Applications of Data Science to Electrochemistry

Mobasher Khan, '18

Submitted to the
Department of Mechanical and Aerospace Engineering
Princeton University
in partial fulfillment of the requirements of
Undergraduate Independent Work.

Final Report

May 2, 2018

Daniel A. Steingart
My Reader
MAE 442: IW
159 pages
Adviser Copy

This thesis represents my own work in accordance with University regulations.

Abstract

Electrochemical Energy storage is a promising, yet deceptively complex method of powering our world in a sustainable manner. The intricacies of the chemical reactions, kinetics, diffusion, and transport mechanisms involved have made it an area of research where there is a lot to discover and understand about creating powerful and efficient batteries. As the need for moving on from traditional energy sources becomes more pressing, so does the demand for higher performance batteries as replacements.

Discerning which combinations of materials and their properties lead to which characteristics is difficult given all the mechanical and electrochemical factors involved, but the growth of data science and statistics as a tool for all disciplines hopes to shed some more light on this problem from a different perspective. Some performance characteristics such as rate retention, capacity retention, and coulombic efficiency can characterize a battery's performance, but these values are not known until after extensive testing. The goal of this project is to use machine learning algorithms and other data analysis tools to contribute to this ongoing effort of relating different battery chemistries directly to their properties without costly experimentation and testing. Some of the possible applications of machine learning to this field include estimating state of charge, predicting performance characteristics, and identifying different regimes and electrochemical behaviors from battery cycling data.

Acknowledgements

Thank you to everyone who has helped me with my thesis whether it was offering support through this stressful time or offering guidance. I want to give a special thanks to my mother who helped me get to where I am and taught me what hard work and perseverance truly looks like, to my advisor Professor Steingart who inspired me with his boundless knowledge, to Clem who was patient with me and helped me find data sets and understand them, and lastly to my friend Shamailah who listened to all my complaints and brought me food whenever I was too caught up in my work or too tired to leave my room.

I would like to dedicate my thesis to the pursuit of knowledge and education. My hopes of becoming a derivatives trader even though I spent the last five years learning mechanical engineering are a testament to the limitlessness of the human mind.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivations, Battery Background, & Goals

As batteries are becoming a serious contender for replacing more traditional energy sources, a lot of investment and research has gone into finding the aspects that would make the cheapest, most energy/power dense, and long lasting batteries. Although a lot of the interactions on a molecular level are well known, the complexities in modeling and predicting all the electrochemical and mechanical processes occurring has led researchers to try other methods of finding the relationships between battery chemistries and performance.

Machine learning has begun to be incorporated into predictions regarding state of health (SOH) and state of charge (SOC). SOH is an indicator of how a battery has deteriorated and is a ratio between the maximum capacity attainable by the battery in its current state to the capacity it was rated for. Over time, as a battery degrades, its maximum capacity decreases due to losses from side reactions, irreversibilities, mechanical degradation, amongst other effects, meaning that the SOH also decreases over time. SOC is a more short-term indication of the current capacity of the battery compared to the maximum possible it can achieve. This value informs us of when the battery charge or discharge should be stopped and also acts as a fuel gauge for a battery.

Although several previous papers have attempted to predict SOC using machine learning methods, this has been done to various degrees of accuracy and also with results that, at times, seem too good to be true. Being able to calculate SOC without using more

costly management techniques is definitely a sought after result in batteries research, and I wanted to see what the capabilities of machine learning models were to this end.

Another set of performance parameters for given electrode chemistries were capacity retention, rate retention, and coulombic efficiency at the first cycle. Capacity retention indicated how much of the rated capacity a battery could hold after long term cycling, rate retention indicated how capacity attainable in a battery changed as current rates were altered, and coulombic efficiency was a measurement of how much charge was harnessed on discharged as compared to the amount transferred during charge. For various sets of anodes, these performance characteristics had been tabulated along with the anode characteristics such as the chemical elements used and the anode composition in Bommier's paper [34]. Finding direct relationships between these features and the performance was difficult due to the mentioned complexities of simulating a battery's performance, so I hoped to use data analysis on this dataset to predict performance parameters and find the relationship between the input features and the corresponding performance.

Data science techniques, such as training regressors to predict quantities, have had applications in many fields, with the only large cost of applying these techniques being computational power and the necessity of gathering data. Although these techniques have been used to supplement battery research, I wanted to assess the power and true potential of machine learning in electrochemical applications. By using datasets that had already been collected, I hoped to alter model parameters and model inputs in order to see whether strong, repeatable performance could be squeezed out of otherwise unassuming data.

A final application of data science to techniques to batteries was creating data visualizations that would help quickly find trends and 'events' in battery cycling data (discussed later in this chapter). Visualizing the decay of capacity as the number of cycles increased was a standard plot, but the real indicator of the underlying mechanisms of capacity decay was found in the plots of the differential capacity curves per cycle: dQ/dV vs. Voltage. These curves, when integrated, showed the charge transferred during charge or discharge, and also indicated the amount of charge transferred at different voltage levels. Since the battery voltage was indicative of the time passed during charge and discharge, and since degradation mechanisms changed as voltage, understanding what was occurring in these plots would help characterize capacity decay [66]. Using cycling data, my goal was to create interactive data visualizations that would quickly show the differential capacity

curves during a specific cycle so that the battery behavior could be analyzed as cycle number changed.

1.2 Data Science in Previous Battery Research

As mentioned previously, the two main areas of batteries research that have used machine learning or other data analysis techniques are for estimating SOH and SOC.

1.2.1 SOH Estimation

SOH decrease can be caused by a variety of factors. Ageing mechanisms occur due to different processes and their interactions, and they are all on similar timescales so the ageing mechanisms are hard to analyze. Two different approaches exist to estimating SOH: experimental and adaptive methods. But both rely on knowledge about the battery's capacity and internal resistance. [31] [56] [40] [62] [30]

- Experimental techniques: Stores cycling data history of the battery and uses chemical intuition about the influence of parameters affecting battery life to estimate SOH. This requires knowledge on how operation and degradation of the battery are related through physical/chemical analysis or by analyzing SOH test data sets.
  - Direct Measurements: estimate SOH from impedance/resistance
    * Use Ohm's Law, R = I/V, with current pulses I and voltage drop V, to calculate resistance at some arbitrary time when the current is pulsed through. Then resistance is calculated based on temperature/SOC. But in general it was noted in the research that SOC and resistance growth are not very related; temperature had a very big effect on resistance growth.
    * Create precise models by noting relationship between voltage characteristics and internal resistance, and then using voltage measurements to calculate internal resistance
    * Using Electrochemical Impedance Spectroscopy to measure impedance. Uses different frequencies to glean different types of information about the battery. Also has problems that it cannot be used for different battery types without major adjustments.
  - Models based on measurements
    * Data fitting by calculating internal resistance at every SOC and temperature by creating a data map

    * Coulomb Counting: count the amp-hours/capacity being discharged to calculate remaining capacity and then use SOH = $\frac{Q_{max}}{Q_{rated}}$. This is the most widely used since current rate or temperature have few effects on this method's accuracy
    * Support Vector Regression: nonlinear algorithm to recognize patterns. Needs to use a particle filter to remove noise and requires a lot of data, but this gives accurate estimates of SOH based on loadings or other input data
    * Measuring mechanical fatigue can also be used to calculate SOH. This requires physics/chemistry based models of the battery so that SEI growth, cell fatigue, and internal cell structure can be tracked so that impedance growth and capacity loss as the battery is cycled can be measured.

- Adaptive techniques: calculates SOH from measured parameters that are known to be sensitive to battery degradation as the battery is in operation. Needs less previous SOH testing and is more adaptable to new chemistries, but has a higher computing load.

  – Kalman filter: Use a prediction state to estimate output variables. Requires recursive equations so that system state is a function of past states. Accurate and efficient but requires large amount of Matrix operations. Usually uses internal resistance to calculate SOH. Since KF are linear models, there are also models that calculate SOC using nonlinear methods, then use the IR and capacity in the KF to calculate SOH. There are also KF that can predict Internal Resistance and capacity independently using a equivalent circuit model.
  – Artificial Neural Networks: Uses equivalent circuit of battery along with a lot of data in order to create predictions for SOH, SOC, and other parameters using NNs.
  – Least squares Regression: Uses an electrochemical model based on IR and solid phase diffusion time. Then uses least square regression based on voltage and current data to estimate parameters like SOH. Other models require voltage, current, and temperature as inputs to model IR and therefore SOH.

1.2.2 SOC Estimation

As a battery is cycled there are non-linear, time-varying characteristics and electrochemical reactions, along with effects due to cycling, temperature, and age, so calculating SOC

is challenging. SOC is usually estimated using voltage, current, and temperature, similar to the measurements taken to calculate Internal Resistance, impedance, and capacity for estimating SOH.

SOC is usually measured as $1 - \frac{Q_{avail}}{Q_{nominal}}$, where the available capacity is calculated by integrating the current over the time. Since the nominal capacity is not constant and changes due to the internal reactions in the battery, the SOC calculation is not linear. There are some conventional methods for calculating SOC as well as algorithmic methods that have been introduced due to the influence of data science. [44] [84] [73] [56] [52] [86] [39] [37]

- Conventional Methods
  - Li-Ion batteries use intercalation where the Open Circuit Voltage changes as the battery discharges, so OCV and SOC can be related as a mapping. So when battery OCV is known, SOC can be estimated. Since this relationship also depends on other factors such as capacity and material of the battery, it is not applicable to other systems without major adjustments.
  - Coulomb counting: Measures the current over time (so it needs accurate current sensors) and integrates in order to calculate SOC. There are inaccuracies due to noise, disturbances, temperature, and other variables. Another issue is that full discharge will be required in order to calculate the maximum/nominal capacity of the battery.
  - Internal Resistance calculation: Measures voltage and current over small timescales to avoid diffusion and ohmic resistance effects and only capture the IR of the battery. SOC's relationship with IR is used in order to try and calculate SOC when the IR is known
  - Electrochemical models that use parameters of the battery (including the effects of thermodynamics, kinetics, side reactions, transport, etc) in order to calculate OCV accurately, and then OCV-SOC lookup tables are used to calculate the SOC
- Machine Learning Algorithms
  - Neural Networks: Uses a large amount of training data, but takes discharge current, terminal voltage, and temperature as inputs and outputs SOC

- Fuzzy logic: Needs a training set, from which classification and rules are created for predicting outputs based on inputs. This works for non-linear systems, but requires a lot of computation
- Support Vector Machines convert non-linear problems to linear problems in higher dimensions by using kernel functions. There are many parameters to adjust for predicting SOC, and the amount of computation is also very high. In general, inputs of voltage, current and temperature are used, and the output is the SOC

1.3 Data Format & Description

1.3.1 Cycling Data

In this project, the main type of data that was analyzed was battery cycling data. Cycling data consists of charging and discharging the battery repeatedly, up to any number of cycles, while recording pertinent observations such as voltage, current, charge passed, energy passed, and the type of event that was occurring during a given time. The particular battery used in this analysis was a 1500 mAh Samsung battery.

Analyzing voltage, current, and capacity transferred as cycles increase helps discern certain battery behaviors such as maximum capacity degradation, capacitive behaviors at higher current rates, and side reactions due to time spent at certain voltages. This data can be further analyzed to calculate SOH and SOC, and different charging regimes, such as constant current vs constant current-constant voltage, can also be compared.

The following is an example of cycling data input into Python and displayed as a DataFrame, a special Python object that will be discussed in the next section.

	Unnamed: 0	unix_time	test_vol	test_cur	cycle	step_id	step_type	test_capchg	test_capdchg	test_engchg	test_engdchg	test_tmp
0	0	1.480620e+09	3.3035	0.0	0.0	1.0	4.0	0.0	0.0	0.0	0.0	0.0
1	1	1.480620e+09	3.3023	0.0	0.0	1.0	4.0	0.0	0.0	0.0	0.0	0.0
2	2	1.480620e+09	3.3023	0.0	0.0	1.0	4.0	0.0	0.0	0.0	0.0	0.0
3	3	1.480620e+09	3.3023	0.0	0.0	1.0	4.0	0.0	0.0	0.0	0.0	0.0
4	4	1.480620e+09	3.3023	0.0	0.0	1.0	4.0	0.0	0.0	0.0	0.0	0.0

Table 1.1: Cycling Data Format

The *Unnamed:0* column here has no significance as it is just a recording of the ob-

servation/row number. The *unix_time* column is the time at which the observation was recorded, with respect to some arbitrary start time. Since the start time of the experiment is not significant, only the time differences between observations has any meaning. The *test_engchg* and *test_engdchg* columns are measures of the energy passed, but are not used in this project.

The *cycle* column represents the cycle number of the observation. In general, the 0th cycle is a unique cycle, where the battery has been charged/discharged to some unknown state by the manufacturer, and therefore the battery does not go through a charging cycle similar to that during proceeding cycles. The *test_tmp* column represents the temperature at which an observation was made, but for our data, the temperature was controlled in order to give precedence to other variables and quantities of interest. In general, temperature can have a strong effect on battery performance, as chemical behaviors of the battery components are influenced by the temperature of operation.

The *test_vol* column is the voltage across the battery terminals during the time the observation was made, in units of Volts (V). The voltage, which is typically the variable that is allowed to vary and observed, can also be held at a constant value. The following is an example of constant voltage charging in a typical power supply. For a given maximum current and constant voltage value desired, by making sure the impedance of the load connected to the power supply is over a certain value, $Z_{minimum} = \frac{V_{constant}}{I_{maximum}}$, ensures that the current drawn will be at most the maximum supplied, and therefore the current can vary to keep the voltage constant for impedances above the minimum. This is reflected in the cycling data where once the maximum voltage value, typically $4.2V$, is reached, the current is then allowed to vary in order to keep the battery charging at the constant voltage.

The *test_cur* column is the current supplied to the battery at the time an observation is made, in units of Amperes (A). The current supplied is often the independent variable set by the experimenter, as there is interest in observing changing battery behaviors at different currents. A negative value in this column represents a state of discharge, while a positive value represents a state of charge. This is according to the convention that positive fluxes bring energy/matter into the battery system.

The *test_capchg* and *test_capdchg* columns are a measure of the charge passed, in units of Ampere-hours (Ah) for a charge step and a discharge step, respectively. These columns

are a cumulative total of the charge passed per charge or discharge step, so the value at the end of a step is the amount of charge passed during the step, after which the running total of charge passed resets to 0.

The *step_id* and *step_type* columns indicate the type of event occurring during the time an observation is made. Of interest in this project, are the step id and step type combinations: $(2, 1), (2, 7)$, and $(4, 2)$. A $(2, 1)$ combination indicates a constant current charge step up to the maximum voltage. A $(2, 7)$ combination indicates a constant current charge to the maximum voltage followed by a constant voltage charge until the current tapers off to a minimum value. A $(4, 2)$ combination indicates a constant current discharge until the battery has reached a certain minimum voltage threshold. There are other events in the data not considered here, such as the relaxation steps in between charge and discharge steps.

In addition to the above columns, an additional column, *run*, was added to indicate the type of charging that had been performed for a given dataset. A dataset that utilized constant current charging at a $1C$ rate was given the value *cc*, a dataset that utilized constant current charge at a $C/10$ rate was given the value *c10*, and a dataset that utilized constant current-constant voltage charging was given the value *cccv*. The different types of charging lead to variations in battery behavior, although they may seem mathematically equivalent otherwise. For example, constant current charging at lower C rates leads to higher capacity/amount of charge transferred, while constant current charging at higher C rates leads to lower charge transferred and therefore lower SOC for a given amount of charge transferred. In the case of constant current-constant voltage charging, maintaining a battery at high voltage while also at a high current leads to enhanced degradation that is imperative to avoid. [85]

1.3.2 Sodium Ion Battery Anode Performance Data

The second type of dataset analyzed in this project was from Bommier's paper on electrochemical performance relations in sodium ion battery (NIB) anodes [34]. The data was meticulously collected from hundreds of papers and was meant to be a collection of results from all corners of research about how different classes of materials perform as anodes in Sodium Ion Batteries. To this end, various performance characteristics and data about the battery formulation were recorded in each observation.

The following is a sample of some rows of this dataset with the columns unnecessary for analysis removed.

	class	material	category_2	current_rate (mA g-1)	capacity (mAh g-1)	rate_retention	CE_first_cycle	loading_mass (mg/cm2)	cycles	current_long_cycling
655	CNF	C	N	100.0	167.0	1.0	0.329	1.20	100.0	500.0
943	Alloy	Sb	Zn	41.4	355.0	1.0	0.45	1.00	200.0	414.0
966	Alloy	Sb	Fe	100.0	200.0	1.0	0.6	1.20	100.0	100.0
971	Alloy	Sb	Fe	18.0	600.0	1.0	0.85	1.50	140.0	300.0
1024	Alloy	Sn	Cu	100.0	150.0	1.0	0.56	1.25	100.0	100.0

Table 1.2: First set of columns of Anode data

	capacity_at_end	capacity_retention	lower voltage	upper voltage	synthesis	carbon_support
655	120.0	0.99	0.01	3.0	dicyandiamide	Nitrogen-doped bamboo-like carbon nanotubes: p...
943	285.0	0.98	0.01	2.0	CVD	carbon black
966	200.0	0.99	0.00	2.0	ball-mill	carbon black
971	440.0	0.99	0.01	1.5	ball-mill	carbon black
1024	150.0	0.95	0.00	2.0	ball-mill	carbon black

Table 1.3: Second set of columns of Anode data

	electrolyte_salt	electrolyte_solvent	FEC_addition	binder_type	electrode_composition
655	NaClO4	EC:DEC	FEC-02	PVDF	80:10:10
943	NaClO4	PC	FEC-05	PVDF	70:15:15
966	NaClO4	EC:PC	FEC-02	PVDF	75:15:15
971	NaClO4	PC	FEC-05	CMC	70:12:18
1024	NaClO4	EC:PC	FEC-02	PVDF	70:15:15

Table 1.4: Second set of columns of Anode data

Within the data, two types of experiments were mixed together, long cycling tests and rate retention tests. The rate retention tests were far larger in number since they were less time consuming, so we expected that the rate retention data would lead to better results when it came to the machine learning models.

Common to both sorts of experiment were the features relating to the battery and anode chemistries, along with some user set parameters. These are considered the independent variables of the data, and the performance data for a given observation was expected to be a result of primarily the anode/battery formulation, but the externally set variables also played a role in the results. For example, a well formulated battery that was set at too high a current would lead to lower capacity as a result of higher overpotentials at the electrode surfaces, increased cell resistance, and side reactions- this means that

current rate and voltages need to be taken into account when considering performance as well. [83] [25]

The externally set variables found in both types of experiments were *lower voltage* and *upper voltage*. These basically indicated the upper and lower limits on the voltage for when charging/discharging would cease, since going above or below these limits would be harmful to the battery.

The columns detailing the anode chemistry were the following: *class, material , category_2, loading_mass, synthesis, carbon_support, electrolyte_salt, electrolyte_solvent, FEC_addition, binder_type* and *electrode composition*. The class of the anode generally indicated the defining characteristic of the anode, such as porous anodes, hard carbon anodes, and co-intercalation anodes. The *material* and *category_2* columns referred to the main chemical elements in the anode, while the *carbon_support* column indicated the source of the carbon used in the anode. The electrolyte salt and solvents were the components that made up the electrolyte between the battery electrodes, and the binder type referred to the type of binder used to stabilize and hold the battery components together. Electrode composition referred to the ratio of active material, conductive material, and binder in the electrodes, and FEC addition indicates whether or not fluoroethylene carbonate (FEC), a popular electrolyte additive to reduce side reactions and increase stability, was added [63]. Loading mass refers to the electrode mass per unit area (milligrams per square centimeter) of current collector, and the synthesis column represented the technique used to create the electrode [58].

The data relevant to solely the rate retention tests were the columns labeled *current_rate (mA g-1)*, *capacity (mAh g-1)*, and *rate_retention*. The current rate indicated the current passed through the battery per unit mass of electrode (units of milliamps per gram), the capacity indicates the nominal capacity per unit mass of the electrode used, and rate retention was our electrochemical performance parameter of interest. Rate Retention indicates the ratio between the capacity attainable at the given current rate to the maximum possible capacity of the battery. Since higher current rates reduced capacity, higher rate retention values indicated that the electrode had strong performance over a range of currents [75] [69].

The columns exclusive to the long cycling data were *cycles, CE_first_cycle, current_long_cycling, capacity_at_end*, and *capacity_retention*. Cycles represented the total number of cycles

that a battery underwent during testing, relevant because of battery degradation with use. The current long cycling parameter (in units of milliamps per gram) was also user chosen, and was the current per unit mass used all throughout all the cycles of the experiment. The capacity at the end represented the capacity per unit mass that the battery was capable of reaching after undergoing the given number of cycles in the experiment.

Capacity retention and Coulombic Efficiency at the first cycle were the two performance characteristics of interest in the long cycling tests, although Coulombic Efficiency of the first cycle was not a function of the cycle amount as it was a measurement taken during the first cycle of the experimentation. Capacity retention indicates the ratio between the capacity of the battery after the long cycling to the capacity of the battery beforehand. Capacity fade during cycling occurs due to increases in resistance/impedance, growth of the SEI layer, parasitic reactions that deplete the amount of available ions for charge/discharge, and other complex mechanisms [51]. This capacity fading means that capacity retention is a value less than 1, as the final capacity is in the numerator of the ratio. Coulombic efficiency is a ratio of the amount of charge output (on discharge) to the charge input (during charge), and researchers typically document this value on the first cycle in order to assess its predictive capabilities regarding other factors of the battery such as total lifetime or the defectiveness of the electrode materials [79].

Chapter 2

Machine Learning Background and Methodologies:

2.1 Statistical Learning

The ability to learn from data has played a key role in advances in fields such as artificial intelligence, statistics, and data mining, but also has found application in finance, engineering, biology, healthcare, amongst many other fields. 'Learning' from data is the process of creating a mathematical model from observations in order to better understand phenomena occurring in the data. These models are expected to interpret new observations and create some meaningful inference, typically classification of some sort.

The *learning* is implemented when models are given a *training set*, which holds *objects* that contain a set of *features* and a set of *outcomes*. From these features and outcomes, we build predictive models that can predict outcomes for unknown data, which would be new objects with known features but unknown outcomes. To test the fidelity of models, *testing sets* are used, which are just a set of data that was not used in the training set, from which we can assess whether our predictor is accurate enough.

A typical example of statistical learning in action is the following: given some samples of flowers, for each of which we know certain attributes (such as petal and sepal length) and the flower species. The flowers are our set of objects, the attributes are the features, and the flower species is the outcome. From this data we train a model such that when given a flower of unknown species, our model can analyze the flower's attributes and give some prediction for what the given flower's species is.

2.1.1 Supervised Learning

Supervised learning tasks are an often used subset of machine learning that seeks to determine relationships between features and outcomes, such that given some set of features, we can predict the outcome associated with those features. We can consider the outcomes to be 'target attributes' or 'labels' of the objects comprising the data, and since these targets/labels are not known for the unseen data we wish to predict them.

There are two subsets of supervised learning:

- **classification**: The objects belong to two or more classes and already labeled data is used to assign unlabeled data under certain classes. The labeling in classification is discrete, not continuous, and we can consider the classes as categories under which the objects must be sorted based on their features. An example of classification would be sorting students into majors based on which classes they have taken. The objects would be the students, the features would be the classes taken, and the outcome/label would be the major.

- **regression**: In regression tasks, objects have features as before, but the outcomes are no longer discrete labels; instead there are continuous quantities. Since outcomes are now continuous, it is possible for our predictor to predict a label for an unseen object that does not match any of the labels that were in the training set. An example of regression would be predicting a human's height based on their weight, age, and gender. The objects would be the humans, the features would be weight, age, and gender of the humans, and the outcome would be the human's height. Here, height is on a continuous scale.

2.1.2 Unsupervised Learning

Unsupervised learning tasks are learning where the dataset is not distinguished based on some target attributes or labels. Examples of tasks include grouping data based on attributes, predicting the distribution of data based on the input, and reducing the dimensions of higher dimensional data so that it is more coherent and consumable.

This form of learning has much less literature and has some additional complexity when compared to supervised learning. In supervised learning tasks, we always have a straightforward measure of success of our predictive model by analyzing whether unseen data is labeled accurately or not. This is not the case in unsupervised learning, where effectiveness of a technique has no objective measure, and scientists must create their own criteria for assessing the efficacy of models.

The main types of tasks in unsupervised learning are:

- **clustering**: This task groups the set of objects such that objects in a given group are more closely related to each other, under some criteria, than to objects in other groups. This task is extremely important in data mining, pattern recognition, and computer graphics.
- **density estimation**: In density estimation tasks, given some random sample of data from a population, the goal is to construct a probability density function (PDF). The PDF estimates the relative likelihood that a data point takes on a certain value. This task is crucial in estimating how likely or unlikely unseen datapoints are to occur under certain conditions.
- **dimensionality-reduction**: In these tasks, the data has far too many dimensions to be easily represented to the human eye, and the structure of the dataset is used for classification such that the dimensions of the data can be reduced, yet preserve the meaning of the original data.

2.2 Mathematical Formulations of Machine Learning Regressors

2.2.1 Ordinary Least Squares Regression

A linear regression model takes a data set, $\{y_i,\ x_{i1}, \ldots, x_{ip}\}_{i=1}^{n}$, and finds a linear relationship between the dependent variable y and a vector of regressors x. This relationship is often simplified and written in matrix notation as

$$y = X\boldsymbol{\beta} + \boldsymbol{\varepsilon} \tag{2.2.1}$$

where y is the dependent variable, and X is a matrix of independent variables. β is a coefficient that measures how strongly the independent variables affect the dependent variable. ε is a vector of size n of error values that measures how much the dependent variable y deviates from the actual relationship between X and y [1].

Consider b to be a possible value for β. The quantity $y_i - x_i^T b$, which is the value of ε_i, measures the vertical distance between the data point (x_i, y_i) and $y = x^T b$. ε_i measures

the degree of fit between the actual data and the model.

The residual sum of squares (RSS) is

$$RSS(b) = (y - Xb)^{\mathrm{T}}(y - Xb) \tag{2.2.2}$$

$$\hat{\beta} = \arg\min_{b} RSS(b) = (X^{\mathrm{T}}X)^{-1}X^{\mathrm{T}}y \tag{2.2.3}$$

The value of b that minimizes RSS is called the ordinary least squares estimate [18]

2.2.2 Ridge Regression

Ordinary least squares aims to minimize the sum of squared ε_i in order to find the model of best fit. In ridge regression, a regularization term is included in the minimization to give weight to desirable solutions [1].

$$\arg\min_{\beta} \|X\beta - y\|_2^2 + \alpha\|\beta\|_2^2 \tag{2.2.4}$$

where A is a matrix, y is a vector of outcomes, and we want to solve for β such that $X\beta = b$. The regularization term, $\alpha\|\beta\|_2^2$ is often a multiple of the identity matrix. This model is intended to reduce the degree of overfitting that the Ordinary Least Squares model may produce. By increasing α the smaller the coefficients and the less chance of collinearity of features. [19].

Kernel Ridge Regression combines ridge regression and the kernel trick. The kernel trick maps problems to higher dimensions, and enables linear solutions to non-linear problems. [49].

2.2.3 K Nearest Neighbors Regression

The k-nearest neighbors algorithm takes an input k and finds that number of training samples that are closest in distance to a new data point, and uses them to determine the new data point's classification. The k-nearest neighbors regression can be used when the data classifications are continuous rather than discrete. The classification of the new data point is computed from the mean of the classification of its k-nearest neighbors [5].

2.2.4 Lasso Regression

Lasso is short form for *least absolute shrinkage and selection operator*. It is a regression model that involves both regularization and variable selection in order to minimize error.

$$\min_{\beta} \left\{ \frac{1}{N} \|y - X\beta\|_2^2 + \alpha\|\beta\|_1 \right\} \tag{2.2.5}$$

[1]. $\alpha\|\beta\|_1$ is the regularization term as in ridge regression. $\|y - X\beta\|_2^2$ is the sum of squared residual terms. N is the number of samples [17].

2.2.5 Elastic Net Regression

Elastic Net Regression follows a model that attempts to overcome the shortcomings of the Lasso Regression model. This model combines the penalty methods of both Ridge Regression and Lasso [16].

$$\hat{\beta} = \underset{\beta}{argmin}(\|y - X\beta\|^2 + \lambda_2\|\beta\|^2 + \lambda_1\|\beta\|_1) \tag{2.2.6}$$

[1]

2.2.6 Support Vector Regression

Support Vector Regression (SVR) analyzes data for classification. It takes training data where items are already classified and uses that data to create a model for classifying new data. The training data is in the form $(\vec{x}_i, y_i)$, where $\vec{x}_i$ is a d-dimensional training vector and y_i is the output [26].

SVR uses a penalty function to construct its classifying model. A penalty is imposed if a predicted value, $\hat{y}_i$, is greater than or equal to a distance ε away from the actual value y_i, or $|y_i - \hat{y}_i| \leq \varepsilon$. In addition, variables outside this boundary are given a slack variable penalty, ξ^+ if it lies below the boundary, or ξ^- if it lies above the boundary. Both $\xi^+, \xi^- > 0$ [26].

In order to classify unseen data, the model must optimize an error function. It mini-

mizes a function of this form:

$$R[\vec{w}, b, \xi] = \frac{1}{2}\|\vec{w}\|^2 + C\sum_{i=1}^{n}(\xi^+ + \xi^-) \tag{2.2.7}$$

subject to $\left\{\langle \vec{w}, \psi(\vec{x_i})\rangle + b - y_i \leq \varepsilon + \xi_i^+, y_i - \langle \vec{w}, \psi(\vec{x_i})\rangle - b \geq \varepsilon + \xi_i^-, \xi_i^+, \xi_i^- \geq 0\right\}$ [18]

2.2.7 Orthogonal Matching Pursuit

Orthogonal Matching pursuit attempts to estimate the solution of two problems. First, the *sparsity constrained* problem

$$\arg\min \|y - X\gamma\|_2^2 \text{ subject to } \|\gamma\|_0 \leq n \tag{2.2.8}$$

which finds the best solution with a given number of non-zero elements, denoted by n, and the *error-constrained* problem

$$\arg\min \|\gamma\|_0 \text{ subject to } \|y - X\gamma\|_2^2 < \varepsilon \tag{2.2.9}$$

which finds the best solution with a specified error margin [1].

2.3 Applying Machine Learning to Batteries

From the above discussion on the capabilities of machine learning in prediction problems, we begin to get an idea of how it could be used in solving problems that are relevant to electrochemical energy storage research. Some of the quantities we'd like to be able to predict for a battery are state of health, state of charge, useful life, capacity retention after cycling, and coulombic efficiency. These are all continuous values and therefore we have exclusively focused on supervised learning regressors, which are capable of predicting values for outcomes even if the training data did not include that specific outcome, as is often the case in dealing with batteries

In this project, we will attempt to use machine learning on two types of datasets, cycling data and anode performance data. From the cycling data, the goal is to predict state of charge from the charged passed and the voltage of the battery. From the anode per-

formance data, the goal is to predict the rate retention, capacity retention, and coulombic efficiency of the first cycle given the anode chemistry. In all of these cases, the outcomes for the machine learning algorithm is the quantity to be predicted and the features are the other columns in the dataset that characterize each observation.

As we will see in the next chapter, the parameters mentioned in the mathematical formulations of the regressors can be systematically varied in order to find which parameters return the best performing model. This was an important consideration when trying to find the best possible machine learning model since the choice of parameters has an appreciable effect on model accuracy. In addition to selecting parameters for the models, the features chosen to be included in the input data can affect the predictive performance as well. These and other considerations for finding the best regressor are discussed at length in the next chapter.

Chapter 3

Experimental Design

3.1 Data Science Basics

Essential to the practice of data science is the effective loading, storing, and manipulation of data. To accomplish this, libraries that extend the capabilities of Python were required. Python was the chosen language as it is simple for anyone to pick up, has robust, open-source data analysis packages, and most importantly, is free and maintained by a large community of enthusiastic individuals.

Since the data to be processed were those stored as Comma-separated value (CSV) files, the packages needed to handle the input of thousands of rows of data and also their efficient storage as tables. Once the data was loaded as some array-like object, data operations on these arrays needed to be efficient and fast, as data analysis requires operations such as comparisons between elements, numerical operations, and matrix rearrangements. NumPy, 'the fundamental package for scientific computing with Python', [10] was chosen, along with Pandas, a library 'providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language' [11].

Numpy provided an N-dimensional array object to store and operate on dense data, and scaled much more proficiently as data buffers grew compared to Python's built in types [71]. Arrays can be indexed, sliced, reshaped, joined, and split in quick, easy to understand commands, and provide the foundation for many other data analysis libraries, and thus was essential in any following work.

Pandas, a library built on top of Numpy, provided the framework for seamlessly converting CSV files to a format that could be easily operated upon in Python, on a scale far larger than that possible from basic Excel or other spreadsheet programs. The DataFrame object provided by Pandas is a multidimensional array with attached column and row labels, with the capability to handle missing data, data of differing types, and the flexibility to handle operations that did not map exactly to linear algebra operations on matrices. Throughout this project, the DataFrame object is exclusively used to handle imported data, so a quick rundown on how a DataFrame is set up follows [71].

A DataFrame is capable of handling mixed data, so it is common to see strings, integers, floats, and lists all contained within the 'cells' of a DataFrame. Similar to an Excel spreadsheet, a DataFrame can be displayed as a rectangular array, but Pandas provides labeling for the rows and columns so they can be easily accessed and categorized. The following is an example of a DataFrame:

	cycle	test_vol	test_capchg	test_capdchg	step_id	step_type
0	0.0	3.3035	0.0	0.0	1.0	4.0
1	0.0	3.3023	0.0	0.0	1.0	4.0
2	0.0	3.3023	0.0	0.0	1.0	4.0
3	0.0	3.3023	0.0	0.0	1.0	4.0
4	0.0	3.3023	0.0	0.0	1.0	4.0

Table 3.1: Sample DataFrame object

Some fundamental operations that can be performed on a DataFrame are indexing, selection, and handling missing data. When data is collected, not all columns for a given observation will contain data as it may not have been applicable or available. These missing values can be altered to some default values, be set aside, or removed entirely. Another object that is derived from a DataFrame is the GroupBy object. This object essentially allows the DataFrame to be organized by unique values in a certain column, which is extremely useful when data is to be analyzed along differences in a certain characteristic. Data can be easily aggregated, filtered, and transformed when organized as a GroupBy object, rather than performing the common operations manually on subsets of the data.

3.2 Common Python Commands and Useful Information

The following section will include some of the basic commands and nuances when working with Python. It should be helpful when reading and understanding the programs and code written in the Appendix.

Basics:

- # creates comment
- in general a method call changes the object permanently, but just performing an operation on it doesn't
- the following are the common imports:

```
import numpy as np
import pandas as pd
from pandas import Series, DataFrame
```

- **General:**

```
type(x) # gives the datatype of an object
len(object) # gives the length of the object
str(26) # turns the integer into a string
int("26") # turns the string into an int
bool('') == False # boolean of an empty string is False, but boolea
n of any other string is True
```

- **Booleans & Comparisons**
- Booleans: ==, <, >, <=, !=, >=, <> (same as !=)
- `and` checks if both boolean statements are true and returns true if so
- `or` checks if either boolean statements are true and return true if so

- **Sequence Types:** https://docs.python.org/2.7/library/stdtypes.html#sequence-types-str-unicode-list-tuple-bytearray-buffer-xrange (https://docs.python.org/2.7/library/stdtypes.html#sequence-types-str-unicode-list-tuple-bytearray-buffer-xrange)
- The main sequence type used is the list
- This table lists the sequence operations sorted in ascending priority. In the table, *s* and *t* are sequences of the same type; *n*, *i* and *j* are integers:

Operation	Result
x in s	True if an item of s is equal to x, else False
x not in s	False if an item of s is equal to x, else True
s + t	the concatenation of s and t
s * n, n * s	equivalent to adding s to itself n times
s[i]	ith item of s, origin 0
s[i:j]	slice of s from i to j
s[i:j:k]	slice of s from i to j with step k
len(s)	length of s
min(s)	smallest item of s
max(s)	largest item of s

Operation	Result
s.index(x)	index of the first occurrence of x in s
s.count(x)	total number of occurrences of x in s

- mutable sequences, such as lists, support these operations:

Operation	Result
s[i] = x	item i of s is replaced by x
s[i:j] = t	slice of s from i to j is replaced by the cont...
del s[i:j]	same as s[i:j] = []
s[i:j:k] = t	the elements of s[i:j:k] are replaced by those...
del s[i:j:k]	removes the elements of s[i:j:k] from the list
s.append(x)	same as s[len(s):len(s)] = [x]
s.extend(x) or s += t	for the most part the same as s[len(s):len(s)]...
s *= n	updates s with its contents repeated n times
s.count(x)	return number of i\u2018s for which s[i] == x
s.index(x[, i[, j]])	return smallest k such that s[k] == x and i <=...
s.insert(i, x)	same as s[i:i] = [x]
s.pop([i])	same as x = s[i]; del s[i]; return x
s.remove(x)	same as del s[s.index(x)]
s.reverse()	reverses the items of s in place
s.sort([cmp[, key[, reverse]]])	sort the items of s in place

- **Dictionaries** https://docs.python.org/2/library/stdtypes.html#mapping-types-dict (https://docs.python.org/2/library/stdtypes.html#mapping-types-dict)

```
#a dictionary is a collection of keys and values
my_dict = {'Joe':2, 'Sam':3}
my_dict['Joe'] # getting the value using the key
d['answer'] = 42 # reassign the existing key's value, or create the key if
 it doesnt exist
```

- dictionary comprehensions: `{x:x**2 for x in range(10)}`, this determines the values and the keys are just 0 to 10 as usual

```
spam.items() # contains the key and value in each iteration: a list of tup
les in Python 2 and separate key and value in Python 3
spam.keys() # contains the key in each iteration (a list)
spam.values() # contains the values in each iteration (a list)
'color' in spam.keys() # check if the string is in the keys
'Zophie' in spam.values() # check if the string is in the values
'color' in spam # checks the keys for the string
'key' in my_dict # boolean of whether the key exists
```

- **Lists & Sets:**

```
###Lists can hold anything! List of dicts, strings, ints...
list1 = [] # creates empty list, good for `for` loops
x in list # returns a boolean of if x is an element of list
cities.append('CHI') # appends a value to a list

###Sets only care about unique values
set1 = {x,y,z} # initialize a set
set(list) # removes any repeats from the list and returns a set
set1.difference(set2) # returns a new set with the values that are NOT in
 set1, but in set2
s1.intersection(s2) # returns a new set with just the intersection
s1.symmetric_difference(s2) # returns a new set with just unique values fr
om both s1 and s2
s1.union(s2) # returns a new set with all the elements of both sets
```

- list comprehensions use `for` loops to generate values for the list's elements in order to initialize a list:
  - the format is `listname = [(element expression) (for loop inline)]`
  - `lst = [x**2 for x in range(0,11)]` returns a list of 11 elements with each element of the form `indexvalue**2`
  - we can also use `if` statements in the `for` loop inline expression so that an iteration's values are only passed to the list if they satisfy a certain condition
    - `lst = [x**2 for x in range(7) if x % 2 == 0]` the for loop only returns the values from the 0th, 2nd, 4th, and 6th iterations so the list will only have 4 elements

- **String Format Operators for Printing:**
- in order to print things concisely we use %x placeholders in our print strings, where x represents what type of variable the actual value we want to put is. `%s` is for strings and `%d` is for integers
- `%s` and `%r` are used to convert whatever is being represented into strings for printing
- Since the string also needs to know what variables to actually put in, we must put a command after the print string (on the same line) that tells what each variable should be sequentially

```
### first declare some random variables
my_eyes = 'Blue'
my_hair = 'Brown'
### now we will show the print formatting
print "He's got %s eyes and %s hair." % (my_eyes, my_hair)
# so we need to specify what goes in each placeholder spot sequenti
ally after the string itself using % (placeholderactual1, placehold
eractual2, etc...)
```

- **Math:**

```
# * for multiplication
# / for division-
# ** for exponents
# - for subtraction
# + for addition
range(x) # creates a list of numbers from 0 up to x, not including
 x, but it is not the same kind of list we can display
range(3,8,2) # creates list of elements from 3 to 7 going by 2s, a
 range object as above, not an actual list
round (float, decimalplace) # rounds the given float to the number
 of desired decimal places, which is defaulted to 0 decimal places
```

- **Loops, etc:**
- the form an iterator is:

```
for x, y, z in object:-
  execute this line
```

- iterating on a list:
  - as many iterations as elements in the list
  - each iteration returns just the element
  - so a list of lists: each iteration returns the nested lists in index order
- iterating on a single unnested dictionary:
  - each iteration just returns the key of the ith pairing in the dictionary
  - we can use the dictionary iterables:
    - `dict.items()` contains a list of tuples with each tuple a key-value pairing, so if we unpack using (x,y) in the for loop we can get the key and value separately per iteration
    - `dict.keys()` we can iterate over the keys as normal
    - `dict.values()` we can iterate over just the values

- **Functions:**

```
# creating functions
def adder(x,y):
    '''this function adds x and y'''
    answer = x + y
    return answer # no return value gives us None as the return
```

- **Reduce:** `reduce(function, iterable)`
  - this built-in function takes two arguments: the function of two arguments and an iterable
  - the function first evaluates the first two elements of the iterable, then evaluates the result with the third element, then that result with the fourth and so on... until only one value is left

```
lst =[47,11,42,13]
reduce(lambda x,y: x+y,lst) # returns (((47 + 11) + 42) + 13)
###
#Find the maximum of a sequence (This already exists as max())
max_find = lambda a,b: a if (a > b) else b
#Find max
reduce(max_find,lst)
```

- **Zip:** `zip([iterables...])`
  - the function takes multiple iterables and truncates all of them to the number of elements in the shortest (least elements) iterable, and then creates a list of tuples, where each tuple contains all the ith elements of the iterables
  - if only one iterable: creates a list of tuples with just one element per tuple

```
x = [1,2,3]
y = [4,5,6,7,8]
# Zip the lists together
zip(x,y) # [(1, 4), (2, 5), (3, 6)]
```

- **Enumerate:** `enumerate(sequence, start=0)`
  - argument is an iterable and the return will be an iterable/sequence of tuples with each tuple containing the position of the item first (analogous to index) and then the item itself. So it basically shows a Series
  - the start kwarg allows us to choose which index to start counting at, but the default is 0
  - returns an enumerate object that we can cast as a list of tuples using `list(enumerate(seasons, start=1))`
  - iterating over an enumerate object is similar to iterating over a tuple except we dont need to unpack, and the index and element are returned per iteration. So if we iterate using just one counter we'll get a tuple, and if we iterate with two counters, the first is the index and the second is the actual item.

DataFrames Basics

http://pandas.pydata.org/pandas-docs/stable/dsintro.html (http://pandas.pydata.org/pandas-docs/stable/dsintro.html)

```
#additional things to import
import pandas as pd
from pandas import Series, DataFrame
```

- a DataFrame (DF) is basically a representation of a dictionary
- initializing a DF uses the array, index values, and column values:

```
dframe = DataFrame(randn(25).reshape((5,5)),index = ['A','B','D',
'E','F'],
                   columns = ['col1','col2','col3','col4','col5'])
dframe1 = DataFrame(np.arange(4).reshape((2,2)), columns = list('A
B'), index = ['NYC','LA'])
```

- in general, we can index using: actual values, a list of values, a boolean vector/Series, and with slices (using `:`).
- `df[columnname][rowname]` returns the value of that cell. The rowname can be integer label indexing but the column name CANNOT be
- `df.loc[rowname,columnname]` returns the value of the cell. Only works with Label indexing for both. NO INTEGER LABELS
  - `df1.loc['a']>0` returns a boolean series for every column that says whether the row 'a' of that column meets the requirements
  - `df1.loc[:,df1.loc['row1']>0]` returns every column meeting the requirement (which is a boolean series/column vector).
  - `dframe.loc[dframe.loc[:,'column1'] > 0,dframe.loc['row1',:] > 0]` returns a DF of every cell meeting the requirement. Basically it is saying 'Which rows meet this requirement and which columns meet this requirement?' This is why when passing a boolean series in an indexing, the length of the Series/vector and the indexes must exactly match the length and indexes of the DF and index position. So `df[:,boolean]`, must use a boolean that is a column vector with indexes exactly matching the existing DF's column index. The Series/vector must also be derived from the DF itself, and not user-made. This is the case always, except for the general indexing when using `df[columnname]`.
  - Can create a row or column or cell that doesn't exist. If just a cell is made, the rest of the row and column will be NaN
- `df.iloc[rowint,columnint]` is the exact same as `df.loc[rowname,columnname]` but only works with integer labels!
  - since it can't use actual labels, we cannot use this to create rows/columns
  - a useful aspect is the using out of range splices doesn't return an error. It'll either return whatever it can in the range, or an empty Series/DF
- `df.ix[rowname,columnname]` returns the value of that cell. Both columnname and rowname can be integer label indexing. It performs all the same functions as `df.iloc[]` and `df.loc[]`. It can create cells/rows/columns as well.

- essential functions using DF: THEY DONT CHANGE THE DF ITSELF, just create new DF! unless equality sign involved

```
nfl_frame.columns # stores names of all columns as an index object
nfl_frame.index # stores names of all rows as an index object
nfl_frame['First Season'] # displays the column that MUST be in single quo
tes
DataFrame(nfl_frame,columns=['Team','First Season']) # creates a new DF us
ing the selected columns only
nfl_frame.head(x) # grab x number of rows from the top of the DF
nfl_frame.tail(x) # grab x number from the rows from the bottom of the DF
nfl_frame.ix[x] # displays all the information corresponding -to the xth i
ndex row

nfl_frame['squad'] = "CodeNation" # this assigns every value in the column
nfl_frame['Squad'] = np.arange(5) # passing an array to the column fills i
n the column sequentially (corresponding to numeric index)
city_frame = DataFrame(dictionary) # passes a dictionary to the DF to crea
te a DF from it
```

Finding Values in a DF:

- for DF:
  - just to look for one value in a column: `df.loc[df['column_name'] == some_value]`, returns DF of the row
  - to look for multiple values or use in an iterable (loops,etc): `df.loc[df['column_name'].isin(values_list)]`, returns DF of the rows
  - to look for one value in a column and return just the index and the value in a Series: `df['column_name'].loc[df['column_name'] == some_value]`
  - to look for multiple values in a column and return the indexes as the indexes in a Series: `df['column_name'].loc[df['column_name'].isin([0,2])]`
  - checking one value in the index: `df.loc['one']`, returns Series of that row
  - checking multiple values in the index: `df.loc[df.index.isin(['one','two'])]`, returns DF of the rows

Missing Data and null values:

```
data.isnull() # creates a boolean series that says if there is any null va
lues in the indexes
data.dropna() # creates a series that removes all the indexes that has nul
l values
dframe.dropna() # drops any rows with null values
dframe.dropna(axis=1) # drops any columns with null values
dframe.dropna(how='all') # only drops rows will every value null
dframe2.dropna(thresh=x) # removes rows that don't have at least x data po
ints (null doesn't count as a data)
dframe2.fillna(1) # fills all the null values as 1
# permanent changes to the DF instead of just making a new DF:
dframe2.fillna(0, inplace = True)
```

3.3 Data Visualization

3.3.1 The Task

Data visualization is typically an underappreciated aspect of data analysis, as the core of the work is already complete, but data visualization is often the most important aspect of an analysis. Without comprehensive, complete, and pleasing representations of the results, conveying information is difficult and counterproductive to the goal of sharing knowledge and findings.

In this project, data visualization is taken in a different direction, where the goal was to create interactive charts and graphs, that have the ability to tell a *story*, rather than static displays. In our case, *story* is more accurately defined as trends and events in the data that would be difficult to discern just by looking at the data or a simple chart.

The first interactive chart to be made was a chart of *Capacity vs Cycle*, where each data point was clickable, juxtaposed with a chart of *dQ/dV vs Voltage*. Upon a point in the *Capacity vs Cycle* being clicked, the *dQ/dV vs Voltage* was to update its differential capacity curves to match the clicked cycle's data. The second chart was one of solely *dQ/dV vs Voltage*, but a slider would be used to select the cycle whose data was to be shown.

3.3.2 Selecting Tools and Packages

Seemingly a simple task, creating interactive graphs that can perform the intended tasks using external data requires a framework for performing the data analysis, connecting the data to the graphical output, and also updating the graphs based on user interaction. Accomplishing this using solely Python was nearly impossible since graphical displays and user interactivity require a lot of packages and building the desired program from the ground up would require an unrealistic amount of time and effort. JavaScript and HTML, two languages that are meant for front end web development already had many of the tools required to build and display charts and accept user input, but did not have the requisite data analysis components that were needed.

Since data analysis is such a burgeoning field, many groups have attempted to integrate the vibrant and dynamic displays of HTML and CSS into Python, but at this time, there is no real standard package that provides all components required to perform the

tasks of interactive data displays. After sorting through several candidates, such as Plotly, Google Charts, Matplotlib, Seaborn, and Pygal, the chosen package, although still in its nascent stages, was Bokeh.

Bokeh, a library meant specifically for interactive visualizations in web browsers utilizing 'very large or streaming datasets' [35], had been created specifically for Python and had already incorporated the creation of HTML pages from Python code. This meant that the data analysis could be performed in Python and that the resulting data vectors could be interpreted into charts and graphs in HTML relatively smoothly.

Bokeh already had the capabilities to perform a whole host of important interactions such as handling button clicks, typed user input, and hover information. The main weakness of Bokeh was that it did not take DataFrames from pandas as input, and instead used dictionaries or lists. This was mostly remedied through the easy conversion between DataFrame columns and dictionaries, but would pose a problem when lists had to be created from DataFrames.

Bokeh's plotting functions are similar to those in Matlab, Matplotlib, and other programming languages, but require the use of ColumnDataSource objects, which are essentially containers for the x and y axis data as vectors. All vectors need to be of the same length as points from corresponding positions in each vector are used to create the plot. For example, to create a plot of $y = x^2$ for the x values 1 to 10, an x vector of length 10 would be used along with a y vector of length 10, where each value at an index position in the y vector was the square of the value in the x vector at the same index position [35].

3.3.3 Interpreting and Formatting Input Data

After importing the cycling data, removing the irrelevant columns, and creating a DataFrame from it, it appeared as the following:

	cycle	test_vol	test_capchg	test_capdchg	step_id	step_type
0	0.0	3.3035	0.0	0.0	1.0	4.0
1	0.0	3.3023	0.0	0.0	1.0	4.0
2	0.0	3.3023	0.0	0.0	1.0	4.0
3	0.0	3.3023	0.0	0.0	1.0	4.0
4	0.0	3.3023	0.0	0.0	1.0	4.0

Table 3.2: Cleaned up cycling data

To organize the data for the *Capacity vs Cycle* graph, where the maximum charge transferred per cycle during the charge step is indicated, the data needed to be grouped according to cycle, the maximum value in the *test_capchg* column located, and then the data pairs of the form (cycle, capacity transferred) were stored in a separate DataFrame. Using Bokeh, the vector for the cycle number and the vector for the capacity value could be transferred to a ColumnDataSource object and then plotted. These steps are indicated in lines 1 to 35 of the Python code in Appendix D, Part 1, Snippet 1.

	cycle	charge
0	0	0.545756
0	1	1.159726
0	2	1.158055
0	3	1.155909
0	4	1.155100

Table 3.3: DataFrame for Capacity vs. Cycle graph

To create the *dQ/dV vs. Voltage* graph was a more complicated process, as there were two of these differential capacity curves for each cycle, and for each curve, the dQ values had to be calculated for each voltage increment, dV. In order to accomplish this, the increments dV first had to be chosen. The smaller the increments, the higher the resolution of the graph, but also the higher the processing time, so a value of 500 was chosen, which could easily be altered to any other value as desired. These increments evenly split up the voltage range of 2 to 4.2, and to calculate representative values of the voltage for the abscissa/x-axis, the endpoints of each increment were averaged.

After the data was grouped by cycle, and then further by charge or discharge step, the change in capacity for a given voltage increment had to be calculated. This was done by finding the maximum charge transferred (in the given voltage range) and the minimum charge transferred (again, in the given voltage range) and then subtracting, giving the range of the capacity, our *dQ/dV* value. This simple technique was possible since the data in the *test_capchg* and *test_capdchg* columns was formatted as the cumulative total of charge transferred thus far, so the values were always increasing as charge/discharge proceeded. So for each voltage increment, a datapair of the form (average voltage, change in capacity in voltage range during discharge, change in capacity in voltage range during charge) was created and added to a DataFrame specific to the given cycle. After this was done for all cycles, a list of DataFrames, where the list index corresponded to cycle number had been created, and could be plotted. These steps are indicated in lines 38 - 102 of the Python code in Appendix D, Part 1, Snippet 1.

	voltage	dq/dv_chg	dq/dv_dchg
0	2.883968	0.000299999	0.0004493
0	2.888377	0.000299999	0.000450015
0	2.892786	0.000299998	0.000449896
0	2.897194	0.000300027	0.000449896
0	2.901603	0.000299999	0.000449896

Table 3.4: dQ/dV data for the Differential Capacity Curves

3.3.4 Using JavaScript to Create Custom Interactions

Although we had now created DataFrames containing the data required to plot the *Capacity vs Cycle* and *dQ/dV vs. Voltage* graphs, a way to create the interactive features was needed. As mentioned earlier, since Bokeh used ColumnDataSource (CDS) objects in order to gather the data for plotting, all the data had to be formatted as simple vectors, which in Python are list objects. Once the data was loaded into a CDS, Bokeh would display the graph in HTML. The data for both graphs had been already been assembled into DataFrames whose columns were readily converted to lists, so displaying the HTML graphs was straightforward. Bokeh came with a built in object called a callback that updated CDSs based on some condition set by the user, so this was chosen as the method to implement the interactivity.

After being triggered by some condition, a callback ran some selected JavaScript code, called the CustomJS code, that could alter the Bokeh plot in HTML by changing the data in the plot's CDS. Although Bokeh conveniently provided this functionality, it had not yet been designed for more complicated data lookups, and after a conversation with Brian Van de Ven, the creator of Bokeh, I discovered that the JavaScript in a callback could only interact with lists created in Python. This would lead to some complications as the list of DataFrames with the dQ/dV data for each cycle was in too complex of a format.

For the first interactive plot, where points clicked on the *Capacity vs Cycle* graph would update the data for the *dQ/dV vs. Voltage* graph, the intended callback procedure was as follows:

1. Once a point on the *Capacity vs Cycle* graph is clicked, it's index in the vector of points can be retrieved and stored in a variable in JavaScript. Since the points were ordered by cycle number beginning at cycle 0 and indexing begins at 0 as well, the index of a point corresponded exactly to the cycle number.

2. The list of DataFrames with the *dQ/dV* data was built such that accessing the ith index of the list contained data for the ith cycle. So the index retrieved in the earlier step could be used to select the correct DataFrame.

3. The callback would retrieve the CDSs for the two differential capacity curves and store them as variables in JavaScript. The x (average voltage values) and y vectors (corresponding *dQ/dV* values for discharge and charge) of each curve are extracted from the CDSs as well so they can be altered.

4. A for loop would run through every point in the extracted x and y vectors, replacing the current value with the appropriate values from the DataFrame for the cycle that was clicked on.

The JavaScript code in the CustomJS object needed a way to access the Python DataFrames, and since directly passing the information was not possible, a clever workaround was used instead. The JavaScript code was essentially a string in Python, so using Python's string formatting to replace values in a string with a variable, the JavaScript code was able to accept the string representation of a list or a list of lists from Python.

To format the list of DataFrames as a list of lists, the DataFrames had to be *flattened*. The flattening procedure would convert an entire DataFrame of m rows and n columns

into a single list, where the first row's values occupy the 1st to nth positions, the second row's values occupy the $(n+1)$th to $(2n)$th positions, and so on, for a total of $m*n$ values in the flattened list.

	voltage	dq/dv_chg	dq/dv_dchg
0	2.883968	0.000299999	0.0004493
0	2.888377	0.000299999	0.000450015
0	2.892786	0.000299998	0.000449896
0	2.897194	0.000300027	0.000449896
0	2.901603	0.000299999	0.000449896

Table 3.5: DataFrame of dQ/dV data before Flattening

	Value at Index
0	2.883968
1	0.000300
2	0.000449
3	2.888377
4	0.000300
5	0.000450
6	2.892786
7	0.000300
8	0.000450
9	2.897194
10	0.000300
11	0.000450
12	2.901603
13	0.000300
14	0.000450

Table 3.6: The Flattened DataFrame

Having created a list of lists, the for loop in JavaScript had to be modified to work with the flattened lists. The retrieved index of the point clicked would still be used to access the desired list corresponding to the clicked cycle, and the flattened list of the $3 * totalnumberofcycles$ values still contained all the information needed. Since the 1 to 3rd positions corresponded to the first row, then the 0 to 2nd indexes corresponded to the 0 index row, and the $3 * i$ to the $((3 * i) + 2)$ indexes corresponded to the ith index row, up until the $(totalnumberofcycles - 1)$th index row. Since the for loop provided

the integers from 0 to $(totalnumberofcycles - 1)$, by indexing the list of lists first using the selected point's index, and then using the indices corresponding to the correct row's data, the data in the current differential capacity curves' CDSs could be replaced.

All of the above considerations came together in building the first interactive graph, which is indicated in lines 102 - 121 of Snippet 1, 1 - 28 of Snippet 2, and 1 - 20 of Snippet 3 of the Python code in Appendix D, Part 2. Bokehs TapTool was used to indicate when a callback to update the differential capacity curves should be triggered based on whichever point on the *Capacity vs Cycle* graph was chosen.

In building the second interactive graph, where a slider tool would be used instead to indicate the cycle number, much of the same procedures as the above were followed. The main difference was that the list of lists with the *dQ/dV* data was now first indexed using the slider's value, which ranged from 0 to (total number of cycles -1), rather than using the index of the selected point. Bokeh's Slider object was used to indicate when the callback to update the differential capacity curves based on whenever the displayed slider's value was changed.

3.4 Finding the Best Machine Learning Models

3.4.1 The Task

Attempting to use data science tools to predict quantities with a variety of nuances such as nonlinearities, different regimes of degradation, and intricate mechanisms of reaction, meant that ample data was required to train models. Unlike for simpler linear relationships where a few observations may provide enough information to train a model accurately, hidden underneath the quantities of SOC, rate retention, and capacity retention are complex non-linear conservation equations, chemical kinetics, side reactions, transport and diffusion phenomena, amongst other factors all of which may play a role in their determination.

The power of machine learning resides in its ability to recognize patterns and relationships in data that would be impossible for a human to discern, and even in cases where the predictive capabilities of trained models is poor, model parameters and relative accuracies of certain models can indicate which directions to proceed in next. As described

in the previous chapter, different machine learning models have different algorithms for predicting values from test data, and therefore a variety of strengths and weaknesses depending on the form and relationships within the data. Although certain predictions can be made as to the efficacy of certain models based on the data being input, the ability to vary models parameters and the number of input features, along with uncertainty in model run times, makes testing multiple models unavoidable.

Although two types of totally different datasets were to be analyzed, the goal was to create a workflow that could accommodate any sort of input data, with minor adjustments. This would facilitate the use of machine learning in understanding and analyzing data beyond whatever work was being performed for this project. Ultimately, since the quantities to be predicted were continuous values, regression models were used, so any future classification tasks would require adjustments in how the models were scored and how data was encoded and decoded. The desired output of our program was not only the best performing machine learning model, but also the parameters and features that enabled that best prediction. By analyzing performance across models, features, and parameters, the relationships between the variables in the data would be easier to elucidate. If there were any cases where simple models were able to fit the data accurately, such as a linear OLS model, it may have been possible to assign weights to different features in the input and assess which had most impact on the output quantities.

3.4.2 Scikit-Learn

Just as we used Numpy and Pandas as libraries to enhance the capabilities of Python for data manipulation and processing, we needed a library that would allow us to access a wide selection of machine learning models and have the tools to adjust the models to fit our needs.

There are various free libraries that exist to implement machine learning in Python such as TensorFlow, Keras, Theano, and scikit-learn. Of the many choices available, the one that was the best documented and relatively easy to pick up was scikit-learn. Scikit-learn, free and open-source, just like Numpy and Pandas, includes an assortment of regressors and classifiers, as well as modules for parameter selection, feature selection, cross-validation, scoring, clustering, dimensionality reduction, and many others that we would not be making use of. One of the most attractive features of scikit-learn was the robust

documentation and active user community, both of which would come in handy when running into trouble setting up our analysis.

After a run through of the available models in scikit-learn and their strengths/weaknesses with regards to the particular data being used, I narrowed down the model types to be used to a list of fifteen different model types (with some overlap). The list was as follows: Linear Support Vector Regression, Ordinary Least Squares Regression, Ridge Regression, Lasso Regression, Elastic Net Regression, Orthogonal Matching Pursuit, Bayesian Ridge Regression, Stochastic Gradient Descent Regression, Decision Tree Regression, Radial Basis Function Support Vector Regression, Polynomial Support Vector Regression, Nu Support Vector Regression, Kernel Ridge Regression, K Nearest Neighbors Regression, and Supervised Neural Networks Regression. By using a substantial list of models with enough variety in their implementations, we hoped that enough of them would perform well enough to lead to helpful conclusions regarding our battery data.

3.4.3 Workflow: Preprocessing, Encoding, Feature Selection, Parameter Selection, & Cross-Validation

Preprocessing Data

Once the dataset was imported into Python using pandas, it was converted to a DataFrame to prepare it for analysis using scikit-learn. These preprocessing steps would be common to any machine learning endeavor, and essentially cleans up and formats the data so it ends up in a standard format acceptable by the different scikit-learn models. This standard format was a DataFrame of the features and a single column DataFrame/vector of the outcomes, where each row of the DataFrames corresponded to a certain observation.

One problem that arises in datasets is the presence of missing values, and how to treat observations with missing values. Removing all rows with missing values may lead to a large loss of data, as certain features/columns may be prone to having missing data so a large number of otherwise complete observations may have that feature's value missing. In addition, features/columns with a great deal of missing data may be considered inadequate for analysis (since there may not be enough information about the feature to ascertain any relationships with the outcome), and depending on the scope of the task, it can be better to remove those columns entirely. In this project, certain columns of

the anode performance data had a large percentage of their data missing ($> 20\%$), and a threshold value was decided for whether a column should remain in the analysis. After removing the columns with excessive numbers of missing values, removing any observations with missing data led to a negligible loss in data for analysis.

Since we were performing regression tasks, the column with the outcome that was to be predicted had to be separated from the rest of the DataFrame, while preserving the order of the observations. This column required no furthering processing. The remainder of the DataFrame, containing the features, needed to be cleaned up with any variables known to be non-essential for the prediction of the outcome removed. This was not a step for aesthetic purposes, as redundant or irrelevant features obfuscate the relationship between the relevant features and the outcome, lowering the accuracy of the model. There were some final steps in preprocessing such as making sure that every column that was meant to be a floating point number was, in fact, a float, rather than a string.

Encoding Features

Scikit-learn, just as any machine learning library, accepts only inputs that are integers or floats, regardless of whether the data was categorical. This means that a categorical feature such as color had to have each unique value converted to an integer before being processed by the machine learning algorithm. This process of converting categorical data to an appropriate form is called encoding, of which there are two main types: label encoding and one hot encoding.

Label Encoding converts categorical data into integers where the categorical labels have an arbitrary correspondence with the integers they were assigned to. Although this is clear to a human observer, since the converted labels now had some numerical value assigned to them, algorithms may take the magnitude of these integers as a measure of difference between unique values rather than as just an arbitrary differentiator. For example, the 'average' of a categorical value assigned to 1 and a categorical value assigned to 3, would be the value 2. Therefore the categorical value assigned to 2 would be seen as a midway point between the other two categorical values, although this is clearly not the case [24].

This effect of Label encoding is typically not profound as no causal relationship should

exist between the integer assigned and the outcome, and the other features will exhibit a far stronger relationship with the outcome in a valid analysis. In certain cases where there may not be enough data or small differences between observations are especially pronounced, label encoding may be a confounding factor in model performance. Regardless of this slight weakness, label encoding is commonly used in machine learning, especially because it maintains the advantage of keeping the number of features before and after encoding the same. Label encoding also helps save a great deal of data by converting long strings to integers [21].

The following is an example of data being encoded using Label encoding:

	Original		Label Encoded	
	Name	Race	Name	Race
0	Goku	Saiyan	2	2
1	Piccolo	Namekian	3	1
2	Bardock	Saiyan	1	2
3	Android-16	Android	0	0

Table 3.7: Label Encoding

One hot encoding takes a different approach to encoding categorical data by performing a binary encoding scheme, where all the unique values of a category end in an orthogonal vector space [46]. For a given category with n unique values, n columns are made, one for each unique value. An observation has one of those unique values as a characteristic if and only if the value in the column designated for that unique value is a 1. This scheme ensures that the values in the columns for the other unique values are 0s, and therefore groups observations with the same unique value for a category into a vector space orthogonal to every single other observation that had a different unique value for that category. This orthogonality indicates to algorithms that the unique values of a category are independent and removes any notions of possible similarity between them, an aspect that Label encoding is not able to capture [57].

The following is an example of data being encoded using One Hot Encoding:

One hot encoding, when used for several categorical features, each with multiple unique values, can lead to numerous amounts of resulting features. As an example, if there are n categorical features, each with m unique values, this leads to the creation of $m * n$ new features. Create such a sparse matrix in a high dimensional vector space can lead to what is known as the 'curse of dimensionality', where the distance between points

	Original Name	Race	One Hot Encoded Name_Android-16	Name_Bardock	Name_Goku	Name_Piccolo	Race_Android	Race_Namekian	Race_Saiyan
0	Goku	Saiyan	0	0	1	0	0	0	1
1	Piccolo	Namekian	0	0	0	1	0	1	0
2	Bardock	Saiyan	0	1	0	0	0	0	1
3	Android-16	Android	1	0	0	0	1	0	0

Table 3.8: One Hot Encoding

has become far too great to lead to a meaningful interpretation of how points are related [9]. Another problem with one hot encoding, as we will later see, is that the creation of numerous features leads to slower data processing and that performing feature selection with an excessive number of features quickly becomes impractical.

Feature Selection

Feature selection is an important part of creating an optimal machine learning model, especially in tasks where the roles played by independent variables/features are not well defined. Features, as discussed in the previous chapter, are the characteristics of an observation that will be used for predicting the outcome, and therefore the features available (or unavailable) to the machine learning model are pivotal in determining its performance. Other than considerations for accuracy, picking the smallest, yet most relevant set of features yields benefits in reducing processing, testing, and training time.

In terms of model performance, picking the most 'efficacious' subset is a subjective term as different optimization methods/criteria exist for picking features that attempt to find a balance between the number of features and the accuracy of the prediction. In general, reducing any redundant or irrelevant features helps reduce variance in the observations that will be trained on [47]. Noise and inconsequential variation, if present in a significant amounts, can lead to overfitting where the model has been made too specialized for the training set by having unnecessary parameters or by altering parameter weights. Ascertaining causal relationships between features and the outcome, rather than random, circumstantial connections, is necessary for predictions to be accurate for testing data, rather than just for the training set.

In scikit-learn, two feature selection functions were chosen to implement and assess two different underlying techniques in feature selection. The first of these functions, SelectKBest, only needed the dataset as an argument, selected the k best scoring features,

where k is a user-defined parameter of the length of the final feature set, and the scoring method was the F-test regression statistic [15]. This function works by first using just one feature and selecting the highest scoring feature (using the F-test) to remain in the model. To this model of one feature, the remaining features are added one by one, and the feature that gives the highest scoring resulting model of two features is selected. This procedure repeats until k features have been selected.

The F-test statistic is a measure of how much 'actual' improvement occurs when adding a variable/parameter to a regressor [13]. This is a helpful statistic in pruning the number of features, as adding a random variable to a regressor would increase its accuracy due to the addition of a parameter that could potentially fit additional points (or the parameter could just be set to 0 and have the exact same accuracy as the regressor without it), but in machine learning algorithms we want to avoid overfitting and adding irrelevant variables. The formula to calculate this statistic, comparing two models (given subscripts 1 and 2) is the following:

$$F = \frac{\left(\frac{RSS_1 - RSS_2}{p_2 - p_1}\right)}{\left(\frac{RSS_2}{n - p_2}\right)} \tag{3.4.1}$$

where p_i is the number of parameters in the ith model, and RSS_i is the residual sum squares of the ith model,

$$RSS = \sum_{i=1}^{n} (y_i - \hat{y})^2 \tag{3.4.2}$$

where y is the true value of the outcome, and $\hat{y}$ is the value predicted by the model [7].

The second function, called RFECV, required both the dataset and a specific machine learning model type, implemented recursive feature elimination along with a built in cross validation test for the optimal number of parameters [14]. This scheme works by first training a model on all the available features using a subset of the data. The features with the lowest weights in the model are eliminated one by one, where after every elimination the model is evaluated, using cross validation (discussed later in this section), on a validation set and given a score. At the end of this process, the number of features that gave the highest score is considered the optimal number of features and the features which gave this highest score are recorded. RFECV requires a model that involves the weights assigned to each feature and therefore the following model types were not able to perform this form of feature selection: Radial Basis Function Support Vector Regression, Polynomial Support Vector Regression, Nu Support Vector Regression, Kernel Ridge Regression, K Nearest Neighbors Regression, and Supervised Neural Networks Regression.

Parameter Selection

As we saw in the previous chapter on Machine Learning methods, each regression technique has parameters which can be tuned. These parameters can have an enormous effect on the model's predictive capabilities as they feature heavily in the optimization equations and provide the values for penalties, losses, and ratios that characterize a model. Although these parameters are user selected, it is difficult to discern what values would be optimal for a given dataset and a given model type. When the data is complex and non intuitive, which is often the case with practical datasets, it is common practice to try a variety of values for each parameter, selecting the best combination afterwards.

Scikit-learn provides a function, GridSearchCV, that uses a given model type and a user-chosen array of values for each parameter, to exhaustively search and score (using cross validation on the dataset) every combination of parameters [20]. Although this uses up a lot of time and computing power, finding the optimal parameters to use for a given dataset leads to substantial increases in predictive accuracy.

Training, Cross Validation, and Scoring

Once a model has been initialized with the proper parameters, it is ready to be trained on the training set. Once trained on a subset of the data, the model can be tested on observations that it has not yet seen, and scoring can be performed in a variety of ways. There are a variety of model evaluation parameters for classification tasks, but for regression tasks where the predictions are in a continuous range, evaluation is typically done using the coefficient of determination.

The coefficient of determination, also known as R^2, is a measure of how well a statistical model replicates the actual outcomes. This is done by calculating the proportion of the variance in the outcomes that is predicted by the model; the closer the predicted variance is to the actual variance, the closer the R^2 value is to 1. The equation for calculating R^2 for a sample of n observations, where y is the true value of the outcome, $\hat{y}$ is the value

predicted by the model, and $\overline{y}$ is the mean of the actual outcomes, is:

$$R^2(y, \hat{y}) = 1 - \frac{\sum_{i=0}^{n_{samples}-1}(y_i - \hat{y_i})^2}{\sum_{i=0}^{n_{samples}-1}(y_i - \overline{y})^2} \tag{3.4.3}$$

where $\overline{y} = \frac{1}{n_{samples}} \sum_{i=0}^{n_{samples}-1} y_i$ [12].

From this we see that a model that always predicted the average of the actual outcomes would have an R^2 score of 0, while models that predicted values that were extremely inaccurate would lead to negative R^2 scores.

Although R^2 scores are often recognized from linear regression analyses, based upon the above definition of the coefficient of determination, it is valid for any analysis comparing predicted values to actual values. In scikit-learn, R^2 values can be applied to all regression models, and is the most commonly used evaluation statistic for machine learning regressors.

Although training a model on a portion of the available data and testing it on the other portion is the most straightforward method of assessing a model's performance, it is often impractical when the amount of available data is limited. This was the case for the Sodium Ion Battery Anode data, and therefore using this conventional evaluation method was not an option.

A very common alternative to the conventional validation method of taking a small subset of the available data as the testing set, is cross validation. There are a few different cross validation methods, but the most common in machine learning is k-fold cross validation [33]. In k-fold cross validation, the available data is randomly split into k parts. One of these k parts is set aside as the testing set, and the other $k - 1$ parts are used as training data. Then the trained model is tested on the kth part of the data that was not used for training. This process is repeated $k-1$ more times, with each of the k parts being set aside once for use as the testing set. The k scores resulting from this process are averaged, and this value can be considered a representative value for the model's performance had it been given an external validation set.

K-fold cross validation also maintains its integrity as an evaluation technique when used on larger datasets, so generalizing our programs to multiple datasets, regardless of

size, would be best done using cross validation rather than conventional validation techniques [33]. Using Scikit-learn's built-in k-fold cross validation function, automating the process of evaluating model performance, regardless of dataset variations, was straightforward.

3.4.4 Creating a Datatype

Handling the Anode Performance and Cycling Data

After understanding the basic workflow of a machine learning analysis, the goal was to now find the best machine learning model for a given regression task, along with the features used and the parameters of that best model.

There were slightly different approaches taken for the two types of data: the anode performance data and the cycling data. The different classes of anodes had to be grouped and analyzed separately, but the data was in the same format for all the classes, so whatever program was written for selecting the best regressor for a certain class could be applied to all the classes. To address this simplifying factor, all of the data was formatted such that a list contained the DataFrames for each class. This meant that a for loop in Python could be used to iterate through all the DataFrames in the list and whatever analysis it would perform on one of the DataFrames would be done for the others. As long as the bookkeeping of which features and parameters belong to which model and class of anode was kept straight, this idea would make the program very modular and applicable to more than just the given anode dataset.

The anode performance data was to be used for predicting there continuous quantities, capacity retention, rate retention, and coulombic efficiency on the first cycle, and therefore the data preprocessing for these three different analyses would have to be slightly different. Otherwise, once the outcomes vector and the DataFrame of features had been built properly, the remainder of the process for finding the best machine learning model would be identical.

For the cycling data, since all of the data for a given battery was groupable together, this meant that there was no need to break up the data. Since the program for the anode dataset was designed to only handle lists of DataFrames, it would also be capable of handling a list of just a single DataFrame, and therefore the program could be easily modified

to handle the cycling data as well. Since we were only attempting to predict SOC using the cycling data, the data from the CSV files had to only be preprocessed once before analysis. Once the data had been formatted into an output vector and a DataFrame of features and put into lists, the rest of the analysis would follow the exact same steps as would the anode performance data after being properly formatted.

Steps in the Analysis:

As discussed above, the difference in handling machine learning analyses on different datasets with different outcomes to predict ends after the data has been properly sorted into a list of DataFrames of features and into a list of vectors of outcomes. The steps to reach this point are discussed in the 'Preprocessing the Data' section, with the exception of the need for encoding the categorical data. Since there were clear differences between the way a model would handle one hot encoded data and label encoded data, an analysis was to be done on both forms of encoded data.

After attempting a few analyses of the anode performance data in scikit-learn, I discovered that one hot encoded data was far more unwieldy to handle than label encoded data. With the addition of an enormous number of columns for categorical data, not only did the basic model training take longer than with label encoded data, attempting recursive or iterative procedures during feature or parameter optimization was impractical in terms of both processing power and time needed. In addition to this, since the number of one hot encoded columns changed depending on the class of anode being analyzed (since certain classes had less unique values in a certain category than did other classes), this meant that a comparison of variables that were significant in determining performance was going to be inconsistent. It was decided that Label encoding, taking less time and power to handle, would be the main form of encoding used in this project, with the feature and parameter optimizations only being applied to label encoded data. One hot data was still included in our analyses as a baseline for how well it could perform.

As discussed in the 'Workflow' section, there were two forms of feature optimization and one form of parameter optimization available. This meant that there were a total of 3 optimizations to choose from, and if we expanded to having two possible optimizations, then we were choosing two of the optimizations from a choice of 4 possibilities: RFECV Feature Selection, SelectKBest Feature Selection, Parameter Selection, and no sec-

ond optimization. Since performing both feature selections at once would contribute to a comparison between them this was not a combination analyzed. So there were a total of (4 choose 2) -1 = 5 optimization procedures that could be performed on the label encoded data. As discussed earlier, in some cases the RFECV Feature Selection would not be possible if the model did not assign weights to features/variables. In these cases, 3 optimization procedures were available.

In the case of mixing parameter and feature optimization, since the Parameter selection was tested on the intended feature DataFrame, performing the feature selection first and creating the narrowed down feature DataFrame for the parameter optimization to use led to stronger model results. In some tests attempting the reverse order of parameter optimization first and then choosing features, the performance was typically similar, but in some cases much worse. It was decided that wasting resources on attempting both orders would not be worth the insignificant, if any, improvements in performance. This decision was also supported by the intended purpose of feature selection: removing irrelevant and redundant data before an analysis should lead to better performance- therefore selecting parameters based on a 'flawed' feature set would produce a worse performing model [8].

In addition to the 5 optimization procedures to be performed, two default/control analyses were to be done: Label encoded data using all the initial features, and One hot encoded data using all the initial features. Therefore each machine learning model had 7 maximum possible customizations: Label Encoded Data with all Features, One Hot Encoded Data with all Features, Parameter Optimized with all Features, RFECV Feature Selection, SelectKBest Feature Selection, RFECV Feature Selection and Parameter Optimization, and SelectKBest Feature Selection and Parameter Optimization. For brevity, in the code and results, SelectKBest Feature Selection is simply referred to as Feature Selection One, and RFECV Feature Selection is referred to as Feature Selection Two.

The following table indicates which models were compatible with which customizations:

Regressor Types	Customizations
Linear Support Vector Regression	Label Encoded Data with all Features
Ordinary Least Squares Regression	One Hot Encoded Data with all Features
Ridge Regression	Parameter Optimized with all Features
Lasso Regression	RFECV Feature Selection
Elastic Net Regression	SelectKBest Feature Selection
Orthogonal Matching Pursuit	RFECV & Parameter Optimization
Bayesian Ridge Regression	SelectKBest & Parameter Optimization
Stochastic Gradient Descent Regression	
Decision Tree Regression	
Radial Basis Function Support Vector Regression	Label Encoded Data with all Features
Polynomial Support Vector Regression	One Hot Encoded Data with all Features
Nu Support Vector Regression	Parameter Optimized with all Features
Kernel Ridge Regression	SelectKBest Feature Selection
K Nearest Neighbors Regression	SelectKBest & Parameter Optimization
Supervised Neural Networks Regression	

Creating a Python Class to Output the Best Predictive Models

To synthesize the above workflow into an efficient and easy to use program, the scheme used was to create a preprocessing function that would be unique to each analysis, a general Python class to perform the actual machine learning analyses, and a post-processing function that was general to all of the analyses that would use the Python class to format the results in a presentable format.

The preprocessing functions were made for the SOC, rate retention, capacity retention, and coulombic efficiency analyses by importing the relevant datasets, removing unnecessary data for the analysis, removing columns that had too many missing values, encoding the data, and making sure the data was in a format that could be used by the Python class. The preprocessing functions would be used in the Python class itself as an instance method since its output would need to be called multiple times.

Another way the preprocessing functions were integrated into the workflow was by adding an argument that was a list of column names (other than those that were unnecessary for the analysis) that would be removed during the data processing. This was done since when feature selection was done, scikit-learn had no particular way of only doing an analysis only on a set of desired columns; instead any unwanted columns had to be

manually removed first. By adding this argument to the preprocessing function along with a single line of code to drop the specified columns, the columns could quickly be dropped so the analysis could proceed only on the selected columns.

The output of the preprocessing functions was:

- a list of DataFrames, where each DataFrame contained the Label encoded data for a specific class
- a list of DataFrames, where each DataFrame contained the one hot encoded data for a specific class
- a list of vectors, where each vector was the outcomes for a specific class
- a list of class names, in order to preserve the class ordering
- a list of DataFrames, where each DataFrame contained the original, unencoded data for a specific class. (This was done for any debugging purposes)

All of the four preprocessing functions are included in the Appendix D, Part 3, Section 2.

Once the preprocessing functions were made, a Python class, ModelEval, was built. The benefits of using a Python class rather than a whole bunch of separate functions was that a Python class has attributes that can be easily set, repeatedly called, easily changed, and easily added to. By containing the machine learning analysis within a class, the amount of necessary code could be shortened and organized well. For anyone using this class in the future, it could be easily modified without affecting the rest of the methods and functions inside.

The class ModelEval is initialized with a single argument: the abbreviated name of the desired regression type as a string. This was the only user input required; the class took care of the entire analysis once this was done. Python classes have an initialization method that runs from top to bottom whenever an object of the class is created, so by including calls to functions and methods in the initialization that would perform our analysis, the calculation of the best optimizations became fully automated. As the analysis proceeded it would update attributes of the object, such as the parameters and features used for a certain optimization type. There were attributes also made so that the best performing optimizations, along with their features and parameters, for each class type

(within the data- not to be confused with the Python class) would be returned in an organized DataFrame. The code for ModelEval is included in Appendix D, Part 1.

The class ModelEval, synthesizing the desired workflow, found the best performing optimization types for a single regressor, so a post-processing function was needed to do analyses for multiple regressors. This gave more user control over which regressors would be used, and kept the flow of the analysis simpler and easier to understand than if a class was made to handle multiple regressors at once. The post-processing function was relatively short, and served the function of organizing the DataFrames output from each regressor's ModelEval object. The final output of the post-processing function was a DataFrame that indicated the best performing regressor for each class in the data, along with the optimizations (and their parameters and features) that led to this best performance. The code for the post-processing function is included in Appendix D, Part 3.

Chapter 4

Results and Discussion

4.1 Data Visualizations

4.1.1 The Graphs

The goal of creating interactive graphs for the differential capacity curves was accomplished using the Python programs from Appendix D Parts 1 and 2. These programs used the Numpy, Pandas, and Bokeh libraries in order to process cycling data, alter it to match the required form for plotting, and update based on the chosen user interactions. The output of the programs were HTML files that could be shared and used by anyone with a web-browser, and therefore also accomplished the mission of making the data more accessible and easier to consume.

The following pictures show some images of the two interactive charts, and their interactive functionalities.

One of the major successes of writing these programs is the ability to adapt to any length of cycling data, as long as the variables required for plotting were included. Since the cycling data used for these examples were of a standard format, this means that interactive charts can be created to visualize the cycling performance of any battery instantly.

Another benefit of a modular programming approach was that the resolution of the graphs could be adjusted as desired by the user. If there was a need for high-precision, without regarding to processing power and time, the voltage increments, dV, could be

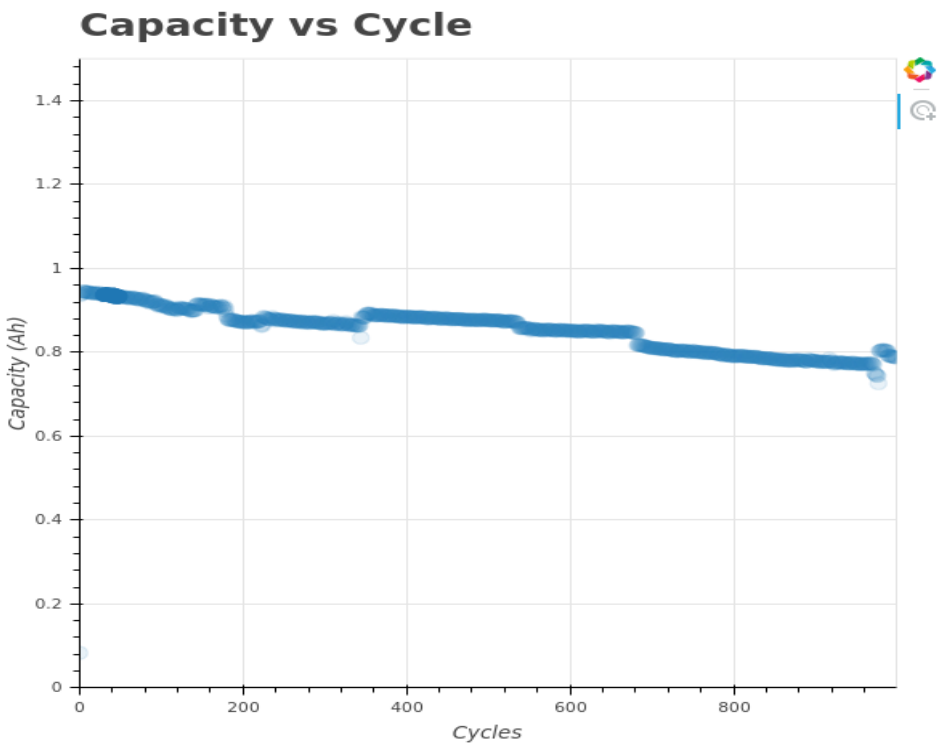

Figure 4.1: Capacity vs Cycle with Clickable Datapoints

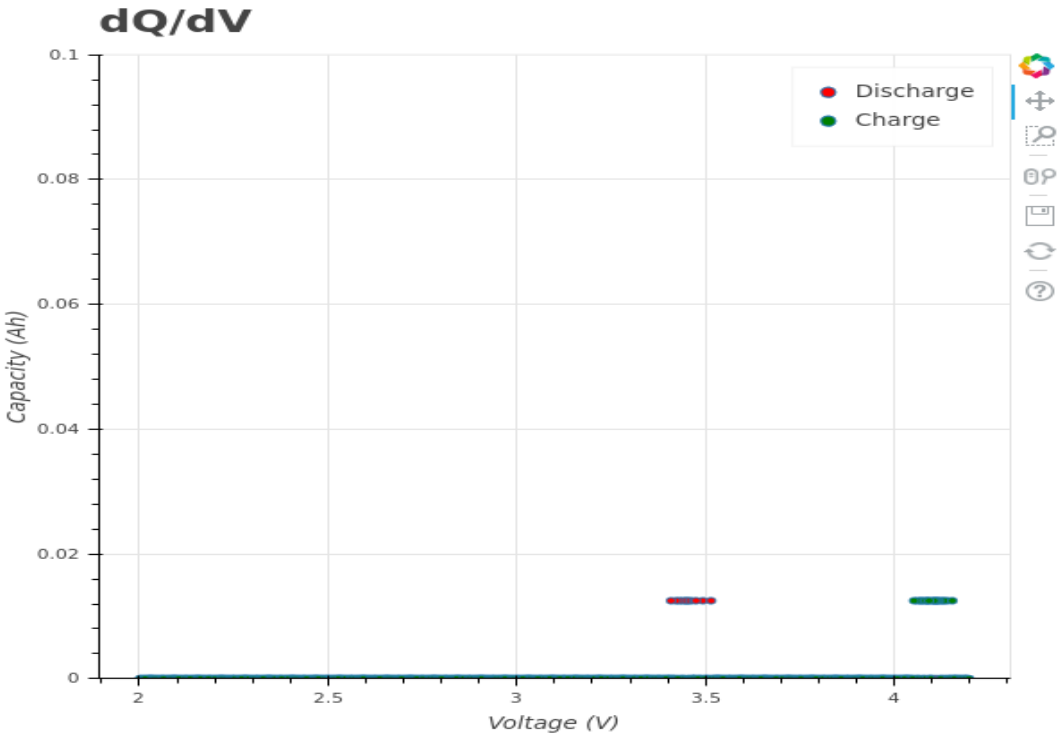

Figure 4.2: dQ/dV vs. Voltage that updates with Capacity vs. Cycle graph

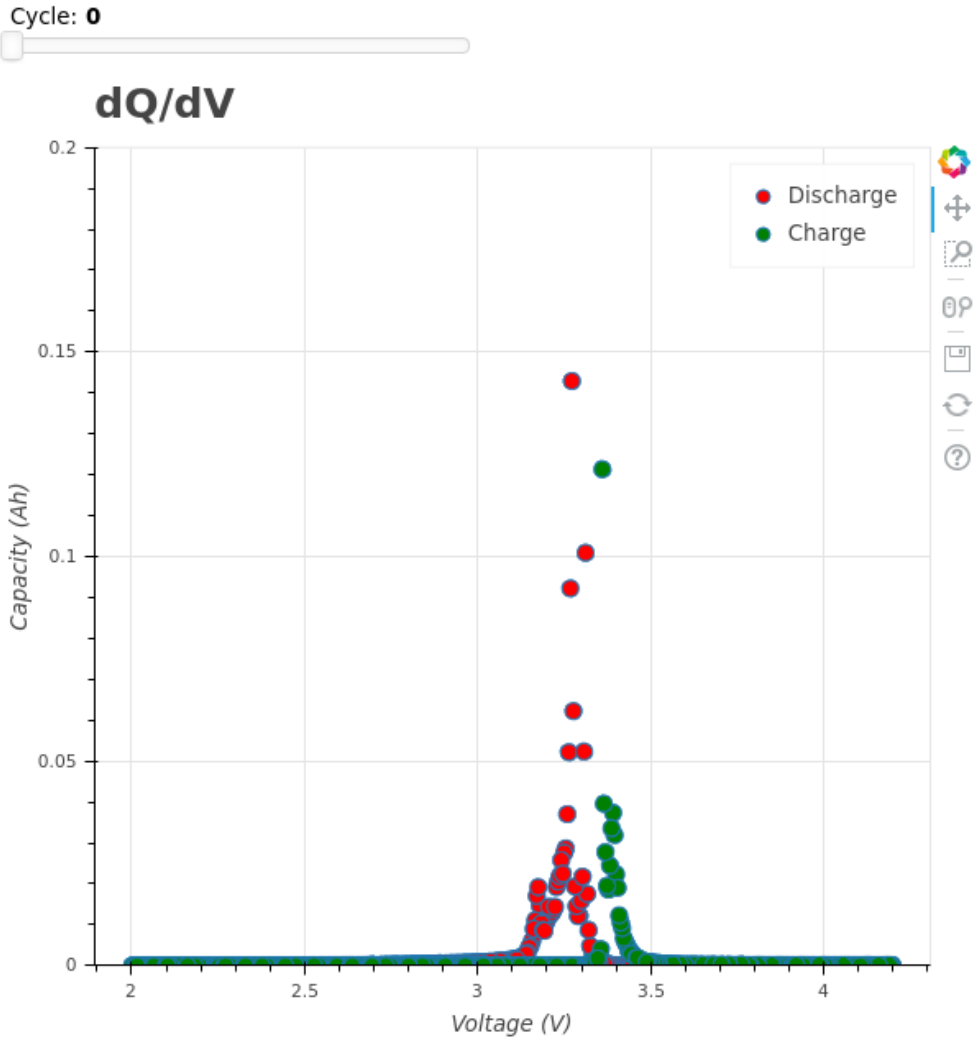

Figure 4.3: dQ/dV vs. Voltage that updates with Slider interaction

made smaller by increasing the number of voltage increments within the given voltage range. For quicker, less detailed results, the number of increments could be decreased, while still yielding a useful data visualization.

4.1.2 Using the Graphs to Understand Battery Behavior

As a battery is cycled and degradation occurs, the transition from a certain regime of degradation to another is of interest. As discussed in [81], during the early stages of cycling the capacity degradation has a linear behavior, while towards later stages this degradation becomes non-linear with a rapid capacity drop and increase in resistance. The linear stage is attributed to SEI growth while the non-linear regime begins as the anode porosity decreases (due to SEI growth and the presence of lithium) and lithium plating undergoes a sharp rate increase. This transition point between regimes is hard to identify using typical data analysis methods as there is variability in its time of occurrence and its manifestation in terms of rate changes. By utilizing the data visualizations it is possible to manually identify locations in the Capacity vs. Cycle graphs where the regime changes seem to occur and identify patterns in the corresponding differential capacity curves to see how the dQ/dV plots are relaying the information about the transition from linear to non-linear behavior. This could be aided with a machine learning model that can recognize trends in the dQ/dV plots as a parameterized version of the Capacity vs. Cycle curve undergoes regime changes.

Other useful aspects of these data visualizations is that they give a battery researcher an instant overview of the electrochemical mechanisms occurring in the battery. The area under the differential capacity curves tells us the total capacity transferred during the cycle, and therefore observing how the curves transform with respect to the voltage axis can immediately describe how much capacity is being lost, as well as for what reasons. When there is a shift in the differential capacity curves peak, we understand from Ohm's law that processes are now occurring at higher voltages due to the increase of internal resistance. Dramatic shifts would indicate large changes in internal resistance, something that would be characteristic of the linear to non-linear regime change. In cases where the capacity curves shape is the same but shorter, this would indicate the depletion of ions necessary for charge transfer and therefore a loss in capacity. Entire shifts of the curves would indicate that lithiation begins at later times due to higher overpotentials, while a broadening of the curve would indicate activity was now occurring at later voltages (due

to electrolyte degradation and anode material transformations).

4.2 Sodium Ion Battery Anode Performance Predictions

4.2.1 Rate Retention Predictions Analysis

Calculating rate retention for the Sodium Ion Battery anodes based on their chemistry features was one of the main goals of this project, especially since the relationship between the anode chemistry and performance was unclear. Various manufacturers and researchers promote the efficacy of certain chemistries and formulations over others, but none have conclusively proven the merits of one type over another. As we saw in Bommier's paper, rate retention comparison across hard, soft, nano, doped, and undoped carbons led to the conclusion that the rate retention did not vary to an appreciable degree based on the carbon formulation [34]. We hoped that a machine learning model that accurately predicted rate retention would elucidate some of the major factors that went into determining rate retention, other than the electrochemical details at a microscopic level.

From the beginning of this analysis, the relative lack of data regarding rate retention for these NIB anodes was noted. Although the manually collected data numbered at around 2500, since degradation and fading mechanisms in different classes of anodes differed, a model could only be trained on a single class's data, which gave an average of around 185 observations per class. With a limited amount of data to train on, a robust machine learning model for every class was unlikely, but with the power of multiple different regressors, feature selection, and parameter optimization, it was possible that certain models could identify patterns indiscernible to humans, and reveal some interesting characteristics of the NIB anodes.

Due to the lack of data, certain regressors could not be used for this analysis. Polynomial Support Vector Regression, although a seemingly feasible implementation of a SVM, took over fifteen hours for a single run through our Python class, and we decided to omit it from the rest of the analysis because of its time consuming behavior and poor results relative to other regressors. The K Nearest Neighbors Regression was not able to perform its analysis either since with only, at best, a couple hundred datapoints involved,

attempting to unify the data from nearest neighbors was impractical. Scikit-learn's function implementing the algorithm would routinely crash because of a lack of nearest neighbor datapoints within the default threshold. When using a Supervised Neural Networks Regression, the complexity of our dataset meant that there were multiple layers, with different learning rates, and therefore the gradients (which are products of gradients of other layers) in certain layers would either become extremely close to 0 or grow quickly towards infinity [59]. Remedies to this problem included changing the learning rates or reducing the number of features, but with our already small amount of data and relevant features, we decided to forego using neural networks in this model.

The tables below shows the best performing model or models for each class of anode, the performance of each model for all anode classes, and the number of observations available for each class. The features and parameters used for the best performing model in each class can be found in Appendix A.

Class	Scores	Best Performer	Best Performer	Best Performer
Alloy	0.385	Ridge Regression		
CNF	0.601	Decision Tree Regression		
Co-intercalation	0.267	Lasso Regression	Elastic Net Regression	
Conversion	0.494	Ridge Regression		
Dual Alloy	0.315	Bayesian Ridge Regression		
Expanded Gr	0.175	Kernel Ridge Regression		
Graphene	0.538	Lasso Regression	Elastic Net Regression	
HardCarbon	0.784	Decision Tree Regression		
Hybrid	0.582	Elastic Net Regression	Orthogonal Matching Pursuit	Bayesian Ridge Regression
Insertion	0.491	Ridge Regression	Bayesian Ridge Regression	
Organic	0.605	Ridge Regression		
Porous	0.623	Decision Tree Regression		
SoftCarbon	0.568	Decision Tree Regression		

Table 4.1: Best Performing Algorithm for each Anode Class for Rate Retention

In the remainder of this section there will be a discussion on a few of the interesting results from the machine learning analysis which could lead to some conclusions and insights regarding the relationship between the anode features and the rate retention.

The best score across all the classes of anodes was 0.784, for the Hard Carbon anodes, using Decision Tree regression, and nearly all the estimators performed extremely well on this class. This was not solely due to the amount of observations contained on Hard Carbons, as there were multiple classes that had a greater number of observations but did

Class	SVR_LIN Scores	OLS Scores	RIDGE Scores	LASSO Scores	ENET Scores	OMP Scores	BAYESIAN_RIDGE Scores	SGD Scores	DTREE Scores	SVR_RBF Scores	SVR_NU Scores	KERNEL Scores
Alloy	-8.620	0.283	0.385	0.384	0.384	0.315	0.384	-8.760715e+32	0.340	-0.022	0.006	0.283
CNF	0.273	0.157	0.579	0.587	0.594	0.596	0.567	-5.080591e+07	0.601	0.197	0.264	0.458
Co-intercalation	-22.177	-19.050	-14.767	0.267	0.267	-14.885	-14.887	-4.354753e+08	-1.655	-0.058	-0.001	-14.844
Conversion	-6.950	0.462	0.494	0.483	0.485	0.483	0.484	-8.171330e+05	0.459	0.136	0.145	0.425
Dual Alloy	-6.551	-8.840	0.309	0.284	0.272	0.062	0.315	-9.549533e+06	0.078	0.035	0.047	-2.681
Expanded Gr	-2.256	-0.359	-4.713	-3.695	-3.871	-4.810	-4.732	-8.557661e+29	0.097	-0.370	-0.408	0.175
Graphene	0.009	0.045	0.501	0.538	0.538	0.405	0.537	-8.687978e+32	0.394	0.125	0.178	0.292
HardCarbon	0.669	0.760	0.762	0.760	0.760	0.751	0.762	-6.562038e+05	0.784	0.391	0.429	0.761
Hybrid	-5.716	0.448	0.580	0.581	0.582	0.582	0.582	-7.755648e+32	0.580	0.075	0.097	0.517
Insertion	-9.974	0.488	0.491	0.339	0.366	0.322	0.491	-4.954200e+32	0.422	0.234	0.282	0.434
Organic	0.270	0.250	0.605	0.438	0.440	0.362	0.595	-7.444087e+08	0.417	-0.334	-0.404	0.467
Porous	-9.237	-56.045	0.291	0.389	0.403	0.264	0.407	-8.913627e+06	0.623	0.017	0.022	-3.154
SoftCarbon	-9.397	0.058	0.010	0.541	0.543	-18.928	-11.602	-8.057703e+31	0.568	0.029	0.046	0.472

Table 4.2: Rate Retention Performance of each Regressor across all Anode classes

Class	Counts
Insertion	525
Conversion	450
Hybrid	346
HardCarbon	305
Alloy	282
Graphene	144
CNF	72
Porous	69
Dual Alloy	69
Organic	64
SoftCarbon	53
Co-intercalation	11
Expanded Gr	8

Table 4.3: Number of Rate Retention Observations by Anode Class

not perform as well. Although Decision Tree regression was the best performing model in this class, the Linear Regressors: OLS, Ridge, Lasso, and Elastic Net regressions all were at around a 0.76 coefficient of determination. These regressors were trained and evaluated in a fraction of the time that it took to do the same for Decision Tree regressors, indicating that simpler models still have a lot of value in any data analysis.

The remainder of the high scores above 0.6 were distributed amongst Ridge Regression and Decision Tree regression, and across all the classes of anodes, Decision Tree Regression had the highest average score. Lasso and Elastic Net regression came in a

close 2nd and 3rd, respectively, but all of the linear regressors performed very well, with their scores being dragged down by especially poor performances on the Co-Intercalation and Expanded Graphene anodes, which made sense given that there were only 11 and 8 observations for those two classes.

An interesting observation from the results was also that feature optimization without parameter optimization could replicate the same results as feature optimization and parameter optimization combined. This situation appeared often in the high performing models, indicating that when the features could predict outcomes well, that the parameters could be at their default values and still predict rate retention well. On the other hand, when the features could not predict the outcomes well, parameter optimization gave the best chance for a higher accuracy. In general, parameter optimization did not improve accuracy an appreciable amount when the score was already high; this was an essential lesson in machine learning and data analysis: a model can only be as strong as the information it was given.

In terms of trying to understand which features were the most relevant in predicting Rate Retention, the results were mostly mixed and inconclusive. One of the problems in using data collected from other's research was that the researchers chose certain external factors that added inconsistency in the data that was being trained on. As an example, lower voltage, the user chosen cutoff voltage before ending discharge, was slightly different amongst different observations, but ended up appearing in many of the models as a heavily weighted factor. This told us very little about the electrode chemistry's relationship to the rate retention, as the lower voltage was not a value determined by the electrode chemistry. This meant that even if the model had performed well in predicting the rate retention, it was almost entirely the machine learning model's algorithm and ability to reiterate endlessly that had led to this predictive power, rather than any relationship that had been found between the features and outcome.

In many of the stronger linear models, the proportion of active material and the proportion of binder in the electrode were the most influential in predicting the rate retention. This was a confirmation of notions in electrochemistry as rate retention is highly correlated with the storage mechanisms on the electrode [34]. For example, electrodes that have surface defects and are capable of capacitive surface storage tend to have higher rate retention, so the proportion of active material in the electrode would be relevant in the amount of intercalation or surface storage that could occur. The proportion of active

material in the electrode also relates to electrode density and therefore plays a role in the speed of charge/discharge behavior, which would affect rate retention (since it varies as the current rate) [53] Although this wasn't a groundbreaking discovery, it was a statistical confirmation of concepts that are considered common knowledge in batteries research today.

4.2.2 Coulombic Efficiency at First Cycle & Capacity Retention Predictions

As mentioned in the Experimental Methods section, there was a lack of data regarding long cycling, and therefore it was unlikely that the machine learning models would be able to achieve high accuracy scores for a majority of the anode classes. This was unfortunately the case during the analysis, with only five of the 12 classes even running through our program without raising errors regarding lack of data in some form. These five classes were: Alloys, Conversion type, Dual Alloy, Hybrid, and Insertion anodes. With several of the classes containing only a handful of observations, no information relating the features to the performance characteristics could be gleaned. As was the case during the rate retention analysis, which had far more observations per class, the following three regressors could not be used: Polynomial Support Vector Regression, K Nearest Neighbors Regression, and Supervised Neural Networks Regression.

In the case of capacity retention, the five classes of anodes that did successfully complete the evaluation process all led to horrible coefficient of determination scores, essentially indicating that the models were useless. Due to the lack of data and non-linear relationship between the features and capacity retention, none of the estimators used were able to achieve an acceptable level of accuracy.

The following tables show the best performing models for each class of anode, and the overall performance of each model for capacity retention. The features and parameters used for the best performing model in each class can be found in Appendix B.

For coulombic efficiency at the first cycle, the story was slightly different. Although 3 of the 5 classes performed very poorly, Alloys and Hybrids performed well using Linear SVR (0.720) and Ridge Regression (0.602), respectively. These were the best performing models, but there were also several other models that performed well for these classes.

Class	Scores	Best Performer	Best Performer	Best Performer
Alloy	-4.781	Elastic Net Regression	Radial Basis Function SVR	Nu SVR
Conversion	0.165	Ridge Regression		
Dual Alloy	0.108	Orthogonal Matching Pursuit		
Hybrid	0.155	Bayesian Ridge Regression		
Insertion	-0.318	Nu Support Vector Regression		

Table 4.4: Best Performing Algorithm for each Anode Class for Capacity Retention

Class	SVR_LIN Scores	OLS Scores	RIDGE Scores	LASSO Scores	ENET Scores	OMP Scores	BAYESIAN_RIDGE Scores	SGD Scores	DTREE Scores	SVR_RBF Scores	SVR_NU Scores	KERNEL Scores
Alloy	-5617.109	-7.300	-13.066	-9.141	-4.781	-17.062	-4.859	-8.885173e+33	-23.350	-4.781	-4.781	-5612.136
Conversion	-3.036	-18.423	0.165	0.158	0.156	-2.770	0.155	-9.074050e+31	-0.625	-0.016	0.038	-3.371
Dual Alloy	-72.842	-7.251	-7.225	-4.753	-6.546	0.108	-7.233	-6.274194e+06	-2.421	-0.496	-0.992	-94.860
Hybrid	-5.267	0.025	0.091	0.154	0.154	-0.691	0.155	-9.859788e+32	-1.704	-0.068	-0.054	-3.443
Insertion	-43.997	-6.801	-6.705	-3.480	-3.824	-3.280	-6.802	-6.455222e+33	-5.068	-0.919	-0.318	-9.603

Table 4.5: Capacity Retention Performance of each Regressor across all Anode classes

OLS regression had a score of 0.716 for Alloys, and Lasso & Elastic Net regression models had scores of around 0.563 for Hybrids.

The following tables show the best performing models for each class of anode, and the overall performance of each model for predicting Coulombic Efficiency of the first cycle. The features and parameters used for the best performing model in each class can be found in Appendix C.

Class	Scores	Best Performer
Alloy	0.720	Linear Support Vector Regression
Conversion	-0.349	Elastic Net Regression
Dual Alloy	-2.931	Lasso Regression
Hybrid	0.602	Ridge Regression
Insertion	0.234	Bayesian Ridge Regression

Table 4.6: Best Performing Algorithm for each Anode Class for Coulombic Efficiency

Across the two classes that gave nontrivial results (Alloys and Hybrids), the best performing model was Bayesian Ridge Regression with a score of .511, followed by OLS regression with a score of .476. The other models performed very poorly on at least one of

Class	SVR_LIN Scores	OLS Scores	RIDGE Scores	LASSO Scores	ENET Scores	OMP Scores	BAYESIAN_RIDGE Scores	SGD Scores	DTREE Scores	SVR_RBF Scores	SVR_NU Scores	KERNEL Scores
Alloy	0.720	0.716	-6.026	-3.651	-3.651	-11.017	0.429	-8.689082e+31	-5.795	-3.651	0.319	-2571.159
Conversion	-7.012	-44.745	-2.634	-0.351	-0.349	-0.941	-2.552	-7.102762e+05	-1.715	-0.520	-0.476	-4.557
Dual Alloy	-72.908	-4.032	-3.916	-2.931	-3.063	-6.949	-4.028	-9.043720e+06	-6.964	-6.111	-2.983	-82.782
Hybrid	-1.170	0.236	0.602	0.565	0.562	-0.565	0.594	-9.223060e+31	-1.252	-0.029	0.016	0.185
Insertion	-3.738	-3.241	-2.392	-1.185	-1.537	-1.608	0.234	-2.540917e+33	-1.357	-0.864	-0.620	-4.709

Table 4.7: Coulombic Efficiency Performance of each Regressor across all Anode classes

the two classes and therefore had much lower average scores.

In terms of understanding which features played a role in determining Coulombic efficiency, there were only the two classes of anodes that gave any additional insight. In the Hybrid type anodes, the second element used in the anode played a strong role in determining CE. This can be explained from an electrochemical perspective as the secondary elements involved in reactions, such as oxides or sulfides, affect the charge storing capabilities of the anode and therefore the Coulombic efficiency [28].

For the Alloy type anodes, loading mass played the most important role in determining the CE value. As discussed in [58] and [53], energy density can play a critical role in the loading mass chosen by a researcher in order to determine battery performance. In Alloy electrodes, alloying with energy dense metals is done to attempt to increase energy density, which would be reflected in the loading mass (which is a measure of electrode mass per area of current collector), since for a given mass, more energy dense electrodes will maintain a larger amount of energy for charge and discharge, so researchers can afford to make thinner electrodes. The importance of this feature gave statistical credence to the notion that loading mass plays a non-trivial role in determining performance, and standardizing experiments with respect to loading mass would help characterize the relationship between anode chemistry and performance (without the influence of researcher chosen parameters).

4.3 Cycling Data State of Charge Predictions

In analyzing the cycling data to perform SOC predictions, I was attempting to find a novel solution to an important issue in all batteries. As mentioned in the introductory sections, state of charge essentially is a fuel gauge for the battery, and as batteries become ubiq-

uitous, precisely knowing the charge state is vital. The current methods of calculating SOC are costly to implement, and researchers have attempted to predict SOC in multiple different ways, to varying degrees of success.

Although many of the papers read during this project indicated that SOC predictions using their machine learning models was close to .99 coefficient of determination, in some cases this was a slightly disingenuous way to present results. When performing machine learning on cycling data to predict SOC, we essentially provide all the information necessary to calculate SOC in the form of charge transferred at different times during a cycle. Since the SOC at a given point in a cycle is simply the charge transferred by that point divided by the total charge that would be transferred, using the charge transferred as an input feature when analyzing only a given type of charging regime meant that the results would always be nearly perfect. I was able to replicate this above procedure and receive scores of .996 using simple ordinary least squares regression.

In order to characterize a battery's SOC using a machine learning model, I had to account for multiple charging regimes, at different C rates, using different charging methods. The cycling data include in this analysis was from different C rates and using both constant current charging and constant current-constant voltage discharging. Using charge as an input now would not completely trivialize the problem since charge transfer behaviors are altered during different charging regimes.

A huge limitation to my analysis was that the cycling data numbered in the tens of thousands of observations, and the processing power and memory required to handle that amount of data was enormous. Using any of the regressors other than the linear regressors would take days to complete an analysis, and using the more complex regressors such as neural networks and kernel ridge regression lead to memory overflow errors. This meant that significant computing resources, such as those managed by professional companies would be needed to get a better idea of the true capabilities of machine learning in predicting SOC.

The following tables show the best performing models for the cycling data used, and the overall performance of each model for predicting the state of charge.

All of the regressors shown above were linear regressors, and although most of them performed well in the sense of a statistical analysis, these scores would not be accurate

Class	Scores	Best Performer	Best Performer	Best Performer	Best Performer
A	0.798	Ordinary Least Squares Regression	Ridge Regression	Orthogonal Matching Pursuit	Bayesian Ridge Regression

Table 4.8: Best Performing Algorithm for Predicting State of Charge

Class	OLS Scores	RIDGE Scores	LASSO Scores	ENET Scores	OMP Scores	BAYESIAN_RIDGE Scores
A	0.798	0.798	0.122	0.559	0.798	0.798

Table 4.9: Scores for each Regressor for SOC Predictions

enough during a real life application. In a sense, the coefficient of determination tells us the percentage of the time the model would accurately predict the SOC of the battery. Since the best scoring models had R^2 of $.798$, this means that only around 80% of the time would the correct state of charge be shown. This would be unacceptable for any commercial purpose as consumers expect their fuel gauges/battery meters to be nearly perfect in indicating how much of the total capacity has been used.

In some separate analyses using some of the other regressors that took a long time to run or had memory/processing power related problems, the performances were never higher than those of the linear regressors. K Nearest Neighbors Regression gave a very close score of 0.763 and Neural Networks Regression gave a 0.557, so none of these techniques warranted use over the linear models.

In analyzing which features were the most relevant in calculating SOC, most of these best performing models used exclusively the charge transferred, as expected, to attempt and predict SOC. Since multiple different run types were included, instead of the usual nearly perfect accuracy, the accuracy took a large hit. This led me to look into omitting the charge transferred as an input feature, to see what measure of accuracy machine learning models could obtain without the most telling piece of information.

In the tests ran without using the charge as an input feature, all the models failed to return positive scores, meaning that the data input was essentially useless in trying to predict the state of charge. This was expected from an electrochemical perspective as the only informative continuous parameter passed was the battery voltage, but as we saw in the differential capacity curves, voltage without the capacity would tell us essentially

nothing.

Although this analysis was not able to predict SOC to an acceptable standard, it indicated a lot about the quest to use data analysis to predict a battery's state of charge: in order to do machine learning successfully on a large scale more than just a personal computer is needed, and that even the strongest mathematically formulated models cannot perform better than the data given to it.

Chapter 5

Further Work & Improvements

5.1 Anode Performance Predictions

This project attempted to apply data analysis techniques to problems in batteries research without focusing on the electrochemical phenomena that was the basis of all the patterns and relationships that could be found. This meant that the data had to speak for itself, and especially in the case of the Anode performance data, the researcher error and bias played a large role in the model performance.

There are hundreds of papers that can be crawled to find data, but the criteria for choosing which observations to include and which to ignore are hard to define, especially when there is a limited amount of data (so discrepancies and errors in observations would increase variance greatly). The dataset used in this project opted to be all-inclusive, although some of the research papers may have had questionable methods and results, because it was hoped that the aggregation of enough observations would cancel out the overestimated and underestimated performance parameters. In many cases, this meant that the anode classes with more data led to better predictions from the models, but this was not always the case (as was discussed in the previous chapter). In terms of ascertaining new patterns or relationships, this work provided statistical evidence for common trends such as the effect of loading mass on coulombic efficiency, which was a great display of the power of machine learning in any field.

To improve upon this analysis, more complete data would go a long way in finding the roles played by different factors in determining anode performance. For example,

different columns of data that had too many missing values had to be removed from the analyses entirely, even if they definitely played a role in the battery's electrochemical behavior. This could be improved by doing more thorough research and contacting authors to fill in the blanks, but this may be unfeasible due to the large amount of papers involved. Another technique, from the machine learning angle, would be to use imputation. Imputation is a technique that deals with missing data by replacing it with substitute values, such as the average value of the available data. This may have had the effect of increasing performance by including more relevant parameters, but also may have had a detrimental affect due to the introduction of observations with inaccurate data.

Another improvement to this project would be using computing clusters to run larger scale parameter and feature optimization runs using a larger variety of machine learning models. Time and resources was a limitation in my analyses, and with more processing power, it could be possible to harness the power of all the different sorts of regressors, even in combination, to get better predictions.

5.2 Differential Capacity Curves Analysis

Although the data visualizations are a helpful tool in understanding the regimes of capacity degradation, a useful extension of the data visualizations would be machine learning models that can identify regime changes after being trained on data that is already annotated with the correct transitions and electrochemical behaviors.

This could be done by parameterizing the curves and plots so that a function could estimate the differential capacity curves and therefore lead to an analysis based on the curve characteristics. For example, differential capacity curves can be estimated using Gaussian curves of different parameters, and the derivatives and integrals of these curves are already known parameters. This would lead to fast analysis of what was occurring on the dQ/dV plots, and therefore could possibly characterize a battery's fading behavior without need for much user input.

Bibliography

[1] 1.1. generalized linear models — scikit-learn 0.19.1 documentation. http://scikit-learn.org/stable/modules/linear_model.html. Accessed: 2018-5-9.

[2] 1.10. decision trees — scikit-learn 0.19.1 documentation. http://scikit-learn.org/stable/modules/tree.html. Accessed: 2018-5-9.

[3] 1.13. feature selection — scikit-learn 0.19.1 documentation. http://scikit-learn.org/stable/modules/feature_selection.html. Accessed: 2018-5-6.

[4] 1.5. stochastic gradient descent — scikit-learn 0.19.1 documentation. http://scikit-learn.org/stable/modules/sgd.html. Accessed: 2018-5-9.

[5] 1.6. nearest neighbors — scikit-learn 0.19.1 documentation. http://scikit-learn.org/stable/modules/neighbors.html. Accessed: 2018-5-9.

[6] 3.3. model evaluation: quantifying the quality of predictions — scikit-learn 0.19.1 documentation. http://scikit-learn.org/stable/modules/model_evaluation.html. Accessed: 2018-5-6.

[7] Difference between selecting features based on "f regression" and based on R^2 values? https://stats.stackexchange.com/questions/204141/difference-between-selecting-features-based-on-f-regression-and-based-on-r2. Accessed: 2018-5-6.

[8] How should feature selection and hyperparameter optimization be ordered in the machine learning pipeline? https://stats.stackexchange.com/questions/264533/how-should-feature-selection-and-hyperparameter-optimization-be-ordered-in-the-m. Accessed: 2018-5-6.

[9] Machine learning curse of dimensionality explained? https://stats.stackexchange.com/questions/65379/machine-learning-curse-of-dimensionality-explained. Accessed: 2018-5-6.

[10] NumPy — NumPy. http://www.numpy.org/. Accessed: 2018-5-4.

[11] Python data analysis library — pandas: Python data analysis library. https://pandas.pydata.org/. Accessed: 2018-5-4.

[12] scikit-learn: machine learning in python — scikit-learn 0.19.1 documentation. http://scikit-learn.org/stable/. Accessed: 2018-5-5.

[13] sklearn.feature_selection.f_regression — scikit-learn 0.19.1 documentation. http://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.f_regression.html. Accessed: 2018-5-6.

[14] sklearn.feature_selection.RFECV — scikit-learn 0.19.1 documentation. http://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.RFECV.html. Accessed: 2018-5-6.

[15] sklearn.feature_selection.SelectKBest — scikit-learn 0.19.1 documentation. http://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SelectKBest.html. Accessed: 2018-5-6.

[16] sklearn.linear_model.ElasticNet — scikit-learn 0.19.1 documentation. http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.ElasticNet.html. Accessed: 2018-5-9.

[17] sklearn.linear_model.lasso — scikit-learn 0.19.1 documentation. http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Lasso.html. Accessed: 2018-5-9.

[18] sklearn.linear_model.LinearRegression — scikit-learn 0.19.1 documentation. http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html. Accessed: 2018-5-9.

[19] sklearn.linear_model.ridge — scikit-learn 0.19.1 documentation. http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Ridge.html. Accessed: 2018-5-9.

[20] sklearn.model_selection.GridSearchCV — scikit-learn 0.19.1 documentation. http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html. Accessed: 2018-5-6.

[21] sklearn.preprocessing.LabelEncoder — scikit-learn 0.19.1 documentation. http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html. Accessed: 2018-5-6.

[22] sklearn.preprocessing.OneHotEncoder — scikit-learn 0.19.1 documentation. http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html. Accessed: 2018-5-6.

[23] Support vector machines and regression. https://stats.stackexchange.com/questions/13194/support-vector-machines-and-regression/287855. Accessed: 2018-5-9.

[24] When to use one hot encoding vs LabelEncoder vs DictVectorizor? https://datascience.stackexchange.com/questions/9443/when-to-use-one-hot-encoding-vs-labelencoder-vs-dictvectorizor. Accessed: 2018-5-6.

[25] Mohammad Alipour. Why capacity of the lithium-ion batteries decrease at high current. https://www.researchgate.net/post/Why_capacity_of_the_lithium-ion_batteries_decrease_at_high_current_rates_and_high_temperatures, May 2017. Accessed: 2018-5-7.

[26] J C Álvarez Antón, P J García Nieto, F J de Cos Juez, F Sánchez Lasheras, M González Vega, and M N Roqueñí Gutiérrez. Battery state-of-charge estimator using the SVM technique. *Appl. Math. Model.*, 37(9):6244–6253, May 2013.

[27] Dave Andre, Christian Appel, Thomas Soczka-Guth, and Dirk Uwe Sauer. Advanced mathematical methods of SOC and SOH estimation for lithium-ion batteries. *J. Power Sources*, 224:20–27, February 2013.

[28] Ashish Aphale, Krushangi Maisuria, Manoj K Mahapatra, Angela Santiago, Prabhakar Singh, and Prabir Patra. Hybrid electrodes by In-Situ integration of graphene and Carbon-Nanotubes in polypyrrole for supercapacitors. *Sci. Rep.*, 5:14445, September 2015.

[29] Guangxing Bai, Pingfeng Wang, and Chao Hu. A self-cognizant dynamic system approach for prognostics and health management. *J. Power Sources*, 278:163–174, March 2015.

[30] Anthony Barré, Benjamin Deguilhem, Sébastien Grolleau, Mathias Gérard, Frédéric Suard, and Delphine Riu. A review on lithium-ion battery ageing mechanisms and

estimations for automotive applications. *J. Power Sources*, 241:680–689, November 2013.

[31] M Berecibar, I Gandiaga, I Villarreal, N Omar, J Van Mierlo, and P Van den Bossche. Critical review of state of health estimation methods of li-ion batteries for real applications. *Renewable Sustainable Energy Rev.*, 56:572–587, April 2016.

[32] Maitane Berecibar, Floris Devriendt, Matthieu Dubarry, Igor Villarreal, Noshin Omar, Wouter Verbeke, and Joeri Van Mierlo. Online state of health estimation on NMC cells based on predictive analytics. *J. Power Sources*, 320:239–250, July 2016.

[33] Christopher M Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, April 2011.

[34] Clement Bommier, David Mitlin, and Xiulei Ji. Internal structure - na storage mechanisms - electrochemical performance relations in carbons. *Prog. Mater Sci.*

[35] Bokeh Contributors. Welcome to bokeh — bokeh 0.12.15 documentation. https://bokeh.pydata.org/en/latest/. Accessed: 2018-5-5.

[36] M Cugnet, M Dubarry, and B Y Liaw. SECONDARY BATTERIES – LEAD– ACID SYSTEMS | modeling. In Jürgen Garche, editor, *Encyclopedia of Electrochemical Power Sources*, pages 816–828. Elsevier, Amsterdam, 2009.

[37] Mehmet Ugras Cuma and Tahsin Koroglu. A comprehensive review on estimation strategies used in hybrid and battery electric vehicles. *Renewable Sustainable Energy Rev.*, 42:517–531, February 2015.

[38] Xuanju Dang, Li Yan, Kai Xu, Xiru Wu, Hui Jiang, and Hanxu Sun. Open-Circuit Voltage-Based state of charge estimation of lithium-ion battery using dual neural network fusion battery model. *Electrochim. Acta*, 188:356–366, January 2016.

[39] Alexander Farmann and Dirk Uwe Sauer. A comprehensive review of on-board State-of-Available-Power prediction techniques for lithium-ion batteries in electric vehicles. *J. Power Sources*, 329:123–137, October 2016.

[40] Alexander Farmann, Wladislaw Waag, Andrea Marongiu, and Dirk Uwe Sauer. Critical review of on-board capacity estimation techniques for lithium-ion batteries in electric and hybrid electric vehicles. *J. Power Sources*, 281:114–130, May 2015.

[41] Christian Fleischer, Wladislaw Waag, Hans-Martin Heyn, and Dirk Uwe Sauer. On-line adaptive battery impedance parameter and state estimation considering physical principles in reduced order equivalent circuit battery models: Part 1. requirements, critical review of methods and modeling. *J. Power Sources*, 260:276–291, August 2014.

[42] James A Gilbert, Ilya A Shkrob, and Daniel P Abraham. Transition metal dissolution, ion migration, electrocatalytic reduction and capacity loss in Lithium-Ion full cells. *J. Electrochem. Soc.*, 164(2):A389–A399, January 2017.

[43] Taedong Goh, Minjun Park, Minhwan Seo, Jun Gu Kim, and Sang Woo Kim. Capacity estimation algorithm with a second-order differential voltage curve for li-ion batteries with NMC cathodes. *Energy*, 135:257–268, September 2017.

[44] M A Hannan, M S H Lipu, A Hussain, and A Mohamed. A review of lithium-ion battery state of charge estimation and management system in electric vehicle applications: Challenges and recommendations. *Renewable Sustainable Energy Rev.*, 78:834–854, October 2017.

[45] Terry Hansen and Chia-Jiu Wang. Support vector based battery state of charge estimator. *J. Power Sources*, 141(2):351–358, March 2005.

[46] David Harris and Sarah Harris. *Digital Design and Computer Architecture*. Morgan Kaufmann, 2 edition edition, August 2012.

[47] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition (Springer Series in Statistics)*. Springer, 2nd edition edition.

[48] Zhiwei He, Mingyu Gao, Guojin Ma, Yuanyuan Liu, and Sanxin Chen. Online state-of-health estimation of lithium-ion batteries using dynamic bayesian networks. *J. Power Sources*, 267:576–583, December 2014.

[49] J N Hu, J J Hu, H B Lin, X P Li, C L Jiang, X H Qiu, and W S Li. State-of-charge estimation for battery management system using optimized support vector machine for regression. *J. Power Sources*, 269:682–693, December 2014.

[50] Ala A Hussein. Derivation and comparison of open-loop and closed-loop neural network battery State-of-Charge estimators. *Energy Procedia*, 75:1856–1861, August 2015.

[51] N Kakimoto and K Goto. Capacity-Fading model of Lithium-Ion battery applicable to multicell storage systems. *IEEE Transactions on Sustainable Energy*, 7(1):108–117, January 2016.

[52] Jana Kalawoun, Krystyna Biletska, Frédéric Suard, and Maxime Montaru. From a novel classification of the battery state of charge estimators toward a conception of an ideal one. *J. Power Sources*, 279:694–706, April 2015.

[53] Namhyung Kim, Sujong Chae, Jiyoung Ma, Minseong Ko, and Jaephil Cho. Fast-charging high-energy lithium-ion batteries via implantation of amorphous silicon nanolayer in edge-plane activated graphite anodes. *Nat. Commun.*, 8(1):812, October 2017.

[54] Verena Klass, Mårten Behm, and Göran Lindbergh. A support vector machine-based state-of-health estimation method for lithium-ion batteries under electric vehicle operation. *J. Power Sources*, 270:262–272, December 2014.

[55] Yi Li, Mohamed Abdel-Monem, Rahul Gopalakrishnan, Maitane Berecibar, Elise Nanini-Maury, Noshin Omar, Peter van den Bossche, and Joeri Van Mierlo. A quick on-line state of health estimation method for li-ion battery with incremental capacity curves processed by gaussian filter. *J. Power Sources*, 373:40–53, January 2018.

[56] Languang Lu, Xuebing Han, Jianqiu Li, Jianfeng Hua, and Minggao Ouyang. A review on the key issues for lithium-ion battery management in electric vehicles. *J. Power Sources*, 226:272–288, March 2013.

[57] Christopher Manning. Representations for language: From word embeddings to sentence meanings.

[58] Jože Moškon, Maja Pivko, Robert Dominko, and Miran Gaberšček. Accurate determination of battery electrode loading (mass) from electrochemical impedance spectroscopy. *ECS Electrochemistry Letters*, 4(1):A4–A6, January 2015.

[59] Michael A Nielsen. *Neural networks and deep learning*. Determination Press, 2015.

[60] Gang Ning, Bala Haran, and Branko N Popov. Capacity fade study of lithium-ion batteries cycled at high discharge rates. *J. Power Sources*, 117(1):160–169, May 2003.

[61] Adnan Nuhic, Tarik Terzimehic, Thomas Soczka-Guth, Michael Buchholz, and Klaus Dietmayer. Health diagnosis and remaining useful life prognostics of lithium-ion batteries using data-driven methods. *J. Power Sources*, 239:680–688, October 2013.

[62] Seyed Mohammad Rezvanizaniani, Zongchang Liu, Yan Chen, and Jay Lee. Review and recent advances in battery health monitoring and prognostics technologies for electric vehicle (EV) safety and mobility. *J. Power Sources*, 256:110–124, June 2014.

[63] Kjell Schroder, Judith Alvarado, Thomas A Yersak, Juchuan Li, Nancy Dudney, Lauren J Webb, Ying Shirley Meng, and Keith J Stevenson. The effect of fluoroethylene carbonate as an additive on the solid electrolyte interphase on silicon Lithium-Ion electrodes. *Chem. Mater.*, 27(16):5531–5542, August 2015.

[64] Mehrnoosh Shahriari and Mohammad Farrokhi. State-of-Charge estimation of VRLA batteries using neural networks and extended kalman filter. *IFAC Proceedings Volumes*, 43(22):52–56, January 2010.

[65] Hanmin Sheng and Jian Xiao. Electric vehicle state of charge estimation: Nonlinear correlation and fuzzy support vector machine. *J. Power Sources*, 281:131–137, May 2015.

[66] Tal Sholklapper. Electrochemistry data hidden in your time series data: dQ/dV. https://www.voltaiq.com/blog/electrochemistry-data-hidden-in-your-time-series-data-dq/dv. Accessed: 2018-5-9.

[67] Chelsea Snyder. *The effects of charge/discharge rate on capacity fade of lithium ion batteries*. 2016.

[68] Surendar V., Mohankumar V., Anand S., and Vadana D Prasanna. Estimation of state of charge of a lead acid battery using support vector regression. *Procedia Technology*, 21:264–270, January 2015.

[69] Vivek Tiwari. How changing the c-rate affects the battery capacity? https://www.researchgate.net/post/How_changing_the_C-rate_affects_the_battery_Capacity, September 2015. Accessed: 2018-5-7.

[70] Soichiro Torai, Masaru Nakagomi, Satoshi Yoshitake, Shuichiro Yamaguchi, and Noboru Oyama. State-of-health estimation of LiFePO4/graphite batteries based on a model using differential capacity. *J. Power Sources*, 306:62–69, February 2016.

[71] Jake VanderPlas. *Python Data Science Handbook: Essential Tools for Working with Data*. O'Reilly Media, 1 edition edition, December 2016.

[72] J Vetter, P Novák, M R Wagner, C Veit, K-C Möller, J O Besenhard, M Winter, M Wohlfahrt-Mehrens, C Vogler, and A Hammouche. Ageing mechanisms in lithium-ion batteries. *J. Power Sources*, 147(1):269–281, September 2005.

[73] Wladislaw Waag, Christian Fleischer, and Dirk Uwe Sauer. Critical review of the methods for monitoring of lithium-ion batteries in electric and hybrid vehicles. *J. Power Sources*, 258:321–339, July 2014.

[74] Caihao Weng, Yujia Cui, Jing Sun, and Huei Peng. On-board state of health monitoring of lithium-ion batteries using incremental capacity analysis with support vector regression. *J. Power Sources*, 235:36–44, August 2013.

[75] D N Wong, D A Wetz, A M Mansour, and J M Heinzel. The influence of high C rate pulsed discharge on lithium-ion battery cell degradation. In *2015 IEEE Pulsed Power Conference (PPC)*, pages 1–6, May 2015.

[76] Ji Wu, Yujie Wang, Xu Zhang, and Zonghai Chen. A novel state of health estimation method of li-ion battery using group method of data handling. *J. Power Sources*, 327:457–464, September 2016.

[77] Long Xu, Junping Wang, and Quanshi Chen. Kalman filtering state of charge estimation for battery management system based on a stochastic fuzzy neural network battery model. *Energy Convers. Manage.*, 53(1):33–39, January 2012.

[78] Duo Yang, Yujie Wang, Rui Pan, Ruiyang Chen, and Zonghai Chen. State-of-health estimation for the lithium-ion battery based on support vector regression. *Appl. Energy*, August 2017.

[79] Fangfang Yang, Dong Wang, Yang Zhao, Kwok-Leung Tsui, and Suk Joo Bae. A study of the relationship between coulombic efficiency and capacity degradation of commercial lithium-ion batteries. *Energy*, 145:486–495, February 2018.

[80] Fangfang Yang, Yinjiao Xing, Dong Wang, and Kwok-Leung Tsui. A comparative study of three model-based algorithms for estimating state-of-charge of lithium-ion batteries under a new combined dynamic loading profile. *Appl. Energy*, 164:387–399, February 2016.

[81] Xiao Guang Yang, Yongjun Leng, Guangsheng Zhang, Shanhai Ge, and Chao Yang Wang. Modeling of lithium plating induced aging of lithium-ion batteries: Transition from linear to nonlinear aging. *Journal of Power Sources*, 360:28–40, 1 2017.

[82] Gae-Won You, Sangdo Park, and Dukjin Oh. Real-time state-of-health estimation for electric vehicle batteries: A data-driven approach. *Appl. Energy*, 176:92–103, August 2016.

[83] D Zhang, B S Haran, A Durairajan, R E White, Y Podrazhansky, and B N Popov. Studies on capacity fade of lithium-ion batteries. *J. Power Sources*, 91(2):122–129, December 2000.

[84] Jingliang Zhang and Jay Lee. A review on prognostics and health monitoring of li-ion battery. *J. Power Sources*, 196(15):6007–6014, August 2011.

[85] Sheng Shui Zhang. The effect of the charging protocol on the cycle life of a li-ion battery. *J. Power Sources*, 161(2):1385–1391, October 2006.

[86] Yuejiu Zheng, Minggao Ouyang, Xuebing Han, Languang Lu, and Jianqiu Li. Investigating the error sources of the online state of charge estimation methods for lithium-ion batteries in electric vehicles. *J. Power Sources*, 377:161–188, February 2018.

Appendix A

Performance of Models in Rate Retention Predictions

A.1 Alloy

	Ridge Regression 0.385
	Parameter and Feature One
0	current_rate
1	capacity
2	carbon_support
3	active
4	binder

	Ridge Regression 0.385
	Parameter and Feature One
alpha	1000
copy_X	True
fit_intercept	True
max_iter	None
normalize	False
random_state	None
solver	auto
tol	0.001

A.2 Lead Acid

	Decision Tree Regression 0.601
	Feature Two Only
0	current_rate
1	capacity
2	lower voltage
3	upper voltage
4	carbon_support
5	electrolyte_salt
6	electrolyte_solvent
7	binder_type
8	electrode_composition

	Decision Tree Regression 0.601
	Feature Two Only
criterion	mse
max_depth	None
max_features	None
max_leaf_nodes	None
min_impurity_decrease	0
min_impurity_split	None
min_samples_leaf	1
min_samples_split	2
min_weight_fraction_leaf	0
presort	False
random_state	None
splitter	best

A.3 Co-Intercalation

	Elastic Net Regression 0.267		
	Parameter Only	**Parameter and Feature One**	**Parameter and Feature Two**
0	material	current_rate	class
1	current_rate	capacity	material
2	capacity	lower voltage	current_rate
3	lower voltage	upper voltage	capacity
4	upper voltage	active	lower voltage
5	carbon_support		upper voltage
6	electrolyte_salt		carbon_support
7	electrolyte_solvent		electrolyte_salt
8	binder_type		electrolyte_solvent
9	electrode_composition		binder_type
10	active		electrode_composition
11	binder		active
12			binder

	Elastic Net Regression 0.267		
	Parameter Only	**Parameter and Feature One**	**Parameter and Feature Two**
alpha	20	20	20
copy_X	True	True	True
fit_intercept	True	True	True
l1_ratio	1	1	1
max_iter	1000	1000	1000
normalize	False	False	False
positive	False	False	False
precompute	False	False	False
selection	cyclic	cyclic	cyclic
tol	0.0001	0.0001	0.0001
warm_start	False	False	False

	Lasso Regression 0.267		
	Parameter Only	**Parameter and Feature One**	**Parameter and Feature Two**
0	material	current_rate	class
1	current_rate	capacity	material
2	capacity	lower voltage	current_rate
3	lower voltage	upper voltage	capacity
4	upper voltage	active	lower voltage
5	carbon_support		upper voltage
6	electrolyte_salt		carbon_support
7	electrolyte_solvent		electrolyte_salt
8	binder_type		electrolyte_solvent
9	electrode_composition		binder_type
10	active		electrode_composition
11	binder		active
12			binder

	Lasso Regression 0.267		
	Parameter Only	Parameter and Feature One	Parameter and Feature Two
alpha	20	20	20
copy_X	True	True	True
fit_intercept	True	True	True
max_iter	1000	1000	1000
normalize	False	False	False
positive	False	False	False
precompute	False	False	False
selection	cyclic	cyclic	cyclic
tol	0.0001	0.0001	0.0001
warm_start	False	False	False

A.4 Conversion

	Ridge Regression 0.494	
	Parameter Only	Parameter and Feature Two
0	material	material
1	current_rate	current_rate
2	capacity	capacity
3	lower voltage	lower voltage
4	upper voltage	upper voltage
5	carbon_support	carbon_support
6	electrolyte_salt	electrolyte_salt
7	electrolyte_solvent	electrolyte_solvent
8	binder_type	binder_type
9	electrode_composition	electrode_composition
10	active	active
11	binder	binder

	Ridge Regression 0.494	
	Parameter Only	**Parameter and Feature Two**
alpha	1000	1000
copy_X	True	True
fit_intercept	True	True
max_iter	None	None
normalize	False	False
random_state	None	None
solver	auto	auto
tol	0.001	0.001

A.5 Dual Alloy

	Bayesian Ridge Regression 0.315
	Parameter and Feature Two
0	material
1	current_rate
2	capacity
3	upper voltage
4	carbon_support
5	electrolyte_salt
6	electrolyte_solvent
7	binder_type
8	electrode_composition
9	active

	Bayesian Ridge Regression 0.315
	Parameter and Feature Two
alpha_1	1e-06
alpha_2	1
compute_score	False
copy_X	True
fit_intercept	True
lambda_1	1
lambda_2	1e-06
n_iter	300
normalize	False
tol	0.001
verbose	False

A.6 Expanded Graphene

	Kernel Ridge Regression 0.175
	Label Encoding
0	material
1	current_rate
2	capacity
3	lower voltage
4	upper voltage
5	carbon_support
6	electrolyte_salt
7	electrolyte_solvent
8	binder_type
9	electrode_composition
10	active
11	binder

	Kernel Ridge Regression 0.175
	Label Encoding
alpha	1
coef0	1
degree	3
gamma	None
kernel	linear
kernel_params	None

A.7 Graphene

	Elastic Net Regression 0.538		
	Parameter Only	Parameter and Feature One	Parameter and Feature Two
0	material	current_rate	class
1	current_rate	capacity	material
2	capacity	lower voltage	current_rate
3	lower voltage	carbon_support	capacity
4	upper voltage	binder	lower voltage
5	carbon_support		upper voltage
6	electrolyte_salt		carbon_support
7	electrolyte_solvent		electrolyte_salt
8	binder_type		electrolyte_solvent
9	electrode_composition		binder_type
10	active		electrode_composition
11	binder		active
12			binder

	Elastic Net Regression 0.538		
	Parameter Only	**Parameter and Feature One**	**Parameter and Feature Two**
alpha	5	5	5
copy_X	True	True	True
fit_intercept	True	True	True
l1_ratio	0.3	0.3	0.3
max_iter	1000	1000	1000
normalize	False	False	False
positive	False	False	False
precompute	False	False	False
selection	cyclic	cyclic	cyclic
tol	0.0001	0.0001	0.0001
warm_start	False	False	False

	Lasso Regression 0.538		
	Parameter Only	**Parameter and Feature One**	**Parameter and Feature Two**
0	material	current_rate	class
1	current_rate	capacity	material
2	capacity	lower voltage	current_rate
3	lower voltage	carbon_support	capacity
4	upper voltage	binder	lower voltage
5	carbon_support		upper voltage
6	electrolyte_salt		carbon_support
7	electrolyte_solvent		electrolyte_salt
8	binder_type		electrolyte_solvent
9	electrode_composition		binder_type
10	active		electrode_composition
11	binder		active
12			binder

	Lasso Regression 0.538		
	Parameter Only	**Parameter and Feature One**	**Parameter and Feature Two**
alpha	1.4	1.4	1.4
copy_X	True	True	True
fit_intercept	True	True	True
max_iter	1000	1000	1000
normalize	False	False	False
positive	False	False	False
precompute	False	False	False
selection	cyclic	cyclic	cyclic
tol	0.0001	0.0001	0.0001
warm_start	False	False	False

A.8 Hard Carbon

	Decision Tree Regression 0.784
	One Hot Encoding
0	material
1	current_rate
2	capacity
3	lower voltage
4	upper voltage
5	carbon_support
6	electrolyte_salt
7	electrolyte_solvent
8	binder_type
9	electrode_composition
10	active
11	binder

	Decision Tree Regression 0.784
	One Hot Encoding
criterion	mse
max_depth	None
max_features	None
max_leaf_nodes	None
min_impurity_decrease	0
min_impurity_split	None
min_samples_leaf	1
min_samples_split	2
min_weight_fraction_leaf	0
presort	False
random_state	None
splitter	best

A.9 Hybrid

	Bayesian Ridge Regression 0.582		
	Parameter Only	**Parameter and Feature One**	**Parameter and Feature Two**
0	material	material	class
1	current_rate	current_rate	material
2	capacity	capacity	current_rate
3	lower voltage	active	capacity
4	upper voltage	binder	lower voltage
5	carbon_support		upper voltage
6	electrolyte_salt		carbon_support
7	electrolyte_solvent		electrolyte_salt
8	binder_type		electrolyte_solvent
9	electrode_composition		binder_type
10	active		electrode_composition
11	binder		active
12			binder

	Bayesian Ridge Regression 0.582		
	Parameter Only	**Parameter and Feature One**	**Parameter and Feature Two**
alpha_1	1e-06	1e-06	1e-06
alpha_2	1	1	1
compute_score	False	False	False
copy_X	True	True	True
fit_intercept	True	True	True
lambda_1	1	1	1
lambda_2	1e-06	1e-06	1e-06
n_iter	300	300	300
normalize	False	False	False
tol	0.001	0.001	0.001
verbose	False	False	False

	Elastic Net Regression 0.582			
	One Hot Encoding	**Label Encoding**	**Feature One Only**	**Feature Two Only**
0	material	material	material	class
1	current_rate	current_rate	current_rate	material
2	capacity	capacity	capacity	current_rate
3	lower voltage	lower voltage	active	capacity
4	upper voltage	upper voltage	binder	lower voltage
5	carbon_support	carbon_support		upper voltage
6	electrolyte_salt	electrolyte_salt		carbon_support
7	electrolyte_solvent	electrolyte_solvent		electrolyte_salt
8	binder_type	binder_type		electrolyte_solvent
9	electrode_composition	electrode_composition		binder_type
10	active	active		electrode_composition
11	binder	binder		active
12				binder

	Elastic Net Regression 0.582			
	One Hot Encoding	Label Encoding	Feature One Only	Feature Two Only
alpha	1	1	1	1
copy_X	True	True	True	True
fit_intercept	True	True	True	True
l1_ratio	0.5	0.5	0.5	0.5
max_iter	1000	1000	1000	1000
normalize	False	False	False	False
positive	False	False	False	False
precompute	False	False	False	False
selection	cyclic	cyclic	cyclic	cyclic
tol	0.0001	0.0001	0.0001	0.0001
warm_start	False	False	False	False

	Orthogonal Matching Pursuit 0.582	
	Parameter Only	Parameter and Feature One
0	material	material
1	current_rate	current_rate
2	capacity	capacity
3	lower voltage	active
4	upper voltage	binder
5	carbon_support	
6	electrolyte_salt	
7	electrolyte_solvent	
8	binder_type	
9	electrode_composition	
10	active	
11	binder	
12		

	Orthogonal Matching Pursuit 0.582	
	Parameter Only	Parameter and Feature One
fit_intercept	True	True
n_nonzero_coefs	2	2
normalize	True	True
precompute	auto	auto

A.10 Insertion

	Bayesian Ridge Regression 0.491	
	Feature One Only	Parameter and Feature One
0	current_rate	current_rate
1	capacity	capacity
2	lower voltage	lower voltage
3	electrolyte_salt	electrolyte_salt
4	active	active

	Bayesian Ridge Regression 0.491	
	Feature One Only	**Parameter and Feature One**
alpha_1	1e-06	1e-06
alpha_2	1e-06	1
compute_score	False	False
copy_X	True	True
fit_intercept	True	True
lambda_1	1e-06	1
lambda_2	1e-06	0.1
n_iter	300	300
normalize	False	False
tol	0.001	0.001
verbose	False	False

	Ridge Regression 0.491	
	Feature One Only	**Parameter and Feature One**
0	current_rate	current_rate
1	capacity	capacity
2	lower voltage	lower voltage
3	electrolyte_salt	electrolyte_salt
4	active	active

	Ridge Regression 0.491	
	Feature One Only	**Parameter and Feature One**
alpha	1	1.3
copy_X	True	True
fit_intercept	True	True
normalize	False	False
solver	auto	auto
tol	0.001	0.001

A.11 Organic

	Ridge Regression 0.605
	Parameter and Feature One
0	material
1	current_rate
2	capacity
3	lower voltage
4	binder

	Ridge Regression 0.605
	Parameter and Feature One
alpha	0.5
copy_X	True
fit_intercept	True
max_iter	None
normalize	False
random_state	None
solver	auto
tol	0.001

A.12 Porous

	Decision Tree Regression 0.623
	Parameter and Feature Two
0	current_rate

	Decision Tree Regression 0.623
	Parameter and Feature Two
criterion	mae
max_depth	None
max_features	None
max_leaf_nodes	None
min_impurity_decrease	0
min_impurity_split	None
min_samples_leaf	1
min_samples_split	2
min_weight_fraction_leaf	0
presort	False
random_state	None
splitter	best

A.13 Soft Carbon

	Decision Tree Regression 0.568
	Parameter and Feature Two
0	current_rate
1	capacity
2	upper voltage
3	carbon_support
4	electrode_composition

	Decision Tree Regression 0.568
	Parameter and Feature Two
criterion	mse
max_depth	None
max_features	None
max_leaf_nodes	None
min_impurity_decrease	0
min_impurity_split	None
min_samples_leaf	1
min_samples_split	2
min_weight_fraction_leaf	0
presort	False
random_state	None
splitter	best

Appendix B

Performance of Models in Capacity Retention Predictions

B.1 Alloy

	Elastic Net Regression -4.781		
	Parameter Only	**Parameter and Feature One**	**Parameter and Feature Two**
0	material	material	class
1	category_2	category_2	material
2	CE_first_cycle	capacity_at_end	category_2
3	loading_mass (mg/cm2)	lower voltage	CE_first_cycle
4	cycles	FEC_addition	loading_mass (mg/cm2)
5	current_long_cycling		cycles
6	capacity_at_end		current_long_cycling
7	lower voltage		capacity_at_end
8	upper voltage		lower voltage
9	synthesis		upper voltage
10	carbon_support		synthesis
11	electrolyte_salt		carbon_support
12	electrolyte_solvent		electrolyte_salt
13	FEC_addition		electrolyte_solvent
14	binder_type		FEC_addition
15	electrode_composition		binder_type
16	active		electrode_composition
17	conductive		active
18	binder		conductive
19			binder

	Elastic Net Regression -4.781		
	Parameter Only	**Parameter and Feature One**	**Parameter and Feature Two**
alpha	5	5	5
copy_X	True	True	True
fit_intercept	True	True	True
l1_ratio	0.6	0.6	0.6
max_iter	1000	1000	1000
normalize	False	False	False
positive	False	False	False
precompute	False	False	False
selection	cyclic	cyclic	cyclic
tol	0.0001	0.0001	0.0001
warm_start	False	False	False

	Nu Support Vector Regression -4.781				
	One Hot Encoding	**Label Encoding**	**Parameter Only**	**Feature One Only**	**Parameter and Feature One**
0	material	material	material	material	material
1	category_2	category_2	category_2	category_2	category_2
2	CE_first_cycle	CE_first_cycle	CE_first_cycle	capacity_at_end	capacity_at_end
3	loading_mass (mg/cm2)	loading_mass (mg/cm2)	loading_mass (mg/cm2)	lower voltage	lower voltage
4	cycles	cycles	cycles	FEC_addition	FEC_addition
5	current_long_cycling	current_long_cycling	current_long_cycling		
6	capacity_at_end	capacity_at_end	capacity_at_end		
7	lower voltage	lower voltage	lower voltage		
8	upper voltage	upper voltage	upper voltage		
9	synthesis	synthesis	synthesis		
10	carbon_support	carbon_support	carbon_support		
11	electrolyte_salt	electrolyte_salt	electrolyte_salt		
12	electrolyte_solvent	electrolyte_solvent	electrolyte_solvent		
13	FEC_addition	FEC_addition	FEC_addition		
14	binder_type	binder_type	binder_type		
15	electrode_composition	electrode_composition	electrode_composition		
16	active	active	active		
17	conductive	conductive	conductive		
18	binder	binder	binder		
19					

	Nu Support Vector Regression -4.781				
	One Hot Encoding	Label Encoding	Parameter Only	Feature One Only	Parameter and Feature One
C	1	1	500	1	500
cache_size	200	200	200	200	200
coef0	0	0	0	0	0
degree	3	3	3	3	3
gamma	auto	auto	auto	auto	auto
kernel	rbf	rbf	rbf	rbf	rbf
max_iter	-1	-1	-1	-1	-1
nu	0.5	0.5	0.6	0.5	0.6
shrinking	True	True	True	True	True
tol	0.001	0.001	0.001	0.001	0.001
verbose	False	False	False	False	False

	Radial Basis Function Support Vector Regression -4.781				
	One Hot Encoding	Label Encoding	Parameter Only	Feature One Only	Parameter and Feature One
0	material	material	material	material	material
1	category_2	category_2	category_2	category_2	category_2
2	CE_first_cycle	CE_first_cycle	CE_first_cycle	capacity_at_end	capacity_at_end
3	loading_mass (mg/cm2)	loading_mass (mg/cm2)	loading_mass (mg/cm2)	lower voltage	lower voltage
4	cycles	cycles	cycles	FEC_addition	FEC_addition
5	current_long_cycling	current_long_cycling	current_long_cycling		
6	capacity_at_end	capacity_at_end	capacity_at_end		
7	lower voltage	lower voltage	lower voltage		
8	upper voltage	upper voltage	upper voltage		
9	synthesis	synthesis	synthesis		
10	carbon_support	carbon_support	carbon_support		
11	electrolyte_salt	electrolyte_salt	electrolyte_salt		
12	electrolyte_solvent	electrolyte_solvent	electrolyte_solvent		
13	FEC_addition	FEC_addition	FEC_addition		
14	binder_type	binder_type	binder_type		
15	electrode_composition	electrode_composition	electrode_composition		
16	active	active	active		
17	conductive	conductive	conductive		
18	binder	binder	binder		
19					

	Radial Basis Function Support Vector Regression -4.781				
	One Hot Encoding	Label Encoding	Parameter Only	Feature One Only	Parameter and Feature One
C	1	1	1	1	1
cache_size	200	200	200	200	200
coef0	0	0	0	0	0
degree	3	3	3	3	3
epsilon	0.1	0.1	0	0.1	0
gamma	auto	auto	auto	auto	auto
kernel	rbf	rbf	rbf	rbf	rbf
max_iter	-1	-1	-1	-1	-1
shrinking	True	True	True	True	True
tol	0.001	0.001	0.001	0.001	0.001
verbose	False	False	False	False	False

B.2 Conversion

	Ridge Regression 0.165
	Parameter and Feature One
0	category_2
1	capacity_at_end
2	FEC_addition
3	electrode_composition
4	conductive

	Ridge Regression 0.165
	Parameter and Feature One
alpha	300
copy_X	True
fit_intercept	True
max_iter	None
normalize	False
random_state	None
solver	auto
tol	0.001

B.3 Dual Alloy

	Orthogonal Matching Pursuit 0.108
	Parameter and Feature One
0	current_long_cycling
1	capacity_at_end
2	FEC_addition
3	active
4	conductive

	Orthogonal Matching Pursuit 0.108
	Parameter and Feature One
fit_intercept	True
n_nonzero_coefs	2
normalize	True
precompute	auto
tol	None

B.4 Hybrid

	Bayesian Ridge Regression 0.155
	Parameter and Feature Two
0	capacity_at_end

	Bayesian Ridge Regression 0.155
	Parameter and Feature Two
alpha_1	1
alpha_2	1e-06
compute_score	False
copy_X	True
fit_intercept	True
lambda_1	1e-06
lambda_2	1
n_iter	300
normalize	False
tol	0.001
verbose	False

B.5 Insertion

	Nu Support Vector Regression -0.318
	Feature One Only
0	electrolyte_salt
1	electrolyte_solvent
2	binder_type
3	active
4	conductive

	Nu Support Vector Regression -0.318
	Feature One Only
C	1
cache_size	200
coef0	0
degree	3
gamma	auto
kernel	rbf
max_iter	-1
nu	0.5
shrinking	True
tol	0.001
verbose	False

Appendix C

Performance of Models in Coulombic Efficiency Predictions

C.1 Alloy

	Linear Support Vector Regression 0.72
	Parameter and Feature One
0	loading_mass (mg/cm2)
1	upper voltage
2	binder_type
3	conductive
4	binder

	Linear Support Vector Regression 0.72
	Parameter and Feature One
0	loading_mass (mg/cm2)
1	upper voltage
2	binder_type
3	conductive
4	binder

C.2 Conversion

	Elastic Net Regression -0.349
	Feature One Only
0	category_2
1	current_long_cycling
2	carbon_support
3	electrolyte_solvent
4	binder_type

	Elastic Net Regression -0.349
	Feature One Only
alpha	1
copy_X	True
fit_intercept	True
l1_ratio	0.5
max_iter	1000
normalize	False
positive	False
precompute	False
random_state	None
selection	cyclic
tol	0.0001
warm_start	False

C.3 Dual Alloy

	Lasso Regression -2.931			
	One Hot Encoding	**Label Encoding**	**Feature One Only**	**Feature Two Only**
0	material	material	category_2	class
1	category_2	category_2	current_long_cycling	material
2	loading_mass (mg/cm2)	loading_mass (mg/cm2)	synthesis	category_2
3	current_long_cycling	current_long_cycling	carbon_support	loading_mass (mg/cm2)
4	capacity_at_end	capacity_at_end	electrolyte_solvent	current_long_cycling
5	capacity_retention	capacity_retention	NaN	capacity_at_end
6	lower voltage	lower voltage	NaN	capacity_retention
7	upper voltage	upper voltage	NaN	lower voltage
8	synthesis	synthesis	NaN	upper voltage
9	carbon_support	carbon_support	NaN	synthesis
10	electrolyte_salt	electrolyte_salt	NaN	carbon_support
11	electrolyte_solvent	electrolyte_solvent	NaN	electrolyte_salt
12	FEC_addition	FEC_addition	NaN	electrolyte_solvent
13	binder_type	binder_type	NaN	FEC_addition
14	electrode_composition	electrode_composition	NaN	binder_type
15	active	active	NaN	electrode_composition
16	conductive	conductive	NaN	active
17	binder	binder	NaN	conductive
18	NaN	NaN	NaN	binder

	Lasso Regression -2.931			
	One Hot Encoding	Label Encoding	Feature One Only	Feature Two Only
alpha	1	1	1	1
copy_X	True	True	True	True
fit_intercept	True	True	True	True
max_iter	1000	1000	1000	1000
normalize	False	False	False	False
positive	False	False	False	False
precompute	False	False	False	False
random_state	None	None	None	None
selection	cyclic	cyclic	cyclic	cyclic
tol	0.0001	0.0001	0.0001	0.0001
warm_start	False	False	False	False

C.4 Hybrid

	Ridge Regression 0.602
	Parameter and Feature One
0	category_2
1	current_long_cycling
2	capacity_at_end
3	capacity_retention
4	binder_type

	Ridge Regression 0.602
	Parameter and Feature One
alpha	25
copy_X	True
fit_intercept	True
max_iter	None
normalize	False
random_state	None
solver	auto
tol	0.001

C.5 Insertion

	Bayesian Ridge Regression 0.234
	Parameter and Feature Two
0	upper voltage
1	electrolyte_salt
2	binder_type

	Bayesian Ridge Regression 0.234
	Parameter and Feature Two
alpha_1	1e-06
alpha_2	1
compute_score	False
copy_X	True
fit_intercept	True
lambda_1	1
lambda_2	0.1
n_iter	300
normalize	False
tol	0.001
verbose	False

Appendix D

Code for Data Visualizations and Machine Learning Model Selection

D.1 Python Code for dQ/dV vs. Voltage Interactive Plot with Slider

```
#import numpy, pandas, and bokeh
import numpy as np
import pandas as pd
from bokeh.io import show, output_file, output_notebook
from bokeh.plotting import figure
from bokeh import events
from bokeh.models import CustomJS, ColumnDataSource, TapTool, Slider
from bokeh.layouts import column, row

#use pandas to create a dataframe from the csv input file of cycle data
dframe = pd.read_csv('volt.csv')

#create an empty dataframe to hold the cycle number and max charge transfer of that cycle on charging and discharging
cycle_data = pd.DataFrame([],columns=['cycle','charge','discharge'])

#finds the cycle number of the last cycle in the data
max_cycle = max(dframe.loc[:,'cycle'])

#for loop that will find the mlistdf[0]ax charge transferred on charge and discharge for every single cycle
for i in range(0,int(max_cycle) + 1,1):

    #create temporary dataframe to hold the given cycle's data
    tmp_cycle = dframe[(dframe['cycle']==i)]

    #pull max value of the charge step of current cycle
    chg = tmp_cycle.loc[:,'test_capchg'].max()

    #pull max value of the discharge step of current cycle
    dchg = tmp_cycle.loc[:,'test_capdchg'].max()

    #assemble the values into a temporary data frame/as a row
    tmp = pd.DataFrame({'cycle':i,'charge':chg,'discharge':dchg},index=[0])

    #concatenate the dataframe row made above to the running total of all the rows
    cycle_data=pd.concat([cycle_data,tmp],axis=0)


```

```
#create a list, whose eventual values will be dataframes that hold all the datapoints for the dq/dv chart
#the index number of the list will correspond to the cycle number of the data
listdf = []

#split up the range from 2 to 4.2 volts into 500 parts, so that the change in capacity can be calculated for each range
bins = np.linspace(2, 4.2, 500, endpoint=True)

#set up a variable to hold the value of the maximum change in capacity
max_val = 0

#create a for loop to go through every cycle
for x in range(0, int(max_cycle) +1, 1):

    #create a temporary dataframe to hold the cycling data for just the current cycle
    foo1 = dframe[(dframe['cycle']==x)]

    #find the max and min voltages for the current cycle
    voltage_min = min(foo1.loc[:,'test_vol'])
    voltage_max = max(foo1.loc[:,'test_vol'])

    #create an empty dataframe to store the average voltage value and the changes in capacity on charge/discharge for the voltage range whose average is given
    dq_dv = pd.DataFrame([],columns=['voltage','dq/dv_chg','dq/dv_dchg'])


    for i in range(0,len(bins)-1,1):

        #isolate the discharge capacity values (Q) that correspond to the voltage window(dV)
        tmp = foo1[(foo1['test_vol'] >= bins[i]) & (foo1['test_vol'] <=    bins[i+1]) & (foo1.loc[:,'step_type'] == 2) ]

        #get the average value for the voltage window by averaging i and i+1
        voltage_tmp = (bins[i]+bins[i+1])/2

        #get the change in capacity (Q) in the voltage window (dV), hence getting dQ
        #not all voltage windows will have representative values in the cycling data, so in those case we use an if statement
        if len(tmp) >= 1:
```

```
            dqdv_dchg_tmp = max(tmp.loc[:,'test_capdchg'])-min(tmp.loc[:,'test_capdchg'])
        else:
            dqdv_dchg_tmp = 0

        #The process is then repeated for the charging step

        #isolate the charge capacity values (Q) that correspond to the voltage window(dV)
        tmp = foo1[(foo1['test_vol'] >= bins[i]) & (foo1['test_vol'] <=    bins[i+1]) & (foo1.loc[:,'step_type'] == 1) ]

        #get the change in capacity (Q) in the voltage window (dV), hence getting dQ
        if len(tmp) >= 1:
            dqdv_chg_tmp = max(tmp.loc[:,'test_capchg'])-min(tmp.loc[:,'test_capchg'])
        else:
            dqdv_chg_tmp = 0

        #create the temporary data frame row for the current voltage window
        foo = pd.DataFrame({'voltage':voltage_tmp,'dq/dv_chg':dqdv_chg_tmp,'dq/dv_dchg':dqdv_dchg_tmp},index=[0])
        foo = foo[['voltage', 'dq/dv_chg', 'dq/dv_dchg']]

        #concatenate the temporary dataframe row with the running dataframe for the given cycle
        dq_dv = pd.concat([dq_dv,foo],axis=0)


    #adds the dataframe for the given cycle to the list of dataframes
    listdf.append(dq_dv)

    #update max change in capacity value based on the max dQ for the given cycle
    if max(max(dq_dv.loc[:,'dq/dv_chg']), max(dq_dv.loc[:,'dq/dv_dchg'])) > max_val:
        max_val = max(max(dq_dv.loc[:,'dq/dv_chg']), max(dq_dv.loc[:,'dq/dv_dchg']))

#create vectors of the voltage, discharge dQ, and charge dQ values in order to initialize the graphs
x1 = list(listdf[0]['voltage'])
y1 = list(listdf[0]['dq/dv_dchg'])
z1 = list(listdf[0]['dq/dv_chg'])

```

```
#in order for CustomJS to accept the dataframes, we create a list of lists rather than a list of dataframes
#flattening the values of the dataframe means that we create a single list where the values are in the order:
#voltage value, dQ on charge, dQ on discharge, and this is repeated until all rows in the dataframe are listed
for i in range(0,len(listdf),1):
    listdf[i] = list(listdf[i].values.flatten())

#the graphs base their data off of ColumnDataSource objects that must be initialized to some values
#initialize the CDS for the discharge curve and the charge curve by using the data from the 0th cycle
sourced = ColumnDataSource(data=dict(x1=x1, y1=y1))
sourcec = ColumnDataSource(data=dict(x1=x1, z1=z1))

#this callback object is run whenever the slider position updates
#the JavaScript code is commented using '//'
callback = CustomJS(args=dict(sourced = sourced, sourcec = sourcec), code=
```

```
      """  //slider value is written to the variable collectind and dq/dv data is written to variable listdf
        var collectind = cb_obj.value;
        var listdf = %s

        //data for the discharge and charge curves are written to variables datad and datac
        var datad = sourced.data;
        var datac = sourcec.data;

        //the discharge and charge curves' voltage and dQ data are extracted to different variables
        dvolt = datad['x1']
        cvolt = datac['x1']
        dcap = datad['y1']
        ccap = datac['z1']

        //the for loop goes through every voltage value and replaces the data for the discharge and charge curves
        //the list listdf was constructed so that the index matched the cycle number so the slider's value indicates both cycle number and list index
        for (i = 0; i < dvolt.length; i++) {
            ind0 = 3 * i
            ind1 = (3*i) + 1
            ind2 = (3*i) + 2
```

```
            dvolt[i] = listdf[collectind][ind0]
            cvolt[i] = listdf[collectind][ind0]
            ccap[i] = listdf[collectind][ind1]
            dcap[i] = listdf[collectind][ind2]
        }
        //trigger update of data sources for the two dq/dv curves
        sourced.trigger('change');
        sourcec.trigger('change') """
```

Listing D.1: Custom JS code

```
    % (listdf))

#create the plot of dQ/dV for the discharge and charge curves
q = figure(title = "dQ/dV", x_axis_label = 'Voltage (V)',
y_axis_label = 'Capacity (Ah)', y_range = [0,round(max_val,1)])
q.scatter('x1', 'y1',source=sourced, legend = "Discharge",
fill_color = 'red', size = 10 )
q.scatter('x1', 'z1',source=sourcec, legend = "Charge", fill_color =
 'green', size = 10)
q.title.text_font_size = "25px"

#create the slider object and have the graph update based on whether
 the slider position was changed
#when the slider position is changed, its value is sent to the '
callback' object that we create
slider = Slider(start=0, end=max_cycle, value=0, step=1, title="
Cycle")
slider.js_on_change('value', callback)

#set up the display of the slider and graph
layout = column(slider, q)
show(layout)
#output the graph to an html file
output_file("slider.html", title="Graph")
```

D.2 Python Code for Clickable Capacity vs. Cycle and updating dQ/dV vs. Voltage Plot

```
#import numpy, pandas, and bokeh
import numpy as np
import pandas as pd
from bokeh.io import show, output_file, output_notebook
from bokeh.plotting import figure
from bokeh import events
from bokeh.models import CustomJS, ColumnDataSource, TapTool, Slider
from bokeh.layouts import column, row

#use pandas to create a dataframe from the csv input file of cycle data
dframe = pd.read_csv('volt.csv')

#create an empty dataframe to hold the cycle number and max charge transfer of that cycle on charging and discharging
cycle_data = pd.DataFrame([],columns=['cycle','charge','discharge'])

#finds the cycle number of the last cycle in the data
max_cycle = max(dframe.loc[:,'cycle'])

#for loop that will find the max charge transferred on charge and discharge for every single cycle
for i in range(0,int(max_cycle) + 1,1):

    #create temporary dataframe to hold the given cycle's data
    tmp_cycle = dframe[(dframe['cycle']==i)]

    #pull max value of the charge step of current cycle
    chg = tmp_cycle.loc[:,'test_capchg'].max()

    #pull max value of the discharge step of current cycle
    dchg = tmp_cycle.loc[:,'test_capdchg'].max()

    #assemble the values into a temporary data frame/as a row
    tmp = pd.DataFrame({'cycle':i,'charge':chg,'discharge':dchg},
index=[0])

    #concatenate the dataframe row made above to the running total of all the rows
    cycle_data=pd.concat([cycle_data,tmp],axis=0)

#create a list, whose eventual values will be dataframes that hold all the datapoints for the dq/dv chart
```

```
#the index number of the list will correspond to the cycle number of
 the data
listdf = []

#split up the range from 2 to 4.2 volts into 500 parts, so that the
change in capacity can be calculated for each range
bins = np.linspace(2, 4.2, 500, endpoint=True)

#set up a variable to hold the value of the maximum change in
capacity
max_val = 0

#create a for loop to go through every cycle
for x in range(0, int(max_cycle) +1, 1):

    #create a temporary dataframe to hold the cycling data for just
the current cycle
    foo1 = dframe[(dframe['cycle']==x)]

    #find the max and min voltages for the current cycle
    voltage_min = min(foo1.loc[:,'test_vol'])
    voltage_max = max(foo1.loc[:,'test_vol'])

    #create an empty dataframe to store the average voltage value
and the changes in capacity on charge/discharge for the voltage
range whose average is given
    dq_dv = pd.DataFrame([],columns=['voltage','dq/dv_chg','dq/
dv_dchg'])


    for i in range(0,len(bins)-1,1):

        #isolate the discharge capacity values (Q) that correspond
to the voltage window(dV)
        tmp = foo1[(foo1['test_vol'] >= bins[i]) & (foo1['test_vol']
 <=    bins[i+1]) & (foo1.loc[:,'step_type'] == 2) ]

        #get the average value for the voltage window by averaging i
 and i+1
        voltage_tmp = (bins[i]+bins[i+1])/2

        #get the change in capacity (Q) in the voltage window (dV),
hence getting dQ
        #not all voltage windows will have representative values in
the cycling data, so in those case we use an if statement
        if len(tmp) >= 1:
            dqdv_dchg_tmp = max(tmp.loc[:,'test_capdchg'])-min(tmp.
loc[:,'test_capdchg'])
```

```
        else:
            dqdv_dchg_tmp = 0

        #The process is then repeated for the charging step

        #isolate the charge capacity values (Q) that correspond to
the voltage window(dV)
        tmp = foo1[(foo1['test_vol'] >= bins[i]) & (foo1['test_vol']
 <=    bins[i+1]) & (foo1.loc[:,'step_type'] == 1) ]

        #get the change in capacity (Q) in the voltage window (dV),
hence getting dQ
        if len(tmp) >= 1:
            dqdv_chg_tmp = max(tmp.loc[:,'test_capchg'])-min(tmp.loc
[:,'test_capchg'])
        else:
            dqdv_chg_tmp = 0

        #create the temporary data frame row for the current voltage
 window
        foo = pd.DataFrame({'voltage':voltage_tmp,'dq/dv_chg':
dqdv_chg_tmp,'dq/dv_dchg':dqdv_dchg_tmp},index=[0])
        foo = foo[['voltage', 'dq/dv_chg', 'dq/dv_dchg']]

        #concatenate the temporary dataframe row with the running
dataframe for the given cycle
        dq_dv = pd.concat([dq_dv,foo],axis=0)


    #adds the dataframe for the given cycle to the list of
dataframes
    listdf.append(dq_dv)

    #update max change in capacity value based on the max dQ for the
 given cycle
    if max(max(dq_dv.loc[:,'dq/dv_chg']), max(dq_dv.loc[:,'dq/
dv_dchg'])) > max_val:
        max_val = max(max(dq_dv.loc[:,'dq/dv_chg']), max(dq_dv.loc
[:,'dq/dv_dchg']))

#create vectors of the voltage, discharge dQ, and charge dQ values
in order to initialize the dq/dv curves
x1 = list(listdf[0]['voltage'])
y1 = list(listdf[0]['dq/dv_dchg'])
z1 = list(listdf[0]['dq/dv_chg'])

#create vectors of the cycle and capacity values in order to create
the Capacity vs Cycle graph
```

```
cyc = list(cycle_data.loc[:,'cycle'])
char = list(cycle_data.loc[:,'charge'])

#in order for CustomJS to accept the dataframes, we create a list of lists rather than a list of dataframes
#flattening the values of the dataframe means that we create a single list where the values are in the order:
#voltage value, dQ on charge, dQ on discharge, and this is repeated until all rows in the dataframe are listed
for i in range(0,len(listdf),1):
    listdf[i] = list(listdf[i].values.flatten())

#the graphs base their data off of ColumnDataSource objects that must be initialized to some values
#initialize the CDS for the discharge curve and the charge curve by using the data from the 0th cycle
sourced = ColumnDataSource(data=dict(x1=x1, y1=y1))
sourcec = ColumnDataSource(data=dict(x1=x1, z1=z1))

#initialize the CDS for the Capacity vs Cycle curve by using the vectors created earlier
source  = ColumnDataSource(data=dict(xc=cyc, yc = char))

#this callback object is run whenever a point is clicked on the Capacity vs Cycle graph
# // index of the clicked on value is written to the
#the JavaScript code is commented using '//'
source.callback = CustomJS(args=dict(sourced = sourced, sourcec = sourcec), code=
```

```
        """ //index of the clicked on value is written to the variable collectind and dq/dv data is written to variable listdf
        var collectind = cb_obj.selected['1d'].indices[0];
        var listdf = %s

        //data for the discharge and charge curves are written to variables datad and datac
        var datad = sourced.data;
        var datac = sourcec.data;

        //the discharge and charge curves' voltage and dQ data are extracted to different variables
        dvolt = datad['x1']
        cvolt = datac['x1']
        dcap = datad['y1']
        ccap = datac['z1']

```

```
        //the for loop goes through every voltage value and replaces the data for the discharge and charge curves
        //the list listdf was constructed so that the index matched the cycle number so the tap point's index value indicates both cycle number and list index
        for (i = 0; i < dvolt.length; i++) {
            ind0 = 3 * i
            ind1 = (3*i) + 1
            ind2 = (3*i) + 2
            dvolt[i] = listdf[collectind][ind0]
            cvolt[i] = listdf[collectind][ind0]
            ccap[i] = listdf[collectind][ind1]
            dcap[i] = listdf[collectind][ind2]
        }
        //trigger update of data sources for the two dq/dv curves
        sourced.trigger('change');
        sourcec.trigger('change') """
```

Listing D.2: Custom JS code

```
    % (listdf))


#the taptool object triggers the callback code whenever a point is clicked on the Capacity vs Cycle graph
taptool = TapTool(callback=source.callback)

#set up the Capacity vs Cycle graph and the dq/dv curves
p = figure(tools = [taptool], title = "Capacity vs Cycle", x_range = (0,max_cycle), y_range = (0,1.5),
           x_axis_label = 'Cycles', y_axis_label = 'Capacity (Ah)' )
p.scatter("xc", "yc", source = source, size = 10)
p.title.text_font_size = "25px"
q = figure(title = "dQ/dV", x_axis_label = 'Voltage (V)', y_axis_label = 'Capacity (Ah)', y_range = [0,round(max_val,1)])
q.scatter('x1', 'y1',source=sourced, legend = "Discharge", fill_color = 'red', size = 10 )
q.scatter('x1', 'z1',source=sourcec, legend = "Charge", fill_color = 'green', size = 10)
q.title.text_font_size = "25px"

#set up the display of the two graphs
show(column(p,q))

#output the graphs to an html file
output_file("clickable.html", title="Graph")
```

D.3 Python Code for Model Evaluation Class & Preprocessing Functions

D.3.1 General ModelEval Class

```
#import libraries
import numpy as np
import pandas as pd
import seaborn as sns
from sklearn import svm, linear_model
from sklearn.svm import SVR, LinearSVR, NuSVR
import copy
from matplotlib import pyplot as plt
from sklearn.model_selection import cross_val_score
from sklearn import linear_model
from sklearn.model_selection import StratifiedKFold
from sklearn.feature_selection import RFECV
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2, f_regression,
mutual_info_regression
from sklearn.model_selection import GridSearchCV
from sklearn.kernel_ridge import KernelRidge
from sklearn.neighbors import KNeighborsRegressor
from sklearn import tree
from sklearn import neural_network
from sklearn.preprocessing import OneHotEncoder, LabelEncoder
from sklearn.linear_model import LinearRegression, Ridge, Lasso,
ElasticNet, OrthogonalMatchingPursuit, BayesianRidge, SGDRegressor


class ModelEval:


    #the following list indicates which regressors can be used with
RFECV feature selection
    validrfecv = ['SVR_LIN', 'OLS', 'RIDGE', 'LASSO', 'ENET', 'OMP',
 'BAYESIAN_RIDGE', 'SGD', 'DTREE']

    #the following lists create a correspondence between the name
shortcuts, the full names of the regressors,
    #and the actual scikit-learn objects representing the regressor
    keys = ['SVR_LIN', 'OLS', 'RIDGE', 'LASSO', 'ENET', 'OMP', '
BAYESIAN_RIDGE', 'SGD', 'DTREE', 'SVR_RBF', 'SVR_POLY', 'SVR_NU','
KERNEL', 'KNN', 'NET']
```

```
    values = ['Linear Support Vector Regression', 'Ordinary Least Squares Regression', 'Ridge Regression', 'Lasso Regression', 'Elastic Net Regression', 'Orthogonal Matching Pursuit', 'Bayesian Ridge Regression', 'Stochastic Gradient Descent Regression', 'Decision Tree Regression', 'Radial Basis Function Support Vector Regression', 'Polynomial Support Vector Regression', 'Nu Support Vector Regression', 'Kernel Ridge Regression', 'K Nearest Neighbors Regression', 'Supervised Neural Networks Regression']
    model_list = [LinearSVR(), LinearRegression(), Ridge(), Lasso(), ElasticNet(), OrthogonalMatchingPursuit(), BayesianRidge(), SGDRegressor(), tree.DecisionTreeRegressor(), SVR(kernel='rbf'), SVR(kernel='poly'), NuSVR(),  KernelRidge(), KNeighborsRegressor(), neural_network.MLPRegressor()]

    #a list of the parameters to be used in GridSearchCV for every single regressor type
    param_list = [{'epsilon': [0,.1, .2, .3,.4,.5],'C': [1, 3, 10, 50, 100, 500,1000]},
                {'normalize': [True, False]},
                {'alpha': [.1, .3 , .5, .9, 1,1.3,1.4, 5, 100, 1000, 500, 300, 20, 25]},
                {'alpha': [.1, .3 , .5, .9, 1,1.3,1.4, 5, 100, 1000, 500, 300, 20, 25]},
                {'alpha': [.1, .3 , .5, .9, 1,1.3,1.4, 5, 100, 1000, 500, 300, 20, 25], 'l1_ratio': [.1,.15, .2, .3,.4,.5,.6, .7, .8,.9, 1]},
                {'n_nonzero_coefs': [1,2]},
                {'alpha_1': [1e-06,1e-05, 1e-04, 1e-03, 1e-02, 1e-01, 1], 'alpha_2': [1e-06,1e-05, 1e-04, 1e-03, 1e-02, 1e-01, 1], 'lambda_1':[1e-06,1e-05, 1e-04, 1e-03, 1e-02, 1e-01, 1], 'lambda_2': [1e-06,1e-05, 1e-04, 1e-03, 1e-02, 1e-01, 1]},
                {'loss': ['squared_loss', 'huber', 'epsilon_insensitive', 'squared_epsilon_insensitive'], 'penalty':['l2', 'l1','elasticnet'], 'l1_ratio': [.1,.15, .2, .3,.4,.5,.6, .7, .8,.9, 1], 'alpha': [1e-06,1e-05, 1e-04, 1e-03, 1e-02, 1e-01, 1]},
                {'criterion': ['mse', 'friedman_mse', 'mae']},
                {'kernel': ['rbf'], 'C': [1, 3, 10, 50, 100, 500,1000],'epsilon': [0,.1, .2, .3,.4,.5] },
                {'kernel': ['poly'], 'C': [1, 3, 10, 50, 100, 500,1000],'epsilon': [0,.1, .2, .3,.4,.5], 'degree':[1,2,3,4,5]},
                {'kernel': ['rbf'], 'C': [1, 3, 10, 50, 100, 500,1000],'nu': [.1, .2, .3,.4,.5,.6,.7,.8]},
                {'alpha': [.1, .3 , .5, .9, 1,1.3,1.4, 5, 100, 1000, 500, 300, 20, 25]},
                {'n_neighbors':[1,2,3,4,5,6,7]},
                {'alpha': [1e-06,1e-05, 1e-04, 1e-03, 1e-02, 1e-01, 1], 'activation': ['identity', 'logistic', 'tanh', 'relu'],'solver': ['lbfgs', 'sgd', 'adam']}]
```

```
52
53      #creates dictionaries from the lists using the name shortcut as
    the key so they can be easily accessed
54      expand = dict(zip(keys, values))
55      models = dict(zip(keys, model_list))
56      param = dict(zip(keys, param_list))
57
58      def __init__(self, model):
59
60          #these are attributes for the Python class that we are
    making
61          #calling .attributename on a Python class will return
    whatever variable we associate with the attributen
62          self.modeltype = model  #returns the full name of the
    regressor
63          self.paramonly = []     #returns the parameters used when
    doing Parameter only optimization
64          self.f2p = []           #returns the parameters used when
    doing RFECV feature selection with parameter opt.
65          self.f1p = []           #returns the parameters used when
    doing SelectKBest feature selection with parameter opt.
66          self.f1f = []           #returns the features used when
    doing SelectKBest feature selection only
67          self.f2f = []           #returns the features used when
    doing RFECV feature selection only
68          self.f1pf = []          #returns the features used when
    doing SelectKBest feature selection with parameter opt.
69          self.f2pf = []          #returns the features used when
    doing RFECV feature selection with parameter opt.
70          self.defreturnp = self.models[model] #returns the parameters
     for when no parameter selection is done
71          self.valid = model in self.validrfecv #Boolean for whether
    RFECV can be performed or not
72          self.modelstring = self.expand[model] #full name of the
    regressor
73          self.estimator = self.models[model]   #stores the estimator
    object from scikit-learn
74
75          #the following line runs the preprocessing code and stores
    the lists of DataFrames, etc...
76          self.def_label, self.def_onehot, self.def_ys, self.
    classorder, self.def_orig = ModelEval.dfout([])
77
78          self.defreturnf = self.classremover() #returns the features
    for when no feature selection is done
79          self.tuned_parameters = self.param[model] #returns the
    parameters to be used for the given regressor
```

```
        self.analyze = self.analysis()            #returns a DF with
 the scoring for the opt. types
        self.paramreturn = self.paramsfix()       #converts the
format of the parameters/features used to DF

        #returns a dataframe of the best performing optimizations
with their features and parameters
        self.bestperf = self.bestperffcn()
        self.addedparams = self.addparamsf()

    #makes sure that the class column isn't removed when the
preprocessing function is called
    def classremover(self):
        liste = list(self.def_label[0].columns.values)
        liste.remove('class')
        return liste

    #THE FOLLOWING ARE ALL FUNCTIONS FOR ORGANIZING THE FEATURES AND
 PARAMETERS OF EACH OPTIMIZATION TYPE#
    def addparamsf(self):
        for x in range(0,self.bestperf.shape[0],1):
            listx = []
            for y in self.bestperf.iloc[x]["Best Performers"]:
                q = self.helperfcn(y, x)
                listx.append(q)
            self.bestperf['Features/Params'][x] = listx

        secondl = list(self.bestperf.columns)
        firstl = []
        firstl.append(self.modelstring)
        firstl = firstl * 3
        columnsn = pd.MultiIndex.from_tuples(list(zip(firstl,secondl
)))
        self.bestperf.columns = columnsn
        return self.bestperf

    def helperfcn(self, typestring, classindex):
        if (typestring == "One Hot Encoding"):
            return [self.defreturnp, self.defreturnf]
        if (typestring == "Label Encoding"):
            return [self.defreturnp, self.defreturnf]
        if (typestring == "Feature One Only"):
            return [self.defreturnp, self.f1f.iloc[classindex][1]]
        if (typestring == "Feature Two Only"):
            return [self.defreturnp, self.f2f.iloc[classindex][1]]
        if (typestring == "Parameter and Feature One"):
            return [self.f1p.iloc[classindex][1], self.f1pf.iloc[
classindex][1]]
```

```
        if (typestring == "Parameter and Feature Two"):
            return [self.f2p.iloc[classindex][1], self.f2pf.iloc[
classindex][1]]
        if (typestring == "Parameter Only"):
            return [self.paramonly.iloc[classindex][1],self.
defreturnf ]



    def bestperffcn(self):
        df = self.analyze.xs('Scores', level=2, axis=1)
        rowmax = df.max(axis=1)
        arr1 = df.values.astype(float) == rowmax[:,None]
        dfi = rowmax.to_frame().rename(columns= {0: 'Scores'})
        dfi['Best Performers'] = np.nan
        dfi['Features/Params'] = np.nan
        dfi['Best Performers']= dfi['Best Performers'].astype(object
)
        dfi['Features/Params']= dfi['Features/Params'].astype(object
)
        for x in range(0,df.shape[0],1):
            value = [y[1] for y in list(df.columns[arr1[x]])]
            dfi['Best Performers'][x] = value
        return dfi


    def paramsfix(self):
        row = len(self.classorder) + 1
        self.f1p = pd.DataFrame(self.f1p).iloc[[(row*(i-1)) for i in
 range(1,row,1)]]
        if (self.valid):
            self.f2p = pd.DataFrame(self.f2p).iloc[[(row*(i-1)) for
i in range(1,row,1)]]
            self.f2pf = pd.DataFrame(self.f2pf)
            self.f2f = pd.DataFrame(self.f2f)
        self.paramonly = pd.DataFrame(self.paramonly)
        self.f1pf = pd.DataFrame(self.f1pf)
        self.f1f = pd.DataFrame(self.f1f)

    def analysis(self):

        #default scoring
        def_score = self.scoreprint(self.def_label, self.estimator,
self.def_ys)
        #one hot scoring
        one_score = self.scoreprint(self.def_onehot, self.estimator,
 self.def_ys)

```

```
        #one hot vs label encoding
        onehotvslabel = ModelEval.scoreformat(one_score, def_score,
self.modelstring)

        #param only scoring
        param_only_score = self.param_sel(self.def_label, self.
def_ys, self.estimator)
        #single score df for param
        paramdf = ModelEval.singlescoreformat(param_only_score, self
.modelstring, "Parameter Only")

        #feature selection 1 only
        feature_sel_one = self.feature_sel(self.def_label, self.
def_ys, self.estimator)[0]
        #feature 1 df
        f1df = ModelEval.singlescoreformat(feature_sel_one, self.
modelstring, "Feature One Only")

        #merge default and param only DFs
        runningdf = ModelEval.dfmerger(onehotvslabel, paramdf)
        #merge running and f1 df
        runningdf = ModelEval.dfmerger(runningdf, f1df)

        #feature selection 2 only
        if (self.valid):
            feature_sel_two = self.feature_sel2(self.def_label, self
.def_ys, self.estimator)[0]
            f2df = ModelEval.singlescoreformat(feature_sel_two, self
.modelstring, "Feature Two Only")
            runningdf = ModelEval.dfmerger(runningdf, f2df)
        else:
            f2df = None

        #feature 1 with param
        f1_param = self.feature_sel_param(self.def_label, self.
def_ys, self.estimator)[0]
        f1pdf = ModelEval.singlescoreformat(f1_param, self.
modelstring, "Parameter and Feature One")

        #merge running and f1pdf
        runningdf = ModelEval.dfmerger(runningdf, f1pdf)

        #feature 2 with param
        if (self.valid):
            f2_param = self.feature_sel2_param(self.def_label, self.
def_ys, self.estimator)[0]
            f2pdf = ModelEval.singlescoreformat(f2_param, self.
modelstring, "Parameter and Feature Two")
```

```
198             runningdf = ModelEval.dfmerger(runningdf, f2pdf)
199         else:
200             f2pdf = None
201 
202 
203         return runningdf.applymap(lambda x: '%.3f' % x)
204 
205     #END OF SECTION FOR ORGANIZING THE FEATURES AND PARAMETERS OF
    EACH OPTIMIZATION TYPE#
206 
207     #this function takes a list of feature DFs, a regressor, and a
    list of outcomes as input
208     #it returns a list of lists (one list per class), with the
    format of the nested list being:
209     #[coefficient of determination, standard deviation of the
    coefficient, name of the class]
210     def scoreprint(self, listofdfs, estimator, listofys):
211         #initialize the empty list to return
212         listperf = []
213         #iterate through all the classes in the list
214         for x in range(0,len(listofdfs),1):
215             scorel = []
216             stds = []
217             #perform ten scoring runs and average all 10 for better
    model evaluation standards
218             for y in range(0,10,1):
219                 scores = cross_val_score(estimator, listofdfs[x],
    listofys[x], cv=2)
220                 scorel.append(scores.mean())
221                 stds.append(scores.std() * 2)
222             #add the list for the current class to the running list
223             listperf.append([(sum(scorel)/len(scorel)),(sum(stds)/
    len(stds)), self.classorder[x]])
224         #return the list
225         return listperf
226 
227     #this function just prints out a nicely formatted DataFrame
    comparing One Hot encoding to Label encoding
228     #takes in a label encoded list of lists and one hot encoded list
     of lists (formatted using the scoreprint function)
229     #and the name of regressor as input
230     def scoreformat(printonehot, printlabel, stringname):
231         label = pd.DataFrame(printlabel, columns = ['Scores', '
    Deviation', 'Class']).set_index('Class')
232         oneh = pd.DataFrame(printonehot, columns = ['Scores', '
    Deviation', 'Class']).set_index('Class')
233         C = [stringname] *4
```

```
234         A = ['One Hot Encoding', 'One Hot Encoding','Label Encoding'
, 'Label Encoding']
235         B = ['Scores', 'Deviation', 'Scores', 'Deviation']
236         columns = pd.MultiIndex.from_tuples(list(zip(C,A,B)))
237         df = pd.merge(oneh, label, left_index=True, right_index=True
)
238         df.columns = columns
239         return df
240 
241     #this function just prints out a nicely formatted DataFrame for
a given analysis type
242     #takes in a list of lists (formatted using scoreprint), the name
 of the regressor, and the name of the analysis
243     def singlescoreformat(printed, stringname1, stringname2):
244         dfq = pd.DataFrame(printed, columns = ['Scores', 'Deviation'
, 'Class']).set_index('Class')
245         C = [stringname1] * 2
246         A = [stringname2] * 2
247         B = ['Scores', 'Deviation']
248         columns = pd.MultiIndex.from_tuples(list(zip(C,A,B)))
249         dfq.columns = columns
250         return dfq
251 
252     #helper function to merge two dataframes with the same index and
 return that DF
253     def dfmerger(df1,df2):
254         return pd.merge(df1,df2, left_index=True, right_index=True )
255 
256     #uses SelectKBest feature selection to select features
257     #takes in a single DataFrame along with the corresponding output
 vector as inputs
258     #returns a list of lists, where the first nested list is a list
of the columns to be remove and the second
259     #is a list of the columns to keep
260     def colremove(df_enc, outs):
261         #initialize the optimizer to use a regression scoring
function (F-test) and 5 fold CV
262         select = SelectKBest(f_regression, k=5)
263         #run the feature selection on the given DF and outcomes
264         select.fit_transform(df_enc, outs)
265         #collect the columns to keep and the columns to remove
266         keep = list(df_enc.columns[select.get_support()])
267         total = list(df_enc.columns)
268         rid = list(np.setdiff1d(total,keep))
269         #don't want to remove the class column since it is necessary
 in the preprocessing
270         if 'class' in rid:
271             rid.remove('class')
```

```
272         return [rid, keep]
273 
274     #the following function is a helper function to perform
    SelectKBest Feature Selection and returns the
275     #scores along with the features used. The inputs are a list of
    DFs, a list of outcomes, and the estimator
276     def feature_sel(self, label, outs, estimator):
277         labelstorage = []
278         keepstorage = []
279         for x in range(0,len(label),1):
280             #get the columns to remove for label encoding
281             labelrem, keeper = ModelEval.colremove(label[x], outs[x
    ])
282 
283             #get the adjusted label DF
284             label_temp1, onehot_temp1, out_temp1, classorder_temp1,
    listorig_temp1 = ModelEval.dfout(labelrem)
285 
286             #add the current class's score
287             labelstorage.append(self.scoreprint(label_temp1,
    estimator, out_temp1)[x])
288 
289             #record the features used
290             keepstorage.append(keeper)
291             self.f1f.append([self.classorder[x], keeper])
292         return [labelstorage, keepstorage]
293 
294     #this function takes a list of feature DFs, a regressor, and a
    list of outcomes as input
295     #it returns a list of lists (one list per class), with the
    format of the nested list being:
296     #[coefficient of determination, standard deviation of the
    coefficient, name of the class]
297     #this function uses parameter optimization on the given DF and
    regressor, unlike the other scoreprint function
298     def scoreprintparam(self, listofdfs, estimator, listofys, number
    ):
299         listperf = []
300         for x in range(0,len(listofdfs),1):
301             scorel = []
302             stds = []
303 
304             #update the estimator after performing parameter
    optimization
305             updated_estimator = self.gridfcn(estimator, listofdfs[x
    ], listofys[x])
306 
307             #keep a record of the updated estimator for each class
```

```
            if (number == "one"):
                self.f1p.append([self.classorder[x],
updated_estimator])
            if (number == "two"):
                self.f2p.append([self.classorder[x],
updated_estimator])

            #perform ten scoring runs and average all 10 for better
model evaluation standards
            for y in range(0,10,1):
                scores = cross_val_score(updated_estimator,
listofdfs[x], listofys[x], cv=2)
                scorel.append(scores.mean())
                stds.append(scores.std() * 2)

            #add the list for the given class to the running list
            listperf.append([(sum(scorel)/len(scorel)),(sum(stds)/
len(stds)), self.classorder[x]])
        return listperf


    #this is a helper function to perform parameter optimization
using GridSearchCV
    #takes in the regressor, a DF, and a vector of outcomes as input
    #returns the updated estimator
    def gridfcn(self, estimator, df, ys):
        clf = estimator
        #initialize and run the parameter optimization
        grid_search = GridSearchCV(clf, param_grid=self.
tuned_parameters, refit = True)
        grid_search.fit(df, ys)

        return grid_search.best_estimator_


    #the following function is a helper function to perform
SelectKBest Feature Selection and parameter optimization
    #and returns the scores along with the features used. The inputs
 are a list of DFs, a list of outcomes,
    #and the estimator
    def feature_sel_param(self, label, outs, estimator):
        labelstorage = []
        keepstorage = []
        #paramstorage = []
        for x in range(0,len(label),1):
            #get the columns to remove for label encoding
            labelrem, keeper = ModelEval.colremove(label[x], outs[x
])
```

```
346
347             #get the adjusted label DF
348             label_temp1, onehot_temp1, out_temp1, classorder_temp1,
listorig_temp1 = ModelEval.dfout(labelrem)
349
350             #get the scores
351             labelstorage.append(self.scoreprintparam(label_temp1,
estimator, out_temp1, "one")[x])
352
353             #record the features used
354             keepstorage.append(keeper)
355             self.f1pf.append([self.classorder[x], keeper])
356         return [labelstorage, keepstorage]
357
358
359     #the following function is a helper function to perform RFECV
Feature Selection and returns the
360     #scores along with the features used. The inputs are a list of
DFs, a list of outcomes, and the estimator
361     def feature_sel2(self, label, outs, estimator):
362         labelstorage = []
363         keepstorage = []
364         for x in range(0,len(label),1):
365             #get the columns to remove for label encoding
366             labelrem, keeper = ModelEval.colremove2(label[x],
estimator, outs[x])
367
368             #get the adjusted label DF
369             label_temp1, onehot_temp1, out_temp1, classorder_temp1,
listorig_temp1 = ModelEval.dfout(labelrem)
370
371             #get the scores
372             labelstorage.append(self.scoreprint(label_temp1,
estimator, out_temp1)[x])
373
374             #record the features used
375             keepstorage.append(keeper)
376             self.f2f.append([self.classorder[x], keeper])
377         return [labelstorage, keepstorage]
378
379
380     #the following function is a helper function to perform RFECV
Feature Selection and parameter optimization
381     #and returns the scores along with the features used. The inputs
 are a list of DFs, a list of outcomes,
382     #and the estimator
383     def feature_sel2_param(self, label, outs, estimator):
384         labelstorage = []
```

```
385         keepstorage = []
386         for x in range(0,len(label),1):
387             #get the columns to remove for label encoding
388             labelrem, keeper = ModelEval.colremove2(label[x],
estimator, outs[x])
389 
390             #get the adjusted label DF
391             label_temp1, onehot_temp1, out_temp1, classorder_temp1,
listorig_temp1 = ModelEval.dfout(labelrem)
392 
393             #get the scores
394             labelstorage.append(self.scoreprintparam(label_temp1,
estimator, out_temp1, "two")[x])
395 
396             #record the features used
397             keepstorage.append(keeper)
398             self.f2pf.append([self.classorder[x], keeper])
399         return [labelstorage, keepstorage]
400 
401 
402     #this function uses RFECV feature selection to select features
403     #takes in a single DataFrame along with the corresponding output
 vector as inputs
404     #returns a list of lists, where the first nested list is a list
of the columns to be remove and the second
405     #is a list of the columns to keep
406     def colremove2(df_enc, estimate, outs):
407 
408         #initialize the optimizer
409         rfecv = RFECV(estimator=estimate, step=1, cv = 2)
410         #run the feature selection on the given DF and outcomes
411         selector = rfecv.fit(df_enc, outs)
412 
413          #collect the columns to keep and the columns to remove
414         keep = list(df_enc.columns[rfecv.get_support(indices=True)])
415         total = list(df_enc.columns)
416         rid = list(np.setdiff1d(total,keep))
417 
418         #don't want to remove the class column since it is necessary
 in the preprocessing
419         if 'class' in rid:
420             rid.remove('class')
421         return [rid, keep]
422 
423 
424     #this is a quick helper function to run parameter only
optimization
```

```
425     #takes in a list of DFs, a list of outcomes, and an estimator as
 input
426     #it returns a list of lists in the same format as scoreprint
427     def param_sel(self, label, outs, estimator):
428         labelstorage = []
429 
430         #for every class in the DF, manually perform parameter
optimization
431         for x in range(0,len(label),1):
432             updated_estimator = self.gridfcn(estimator, label[x],
outs[x])
433             #keep a record of the regressor parameters
434             self.paramonly.append([self.classorder[x],
updated_estimator])
435             #use the scoreprint function to score the estimator
436             labelstorage.append(self.scoreprint(label,
updated_estimator, outs)[x])
437 
438         return labelstorage
```

Listing D.3: ModelEval Class without Preprocessing Function

D.3.2 Preprocessing Functions for all Analyses

```

    #this is the preprocessing code for the Rate Retention Analysis of the NIB Anodes data
    def dfout(listofcolnames):
        #import csv file with the data in it
        dframe = pd.read_csv('NIBanodes.csv', encoding='cp1252')

        #rename the unnamed column to Unknown
        dframe.rename(columns={'Unnamed: 0': 'Unknown'}, inplace=True)

        #drop the four unnecessary columns for any data analysis
        dframe = dframe.drop(['DOI', 'Title', 'Unknown', 'Year'], axis = 1)

        #drop the columns that are unrelated to rate retention experiments
        dframe = dframe.drop(['cycles', 'CE_first_cycle', 'current_long_cycling', 'capacity_at_end', 'capacity_retention'], axis = 1)

        #go through each column and drop any of those that contain more than the threshold allowed missing information
        threshold = 338
        dropl = []
        for i in range(0, len(dframe.columns), 1):
            x = dframe.iloc[:,i].isnull().sum()
            if x > threshold: dropl.append(list(dframe.columns)[i])
        dframed = dframe.drop(dropl, axis=1).dropna() #drops the columns and then drops any rows with null values

        #drop any columns that have excessive rate_retention values since those are errors in data collection
        dframed = dframed[dframed['rate_retention'] < 1.1]

        #ensure that the lower voltage column is all of a numeric datatype, rather than strings
        dframed['lower voltage']  = pd.to_numeric(dframed['lower voltage'], errors='coerce')
        dframed = dframed.dropna()

        #rename columns for cleaner results
        dframed = dframed.rename(columns = {'current_rate (mA g−1)' : 'current_rate', 'capacity (mAh g−1)' : 'capacity'})


```

```
        #drop columns that are deemed irrelevant by the various optimization methods
        dframew = dframed.drop(listofcolnames, axis=1)

        #the data must be sorted into different groups based on the class of anode, and the rate retention values must
        #be collected into a separate vector as the desired target value to be predicted

        #create empty lists to contain the dataframes for the different groups
        listbyclass = []
        list2 = []
        rates = []

        #for each class, create a vector of the rate retention values, drop the rate retention column from the dataframe
        #which is for input features only, and then add the dataframe for the given class into the lists of dataframes
        for classcol, df_col in dframew.groupby('class'):
            rates.append(df_col['rate_retention'])
            df_col = df_col.drop(['rate_retention'], axis=1)
            listbyclass.append(df_col)
            list2.append(df_col.copy(deep = True))  #make sure that the dataframes for each list are separate

        #need to convert categorical data into numerical data by using some form of encoding
        #the dataframes in the list listbyclass will use basic labelencoding, while the dataframes in the list listnew
        #will use one hot encoding

        ##################
        #LABEL ENCODING##
        ##################
        #apply label encoding to the categorical columns
        #iterate through every single DF (one for each class of anode)
        le = LabelEncoder()
        classorder = []
        for count, dframew in enumerate(listbyclass):
            #find the name of the class so we can keep track of which index was which class of anode
            classorder.append(dframew['class'].iloc[0])
            #identify the categorical columns and then apply the label encoding to each of them and store in a DF
            encl = dframew[list(dframew[dframew.T[dframew.dtypes==np.object].index].columns)].apply(le.fit_transform)
```

```
            #replace the categorical values in the original
dataframe with the encoded ones
            for x in list(dframew[dframew.T[dframew.dtypes==np.
object].index].columns):
                listbyclass[count][x] = encl[x]

        ###################
        #ONE HOT ENCODING##
        ###################
        #apply one hot encoding to the categorical columns
        #create a new list to hold the one hot encoded data. the
list list2 will hold the original unencoded data
        listnew = []

        #create a vector of the names of all the categorical columns
        df = list2[0]
        columnval = list(df[df.T[df.dtypes==np.object].index].
columns)
        #iterate through every DF and apply one hot encoding to the
relevant columns only
        for x in list2:
            listnew.append(pd.get_dummies(x, columns = columnval))

        #return a list with the label encoded list of DFs, one hot
encoded list of DFs, list of outcomes (RR values),
        #list with the order of the classes, and a list of the
unencoded DFs
        return [listbyclass, listnew, rates, classorder, list2]
```

Listing D.4: Preprocessing Function for Rate Retention

```
    #this is the preprocessing code for the Capacity Retention
Analysis of the NIB Anodes data
    def dfout(listofcolnames):
        #import csv file with the data in it
        dframe = pd.read_csv('NIBanodes.csv', encoding='cp1252')

        #rename the unnamed column to Unknown
        dframe.rename(columns={'Unnamed: 0': 'Unknown'}, inplace=
True)

        #remove the rows that are not related to long cycling
        dframe = dframe[dframe['Unknown'] == 1]

        #drop the four unnecessary columns for any data analysis
        dframe = dframe.drop(['DOI', 'Title', 'Unknown', 'Year'],
axis = 1)

```

```
        #drop the columns that are unrelated to Capacity retention
experiments
        dframe = dframe.drop(['current_rate (mA g−1)', 'capacity (
mAh g−1)','rate_retention'], axis = 1)

        #go through each column and drop any of those that contain
more than the threshold allowed missing information
        threshold = 338
        dropl = []
        for i in range(0, len(dframe.columns), 1):
            x = dframe.iloc[:,i].isnull().sum()
            if x > threshold: dropl.append(list(dframe.columns)[i])
        dframed = dframe.drop(dropl, axis=1).dropna() #drops the
columns and then drops any rows with null values

        #drop any columns that have excessive capacity_retention
values since those are errors in data collection
        dframed = dframed[dframed['capacity_retention'] < 1.1]

        #ensure that the lower voltage, and CE@1st column is all of
a numeric datatype, rather than strings
        dframed['lower voltage']  = pd.to_numeric(dframed['lower
voltage'], errors='coerce')
        dframed['CE_first_cycle']  = pd.to_numeric(dframed['
CE_first_cycle'], errors='coerce')
        dframed = dframed.dropna()

        #drop columns that are deemed irrelevant by the various
optimization methods
        dframew = dframed.drop(listofcolnames, axis=1)


        #long cycling data only had 5 feasible classes, so only
select those
        dframew = dframew.loc[dframew['class'].isin(['Alloy' ,'
Conversion','Dual Alloy','Hybrid','Insertion'])]

        #the data must be sorted into different groups based on the
class of anode, and the capacity retention values must
        #be collected into a separate vector as the desired target
value to be predicted


        #create empty lists to contain the dataframes for the
different groups
        listbyclass = []
        list2 = []
        rates = []
```

```
49
50          #for each class, create a vector of the capacity retention
    values, drop the capacity retention column from the dataframe
51          #which is for input features only, and then add the
    dataframe for the given class into the lists of dataframes
52          for classcol, df_col in dframew.groupby('class'):
53              rates.append(df_col['capacity_retention'])
54              df_col = df_col.drop(['capacity_retention'], axis=1)
55              listbyclass.append(df_col)
56              list2.append(df_col.copy(deep = True))  #make sure that
    the dataframes for each list are separate
57
58          #need to convert categorical data into numerical data by
    using some form of encoding
59          #the dataframes in the list listbyclass will use basic
    labelencoding, while the dataframes in the list listnew
60          #will use one hot encoding
61
62          ##################
63          #LABEL ENCODING##
64          ##################
65          #apply label encoding to the categorical columns
66          #iterate through every single DF (one for each class of
    anode)
67          le = LabelEncoder()
68          classorder = []
69          for count, dframew in enumerate(listbyclass):
70              #find the name of the class so we can keep track of
    which index was which class of anode
71              classorder.append(dframew['class'].iloc[0])
72              #identify the categorical columns and then apply the
    label encoding to each of them and store in a DF
73              encl = dframew[list(dframew[dframew.T[dframew.dtypes==np
    .object].index].columns)].apply(le.fit_transform)
74              #replace the categorical values in the original
    dataframe with the encoded ones
75              for x in list(dframew[dframew.T[dframew.dtypes==np.
    object].index].columns):
76                  listbyclass[count][x] = encl[x]
77
78          ####################
79          #ONE HOT ENCODING##
80          ####################
81          #apply one hot encoding to the categorical columns
82          #create a new list so we can store the one hot encoded data
83          listnew = []
84
85          #create a vector of the names of all the categorical columns
```

```
        df = list2[0]
        columnval = list(df[df.T[df.dtypes==np.object].index].
columns)
        #iterate through every DF and apply one hot encoding to the
relevant columns only
        for x in list2:
            listnew.append(pd.get_dummies(x, columns = columnval))

        #return a list with the label encoded list of DFs, one hot
encoded list of DFs, list of outcomes (CR values),
        #list with the order of the classes, and a list of the
unencoded DFs
        return [listbyclass, listnew, rates, classorder, list2]
```

Listing D.5: Preprocessing Function for Capacity Retention

```
    #this is the preprocessing code for the Coulombic Efficiency at
the First Cycle Analysis of the NIB Anodes data
    def dfout(listofcolnames):
        #import csv file with the data in it
        dframe = pd.read_csv('NIBanodes.csv', encoding='cp1252')

        #rename the unnamed column to Unknown
        dframe.rename(columns={'Unnamed: 0': 'Unknown'}, inplace=
True)

        #remove the rows that are not related to long cycling
        dframe = dframe[dframe['Unknown'] == 1]

        #drop the four unnecessary columns for any data analysis
        dframe = dframe.drop(['DOI', 'Title', 'Unknown', 'Year'],
axis = 1)

        #drop the columns that are unrelated to CE@1st experiments
        dframe = dframe.drop(['cycles','current_rate (mA g-1)', '
capacity (mAh g-1)','rate_retention'], axis = 1)

        #go through each column and drop any of those that contain
more than the threshold allowed missing information
        threshold = 338
        dropl = []
        for i in range(0, len(dframe.columns), 1):
            x = dframe.iloc[:,i].isnull().sum()
            if x > threshold: dropl.append(list(dframe.columns)[i])
        dframed = dframe.drop(dropl, axis=1).dropna() #drops the
columns and then drops any rows with null values

        #drop any columns that have excessive capacity_retention
values since those are errors in data collection
```

```
        dframed = dframed[dframed['capacity_retention'] < 1.1]

        #ensure that the lower voltage, and CE@1st column is all of a numeric datatype, rather than strings
        dframed['lower voltage']  = pd.to_numeric(dframed['lower voltage'], errors='coerce')
        dframed['CE_first_cycle']  = pd.to_numeric(dframed['CE_first_cycle'], errors='coerce')
        dframed = dframed.dropna()

        #drop columns that are deemed irrelevant by the various optimization methods
        dframew = dframed.drop(listofcolnames, axis=1)


        #long cycling data only had 5 feasible classes, so only select those
        dframew = dframew.loc[dframew['class'].isin(['Alloy' ,'Conversion','Dual Alloy','Hybrid','Insertion'])]

        #the data must be sorted into different groups based on the class of anode, and the CE@1st values must
        #be collected into a separate vector as the desired target value to be predicted


        #create empty lists to contain the dataframes for the different groups
        listbyclass = []
        list2 = []
        rates = []

        #for each class, create a vector of the CE@1st values, drop the CE@1st column from the dataframe
        #which is for input features only, and then add the dataframe for the given class into the lists of dataframes
        for classcol, df_col in dframew.groupby('class'):
            rates.append(df_col['CE_first_cycle'])
            df_col = df_col.drop(['CE_first_cycle'], axis=1)
            listbyclass.append(df_col)
            list2.append(df_col.copy(deep = True))  #make sure that the dataframes for each list are separate

        #need to convert categorical data into numerical data by using some form of encoding
        #the dataframes in the list listbyclass will use basic labelencoding, while the dataframes in the list listnew
        #will use one hot encoding
```

```
61
62          ##################
63          #LABEL ENCODING##
64          ##################
65          #apply label encoding to the categorical columns
66          #iterate through every single DF (one for each class of
    anode)
67          le = LabelEncoder()
68          classorder = []
69          for count, dframew in enumerate(listbyclass):
70              #find the name of the class so we can keep track of
    which index was which class of anode
71              classorder.append(dframew['class'].iloc[0])
72              #identify the categorical columns and then apply the
    label encoding to each of them and store in a DF
73              encl = dframew[list(dframew[dframew.T[dframew.dtypes==np
    .object].index].columns)].apply(le.fit_transform)
74              #replace the categorical values in the original
    dataframe with the encoded ones
75              for x in list(dframew[dframew.T[dframew.dtypes==np.
    object].index].columns):
76                  listbyclass[count][x] = encl[x]
77
78          ####################
79          #ONE HOT ENCODING##
80          ####################
81          #apply one hot encoding to the categorical columns
82          #create a new list so we can store the one hot encoded data
83          listnew = []
84
85          #create a vector of the names of all the categorical columns
86          df = list2[0]
87          columnval = list(df[df.T[df.dtypes==np.object].index].
    columns)
88          #iterate through every DF and apply one hot encoding to the
    relevant columns only
89          for x in list2:
90              listnew.append(pd.get_dummies(x, columns = columnval))
91
92          #return a list with the label encoded list of DFs, one hot
    encoded list of DFs, list of outcomes (CR values),
93          #list with the order of the classes, and a list of the
    unencoded DFs
94          return [listbyclass, listnew, rates, classorder, list2]
```

Listing D.6: Preprocessing Function for Coulombic Efficiency at First Cycle

```
1      #this is the preprocessing code for the State of Charge
    predictions from the Cycling Data
```

```
    def dfout(listofcolnames):

        #import csv files with the data in it
        df1 = pd.read_csv('cc.csv', encoding='cp1252')
        dframe2 = pd.read_csv('c10.csv', encoding='cp1252')
        dframe3 = pd.read_csv('cccv.csv', encoding='cp1252')
        df1['run'] = 'cc'
        dframe2['run'] = 'c10'
        dframe3['run'] = 'cccv'


        df1 = df1[df1['step_type'] != 4]
        dframe2 = dframe2[dframe2['step_type'] != 4]
        dframe3 = dframe3[dframe3['step_type'] != 4]

        df1 = df1.ix[((df1['cycle']!= 0) | (df1['step_id'] != 2))]
        dframe2 = dframe2.ix[((dframe2['cycle']!= 0) | (dframe2['
step_id'] != 2))]
        dframe3 = dframe3.ix[((dframe3['cycle']!= 0) | (dframe3['
step_id'] != 2))]

        #the loop below creates a column indicating whether the
observation is a charge/discharge,
        #the charge transferred thus far in the given cycle, and the
 SOC of the observation
        for dframe1 in [df1,dframe2,dframe3]:


            dframe1['type'] = 'str'

            dframe1['abscap'] = np.nan

            dframe1['SOC_discharge'] = np.nan
            dframe1['SOC_charge'] = np.nan
            dframe1['SOC'] = np.nan

            max_cycle = max(dframe1.loc[:,'cycle'])
            for i in range(0,int(max_cycle) + 1,1):

                #isolate vector of individual cycles
                tmp_cycle = dframe1[(dframe1['cycle']==i)]

                #pull max value of the charge step
                chg = tmp_cycle.loc[:,'test_capchg'].max()

                #pull max value of the discharge step
                dchg = tmp_cycle.loc[:,'test_capdchg'].max()

```

```
                #assemble the values into a temporary data frame
                if dchg != 0:
                    dframe1.loc[dframe1['cycle']==i, 'SOC_discharge'
]  = (dframe1[(dframe1['cycle']==i)]['test_capdchg'] / dchg)
                    dframe1.loc[((dframe1['cycle']==i) & (dframe1['
step_id']==4)), 'SOC']  = (dframe1[(dframe1['cycle']==i)]['
test_capdchg'] / dchg)
                #assemble the values into a temporary data frame
                if chg != 0:
                    dframe1.loc[dframe1['cycle']==i, 'SOC_charge']
= (dframe1[(dframe1['cycle']==i)]['test_capchg'] / chg)
                    dframe1.loc[((dframe1['cycle']==i) & (dframe1['
step_id']==2)), 'SOC']  = (dframe1[(dframe1['cycle']==i)]['
test_capchg'] / chg)

            dframe1.loc[dframe1['step_id'] == 2, 'type'] = 'charging
'
            dframe1.loc[dframe1['step_id'] == 4, 'type'] = '
discharging'


            dframe1['abscap'] = np.where(dframe1['step_id'] == 4,
dframe1['test_capdchg'], dframe1['test_capchg'])

        #drop the unnecessary columns in all the 3 dataframes and
combine them into one dataframe
        listhold = []
        for df in [df1,dframe2,dframe3]:
            listhold.append(df.drop(['Unnamed: 0', 'unix_time', '
step_id',
                'step_type', 'test_capchg', 'test_capdchg', '
test_engchg',
                'test_engdchg', 'test_tmp','SOC_discharge',
                'SOC_charge'],axis = 1))
        dframew = pd.concat(listhold)

        #create a fake class category for the data so it can be used
 with the ModelEval class
        dframew['class'] = "A"

        #drop columns that are deemed irrelevant by the various
optimization methods
        dframew = dframew.drop(listofcolnames, axis=1)

        #the data must be sorted into different groups based on the
class of anode, and the CE@1st values must
        #be collected into a separate vector as the desired target
value to be predicted
```

```
78
79          #create empty lists to contain the dataframes for the
    different groups
80          listbyclass = []
81          list2 = []
82          rates = []
83
84          #for each class, create a vector of the SOC values, drop the
     SOC column from the dataframe
85          #which is for input features only, and then add the
    dataframe for the given class into the lists of dataframes
86          for classcol, df_col in dframew.groupby('class'):
87              rates.append(df_col['SOC'])
88              df_col = df_col.drop(['SOC'], axis=1)
89              listbyclass.append(df_col)
90              list2.append(df_col.copy(deep = True))  #make sure that
    the dataframes for each list are separate
91
92
93          #need to convert categorical data into numerical data by
    using some form of encoding
94          #the dataframes in the list listbyclass will use basic
    labelencoding, while the dataframes in the list list2
95          #will use one hot encoding
96          from sklearn.preprocessing import OneHotEncoder,
    LabelEncoder
97
98          #################
99          #LABEL ENCODING##
100         #################
101         #apply label encoding to the categorical columns
102         #iterate through every single DF (one for each class of
    anode)
103         le = LabelEncoder()
104         classorder = []
105         for count, dframew in enumerate(listbyclass):
106             #find the name of the class so we can keep track of
    which index was which class of anode
107             classorder.append(dframew['class'].iloc[0])
108             #identify the categorical columns and then apply the
    label encoding to each of them and store in a DF
109             encl = dframew[list(dframew[dframew.T[dframew.dtypes==np
    .object].index].columns)].apply(le.fit_transform)
110             #replace the categorical values in the original
    dataframe with the encoded ones
111             for x in list(dframew[dframew.T[dframew.dtypes==np.
    object].index].columns):
112                 listbyclass[count][x] = encl[x]
```

```
113
114         ###################
115         #ONE HOT ENCODING##
116         ###################
117         #apply one hot encoding to the categorical columns
118         #create a new list to hold the one hot encoded data. the
    list list2 will hold the original unencoded data
119         listnew = []
120
121         #create a vector of the names of all the categorical columns
122         df = list2[0]
123         columnval = list(df[df.T[df.dtypes==np.object].index].
    columns)
124         #iterate through every DF and apply one hot encoding to the
    relevant columns only
125         for x in list2:
126             listnew.append(pd.get_dummies(x, columns = columnval))
127
128         return [listbyclass, listnew, rates, classorder, list2]
```

Listing D.7: Preprocessing Function for State of Charge

D.3.3 Post-processing Function

```
#this block of code create a ModelEval object for every type of
Regressor in the list of regressors
listcol = []
for x in ['SVR_LIN', 'OLS', 'RIDGE', 'LASSO', 'ENET', 'OMP', '
BAYESIAN_RIDGE', 'SGD', 'DTREE', 'SVR_RBF', 'SVR_NU','KERNEL']:
    jax = ModelEval(x)
    listcol.append(jax)


#the following block of code creates a DataFrame indicating the best
 performing regressor(s) for each class,
#the type of optimization(s) that led to the best score, and the
features/parameters used for the given optimization

from functools import reduce

dfList = []
for x in listcol:
    dfList.append(x.addedparams)
df = reduce(lambda df1,df2: pd.merge(df1, df2, left_index=True,
right_index=True), dfList)

dfn = df.xs('Scores', level=1, axis=1)
rowmax = dfn.max(axis=1)
arr1 = dfn.values.astype(float) == rowmax[:,None]
dfi = rowmax.to_frame().rename(columns= {0: 'Scores'})
dfi['Best Performers'] = np.nan
dfi['Features/Params'] = np.nan
dfi['Best Performing Opt'] = np.nan
dfi['Best Performers']= dfi['Best Performers'].astype(object)
dfi['Features/Params']= dfi['Features/Params'].astype(object)
dfi['Best Performing Opt']= dfi['Best Performing Opt'].astype(object
)

for x in range(0,dfn.shape[0],1):
    value = [y for y in list(dfn.columns[arr1[x]])]
    dfi['Best Performers'][x] = value

for x in range(0,dfi.shape[0],1):
    listopt = []
    listfp = []
    for i in dfi['Best Performers'][x]:
        listopt.append(df[i]['Best Performers'][x])
        listfp.append(df[i]['Features/Params'][x])

    dfi['Best Performing Opt'][x] =  listopt
```

```
41      dfi['Features/Params'][x] =  listfp
```

Listing D.8: General Postprocessing Function to Display Data