```
# -*- coding: utf-8 -*-
"""
Neural network model code used to generate the results presented in the study
"A Rapid Neural Network Based State of Health Estimation Scheme For Screening of End of Life Electric Vehicle Batteries"

Jamie Hathaway 2020
"""

from __future__ import absolute_import, division, print_function, unicode_literals
import tensorflow as tf
import math
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.ticker
from matplotlib import rcParams
import lmfit
import os
import skopt
import skopt.plots
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_validate
from sklearn.ensemble import ExtraTreesRegressor
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process import kernels
import itertools
import time


# OPTIONS
# =======
NEWCASTLE_DATA_DIR = 'Newcastle'
NEWCASTLE_DATA_EXT = '.xlsx'
NEWCASTLE_DATA_CAPACITIES = 'Capacities.xlsx'
NEWCASTLE_DATA_FITTED = 'ECM_Fitted_Data.csv'
NEWCASTLE_DATA_CORRELATIONS = 'ECM_Correlations.csv'
OUTPUT_DIR = 'Outputs'
CELL_CAPACITY = 65 # Ah

MODEL_TRIAL_CONFIGS = [['sigmoid', 5, 1, 0.01, 'funnel', 0.0, 0.0], # 0 baseline model
                       ['elu', 18, 4, 0.01, 'brick', 0.0008887845425637469, 0.1362289727740415]] # 1 optimised model: 0.01630455736930552 val. result, 0.017333753567475525 single CV test result

MODEL_NO = 1                # Which of the NN model trial configs to use. 0 for baseline, 1 for optimised model. Only used for model type 'NN'.
MODEL_TYPE = 'NN'           # The model type to run: 'NN', 'RF' or 'GP' regression.
MODEL_EXCLUDED_PARAMS = None    # Remove a list of fitted parameters from being predictors in the NN model, e.g. ['n_CPE', 'R_w', 'T_w', 'L'].
MODEL_CV_RUNS = None        # Any positive integer number of runs, or None for the maximum of K runs. Useful for running model only once for example.
MODEL_CV_K = 6              # The number K for K-fold cross-validation

HYPEROPT_SEARCH_SPACE = [skopt.space.Categorical(['tanh', 'elu', 'relu', 'sigmoid'], name='activation'),
```

```
                skopt.space.Integer(3, 50, prior='uniform', name='first_neurons'),
                skopt.space.Integer(1, 5, prior='uniform', name='hidden_layers'),
                skopt.space.Real(0.01, 0.5, prior='log-uniform', name='learning_rate'),
                skopt.space.Categorical(['funnel', 'brick'], name='shape'),
                skopt.space.Real(0, 0.05, prior='uniform', name='regularisation'),
                skopt.space.Real(0, 0.5, prior='uniform', name='dropout')]

RUN_POPULATE_DATASET = False       # Forces a repopulation of the ECM dataset by refitting the EIS
 data.
RUN_MODEL = False                  # Train and cross-validate the configured model if True.
RUN_HYPEROPT = False               # Run the hyperparameter optimisation scheme
RUN_PLOT_MODEL_TRAIN_GRAPH = False  # Plot a graph of the model performance with number of t
raining samples. Only for NN model.
RUN_PLOT_CORRELATIONS = False      # Plot the correlations and linear fit between each ECM param
eter and the SoH if True.
# =======

def formatted_param_name(param_name):
    uindex = param_name.find('_')
    if uindex != -1: return '$' + param_name[:uindex+1] + '{' + param_name[uindex+1:] + '}$'
    else: return param_name

def formatted_column_names(column_names):
    names = []
    for param_name in column_names:
        names.append(formatted_param_name(param_name))
    return names

def format_corr_data(x, pos):
    return "{:.2f}".format(x).replace("1.00", "")

# Correlation matrix heatmap code reproduced from https://matplotlib.org/gallery/images_contours_and_fi
elds/image_annotated_heatmap.html#sphx-glr-gallery-images-contours-and-fields-image-annotated-heat
map-py
# J. D. Hunter, "Matplotlib: A 2D Graphics Environment", Computing in Science & Engineering, vol. 9, no.
3, pp. 90-95, 2007
def heatmap(data, row_labels, col_labels, ax=None,
            cbar_kw={}, cbarlabel="", **kwargs):
    """
    Create a heatmap from a numpy array and two lists of labels.

    Parameters
    ----------
    data
        A 2D numpy array of shape (N, M).
    row_labels
        A list or array of length N with the labels for the rows.
    col_labels
        A list or array of length M with the labels for the columns.
    ax
        A `matplotlib.axes.Axes` instance to which the heatmap is plotted.  If
        not provided, use current axes or create a new one.  Optional.
    cbar_kw
        A dictionary with arguments to `matplotlib.Figure.colorbar`.  Optional.
    cbarlabel
```

```
    The label for the colorbar.  Optional.
**kwargs
    All other arguments are forwarded to 'imshow'.
"""

if not ax:
    ax = plt.gca()
im = ax.imshow(data, **kwargs)

cbar = ax.figure.colorbar(im, ax=ax, **cbar_kw)
cbar.ax.set_ylabel(cbarlabel, rotation=-90, va="bottom")

ax.set_xticks(np.arange(data.shape[1]))
ax.set_yticks(np.arange(data.shape[0]))
ax.set_xticklabels(col_labels)
ax.set_yticklabels(row_labels)

# Let the horizontal axes labeling appear on top.
ax.tick_params(top=True, bottom=False,
               labeltop=True, labelbottom=False)
# Rotate the tick labels and set their alignment.
plt.setp(ax.get_xticklabels(), rotation=-30, ha="right",
         rotation_mode="anchor")

# Turn spines off and create white grid.
for edge, spine in ax.spines.items():
    spine.set_visible(False)

ax.set_xticks(np.arange(data.shape[1]+1)-.5, minor=True)
ax.set_yticks(np.arange(data.shape[0]+1)-.5, minor=True)
ax.grid(which="minor", color="w", linestyle='-', linewidth=3)
ax.tick_params(which="minor", bottom=False, left=False)

return im, cbar

# Correlation matrix heatmap code reproduced from https://matplotlib.org/gallery/images_contours_and_fi
elds/image_annotated_heatmap.html#sphx-glr-gallery-images-contours-and-fields-image-annotated-heat
map-py
# J. D. Hunter, "Matplotlib: A 2D Graphics Environment", Computing in Science & Engineering, vol. 9, no.
3, pp. 90-95, 2007
def annotate_heatmap(im, data=None, valfmt="{x:.2f}",
                     textcolors=["black", "white"],
                     threshold=None, absthreshold=False, **textkw):
    """
    A function to annotate a heatmap.

    Parameters
    ----------
    im
        The AxesImage to be labeled.
    data
        Data used to annotate.  If None, the image's data is used.  Optional.
    valfmt
        The format of the annotations inside the heatmap.  This should either
        use the string format method, e.g. "$ {x:.2f}", or be a
```

```
        ‘matplotlib.ticker.Formatter‘.  Optional.
    textcolors
        A list or array of two color specifications.  The first is used for
        values below a threshold, the second for those above.  Optional.
    threshold
        Value in data units according to which the colors from textcolors are
        applied.  If None (the default) uses the middle of the colormap as
        separation.  Optional.
    absthreshold
        Whether the threshold is expressed in terms of the absolute value of
        the data. Useful for diverging or symmetric color maps. Optional.
    **kwargs
        All other arguments are forwarded to each call to ‘text‘ used to create
        the text labels.
    """

    if not isinstance(data, (list, np.ndarray)): data = im.get_array()

    # Normalize the threshold to the images color range.
    if threshold is not None: threshold = im.norm(threshold)
    else: threshold = im.norm(data.max())/2.

    # Set default alignment to center, but allow it to be
    # overwritten by textkw.
    kw = dict(horizontalalignment="center",
              verticalalignment="center")
    kw.update(textkw)

    # Get the formatter in case a string is supplied
    if isinstance(valfmt, str): valfmt = matplotlib.ticker.StrMethodFormatter(valfmt)

    # Loop over the data and create a ‘Text‘ for each "pixel".
    # Change the text’s color depending on the data.
    texts = []
    for i in range(data.shape[0]):
        for j in range(data.shape[1]):
            if absthreshold: kw.update(color=textcolors[int(2*abs(im.norm(data[i, j]) - 0.5) > threshold)])
            else: kw.update(color=textcolors[int(im.norm(data[i, j]) > threshold)])
            text = im.axes.text(j, i, valfmt(data[i, j], None), **kw)
            texts.append(text)

    return texts

def plot_cmatrix(cmatrix, figure_name):
    fig, ax = plt.subplots(1)
    im, cbar = heatmap(cmatrix,
                       row_labels=formatted_column_names(cmatrix.columns),
                       col_labels=formatted_column_names(cmatrix.columns),
                       vmin=-1, vmax=1, ax=ax, cbarlabel=’Correlation coefficient’, cmap=’RdBu’)
    annotate_heatmap(im, valfmt=matplotlib.ticker.FuncFormatter(format_corr_data), threshold=0.25, abst
hreshold=True)
    fig.tight_layout()
    plt.show()
    fig.savefig(figure_name)
```

```
def fit_model(x_data, y_data, model, fit_name=None, fit_weights=None):
    guess_params = model.guess(y_data, x=x_data)
    fit = model.fit(y_data, guess_params, x=x_data, weights=fit_weights)
    if fit.covar is not None: model_fit_errors = np.sqrt(np.diag(fit.covar))
    else: model_fit_errors = np.zeros(len(model.param_names))

    model_fit_params = fit.best_values.copy()
    for index, param in enumerate(model.param_names): model_fit_params[param + '_err'] = model_fit_err
ors[index]
    if fit_name != None: model_fit_params['Name'] = fit_name
    model_fit_params['Chisqr'] = fit.chisqr
    return fit, model_fit_params

def plot_fit(x_data, y_data, fit, data_label, fit_label, x_label, y_label, fig_name, x_err=None, y_err=None):
    fig, ax = plt.subplots(1)
    if x_err is None and y_err is None: ax.plot(x_data, y_data, 'b.', label=data_label)
    else: ax.errorbar(x_data, y_data, yerr=y_err, xerr=x_err, fmt='b.', capsize=4)
    ax.plot(x_data, fit.best_fit, 'k-', label=fit_label)
    ax.set_xlabel(x_label)
    ax.set_ylabel(y_label)
    ax.legend()
    plt.tight_layout()
    plt.show()
    fig.savefig(fig_name)

def tanh(x):
    return (1-np.exp(-2*x))/(1+np.exp(-2*x))

def coth(x):
    return (1+np.exp(-2*x))/(1-np.exp(-2*x))

def Z_L(w, L):
    return 1j*w*L

def Z_C(w, C):
    return 1/(1j*w*C)

def Z_CPE(w, T_CPE, n_CPE):
    return 1 / (T_CPE * (1j*w)**n_CPE)

def Z_W(w, R_w, T_w):
    return R_w / ((1j*w*T_w)**0.5)

def Z_FLW(w, R_w, T_w):
    return Z_W(w, R_w, T_w) * tanh((1j*w*T_w)**0.5)

def Z_FSW(w, R_w, T_w):
    return Z_W(w, R_w, T_w) * coth((1j*w*T_w)**0.5)

def Z_FLW_Gen(w, R_w, T_w, n_w):
    return R_w * tanh((1j*w*T_w)**n_w) / ((1j*w*T_w)**n_w)

"""
def randles(x, R_e, R_ct, T_CPE, n_CPE, R_w, T_w, L):
    w = 2 * math.pi * x
```

```
    return Z_L(w, L) + R_e + 1 / (1/Z_CPE(w, T_CPE, n_CPE) + 1/(R_ct + Z_W0(w, R_w, T_w)))
"""
# SoC depedence from Wang, Wei and Dai (2019)
def SoC_dependence(x, a, b_1, b_2):
    return a / np.sqrt(x**2 + b_1*x + b_2)

# Alternative Randles circuit, better results perhaps?
def randles(x, R_e, R_ct, T_CPE, n_CPE, R_w, T_w, L):
    w = 2 * math.pi * x
    return Z_L(w, L) + R_e + Z_FLW(w, R_w, T_w) + 1 / (1/Z_CPE(w, T_CPE, n_CPE) + 1/R_ct)

def randles_general_warburg(x, R_e, R_ct, Q_dl, n_dl, R_w, T_w, n_w, L):
    w = 2 * math.pi * x
    return Z_L(w, L) + R_e + Z_FLW_Gen(w, R_w, T_w, n_w) + 1 / (1/Z_CPE(w, Q_dl, n_dl) + 1/R_ct)

def randles_with_film(x, R_e, R_film, R_ct, T_film, T_dl, n, R_w, T_w, L):
    w = 2 * math.pi * x
    return Z_L(w, L) + R_e + Z_FLW(w, R_w, T_w) + 1 / (1/Z_CPE(w, T_film, n) + 1/R_film) + 1 / (1/Z_CPE
(w, T_dl, n) + 1/R_ct)

def simplified_randles_with_film(x, R_e, R_film, R_ct, C_film, C_dl, R_w, T_w, L):
    w = 2 * math.pi * x
    return Z_L(w, L) + R_e + Z_FLW(w, R_w, T_w) + 1 / (1/Z_C(w, C_film) + 1/R_film) + 1 / (1/Z_C(w, C_dl
) + 1/R_ct)

def simplified_randles(x, R_e, R_ct, C_dl, R_w, T_w, L):
    w = 2 * math.pi * x
    return Z_L(w, L) + R_e + 1 / (1/Z_C(w, C_dl) + 1/(R_ct + Z_W(w, R_w, T_w)))

class SoC_fit(lmfit.model.Model):
    __doc__ = "Randles model" + lmfit.models.COMMON_DOC

    def __init__(self, *args, **kwargs):
        super().__init__(SoC_dependence, *args, **kwargs)

    def guess(self, data, x=None, **kwargs):
        verbose = kwargs.pop('verbose', None)
        if x is None:
            return

        a = 4.2458e-4
        b_1 = 0.3303
        b_2 = 0.0338

        if verbose:
            print("a=", a)
            print("b_1=", b_1)
            print("b_2=", b_2)
        params = self.make_params(a=a, b_1=b_1, b_2=b_2)
        return lmfit.models.update_param_vals(params, self.prefix, **kwargs)

    param_full_descs = {'a': 'Fit constant $\alpha$ ($\Omega$)',
                        'b_1': 'Fit constant $\beta_1$',
                        'b_2': 'Fit constant $\beta_2$'}
```

```
class RandlesModel(lmfit.model.Model):
    __doc__ = "Randles model" + lmfit.models.COMMON_DOC

    def __init__(self, *args, **kwargs):
        super().__init__(randles, *args, **kwargs)

    def guess(self, data, x=None, **kwargs):
        verbose = kwargs.pop('verbose', None)
        if x is None:
            return

        R_e = get_intersects(data.real, data.imag)[0]
        R_ct = get_extremum(data.real, data.imag) - R_e
        T_CPE = 75
        n_CPE = 0.8
        R_w = 6e-4
        T_w = 35
        L = 6e-8

        if verbose:
            print("R_e=", R_e, "R_ct=", R_ct)
            print("T_CPE=", T_CPE, "n_CPE=", n_CPE)
            print("R_w=", R_w, "T_w", T_w)
            print("L=", L)
        params = self.make_params(R_e=R_e, R_ct=R_ct, T_CPE=T_CPE, n_CPE=n_CPE, R_w=R_w, T_
w=T_w, L=L)
        params['%sR_e' % self.prefix].set(min=0)
        params['%sR_ct' % self.prefix].set(min=0)
        params['%sT_CPE' % self.prefix].set(min=0)
        params['%sn_CPE' % self.prefix].set(min=0, max=1)
        params['%sR_w' % self.prefix].set(min=0)
        params['%sT_w' % self.prefix].set(min=0)
        params['%sL' % self.prefix].set(min=0)
        return lmfit.models.update_param_vals(params, self.prefix, **kwargs)

    param_full_descs = {'R_e': 'Equivalent series resistance $R_{e}$ ($\Omega$)',
                        'R_ct': 'Charge transfer resistance $R_{ct}$ ($\Omega$)',
                        'T_CPE': 'Double layer phasance $Q_{dl}$ ($\Omega^{-1}s^n$)',
                        'n_CPE': 'Double layer exponent $n$',
                        'R_w': 'Warburg resistance $R_{W}$ ($\Omega$)',
                        'T_w': 'Warburg time constant $T_{W}$ (s)',
                        'L': 'Inductance $L$ ($H$)'}

class RandlesModelGeneralWarburg(lmfit.model.Model):
    __doc__ = "Randles model" + lmfit.models.COMMON_DOC

    def __init__(self, *args, **kwargs):
        super().__init__(randles_general_warburg, *args, **kwargs)

    def guess(self, data, x=None, **kwargs):
        verbose = kwargs.pop('verbose', None)
        if x is None:
            return
        R_e = get_intersects(data.real, data.imag)[0]
        R_ct = get_extremum(data.real, data.imag) - R_e
```

```
        Q_dl = 75
        n_dl = 0.8
        R_w = 6e-4
        T_w = 35
        n_w = 0.5
        L = 6e-8
        if verbose:
            print("R_e=", R_e)
            print("R_ct=", R_ct)
            print("Q_dl=", Q_dl)
            print("n_dl=", n_dl)
            print("R_w=", R_w)
            print("T_w=", T_w)
            print("n_w=", n_w)
            print("L=", L)
        params = self.make_params(R_e=R_e,
                                  R_ct=R_ct,
                                  Q_dl=Q_dl,
                                  n_dl=n_dl,
                                  R_w=R_w,
                                  T_w=T_w,
                                  n_w=n_w,
                                  L=L)
        params['%sR_e' % self.prefix].set(min=0)
        params['%sR_ct' % self.prefix].set(min=0)
        params['%sQ_dl' % self.prefix].set(min=0)
        params['%sn_dl' % self.prefix].set(min=0, max=1)
        params['%sR_w' % self.prefix].set(min=0)
        params['%sT_w' % self.prefix].set(min=0)
        params['%sn_w' % self.prefix].set(min=0, max=1)
        params['%sL' % self.prefix].set(min=0)
        return lmfit.models.update_param_vals(params, self.prefix, **kwargs)

    param_full_descs = {'R_e': 'Equivalent series resistance $R_{e}$ ($\Omega$)',
                        'R_ct': 'Charge transfer resistance $R_{ct}$ ($\Omega$)',
                        'Q_dl': 'Double layer phasance $Q_{dl}$ ($\Omega^{-1}s^n$)',
                        'n_dl': 'Double layer exponent $n_{dl}$',
                        'R_w': 'Warburg resistance $R_{W}$ ($\Omega$)',
                        'T_w': 'Warburg time constant $T_{W}$ (s)',
                        'n_w': 'Warburg exponent $n_{W}$',
                        'L': 'Inductance $L$ ($H$)'}

class RandlesModelWithFilm(lmfit.model.Model):
    __doc__ = "Randles model with film" + lmfit.models.COMMON_DOC

    def __init__(self, *args, **kwargs):
        super().__init__(randles_with_film, *args, **kwargs)

    def guess(self, data, x=None, **kwargs):
        verbose = kwargs.pop('verbose', None)
        if x is None:
            return

        R_e = get_intersects(data.real, data.imag)[0]
        R_ct = get_extremum(data.real, data.imag) - R_e
```

```
        R_film = R_ct
        T_film = 6
        T_dl = 75
        n = 0.8
        R_w = 6e-4
        T_w = 35
        L = 6e-8
        if verbose:
            print("R_e=", R_e)
            print("R_film=", R_film)
            print("R_ct=", R_ct)
            print("T_film=", T_film)
            print("T_dl=", T_dl)
            print("n=", n)
            print("R_w=", R_w)
            print("T_w=", T_w)
            print("L=", L)
        params = self.make_params(R_e=R_e,
                                  R_film=R_film,
                                  R_ct=R_ct,
                                  T_film=T_film,
                                  T_dl=T_dl,
                                  n=n,
                                  R_w=R_w,
                                  T_w=T_w,
                                  L=L)
        params['%sR_e' % self.prefix].set(min=0)
        params['%sR_film' % self.prefix].set(min=0)
        params['%sR_ct' % self.prefix].set(min=0)
        params['%sT_film' % self.prefix].set(min=0)
        params['%sT_dl' % self.prefix].set(min=0)
        params['%sn' % self.prefix].set(min=0, max=1)
        params['%sR_w' % self.prefix].set(min=0)
        params['%sT_w' % self.prefix].set(min=0)
        params['%sL' % self.prefix].set(min=0)
        return lmfit.models.update_param_vals(params, self.prefix, **kwargs)

    param_full_descs = {'R_e': 'Equivalent series resistance $R_{e}$ ($\Omega$)',
                        'R_film': 'Film resistance $R_{film}$ ($\Omega$)',
                        'R_ct': 'Charge transfer resistance $R_{ct}$ ($\Omega$)',
                        'T_film': 'Film phasance $T_{dl}$ ($\Omega^{-1}s^n$)',
                        'T_dl': 'Double-layer phasance $T_{dl}$ ($\Omega^{-1}s^n$)',
                        'n': 'Constant phase element exponent $n$',
                        'R_w': 'Warburg resistance $R_{W}$ ($\Omega$)',
                        'T_w': 'Warburg time constant $T_{W}$ (s)',
                        'L': 'Inductance $L$ ($H$)'}

class SimplifiedRandlesModel(lmfit.model.Model):
    __doc__ = "Simplified randles model" + lmfit.models.COMMON_DOC

    def __init__(self, *args, **kwargs):
        super().__init__(simplified_randles, *args, **kwargs)

    def guess(self, data, x=None, **kwargs):
        verbose = kwargs.pop('verbose', None)
```

```
        if x is None:
            return

        R_e = get_intersects(data.real, data.imag)[0]
        R_ct = get_extremum(data.real, data.imag) - R_e
        C_dl = 75
        R_w = 6e-4
        T_w = 35
        L = 6e-8

        if verbose:
            print("R_e=", R_e, "R_ct=", R_ct)
            print("C_dl=", C_dl)
            print("R_w=", R_w, "T_w", T_w)
            print("L=", L)
        params = self.make_params(R_e=R_e, R_ct=R_ct, C_dl=C_dl, R_w=R_w, T_w=T_w, L=L)
        params['%sR_e' % self.prefix].set(min=0)
        params['%sR_ct' % self.prefix].set(min=0)
        params['%sC_dl' % self.prefix].set(min=0)
        params['%sR_w' % self.prefix].set(min=0)
        params['%sT_w' % self.prefix].set(min=0)
        params['%sL' % self.prefix].set(min=0)
        return lmfit.models.update_param_vals(params, self.prefix, **kwargs)

    param_full_descs = {'R_e': 'Equivalent series resistance $R_{e}$ ($\Omega$)',
                        'R_ct': 'Charge transfer resistance $R_{ct}$ ($\Omega$)',
                        'C_dl': 'Double-layer capacitance $C_{dl}$ (F)',
                        'R_w': 'Warburg resistance $R_{W}$ ($\Omega$)',
                        'T_w': 'Warburg time constant $T_{W}$ (s)',
                        'L': 'Inductance $L$ ($H$)'}

class SimplifiedRandlesModelWithFilm(lmfit.model.Model):
    __doc__ = "Randles model with film" + lmfit.models.COMMON_DOC

    def __init__(self, *args, **kwargs):
        super().__init__(simplified_randles_with_film, *args, **kwargs)

    def guess(self, data, x=None, **kwargs):
        verbose = kwargs.pop('verbose', None)
        if x is None:
            return

        R_e = get_intersects(data.real, data.imag)[0]
        R_ct = (2/3)*(get_extremum(data.real, data.imag) - R_e)
        R_film = 0.5*R_ct
        C_film = 6
        C_dl = 75
        R_w = 6e-4
        T_w = 35
        L = 6e-8
        if verbose:
            print("R_e=", R_e)
            print("R_film=", R_film)
            print("R_ct=", R_ct)
            print("C_film=", C_film)
```

```
            print("C_dl=", C_dl)
            print("R_w=", R_w)
            print("T_w=", T_w)
            print("L=", L)
        params = self.make_params(R_e=R_e,
                                  R_film=R_film,
                                  R_ct=R_ct,
                                  C_film=C_film,
                                  C_dl=C_dl,
                                  R_w=R_w,
                                  T_w=T_w,
                                  L=L)
        params['%sR_e' % self.prefix].set(min=0)
        params['%sR_film' % self.prefix].set(min=0, max=8e-5)
        params['%sR_ct' % self.prefix].set(min=8e-5)
        params['%sC_film' % self.prefix].set(min=0)
        params['%sC_dl' % self.prefix].set(min=0)
        params['%sR_w' % self.prefix].set(min=0)
        params['%sT_w' % self.prefix].set(min=0)
        params['%sL' % self.prefix].set(min=0)
        return lmfit.models.update_param_vals(params, self.prefix, **kwargs)

    param_full_descs = {'R_e': 'Equivalent series resistance $R_{e}$ ($\Omega$)',
                        'R_film': 'Film resistance $R_{film}$ ($\Omega$)',
                        'R_ct': 'Charge transfer resistance $R_{ct}$ ($\Omega$)',
                        'C_film': 'Film capacitance $C_{dl}$ ($F$)',
                        'C_dl': 'Double-layer capacitance $C_{dl}$ ($F$)',
                        'R_w': 'Warburg resistance $R_{W}$ ($\Omega$)',
                        'T_w': 'Warburg time constant $T_{W}$ (s)',
                        'L': 'Inductance $L$ ($H$)'}


def interp(x, x1, x2, y1, y2):
    return (y2-y1)/(x2-x1) * (x-x1) + y1

def get_intersects(x_data, y_data, with_y=0):
    crossings = []
    for i in range(len(y_data)-1):
        if (y_data[i+1] > with_y and y_data[i] < with_y) or (y_data[i] > with_y and y_data[i+1] < with_y):
            crossings.append(interp(with_y, y_data[i], y_data[i+1], x_data[i], x_data[i+1]))
        elif y_data[i] == with_y:
            crossings.append(x_data[i])
    return crossings

def get_extremum(x_data, y_data, after_point=0):
    return np.average(get_intersects(x_data[1:], np.diff(y_data)))

def norm_mean_std(x, mean_x, std_x):
    return (x - mean_x) / np.where(std_x == 0, std_x + 1, std_x)

def norm_min_max(x, min_x, max_x):
    return (x - min_x) / (max_x - min_x)

def normalise_dataset_single(train_x, train_y):
    train_stats = train_x.describe().transpose()
```

```
    train_y_min = train_y.min()
    train_y_max = train_y.max()
    train_x = norm_mean_std(train_x, train_stats['mean'], train_stats['std'])
    train_y = norm_min_max(train_y, train_y_min, train_y_max)

    return train_x, train_y

def normalise_dataset(train_x, train_y, test_x, test_y):
    train_stats = train_x.describe().transpose()
    train_y_min = np.min(train_y.to_numpy())
    train_y_max = np.max(train_y.to_numpy())
    train_x = norm_mean_std(train_x, train_stats['mean'], train_stats['std'])
    train_y = norm_min_max(train_y, train_y_min, train_y_max)
    test_x = norm_mean_std(test_x, train_stats['mean'], train_stats['std'])
    test_y = norm_min_max(test_y, train_y_min, train_y_max)

    return train_x, train_y, test_x, test_y, train_y_min, train_y_max

# Extension with the dot, i.e. '.csv', '.mat', etc.
def get_files_in_folder(folder, ext):
    return [os.path.splitext(filename)[0] for filename in os.listdir(folder) if filename.endswith(ext)]


# == Model predictions ==

def build_model(train_x, train_y, val_x, val_y, activation, first_neurons, hidden_layers,
                learning_rate, shape, regularisation, dropout, verbose=1):
    # Setup model parameter initialisation based on hidden activation function
    kernel_initializer = 'glorot_uniform'; dropout_layer = tf.keras.layers.Dropout(dropout)
    if activation == 'relu' or 'elu': kernel_initializer = 'he_uniform'
    elif activation == 'selu': kernel_initializer = 'lecun_normal'; dropout_layer = tf.keras.layers.AlphaDropout
(dropout)

    # Setup model hidden layer structure
    if shape == 'funnel': neurons = np.round(np.linspace(first_neurons, 1, num=(hidden_layers + 1))).astyp
e(np.int)
    elif shape == 'brick': neurons = first_neurons * np.ones(hidden_layers + 1).astype(np.int)

    model = tf.keras.models.Sequential()
    model.add(tf.keras.layers.Dense(neurons[0], activation=activation, kernel_initializer=kernel_initializer, k
ernel_regularizer=tf.keras.regularizers.l2(regularisation), input_dim=len(train_x.keys()))) # Input + 1st hidd
en layer
    model.add(dropout_layer)
    for i in range(1, len(neurons) - 1):
        model.add(tf.keras.layers.Dense(neurons[i], activation=activation, kernel_initializer=kernel_initializer,
 kernel_regularizer=tf.keras.regularizers.l2(regularisation))) # More hidden layers
        if i < (len(neurons) - 1):
            if activation == 'selu': model.add(tf.keras.layers.AlphaDropout(dropout))
            else: model.add(tf.keras.layers.Dropout(dropout))
    model.add(tf.keras.layers.Dense(1, activation='linear')) # Output layer
    model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate),
              loss='mse',
              metrics=[tf.keras.metrics.MeanAbsoluteError(), tf.keras.metrics.RootMeanSquaredError()])

    # Train model from data
```

```
    history = model.fit(train_x, train_y,
                        epochs=2000,
                        verbose=verbose,
                        validation_data=(val_x, val_y),
                        callbacks=[tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=30)])
    return history, model

def plot_model_history(history):
    hist = pd.DataFrame(history.history) # Get the model history
    fig, ax = plt.subplots(1)
    ax.semilogy(hist.index, hist['mean_absolute_error'], 'b-', label='Training error')
    ax.semilogy(hist.index, hist['val_mean_absolute_error'], 'k--', label='Validation error')
    ax.set_xlabel('Epoch')
    ax.set_ylabel('Mean absolute error')
    ax.legend()
    plt.show()
    fig.savefig("Train_MAE.pdf")

    fig, ax = plt.subplots(1)
    ax.semilogy(hist.index, hist['root_mean_squared_error'], 'b-', label='Training error')
    ax.semilogy(hist.index, hist['val_root_mean_squared_error'], 'k--', label='Validation error')
    ax.set_xlabel('Epoch')
    ax.set_ylabel('Mean squared error')
    ax.legend()
    plt.show()
    fig.savefig("Train_MSE.pdf")

def test_model(x_test, y_test, y_min, y_max, model, plot_predictions=False, verbose=1):
    metrics_list = model.evaluate(x_test, y_test, verbose=verbose) # Evaluate the performance of the mod
el
    test_metrics = {model.metrics_names[i]: metrics_list[i] * (y_max - y_min) for i in range(len(metrics_list))}

    y_pred = model.predict(x_test)
    error = y_pred - y_test.to_numpy()
    test_metrics['peak_error'] = np.max(np.abs(error)) * (y_max - y_min)
    #, subplot_kw={'aspect': 'equal'}

    if plot_predictions:
        fig, ax = plt.subplots(1, figsize=(4.8, 4.8))
        ax.plot(y_test, y_pred, 'b.')
        ax.plot([0,1], [0,1], 'k-')
        ax.set_xlabel('True SoH (normalised)')
        ax.set_ylabel('Predicted SoH (normalised)')
        fig.tight_layout()
        plt.show()
        fig.savefig('Train_Predictions.pdf')

        fig, ax = plt.subplots(1)
        ax.hist(error, bins=20)
        left, right = ax.get_xlim()
        newlim = max([abs(left), abs(right)])
        ax.set_xlim(-newlim, newlim)
        ax.set_xlabel('SoH prediction error (of normalised SoH)')
        ax.set_ylabel('Count')
        fig.tight_layout()
```

```
        plt.show()
        fig.savefig('Train_Predictions_Hist.pdf')

    return test_metrics

# Run k-fold cross validation on a model. **kwargs are the extra arguments to build_model.
def k_fold_cv(data_x, data_y, folds=5, shuffle=True, **kwargs):
    metric_list = []
    kf = KFold(n_splits=folds, shuffle=shuffle)
    for train_indices, test_indices in kf.split(data_x, data_y):
        train_x = data_x.iloc[train_indices]
        train_y = data_y.iloc[train_indices]
        test_x = data_x.iloc[test_indices]
        test_y = data_y.iloc[test_indices]
        train_x, train_y, test_x, test_y, train_y_min, train_y_max = normalise_dataset(train_x, train_y, test_x,
 test_y)

        history, model = build_model(train_x, train_y, test_x, test_y, **kwargs)
        metric_list.append(test_model(test_x, test_y, train_y_min, train_y_max, model, plot_predictions=Fals
e, verbose=0)['root_mean_squared_error'])
    return np.average(metric_list)

# This is a separate function to evaluate k-fold CV as need a function with the named args annotation and
 to clear
# the Keras session after every evalation to avoid memory leaks.
@skopt.utils.use_named_args(HYPEROPT_SEARCH_SPACE)
def objective(activation, first_neurons, hidden_layers, learning_rate, shape, regularisation, dropout):
    metric = k_fold_cv(inner_x, inner_y, folds=MODEL_CV_K - 1, shuffle=True,
                       activation=activation,
                       first_neurons=first_neurons,
                       hidden_layers=hidden_layers,
                       learning_rate=learning_rate,
                       shape=shape,
                       regularisation=regularisation,
                       dropout=dropout,
                       verbose=0)
    tf.keras.backend.clear_session() # Needed because tensorflow will otherwise not free memory from old
 models.
    return metric

# Fitting / loading of data
ECM_model = RandlesModel()
rcParams['font.family'] = 'sans-serif'
rcParams['font.sans-serif'] = ['Arial']
plt.rcParams.update({'font.size': 12})

"""
# Fit and display one particular EIS measurement
EIS_data = {}
files = get_files_in_folder(NEWCASTLE_DATA_DIR, NEWCASTLE_DATA_EXT)
for file in files:
    EIS_data[file] = pd.read_excel(
        os.path.join(NEWCASTLE_DATA_DIR, file + NEWCASTLE_DATA_EXT),
        sheet_name=None,
        header=1)
```

```
fig, ax = plt.subplots(1)
data_field = EIS_data['MD25_RW_EIS_SOC']['60%']
f = np.geomspace(0.015, 1000, num=data_field['Re(Z)/Ohm'].size)[::-1][0:19]
Z = (data_field['Re(Z)/Ohm'] - 1j*data_field['-Im(Z)/Ohm']).to_numpy()[0:19]
fit, model_fit_params = fit_model(f, Z, ECM_model, fit_name='MD25_RW_EIS_SOC', fit_weights=1/np.abs(Z))
Z_eval = ECM_model.eval(params=fit.params, x=f)
color = next(ax._get_lines.prop_cycler)['color']
ax.plot(Z.real, -Z.imag, linestyle='', marker='.', color='b', label='Measured (60% SoC)')
ax.plot(Z_eval.real, -Z_eval.imag, linestyle='-', color='k', label='ECM fit (60% SoC)')
#ax.semilogx(f, -np.angle(Z - model_fit_params['R_e'], deg=True), linestyle='', marker='.', color='b', label='Measured (80% SoC)')
#ax.semilogx(f, -np.angle(Z_eval - model_fit_params['R_e'], deg=True), linestyle='-', color='k', label='ECM fit (80% SoC)')
#ax.semilogx(f, -(Z.imag - Z_eval.imag), linestyle='-', marker='.', color='b', label='Residual')
#ax.set_xlim(ax.get_xlim()[::-1])
ax.set_xlabel('Re{$Z$} ($\Omega$)')
ax.set_ylabel('-Im{$Z$} ($\Omega$)')
ax.legend()
plt.setp(ax.get_xticklabels(), rotation=30, ha="right", rotation_mode="anchor")
fig.tight_layout()
plt.show()
"""

if not os.path.isfile(NEWCASTLE_DATA_FITTED) or RUN_POPULATE_DATASET:
    EIS_data = {}
    files = get_files_in_folder(NEWCASTLE_DATA_DIR, NEWCASTLE_DATA_EXT)
    for file in files:
        EIS_data[file] = pd.read_excel(
            os.path.join(NEWCASTLE_DATA_DIR, file + NEWCASTLE_DATA_EXT),
            sheet_name=None,
            header=1)

    ECM_data = []
    for key, file_data in EIS_data.items(): # For each set of 2 cells
        fig, ax = plt.subplots(1)
        for key2, data_field in file_data.items(): # For each measurement at each SoC
            f = np.geomspace(0.015, 1000, num=data_field['Re(Z)/Ohm'].size)[::-1]#[0:19]
            Z = (data_field['Re(Z)/Ohm'] - 1j*data_field['-Im(Z)/Ohm']).to_numpy()#[0:19] # Note this is the actual impedance, not impedance expressed with Im{Z} = -Im{Z}

            fit, model_fit_params = fit_model(f, Z, ECM_model, fit_name=key, fit_weights=1/np.abs(Z))
            model_fit_params['SoC'] = float(key2.strip('%')) / 100
            ECM_data.append(model_fit_params)

            Z_eval = ECM_model.eval(params=fit.params, x=f)
            color = next(ax._get_lines.prop_cycler)['color']
            ax.plot(Z.real, -Z.imag, linestyle='', marker='.', color=color, label=('Measured (' + key2 + ' SoC)'))
            ax.plot(Z_eval.real, -Z_eval.imag, linestyle='-', color=color, label=('ECM fit (' + key2 + ' SoC)'))

        ax.set_xlabel('Re{Z} ($\Omega$)')
        ax.set_ylabel('-Im{Z} ($\Omega$)')
        ax.legend()
        plt.show()
```

```
        fig.savefig(os.path.join(OUTPUT_DIR, key) + '.pdf')

    dataset = pd.read_excel(NEWCASTLE_DATA_CAPACITIES).dropna()
    dataset['SoH'] = dataset['Capacity/Ah'] / CELL_CAPACITY
    dataset = dataset.merge(pd.DataFrame(ECM_data), on='Name')
    dataset.to_csv(NEWCASTLE_DATA_FITTED)

else: dataset = pd.read_csv(NEWCASTLE_DATA_FITTED, index_col=0)

g_cols_x = ECM_model.param_names + ['SoC']
g_cols_y = ['SoH']
g_cols_err = [name + '_err' for name in ECM_model.param_names]
dataset = dataset.drop(columns=['Name', 'Capacity/Ah', 'Resistance/mOhm', 'Chisqr']) # Drop the cruft

# Running neural network / RF / GP regression models
if RUN_MODEL:
    if MODEL_EXCLUDED_PARAMS is not None: g_cols_x = [element for element in g_cols_x if element
not in MODEL_EXCLUDED_PARAMS]
    if MODEL_TYPE == 'NN':
        cv_metrics = []
        kf_outer = KFold(n_splits=MODEL_CV_K, shuffle=True)
        count = 0
        times = []
        for train_indices, test_indices in kf_outer.split(dataset[g_cols_x], dataset[g_cols_y]):
            if MODEL_CV_RUNS != None:
                if count >= MODEL_CV_RUNS: break
            meas_time = time.time()
            inner_x = dataset.iloc[train_indices][g_cols_x]
            inner_y = dataset.iloc[inner_x.index][g_cols_y]
            test_x = dataset.iloc[test_indices][g_cols_x]
            test_y = dataset.iloc[test_indices][g_cols_y]

            inner_x, inner_y, test_x, test_y, train_y_min, train_y_max = normalise_dataset(inner_x, inner_y, te
st_x, test_y)
            history, model = build_model(inner_x, inner_y, test_x, test_y,
                        activation=MODEL_TRIAL_CONFIGS[MODEL_NO][0],
                        first_neurons=MODEL_TRIAL_CONFIGS[MODEL_NO][1],
                        hidden_layers=MODEL_TRIAL_CONFIGS[MODEL_NO][2],
                        learning_rate=MODEL_TRIAL_CONFIGS[MODEL_NO][3],
                        shape=MODEL_TRIAL_CONFIGS[MODEL_NO][4],
                        regularisation=MODEL_TRIAL_CONFIGS[MODEL_NO][5],
                        dropout=MODEL_TRIAL_CONFIGS[MODEL_NO][6],
                        verbose=0)
            #plot_model_history(history)
            #cv_metrics.append(test_model(test_x, test_y, train_y_min, train_y_max, model, plot_predictions
=False))
            meas_time = time.time() - meas_time
            times.append(meas_time)
            count += 1
        tf.keras.backend.clear_session()
        #test_metrics = pd.DataFrame(cv_metrics)
        #test_metrics_stats = test_metrics.describe()

    elif MODEL_TYPE == 'RF':
        rf_model = ExtraTreesRegressor(criterion='mse')
```

```
        scores = cross_validate(rf_model, dataset[g_cols_x], dataset[g_cols_y],
                                scoring=['neg_root_mean_squared_error', 'neg_mean_absolute_error', 'max_error'],
                                cv=MODEL_CV_K, n_jobs=-1)
        test_metrics = np.abs(pd.DataFrame(scores))

        test_data = dataset.sample(n=21)
        train_data = dataset.drop(index=test_data.index)
        test_data['SoH'] = norm_min_max(test_data['SoH'], np.min(train_data['SoH']), np.max(train_data['So
H']))
        train_data['SoH'] = norm_min_max(train_data['SoH'], np.min(train_data['SoH']), np.max(train_data['S
oH']))

        pred_y = rf_model.fit(train_data[g_cols_x], train_data[g_cols_y].to_numpy().ravel()).predict(test_data
[g_cols_x])
        fig, ax = plt.subplots(1, subplot_kw={'aspect': 'equal'})
        ax.plot(test_data[g_cols_y], pred_y, 'b.', label='Predictions')
        ax.plot([0,1], [0,1], 'k-')
        ax.set_xlabel('True SoH (normalised)')
        ax.set_ylabel('Predicted SoH (normalised)')
        ax.legend()
        plt.show()
        fig.savefig("Train_Predictions_RF")
    elif MODEL_TYPE == 'GP':
        data_x, data_y = normalise_dataset_single(dataset[g_cols_x], dataset[g_cols_y])
        gp_model = GaussianProcessRegressor(kernel=kernels.ConstantKernel(constant_value=1.0) * kern
els.RBF(1.0) + kernels.ConstantKernel(constant_value=1.0))
        scores = cross_validate(gp_model, data_x, data_y,
                                scoring=['neg_root_mean_squared_error', 'neg_mean_absolute_error', 'max_error'],
                                cv=MODEL_CV_K, n_jobs=-1)
        test_metrics = np.abs(pd.DataFrame(scores))

        """
        fig, ax = plt.subplots(1)
        ax.errorbar(output['Samples'], output['RMSE'], yerr=output['RMSE_Err'], linestyle='-', marker='.', colo
r='b')
        ax.set_xlabel('Training samples')
        ax.set_ylabel('RMS SoH prediction error (%SoH)')
        ax.set_xscale('log')
        ax.set_yscale('log')
        ax.legend()
        plt.show()
        fig.tight_layout()
        """


if RUN_HYPEROPT:
    kf_outer = KFold(n_splits=MODEL_CV_K, shuffle=True)
    count = 0
    hyperopt_results = []
    cv_metrics = []
    for train_indices, test_indices in kf_outer.split(dataset[g_cols_x], dataset[g_cols_y]):
        if MODEL_CV_RUNS != None:
            if count >= MODEL_CV_RUNS: break
        inner_x = dataset.iloc[train_indices][g_cols_x]
        inner_y = dataset.iloc[inner_x.index][g_cols_y]
```

```
        test_x = dataset.iloc[test_indices][g_cols_x]
        test_y = dataset.iloc[test_indices][g_cols_y]

        hyperopt_results.append(skopt.gp_minimize(objective, HYPEROPT_SEARCH_SPACE, n_calls=500
, n_random_starts=250, model_queue_size=1, verbose=True))

        inner_x, inner_y, test_x, test_y, train_y_min, train_y_max = normalise_dataset(inner_x, inner_y, test
_x, test_y)
        history, model = build_model(inner_x, inner_y, test_x, test_y,
                                     activation=hyperopt_results[count].x[0],
                                     first_neurons=hyperopt_results[count].x[1],
                                     hidden_layers=hyperopt_results[count].x[2],
                                     learning_rate=hyperopt_results[count].x[3],
                                     shape=hyperopt_results[count].x[4],
                                     regularisation=hyperopt_results[count].x[5],
                                     dropout=hyperopt_results[count].x[6],
                                     verbose=0)
        cv_metrics.append(test_model(test_x, test_y, train_y_min, train_y_max, model, plot_predictions=Fal
se))
        count += 1
        tf.keras.backend.clear_session()

if RUN_PLOT_MODEL_TRAIN_GRAPH:
    if MODEL_EXCLUDED_PARAMS is not None: g_cols_x = [element for element in g_cols_x if element
not in MODEL_EXCLUDED_PARAMS]
    train_data_range = np.linspace(2, int(len(dataset) - len(dataset)/MODEL_CV_K), num=20).astype(np.in
t)
    train_data_metrics = []
    train_data_errors = []
    for num_train_samples in train_data_range:
        cv_metrics = []
        for i in range(10):
            kf_outer = KFold(n_splits=MODEL_CV_K, shuffle=True)
            count = 0
            for train_indices, test_indices in kf_outer.split(dataset[g_cols_x], dataset[g_cols_y]):
                if MODEL_CV_RUNS != None:
                    if count >= MODEL_CV_RUNS: break
                inner_x = dataset.iloc[train_indices][g_cols_x]
                inner_x = inner_x.sample(n=num_train_samples)
                inner_y = dataset.iloc[inner_x.index][g_cols_y]
                test_x = dataset.iloc[test_indices][g_cols_x]
                test_y = dataset.iloc[test_indices][g_cols_y]

                inner_x, inner_y, test_x, test_y, train_y_min, train_y_max = normalise_dataset(inner_x, inner_y,
 test_x, test_y)
                history, model = build_model(inner_x, inner_y, test_x, test_y,
                                             activation=MODEL_TRIAL_CONFIGS[MODEL_NO][0],
                                             first_neurons=MODEL_TRIAL_CONFIGS[MODEL_NO][1],
                                             hidden_layers=MODEL_TRIAL_CONFIGS[MODEL_NO][2],
                                             learning_rate=MODEL_TRIAL_CONFIGS[MODEL_NO][3],
                                             shape=MODEL_TRIAL_CONFIGS[MODEL_NO][4],
                                             regularisation=MODEL_TRIAL_CONFIGS[MODEL_NO][5],
                                             dropout=MODEL_TRIAL_CONFIGS[MODEL_NO][6],
                                             verbose=0)
                cv_metrics.append(test_model(test_x, test_y, train_y_min, train_y_max, model, plot_prediction
```

```
s=False))
            count += 1
            tf.keras.backend.clear_session()
        print(i)
    test_metrics = pd.DataFrame(cv_metrics)
    test_metrics_stats = test_metrics.describe()
    train_data_metrics.append(test_metrics_stats['root_mean_squared_error']['mean'])
    train_data_errors.append(test_metrics_stats['root_mean_squared_error']['std'])

output = pd.DataFrame({'Samples': train_data_range, 'RMSE': np.array(train_data_metrics)*100, 'RMS
E_Err': np.array(train_data_errors)*100/np.sqrt(60)})
fig, ax = plt.subplots(1)
ax.errorbar(output['Samples'], output['RMSE'], yerr=output['RMSE_Err'], linestyle='-', marker='.', color='
b')
ax.set_xlabel('Training samples')
ax.set_ylabel('RMS SoH prediction error (%SoH)')
ax.set_xscale('log')
ax.set_yscale('log')
ax.legend()
plt.show()
#fig.savefig('Sample_Trend_Comparison')

# Plotting correlation matrices of the dataset variables
if RUN_PLOT_CORRELATIONS:
    grouped_dataset = dataset.groupby('SoC')
    corr_cols = ['SoH']
    corr_cols.extend(g_cols_x)
    corr_cols.remove('SoC')
    avg_cmatrix = grouped_dataset.get_group(list(grouped_dataset.groups.keys())[0]).drop(columns=['SoC
'])[corr_cols].corr() * 0
    model_fits = []

    for group in grouped_dataset.groups.keys(): # For each SoC
        SoC_data = grouped_dataset.get_group(group).drop(columns=['SoC'])
        for param in corr_cols: # For each input param do a linear fit between that param and the SoH
            if param != 'SoH':
                model = lmfit.models.LinearModel()
                fit_name = 'Correlation_{}'.format(int(group * 100)) + '%_SoC_' + param
                if np.any(SoC_data[param + '_err'] == 0): fit_weights = None
                else: fit_weights = 1 / SoC_data[param + '_err']
                fit, model_fit_params = fit_model(SoC_data['SoH'], SoC_data[param], model, fit_name, fit_weig
hts=fit_weights)
                model_fit_params['SoC'] = group
                model_fits.append(model_fit_params)
                plot_fit(SoC_data['SoH'], SoC_data[param], fit,
                        data_label='Measured',
                        fit_label='Linear fit\n$\chi^2$/DoF = {0:.2E}\nm = {1:.2E}$\pm${2:.2E}\nc = {3:.2E}$\pm${4
:.2E}'.format(
                                model_fit_params['Chisqr']/SoC_data['SoH'].count(),
                                model_fit_params['slope'],
                                model_fit_params['slope_err'],
                                model_fit_params['intercept'],
                                model_fit_params['intercept_err']),
                        y_label=ECM_model.param_full_descs[param],
                        x_label='State of health',
```

```
                fig_name=fit_name,
                y_err=SoC_data[param + '_err'])

    cmatrix = SoC_data[corr_cols].corr() # Also get the correlation matrix between all dataset variables
    plot_cmatrix(cmatrix, 'Correlations_{}'.format(int(group * 100)) + '%_SoC')
    avg_cmatrix += cmatrix # And the average correlation matrix
avg_cmatrix = avg_cmatrix / len(grouped_dataset.groups.keys())
plot_cmatrix(avg_cmatrix, 'Correlations_Average')
```