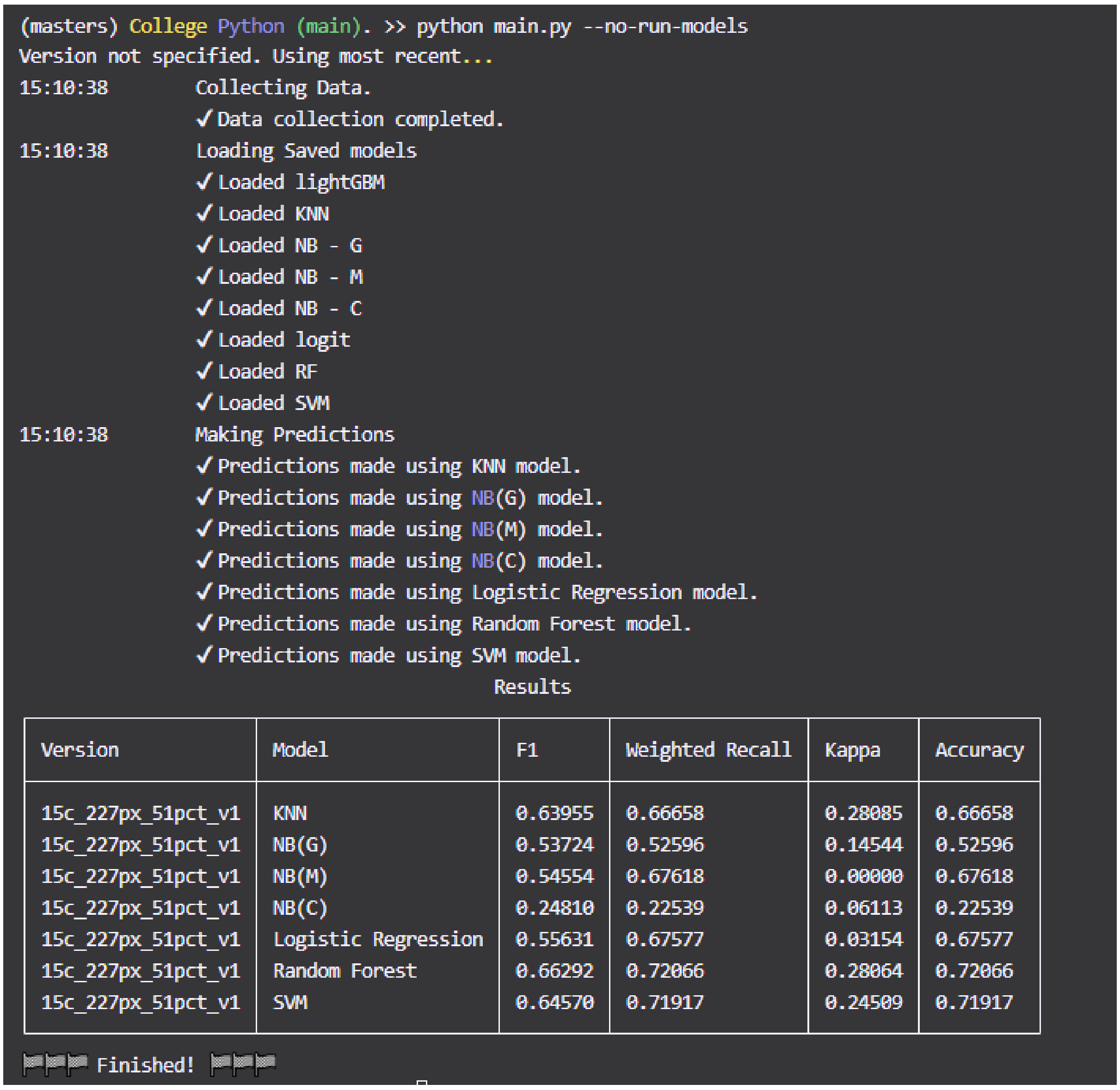
Contains only one function, used to identify which files are associated with which classes. This function is used in dataRestructure.ipynb to reorganise the files into their respective folders. The contents of labels.py can be seen in *Figure 6: labels.py* on page 11.

2.15 main.py

The python script used to load all of the models[2], make predictions, and output a table containing key metrics of the models. The script should be run at the command line. The arguments that can be passed when running the model are `[-v, --version]` to specify the PCA version to use as input data; `[-knn, --n_neighbours]` to manually input the number of neighbours to use in the KNN

[2] Excludes the CNN model as training takes multiple hours. Other models can be loaded with training scores included in the output more easily.

model; and `[--run-models, --no-run-models]` to indicate whether the models should be trained anew, or previously saved models should be used. By default, the `--version` argument returns the most recently saved version. The `--knn` argument uses the default of the KNN function as its default. One of `--run-models` or `--no-run-models` is required. A sample output can be seen in Figure 3 below. This file is further examined in section 5.5 below.



```
(masters) College Python (main). >> python main.py --no-run-models
Version not specified. Using most recent...
15:10:38        Collecting Data.
                ✓Data collection completed.
15:10:38        Loading Saved models
                ✓Loaded lightGBM
                ✓Loaded KNN
                ✓Loaded NB - G
                ✓Loaded NB - M
                ✓Loaded NB - C
                ✓Loaded logit
                ✓Loaded RF
                ✓Loaded SVM
15:10:38        Making Predictions
                ✓Predictions made using KNN model.
                ✓Predictions made using NB(G) model.
                ✓Predictions made using NB(M) model.
                ✓Predictions made using NB(C) model.
                ✓Predictions made using Logistic Regression model.
                ✓Predictions made using Random Forest model.
                ✓Predictions made using SVM model.
                                           Results
 ┌────────────────────┬─────────────────────┬─────────┬─────────────────┬─────────┬──────────┐
 │ Version            │ Model               │ F1      │ Weighted Recall │ Kappa   │ Accuracy │
 ├────────────────────┼─────────────────────┼─────────┼─────────────────┼─────────┼──────────┤
 │ 15c_227px_51pct_v1 │ KNN                 │ 0.63955 │ 0.66658         │ 0.28085 │ 0.66658  │
 │ 15c_227px_51pct_v1 │ NB(G)               │ 0.53724 │ 0.52596         │ 0.14544 │ 0.52596  │
 │ 15c_227px_51pct_v1 │ NB(M)               │ 0.54554 │ 0.67618         │ 0.00000 │ 0.67618  │
 │ 15c_227px_51pct_v1 │ NB(C)               │ 0.24810 │ 0.22539         │ 0.06113 │ 0.22539  │
 │ 15c_227px_51pct_v1 │ Logistic Regression │ 0.55631 │ 0.67577         │ 0.03154 │ 0.67577  │
 │ 15c_227px_51pct_v1 │ Random Forest       │ 0.66292 │ 0.72066         │ 0.28064 │ 0.72066  │
 │ 15c_227px_51pct_v1 │ SVM                 │ 0.64570 │ 0.71917         │ 0.24509 │ 0.71917  │
 └────────────────────┴─────────────────────┴─────────┴─────────────────┴─────────┴──────────┘
🏁🏁🏁 Finished! 🏁🏁🏁
```

Figure 3: main.py sample

2.16 model_functions.py

Contains functions useful to running each of the models in main.py if the `--run-models` flag is specified. Rather than each model requiring the full process of setup, fitting, and predicting within main.py, model_functions.py contains functions for each model that conducts all of these steps. These functions are then imported to main.py and can be run in place. This file is further examined in section 5.4 below.

2.17 reporting.py

This file contains functions used to report on the outputs of the models. This includes confusion matrix generation, metric generation and recording, and simple value counts of predictions. The `getMetrics()` function is used to record the f-score, weighted recall, kappa, and accuracy figures used for analysis. The `recordScores()` function takes these scores as an input to save the details to *collated.tsv*. This file is further examined in section 5.6 below.

2.18 requirements.txt

A file generated by the python pip module using the `pip freeze` command. Contains a list of all modules used, dependencies of these models, and the version installed at the time the command was run.

2.19 setup.py

A file containing functions useful in the setting up of the modelling environment. Contains some overlap with config.py (Section 2.2.1). This file is further examined in section 5.7 below.

2.20 vars.py

Artefact from previous file structure. Not used in final analysis.

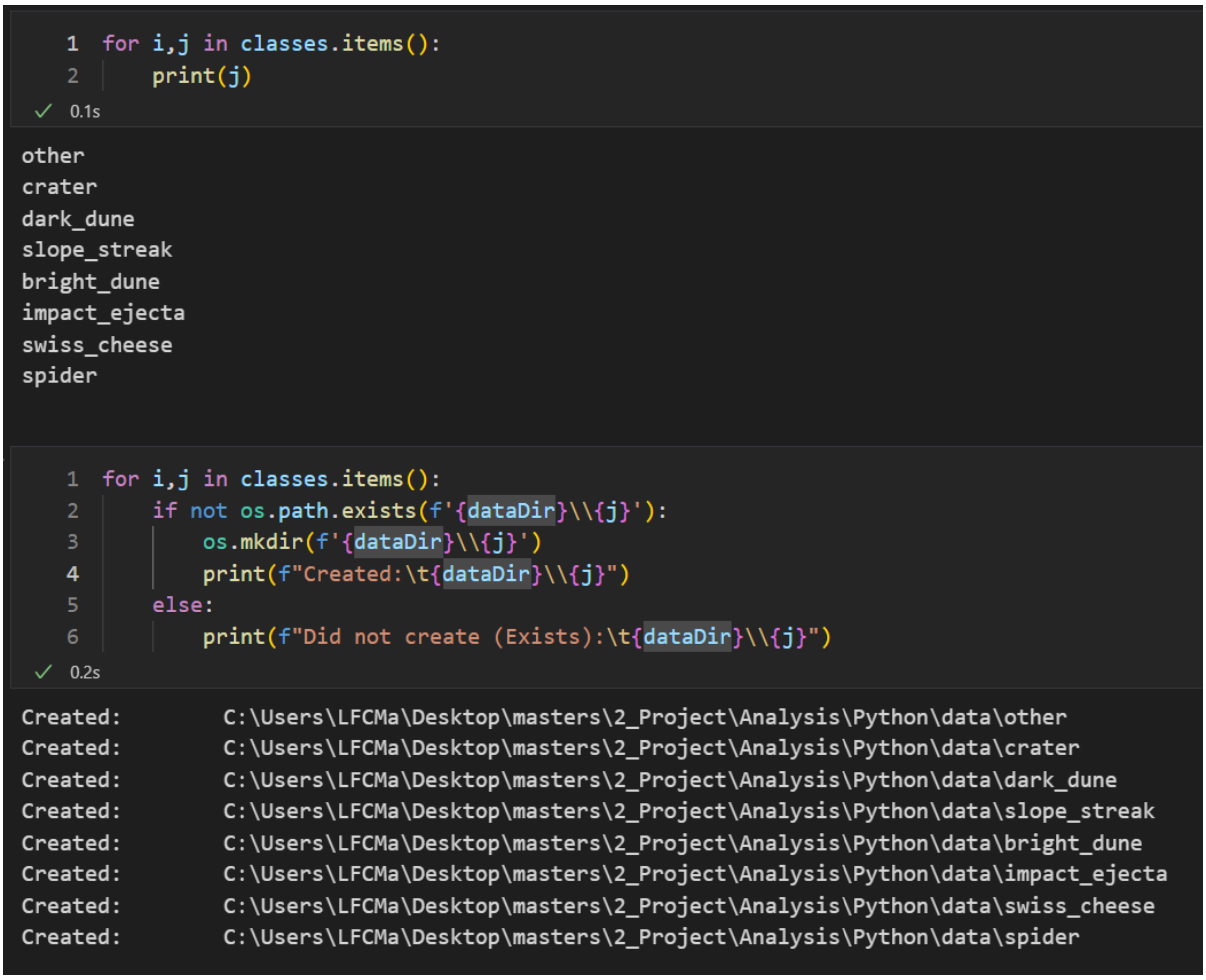
3 Data and Setup

Data were downloaded from the NASA Open Data Portal in the form of a zip folder[3]. Once extracted from the zip, a folder named *Images* was present, alongside a .txt file which acted as a dictionary to identify the class of each image. The *Images* folder contained 73,031 images with names in the format ESP_011623_2100_RED-0069_augmentation. There are 10,433 unique images in the dataset, each appearing once unaltered, and with six copies that have each been augmented in some way. The augmentations and their encoding in the file names are:

Adjustment	**File name example**
Unaltered	ESP_011623_2100_RED-0069.jpg
90 degree clockwise rotation	ESP_011623_2100_RED-0069-**r90**.jpg
180 degree clockwise rotation	ESP_011623_2100_RED-0069-**r180**.jpg
270 degree clockwise rotation	ESP_011623_2100_RED-0069-**r270**.jpg
Flip horizontal	ESP_011623_2100_RED-0069-**fh**.jpg
Flip vertical	ESP_011623_2100_RED-0069-**fv**.jpg
Random brightness adjustment	ESP_011623_2100_RED-0069-**brt**.jpg

In order to properly structure the images for a model, each image needed to be placed in a subfolder. Each class should have a folder containing only images from that class. This file structure was achieved programmatically using Python with the *os* library. This was carried out in the *dataRestructure.ipynb* file. The data directory was specified and `os.mkdir()` was used to create subfolders within that directory (Figure 4).

[3] https://data.nasa.gov/Space-Science/Mars-orbital-image-HiRISE-labeled-data-set-version/egmv-36wq



```python
for i,j in classes.items():
    print(j)
```

✓ 0.1s

```
other
crater
dark_dune
slope_streak
bright_dune
impact_ejecta
swiss_cheese
spider
```

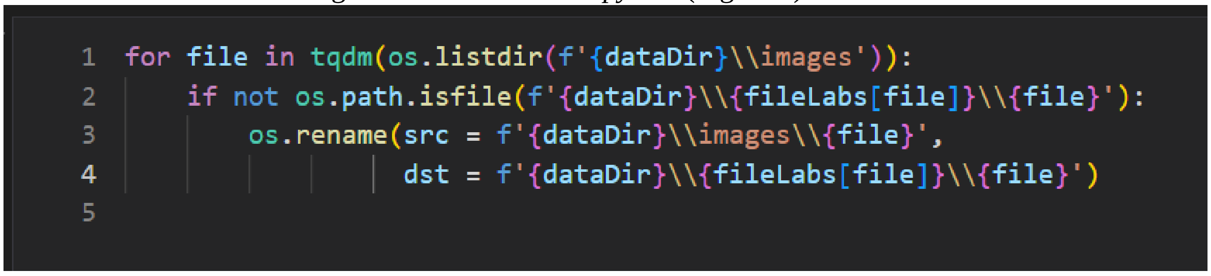
```python
for i,j in classes.items():
    if not os.path.exists(f'{dataDir}\\{j}'):
        os.mkdir(f'{dataDir}\\{j}')
        print(f"Created:\t{dataDir}\\{j}")
    else:
        print(f"Did not create (Exists):\t{dataDir}\\{j}")
```

✓ 0.2s

```
Created:        C:\Users\LFCMa\Desktop\masters\2_Project\Analysis\Python\data\other
Created:        C:\Users\LFCMa\Desktop\masters\2_Project\Analysis\Python\data\crater
Created:        C:\Users\LFCMa\Desktop\masters\2_Project\Analysis\Python\data\dark_dune
Created:        C:\Users\LFCMa\Desktop\masters\2_Project\Analysis\Python\data\slope_streak
Created:        C:\Users\LFCMa\Desktop\masters\2_Project\Analysis\Python\data\bright_dune
Created:        C:\Users\LFCMa\Desktop\masters\2_Project\Analysis\Python\data\impact_ejecta
Created:        C:\Users\LFCMa\Desktop\masters\2_Project\Analysis\Python\data\swiss_cheese
Created:        C:\Users\LFCMa\Desktop\masters\2_Project\Analysis\Python\data\spider
```

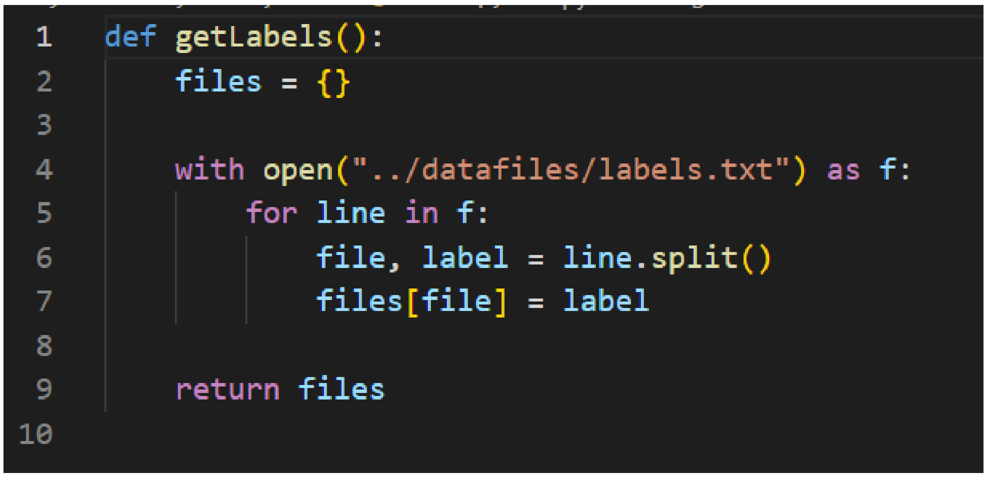
Figure 4: Folder creation

After this step the *data* folder contained a subfolder for each of the data classes, and the original *Images* folder. All images remained in the *Images* folder. The `os.rename()` function was then used to move each of the images into its appropriate folder (Figure 5). This was done by referencing the *labels.txt* file from the original zip folder. The `fileLabs` variable is a python dictionary with the contents of *labels.txt* and is generated in the *labels.py* file (Figure 6).



```python
for file in tqdm(os.listdir(f'{dataDir}\\images')):
    if not os.path.isfile(f'{dataDir}\\{fileLabs[file]}\\{file}'):
        os.rename(src = f'{dataDir}\\images\\{file}',
                  dst = f'{dataDir}\\{fileLabs[file]}\\{file}')

```

Figure 5: Moving image files



```python
def getLabels():
    files = {}

    with open("../datafiles/labels.txt") as f:
        for line in f:
            file, label = line.split()
            files[file] = label

    return files
```

Figure 6: labels.py

---

4 Model creation

The model_notebooks folder (Section 2.5, Page 6) contains scripts for each of the models generated. Each of these notebooks follows the same structure, changing only the model function which is used. Each of the files is split into multiple sections.

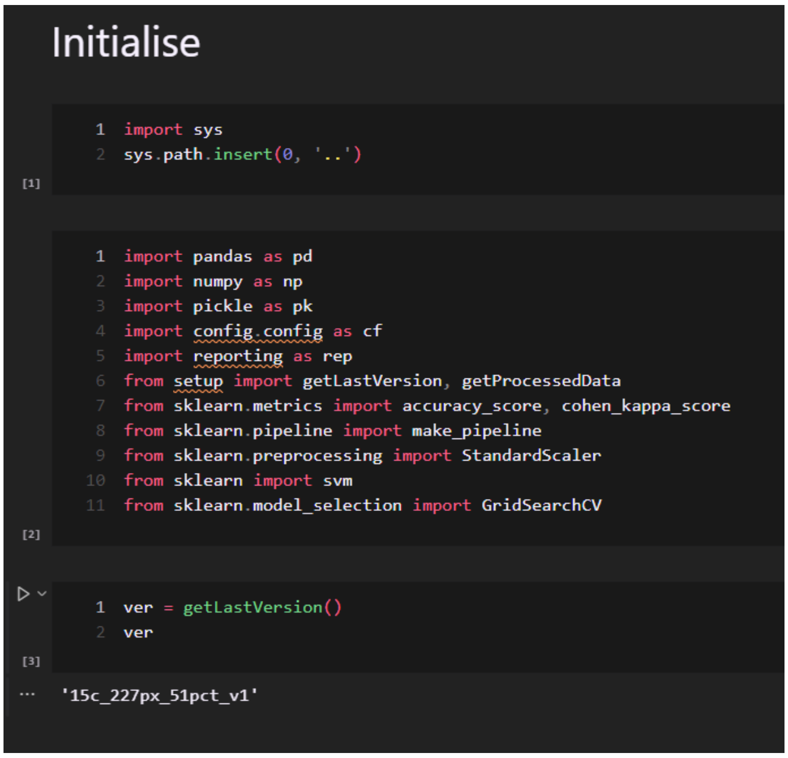
1. Initialise
2. Load Data
3. Initialise Model
4. Make Predictions
5. Reporting
6. Save Model

Each of these sections is explained further in the following section. For this example, *SVM.ipynb* was chosen. As mentioned above, the general structure is the same for all of the modelling approaches.

4.1 Model Notebooks

4.1.1 Initialise

The initialise section contains three code blocks, as seen in Figure 7 below. The first code block imports the *sys* module and adds `..` to the path, allowing imports of modules stored one folder above the current working folder. Cell 2 imports further modules, including those that are now available thanks to the import from cell 1. These imports are themselves grouped. Lines 1-3 relate to the models and the data. Lines 4-6 import the other modules which were created by the author. The remainder of the imports relate to the model implementation. The 3rd cell creates the `ver` variable, which contains the version of Principal Component Analysis data to use. `ver` is generated through the `getLastVersion()` function from *setup.py*. This file is further examined in section 5.7 below.



Initialise

```
[1]
import sys
sys.path.insert(0, '..')
```

```
[2]
import pandas as pd
import numpy as np
import pickle as pk
import config.config as cf
import reporting as rep
from setup import getLastVersion, getProcessedData
from sklearn.metrics import accuracy_score, cohen_kappa_score
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn import svm
from sklearn.model_selection import GridSearchCV
```

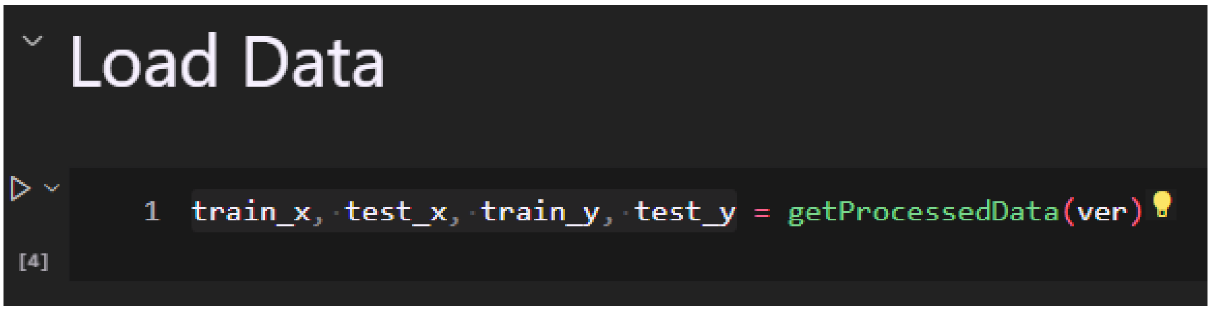
```
[3]
ver = getLastVersion()
ver
```

```
'15c_227px_51pct_v1'
```

Figure 7: Initialise section

4.1.2 Load Data

Contains one cell (Figure 8). This cell is used to generate four datasets, `train_x`, `test_x`, `train_y`, and `test_y`. These datasets are generated using the `getProcessedData()` function from *setup.py*. This file is further examined in section 5.7 below. `train_x` contains the training data from the *PCA_version_number_train.csv* file in the *data_processed* folder. `test_x` contains the test data from the *PCA_version_number_test.csv* file in the *data_processed* folder. `train_y`, and `test_y` contain the instance labels for `train_x` and `test_x`, from the *PCA_version_number_y_train.csv* and *PCA_version_number_y_test.csv files* from the *data_processed* folder.



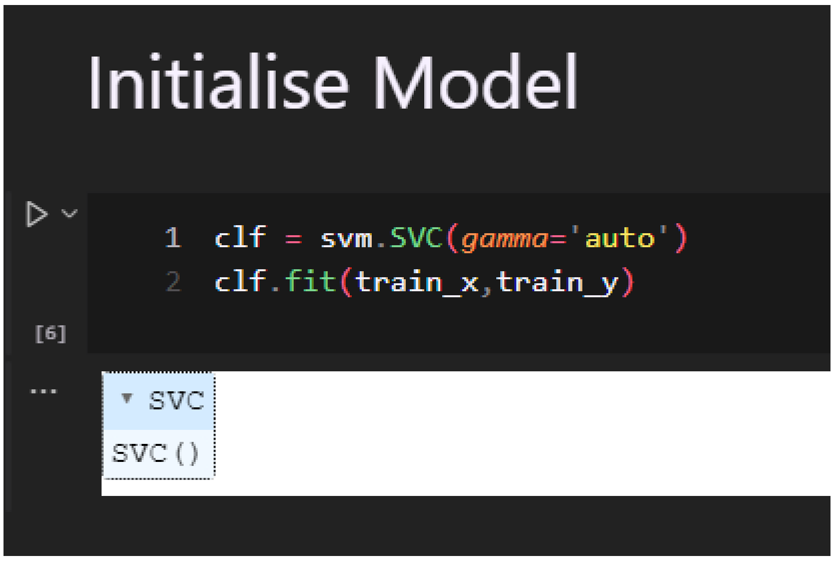
Load Data

```
[4]
train_x, test_x, train_y, test_y = getProcessedData(ver)
```

Figure 8: Load Data

4.1.3 Initialise Model

Contains one cell (Figure 9). Here, the `clf` variable is the model. `clf` is set to an SVC object from the scikit-learn SVM module. The model is fit/trained in this cell, using the `fit` method. The model is fit using the `train_x` and `train_y` variables, teaching the data to associate the values in `train_x` with the classes in `train_y`.



Initialise Model

```
clf = svm.SVC(gamma='auto')
clf.fit(train_x,train_y)
```

```
[6]
SVC
SVC()
```

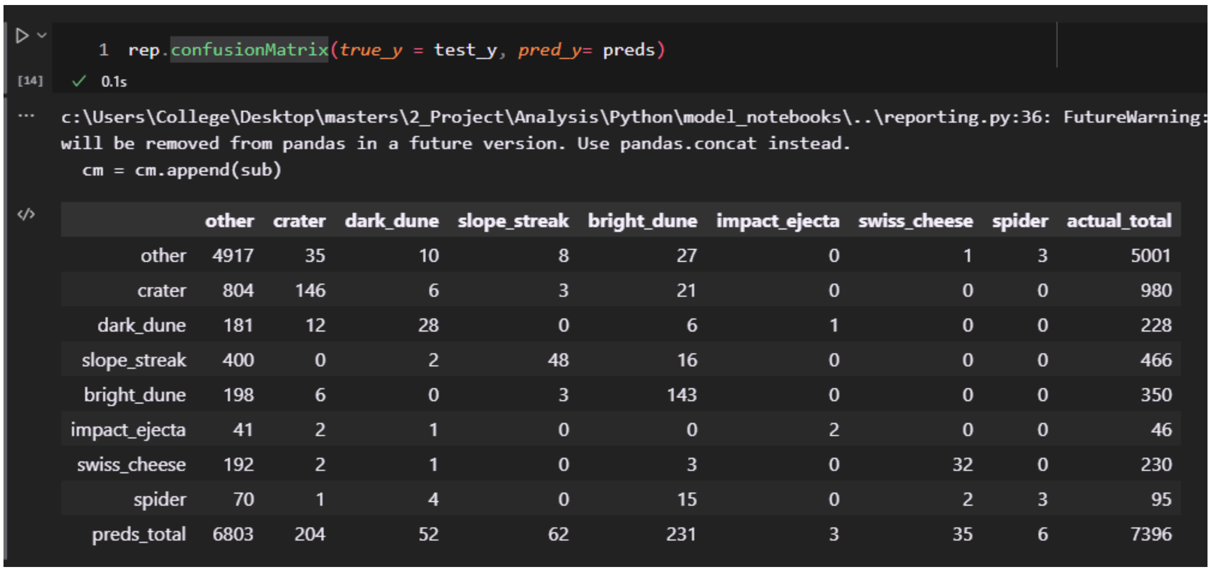
Figure 9: Init Model

4.1.4 Make Predictions

Predictions are made using the `predict` method of the `clf` object. `test_x`, the data values for the test set are used to predict on. The predictions are saved into a variable `preds`. `preds` is a numpy ndarray[4] containing one value per row in the test set. This value is compared to `test_y` in the reporting stage.

4.1.5 Reporting

The reporting stage contains four cells, each of which contain one function taken from reporting.py. These functions are explained in further detail in section 5.6 below. An example output is seen in Figure 10 below.



```
rep.confusionMatrix(true_y = test_y, pred_y= preds)
```

```
[14] ✓ 0.1s
c:\Users\College\Desktop\masters\2_Project\Analysis\Python\model_notebooks\..\reporting.py:36: FutureWarning:
will be removed from pandas in a future version. Use pandas.concat instead.
  cm = cm.append(sub)
```

	other	crater	dark_dune	slope_streak	bright_dune	impact_ejecta	swiss_cheese	spider	actual_total
other	4917	35	10	8	27	0	1	3	5001
crater	804	146	6	3	21	0	0	0	980
dark_dune	181	12	28	0	6	1	0	0	228
slope_streak	400	0	2	48	16	0	0	0	466
bright_dune	198	6	0	3	143	0	0	0	350
impact_ejecta	41	2	1	0	0	2	0	0	46
swiss_cheese	192	2	1	0	3	0	32	0	230
spider	70	1	4	0	15	0	2	3	95
preds_total	6803	204	52	62	231	3	35	6	7396

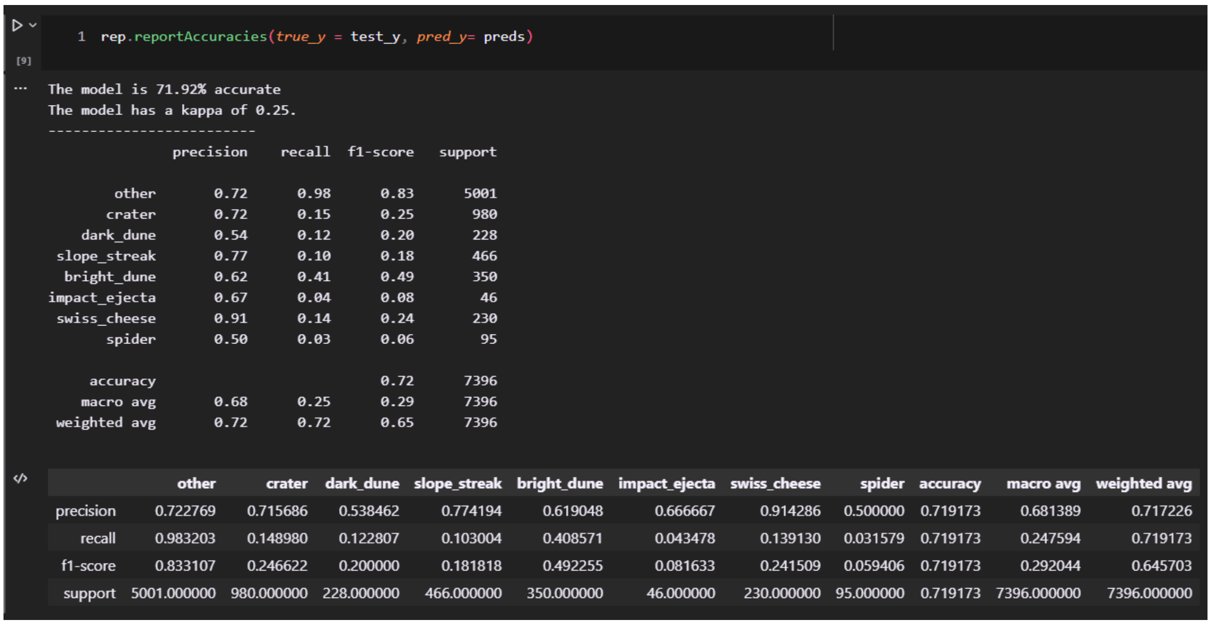
Figure 10: Rep - confusion matrix

In the first reporting step, a confusion matrix is generated, with the predictions presented column-wise and the actual values presented row wise. These confusion matrices were used to determine accuracy of the model through visual inspection, and data were not saved every time a matrix was generated.

[4] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html

These matrices help show how the data imbalance affected predictions, with all cases having a favouritism towards predicting the *other* class.

The `reportAccuracies()` function is used to generate an output containing accuracy, precision, recall, f1, and support scores on a class level and a dataset level (Figure 11).



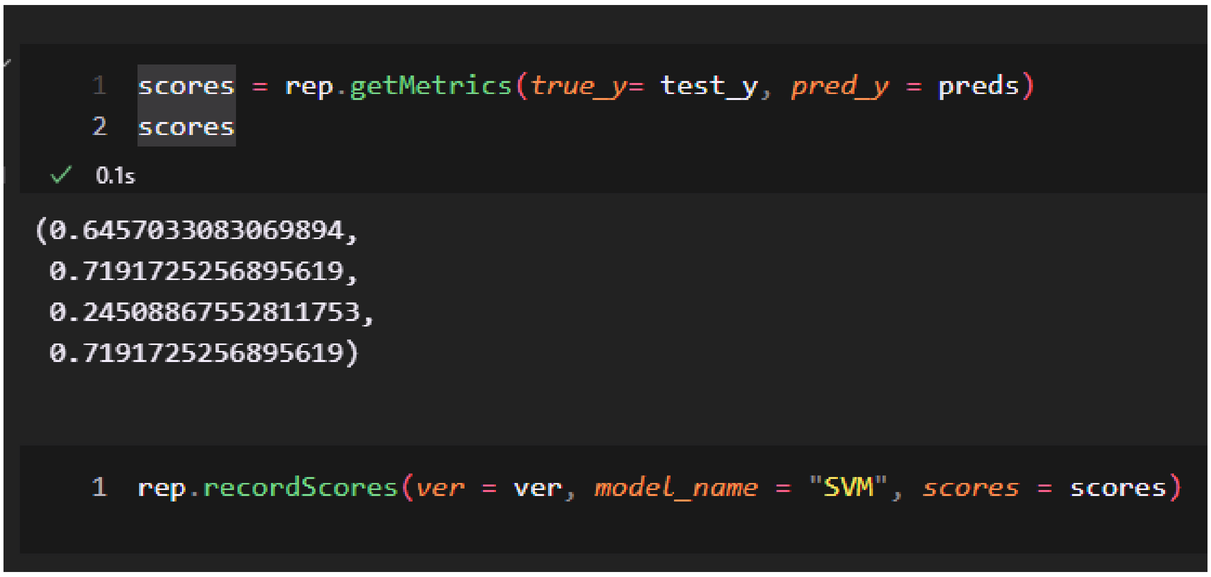
```
rep.reportAccuracies(true_y = test_y, pred_y= preds)
```

```
The model is 71.92% accurate
The model has a kappa of 0.25.
------------------------
               precision    recall  f1-score   support

        other       0.72      0.98      0.83      5001
       crater       0.72      0.15      0.25       980
    dark_dune       0.54      0.12      0.20       228
 slope_streak       0.77      0.10      0.18       466
  bright_dune       0.62      0.41      0.49       350
impact_ejecta       0.67      0.04      0.08        46
 swiss_cheese       0.91      0.14      0.24       230
       spider       0.50      0.03      0.06        95

     accuracy                           0.72      7396
    macro avg       0.68      0.25      0.29      7396
 weighted avg       0.72      0.72      0.65      7396
```

	other	crater	dark_dune	slope_streak	bright_dune	impact_ejecta	swiss_cheese	spider	accuracy	macro avg	weighted avg
precision	0.722769	0.715686	0.538462	0.774194	0.619048	0.666667	0.914286	0.500000	0.719173	0.681389	0.717226
recall	0.983203	0.148980	0.122807	0.103004	0.408571	0.043478	0.139130	0.031579	0.719173	0.247594	0.719173
f1-score	0.833107	0.246622	0.200000	0.181818	0.492255	0.081633	0.241509	0.059406	0.719173	0.292044	0.645703
support	5001.000000	980.000000	228.000000	466.000000	350.000000	46.000000	230.000000	95.000000	0.719173	7396.000000	7396.000000

Figure 11: reportAccuracies

A variable `scores` is defined, which is set to equal the output of the `getMeterics()` function. The four values correspond to the f1, weighted recall, kappa value, and accuracy of the model respectively. These figures are truncated to five decimal places and saved to Collated.tsv using the `recordScores()` function (Figure 12).



```
scores = rep.getMetrics(true_y= test_y, pred_y = preds)
scores
```

✓ 0.1s

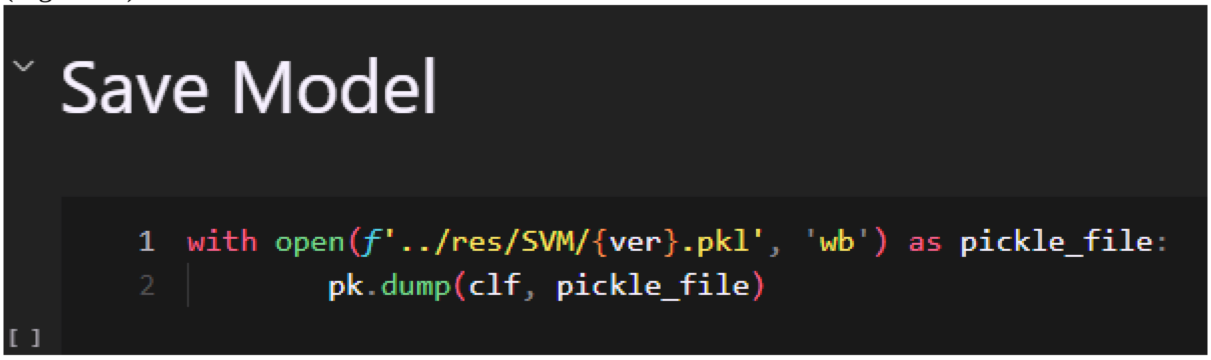
```
(0.6457033083069894,
 0.7191725256895619,
 0.24508867552811753,
 0.7191725256895619)
```

```
rep.recordScores(ver = ver, model_name = "SVM", scores = scores)
```

Figure 12: Collecting and saving scores

4.1.6 Save Model

The model is saved as a .pkl file in order to be accessible after the Jupyter kernel is shut down. (Figure 13)



Save Model

```
with open(f'../res/SVM/{ver}.pkl', 'wb') as pickle_file:
        pk.dump(clf, pickle_file)
```

Figure 13: Saving model

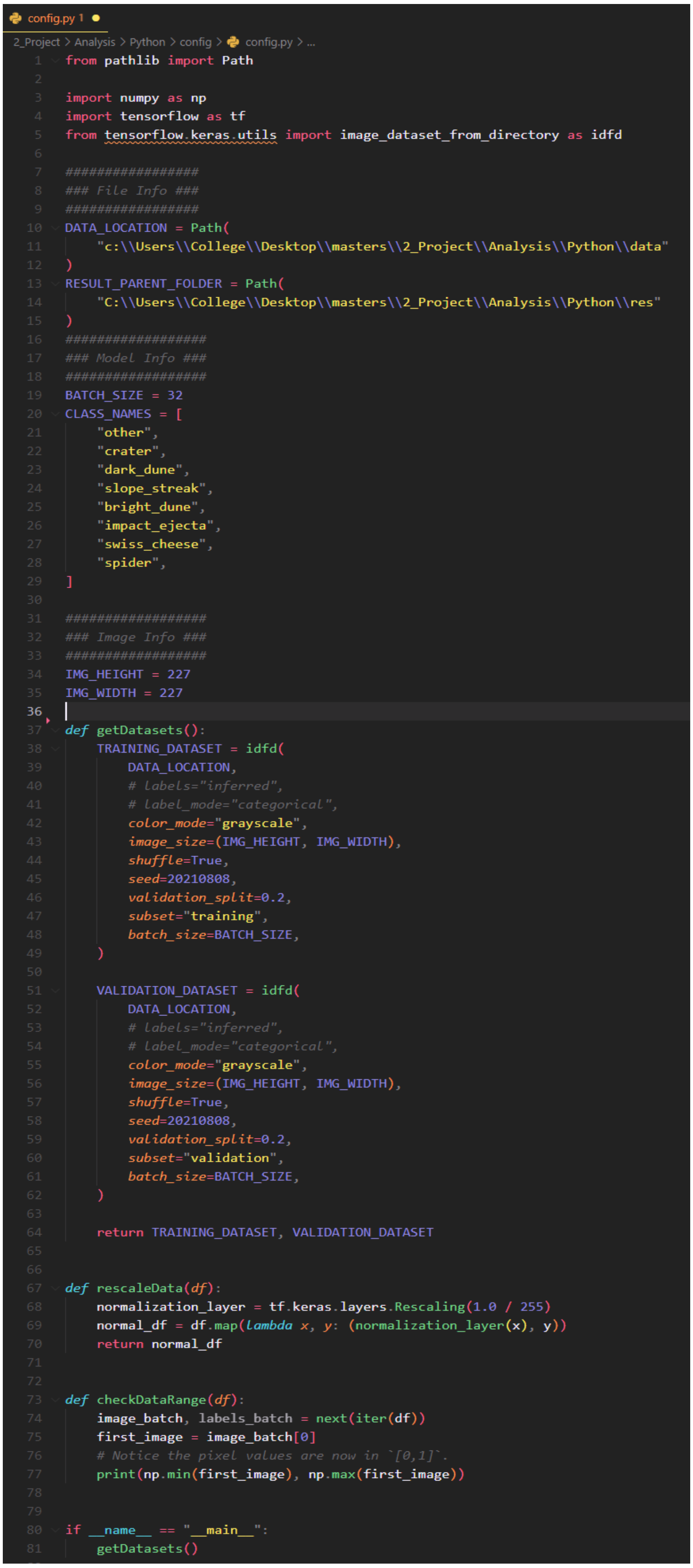
5 Further detail on key files/folders

Some folders and files contain code which is vital to the operation. These are outlined further in this section.

5.1 config.py

config.py contains some information on model structures, file locations, and image details that are key to the successful generation of some of the models. The entire file contents are pictured in Figure 14 on Page 16 below. Details of each of the variables, constants, and functions are outlined below.

Element	**Type**	**Description**
DATA_LOCATION	Constant	The Path to the folder which contains the image data.
RESULT_PARENT_FOLDER	Constant	The Path to the folder where model results are to be saved.
IMG_HEIGHT	Constant	The height of the images in pixels before processing
IMG_WIDTH	Constant	The width of the images in pixels before processing
BATCH_SIZE	Constant	The size of the input batches to a TF model.
CLASS_NAMES	Constant	All of the classes in the original dataset.
getDatasets()	Function	Uses the tensorflow (TF) `image_dataset_from_directory` function to generate training and validation data for use in the TF models.
rescaleData	Function	Normalises the values in a TF Dataset object to between 0…1
checkDataRange	Function	Used to confirm the rescaleData function was successful.



config.py 1 ●

2_Project > Analysis > Python > config > config.py > ...

```python
from pathlib import Path

import numpy as np
import tensorflow as tf
from tensorflow.keras.utils import image_dataset_from_directory as idfd

#################
### File Info ###
#################
DATA_LOCATION = Path(
    "c:\\Users\\College\\Desktop\\masters\\2_Project\\Analysis\\Python\\data"
)
RESULT_PARENT_FOLDER = Path(
    "C:\\Users\\College\\Desktop\\masters\\2_Project\\Analysis\\Python\\res"
)
##################
### Model Info ###
##################
BATCH_SIZE = 32
CLASS_NAMES = [
    "other",
    "crater",
    "dark_dune",
    "slope_streak",
    "bright_dune",
    "impact_ejecta",
    "swiss_cheese",
    "spider",
]

##################
### Image Info ###
##################
IMG_HEIGHT = 227
IMG_WIDTH = 227

def getDatasets():
    TRAINING_DATASET = idfd(
        DATA_LOCATION,
        # labels="inferred",
        # label_mode="categorical",
        color_mode="grayscale",
        image_size=(IMG_HEIGHT, IMG_WIDTH),
        shuffle=True,
        seed=20210808,
        validation_split=0.2,
        subset="training",
        batch_size=BATCH_SIZE,
    )

    VALIDATION_DATASET = idfd(
        DATA_LOCATION,
        # labels="inferred",
        # label_mode="categorical",
        color_mode="grayscale",
        image_size=(IMG_HEIGHT, IMG_WIDTH),
        shuffle=True,
        seed=20210808,
        validation_split=0.2,
        subset="validation",
        batch_size=BATCH_SIZE,
    )

    return TRAINING_DATASET, VALIDATION_DATASET


def rescaleData(df):
    normalization_layer = tf.keras.layers.Rescaling(1.0 / 255)
    normal_df = df.map(lambda x, y: (normalization_layer(x), y))
    return normal_df


def checkDataRange(df):
    image_batch, labels_batch = next(iter(df))
    first_image = image_batch[0]
    # Notice the pixel values are now in `[0,1]`.
    print(np.min(first_image), np.max(first_image))


if __name__ == "__main__":
    getDatasets()
```

Figure 14: config.py

5.2 PCA.ipynb

The notebook used to generate the datasets used in the non-TF models works by collecting the image data, storing the pixel values for each image. Each image is stored as one row in a DataFrame, with each pixel's greyscale value representing one column. PCA is applied using the PCA function from the sklearn.decomposition library. Many different versions were created, some where the image data were resized before PCA, some with varying numbers of components in the PCA. A final dataset was created with 15 components on data that were not resized. This accounted for 87.45% of the variance in the data.

5.3 functions.py

The functions.py file contains 7 functions which are used in the modelling process.

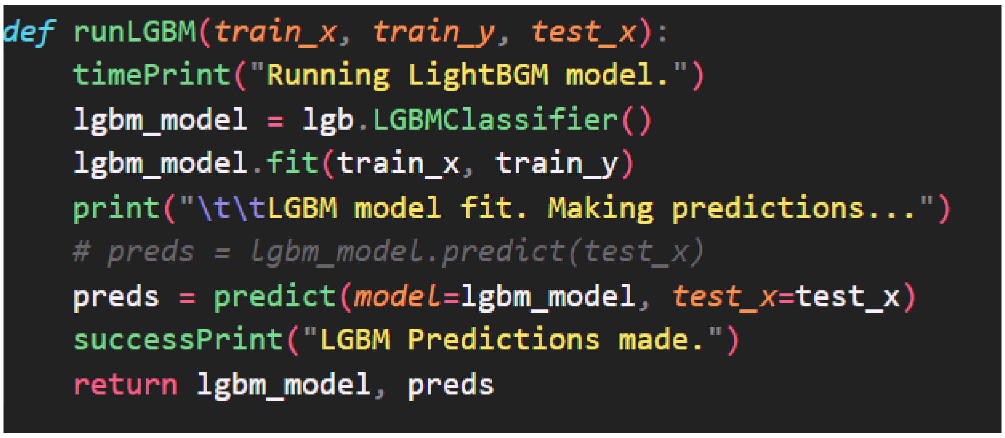
Element	**Type**	**Description**
timePrint	Function	Prints an input string and the time it was called in the format `HH:MM:SS String`
pathExists	Function	A wrapper for the pathlib `Path.exists()` function.
saveModelHistory	Function	Saves a TensorFlow model’s history attribute to a specified path as a .pkl file.
loadModelHistory	Function	Loads a pkl file that was generated by `saveModelHistory` above.
saveTensorModel	Function	Saves a TensorFlow Sequential object to the specified path using the `tf.keras.models.save_model` function.
loadTensorModel	Function	Wrapper for the `tf.keras.models.load_model` function.
loadDataFrame	Function	Wrapper for the pandas `read_csv` function.

5.4 model_functions.py

model_functions.py contains the functions to run the models which are utilised in main.py. Each of the model generation functions consists of the same three key steps: initialising, fitting, and predicting. These are typically carried out using the same keywords, as most of the models are from the scikit-learn module. An example can be seen in Figure 15 below.

Element	**Type**	**Inputs**	**Description**
runLGBM	Function	`train_x`, `train_y`, `test_x`	Wrapper to initialise, fit, and make predictions on the LightGBM model.
runKNN	Function	`train_x`, `train_y`, `test_x` `n_neighbours`	Wrapper to initialise, fit, and make predictions on the K nearest neighbours model. User specifies the number of neighbours using the `n_neighbours` input variable.
runNB	Function	`train_x`, `train_y`, `test_x`, `nb_type`	Wrapper to initialise, fit, and make predictions on the Naïve Bayes model. User specifies the type of naïve bayes (Gaussian, Complement, Multinomial) using the `nb_type` input variable.
runLogit	Function	`train_x`, `train_y`, `test_x`	Wrapper to initialise, fit, and make predictions on the logistic regression model.

runRF	Function	`train_x`, `train_y`, `test_x`	Wrapper to initialise, fit, and make predictions on the Random Forest model.
runSVM	Function	`train_x`, `train_y`, `test_x`	Wrapper to initialise, fit, and make predictions on the SVM model.
runCNN	Function	`train_x`, `train_y`, `test_x`	Wrapper to initialise, fit, and make predictions on the CNN model.
predict	Function	`model`, `test_y`	A wrapper function for the model.predict method.

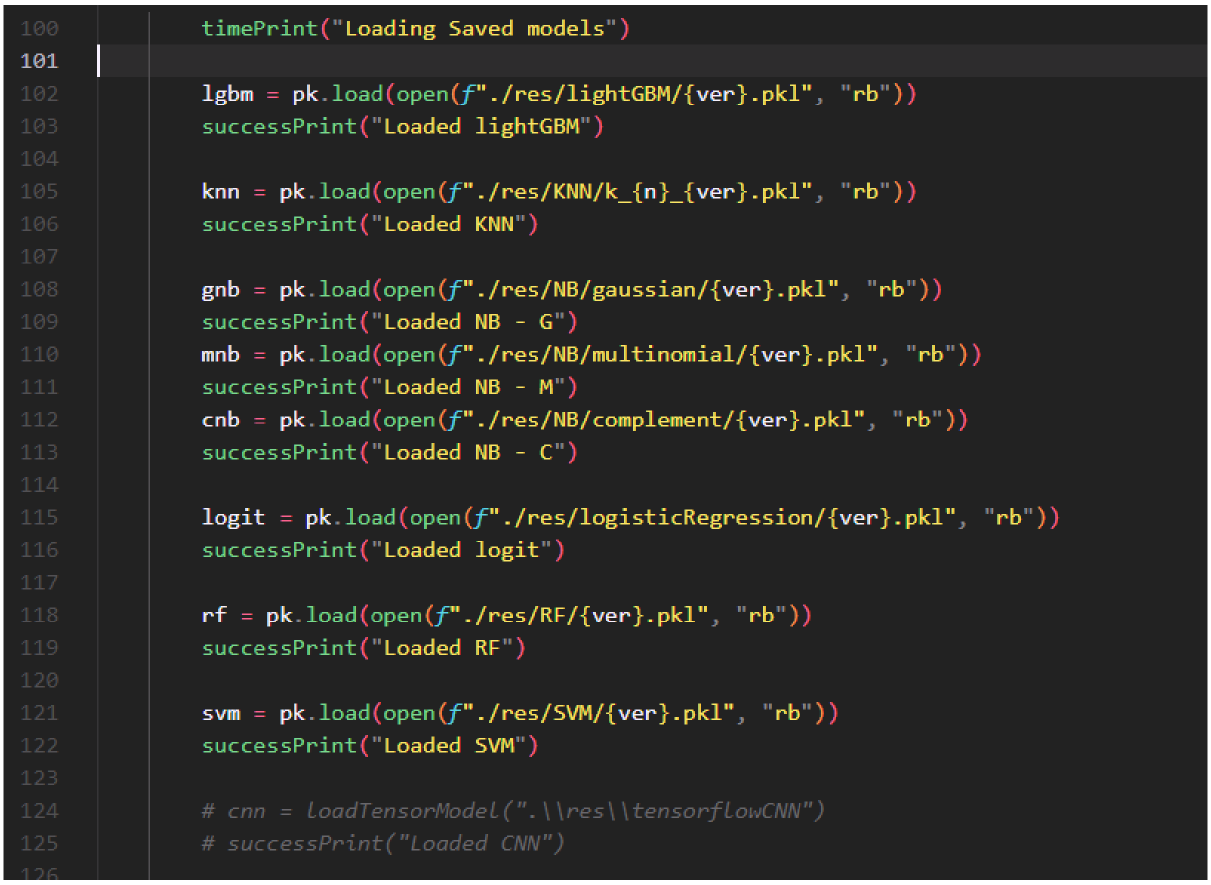


```python
def runLGBM(train_x, train_y, test_x):
    timePrint("Running LightBGM model.")
    lgbm_model = lgb.LGBMClassifier()
    lgbm_model.fit(train_x, train_y)
    print("\t\tLGBM model fit. Making predictions...")
    # preds = lgbm_model.predict(test_x)
    preds = predict(model=lgbm_model, test_x=test_x)
    successPrint("LGBM Predictions made.")
    return lgbm_model, preds
```

Figure 15: runLGBM function

5.5 main.py

main.py is the primary script used in generating the data for all models. Each of the functions created in model_functions.py are used to generate all of the models and results in one file. An example of running main.py can be seen in Figure 3 above. main.py does not introduce any new functionality, it simply uses functions taken from other files in the directory. The argparse module is used to define three variables when calling the module from the command line. These variables are the version of data to use, the number of neighbours in the KNN, and whether new models should be generated or saved models be used. When saved models are used, the results are reported in a table on the command line and not saved. This is due to the script being run a large number of times in development, and as the models do not need to be generated it is not time intensive to run again. A screenshot of the process where trained models are loaded is seen in Figure 16 below.



```
        timePrint("Loading Saved models")

        lgbm = pk.load(open(f"./res/lightGBM/{ver}.pkl", "rb"))
        successPrint("Loaded lightGBM")

        knn = pk.load(open(f"./res/KNN/k_{n}_{ver}.pkl", "rb"))
        successPrint("Loaded KNN")

        gnb = pk.load(open(f"./res/NB/gaussian/{ver}.pkl", "rb"))
        successPrint("Loaded NB - G")
        mnb = pk.load(open(f"./res/NB/multinomial/{ver}.pkl", "rb"))
        successPrint("Loaded NB - M")
        cnb = pk.load(open(f"./res/NB/complement/{ver}.pkl", "rb"))
        successPrint("Loaded NB - C")

        logit = pk.load(open(f"./res/logisticRegression/{ver}.pkl", "rb"))
        successPrint("Loaded logit")

        rf = pk.load(open(f"./res/RF/{ver}.pkl", "rb"))
        successPrint("Loaded RF")

        svm = pk.load(open(f"./res/SVM/{ver}.pkl", "rb"))
        successPrint("Loaded SVM")

        # cnn = loadTensorModel(".\\res\\tensorflowCNN")
        # successPrint("Loaded CNN")
```

Figure 16: main.py loading models

5.6 reporting.py

reporting.py contains functions used in analysing the results of the models. These functions are used both as a means of quickly visually assessing model outputs and for thoroughly recording and reporting on key metrics of the models.

Element	**Type**	**Inputs**	**Description**
value_counts	Function	`arr`	Returns the value counts of an array. Used to get the value counts of a target variable array
confusionMatrix	Function	`true_y` `pred_y` `save` `filepath` `filename`	Wrapper for the sklearn.metrics `confusion_matrix` function with some added functionality. Allows saving to disk. Total column and row are added to normal sklearn matrix.
confusionMatrixAccuracy	Function	`cmat`	Prints the accuracy for each class in a confusion matrix row-wise.
reportAccuracies	Function	`true_y` `pred_y`	Collates and prints various functions from sklearn.metrics. Includes the `accuracy_score` `cohen_kappa_score` and `classification_report` functions.
getMetrics	Function	`true_y` `pred_y`	Collates and returns the `f1_score`, `recall_score`,

			`cohen_kappa_score`, and `accuracy_score` scikit-learn metrics.
recordScores	Function	`ver` `model_name` `scores`	Records the version number, model name, and output of `getMetrics` into collated.tsv

5.7 setup.py

setup.py contains three functions. These are mainly used in relation to the PCA data. The functions are as below:

Element	**Type**	**Inputs**	**Description**
getLastVersion	Function	`return_all` `path`	Returns the version number of the latest saved PCA data.
getProcessedData	Function	`version` `rescale_data`	Uses pandas `read_csv` to load `train_x` `train_y` `test_x` and `test_y` variables. The csv files relate to the data specified using the `version` input variable. The user can specify if the values should be normalised between 0 and 1 using the `rescale_data` flag.
rescale	Function	`df`	Normalises the data between 0 and 1 in a dataframe.

6 Bibliography

There are no sources in the current document.