A machine learning approach to battery cycle lifetime prediction and classification using early cycle data

Rohit Rungta, Samay Garg

Abstract

This work applies machine learning algorithms to analyze early cycle data for 124 commercial lithium iron phosphate cells and predict the battery's cycle life. Principal component analysis (PCA) is conducted to search for redundant information which may simply be adding additional noise. We find that the first principal component is a good indicator of a cell's cycle life. We then develop three linear regression models; the best model predicts the cell's cycle life with a mean percent error of 30% using data from only the first 5 cycles. We also demonstrate a logistic regression model that is able to classify the cells into low- and high-lifetime groups with an accuracy of 90.3%, and a multiclass one-versus-rest logistic regression model which is able to classify the cells into 4 groups with an accuracy of 71%.

Introduction

Batteries have become an integral part of many industries today and are a crucial energy storage technology as we transition to distributed electricity generation from renewable resources.[1,2] Thus, many researchers have been developing methods to improve the performance of current battery technologies. One of the most time-intensive parts of battery research is battery diagnostics, since batteries need to be cycled thousands of times under various conditions in order to prove their robustness. Cell lifetimes are also predicted using physical and empirical models which can take months to develop, and are highly dependent on battery chemistries and degradation mechanisms.[3,4] Thus, the ability to accurately predict the performance of a battery given the first hundred cycles has the power to greatly speed up the cell development process, as researchers would quickly be able to separate promising batteries from the rest. Such a model would also be extremely generalizable since it does not depend on battery chemistry or an understanding of complex degradation mechanisms. A recent paper by Severson, et al. has developed a powerful model which has been able to predict cell cycle lifetimes using data from the first 100 cycles[5]. This model was proven to be significantly more accurate than other models, with a mean percent error of 10.7%, for the most comprehensive model[5]. Our study will use the publicly available data provided by Severson, et al. and go one step further by developing a model that will predict the lifetime of batteries from just the first five cycles, as well as classify batteries into several groups using early cycle data.

Category	Description	Number
ΔQ(V) features	Minimum	0
	Mean	1
	Variance	2
	Skewness	3
	Kurtosis	4
Discharge features	Slope of the linear fit to the capacity fade curve	5
	Intercept of the linear fit to the capacity fade curve	6
	Discharge capacity, cycle 2	7
	Difference between max discharge capacity and cycle 2	8
	Discharge capacity, cycle n	9
Other features	Average charge time, first 5 cycles	10
	Maximum temperature, cycles 2 to n	11
	Minimum temperature, cycles 2 to n	12
	Internal resistance, cycle 2	13
	Minimum internal resistance, cycles 2 to n	14
	Internal resistance, difference between cycle n and cycle 2	15

Table 1. Features used in this paper. *n* indicates that cycle *n* was used to compute the feature.

Methods

This data set consists of 124 commercially available lithium iron phosphate/ graphite cells which were cycled using varying fast-charging protocols but discharged using identical conditions.[5] The data were obtained from Severson et al.[5] We use the same features outlined in Supplementary Table 1 and Supplementary Note 1 of ref. 1; however, we note that the $\Delta Q(V{=}2V)$ feature and the temperature integral feature are omitted, since the authors did not detail how these features were computed. We also consider early cycle data, so two of these features are omitted since they do not have a meaningful analog when applied to the first several cycles[1]. Consequently, our data set consists of 124 samples with 16 features each. The feature sets used in this paper are outlined in Table 1. These features were chosen by Severson et al. from domain knowledge of lithium-ion batteries since they depend on neither cell chemistry nor specific degradation mechanisms.[5] The features are further divided into 3 categories: discharge voltage curve evolution ($\Delta Q(V)$) features, discharge capacity fade curve (discharge) features, and other features. Our work aims to use this early cycle data to accurately predict the cycle lifetime of cells as well as classify the cells into groups based on their lifetime. To quantitatively predict cell lifetimes, a principal component analysis was performed on the data, and a linear regression model was fitted to the first several principal components. To classify cells, we present two models: First, a logistic regression model is used to classify the cells into low-lifetime and high-lifetime groups. Next, a one-versus-rest (OVR) scheme using four separate logistic regression models is used to classify the cells into four categories. The data are scaled and standardized, and for both the regression model and the classification models, the data are randomly split into a training set and test set (75/25 train/test split) prior to any analysis. These models were developed in python using scikit-learn machine learning library.[6]

Results and Discussion

Principal Component Regression

Since the data provided contained many features, we predicted that some of these features might be highly correlated or redundant with other features. In order to provide insight to the level of noise in the data, we conducted principle component analysis (PCA). Upon breaking the data into its principle components, we plotted the fraction of total variance captured by each principal component (fig. 1). As expected several of the columns seem to be quite derivative of the others. This redundancy can often amplify the noise in the data; thus, we drop the last couple of principal components in further analysis.

Next, we were curious if the first couple of principal components were good indicators of the longevity of a battery. We thus plotted the principal components against each other to see if any of the batteries started to form individual clusters (fig. 2). Although the batteries did not appear to cluster, indicating no correlation between the principal components, we did notice that PC1 was effectively dividing the top tier batteries from the bottom tier batteries. Negative PC1 values consisted primarily of the bottom 50% of batteries while positive PC1 values consisted primarily of the top 50% of batteries life-cycles. This indicated that PC1 might capture certain variables in the data that are good predictors of the battery cycle-life.

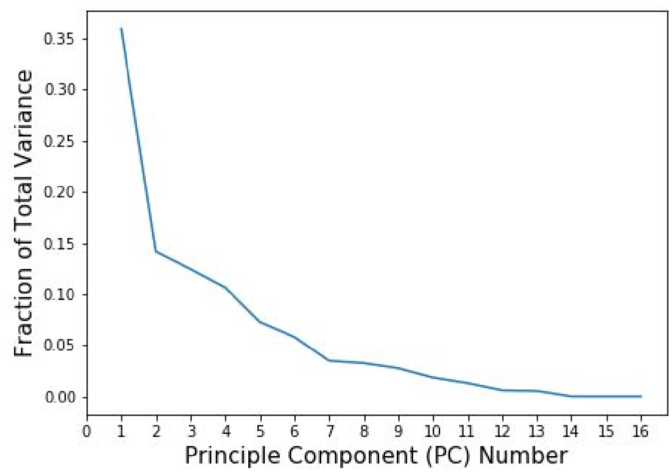


Figure 1: Total Fractional Variance of Each PC
The fraction of total variance for each consecutive principal component decreases, as expected from singular value decomposition. The last few principal components contribute to very little variance in the data. These extra features can potentially add extra noise to the data. This noise can then lead to larger errors in model development.

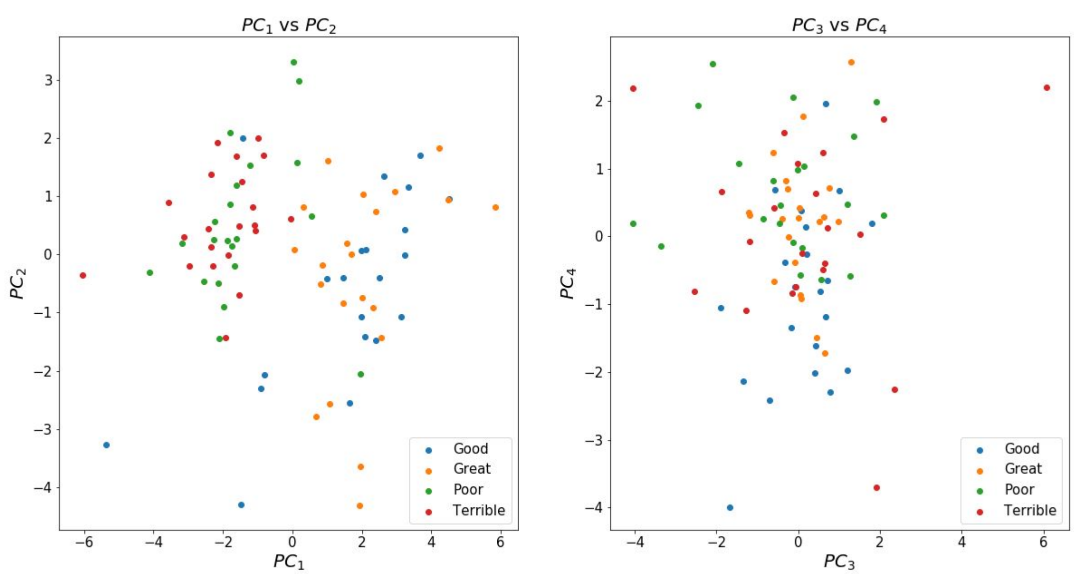


Figure 2: Correlation Between the First Four Principal Components
a) The scatterplot for PC1 vs. PC2 indicates that PC1 differentiates between the top and bottom 50% of battery cycle life. PC2 does not seem to cluster based on battery cycle life. b) The scatterplot for PC3 vs. PC4 indicates the lack of correlation between the two principal components.

To understand what features were heavily weighted in the first principle components, we plotted the weight of each feature on the first principle component (fig. 3). One key feature in PC1 was the variance of the discharge voltage curve evolution var(ΔQ(V)). This aligns with the findings of Severson et al. Their simplest model (involving only the var(ΔQ(V)) feature) was proven to be quite predictive of a cell's lifetime.

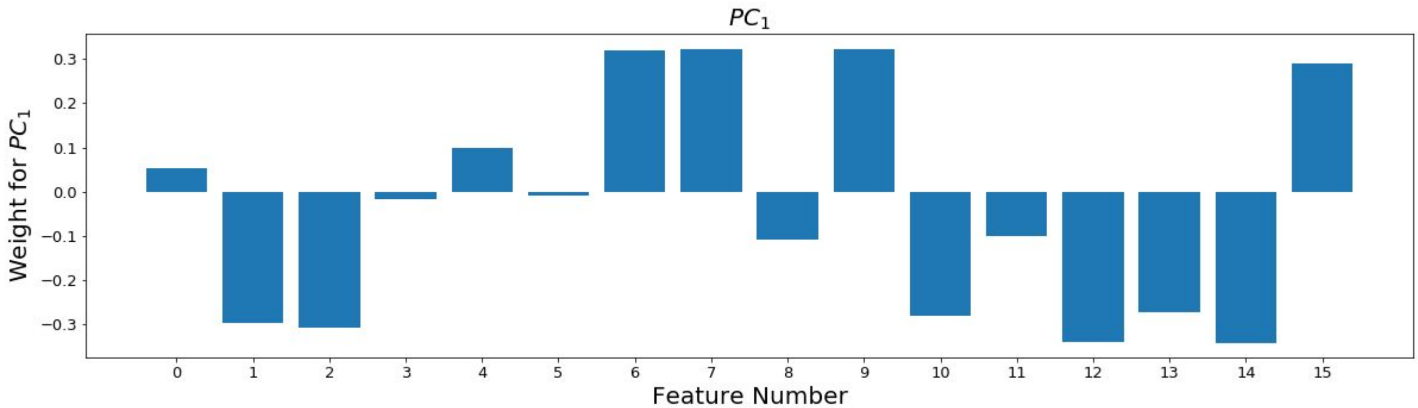


Figure 3: Contributions of Each Feature to PC1
The contribution of each feature to PC1 indicates that the var(ΔQ(V)) and mean(ΔQ(V)) are two important features in calculating the first principal component. We hypothesize that these features will have the greatest predictive power. A description of each feature number can be found in Table 1.

Understanding the physical intuitions, behind the different principle components, we created 3 distinct linear regression models. The first model used the first 4 principle components of the original data, as they explained the majority of the variance in the data set. The second model simply used and the third simply used them as their features. The root mean squared and percent errors were chosen as the statistics to evaluate the accuracy of our model (Eq. 1).

$$RMSE = \sqrt{\frac{1}{n}\Sigma(y_i - \hat{y_i})^2} \qquad \% \, err = \frac{1}{n}\Sigma\frac{|y_i - \hat{y_i}|}{y_i} * 100 \qquad \textbf{Eq. 1)}$$

A summary of the RMSE and percent errors for each model are shown in Table 2. The variance and mean of were shown to be the best features in predicting cell life-cycles. While the PCA model had the lowest train error, it had the highest testing error. This hinted that even with simply four features, the model was starting to overfit by a large amount. After comparing our results to the results presented by Severson et al. we noticed very similar trends in our analysis. Our model accuracy was consistently worse than the paper's but this is expected as we are only looking at data with the first 5 cycles. As we input data from more cycles in the model, the accuracy of the mode will increase as well. Thus, it is up to the user of these models to determine the optimal tradeoff between testing time and model accuracy.

	RMSE (cycles)		**Percent Error (%)**	
	Train	Test	Train	Test
PCA Model	320	470	23.3	37.5
Variance Model	344	452	30.3	30.0
Mean Model	352	446	34.1	30.0

Table 2: Performance Summary for Three Different Models
The RMSE and percent errors for each of the different models show similar trends as shown by Severson et al.

Classification

We also consider classification models to be used in scenarios in which predictions are required after fewer cycles, but prediction accuracy is less critical. To this end, we use the same features used to fit the principal component regression (PCR) model, but we compute these features using data from the first ten cycles. Data from the first 5 cycles and first 10 were tested, as Severson et al. indicate that the first 10 or fewer cycles can be considered "early cycles," and using data from the first 10 cycles proved to be more accurate. These results are summarized in Table 3. Two classification models are presented: a binary logistic regression model which classifies the batteries into low-lifetime and high-lifetime groups based on the 50th percentile (788 cycles) of the training data, and a multiclass OVR logistic regression model which classifies the batteries into four groups based on the 25th (495 cycles), 50th (788 cycles), and 75th percentile (966 cycles) of the training data. Bootstrap aggregating (bagging) using 25 base estimators is applied to both models to avoid overfitting and thus increase generalizability. A random forest model was also briefly considered for classification, but it exhibited severe overfitting was deemed unsuitable for this application.

	First 5 Cycles				First 10 Cycles			
	Binary		OVR		Binary		OVR	
	Train	Test	Train	Test	Train	Test	Train	Test
Variance	54.8%	61.3%	34.4%	12.9%	55.9%	48.4%	29.0%	22.6%
ΔQ(V)	64.5%	67.7%	45.2%	38.7%	83.9%	77.4%	59.1%	58.1%
Discharge	78.5%	64.5%	53.8%	38.7%	86.0%	90.3%	63.4%	61.3%
Full	89.2%	83.2%	72.0%	48.4%	91.4%	90.3%	75.3%	71.0%

Table 3. Performance of the classification models when using different subsets of early-cycle data

Binary Classifier

The binary classifier performs poorly when fit to only the variance of $\Delta Q_{10-2}(V)$, as illustrated in fig. 4, with a test accuracy of 48.4% and an area under the receiver operator characteristic (ROC) curve (AUC) of 0.469. This is contrary to the results presented by Severson et al., as they presented a model that was able to obtain up to 97.5% test accuracy using variance of $\Delta Q_{5-4}(V)$ as the only feature. After analyzing their data, we found that they did not use a random split to separate the data into training and test sets, and in fact their secondary test set consisted of a disproportionately large number of high-lifetime cells. This may have artificially inflated their test accuracy.

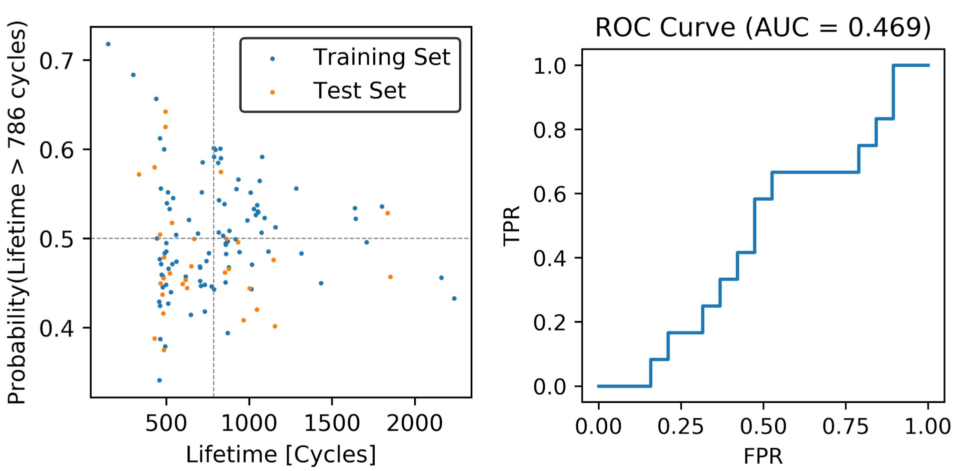


Figure 4. Performance of the binary classifier fitted to only the variance of $\Delta Q_{10-2}(V)$. **(a)** Decision boundary using a probability threshold of 0.5 and a lifetime cutoff of 786 cycles Data points in the upper left and lower right quadrants are misclassified. **(b)** ROC curve for this classifier.

However, the classifier performs much better when it considers the full set of features, as illustrated in fig. 5, with a test accuracy of 90.3% and an AUC of 0.939. This accuracy represents a misclassification of just 3 cells.

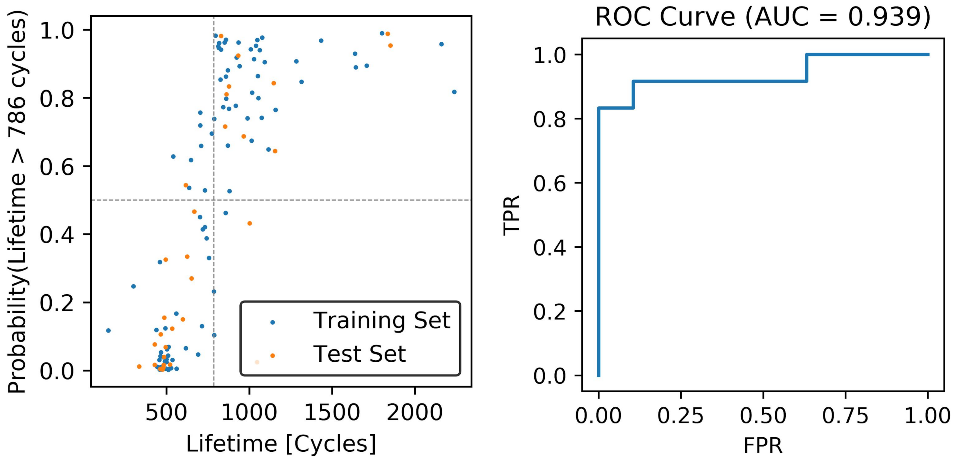


Figure 5. Performance of the binary classifier fitted to the full set of 16 features **(a)** Decision boundary using a probability threshold of 0.5 and a lifetime cutoff of 786 cycles Data points in the upper left and lower right quadrants are misclassified. **(b)** ROC curve for this classifier.

The test accuracy decreases when the model considers the full feature set, which suggests that the model is overfitting to the training data, so we suggest that the binary classifier only consider the $\Delta Q_{10\text{-}2}(V)$ features and discharge features. The performance of the model when fit to various feature sets is summarized in fig. 6.

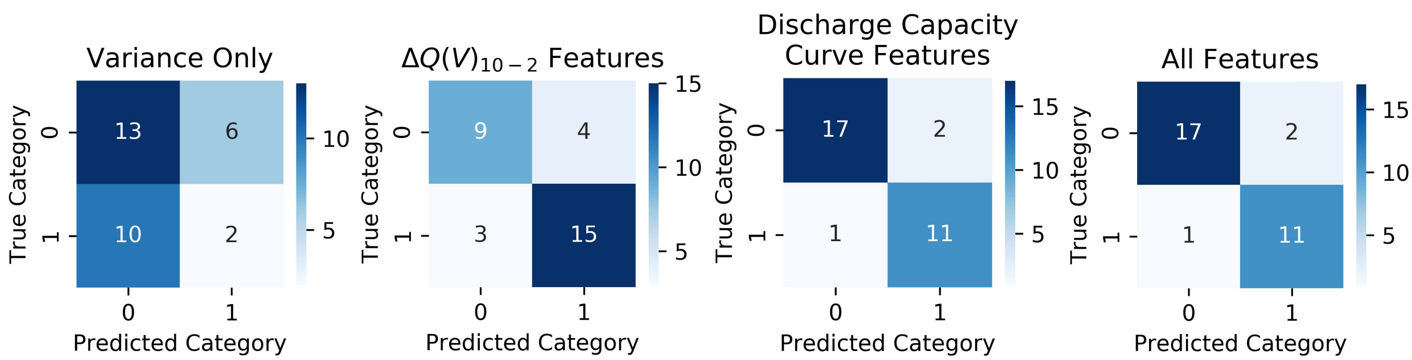


Figure 6. Confusion matrices for the binary classifier fitted to various feature sets

Multiclass Classifier

Similarly, the OVR classifier performs poorly when fit to only the variance of $\Delta Q_{10\text{-}2}(V)$, with a test accuracy of 22.6%. When fit to the full feature set, the classifier performs significantly better when fit to the full feature set, with a test accuracy of 71%. This represents a misclassification of just 9 cells, so we believe this classifier is a promising multiclass classifier if it can be fit to a larger data set. Further domain knowledge may also be required to determine if using quantiles is in fact an appropriate method for calculating the cutoffs between lifetime groups

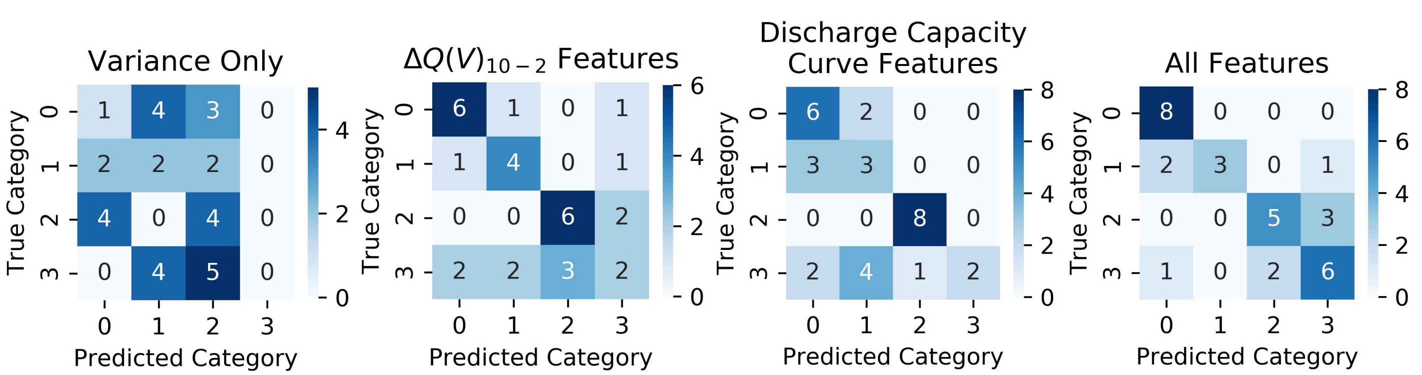


Figure 7. Confusion matrices for the OVR classifier fitted to various feature sets

Conclusion

We have demonstrated a principal component regression model which is able to quantitatively predict cell cycle lifetimes with a mean percent error of 30% using data from the first 5 cycles. We have also presented a binary classifier which classifies cells into low- high-lifetime groups with an accuracy of 90.3%, and a multiclass classifier which classifies cells into 4 groups with an accuracy of 71%. These models are a promising, highly generalizable method for predicting cycle lifetimes from commonly used early-cycle diagnostic data. However, this work was performed using a very limited data set, so further work should be done using a larger data set and different types of batteries to better assess the validity and utility of these models.

References

1. Gür, T.M. Review of electrical energy storage technologies, materials, and systems: challenges and prospects for large-scale grid storage.*Energy Environ. Sci.* **11**, 2696 (2018)
2. Krishan O, Suhag S. An updated review of energy storage systems: Classification and applications in distributed generation power systems incorporating renewable energy resources. *Int J Energy Res*. **43**, 6171 (2019).
3. Christensen, J. & Newman, J. A mathematical model for the lithium-ion negative electrode solid electrolyte interphase. *J. Electrochem. Soc.* **151**, A1977 (2004).
4. Broussely, M. *et al*. Aging mechanism in Li ion cells and calendar life predictions. *J. Power Sources* **97–98**, 13 (2001).
5. Severson, K.A., Attia, P.M., Jin, N. *et al*. Data-driven prediction of battery cycle life before capacity degradation. *Nat. Energy* **4**, 383 (2019).
6. Pedregosa *et al.* Scikit-learn: Machine Learning in Python, *JMLR* **12**, 2825-2830 (2011).

Data Processing

May 10, 2020

1 Load Data

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from scipy import stats
import pickle
```

[2]:
```
batch1 = pickle.load(open(r'./Data/batch1.pkl', 'rb'))
#remove batteries that do not reach 80% capacity
del batch1['b1c8']
del batch1['b1c10']
del batch1['b1c12']
del batch1['b1c13']
del batch1['b1c22']
```

[3]:
```
numBat1 = len(batch1.keys())
numBat1
```

[3]: 41

[4]:
```
batch2 = pickle.load(open(r'./Data/batch2.pkl','rb'))
```

[5]:
```
# There are four cells from batch1 that carried into batch2, we'll remove the␣
↪data from batch2
# and put it with the correct cell from batch1
batch2_keys = ['b2c7', 'b2c8', 'b2c9', 'b2c15', 'b2c16']
batch1_keys = ['b1c0', 'b1c1', 'b1c2', 'b1c3', 'b1c4']
add_len = [662, 981, 1060, 208, 482];
```

[6]:
```
for i, bk in enumerate(batch1_keys):
    batch1[bk]['cycle_life'] = batch1[bk]['cycle_life'] + add_len[i]
    for j in batch1[bk]['summary'].keys():
        if j == 'cycle':
            batch1[bk]['summary'][j] = np.hstack((batch1[bk]['summary'][j],␣
↪batch2[batch2_keys[i]]['summary'][j] + len(batch1[bk]['summary'][j])))
        else:
```

```
            batch1[bk]['summary'][j] = np.hstack((batch1[bk]['summary'][j],␣
↪batch2[batch2_keys[i]]['summary'][j]))
        last_cycle = len(batch1[bk]['cycles'].keys())
        for j, jk in enumerate(batch2[batch2_keys[i]]['cycles'].keys()):
            batch1[bk]['cycles'][str(last_cycle + j)] =␣
↪batch2[batch2_keys[i]]['cycles'][jk]
```

[7]:
```
del batch2['b2c7']
del batch2['b2c8']
del batch2['b2c9']
del batch2['b2c15']
del batch2['b2c16']
```

[8]:
```
numBat2 = len(batch2.keys())
numBat2
```

[8]: 43

[9]:
```
batch3 = pickle.load(open(r'./Data/batch3.pkl','rb'))
# remove noisy channels from batch3
del batch3['b3c37']
del batch3['b3c2']
del batch3['b3c23']
del batch3['b3c32']
del batch3['b3c38']
del batch3['b3c39']
```

[10]:
```
numBat3 = len(batch3.keys())
numBat3
```

[10]: 40

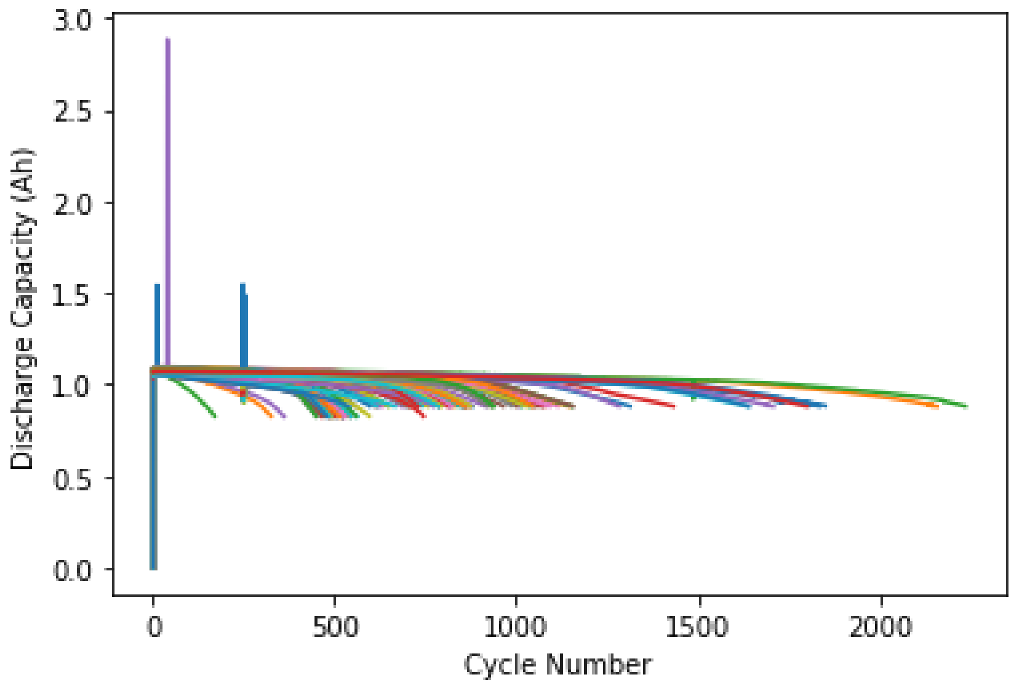
[11]:
```
numBat = numBat1 + numBat2 + numBat3
numBat
```

[11]: 124

[12]:
```
bat_dict = {**batch1, **batch2, **batch3}
```

[13]:
```
for i in bat_dict.keys():
    plt.plot(bat_dict[i]['summary']['cycle'], bat_dict[i]['summary']['QD'])
plt.xlabel('Cycle Number')
plt.ylabel('Discharge Capacity (Ah)')
```

[13]: Text(0, 0.5, 'Discharge Capacity (Ah)')



1.0.1 Train and Test Split

If you are interested in using the same train/test split as the paper, use the indices specified below

[14]:
```
test_ind = np.hstack((np.arange(0,(numBat1+numBat2),2),83))
train_ind = np.arange(1,(numBat1+numBat2-1),2)
secondary_test_ind = np.arange(numBat-numBat3,numBat);
```

2 EDA / Cleaning

[17]:
```
cellid = 'b1c0'
cell = bat_dict[cellid]
cell_cycle = bat_dict[cellid]['cycles']
cell_summary = bat_dict[cellid]['summary']
```

[18]:
```
cell.keys()
```

[18]:
```
dict_keys(['cycle_life', 'charge_policy', 'summary', 'cycles'])
```

[19]:
```
summary_keys = list(cell_summary.keys())
cycle_keys = list(cell_cycle['0'].keys())
print(summary_keys)
```

```
print(cycle_keys)
```

```
['IR', 'QC', 'QD', 'Tavg', 'Tmin', 'Tmax', 'chargetime', 'cycle']
['I', 'Qc', 'Qd', 'Qdlin', 'T', 'Tdlin', 'V', 'dQdV', 't']
```

2.1 Linear Regression Features

2.1.1 ΔQ features

[20]:
```
delq_mean = np.zeros(len(bat_dict))
delq_min = np.zeros(len(bat_dict))
delq_var = np.zeros(len(bat_dict))
delq_skew = np.zeros(len(bat_dict))
delq_kurtosis = np.zeros(len(bat_dict))

for n, cell in enumerate(bat_dict.values()):
    cell = cell['cycles']
    delq_arr = cell['100']['Qdlin'] - cell['10']['Qdlin']
    delq_mean[n] = np.mean(delq_arr)
    delq_min[n] = np.min(delq_arr)
    delq_var[n] = np.var(delq_arr, ddof=1)
    delq_skew[n] = stats.skew(delq_arr)
    delq_kurtosis[n] = stats.kurtosis(delq_arr)
```

2.1.2 Discharge capacity fade curve features

[21]:
```
slope_cycle2 = np.zeros(len(bat_dict))
intercept_cycle2 = np.zeros(len(bat_dict))
slope_cycle91 = np.zeros(len(bat_dict))
intercept_cycle91 = np.zeros(len(bat_dict))
qd_cycle2 = np.zeros(len(bat_dict))
del_qd = np.zeros(len(bat_dict))
qd_cycle100 = np.zeros(len(bat_dict))

for n, cell in enumerate(bat_dict.values()):
    cell = cell['summary']
    x = np.arange(2,101)
    q = cell['QD'][2:101]
    slope_cycle2[n], intercept_cycle2[n], r_value, p_value, std_err = stats.
↪linregress(x, q)
    x = np.arange(91,101)
    q = cell['QD'][91:101]
    slope_cycle91[n], intercept_cycle91[n], r_value, p_value, std_err = stats.
↪linregress(x, q)
    qd_cycle2[n] = cell['QD'][2]
```

```
    del_qd[n] = max(cell['QD']) - cell['QD'][2]
    qd_cycle100[n] = cell['QD'][100]
```

2.1.3 Other Features

[22]:
```
ir_cycle2 = np.zeros(len(bat_dict))
chargetime = np.zeros(len(bat_dict))
del_ir = np.zeros(len(bat_dict))
ir_min = np.zeros(len(bat_dict))
temp_min = np.zeros(len(bat_dict))
temp_max = np.zeros(len(bat_dict))

for n, cell in enumerate(bat_dict.values()):
    cell = cell['summary']
    ir_cycle2[n] = cell['IR'][2]
    del_ir[n] = cell['IR'][100] - cell['IR'][2]
    ir_min[n] = min(cell['IR'][2:100])
    chargetime[n] = np.mean(cell['chargetime'][2:7])
    temp_min[n] = min(cell['Tmin'][2:100])
    temp_max[n] = max(cell['Tmax'][2:100])
```

[23]:
```
temp_integral = np.zeros(len(bat_dict))
delq_2v = np.ones(len(bat_dict))
```

2.1.4 Build feature matrix

[24]:
```
delq_features = np.vstack((delq_min, delq_mean, delq_var, delq_skew,␣
 ↪delq_kurtosis, delq_2v))
delq_features = np.log10(np.abs(delq_features))
discharge_features = np.vstack((slope_cycle2, intercept_cycle2, slope_cycle91,␣
 ↪intercept_cycle91,
                                qd_cycle2, del_qd, qd_cycle100))
other_features = np.vstack((chargetime, temp_max, temp_min, temp_integral,␣
 ↪ir_cycle2, ir_min, del_ir))

all_features = np.vstack((delq_features, discharge_features, other_features))
all_features.shape
```

[24]: (20, 124)

[25]:
```
feature_df = pd.DataFrame(all_features)
feature_df.columns=bat_dict.keys()
feature_df.head()
```

[25]:
```
       b1c0      b1c1      b1c2      b1c3      b1c4      b1c5      b1c6  \
0 -2.050261 -2.045150 -1.986994 -1.703321 -1.837397 -1.583636 -1.414229
1 -2.553712 -2.428027 -2.383163 -2.104748 -2.202790 -1.937981 -1.791400
2 -5.051092 -5.135342 -4.951448 -4.385913 -4.604345 -4.155398 -3.770848
3 -0.138643 -0.499000 -0.359039 -0.322093 -0.456945 -0.849869 -0.405104
4 -0.040960  0.016426  0.069572  0.050851  0.133910  0.076409  0.082792

       b1c7      b1c9     b1c11  …     b3c33     b3c34     b3c35     b3c36  \
0 -1.428857 -1.528095 -1.598897  … -1.687756 -1.661724 -1.628372 -1.627409
1 -1.784616 -1.913511 -2.045593  … -2.078782 -2.013579 -1.996052 -1.989872
2 -3.821400 -3.978210 -4.098637  … -4.370155 -4.292966 -4.217803 -4.236271
3 -0.510463 -0.337653 -0.217875  … -0.415844 -0.778340 -0.528778 -0.657779
4  0.046264  0.080953  0.073224  …  0.015549  0.062758  0.057702  0.047144

      b3c40     b3c41     b3c42     b3c43     b3c44     b3c45
0 -1.655369 -1.608813 -2.233379 -1.699730 -1.584652 -1.771168
1 -2.039456 -1.995593 -3.171143 -2.150195 -1.945603 -2.135138
2 -4.309271 -4.198915 -4.384516 -4.151722 -4.132646 -4.504979
3 -0.480042 -0.511520  0.281782 -0.477362 -0.665259 -0.493829
4  0.022415  0.040280  0.437447 -0.283129  0.090173  0.069021

[5 rows x 124 columns]
```

[26]:
```
feature_df.to_csv('featurematrix_regression.csv')
```

2.2 Classification Features

2.2.1 ΔQ Features

[15]:
```
delq_mean = np.zeros(len(bat_dict))
delq_min = np.zeros(len(bat_dict))
delq_var = np.zeros(len(bat_dict))
delq_skew = np.zeros(len(bat_dict))
delq_kurtosis = np.zeros(len(bat_dict))

for n, cell in enumerate(bat_dict.values()):
    cell = cell['cycles']
    delq_arr = cell['20']['Qdlin'] - cell['2']['Qdlin']
    delq_mean[n] = np.mean(delq_arr)
    delq_min[n] = np.min(delq_arr)
    delq_var[n] = np.var(delq_arr, ddof=1)
    delq_skew[n] = stats.skew(delq_arr)
    delq_kurtosis[n] = stats.kurtosis(delq_arr)
```

2.2.2 Discharge capacity fade curve features

[16]:
```
slope_cycle2 = np.zeros(len(bat_dict))
intercept_cycle2 = np.zeros(len(bat_dict))
qd_cycle2 = np.zeros(len(bat_dict))
del_qd = np.zeros(len(bat_dict))
qd_cycle5 = np.zeros(len(bat_dict))

for n, cell in enumerate(bat_dict.values()):
    cell = cell['summary']
    x = np.arange(2,21)
    q = cell['QD'][2:21]
    slope_cycle2[n], intercept_cycle2[n], r_value, p_value, std_err = stats.
 ↪linregress(x, q)
    qd_cycle2[n] = cell['QD'][2]
    del_qd[n] = max(cell['QD'][:21]) - cell['QD'][2]
    qd_cycle5[n] = cell['QD'][20]
```

2.2.3 Other Features

[39]:
```
ir_cycle2 = np.zeros(len(bat_dict))
chargetime = np.zeros(len(bat_dict))
del_ir = np.zeros(len(bat_dict))
ir_min = np.zeros(len(bat_dict))
temp_min = np.zeros(len(bat_dict))
temp_max = np.zeros(len(bat_dict))

for n, cell in enumerate(bat_dict.values()):
    cell = cell['summary']
    ir_cycle2[n] = cell['IR'][20]
    del_ir[n] = cell['IR'][20] - cell['IR'][2]
    ir_min[n] = min(cell['IR'][:21])
    chargetime[n] = np.mean(cell['chargetime'][:21])
    temp_min[n] = min(cell['Tmin'][:21])
    temp_max[n] = max(cell['Tmax'][:21])
```

2.2.4 Build and export feature matrix

[40]:
```
delq_features = np.vstack((delq_min, delq_mean, delq_var, delq_skew,␣
 ↪delq_kurtosis))
delq_features = np.log10(np.abs(delq_features))
discharge_features = np.vstack((slope_cycle2, intercept_cycle2, qd_cycle2,␣
 ↪del_qd, qd_cycle5))
```

```
other_features = np.vstack((chargetime, temp_max, temp_min, ir_cycle2, ir_min,␣
↪del_ir))

all_features = np.vstack((delq_features, discharge_features, other_features))
all_features.shape
```

[40]: (16, 124)

[41]:
```
feature_df = pd.DataFrame(all_features)
feature_df.columns=bat_dict.keys()
feature_df.head()
```

[41]:
```
       b1c0      b1c1      b1c2      b1c3      b1c4      b1c5      b1c6  \
0 -4.523449 -3.926048 -3.685959 -4.960902 -3.800357 -4.103946 -4.040327
1 -2.727409 -2.614204 -2.730973 -2.615284 -2.667526 -2.597706 -2.593257
2 -5.536965 -5.329131 -5.352623 -5.169849 -5.335286 -5.566559 -5.492307
3  0.475910  0.487980  0.574914  0.316841  0.171448 -0.428845  0.161704
4  1.050214  1.078159  1.216015  0.602496  0.171331 -0.839865  0.664793

       b1c7      b1c9     b1c11  …     b3c33     b3c34     b3c35     b3c36  \
0 -4.838352 -4.251121 -3.711445  … -4.330283 -3.729600 -4.230939 -4.666520
1 -2.769604 -2.638674 -2.699830  … -2.829509 -2.931252 -2.945310 -2.859951
2 -5.837231 -5.485907 -5.515717  … -5.699849 -5.969588 -5.800616 -5.840052
3  0.219657  0.191106  0.057883  …  0.350240  0.258793  0.414233  0.335193
4  0.718708  0.467162 -0.130905  …  0.770106  0.708900  0.869710  0.786164

      b3c40     b3c41     b3c42     b3c43     b3c44     b3c45
0 -5.226915 -4.021168 -4.006196 -3.805935 -5.007420 -5.415515
1 -2.610444 -2.656759 -2.692621 -2.337513 -2.838654 -2.792955
2 -5.354198 -5.377132 -5.466114 -4.465251 -5.981095 -5.649811
3  0.226461  0.314611  0.230280  0.295607  0.041934  0.273017
4  0.493240  0.651329  0.529454  0.456407  0.387399  0.570330

[5 rows x 124 columns]
```

[42]:
```
feature_df.to_csv('featurematrix_classification3.csv')
```

2.3 Cell Lifetimes

[34]:
```
lifetimes = np.zeros(len(bat_dict))
for n, cell in enumerate(bat_dict.values()):
    lifetimes[n] = cell['cycle_life'].flatten()[0]
```

[35]:
```
lifetime_df = pd.DataFrame(lifetimes).T
lifetime_df.columns = bat_dict.keys()
lifetime_df
```

[35]:
```
    b1c0    b1c1    b1c2    b1c3    b1c4    b1c5   b1c6   b1c7    b1c9  \
0  1852.0  2160.0  2237.0  1434.0  1709.0  1074.0  636.0  870.0  1054.0

   b1c11  …   b3c33   b3c34   b3c35  b3c36  b3c40  b3c41   b3c42   b3c43  \
0  788.0  …  1284.0  1158.0  1093.0  923.0  796.0  786.0  1642.0  1046.0

   b3c44   b3c45
0  940.0  1801.0

[1 rows x 124 columns]
```

[ ]:
```
lifetime_df.to_csv('lifetimematrix.csv')
```

Final Project Analysis

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.decomposition import PCA
from sklearn import linear_model as lm
from sklearn.linear_model import Lasso, LassoCV
from sklearn.pipeline import Pipeline
```

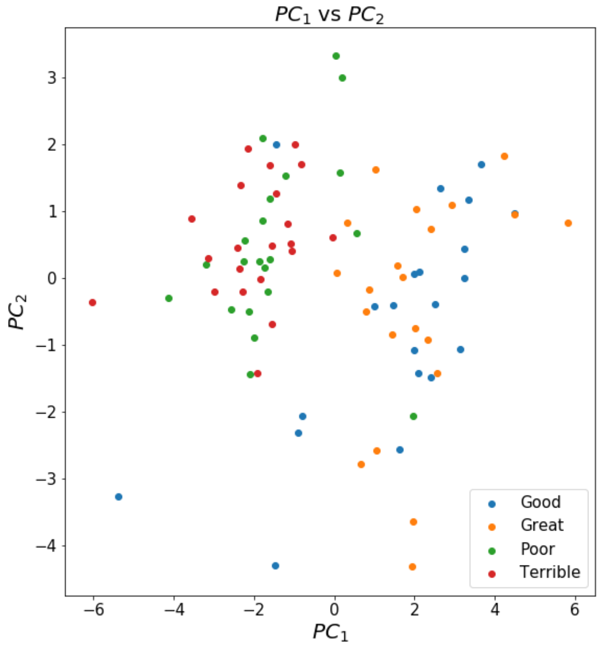
In [2]:
```
# Importing Processed Data
# Rows are samples, Columns are features
Feature = pd.read_csv('featurematrix_classification.csv')
Lifetime = pd.read_csv('lifetimematrix.csv')
X_train = Feature.iloc[:,1:85].T
X_test = Feature.iloc[:,85:].T
Y_train = Lifetime.iloc[:,1:85].T
Y_test = Lifetime.iloc[:,85:].T
# Standardizing Data
X_train = ((X_train-X_train.mean())/X_train.std()).fillna(0)
X_test = ((X_test-X_test.mean())/X_test.std()).fillna(0)
```

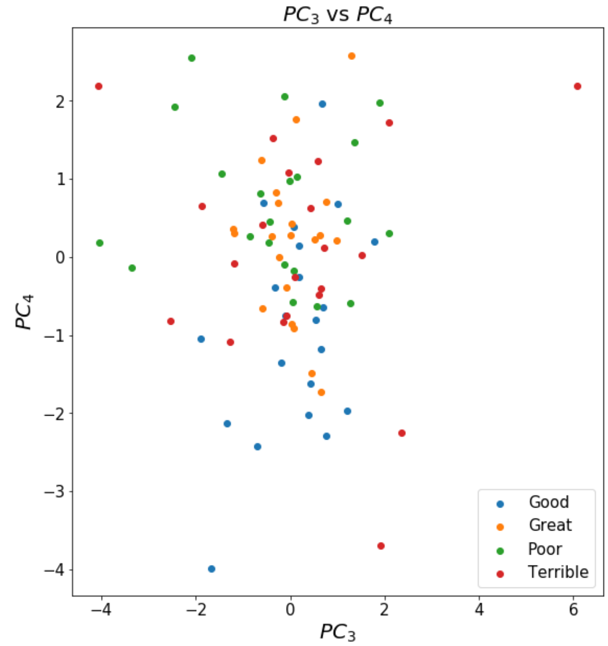
In [3]:
```
per_25 = np.percentile(Y_train, 25)
per_50 = np.percentile(Y_train, 50)
per_75 = np.percentile(Y_train, 75)
print('Terrible Batteries: # of cycles < %d' %(per_25))
print('Poor Batteries: %d < # of cycles < %d' %(per_25, per_50))
print('Good Batteries: %d < # of cycles < %d' %(per_50, per_75))
print('Great Batteries: # of cycles > %d' %(per_75))
```

```
Terrible Batteries: # of cycles < 480
Poor Batteries: 480 < # of cycles < 547
Good Batteries: 547 < # of cycles < 801
Great Batteries: # of cycles > 801
```

In [4]:
```
classified = np.array([])
for i in np.arange(Y_train.shape[0]):
    if Y_train.iloc[i,0] < per_25:
        classified = np.append(classified, 1)
    elif Y_train.iloc[i,0] < per_50:
        classified = np.append(classified, 2)
    elif Y_train.iloc[i,0] < per_75:
        classified = np.append(classified, 3)
    else:
        classified = np.append(classified, 4)
classified = pd.DataFrame(data=classified, columns=['Number'])
```

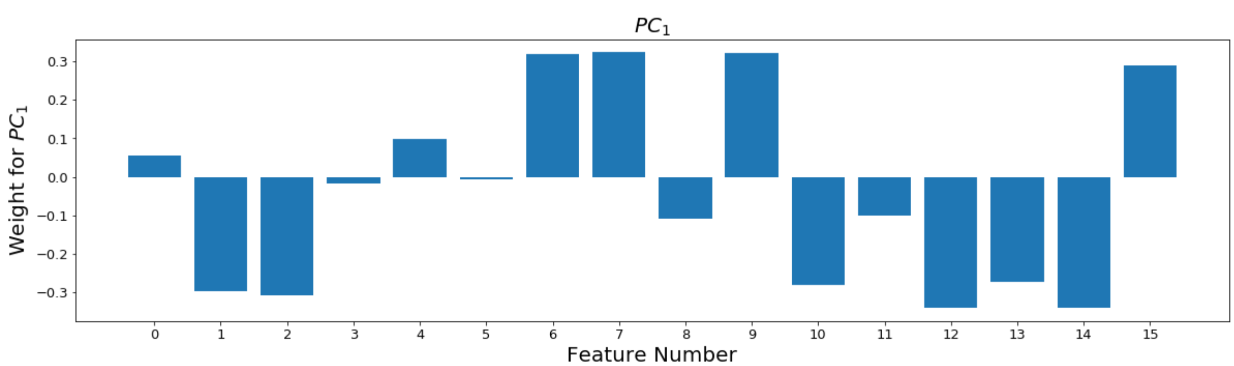
In [5]:
```
# Performing PCA Analysis, and Plotting PC1 vs PC2
u, s, vt = np.linalg.svd(X_train, full_matrices = False)
P = u @ np.diag(s)
first_4_pcs = pd.DataFrame(P[:,0:4], columns=['PC 1', 'PC 2', 'PC 3',
'PC 4'])
pcaDF = pd.concat([first_4_pcs, classified], axis=1)
pcaDF = pcaDF.replace({'Number': {1: 'Terrible', 2: 'Poor', 3: 'Good',
4: 'Great'}})
# Plotting Solutions
fig = plt.figure(figsize=(20, 10)) # Initialize Figure Size
plt.subplot(1, 2, 1) # PC1 vs PC2
groups = pcaDF.groupby("Number")
for name, group in groups:
    plt.scatter(group["PC 1"], group["PC 2"], marker="o", label=name);
plt.legend(loc="lower right",fontsize = 15);
plt.title('$PC_1$ vs $PC_2$', fontsize=20);
plt.xticks(fontsize=15);
plt.yticks(fontsize=15);
plt.xlabel('$PC_1$', fontsize=20)
plt.ylabel('$PC_2$', fontsize=20)
plt.subplot(1, 2, 2) # PC3 vs PC4
for name, group in groups:
    plt.scatter(group["PC 3"], group["PC 4"], marker="o", label=name);
plt.legend(loc="lower right", fontsize = 15);
plt.title('$PC_3$ vs $PC_4$', fontsize=20);
plt.xlabel('$PC_3$', fontsize=20);
plt.ylabel('$PC_4$', fontsize=20);
plt.xticks(fontsize=15);
plt.yticks(fontsize=15);
plt.savefig('PCA_Analysis.png')
```





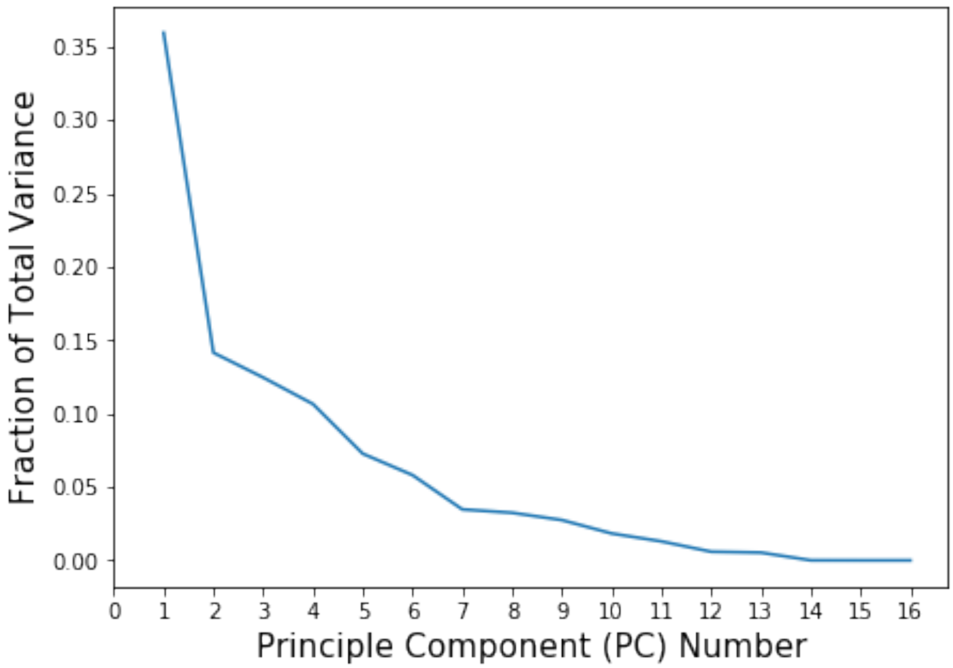
PC1 seems to be the best indicator of the quality of the batteries. PC1 seems to differentiate beetweeen top and botton tier batteries. PC3 and PC4 do not seem to be correlated, and are poor differentiators of the quality of the battery.

In [6]:
```
# Contributions of each feature results on PC1
plt.figure(figsize=(20, 5))
plt.bar(X_train.columns, vt[0, :])
plt.xticks(X_train.columns, fontsize=13);
plt.yticks(fontsize=13);
plt.title('$PC_1$', fontsize=20);
plt.xlabel('Feature Number', fontsize=20);
plt.ylabel('Weight for $PC_1$', fontsize=20);
plt.savefig('PCA_Weights.png')
```



Important features include minimum, mean, variance, value at 2V, intercept, and difference between max

In [7]:
```
plt.figure(figsize=(7, 5))
plt.plot(np.arange(1,P.shape[1]+1), s**2 / sum(s**2));
plt.xlabel('Principle Component (PC) Number', fontsize=15);
plt.ylabel('Fraction of Total Variance', fontsize=15);
plt.xticks(np.arange(17), fontsize=10);
plt.yticks(fontsize=10);
plt.savefig('Total_Variance.png')
```



Many of the features are redundant information and may be adding noise. We thus remove some PCs

In [8]:
```
updated_X_train = pd.DataFrame(P[:,0:4], columns=['PC 1', 'PC 2', 'PC 3', 'PC 4'])

# Fitting PCA Model
linear_model = lm.LinearRegression(fit_intercept=True)
linear_model.fit(updated_X_train, Y_train)
y_fitted_PC = linear_model.predict(updated_X_train)
predictions = pd.concat([Y_train.reset_index(), pd.DataFrame(y_fitted_PC, columns=['Prediction_PCA'])],
                        axis=1).drop(columns=['index']).rename(columns={0: 'Actual'})
# Fitting Minimum Model
min_X_train = X_train.iloc[:,0:1]
linear_model.fit(min_X_train, Y_train)
y_fitted_min = linear_model.predict(min_X_train)
predictions = pd.concat([predictions, pd.DataFrame(y_fitted_min, columns=['Prediction_Min'])], axis=1)
# Fitting Mean Model
mean_X_train = X_train.iloc[:,1:2]
linear_model.fit(mean_X_train, Y_train)
y_fitted_mean = linear_model.predict(mean_X_train)
predictions = pd.concat([predictions, pd.DataFrame(y_fitted_mean, columns=['Prediction_Mean'])], axis=1)
# Fitting Var Model
var_X_train = X_train.iloc[:,2:3]
linear_model.fit(var_X_train, Y_train)
y_fitted_var = linear_model.predict(var_X_train)
predictions = pd.concat([predictions, pd.DataFrame(y_fitted_var, columns=['Prediction_Var'])], axis=1)
predictions
```

Out[8]:

	Actual	Prediction_PCA	Prediction_Min	Prediction_Mean	Prediction_Var
0	1852.0	708.452752	719.979797	687.491749	706.857706
1	2160.0	705.635007	692.115612	678.572147	677.750750
2	2237.0	704.894401	681.963947	706.713154	665.000561
3	1434.0	838.213840	672.688072	678.098768	659.163152
4	1709.0	807.203715	679.398015	691.272738	713.290582
...	...	...	...	...	...
79	462.0	599.754395	663.306219	767.193036	667.399796
80	457.0	542.535440	658.363161	650.363750	604.317061
81	487.0	555.274071	656.996922	636.084880	587.791450
82	429.0	612.643307	663.831292	669.951039	641.358228
83	713.0	582.379474	667.603085	681.581823	692.434874

84 rows × 5 columns

In [9]:

```
# Finding RMSE and % error for each linear fit.
def rmse(predicted, actual):
    return np.sqrt(np.mean((actual - predicted)**2))
def per_err(predicted, actual):
    return np.mean(np.abs(actual - predicted)/actual*100)
errors = np.array([0])
per_error = np.array([0])
for i in np.arange(1, predictions.shape[1]):
    errors = np.append(errors, rmse(predictions.iloc[:,i], predictions
.iloc[:,0]))
    per_error = np.append(per_error, per_err(predictions.iloc[:,i], pr
edictions.iloc[:,0]))
comb_errors = np.vstack((errors, per_error))
Error_Table = pd.DataFrame(comb_errors, columns=predictions.columns).r
ename(index={0: 'RMSE', 1: 'Percent Error'})
Error_Table
```

Out[9]:

	Actual	Prediction_PCA	Prediction_Min	Prediction_Mean	Prediction_Var
RMSE	0.0	319.686179	359.312852	351.552979	344.455040
Percent Error	0.0	23.322014	37.772958	34.095084	30.300166

In [10]:
```
# Performing Same analysis on Test Data

# Performing PCA Analysis, and Plotting PC1 vs PC2
u, s, vt = np.linalg.svd(X_test, full_matrices = False)
P = u @ np.diag(s)
updated_X_test = pd.DataFrame(P[:,0:4], columns=['PC 1', 'PC 2', 'PC 3
', 'PC 4'])

# Fitting PCA Model
linear_model = lm.LinearRegression(fit_intercept=True)
linear_model.fit(updated_X_train, Y_train)
y_fitted_PC = linear_model.predict(updated_X_test)
predictions = pd.concat([Y_test.reset_index(), pd.DataFrame(y_fitted_P
C, columns=['Prediction_PCA'])],
                        axis=1).drop(columns=['index']).rename(columns
={0: 'Actual'})
# Fitting Minimum Model
min_X_test = X_test.iloc[:,0:1]
linear_model.fit(min_X_train, Y_train)
y_fitted_min = linear_model.predict(min_X_test)
predictions = pd.concat([predictions, pd.DataFrame(y_fitted_min, colum
ns=['Prediction_Min'])], axis=1)
# Fitting Mean Model
mean_X_test = X_test.iloc[:,1:2]
linear_model.fit(mean_X_train, Y_train)
y_fitted_mean = linear_model.predict(mean_X_test)
predictions = pd.concat([predictions, pd.DataFrame(y_fitted_mean, colu
mns=['Prediction_Mean'])], axis=1)
# Fitting Var Model
var_X_test = X_test.iloc[:,2:3]
linear_model.fit(var_X_train, Y_train)
y_fitted_var = linear_model.predict(var_X_test)
predictions = pd.concat([predictions, pd.DataFrame(y_fitted_var, colum
ns=['Prediction_Var'])], axis=1)
errors = np.array([0])
per_error = np.array([0])
for i in np.arange(1, predictions.shape[1]):
    errors = np.append(errors, rmse(predictions.iloc[:,i], predictions
.iloc[:,0]))
    per_error = np.append(per_error, per_err(predictions.iloc[:,i], pr
edictions.iloc[:,0]))
comb_errors = np.vstack((errors, per_error))
Error_Table = pd.DataFrame(comb_errors, columns=predictions.columns).r
ename(index={0: 'RMSE', 1: 'Percent Error'})
Error_Table
```

Out[10]:

	Actual	Prediction_PCA	Prediction_Min	Prediction_Mean	Prediction_Var
RMSE	0.0	470.470354	438.501156	446.348893	451.673564
Percent Error	0.0	37.527539	28.947113	29.963096	30.004618

In [ ]:

Logistic Regression Classifier

May 10, 2020

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy import stats
import seaborn as sns
from sklearn.linear_model import LogisticRegression
from sklearn.linear_model import LogisticRegressionCV
from sklearn.multiclass import OneVsRestClassifier
from sklearn import metrics
from sklearn import model_selection
from sklearn import preprocessing
from sklearn.decomposition import PCA
from sklearn.ensemble import BaggingClassifier
```

1 Load Data

[2]:
```
# load feature matrix
feature_df = pd.read_csv('../data/featurematrix_classification2.csv')
feature_df.drop(columns='Unnamed: 0', inplace=True)
# feature_df = feature_df.drop([5,16]).reset_index(drop=True)
feature_df.head()
```

[2]:
```
       b1c0      b1c1      b1c2      b1c3      b1c4      b1c5      b1c6  \
0 -4.181804 -3.969042 -2.788997 -4.700016 -4.132990 -3.905395 -4.466846
1 -2.584744 -2.540101 -2.697642 -2.572648 -2.527145 -2.452704 -2.438709
2 -5.514008 -5.519434 -5.647117 -5.553178 -5.307388 -5.250376 -5.173391
3  0.207469 -0.149255 -0.407511 -0.504164  0.055138 -0.295200  0.186631
4  0.770794  0.457356  0.370178 -0.298181  0.157237 -0.434155  0.665555

       b1c7      b1c9     b1c11  …     b3c33     b3c34     b3c35     b3c36  \
0 -4.970406 -3.884769 -3.948446  … -4.045138 -3.987800 -5.067957 -4.643561
1 -2.730548 -2.442842 -2.425090  … -2.531871 -2.639872 -2.657420 -2.534023
2 -5.874992 -5.128251 -4.781530  … -4.983046 -5.217281 -5.163107 -4.986688
3 -0.163842  0.077411  0.405436  …  0.410522  0.419143  0.452853  0.445606
4  0.428794  0.325888  0.850656  …  0.837771  0.884494  0.929203  0.930771
```

```
      b3c40     b3c41     b3c42     b3c43     b3c44     b3c45
0 -4.675587 -4.547703 -3.900646 -2.415616 -4.571223 -5.265527
1 -2.347406 -2.421160 -2.535104 -4.090369 -2.594459 -2.581789
2 -4.734173 -4.722736 -5.167258 -5.719654 -5.365331 -5.094440
3  0.341317  0.363618  0.266625 -0.138596  0.263355  0.413564
4  0.733024  0.724385  0.640506 -1.321118  0.664487  0.861365

[5 rows x 124 columns]
```

[3]:
```
# load lifetime matrix
lifetime_df = pd.read_csv('../data/lifetimematrix.csv')
lifetime_df.drop(columns='Unnamed: 0', inplace=True)
lifetime_df
```

[3]:
```
     b1c0    b1c1    b1c2    b1c3    b1c4    b1c5   b1c6   b1c7    b1c9  \
0  1852.0  2160.0  2237.0  1434.0  1709.0  1074.0  636.0  870.0  1054.0

   b1c11  …   b3c33   b3c34   b3c35  b3c36  b3c40  b3c41   b3c42   b3c43  \
0  788.0  …  1284.0  1158.0  1093.0  923.0  796.0  786.0  1642.0  1046.0

   b3c44   b3c45
0  940.0  1801.0

[1 rows x 124 columns]
```

[4]:
```
# convert data to numpy arrays/matrices
features = np.array(feature_df).T
lifetimes = np.array(lifetime_df).flatten()
```

[5]:
```
features.shape, lifetimes.shape
```

[5]:
```
((124, 16), (124,))
```

[6]:
```
# standardize data
features = preprocessing.scale(features, axis=0)
```

2 Test/Train Split

[7]:
```
# 75/25 train/test split
xtrain, xtest, ytrain, ytest = model_selection.train_test_split(features,␣
↪lifetimes, test_size=0.25)
```

[8]:
```
# # test/train indices specified in the original paper
# train_idx = np.arange(84)
# test_idx = np.arange(84,124)
```

[9]:
```
# xtrain, ytrain = features[train_idx,:], lifetimes[train_idx]
# xtest, ytest = features[test_idx,:], lifetimes[test_idx]
```

[10]:
```
xtrain.shape, ytrain.shape, xtest.shape, ytest.shape
```

[10]:
```
((93, 16), (93,), (31, 16), (31,))
```

3 Feature Sets

[11]:
```
# variance of  ΔQ(V) only
xtrain_var = xtrain[:,2].reshape(-1,1)
xtest_var = xtest[:,2].reshape(-1,1)
xtrain_var.shape, xtest_var.shape
```

[11]: ((93, 1), (31, 1))

[12]:
```
# all ΔQ(V) features
xtrain_delq, xtest_delq = xtrain[:,:5], xtest[:,:5]
xtrain_delq.shape, xtest_delq.shape
```

[12]: ((93, 5), (31, 5))

[13]:
```
# ΔQ(V) and discharge capacity fade curve features
xtrain_fade, xtest_fade = xtrain[:,:10], xtest[:,:10]
xtrain_fade.shape, xtest_fade.shape
```

[13]: ((93, 10), (31, 10))

4 Lifetime Classification Encoder

[14]:
```
# Classifies batteries into lifetime categories based on the specified number␣
↪thresholds
def classify_lifetimes(yvals, thresholds):
    thresh = np.array(thresholds).flatten()
    thresh = np.append(thresh, max(yvals))
    classes = np.array([])
    for y in yvals:
        for i in range(len(thresh)):
            if y <= thresh[i]:
                classes = np.append(classes, i)
                break
    return classes

# Calculates the cutoffs for a given number of quantiles
def get_percentiles(yvals, n):
    percentiles = np.arange(0, 1, 1/n)*100
    percentiles = percentiles[1:]
    return np.percentile(yvals, percentiles)
```

5 Binary Logistic Regression Model

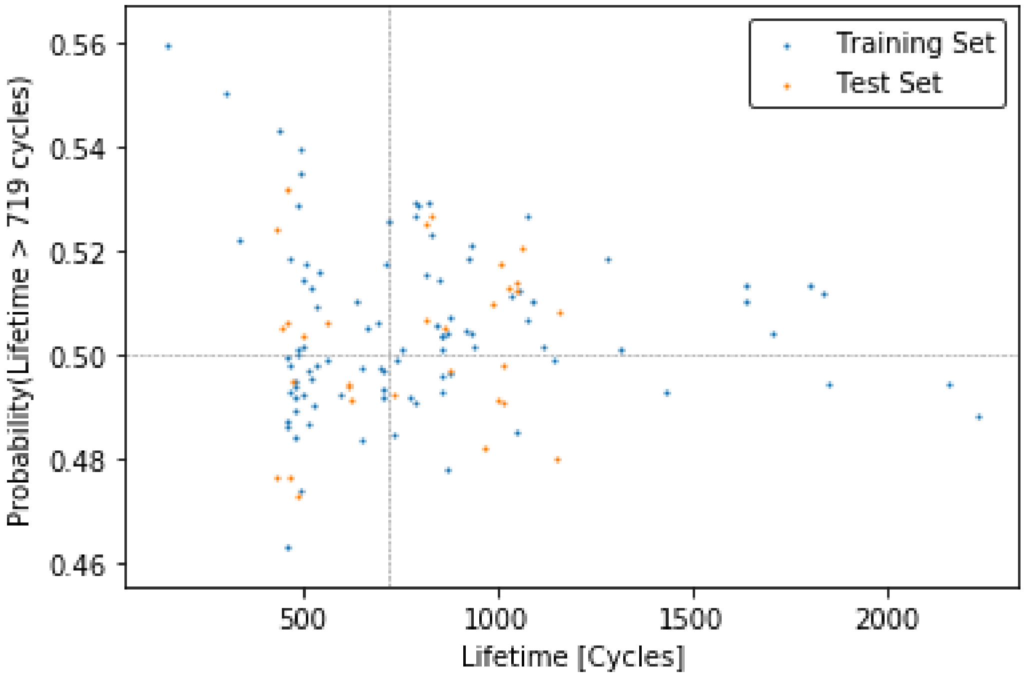
5.1 Variance Only

[15]:
```
xtrain_curr, xtest_curr = xtrain_var, xtest_var

threshold = get_percentiles(ytrain, 2)
ytrain_class = classify_lifetimes(ytrain, threshold)
ytest_class = classify_lifetimes(ytest, threshold)
print(f'Lifetime Cutoff: {threshold}\n')

model = BaggingClassifier(LogisticRegression(), n_estimators=25, oob_score=True)

model.fit(xtrain_curr, ytrain_class)
ytrain_pred = model.predict(xtrain_curr)
ytrain_score = model.predict_proba(xtrain_curr)[:,1]
ytest_pred = model.predict(xtest_curr)
ytest_score = model.predict_proba(xtest_curr)[:,1]

acc_var_train = metrics.accuracy_score(ytrain_class, ytrain_pred)
acc_var = metrics.accuracy_score(ytest_class, ytest_pred)
conf_var_train = metrics.confusion_matrix(ytrain_class, ytrain_pred)
conf_var = metrics.confusion_matrix(ytest_class, ytest_pred)
auc_var_train = metrics.roc_auc_score(ytrain_class, ytrain_score)
auc_var = metrics.roc_auc_score(ytest_class, ytest_score)
print(f'Training Accuracy: {acc_var_train}')
print(f'Test Accuracy: {acc_var}')
print(f'Training AUC: {auc_var_train}')
print(f'Test AUC {auc_var}')
```

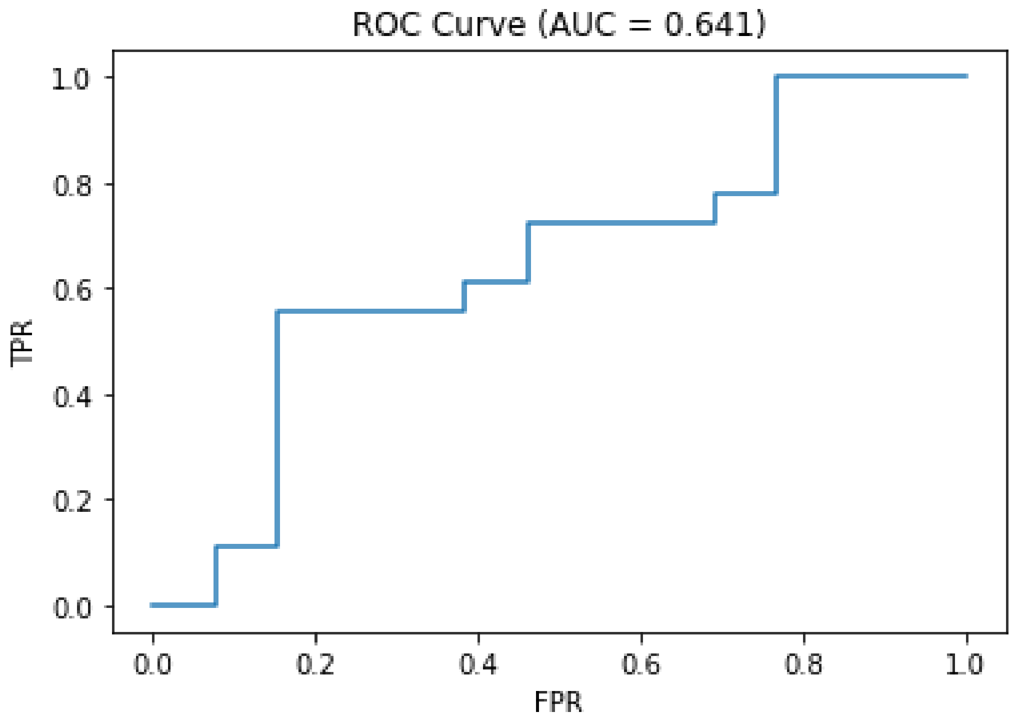
```
Lifetime Cutoff: [719.]

Training Accuracy: 0.6344086021505376
Test Accuracy: 0.5806451612903226
Training AUC: 0.5781683626271971
Test AUC 0.641025641025641
```

[16]:
```
# fig, ax = plt.subplots(figsize=(3,3), dpi=600)
fig,ax = plt.subplots()
ax.scatter(ytrain, ytrain_score, s=1, label='Training Set')
ax.scatter(ytest, ytest_score, s=1, label='Test Set')
ax.axhline(0.5, c='grey', ls='--', lw=0.5)
ax.axvline(int(threshold[0]), c='grey', ls='--', lw=0.5)
ax.set_xlabel('Lifetime [Cycles]')
ax.set_ylabel(f'Probability(Lifetime > {int(threshold[0])} cycles)')
ax.legend(edgecolor='k');
```

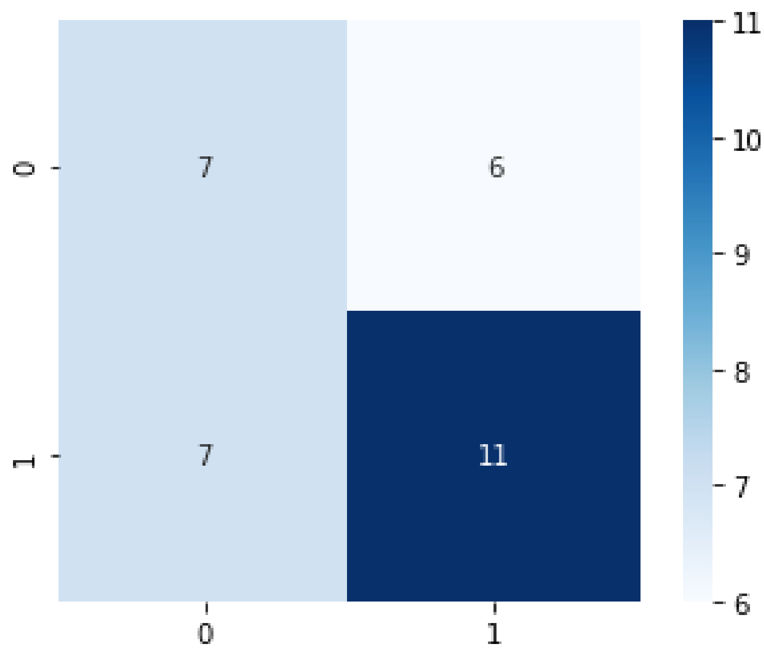
```
# plt.savefig('../figures/set2_decision_var.png', bbox_inches='tight')
```



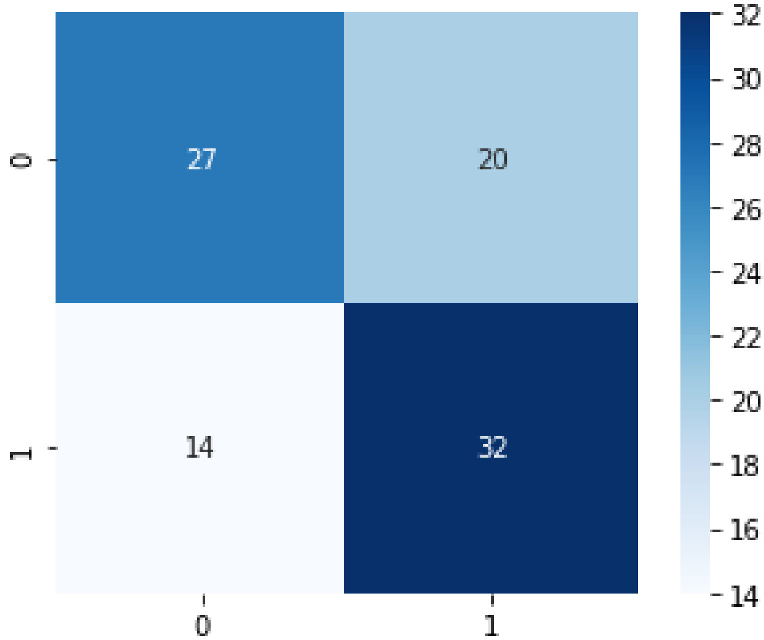
[17]:
```
fpr, tpr, thresholds = metrics.roc_curve(ytest_class, ytest_score)
# fig,ax = plt.subplots(figsize=(3,3), dpi=600)
fig, ax = plt.subplots()
ax.plot(fpr, tpr)
ax.set_title(f'ROC Curve (AUC = {np.round(auc_var, 3)})')
ax.set_xlabel('FPR')
ax.set_ylabel('TPR');
# plt.savefig('../figures/set2_roc_var.png', bbox_inches='tight')
```



[18]:
```
fig, ax = plt.subplots()
sns.heatmap(conf_var, annot=True, square=True, cmap='Blues', ax=ax);
```



[19]:
```
fig, ax = plt.subplots()
sns.heatmap(conf_var_train, annot=True, square=True, cmap='Blues', ax=ax);
```

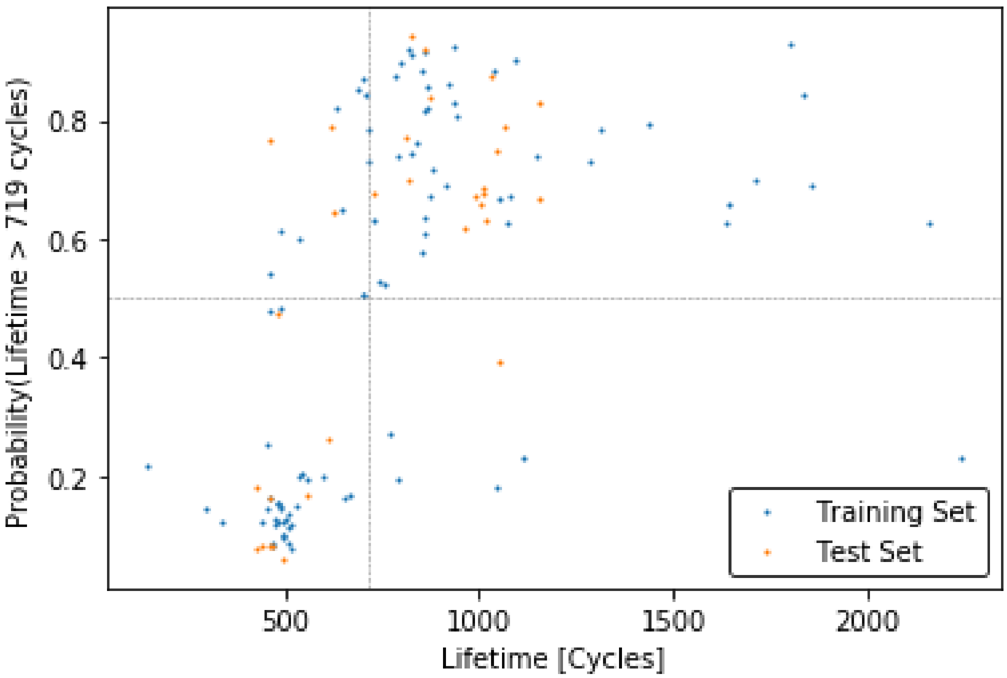


5.2 ΔQ(V) Features

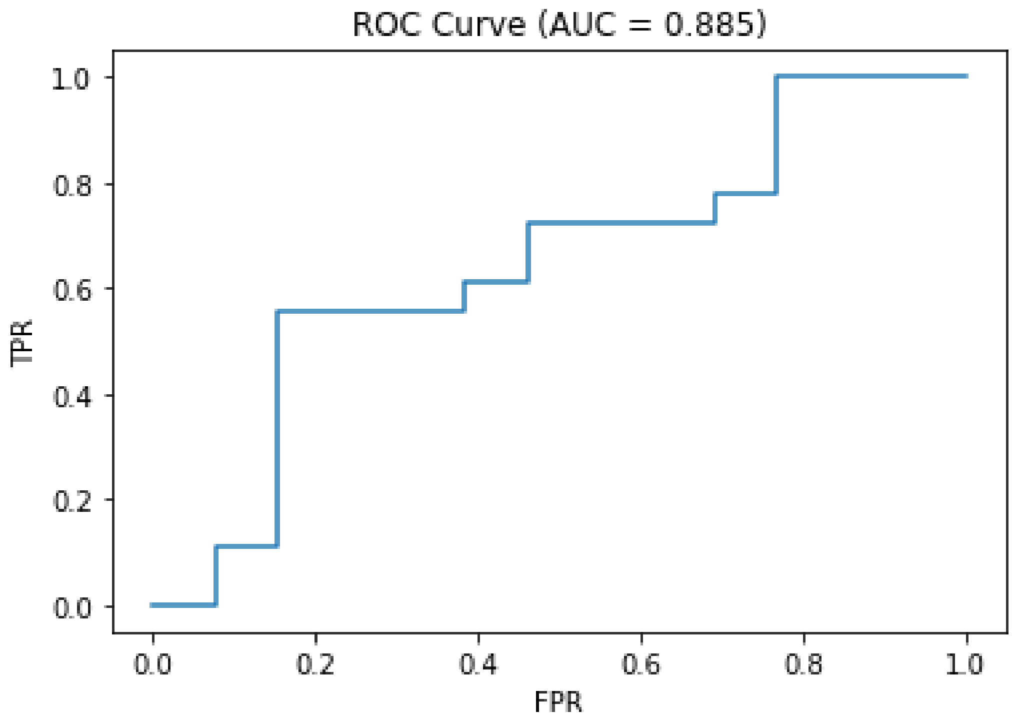
[20]:
```
xtrain_curr, xtest_curr = xtrain_delq, xtest_delq

threshold = get_percentiles(ytrain, 2)
ytrain_class = classify_lifetimes(ytrain, threshold)
ytest_class = classify_lifetimes(ytest, threshold)
print(f'Lifetime Cutoff: {threshold}\n')

model = BaggingClassifier(LogisticRegression(), n_estimators=25, oob_score=True)

model.fit(xtrain_curr, ytrain_class)
ytrain_pred = model.predict(xtrain_curr)
ytrain_score = model.predict_proba(xtrain_curr)[:,1]
ytest_pred = model.predict(xtest_curr)
ytest_score = model.predict_proba(xtest_curr)[:,1]

acc_delq_train = metrics.accuracy_score(ytrain_class, ytrain_pred)
acc_delq = metrics.accuracy_score(ytest_class, ytest_pred)
conf_delq_train = metrics.confusion_matrix(ytrain_class, ytrain_pred)
conf_delq = metrics.confusion_matrix(ytest_class, ytest_pred)
auc_delq_train = metrics.roc_auc_score(ytrain_class, ytrain_score)
auc_delq = metrics.roc_auc_score(ytest_class, ytest_score)
print(f'Training Accuracy: {acc_delq_train}')
print(f'Test Accuracy: {acc_delq}')
print(f'Training AUC: {auc_delq_train}')
print(f'Test AUC {auc_delq}')
```

```
Lifetime Cutoff: [719.]

Training Accuracy: 0.8172043010752689
Test Accuracy: 0.8709677419354839
Training AUC: 0.8802035152636448
Test AUC 0.8846153846153846
```

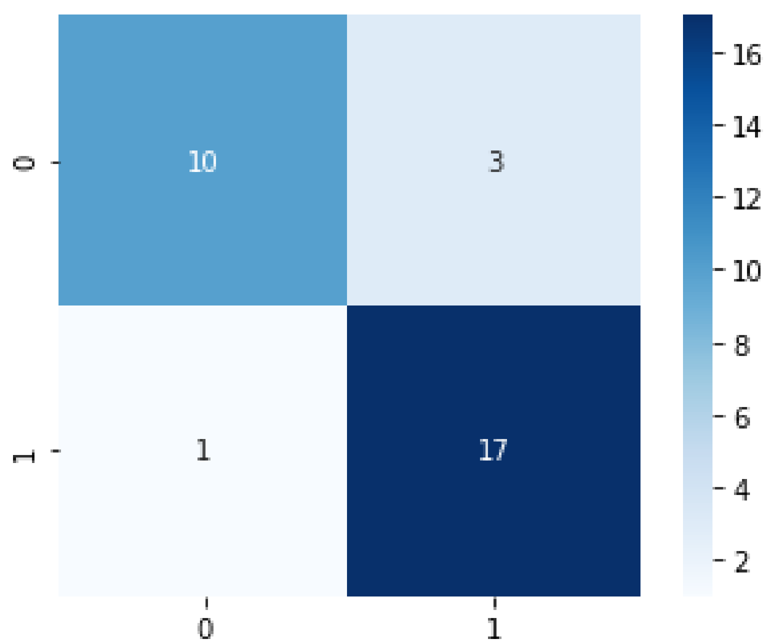
[21]:
```
# fig, ax = plt.subplots(figsize=(3,3), dpi=600)
fig,ax = plt.subplots()
ax.scatter(ytrain, ytrain_score, s=1, label='Training Set')
ax.scatter(ytest, ytest_score, s=1, label='Test Set')
ax.axhline(0.5, c='grey', ls='--', lw=0.5)
ax.axvline(int(threshold[0]), c='grey', ls='--', lw=0.5)
ax.set_xlabel('Lifetime [Cycles]')
ax.set_ylabel(f'Probability(Lifetime > {int(threshold[0])} cycles)')
ax.legend(edgecolor='k');

# plt.savefig('../figures/set2_decision_delq.png', bbox_inches='tight')
```



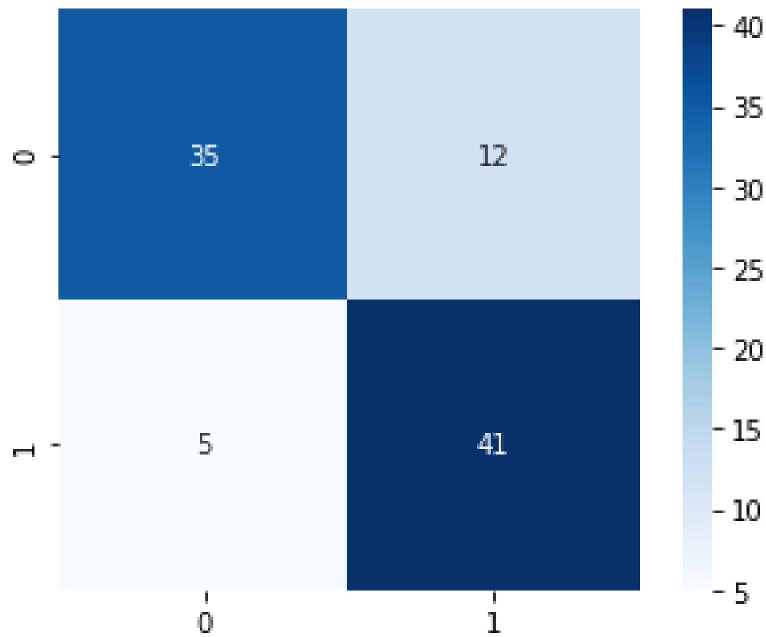
[22]:
```
# fpr, tpr, thresholds = metrics.roc_curve(ytest_class, ytest_score)
# fig,ax = plt.subplots(figsize=(3,3), dpi=600)
fig, ax = plt.subplots()
ax.plot(fpr, tpr)
ax.set_title(f'ROC Curve (AUC = {np.round(auc_delq, 3)})')
ax.set_xlabel('FPR')
ax.set_ylabel('TPR');
# plt.savefig('../figures/set2_roc_delq.png', bbox_inches='tight')
```



[23]:
```
fig, ax = plt.subplots()
sns.heatmap(conf_delq, annot=True, square=True, cmap='Blues', ax=ax);
```



[24]:
```
fig, ax = plt.subplots()
sns.heatmap(conf_delq_train, annot=True, square=True, cmap='Blues', ax=ax);
```

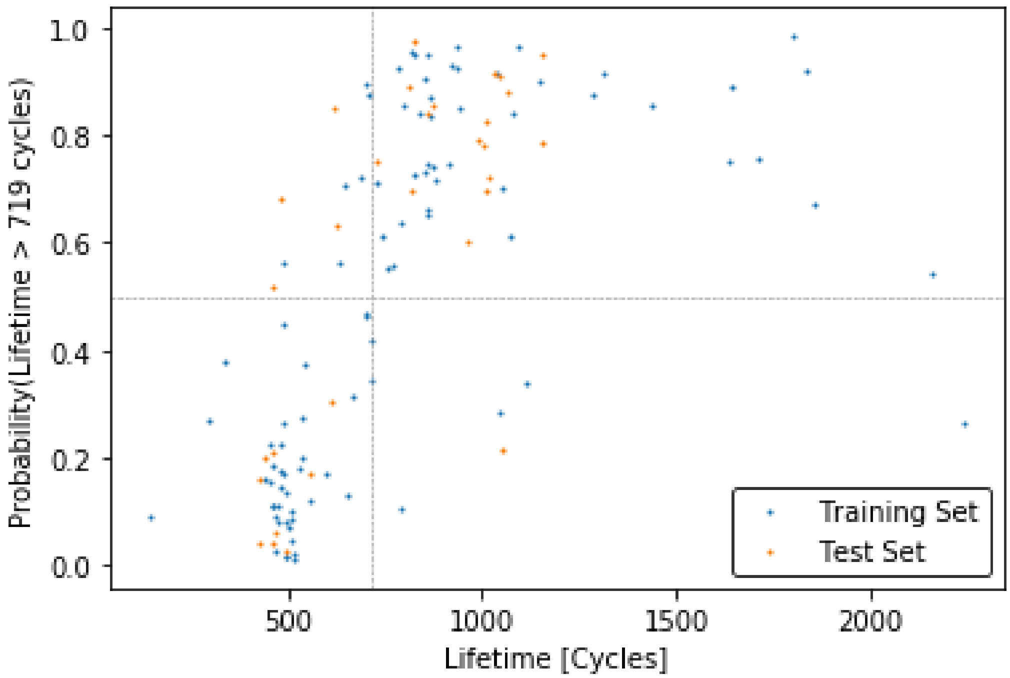


5.3 Discharge Curve Features

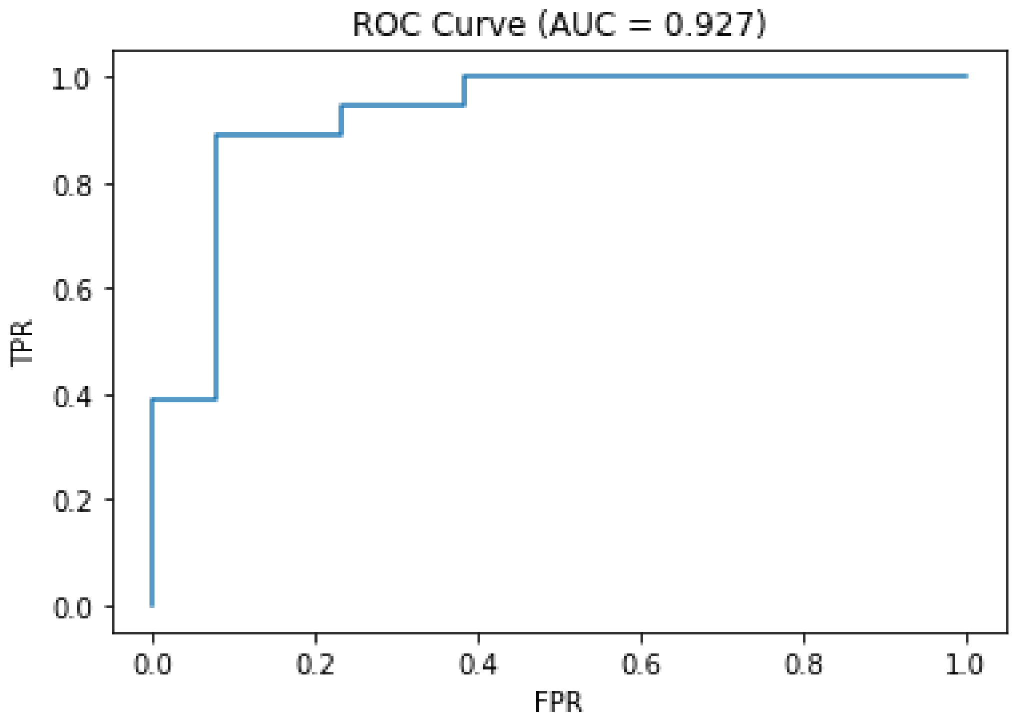
[25]:
```
xtrain_curr, xtest_curr = xtrain_fade, xtest_fade

threshold = get_percentiles(ytrain, 2)
ytrain_class = classify_lifetimes(ytrain, threshold)
ytest_class = classify_lifetimes(ytest, threshold)
print(f'Lifetime Cutoff: {threshold}\n')

model = BaggingClassifier(LogisticRegression(), n_estimators=25, oob_score=True)

model.fit(xtrain_curr, ytrain_class)
ytrain_pred = model.predict(xtrain_curr)
ytrain_score = model.predict_proba(xtrain_curr)[:,1]
ytest_pred = model.predict(xtest_curr)
ytest_score = model.predict_proba(xtest_curr)[:,1]

acc_fade_train = metrics.accuracy_score(ytrain_class, ytrain_pred)
acc_fade = metrics.accuracy_score(ytest_class, ytest_pred)
conf_fade_train = metrics.confusion_matrix(ytrain_class, ytrain_pred)
conf_fade = metrics.confusion_matrix(ytest_class, ytest_pred)
auc_fade_train = metrics.roc_auc_score(ytrain_class, ytrain_score)
auc_fade = metrics.roc_auc_score(ytest_class, ytest_score)
print(f'Training Accuracy: {acc_fade_train}')
print(f'Test Accuracy: {acc_fade}')
print(f'Training AUC: {auc_fade_train}')
print(f'Test AUC {auc_fade}')
```

```
Lifetime Cutoff: [719.]

Training Accuracy: 0.8924731182795699
Test Accuracy: 0.8387096774193549
Training AUC: 0.9255319148936171
Test AUC 0.9273504273504274
```

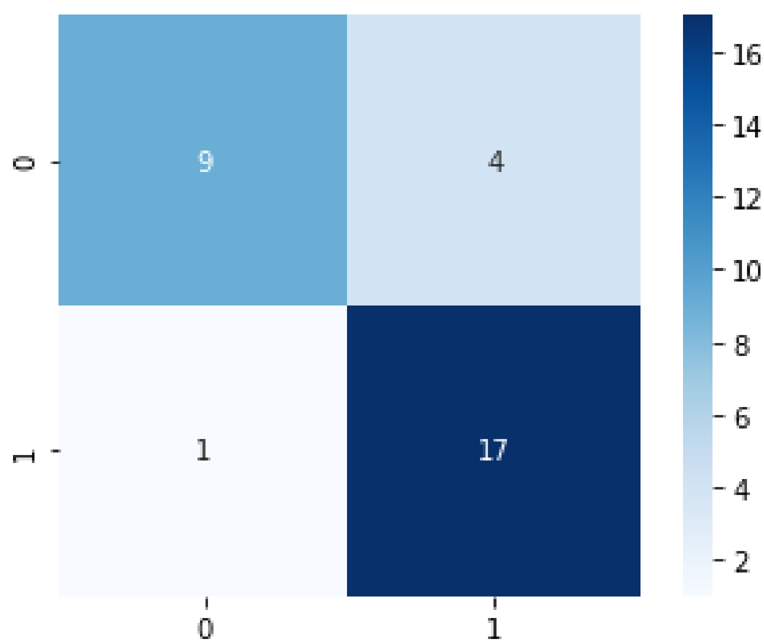
[26]:
```
# fig, ax = plt.subplots(figsize=(3,3), dpi=600)
fig,ax = plt.subplots()
ax.scatter(ytrain, ytrain_score, s=1, label='Training Set')
ax.scatter(ytest, ytest_score, s=1, label='Test Set')
ax.axhline(0.5, c='grey', ls='--', lw=0.5)
ax.axvline(int(threshold[0]), c='grey', ls='--', lw=0.5)
ax.set_xlabel('Lifetime [Cycles]')
ax.set_ylabel(f'Probability(Lifetime > {int(threshold[0])} cycles)')
ax.legend(edgecolor='k');

# plt.savefig('../figures/set2_decision_fade.png', bbox_inches='tight')
```



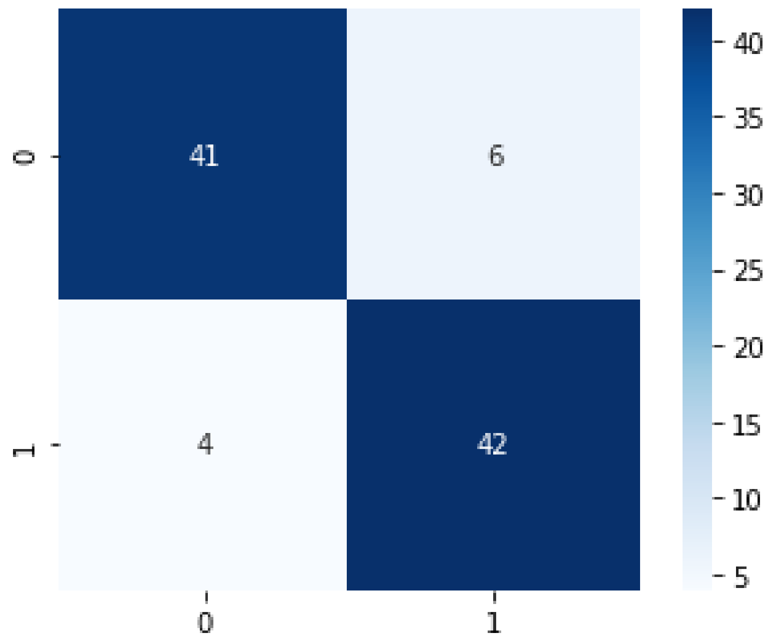
[27]:
```
fpr, tpr, thresholds = metrics.roc_curve(ytest_class, ytest_score)
# fig,ax = plt.subplots(figsize=(3,3), dpi=600)
fig, ax = plt.subplots()
ax.plot(fpr, tpr)
ax.set_title(f'ROC Curve (AUC = {np.round(auc_fade, 3)})')
ax.set_xlabel('FPR')
ax.set_ylabel('TPR');
# plt.savefig('../figures/set2_roc_fade.png', bbox_inches='tight')
```



[28]:
```
fig, ax = plt.subplots()
sns.heatmap(conf_fade, annot=True, square=True, cmap='Blues', ax=ax);
```



[29]:
```
fig, ax = plt.subplots()
sns.heatmap(conf_fade_train, annot=True, square=True, cmap='Blues', ax=ax);
```

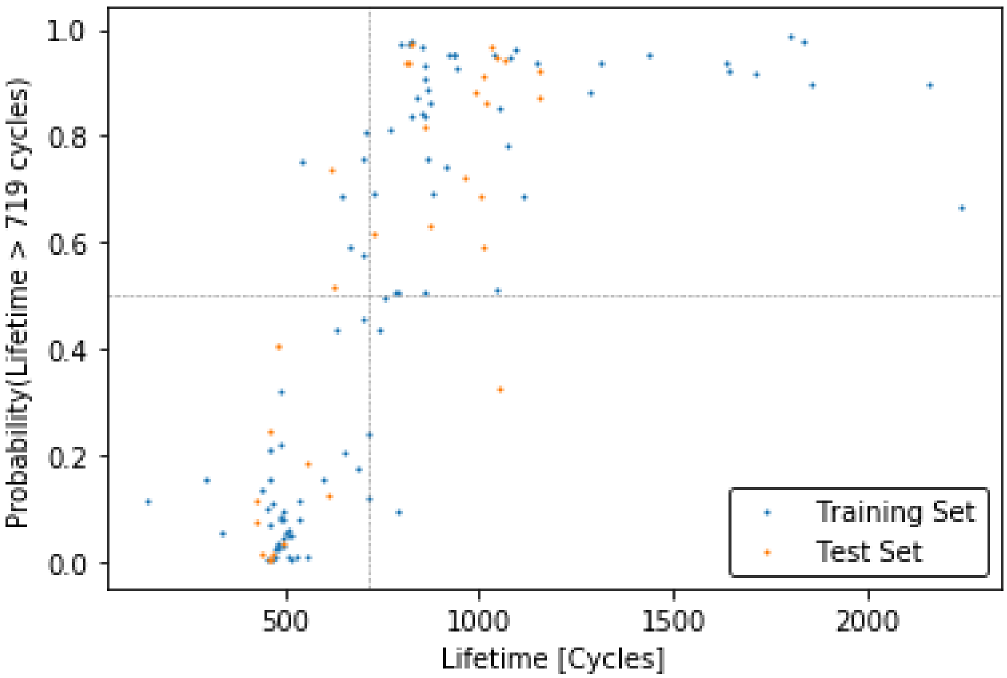


5.4 All Features

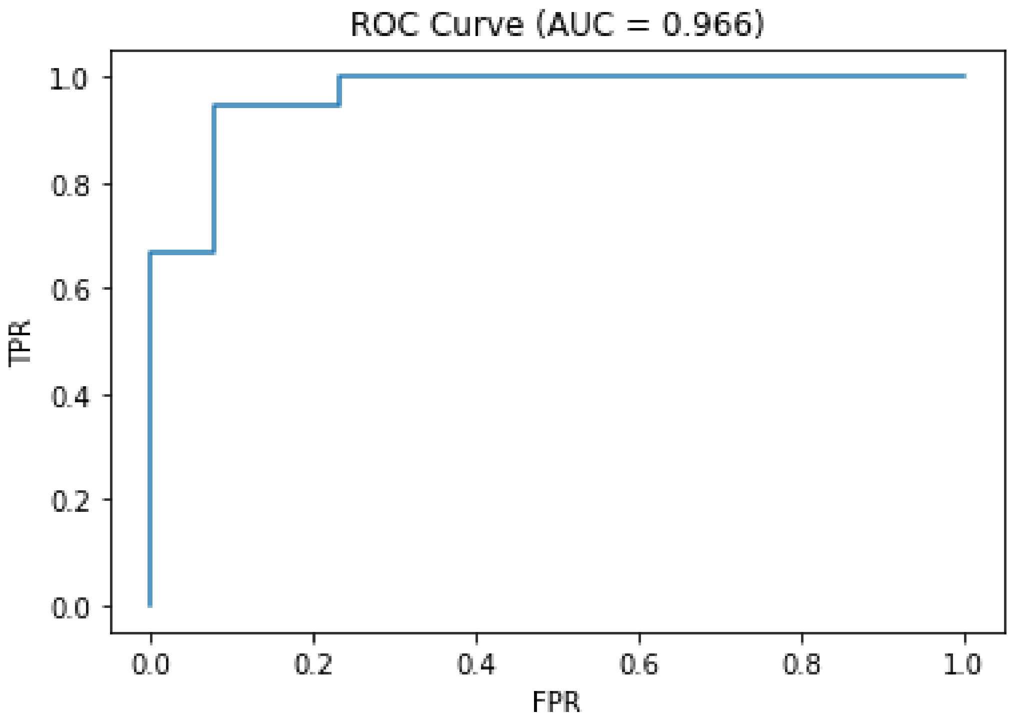
[30]:
```
xtrain_curr, xtest_curr = xtrain, xtest

threshold = get_percentiles(ytrain, 2)
ytrain_class = classify_lifetimes(ytrain, threshold)
ytest_class = classify_lifetimes(ytest, threshold)
print(f'Lifetime Cutoff: {threshold}\n')

model = BaggingClassifier(LogisticRegression(), n_estimators=25, oob_score=True)

model.fit(xtrain_curr, ytrain_class)
ytrain_pred = model.predict(xtrain_curr)
ytrain_score = model.predict_proba(xtrain_curr)[:,1]
ytest_pred = model.predict(xtest_curr)
ytest_score = model.predict_proba(xtest_curr)[:,1]

acc_full_train = metrics.accuracy_score(ytrain_class, ytrain_pred)
acc_full = metrics.accuracy_score(ytest_class, ytest_pred)
conf_full_train = metrics.confusion_matrix(ytrain_class, ytrain_pred)
conf_full = metrics.confusion_matrix(ytest_class, ytest_pred)
auc_full_train = metrics.roc_auc_score(ytrain_class, ytrain_score)
auc_full = metrics.roc_auc_score(ytest_class, ytest_score)
print(f'Training Accuracy: {acc_full_train}')
print(f'Test Accuracy: {acc_full}')
print(f'Training AUC: {auc_full_train}')
print(f'Test AUC {auc_full}')
```

```
Lifetime Cutoff: [719.]

Training Accuracy: 0.9032258064516129
Test Accuracy: 0.9032258064516129
Training AUC: 0.9629972247918595
Test AUC 0.9658119658119657
```

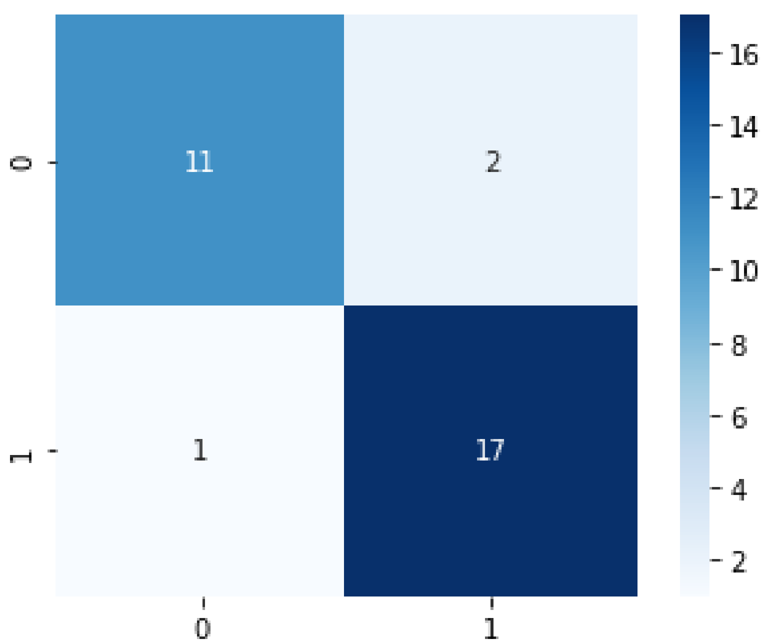
[31]:
```
# fig, ax = plt.subplots(figsize=(3,3), dpi=600)
fig,ax = plt.subplots()
ax.scatter(ytrain, ytrain_score, s=1, label='Training Set')
ax.scatter(ytest, ytest_score, s=1, label='Test Set')
ax.axhline(0.5, c='grey', ls='--', lw=0.5)
ax.axvline(int(threshold[0]), c='grey', ls='--', lw=0.5)
ax.set_xlabel('Lifetime [Cycles]')
ax.set_ylabel(f'Probability(Lifetime > {int(threshold[0])} cycles)')
ax.legend(edgecolor='k');

# plt.savefig('../figures/set2_decision_full.png', bbox_inches='tight')
```



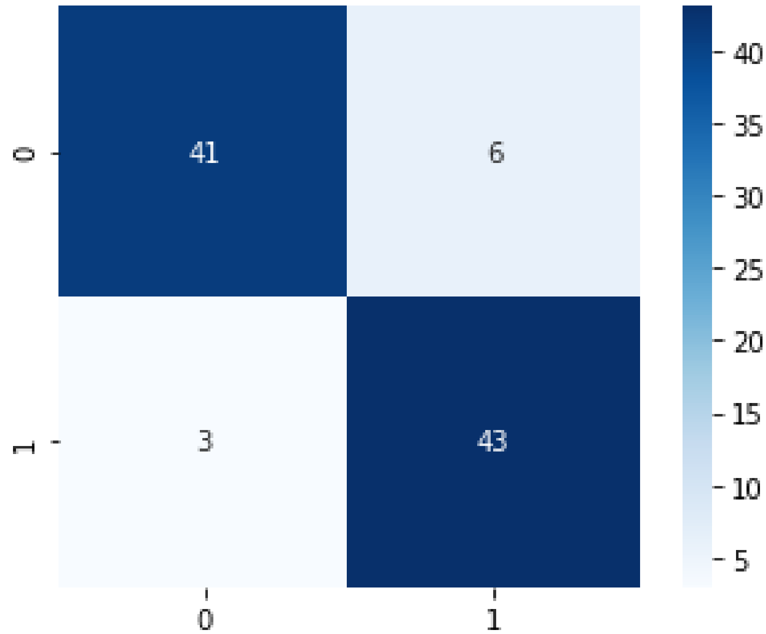
[32]:
```
fpr, tpr, thresholds = metrics.roc_curve(ytest_class, ytest_score)
# fig,ax = plt.subplots(figsize=(3,3), dpi=600)
fig, ax = plt.subplots()
ax.plot(fpr, tpr)
ax.set_title(f'ROC Curve (AUC = {np.round(auc_full, 3)})')
ax.set_xlabel('FPR')
ax.set_ylabel('TPR');
# plt.savefig('../figures/set2_roc_full.png', bbox_inches='tight')
```



[33]:
```
fig, ax = plt.subplots()
sns.heatmap(conf_full, annot=True, square=True, cmap='Blues', ax=ax);
```

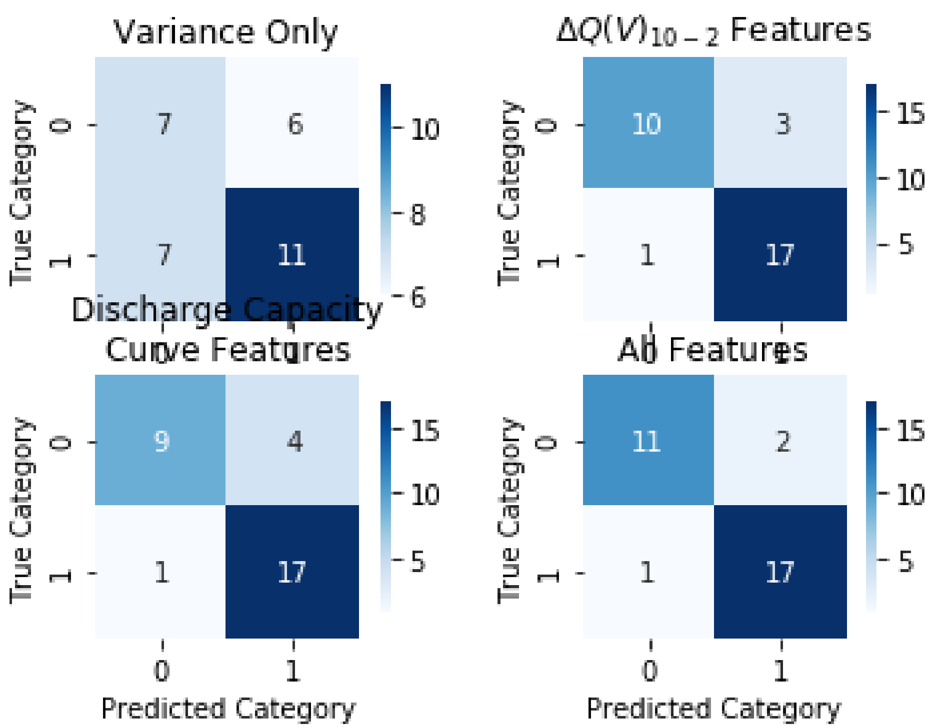


[34]:
```
fig, ax = plt.subplots()
sns.heatmap(conf_full_train, annot=True, square=True, cmap='Blues', ax=ax);
```



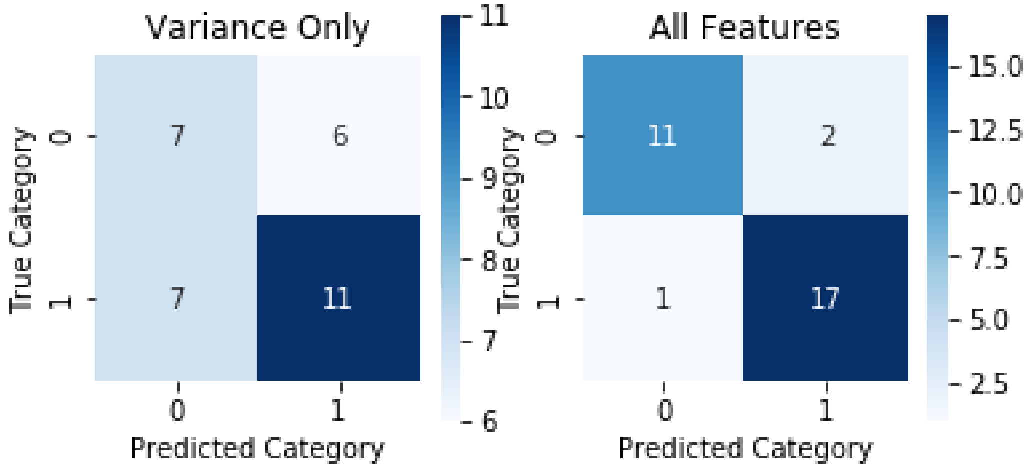
6 Figures

[35]:
```
# fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(5,5), dpi=600,␣
↪gridspec_kw={'wspace':0.4, 'hspace':0.4})
fig, ax = plt.subplots(nrows=2, ncols=2)
confusions = [conf_var, conf_delq , conf_fade, conf_full]
titles = ['Variance Only', r'$\Delta Q(V)_{10-2}$ Features', 'Discharge␣
↪Capacity\nCurve Features', 'All Features']
for a, t, c in zip(ax.flatten(), titles, confusions):
    sns.heatmap(c, annot=True, square=True, cmap='Blues', cbar_kws={'shrink':0.
↪8}, ax=a)
    a.set_title(t)
    a.set_xlabel('Predicted Category')
    a.set_ylabel('True Category')

# plt.savefig('../figures/set2_confusion_logistic.png')
```



[36]:
```
# fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(6,3), dpi=600,␣
↪gridspec_kw={'wspace':0.4, 'hspace':0.4})
fig, ax = plt.subplots(nrows=1, ncols=2)
confusions = [conf_var, conf_full]
titles = ['Variance Only', 'All Features']
```

```
for a, t, c in zip(ax.flatten(), titles, confusions):
    sns.heatmap(c, annot=True, square=True, cmap='Blues', cbar_kws={'shrink':0.
↪7}, ax=a)
    a.set_title(t)
    a.set_xlabel('Predicted Category')
    a.set_ylabel('True Category')

# plt.savefig('../figures/set2_confusion2_logistic.png')
```



One vs Rest Classifier

May 10, 2020

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy import stats
import seaborn as sns
from sklearn.linear_model import LogisticRegression
from sklearn.linear_model import LogisticRegressionCV
from sklearn.multiclass import OneVsRestClassifier
from sklearn import metrics
from sklearn import model_selection
from sklearn import preprocessing
from sklearn.ensemble import BaggingClassifier
```

1 Load Data

[2]:
```
# load feature matrix
feature_df = pd.read_csv('../data/featurematrix_classification2.csv')
feature_df.drop(columns='Unnamed: 0', inplace=True)
# feature_df = feature_df.drop([5,16]).reset_index(drop=True)
feature_df.head()
```

[2]:
```
       b1c0      b1c1      b1c2      b1c3      b1c4      b1c5      b1c6  \
0 -4.181804 -3.969042 -2.788997 -4.700016 -4.132990 -3.905395 -4.466846
1 -2.584744 -2.540101 -2.697642 -2.572648 -2.527145 -2.452704 -2.438709
2 -5.514008 -5.519434 -5.647117 -5.553178 -5.307388 -5.250376 -5.173391
3  0.207469 -0.149255 -0.407511 -0.504164  0.055138 -0.295200  0.186631
4  0.770794  0.457356  0.370178 -0.298181  0.157237 -0.434155  0.665555

       b1c7      b1c9     b1c11  …     b3c33     b3c34     b3c35     b3c36  \
0 -4.970406 -3.884769 -3.948446  … -4.045138 -3.987800 -5.067957 -4.643561
1 -2.730548 -2.442842 -2.425090  … -2.531871 -2.639872 -2.657420 -2.534023
2 -5.874992 -5.128251 -4.781530  … -4.983046 -5.217281 -5.163107 -4.986688
3 -0.163842  0.077411  0.405436  …  0.410522  0.419143  0.452853  0.445606
4  0.428794  0.325888  0.850656  …  0.837771  0.884494  0.929203  0.930771

      b3c40     b3c41     b3c42     b3c43     b3c44     b3c45
```

```
0 -4.675587 -4.547703 -3.900646 -2.415616 -4.571223 -5.265527
1 -2.347406 -2.421160 -2.535104 -4.090369 -2.594459 -2.581789
2 -4.734173 -4.722736 -5.167258 -5.719654 -5.365331 -5.094440
3  0.341317  0.363618  0.266625 -0.138596  0.263355  0.413564
4  0.733024  0.724385  0.640506 -1.321118  0.664487  0.861365

[5 rows x 124 columns]
```

[3]:
```python
# load lifetime matrix
lifetime_df = pd.read_csv('../data/lifetimematrix.csv')
lifetime_df.drop(columns='Unnamed: 0', inplace=True)
lifetime_df
```

[3]:
```
     b1c0    b1c1    b1c2    b1c3    b1c4    b1c5   b1c6   b1c7    b1c9  \
0  1852.0  2160.0  2237.0  1434.0  1709.0  1074.0  636.0  870.0  1054.0

   b1c11  …   b3c33   b3c34   b3c35  b3c36  b3c40  b3c41   b3c42   b3c43  \
0  788.0  …  1284.0  1158.0  1093.0  923.0  796.0  786.0  1642.0  1046.0

   b3c44   b3c45
0  940.0  1801.0

[1 rows x 124 columns]
```

[4]:
```python
# convert data to numpy arrays/matrices
features = np.array(feature_df).T
lifetimes = np.array(lifetime_df).flatten()
```

[5]:
```python
features.shape, lifetimes.shape
```

[5]: ((124, 16), (124,))

[6]:
```python
# standardize data
features = preprocessing.scale(features, axis=0)
```

2 Test/Train Split

[7]:
```
# 75/25 train/test split
xtrain, xtest, ytrain, ytest = model_selection.train_test_split(features,␣
 ↪lifetimes, test_size=0.25)
```

[8]:
```
# train_idx = np.arange(84)
# test_idx = np.arange(84,124)
```

[9]:
```
# xtrain, ytrain = features[train_idx,:], lifetimes[train_idx]
# xtest, ytest = features[test_idx,:], lifetimes[test_idx]
```

[10]:
```
xtrain.shape, ytrain.shape, xtest.shape, ytest.shape
```

[10]:
```
((93, 16), (93,), (31, 16), (31,))
```

3 Feature Sets

[11]:
```
xtrain_var = xtrain[:,2].reshape(-1,1)
xtest_var = xtest[:,2].reshape(-1,1)
xtrain_var.shape, xtest_var.shape
```

[11]: ((93, 1), (31, 1))

[12]:
```
xtrain_delq, xtest_delq = xtrain[:,:5], xtest[:,:5]
xtrain_delq.shape, xtest_delq.shape
```

[12]: ((93, 5), (31, 5))

[13]:
```
xtrain_fade, xtest_fade = xtrain[:,:10], xtest[:,:10]
xtrain_fade.shape, xtest_fade.shape
```

[13]: ((93, 10), (31, 10))

4 Lifetime Classification Encoder

[14]:
```
# Classifies batteries into lifetime categories based on the specified number␣
↪thresholds
def classify_lifetimes(yvals, thresholds):
    thresh = np.array(thresholds).flatten()
    thresh = np.append(thresh, max(yvals))
    classes = np.array([])
    for y in yvals:
        for i in range(len(thresh)):
            if y <= thresh[i]:
                classes = np.append(classes, i)
                break
    return classes

# Calculates the cutoffs for a given number of quantiles
def get_percentiles(yvals, n):
    percentiles = np.arange(0, 1, 1/n)*100
    percentiles = percentiles[1:]
    return np.percentile(yvals, percentiles)
```

5 One vs. Rest Logistic Regression

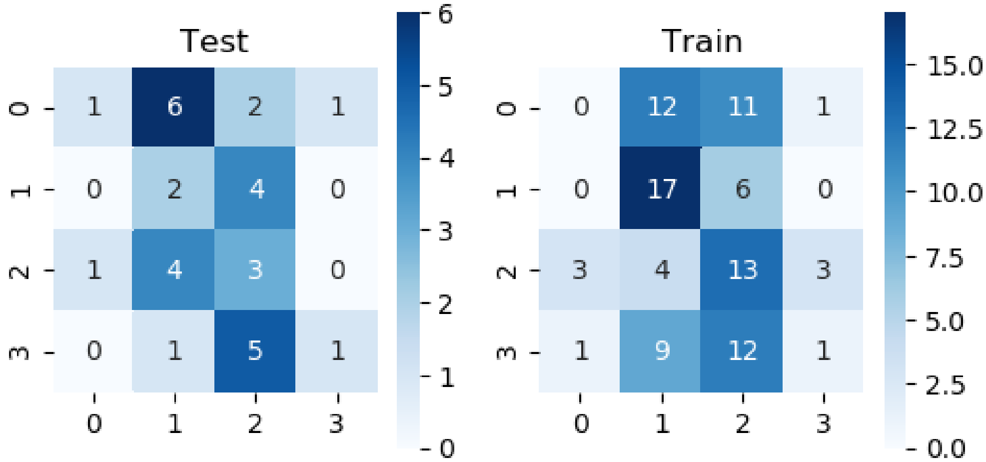
5.1 Variance only

[15]:
```
num_cat=4
```

[16]:
```
xtrain_curr, xtest_curr = xtrain_var, xtest_var

threshold = get_percentiles(ytrain, num_cat)
ytrain_class = classify_lifetimes(ytrain, threshold)
ytest_class = classify_lifetimes(ytest, threshold)
print(f'Lifetime Cutoffs: {threshold}\n')

model =␣
↪BaggingClassifier(OneVsRestClassifier(LogisticRegression(max_iter=1000)),␣
↪n_estimators=25, oob_score=True)

model.fit(xtrain_curr, ytrain_class)
ytrain_pred = model.predict(xtrain_curr)
ytrain_score = model.predict_proba(xtrain_curr)
ytest_pred = model.predict(xtest_curr)
ytest_score = model.predict_proba(xtest_curr)

acc_var_train = metrics.accuracy_score(ytrain_class, ytrain_pred)
acc_var = metrics.accuracy_score(ytest_class, ytest_pred)
conf_var_train = metrics.confusion_matrix(ytrain_class, ytrain_pred)
conf_var = metrics.confusion_matrix(ytest_class, ytest_pred)
# auc_var_ovr_train = metrics.roc_auc_score(ytrain_class, ytrain_score)
# auc_var_ovr = metrics.roc_auc_score(ytest_class, ytest_score)
print(f'Training Accuracy: {acc_var_train}')
print(f'Test Accuracy: {acc_var}')
# # print(f'Training AUC: {auc_full_train}')
# print(f'Test AUC {auc_full}')
```

```
Lifetime Cutoffs: [502. 742. 966.]

Training Accuracy: 0.3333333333333333
Test Accuracy: 0.22580645161290322
```

[17]:
```
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(6,3), dpi=100)
sns.heatmap(conf_var, annot=True, square=True, cmap='Blues', ax=ax[0]);
sns.heatmap(conf_var_train, annot=True, square=True, cmap='Blues', ax=ax[1]);
ax[0].set_title('Test')
ax[1].set_title('Train');
```

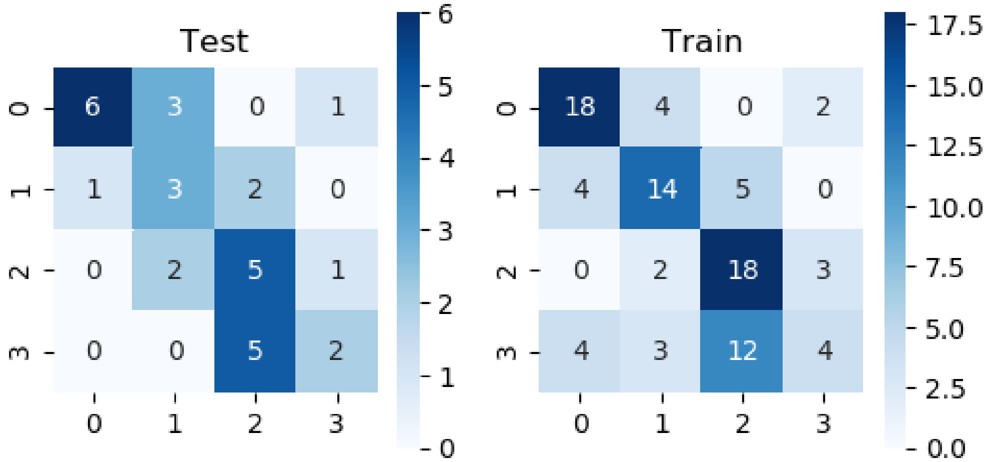

Test
Train

5.2 ΔQ(V) Features

[18]:
```python
xtrain_curr, xtest_curr = xtrain_delq, xtest_delq

threshold = get_percentiles(ytrain, num_cat)
ytrain_class = classify_lifetimes(ytrain, threshold)
ytest_class = classify_lifetimes(ytest, threshold)
print(f'Lifetime Cutoffs: {threshold}\n')

model =␣
↪BaggingClassifier(OneVsRestClassifier(LogisticRegression(max_iter=1000)),␣
↪n_estimators=25, oob_score=True)

model.fit(xtrain_curr, ytrain_class)
ytrain_pred = model.predict(xtrain_curr)
ytrain_score = model.predict_proba(xtrain_curr)
ytest_pred = model.predict(xtest_curr)
ytest_score = model.predict_proba(xtest_curr)

acc_delq_train = metrics.accuracy_score(ytrain_class, ytrain_pred)
acc_delq = metrics.accuracy_score(ytest_class, ytest_pred)
conf_delq_train = metrics.confusion_matrix(ytrain_class, ytrain_pred)
conf_delq = metrics.confusion_matrix(ytest_class, ytest_pred)
# auc_var_ovr_train = metrics.roc_auc_score(ytrain_class, ytrain_score)
# auc_var_ovr = metrics.roc_auc_score(ytest_class, ytest_score)
print(f'Training Accuracy: {acc_delq_train}')
print(f'Test Accuracy: {acc_delq}')
# # print(f'Training AUC: {auc_full_train}')
# print(f'Test AUC {auc_full}')
```

```
Lifetime Cutoffs: [502. 742. 966.]

Training Accuracy: 0.5806451612903226
Test Accuracy: 0.5161290322580645
```

[19]:
```python
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(6,3), dpi=100)
sns.heatmap(conf_delq, annot=True, square=True, cmap='Blues', ax=ax[0]);
sns.heatmap(conf_delq_train, annot=True, square=True, cmap='Blues', ax=ax[1]);
ax[0].set_title('Test')
ax[1].set_title('Train');
```

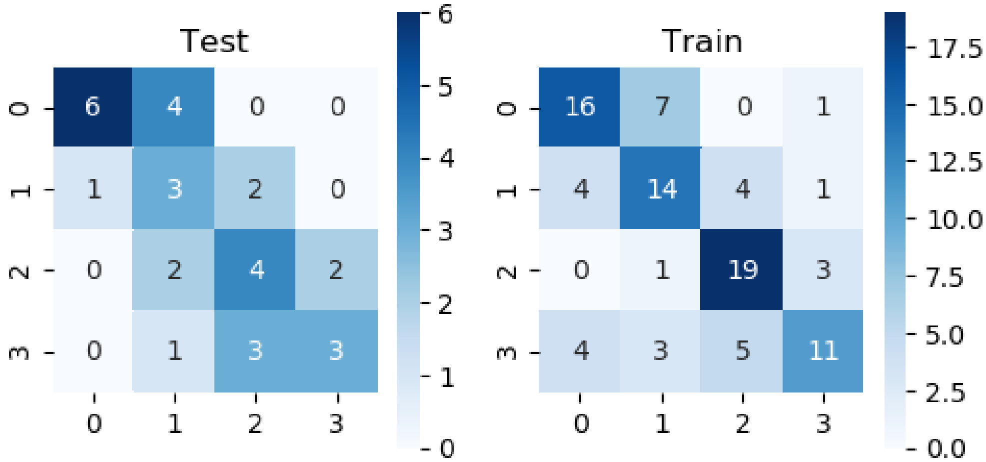

Test
Train

5.3 Discharge Curve Features

[20]:
```
xtrain_curr, xtest_curr = xtrain_fade, xtest_fade

threshold = get_percentiles(ytrain, num_cat)
ytrain_class = classify_lifetimes(ytrain, threshold)
ytest_class = classify_lifetimes(ytest, threshold)
print(f'Lifetime Cutoffs: {threshold}\n')

model =␣
↪BaggingClassifier(OneVsRestClassifier(LogisticRegression(max_iter=1000)),␣
↪n_estimators=25, oob_score=True)

model.fit(xtrain_curr, ytrain_class)
ytrain_pred = model.predict(xtrain_curr)
ytrain_score = model.predict_proba(xtrain_curr)
ytest_pred = model.predict(xtest_curr)
ytest_score = model.predict_proba(xtest_curr)

acc_fade_train = metrics.accuracy_score(ytrain_class, ytrain_pred)
acc_fade = metrics.accuracy_score(ytest_class, ytest_pred)
conf_fade_train = metrics.confusion_matrix(ytrain_class, ytrain_pred)
conf_fade = metrics.confusion_matrix(ytest_class, ytest_pred)
# auc_var_ovr_train = metrics.roc_auc_score(ytrain_class, ytrain_score)
# auc_var_ovr = metrics.roc_auc_score(ytest_class, ytest_score)
print(f'Training Accuracy: {acc_fade_train}')
print(f'Test Accuracy: {acc_fade}')
# # print(f'Training AUC: {auc_full_train}')
# print(f'Test AUC {auc_full}')
```

```
Lifetime Cutoffs: [502. 742. 966.]

Training Accuracy: 0.6451612903225806
Test Accuracy: 0.5161290322580645
```

[21]:
```
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(6,3), dpi=100)
sns.heatmap(conf_fade, annot=True, square=True, cmap='Blues', ax=ax[0]);
sns.heatmap(conf_fade_train, annot=True, square=True, cmap='Blues', ax=ax[1]);
ax[0].set_title('Test')
ax[1].set_title('Train');
```


Test
Train
6
4
0
0
1
3
2
0
0
2
4
2
0
1
3
3
16
7
0
1
4
14
4
1
0
1
19
3
4
3
5
11
0
1
2
3
4
5
6
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5

5.4 All Features

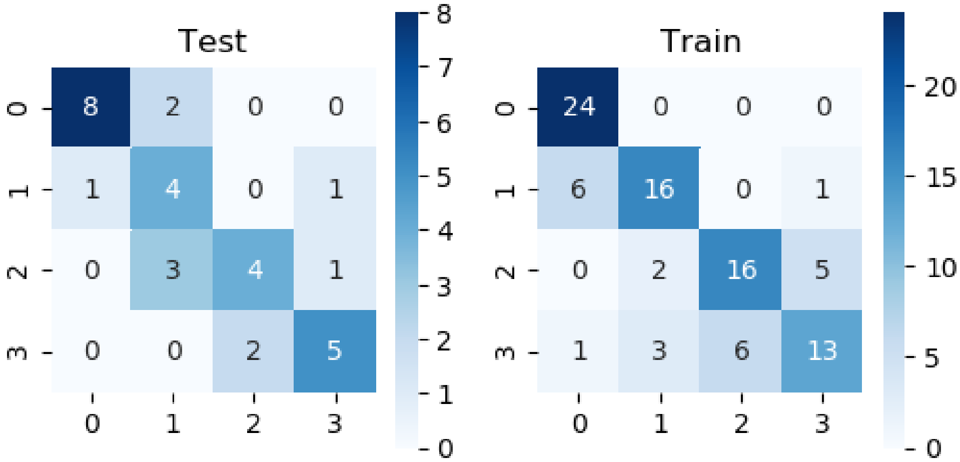
[22]:
```
xtest.shape
```

[22]: (31, 16)

[23]:
```
xtrain_curr, xtest_curr = xtrain, xtest

threshold = get_percentiles(ytrain, num_cat)
ytrain_class = classify_lifetimes(ytrain, threshold)
ytest_class = classify_lifetimes(ytest, threshold)
print(f'Lifetime Cutoffs: {threshold}\n')

model =␣
 ↪BaggingClassifier(OneVsRestClassifier(LogisticRegression(max_iter=1000)),␣
 ↪n_estimators=25, oob_score=True)

model.fit(xtrain_curr, ytrain_class)
ytrain_pred = model.predict(xtrain_curr)
ytrain_score = model.predict_proba(xtrain_curr)
ytest_pred = model.predict(xtest_curr)
ytest_score = model.predict_proba(xtest_curr)

acc_full_train = metrics.accuracy_score(ytrain_class, ytrain_pred)
acc_full = metrics.accuracy_score(ytest_class, ytest_pred)
conf_full_train = metrics.confusion_matrix(ytrain_class, ytrain_pred)
conf_full = metrics.confusion_matrix(ytest_class, ytest_pred)
# auc_var_ovr_train = metrics.roc_auc_score(ytrain_class, ytrain_score)
# auc_var_ovr = metrics.roc_auc_score(ytest_class, ytest_score)
print(f'Training Accuracy: {acc_full_train}')
print(f'Test Accuracy: {acc_full}')
# # print(f'Training AUC: {auc_full_train}')
# print(f'Test AUC {auc_full}')
```

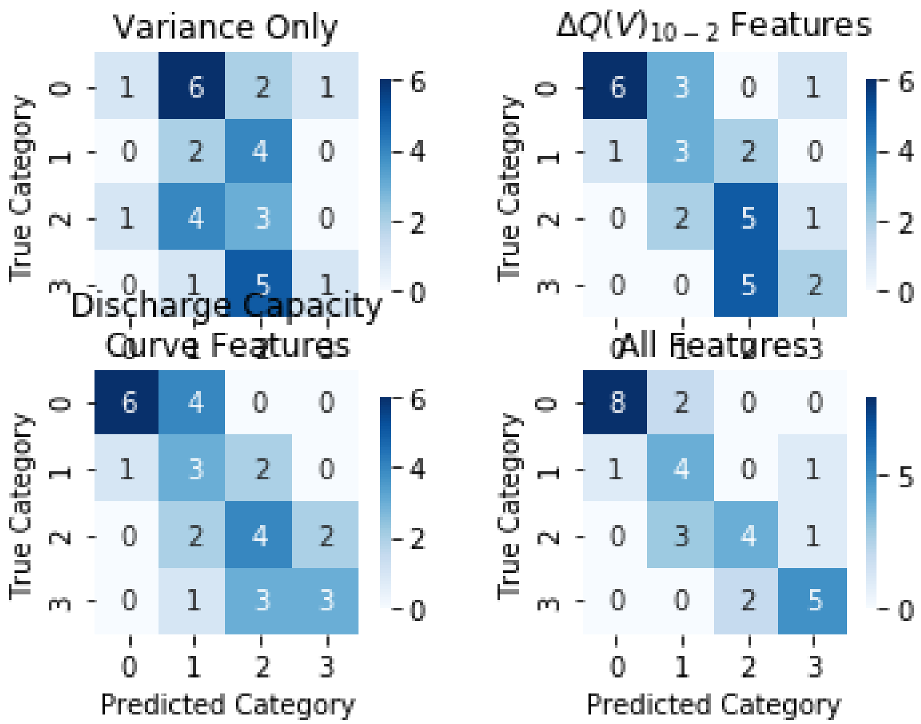
```
Lifetime Cutoffs: [502. 742. 966.]

Training Accuracy: 0.7419354838709677
Test Accuracy: 0.6774193548387096
```

[24]:
```
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(6,3), dpi=100)
sns.heatmap(conf_full, annot=True, square=True, cmap='Blues', ax=ax[0]);
sns.heatmap(conf_full_train, annot=True, square=True, cmap='Blues', ax=ax[1]);
ax[0].set_title('Test')
ax[1].set_title('Train');
```


Test
Train
0
1
2
3
8
2
0
0
1
4
0
1
0
3
4
1
0
0
2
5
24
0
0
0
6
16
0
1
0
2
16
5
1
3
6
13
20
15
10
5

6 Figures

[25]:
```
# fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(5,5), dpi=600,␣
↪gridspec_kw={'wspace':0.4, 'hspace':0.4})
fig, ax = plt.subplots(nrows=2, ncols=2)
confusions = [conf_var, conf_delq, conf_fade, conf_full]
titles = ['Variance Only', r'$\Delta Q(V)_{10-2}$ Features', 'Discharge␣
↪Capacity\nCurve Features', 'All Features']
for a, t, c in zip(ax.flatten(), titles, confusions):
    sns.heatmap(c, annot=True, square=True, cmap='Blues', cbar_kws={'shrink':0.
↪8}, ax=a)
    a.set_title(t)
    a.set_xlabel('Predicted Category')
    a.set_ylabel('True Category')

# plt.savefig('../figures/set2_confusion_ovr4.png')
```



[26]:
```
# fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(6,3), dpi=600,␣
↪gridspec_kw={'wspace':0.4, 'hspace':0.4})
fig, ax = plt.subplots(nrows=1, ncols=2)
confusions = [conf_var, conf_full]
titles = ['Variance Only', 'All Features']
```

```
for a, t, c in zip(ax.flatten(), titles, confusions):
    sns.heatmap(c, annot=True, square=True, cmap='Blues', cbar_kws={'shrink':0.
↪7}, ax=a)
    a.set_title(t)
    a.set_xlabel('Predicted Category')
    a.set_ylabel('True Category')

# plt.savefig('../figures/set2_confusion2_ovr.png')
```