Predicting Appliance Energy Consumption

Zachary Buckley

George Washington University
DATS 6450 - Multivariate Modeling
Dr. Reza Jafari
Term Project

April 22, 2020

Contents

Abstract

This report looks at the application of several Time Series Forecasting Methods, and Multiple Linear Regression modeling techniques, to a dataset developed by members of the Thermal Engineering and Combustion Laboratory at the University of Mons in Belgium. We'll use the resulting models to predict and forecast Appliance Energy Consumption. The models will be developed by applying the Multivariate Modeling Process. The time-series forecasting approaches being considered include the Holt-Winter Method, and ARMA Models. The ARMA Model parameters are estimated using a custom Levenberg-Marquardt Algorithm developed for the purpose.

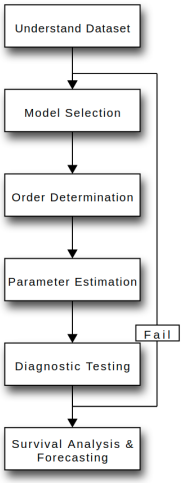


Figure 1: Multivariate Modeling Process

1 Introduction

This report looks at the application of multiple Time Series Models, and Linear Regression Models, to a dataset developed by members of the Thermal Engineering and Combustion Laboratory at the University of Mons in Belgium (Candanedo et al. (2017)). In an attempt to forecast Appliance energy consumption, both with and without additional explanatory variables, we'll develop a custom collection of modeling tools, based heavily on software built for various assignments throughout this course.

1.1 Methods

This section attempts to explain the variety of techniques used to develop, and construct the models. For all of these models, we'll be applying the Multivariate Modeling Process described in class (Figure 1). Our goal is to produce a variety of Multivariate Models for predicting (Regression), and forecasting (Time Series) the dependent variable (Appliance energy consumption) and evaluating the results against a test set.

1.1.1 Understand Dataset

Understanding the dataset involves what typically is called Exploratory Data Analysis (EDA) in other contexts. Using various tools and techniques at our disposal we explore the dataset, learning more about how the features relate to each other, and the dependent variable. For this dataset, we'll likely be looking at a plot of dependent variable vs time to get an initial visual understanding, an

Autocorrelation Function Plot to better indicate how related the next value is to the previous value, and a correlation matrix to better understand correlations between the variables. As we're specifically targeting time series models in this project we'll also cover the issue of stationarity, and time series decomposition during this process. The information gleaned from all this data exploration helps to inform our decisions during the Model Selection stage of the process.

Stationarity "A Stationary time series is one whose properties do not depend on the time at which the series is observed" (Hyndman et al., 2014, p. 223). The book later goes on to clarify that really we mean that the distribution of the time series doesn't depend on time, but it does seem a reasonable place to start. Our goal in determining stationarity is to try and determine if we can apply time-series methods and models that assume stationarity with or without applying a transform to the data first. Our approach for making this determination is typically visual inspection of the Signal plotted versus time, a histogram of the signal, the Autocorrelation Function, and finally the Augmented Dickey-Fuller test to determine if a unit root exists for the time series. If a unit root doesn't exist, we can usually assume that the time series data is stationary, but there are exceptions. Should the time series data be non-stationary, it could be necessary to apply a transform to the data, before some forecasting and prediction methods can be applied.

Difference Transform The difference transform, is quite good at removing linear trend from a dataset, effectively it takes the derivative of the time series with respect to time. One potential pit-fall, is that in reversing the transform, any error in the model has more and more effect as you forecast farther ahead, this effect is even more pronounced when using a 2nd order difference transform. Equation 1 shows the 1st order difference transform. When the first difference transform itself is stationary and random, the timeseries is sometimes referred to as a 'random walk'.

$$z_t = y_t - y_{t-1} \tag{1}$$

The inverse of the Difference transform is effectively the cumulative summation of the predicted signal being transformed. Equation 2 shows the more formal definition. Note that we need to save off y_l as it's needed to invert the transform.

$$y_k = \sum_{i=1}^{k-l} z_{k-i} + y_l \tag{2}$$

Logarithmic Transform The Logarithmic Transform, can be helpful with non-linear trends. The transform involves taking the natural log of a signal. Equation 3 shows the inverse.

$$y_k = e^{z_k} \tag{3}$$

Normal Transform This seems to be referred to as Normalization and/or Standardization inconsistently depending on the community involves, as statisticians appear to think of it this way, I'll stick with that term. We'll need the mean ($\bar{y}$) and standard deviation (σ) of the training data to invert the transform. Equation 4 and 5 cover the transform and it's inverse (Normalization (statistics), 2020).

$$z_t = \frac{y_t - \bar{y}}{\sigma} \tag{4}$$

$$y_t = \sigma(z_t + \bar{y}) \tag{5}$$

Time Series Decomposition The goal of time series decomposition is to identify any trend or seasonality behaviour, so we can account for appropriately. There are a number of tools available for this, but our approach will be to visually inspect the signal vs time, and the ACF, to try and determine if additive or multiplicative decomposition is required, and what periods of seasonality may be applicable. We'll then verify our visual inspection results using the seasonal decomposition function available in the statsmodel python package.

1.1.2 Model Selection

Model Selection involves using information gained up to this point, and through any previous iterations of the process, to choose a model, and in some cases associated data transforms to reduce bias in model residuals. Typically there is really no limit to the Model being chosen, as that choice is usually driven by the dataset, and the specific goal you have. For purposes of this project, we'll be constructing at least one of each of these models, and evaluating them all against the testing set to choose a final model.

Linear Regression Model Linear Regression Models assume that the dependent variable can be modeled against various independent variables using a linear form, as depicted in Equation 6 (Hyndman et al., 2014, p. 106).

$$y_t = \beta_0 + \beta_1 x_{1,t} + \beta_2 x_{2,t} + ... + \beta_k x_{k,t} + \epsilon_t \tag{6}$$

Benchmark Methods The following 3 methods are a subset of the 4 benchmarking methods referred to in *Forecasting Principles and Practice*. Note that these methods require no estimated parameters (Hyndman et al. (2014)).

Average Method This method is applied by simply taking the average of all the values available in the training set, and using that average as the forecast. Using $y_1, ..., y_T$ to denote our historical data, Equation 7 shows the forecast value $\hat{y}_{T+h|T}$, where h is the number of time steps ahead of T (Hyndman et al., 2014, p. 47).

$$\hat{y}_{T+h|T} = \bar{y} = (y_1 + \cdots + y_T)/T \tag{7}$$

Naive Method This method is applied by simply taking the most recent value available from the training set, and using that value as the forecast. This can be seen in Equation 8 (Hyndman et al., 2014, p. 48).

$$\hat{y}_{T+h|T} = y_T \tag{8}$$

Drift Method This method is applied by drawing a line between the first and last observations, and extrapolating into the future. In Equation 9, we can see that this is effectively the equation for a line (Hyndman et al., 2014, p. 49).

$$\hat{y}_{T+h|T} = y_T + h\left(\frac{y_T - y_1}{T-1}\right) \tag{9}$$

Simple Exponential Smoothing Simple Exponential Smoothing (SES) can be represented similarly to the Benchmarking Methods using it's component form, shown in Equation 10. Note that the equation includes two estimated parameters α, and l_0 (Hyndman et al., 2014, p. 189).

$$\begin{array}{cc} \text{Forecast equation} & \hat{y}_{t+h|t} = l_t \\ \text{Smoothing equation} & l_t = \alpha y_t + (1-\alpha) l_{t-1} \end{array} \tag{10}$$

Holt's Linear Trend Holt's Linear Trend Method can also be expressed in component form, as seen in Equation 11. Note that the equation includes four estimated parameters α, β^*, l_0, and b_0 (Hyndman et al., 2014, p. 193).

$$\begin{array}{cc} \text{Forecast equation} & \hat{y}_{t+h|t} = l_t + hb_t \\ \text{Level equation} & l_t = \alpha y_t + (1-\alpha) l_{t-1} \\ \text{Trend equation} & b_t = \beta^* (l_t - l_{t-1}) + (1-\beta^*) b_{t-1} \end{array} \tag{11}$$

Holt-Winters' Additive Method Holt-Winters' additive method can also be expressed in component form, as seen in Equation 12. Note that the equation includes six estimated parameters α, β^*, γ, l_0, b_0, and s_0 (Hyndman et al., 2014, p. 199).

$$\begin{array}{cc} \text{Forecast equation} & \hat{y}_{t+h|t} = l_t + hb_t \\ \text{Level equation} & l_t = \alpha y_t + (1-\alpha) l_{t-1} \\ \text{Trend equation} & b_t = \beta^* (l_t - l_{t-1}) + (1-\beta^*) b_{t-1} \\ \text{Seasonal equation} & s_t = \gamma(y_t - l_{t-1} - b_{t-1}) + (1-\gamma) s_{t-m} \end{array} \tag{12}$$

ARMA Models The Autoregressive Moving-Average Model, moves us away from the methods described previously, and gives a more scalable capacity for modeling complex systems, as the number of parameters available to us can be varied, in addition to those parameters being estimated. ARMA Models assume that a signal, and it's noise, can be recursively defined as shown in Equation 13 Where:

- $y(t)$ is the signal at time t
- $e(t)$ is the error at time t
- $a_1, ..., a_{na}$ represent the Autoregressive (AR) coefficients
- $b_1, ..., a_{nb}$ respresent the Moving Average (MA) coefficients
- na is said to be the AR Order
- nb is said to be the MA Order

$$y(t) + a_1 y(t-1) + ... + a_{na} y(t-na) = e(t) + b_1 e(t-1) + ... + b_{nb} e(t-nb) \quad (13)$$

1.1.3 Order Determination

Not all of the models we'll be constructing require an order determination step, in fact it only really applies to ARMA and Linear Regression Models.

ARMA Model Identification ARMA Model Identification (or Order Determination) is all about figuring out the AR and MA orders (na and nb from Equation 13) that best fit the dataset. This is necessary to bound the optimization problem that we'll discuss next. The primary tool we've discussed in class for identifying the appropriate ARMA Model is the GPAC array, which can be used with visual inspection to identify patterns indicating possible ARMA models that could fit the data. Let's look at building GPAC Arrays in more detail (Note: in the source code we refer to GPAC Arrays and GPAC Tables interchangeably).

GPAC Array The GPAC array, is ultimately an array of partial autocorrelation based autoregressive coefficient estimates using the Yule-Walker equations for ARMA processes of different orders. Specifically the solution for the last AR coefficient, (a_{na}, using the notation in Equation 13). The Box-Jenkins approach to ARMA Model Identification was a partial autocorrelation (PAC) function (Equation 14), using Cramers Rule, and the first k Yule-Walker equations to solve for the kth partial autocorrelation coefficient for an ARMA(k, 0) process (Woodward and Gray, 1981, p. 1-5).

j	$k=1$	$k=2$	$k=3$	$\dots$
0	$\phi_{11}(0)$	$\phi_{22}(0)$	$\phi_{33}(0)$	$\dots$
1	$\phi_{11}(1)$	$\phi_{22}(1)$	$\phi_{33}(1)$	$\dots$
2	$\phi_{11}(2)$	$\phi_{22}(2)$	$\phi_{33}(2)$	$\dots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Table 1: GPAC Array

$$\phi_{kk} = \frac{\begin{bmatrix} 1 & Ry(1) & Ry(2) & \dots & Ry(k-2) & Ry(1) \\ Ry(1) & 1 & Ry(1) & \dots & Ry(k-3) & Ry(2) \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ Ry(k-1) & Ry(k-2) & Ry(k-3) & \dots & Ry(1) & 1 \end{bmatrix}}{\begin{bmatrix} 1 & Ry(1) & Ry(2) & \dots & Ry(k-2) & Ry(k-1) \\ Ry(1) & 1 & Ry(1) & \dots & Ry(k-3) & Ry(k-2) \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ Ry(k-1) & Ry(k-2) & Ry(k-3) & \dots & Ry(1) & 1 \end{bmatrix}} \tag{14}$$

Woodward and Gray's 1981 paper introduced GPAC, as a more generalized form of that estimate, which isn't restricted to ARMA(k, 0), and is capable of estimating an ARMA(k, j) processes k'th autoregressive coefficient, by using a subset of the Yule-Walker equations used in the Box-Jenkins approach. Giving us $\phi_{kk}(j)$ as shown in Equation 15 (Woodward and Gray, 1981, p. 4).

$$\phi_{kk}(j) = \frac{\begin{bmatrix} Ry(j) & Ry(j-1) & \dots & Ry(j-k+2) & Ry(j+1) \\ Ry(j+1) & Ry(j) & \dots & Ry(j-k+3) & Ry(j+2) \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ Ry(j+k-1) & Ry(j+k-2) & \dots & Ry(j+1) & Ry(j+k) \end{bmatrix}}{\begin{bmatrix} Ry(j) & Ry(j-1) & \dots & Ry(j-k+2) & Ry(j-k+1) \\ Ry(j+1) & Ry(j) & \dots & Ry(j-k+3) & Ry(j-k+2) \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ Ry(j+k-1) & Ry(j+k-2) & \dots & Ry(j+1) & Ry(j) \end{bmatrix}} \tag{15}$$

From this definition of $\phi_{kk}(j)$, the GPAC Array construction is shown in Table 1. Where j, for each row is the Moving Average Order, and k for each column is the Auto-Regression Order (Woodward and Gray, 1981, p. 5).

Generally based on in-class discussion the interpretation of the GPAC table, is a matter of attempting to identify both the MA and AR orders. for the AR order, we attempt to find repeated numbers in a given column, indicating a high likely-hood that the underlying data has an AR order of k (from the column identified). For the MA order, we attempt to find a row with a large number of 0's. and say that the MA order of the data is j associated with that row.

In developing the implementation of the *gpac* function found in the python code (see appendix) I borrowed some variable names for various helper functions from a follow-on paper, which referred to the numerator of $\phi_{kk}(j)$ as A, and the denominator as B. The described process for building those matrices was much clearer, and further broken out, which aided in the implementation (Woodward and Gray, 1988).

Feature Selection Performing Feature Selection fits under same category as Order Determination, as ultimately the goal is to identify the number of terms the equations will have in it, with similar challenges around overfitting and importance. However the process of identifying the appropriate features is very different. The approach we'll be taking in this case if fairly straight forward. First we'll filter the features to achieve a 95% confidence that each individual feature improves predictive information versus an intercept only model, using an F-Test. Next, we filter based on a threshold, to ensure none of features are overly correlated. Then we'll proceed to iterative removal of insignificant features (based on a t-test) while keeping an eye on several metrics, primarily RMSE, and the white-noise behavior of the residuals. If successful the manual process should yield us simplified model, which generalizes well for use on the test set.

Another aspect of Feature Selection for Regression Models is feature engineering, while we haven't done a lot with that in this course, the paper which was published along-side this course did include a couple engineered features, so we'll experiment with at least one of those, that the paper indicates is significant. More on that later.

1.1.4 Parameter Estimation

Having identified the Model, performed the Order Determination (or Features Selection), we need find the estimated coefficients for the model. To achieve this for regression model, the best approach is usually the Least Squares Estimation. For the ARMA models, we apply a Maximum Likelihood Estimation (MLE) Algorithm called the Levenberg-Marquardt Algorithm (LM or LMA). It's worth noting that while SES and Holt's Linear Method based models do require parameters estimation to work effectively, we are doing that by hand, understanding that we will get sub-optimal results from it. An interesting expansion on the work presented in this paper would be to further generalize the LMA implementation, for application to the SES, Holt's Linear Method, and Holt-Winter Method based models. For now, we'll attempt to hand-jam the estimated parameters for any SES and Holt's Linear Method based models, and will utilize statsmodels implementation of the Holt-Winter Method for a better optimization solution there, though that does run extremely slowly against the number of samples we have in the training set.

Least Squares Estimation Least Squares Estimator (LSE) is a batch optimization that minimizes SSE (See Equation 30), in a single iteration. This is an

exceptionally fast and effective optimization technique, but it is only applicable to our Linear Regression Model. Building off of Equation 6, it's possible to show that the parameters vector $[\hat{\beta}_1, \hat{\beta}_2, ..., \hat{\beta}_k]^T$ can be estimated using Equation 16, where X and Y are as shown in Equation 17 (Least Squares, 2020). Note that the additional of the column of 1 in the X matrix adds the intercept term in $\hat{\beta}$, that can be removed, to estimate a model without an intercept.

$$\hat{\beta} = \left(X^T X\right)^{-1} X^T Y \tag{16}$$

$$X = \begin{pmatrix} 1 & x_{1,1} & x_{2,1} & \cdots & x_{k,1} \\ 1 & x_{1,2} & x_{2,2} & \cdots & x_{k,2} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_{1,T} & x_{2,T} & \cdots & x_{k,T} \end{pmatrix} Y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_T \end{pmatrix} \tag{17}$$

In the python code provided along with this project, we've developed classes to wrap the Linear Regression model operations, to reduce some of the boilerplate needed to appropriately construct X and Y. And in order to predict Y based on the estimated parameters as shown in Equation 18.

$$\hat{Y} = X\hat{B} \tag{18}$$

Levenberg-Marquardt Algorithm The LM Algorithm is an iterative optimization algorithm which combines both Gradient Descent, and the Gauss-Newton Method. It is sometimes referred to as Damped Least-Squares (Levenberg-Marquardt algorithm, 2020). The Algorithm consists of 3 steps, not counting some initialization, which can be considered step 0. Step 1, consists of backing out the noise based on an initial guess at the parameters θ, (in our case this is always $\vec{0}$). The function for generating e based on a set of parameters θ, is actually quite similar to the function dlsim, which we'd developed in previous labs. Only the result we're looking for, is $e(t)$ given $\vec{y}$, instead of $y(t)$ given generated noise $\vec{e}$. Using the computed $\vec{e}$, we can compute the SSE for the current model. Next, we estimate the matrix X, which with some help from the slides provided regarding the LM process, isn't too bad. The X matrix appears to effectively be the Jacobian Matrix mentioned online, but I didn't spend much time confirming that. Step 1 wraps up by using the computed e, and X values to construct an estimated gradient g and hessian A, needed by the Gradient Descent (Gradient descent, 2020), and Gauss-Newton (Gauss-Newton algorithm, 2020) components of the algorithm.

Step 2, is all about computing $\Delta\theta$, and computing metrics we'll use in Step 3, based on a new e, using the updated parameters. Primarily we're interested in SSE, as the algorithms uses it to determine convergence.

Step 3, decides whether the model has converged, and either decreases or increases μ based on whether SSE has increased or decreased. μ is the parameter which slides us back and forth between the Gauss-Newton method, and Gradient Descent as we iterate over new parameter sets.

A successful run of the LM algorithm implemented in the provided source code, returns a LMAResult class, that provides a number of helpful tools for access various results from the algorithms execution, and utilities for quickly generating forecast signals using the derived system.

1.1.5 Diagnostic Testing

After estimating the parameters for the selected Model, we'll need to evaluate the model's ability to predict the data. We'll be performing this step against the training set used to build the model. We'll estimate the confidence intervals for each coefficient, do a t-test to validate statistical significance of the parameters. And look closely at the residuals to understand if there is bias that we might be able to further reduce with another model. This will be done using tests similar to those mentioned in the Model Metrics portion below, though this step is performed against the training data to ensure we don't over fit the model by bleeding the testing data into model training and development process.

1.1.6 Survival Analysis and Forecasting

Finally, after we're happy with the model's performance against the training data, we'll evaluate the model against the testing dataset, using the Model Metrics described in more detail in below. All of these performance tests, must be performed on the test set, to ensure that they give us a better sense of the predictive or forecasting power of each model, and hopefully alert us to any over-fitting problems, when we compare the metrics to those previously computed against the training data. A dramatic change in performance between the training and testing data metrics, is a fair indication that the model was over-fitted to the training data, and won't generalize well when used for forecasting and prediction. Having compared all the metrics, and weighed our options, vs. our goals, we can select the best models for our application.

Model Metrics In addition to the mean, variance, and standard deviations of the residuals, we'll be computing the following performance metrics using the test set for each model.

Coefficient of Determination (r^2) This value is simply the square of the correlation coefficient (Pearson's r). In the context of regression problems it is specifically the square of the correlation coefficient between $\hat{y}$ (predicted) and y (actual). And is used as a description of the amount of variance explained by a given model. (Coefficient of determination, 2020). For clarity Equation 19 shows how to compute Pearson's r, between x, and y datasets such that x_t, and y_t are the element of x and y at time t, $\bar{y}$ is the mean of y, and $\bar{x}$ is the mean of x.

$$r = \frac{\sum (x_t - \bar{x})(y_t - \bar{y})}{\sqrt{\sum (x_t - \bar{x})^2}\sqrt{\sum (y_t - \bar{y})^2}} \tag{19}$$

Adjusted r^2 Regular r^2 can behave badly as extra explanatory values are added, giving a false indication of the model being well fit. The Adjusted r^2 was designed by Henri Theil to account for that issue by adjusting the r^2 value accordingly, as shown in Equation 20 (Coefficient of determination, 2020).

$$\bar{r}^2 = 1 - \left(1 - r^2\right) \frac{n-1}{n-p-1} \tag{20}$$

AIC The Akaike Information Criterion is primarily used for comparing models, rather than evaluating a single model. AIC quantifies the trade-off between goodness of fit, and the simplicity of the model. Equation 21 shows the formula or AIC, where k is the number of estimated parameters, and $\hat{L}$ is the maximum of the likelihood function (Akaike information criterion, 2020).

$$AIC = 2k - 2\ln\left(\hat{L}\right) \tag{21}$$

Based on the slides for class, and the text, we can approximate this, using Equation 22. As this related AIC to Sum of Squared Error (SSE) and the sample size T, it's a form that is much easier to compute (Hyndman et al., 2014, p. 131).

$$AIC = T\log\left(\frac{\text{SSE}}{T}\right) + 2(k+2) \tag{22}$$

AICc When sample size is small, there is a good chance that AIC will select models that have too many parameters. In a similar fashion to the Adjusted r^2, AICc is the corrected AIC. Equation 23 shows how the correction is performed (Hyndman et al., 2014, p. 132). Note I have adjusted the notation a bit to match that used in class.

$$AICc = AIC + \frac{2k^2 + 2k}{T - k - 1} \tag{23}$$

BIC The Bayesian information criterion is also based on the likelihood function, and is closely related to the Akaike Information Criterion. Both AIC and BIC can be increased with the addition of parameters, but doing so can result in overfitting. BIC's penalty term, for correcting this, is larger in comparison to AIC. Equation 24 shows BIC in a similar fashion to AIC, shown in Equation 21, where T is the number of data points, k is the number of estimated parameters, and $\hat{L}$ is maximum value of the likelihood function (Bayesian information criterion, 2020). As with AIC, we're provided with a more executable form in Equation 25 from the lecture notes, and textbook (Hyndman et al., 2014, p. 132). Note I have adjusted the notation of Equation 24 a bit to match that used in class.

$$BIC = \ln(T)k - 2\ln\left(\hat{L}\right) \tag{24}$$

$$BIC = T\log\left(\frac{SSE}{T}\right) + (k+2)\log(T) \tag{25}$$

Q Q is the test statistic for the Box-Pierce portmanteau test, and is a measure of whether the first h autocorrelations are significantly different from what would be expected from white noise. Given ACF r_k at lag k, a sample size T, and window of lags h, the Q test is shown in Equation 26. Determining the proper window size h, is something worth discussion. The book proposes a rule of thumb shown in Equation 27, where m is the period of seasonality (Hyndman et al., 2014, p. 60-61). This guidance was later updated in Rob Hyndman's blog, giving us rule of thumb shown in Equation 28, where m is the period of seasonality, and T is the number of data samples (Hyndman, 2014).

$$Q = T\sum_{k=1}^{h} r_k^2 \tag{26}$$

$$h = \begin{cases} 10 & \text{for non-seasonal data} \\ 2m & \text{for seasonal data} \end{cases} \tag{27}$$

$$h = \begin{cases} \min(10, T/5) & \text{for non-seasonal data} \\ \min(2m, T/5) & \text{for seasonal data} \end{cases} \tag{28}$$

Q^* Also a portmanteau test, the Ljung-Box test is a more accurate test based on the same concepts in the Box-Pierce test. Equation 29 shows the Ljung-Box test, we'll be using the definition of h as shown in Equation 28 for this project.

$$Q^* = T(T+2)\sum_{k=1}^{h} \frac{r_k^2}{(T-k)} \tag{29}$$

SSE Sum of Squared Error is frequently used as metric we can minimum for optimization algorithms. (In particular LMA, as discussed previously). It can also be used as a model selection criteria, in much the same way. The model that minimizes SSE, has the best fit against the test dataset. Equation 30 shows how SSE is related error e at time t, where T is the time steps available in the sample. (Hyndman et al., 2014, p. 130).

$$SSE = \sum_{t=1}^{T} \left(e_t^2\right) \tag{30}$$

RMSE Root Mean Squared Error is a good measure of overall fit against the mean. When used as the function being minimized, it will fit parameters to best forecast the mean. Equation 31 shows RMSE given error e_t at time t (Hyndman et al., 2014, p. 64).

$$RMSE = \sqrt{\text{mean}(e_t^2)} \tag{31}$$

MAE Mean Absolute Error is a good measure of overall fit against the median. When used as the function being minimized, it will fit parameters to best forecast the median. Equation 32 shows MAE given error e_t at time t (Hyndman et al., 2014, p. 64).

$$MAE = \text{mean}\left(|e_t|\right) \tag{32}$$

MAPE Mean Absolute Percentage Error is beneficial because it's unit free, making it a good choice when comparing forecast performances between data sets. MAPE given error e_t, and true value y_t for time t, is shown in Equation 33. It does have some significant disadvantages that are worth noting (Hyndman et al., 2014, p. 65):

- becomes undefined if the signal is zero ($y_t = 0$)
- assumes that the unit of measure involved has a meaningful zero
- puts a heavier penalty on negative errors than on positive errors

$$MAPE = \text{mean}\left(\left|\frac{100e_t}{y_t}\right|\right) \tag{33}$$

1.2 The Data

This data was collecting using a number of sensors installed in a house in Stambruges, about 24 km from the City of Mons, in Belgium (Candanedo et al., 2017, p. 84). It consists of both temperature and humidity data collected at 10 minute intervals over the course of several months, from sensors located in strategic locations about the house (Candanedo et al., 2017, p. 84). Maps of the house, showing how the sensor network was laid out, are provided in Figures 2 and 3.

The dataset includes weather data from the nearest airport weather station (Chievres Airport, Belgium) merged based on date and time, this gives us outdoor temperature, humidity, and pressure. As the weather data was hourly, the researchers applied linear interpolation to provide that data at 10 minute intervals, to match the data collection. Finally the dataset includes Appliance, and lighting energy use data (Wh) collected every 10 minutes as well. (Candanedo et al., 2017, p. 85).

Figure 2: Sensor Locations. First Floor. C refers to the hub used for collecting the data (Candanedo et al., 2017, p. 85).

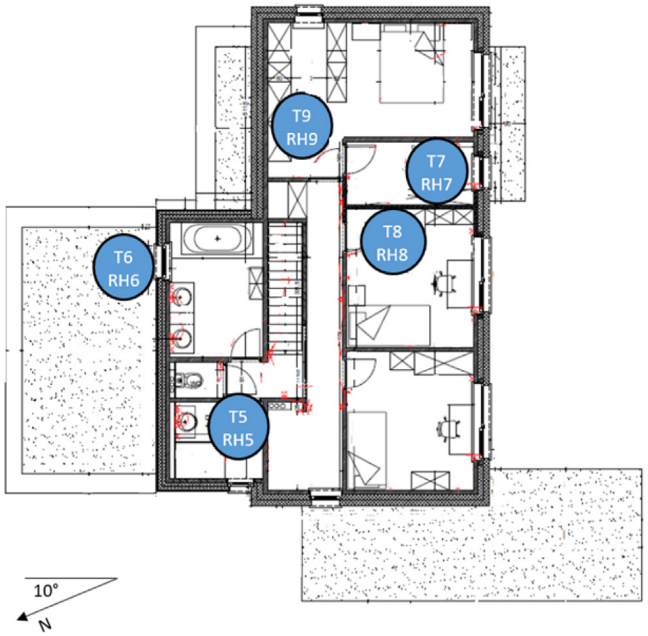


Figure 3: Sensor Locations. Second Floor. (Candanedo et al., 2017, p.85)

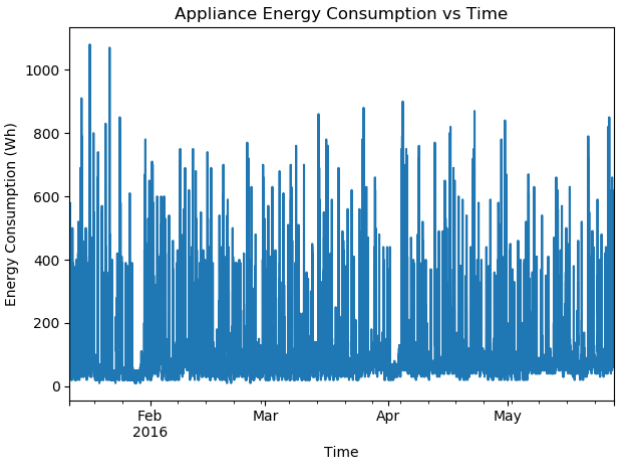


Figure 4: Application Energy Consumption vs Time

plot vs time The dependent variable can be seen plotted versus time in Figure 4. In that graph we can see fairly regular spikes in the signal, which appears to be caused by some daily seasonality in the dataset. We'll explore this further later-on.

ACF Plot The dependent variables Autocorrelation Function (ACF) Plots for 20 lags, 50 lags, and 200 lags are shown in Figures 5, 6, and 7 respectively. Just looking at the 20 lag ACF seems to indicate that the dataset shows MA, and AR behavior and then becomes quite noisy. Expanding the window to 50 and 200 lags, gives us a picture that seems more consistent with some form of seasonal behavior.

Correlation Matrix The Correlation Matrix (generated using the seaborn python package, and pearson's correlation coefficient) is shown in Figure 8. Clearly from this matrix we can see that we'll need to be cautious with regards to multicolinearity.

Data Cleaning It appears that for our purposes, no additional data cleaning is necessary. The dataset already contains no missing data or NANs, and the imputation needed to translate the hourly weather data into our dataset's 10 minute observation time has already been completed (Candanedo et al., 2017, p. 85).

Splitting We've split the dataset into training and testing data, using 67% of the dataset for training, and 33% of the dataset for testing. This gives us $15,788$ samples in our training set, and $3,947$ samples in our testing set. The split was

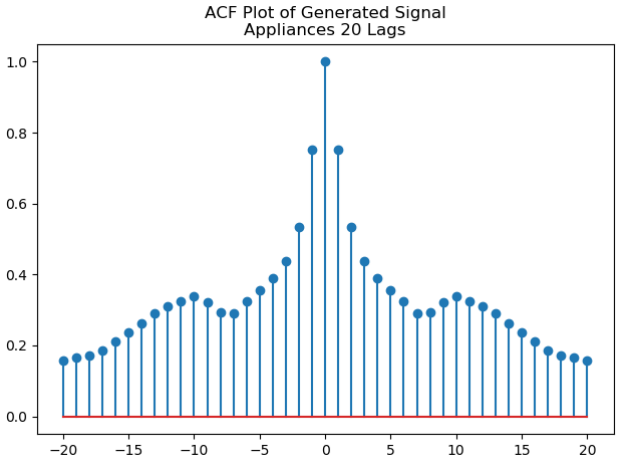


Figure 5: Application Energy Consumption ACF Plot (20 lags)

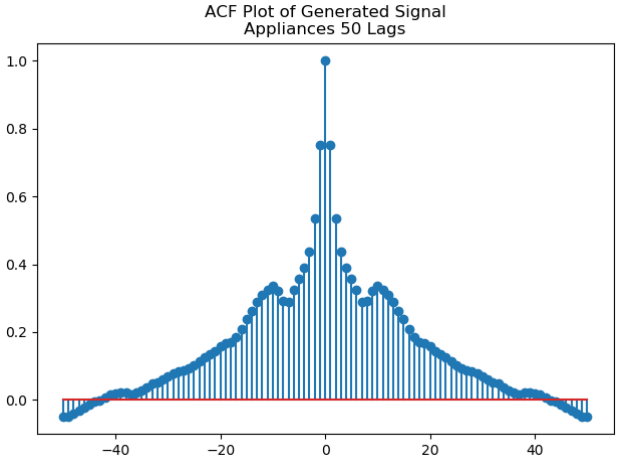


Figure 6: Application Energy Consumption ACF Plot (50 lags)

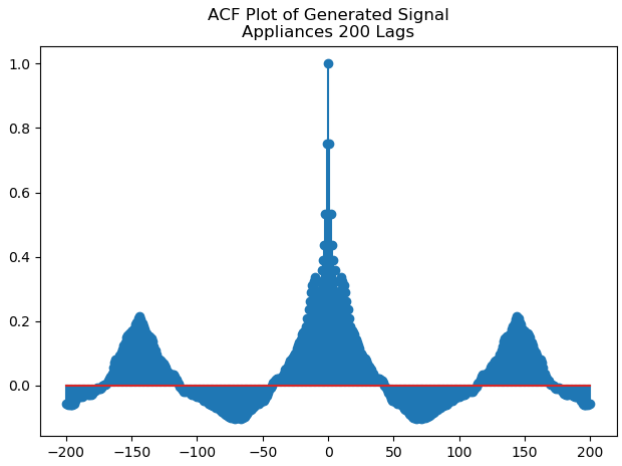


Figure 7: Application Energy Consumption ACF Plot (200 lags)

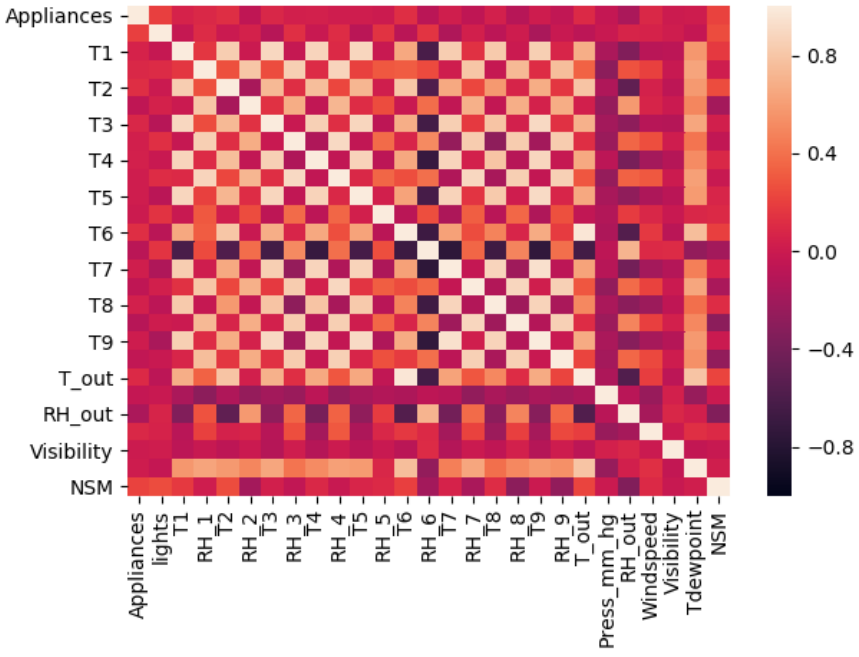


Figure 8: Correlation Matrix prior to Test/Train split.

performed using sklearn's *train_test_split* function, with shuffle disabled. With shuffle disabled we can apply the same testing and training split to both the regression and time-series analysis approaches.

2 Stationary

Starting with our visual inspection of the dataset, we'll need to take a look at couple different graphics based only on the training data.

- Appliance Energy Consumption vs Time (Figure 9)
- ACF at various lags (Figures 10, 11, and 12)
- Histogram (Figure 13)

Based on visual inspection, I do have some concern regarding the possible seasonality shown in the ACF plot (Figure 12 in particular), which likely correspond with the sharp peaks we see happening fairly regularly in the plot of the signal vs time (Figure 9). It seems we may need to account for that further in developing our models. The ADF Test gives us a test statistic of -19, and a p-value 0.0, it's nearly impossible for a unit root to exist within the data, despite the seasonal-like behavior. I suspect this may be due to inconsistencies in the seasonality, as it seems to pause, or change briefly at irregular intervals. We'll explore the seasonality more in the Time-Series Decomposition section of the paper. Another thing we may need to account for in our model building, likely through some combination of transforms, is the skew seen reflected in both the signal vs time (Figure 9), and Histogram plots (Figure 13).

Though need to revisit the need for transforms to better tackle skew, and seasonality for specific models, the data behaves stationary enough without adjustment that it seems reasonable to proceed with the seasonal decomposition prior to adding any of these. I'm thinking the transforms required will vary based on the method/modeling technique we're applying.

3 Time Series Decomposition

Continuing from our visual inspection for determining if the data is stationary, we saw what appears to be daily seasonality. Let's see if that decomposition, and hourly, and weekly as well, make any sense. In Figure 14, we can see that the hourly decomposition isn't very useful, as the Trend looks pretty much the same observed this won't help our predictions much. The daily decomposition in Figure 16 looks a little more useful as the trend is a little cleaner, but it doesn't appear to really pull a clear trend line for us. We'll probably attempt some of this using a seasonal method, but I'm thinking applying it for all the models probably isn't warranted. Lastly looking at the weekly decomposition in Figure 16, we actually do see a fairly nice clean-up for the trend line, with only a couple regions of 'extra' variability. Looking at these, it appears we certainly

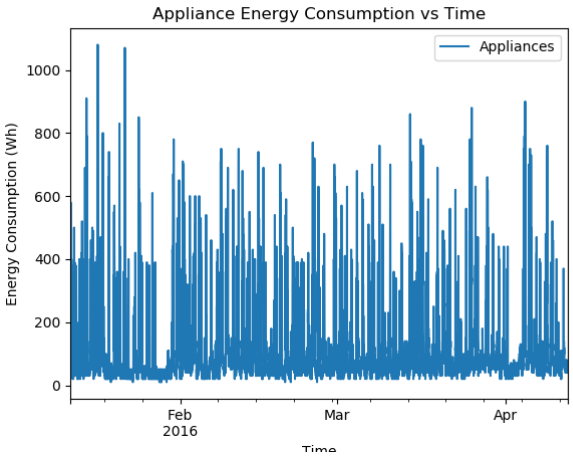


Figure 9: Application Energy Consumption vs Time (Training Data)

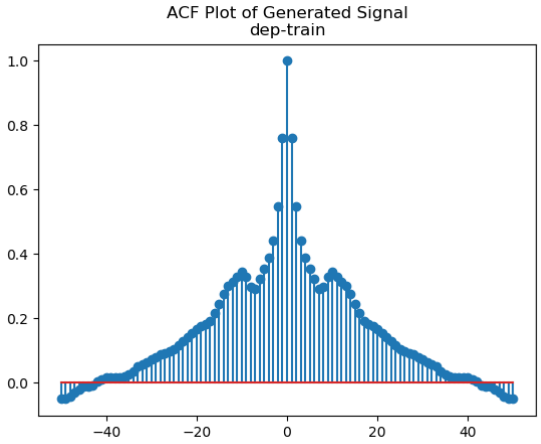


Figure 10: Application Energy Consumption ACF Plot (Training Data, 50 lags)

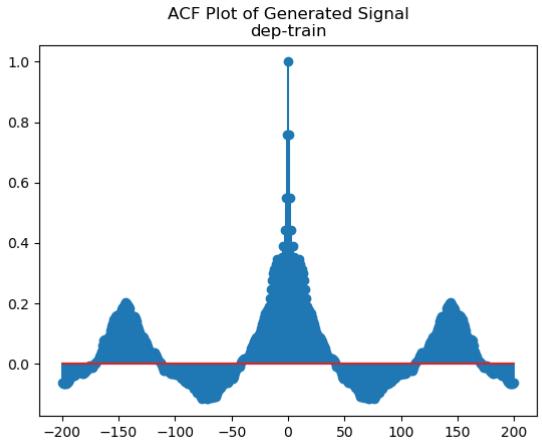


Figure 11: Application Energy Consumption ACF Plot (Training Data, 200 lags)

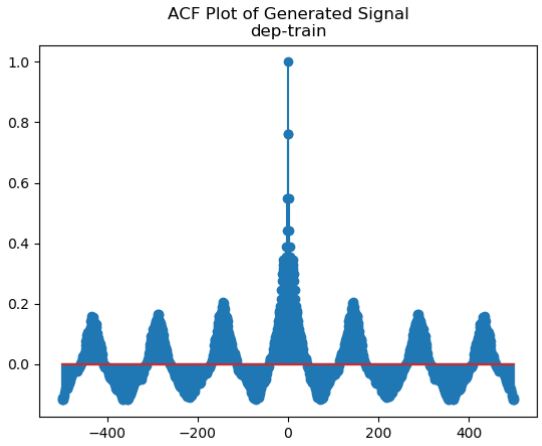


Figure 12: Application Energy Consumption ACF Plot (Training Data, 500 lags)

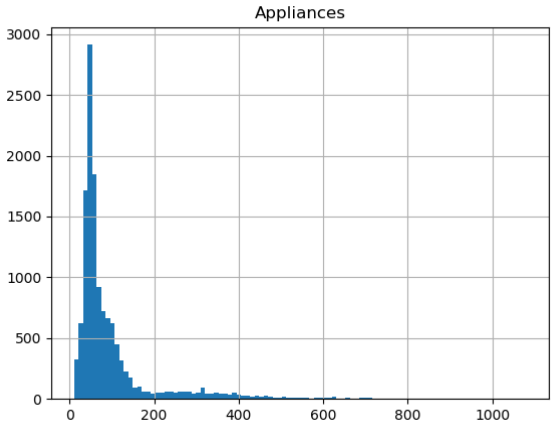


Figure 13: Application Energy Consumption Histogram (Training Data)

Model	Adj r^2	AIC	BIC	Q	RMSE	Res. Mean	Res. Std Dev
Avg	0.000	110883	110898	22585	94.4	1.15	94.3
Naive	0.000	113804	113819	22585	105.4	-46.9	94.3
Drift	0.000	114086	114101	22599	106.5	-49.4	94.4
SES	0.000	112938	112968	22585	102.0	-38.7	94.3
Linear	0.000	116278	116323	22585	115.7	-66.9	94.3
Holt-Winter	0.058	113433	113493	19445	103.9	-48.5	91.8

Table 2: Summary of Holt-Winters Section Models

have multiple seasonal effects occurring, but due to the inconsistency (possible due to multiple season impacts overlapping) it may be difficult for us to cleanly decompose the signal further.

4 Holt-Winters Method

First, before applying the Holt-Winter Method, we'll be applying the 3 Benchmark Methods introduced previously. As order and parameter approximations aren't necessary for these models, we'll simply show the results, and discuss as necessary.

Table 2 summarize the model evaluation metrics from the methods used in this section.

Despite trying many different combinations, I've been unable to get better performance out of my Holt-Winter Models that shown in the table. The Holt-Winter Model is estimated using statsmodels api, I've turned off the trend component of the model as they performed even less well with additive trend

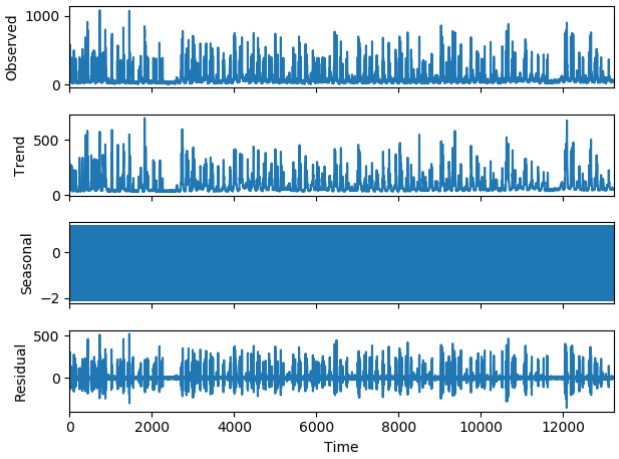


Figure 14: Time Series Decomposition - Hourly Seasonal Period

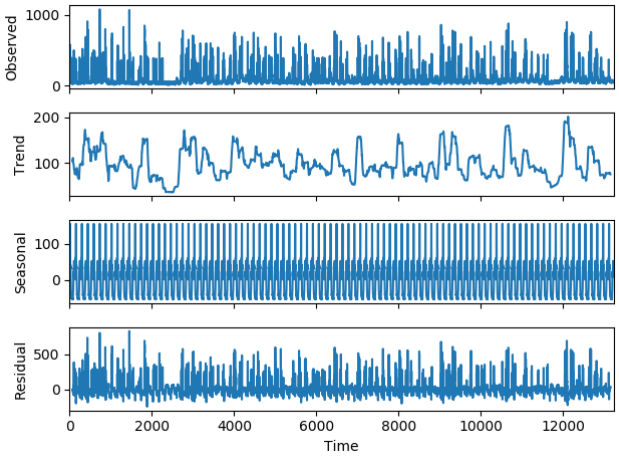


Figure 15: Time Series Decomposition - Daily Seasonal Period

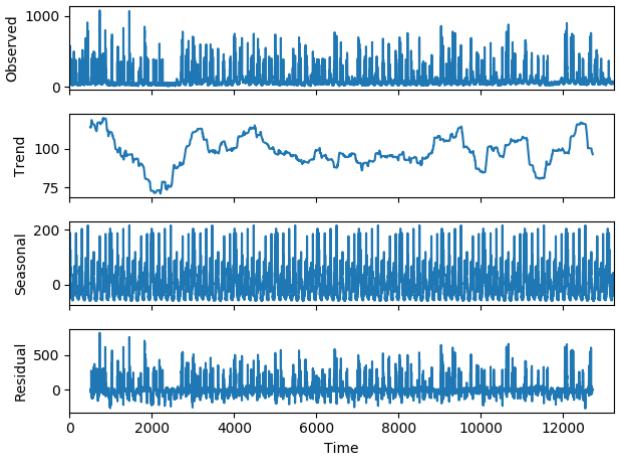


Figure 16: Time Series Decomposition - Weekly Seasonal Period

enabled. I believe that behavior was driven by the skewness of the distribution. I attempted to mitigate that some by applying logarithmic and normalization transforms simultaneously, it did seem to help, but not overly much. The Model summarized above was build using both transforms.

4.1 Benchmark Methods

4.1.1 Average Method

The forecast output is shown in Figure 17.

4.1.2 Naive Method

The forecast output is shown in Figure 18.

4.1.3 Drift Method

The forecast output is shown in Figure 19.

4.2 Simple Exponential Smoothing

The forecast output is shown in Figure 20.

4.3 Holt's Linear Method

The forecast output is shown in Figure 21.

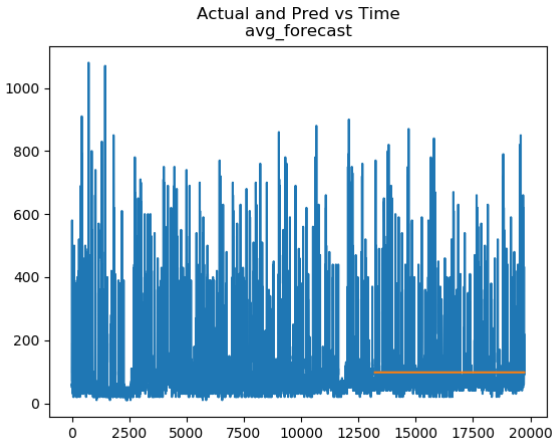


Figure 17: Average Method Forecast

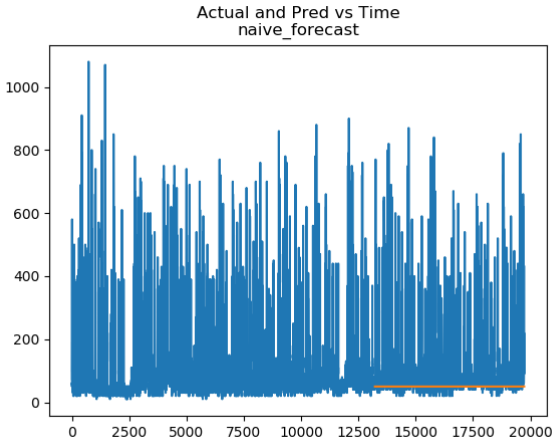


Figure 18: Naive Method Forecast

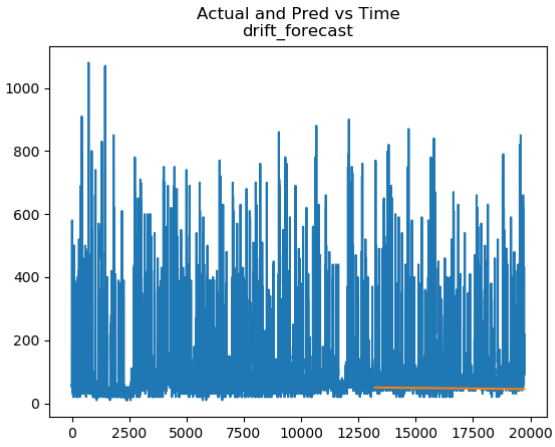


Figure 19: Drift Method Forecast

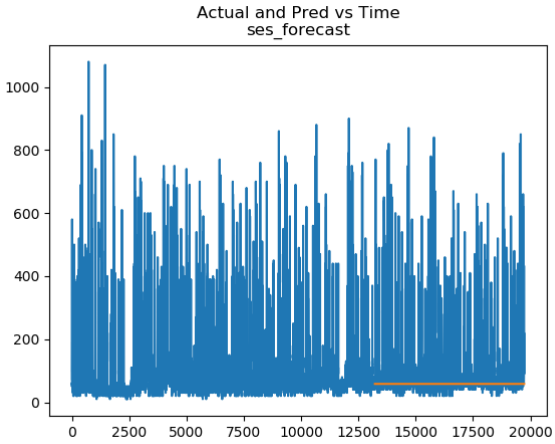


Figure 20: SES Method Forecast

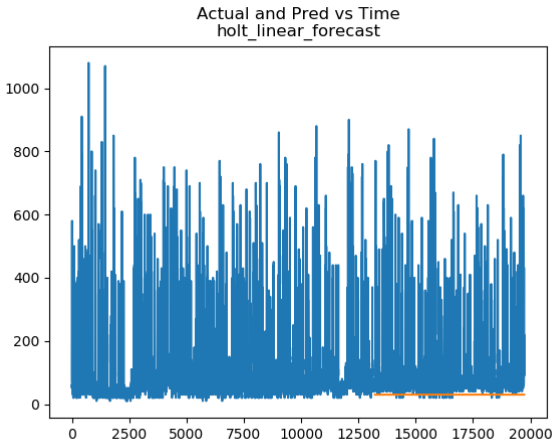


Figure 21: Holt Linear Method Forecast

4.4 Holt-Winter Method

The forecast output is shown in Figure 22.

5 Multiple Linear Regression Model

This section discusses 2 linear regression models, one based on our feature selection process described in the introduction(1), and one using a smaller subset of the features identified for use in the paper associated with this data set(2) (Candanedo et al., 2017, p. 87-88). Looking at how the feature selection was done in the paper, I believe, they really should have used only used the first 6 features identified by the Recursive Feature Elimination (RFE) Algorithm for the Linear Regression Model, rather than taking the minimum RMSE possible against the training set. They used 10-fold cross validation, which I suspect may be too many folds for this smaller dataset. I'm quite surprised that the optimal RMSE was considered minimum possible, though I suppose the cross-validation was thought to avoid any overfitting. Further, they used the same set of features for all the nonlinear models as well, seemingly causing those to over fit. We see a significant drop in the R^2, RMSE, MAE, and MAPE between the training and testing set for all three of their non-linear models, though to be fair there is minimal variation with regard to the Linear Regression Model. I have suspicions that may be more due to the noise of so many insignificant, and correlated, parameters. (Candanedo et al., 2017, p. 91). So we'll try the features identified by our process, and the first 6 features chosen by the authors of the paper.

We did bring in one engineered feature, that was described in paper as being very significant. Number of Seconds since Midnight (NSM), this appears to deal

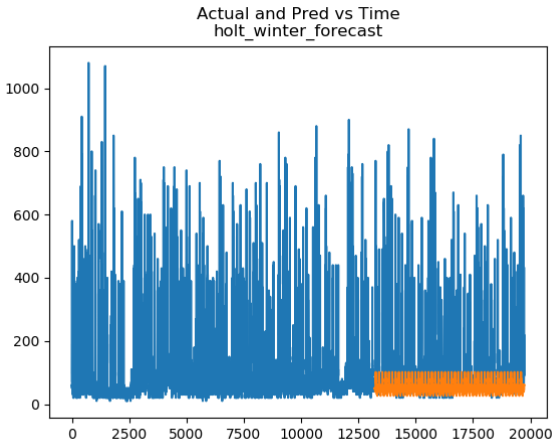


Figure 22: Holt-Winter Method Forecast

Model	Adj r^2	AIC	BIC	Q	RMSE	Res. Mean	Res. Std Dev
1	0.078	109958	110018	16662	91.1	5.4	90.9
2	0.048	110820	110880	18794	94.1	18.1	92.3

Table 3: Summary of Regression Model Evaluation

with some of the daily seasonality we'd seen in the Time Series Decomposition, but in a form more readily usable by the regression model.

Table 3 summarizes the metrics resulting from the two models. Model 1 is based on the Features Selection process using F-Test and T-Test as described, and Model 2 is based on the first 6 Feature selected by the RFE process utilized for the paper (Candanedo et al., 2017, p. 88). Model 1's prediction is plotted in Figure 23, and Model 2's prediction is plotted in Figure 24.

an ACF Plot of Model 1's residuals are provided in Figure

5.1 Feature Selection

After a lot of trial and error, I ended up adding a Normalization Transform (utilizing the StandardScaler, available in sklearn) on both the dependent and independent variables, which appears to improve the residual behaviours, but didn't impact r^2, or RMSE very much at all. Before testing each of the models, we invert the predicted dependent variables through the transform, to ensure model comparisons are still possible using the metrics.

The initial F-Test filtering, based on a 95% confidence threshold, gave us 3 features to remove: Visibility, RH_5 and RH_4.

Next we get a sorted list of features based on their F-test pvalue scores, and add the parameters in order, as long as none of the previously added features

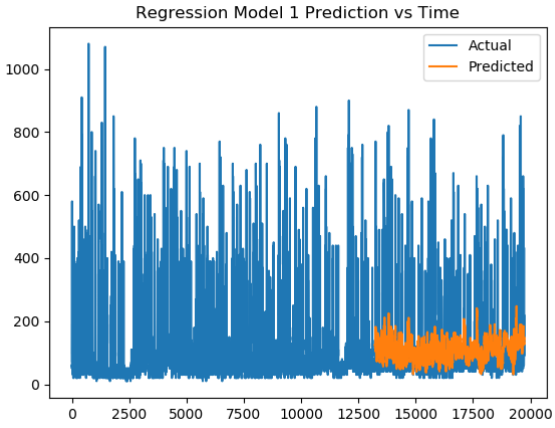


Figure 23: Regression Model 1 Prediction

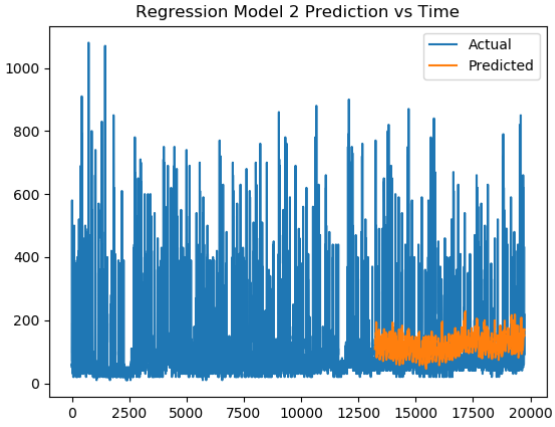


Figure 24: Regression Model 2 Prediction

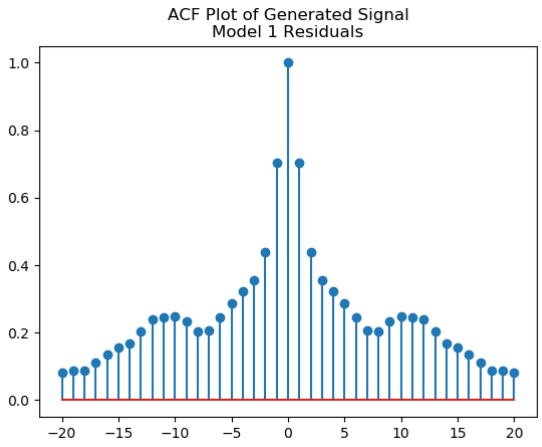


Figure 25: Regression Model 1 ACF Plot of Residuals

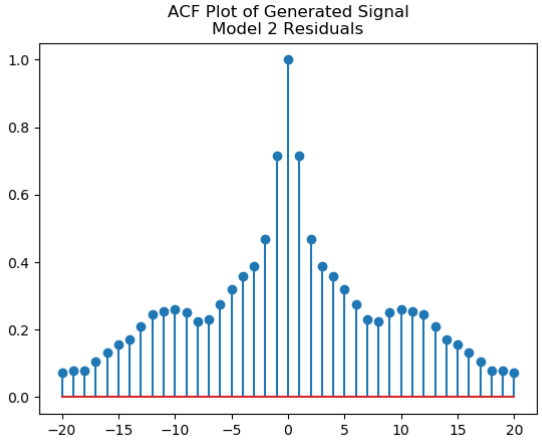


Figure 26: Regression Model 2 ACF Plot of Residuals

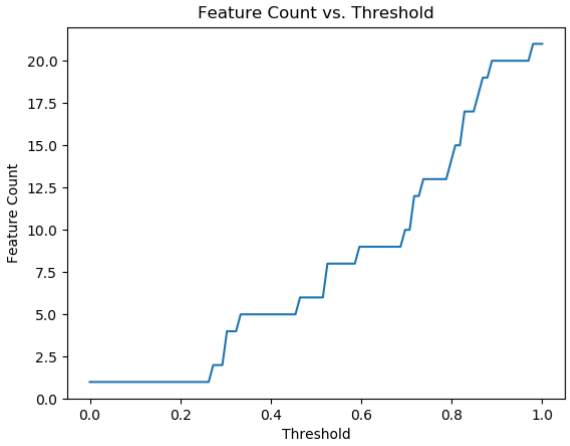


Figure 27: Feature Count vs Threshold

are overly correlated with next feature being added.

We define 'overly correlated' based on a threshold value. To get a little information on which to base that threshold value, it was necessary to run the feature sorting function for various thresholds, and plot the number of features vs the thresholds, as shown in Figure 27. Referring to this diagram, and the correlation plots we've reviewed previously (Figure 8), I settled on using a threshold of 0.8. Leaving us with 13 features.

Finally we incrementally remove features one at a time, until all the features are statistically significant to 99% confidence based on the t-tests. At which point we have our model, consisting of Lights, T2, RH_8, T3, RH_1, and T5. We'll call this model 1.

We then have Model 2, simply by referring to the research paper, which will consist of NSM, Lights, Pressure, RH_5, T3, and RH_3.

6 ARMA Model

6.1 ARMA Model Identification

The ACF Plot is shown in Figure 10.

After many iterations and experimentation, I settled on using two transforms for better ARMA modeling results. First we apply the logarithmic transform, then the normalization transform. This appears to help with the skewness we'd previously discussed, seen in the histogram (Figure 13).

The GPAC Array generated after these transforms were performed is shown in Figure 28. Based on this GPAC, I decided to look at an ARMA(1,0) and ARMA(4,3) model.

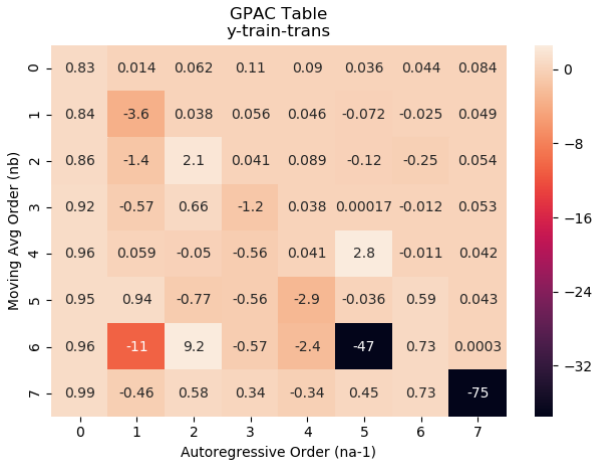


Figure 28: GPAC of Transformed Appliance Data

```
ARMA coefficients:
       label   coeff   lower   upper   t-test  p-value
       a_1     -0.8323 -0.8417 -0.8228 -2.385e+04      0
```

Figure 29: ARMA(1, 0) Coefficient Results

6.2 Parameter Estimation

For the ARMA(1,0) Model the coefficient table is shown in Figure 29. The LM algorithm converged in 2 iterations. All the parameters are significant. The Residual ACF Plot is available in Figure 30. The ARMA(1,0) predictions are show in Figure 31.

For the ARMA(4, 3) Model the coefficient table is shown in Figure 32. The LM Algorithm converged in 17 iterations. All the parameters are significant. The Residual ACF Plot is available in Figure 33. The ARMA(1,0) predictions are show in Figure 34.

6.3 ARMA Model Selection

Table 4 shows the model evaluation metrics for the final ARMA models. By a very slim margin, the ARMA(4, 3) model performs slightly better on the test data.

Model	Adj r^2	AIC	BIC	Q	RMSE	Res. Mean	Res. Std Dev
(1,0)	0.000	116159	116182	21739	115.2	7.1	114.9
(4,3)	0.001	116101	116168	25832	114.9	6.8	114.7

Table 4: Summary of ARMA Model Evaluation

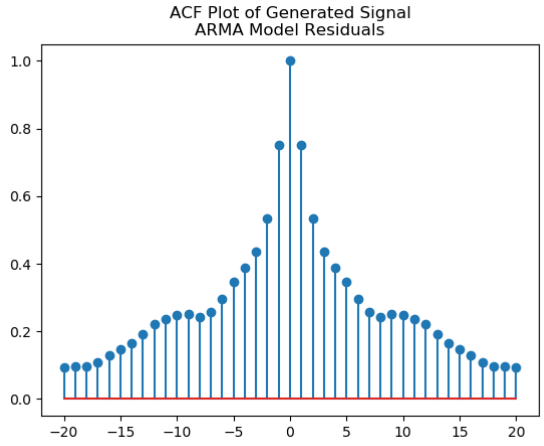


Figure 30: ARMA(1, 0) ACF of Residuals

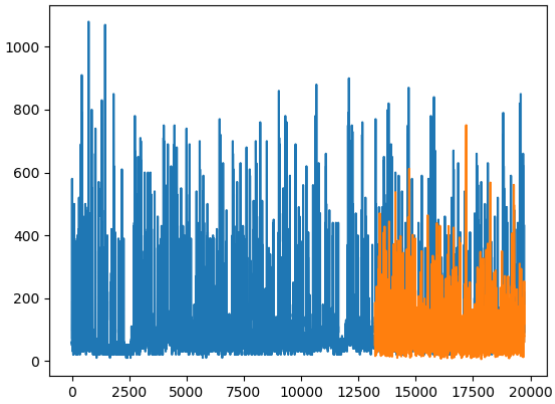


Figure 31: ARMA(1, 0) Forecast

ARMA coefficients:

	label	coeff	lower	upper	t-test	p-value
	a_1	-1.504	-1.552	-1.456	-4680	0
	a_2	-0.3471	-0.3802	-0.314	-6809	0
	a_3	1.505	1.457	1.554	-4643	0
	a_4	-0.6406	-0.6714	-0.6097	-7303	0
	b_1	-0.7374	-0.7875	-0.6873	-4501	0
	b_2	-0.9284	-0.9426	-0.9142	-1.584e+04	0
	b_3	0.7992	0.7551	0.8432	-5115	0

Figure 32: ARMA(4, 3) Coefficient Results

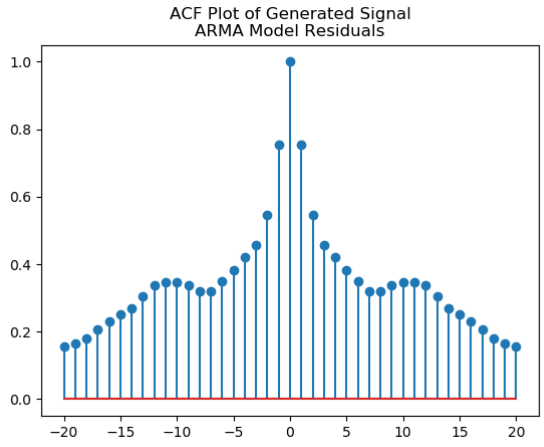


Figure 33: ARMA(4, 3) ACF of Residuals

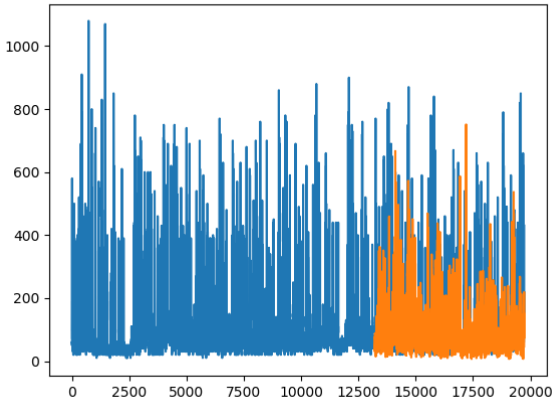


Figure 34: ARMA(4, 3) Forecast

Model	Adj r^2	AIC	BIC	Q	RMSE	Res. Mean	Res. Std Dev
ARMA(4, 3)	0.001	116101	116168	25832	114.9	6.8	114.7
LinReg 1	0.078	109958	110018	16662	91.1	5.4	90.9
Avg Method	0.000	110883	110898	22585	94.4	1.15	94.3
Holt-Winter	0.058	113433	113493	19445	103.9	-48.5	91.8

Table 5: Final Model Selection

7 Final Model Selection

Table 5 shows the model evaluation metrics for the best models we've developed throughout the course of this project. Which model we select is a bit nuanced. Clearly the model that performed the best against the dataset by a small margin, was the Linear Regression Model, using our custom feature selection technique. However, it's worth considering that the model is only applicable when the predictors are available.

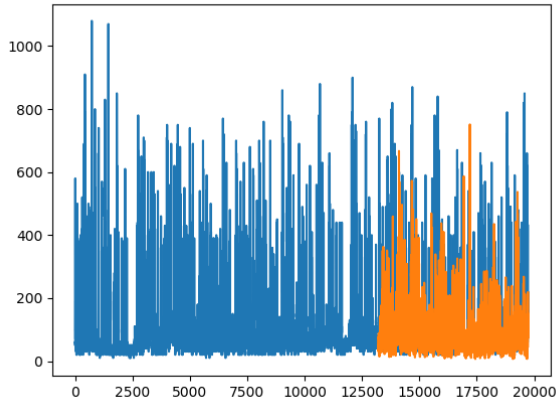
One benefit to the forecasting models, is that we don't need to collect all of the feature data in order to predict the Appliance Energy Consumption. Among the Forecasting Models, the best option we've come up with is the Avg Method. I still, might choose the ARMA model if I wanted an output that would be more visually similar to the time series data, but in this case, it wasn't able to outperform the Average bench marking method quantitatively. I believe this was likely the affect of additional seasonality I wasn't able to account for.

8 Conclusion

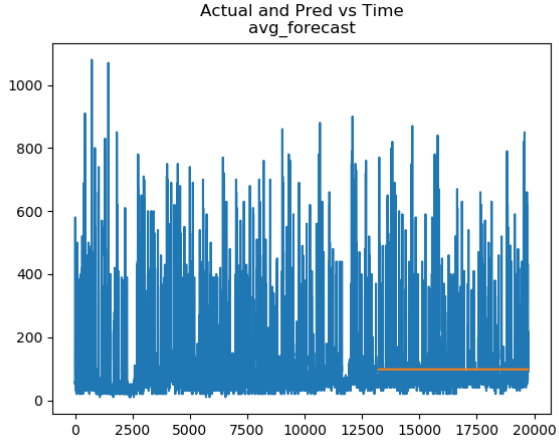
Ultimately, the conclusion, is that we failed to identify a good forecasting model for the Appliance Energy Consumption data. But we have managed to confirm that a Linear Regression Model performs better than the forecasting models we tried (assuming the predictors are available). Amongst the forecasting models, sadly the Average model outperformed our other attempts

Visually, it really does seem as though we should prefer the ARMA model, but despite that qualitative assessment, the quantitative metrics indicate that the Model doesn't actually map well to the original signal, so we should prefer the Average Method Based Model.

ARMA(4,3) Model:



(Best fitting model) Average Method:



Future Efforts Future Efforts associated with this work, should probably focus more on methods of accounting for the daily and weekly seasonal effects we were able to identify in the time series decomposition portion of the report. I believe that was biggest flaw in my approach, and would be my next step. I did briefly experiment with 1st and 2nd order lagged difference transforms to see if that might help, but immediately had issues inverting the transforms, as even low amount of variance within the transformed context, led to exaggerated variances related to t, for the long-range forecasts I was targeting. Referring to the text, I may be able to do something with decomposition information, simply by forecasting both the seasonal, and seasonally adjusted components separately (Hyndman et al., 2014, p. 177). Similarly there appear to options for decomposition accounting for multiple seasonal effects simultaneously. As we know there is weekly and daily impact, it seems likely those techniques would

greatly help our forecasting models (Hyndman et al., 2014, p. 351).

It also seems likely that Dynamic Models, which are able to combine ARMA-like behaviour with Regression techniques (Hyndman et al., 2014, p. 277), could be an extremely useful technique for this dataset, assuming the seasonality could be accounted for. Briefly skimming that's section of the text, leads me to believe it is.

Less critical, but as I've done my best to utilize code developed from scratch for the ARMA and Regression models, it would be interesting to generalize the LMA implementation, and attempt to use it for estimating the SES, Holt's Linear, and Holt-Winter model parameters in the future.

References

Akaike information criterion (2020). Akaike information criterion — Wikipedia, the free encyclopedia. [Online; accessed 21-April-2020].

Bayesian information criterion (2020). Bayesian information criterion — Wikipedia, the free encyclopedia. [Online; accessed 21-April-2020].

Candanedo, L. M., Feldheim, V., and Deramaix, D. (2017). Data driven prediction models of energy use of appliances in a low-energy house. *Energy and Buildings*, 140:81 – 97.

Coefficient of determination (2020). Coefficient of determination — Wikipedia, the free encyclopedia. [Online; accessed 21-April-2020].

Gauss-Newton algorithm (2020). Gauss–newton algorithm — Wikipedia, the free encyclopedia. [Online; accessed 21-April-2020].

Gradient descent (2020). Gradient descent — Wikipedia, the free encyclopedia. [Online; accessed 21-April-2020].

Hyndman, R. J. (2014). Thoughts on the ljung-box test.

Hyndman, R. J., Athanasopoulos, G., and OTexts.com (2014 2014). *Forecasting : principles and practice / Rob J Hyndman and George Athanasopoulos.* OTexts.com [Heathmont?, Victoria], print edition. edition.

Least Squares (2020). Proofs involving ordinary least squares — Wikipedia, the free encyclopedia. [Online; accessed 21-April-2020].

Levenberg-Marquardt algorithm (2020). Levenberg–marquardt algorithm — Wikipedia, the free encyclopedia. [Online; accessed 21-April-2020].

Normalization (statistics) (2020). Normalization (statistics) — Wikipedia, the free encyclopedia. [Online; accessed 21-April-2020].

Woodward, W. A. and Gray, H. L. (1981). On the relationship between the s array and the box-jenkins method of arma model identification. *Journal of the American Statistical Association*, 76(375):579–587.

Woodward, W. A. and Gray, H. L. (1988). Arma model identification using the generalized partial autocorrelation array. Technical Report SMU/DS/TR-222, Department of Statistical Science, Southern Methodist University.

A Python Code

Each of the python files used the generate the models, and graphics used in the report are included. Each driver file has it's own Section in this appendix. The utils package, developed to support this work, and each of the python files in it also have their own sections.

A.1 dataset_info_split.py

```
# This file generates the content for the Introduction
#   and performs the train-test-split
# Core Python Dependencies
from os.path import sep # attempts to maintain OS portability
from datetime import datetime

# Import Third-part Libraries
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

# Import things developed by me either
#   throughout the coursework, or
#   specifically for the term project
#   (provided in additional python files)
from utils import conf, data, stats, visualizations as viz

# init_folders takes care of setting up folders for organizing outputs
conf.init_tmp_folders()

# Load dataset, and deal with parsing data information
df = data.load_original_data()

# Basic Info
print('Shape of Dataset:', df.shape)

# Adding NSM Feature as described in the paper
#  Number of seconds since midnight
#  https://stackoverflow.com/questions/15971308/get-seconds-since-midnight-in-python
```

```
#  https://www.w3resource.com/python-exercises/date-time-exercise/python-date-time-exercise-
#  https://stackoverflow.com/questions/3743222/how-do-i-convert-datetime-to-date-in-python
def NSM(dt):
    return (dt - dt.replace(hour=0, minute=0, second=0, microsecond=0)).total_seconds()

# vectorize the function
# NSM = np.vectorize(NSM_simple)
dts = pd.to_datetime(df.index)
print(type(dts[0]), dts[0])
df['NSM'] = [NSM(dt) for dt in pd.to_datetime(df.index).tolist()]
print(df['NSM'].head())


# This takes a long time to generate, so we'll comment it out for submission
#   but it does give a very useful/pretty graphic for analyzing relationships
#   between variables.
# sns.pairplot(df, height=3)
# plt.savefig(f'{conf.tmp_graphics_folder}{sep}large-sns-pairplot-all-data')
# plt.figure()

# Let's drop things that really shouldn't be used to predict
#  or forecast Appliance energy use.
df = df.drop(['rv1', 'rv2'], axis=1)

# a - plot the dependent variable vs. time
df['Appliances'].plot()
plt.title('Appliance Energy Consumption vs Time')
plt.ylabel('Energy Consumption (Wh)')
plt.xlabel('Time')
plt.tight_layout()
plt.savefig(f'{conf.tmp_graphics_folder}{sep}dep-vs-time')
plt.figure()

# b - ACF
viz.acf_plot(df['Appliances'].to_numpy(), 'Appliances 20 Lags', 20)
plt.savefig(f'{conf.tmp_graphics_folder}{sep}dep-acf-20-lag')
plt.figure()

viz.acf_plot(df['Appliances'].to_numpy(), 'Appliances 50 Lags', 50)
plt.savefig(f'{conf.tmp_graphics_folder}{sep}dep-acf-50-lag')
plt.figure()

viz.acf_plot(df['Appliances'].to_numpy(), 'Appliances 200 Lags', 200)
plt.savefig(f'{conf.tmp_graphics_folder}{sep}def-acf-200-lag')
plt.figure()
```

```
# viz.acf_plot(df['Appliances'].to_numpy(), 'Appliances', 2000)
# plt.savefig(f'{conf.tmp_graphics_folder}{sep}def-acf-2000-lag')
# plt.figure()

# c - correlation matrix
# df.corr takes an optional callable, so let's use
#   the implementation of pearson's r that we developed
#   previously in the course
# https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.corr.html
corr_table = df.corr(
    method=stats.corr
)

# locked color scale bounds, based on theoretical min and max
# https://seaborn.pydata.org/generated/seaborn.heatmap.html
sns.heatmap(
    corr_table,
    vmin=-1, vmax=1
)
plt.tight_layout() # fixes issue with image bounds cutting off labels
plt.savefig(f'{conf.tmp_graphics_folder}{sep}corrplot-all-beforesplit')
plt.figure()


# d - cleaning procedures
print('Info:', df.info())

# based on this info, there are no missing values to account for

# e - split training and testing sets
y = df['Appliances']
x = df.drop(['Appliances'], axis=1)
split_labels = ['x_train', 'x_test', 'y_train', 'y_test']
splits = dict(zip(split_labels, train_test_split(
    x,
    y,
    shuffle=False,
    test_size=0.33
)))

# f - makes sense... let's check sizes of train and test split
# Let's save it all to the data folder
for label, split in splits.items():
    print(f'{label}:', split.shape, type(split))
    split.to_csv(f'{conf.tmp_data_folder}{sep}{label}.csv', header=True)
```

A.2 stationary_testing.py

```
# This file dedicated to performing stationarity testing
#   and if necessary developing transforms needed for
#   getting stationary behaviour from the dataset
# For consistency, we'll perform this testing against the
#   training set build in dataset_info_split.py
# Core Python Dependencies
from os.path import sep # attempts to maintain OS portability

# Import Third-part Libraries
import matplotlib.pyplot as plt
from statsmodels.tsa.stattools import adfuller

# Import things developed by me either
#   throughout the coursework, or
#   specifically for the term project
#   (provided in additional python files)
from utils.stats import corr, cor_and_ttest, acf_plot
from utils.data import load_y_train
from utils.conf import tmp_graphics_folder

# Testing for stationary with the dependent variable
#   training data only
y_train = load_y_train()

# plt vs time (just training set)
y_train.plot()
plt.title('Appliance Energy Consumption vs Time')
plt.ylabel('Energy Consumption (Wh)')
plt.xlabel('Time')
plt.tight_layout
plt.savefig(f'{tmp_graphics_folder}{sep}dep-train-vs-time')
plt.figure()

# histogram
y_train.hist(bins=100)
plt.savefig(f'{tmp_graphics_folder}{sep}dep-train-hist')
plt.figure()

# acf plot
acf_plot(y_train.to_numpy(), 'dep-train', 50)
plt.savefig(f'{tmp_graphics_folder}{sep}dep-train-acf-50')
plt.figure()
```

```
acf_plot(y_train.to_numpy(), 'dep-train', 200)
plt.savefig(f'{tmp_graphics_folder}{sep}dep-train-acf-200')
plt.figure()

acf_plot(y_train.to_numpy(), 'dep-train', 500)
plt.savefig(f'{tmp_graphics_folder}{sep}dep-train-acf-500')
plt.figure()

# adf test
print('ADF Test p-value for dependent training data:', adfuller(y_train.to_numpy().reshape(-
```

A.3 decomposition.py

```
# This file will find a decomposition that is reasonable
#   for the dataset.
from os.path import sep # attempts to maintain OS portability

# Import Third-part Libraries
from statsmodels.tsa.seasonal import seasonal_decompose
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

# Import things developed by me either
#   throughout the coursework, or
#   specifically for the term project
#   (provided in additional python files)
from utils.data import load_y_train, weekly_freq, daily_freq, hourly_freq
from utils.conf import tmp_graphics_folder

# Load data and convert to numpy array
y_train = load_y_train()
data = y_train.to_numpy().reshape(-1)

# really doesn't seem to be a need to look at a multiplicative model...
#   found a better way to think of the freq parameter... based on
#   ExponentialSmoothing model's documentation.
#   https://www.statsmodels.org/dev/generated/statsmodels.tsa.holtwinters.ExponentialSmoothi
#   "The number of periods in a complete seasonal cycle, e.g., 4 for quarterly data or 7 for
res = seasonal_decompose(data, model='additive', freq=weekly_freq)
res.plot()
plt.savefig(f'{tmp_graphics_folder}{sep}dep-decompose-additive-weekly')
plt.figure()
```

```
# let's try daily
#   number of minutes in a day
mins_per_day = 60*24
res = seasonal_decompose(data, model='additive', freq=daily_freq)
res.plot()
plt.savefig(f'{tmp_graphics_folder}{sep}dep-decompose-additive-daily')
plt.figure()

# let's try hourly
#   number of minutes in a hour
mins_per_hour = 60
res = seasonal_decompose(data, model='additive', freq=hourly_freq)
res.plot()
plt.savefig(f'{tmp_graphics_folder}{sep}dep-decompose-additive-hourly')
plt.figure()
```

A.4 holt.py

```
# This file is responsibly for applying
#   the Holt-Winters Method to the dataset
# Core Python Dependencies
from os.path import sep # attempts to maintain OS portability

# Import Third-part Libraries
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

# Import things developed by me either
#   throughout the coursework, or
#   specifically for the term project
#   (provided in additional python files)
from utils.data import load_y_train, load_y_test, weekly_freq, daily_freq, hourly_freq
from utils.conf import tmp_graphics_folder
import utils.models as mods
import utils.stats as stats
from utils.arma import compose_transform, logarithmic_transform, normalization_transform

# Let's load y_train and y_test.
#   attempt to fit and predict using a holtwinter model
y_train = load_y_train()
y_test = load_y_test()

# Average
```

```
forecast = mods.avg_trainer(y_train)
y_forecasted = forecast(len(y_test))
y_test['avg_forecast'] = pd.Series(y_forecasted, index=y_test.index)

# Naive
forecast = mods.naive_trainer(y_train)
y_forecasted = forecast(len(y_test))
y_test['naive_forecast'] = pd.Series(y_forecasted, index=y_test.index)

# Drift
forecast = mods.drift_trainer(y_train)
y_forecasted = forecast(len(y_test))
y_test['drift_forecast'] = pd.Series(y_forecasted, index=y_test.index)

# Simple Exponential Smoothing
#  TODO: Can we use LMA to optimize alpha, a_0
forecast = mods.ses_trainer(y_train)
y_forecasted = forecast(len(y_test))
y_test['ses_forecast'] = pd.Series(y_forecasted, index=y_test.index)

# Holt Linear Method
#  TODO: Can we use LMA to optimize alpha, beta, l_0, b_0
forecast = mods.holt_linear_trainer(y_train,)
y_forecasted = forecast(len(y_test))
y_test['holt_linear_forecast'] = pd.Series(y_forecasted, index=y_test.index)

# Holt-Winter
#  This takes a very long time... wow.
#  TODO: Can we implement our own version of this?
#  TODO: Can we use LMA to optimize for the parameters? Maybe faster??? hmmm..

y_train_np = y_train.to_numpy().reshape(-1, 1)
y_train_scaled, inverter = compose_transform(
    logarithmic_transform,
    normalization_transform
)(y_train_np)

forecast = mods.holt_winters_trainer_full(y_train_scaled, trend=None, seasonal_periods=int(c
y_forecasted = forecast(len(y_test))
y_test['holt_winter_forecast'] = pd.Series(inverter(y_forecasted), index=y_test.index)


# plot all the forecasts
cols = list(y_test.columns)

y_tmp = y_test['Appliances'].to_numpy().reshape(-1, 1)
```

```
y_train = y_train.to_numpy().reshape(-1, 1)
y_actual = np.append(y_train, y_tmp)
xs = list(range(y_train.shape[0], y_actual.shape[0]))

cols.remove('Appliances')

# assuming this should be the number of parameters
#  applying to the forecast
#  so i'm ignoring parameters that are
#  only used in the fit phase of the model.
num_params = {
    'avg_forecast': 0,
    'naive_forecast': 0,
    'drift_forecast': 0,
    'ses_forecast': 2,
    'holt_linear_forecast': 4,
    'holt_winter_forecast': 6,
    'holt_winter_forecast_normalization': 6
}

for col in cols:
    legend = ['Actual']
    legend.append(col)
    plt.plot(y_actual)
    y_tmp2 = y_test[col].to_numpy().reshape(-1, 1)
    plt.plot(xs, y_tmp2)
    plt.title(f'Actual and Pred vs Time\n{col}')
    plt.savefig(f'{tmp_graphics_folder}{sep}{col}-actual-pred-vs-time')
    plt.figure()
    print('Model Metrics for', col)
    stats.print_metrics(y_tmp, y_tmp2, num_params[col], y_train.shape[0])
```

A.5 feature_selection_regression.py

```
# This file is responsible for performing feature selection

# Core Python Dependencies
from os.path import sep # attempts to maintain OS portability

# Import Third-part Libraries
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
```

```
from sklearn.feature_selection import f_regression
from sklearn.preprocessing import StandardScaler

# Import things developed by me either
#   throughout the coursework, or
#   specifically for the term project
#   (provided in additional python files)
from utils.data import load_y_train, load_x_train
from utils.conf import tmp_graphics_folder
from utils.stats import ttest, corr, cor_and_ttest, print_metrics
from utils.regression import LinRegModel

# function for filtering through through best ftest scored features
#   based on a maximum correlation threshold against features
#   that have already been accepted
def filter_ordered_features(x, ordered_features, corr_threshold):
    # we'll need a correlation table for look-up purposes
    feature_corr_table = x.corr().abs()
    #print(feature_corr_table)

    selected_features = []
    deselected_features = []
    for i in range(len(ordered_features)):
        # select feature[i], if not in deselected list
        if i not in deselected_features:
            selected_features.append(i)

        if i < len(ordered_features) - 1:
            for j in range(i + 1, len(ordered_features)):
                if feature_corr_table[ordered_features[j]][ordered_features[i]] > corr_thres
                    deselected_features.append(j)

    # print(selected_features)
    selected_features = [ordered_features[k] for k in selected_features]
    # print(selected_features)

    return selected_features

def fit_and_eval(model, x_scaled, y_scaled, y, scaler):
    # fit model
    model.fit(x_scaled, y_scaled)

    # get y_pred for residual computes
    y_pred_scaled = model.predict(x_scaled)
    y_pred = scaler.inverse_transform(y_pred_scaled)
```

```
    # need y_np for print_metrics helper
    #   reshape into column vector
    y_np = y.to_numpy().reshape(-1, 1)

    # determine number of params
    num_params = x_scaled.shape[1]
    if model.intercept:
        num_params += 1

    # print coefficients
    print()
    print(model.coeffTable())

    # print metrics
    print()
    print_metrics(y_pred, y_np, num_params, x_scaled.shape[0])

# Let's start with a corrplot, similar to that which we
#   generated before, but only on the training datasets now.
x_train = load_x_train()
y_train = load_y_train()

# let's get a combined train dataframe
train = x_train.copy()
train['Appliances'] = y_train['Appliances']

# df.corr takes an optional callable, so let's use
#   the implementation of pearson's r that we developed
#   previously in the course
# https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.corr.html
corr_table = train.corr(
    method=corr
)

# locked color scale bounds, based on theoretical min and max
# https://seaborn.pydata.org/generated/seaborn.heatmap.html
sns.heatmap(
    corr_table,
    vmin=-1, vmax=1
)
plt.tight_layout() # fixes issue with image bounds cutting off labels
plt.savefig(f'{tmp_graphics_folder}{sep}corrplot-train')
plt.figure()

# let's take the abs of the corr_table and do it again
#   This should make it more obvious which features are
```

```
#   correlated with the target variable
sns.heatmap(
    np.abs(corr_table),
    vmin=0, vmax=1
)
plt.tight_layout()
plt.savefig(f'{tmp_graphics_folder}{sep}corrplot-abs-train')
plt.figure()

# Let's perform column-wise normalization on the features and y.
x_scaler = StandardScaler()
x_scaler.fit(x_train)
x_train_scaled = x_scaler.transform(x_train)
print('DEBUG: x_train_scaled', type(x_train_scaled), x_train_scaled.shape)
x_train_scaled = pd.DataFrame(data=x_train_scaled, index=x_train.index, columns=x_train.colu

y_scaler = StandardScaler()
y_scaler.fit(y_train)
y_train_scaled = y_scaler.transform(y_train)
print('DEBUG: y_train_scaled', type(y_train_scaled), y_train_scaled.shape)
y_train_scaled = pd.DataFrame(data=y_train_scaled, index=y_train.index, columns=y_train.colu

# Let's look at the F-Test scores
#   using f_regression function provided by sklearn to
#   run the F-Test on each of features.
f_res = f_regression(x_train_scaled, y_train_scaled)
ftests = zip(x_train.columns, f_res[0], f_res[1])

# feature selection, sort by p-value
ftests_sorted = sorted(ftests, key=lambda x: x[2])

print('features ordered by ftest significance')
for (col, ftest, pvalue) in ftests_sorted:
    print(f'\t{ftest}\t{pvalue}\t{col}')

# F-test null hypothesis states that the model with no
#   independent variables fits the data as well as your model
# Alternative Hypothesis, is that your model fit's the data
#   better than a model with no independent variables.
print('features to drop:')
drop_list = []
for (col, ftest, pvalue) in ftests_sorted:
    if (pvalue > 0.05): # 95% confidence
        print(f'\t{ftest}\t{pvalue}\t{col}')
        drop_list.append(col)
```

```
# Whats left are features which give us a 95%
#  confidence of them improving on the fit from
#  model with no independent variables
x_train_scaled = x_train_scaled.drop(drop_list, axis=1)
print('remaining features:', x_train_scaled.columns)

# Now, let's use corrplot, and the t-test to
#   ensure the we have no multicollinearity.
corr_table = x_train_scaled.corr(
    method=corr
)

sns.heatmap(
    corr_table,
    vmin=-1, vmax=1
)
plt.tight_layout()
plt.savefig(f'{tmp_graphics_folder}{sep}corrplot-ftest-filtered')
plt.figure()

# Let's use the t-test and correlation to determine
#   which regressands are significantly related
ttest_table = x_train_scaled.corr(
    method= lambda x, y: cor_and_ttest(x, y)[2]
)

# So... this actually works... which is really cool :)
#   Let's tailor the heatmap color scale a bit
sns.heatmap(
    ttest_table,
    vmin=0.05, vmax=1
)
plt.tight_layout()
plt.savefig(f'{tmp_graphics_folder}{sep}ttestplot-ftest-filtered')
plt.figure()

# Based on that it seems reasonable to say that pretty much
#   all the correlation coefficients being generated
#   are statistically significant.
# In hindsight... based on what the ttest of a correlation
#   coefficient should actually be asking... this totally
#   makes sense given the number of samples we have.
# So, going back to the original corrplot, let's worry
#   more about the correlation magnitude

# Redo F-test for new ordered list of remaining features
```

```
f_res = f_regression(x_train_scaled, y_train_scaled)
ftests = zip(x_train_scaled.columns, f_res[0], f_res[1])

# feature selection, sort by p-value
ftests_sorted = sorted(ftests, key=lambda x: x[2])

# only need the features in order
ftests_sorted = [col for col, _, _ in ftests_sorted]

print('remaining features in order:', ftests_sorted)

# let's decide on a correlation threshold for these...
#   to do that I'd like a plot of threhold vs number of features in the model.
# we'll need a function for grabbing the features based on the threshold...
thresholds = np.linspace(0, 1.0, num=100)
feature_counts = [len(filter_ordered_features(x_train_scaled, ftests_sorted, threshold)) for
plt.plot(thresholds, feature_counts)
plt.title('Feature Count vs. Threshold')
plt.xlabel('Threshold')
plt.ylabel('Feature Count')
plt.savefig(f'{tmp_graphics_folder}{sep}feature-count-vs-threshold')
plt.figure()

# Based on the chart, I'm thinking we start with a threshold around .8
#   correlation between features, to avoid the worst multicollinearity
#   offenders amongst the features. We can then incrementally drop
#   features that aren't highly signficant.
features1 = filter_ordered_features(x_train_scaled, ftests_sorted, 0.8)
print('attempting feature-set:', features1)
model = LinRegModel()
x_train1 = x_train_scaled[features1]
print('DEBUG: x_train1.shape', x_train1.shape)

fit_and_eval(model, x_train1, y_train_scaled, y_train, y_scaler)

# Let's remove the intercept as, it's very insignificant.
model = LinRegModel(intercept=False)
print('removing intercept:')

fit_and_eval(model, x_train1, y_train_scaled, y_train, y_scaler)

# remove T6
x_train1 = x_train1.drop('T6', axis=1)
print('removed T6:')

fit_and_eval(model, x_train1, y_train_scaled, y_train, y_scaler)
```

```
# remove Windspeed
x_train1 = x_train1.drop('Windspeed', axis=1)
print('removed Windspeed')

fit_and_eval(model, x_train1, y_train_scaled, y_train, y_scaler)

# remove Press_mm_hg
x_train1 = x_train1.drop('Press_mm_hg', axis=1)
print('removed Press_mm_hg')

fit_and_eval(model, x_train1, y_train_scaled, y_train, y_scaler)

# remove T8
x_train1 = x_train1.drop('T8', axis=1)

fit_and_eval(model, x_train1, y_train_scaled, y_train, y_scaler)

# remove RH_6
x_train1 = x_train1.drop('RH_6', axis=1)

fit_and_eval(model, x_train1, y_train_scaled, y_train, y_scaler)

# remove RH_out
x_train1 = x_train1.drop('RH_out', axis=1)

fit_and_eval(model, x_train1, y_train_scaled, y_train, y_scaler)

# remove NSM
x_train1 = x_train1.drop('NSM', axis=1)

fit_and_eval(model, x_train1, y_train_scaled, y_train, y_scaler)

# remove T4
x_train1 = x_train1.drop('T4', axis=1)

fit_and_eval(model, x_train1, y_train_scaled, y_train, y_scaler)

# We'll utilize this model, but out of curiosity, the published
#  article/paper associated with the dataset, used a recursive
#  feature elimination (RFE) algorithm provided by the caret
#  package in R. In python, A similar algorithm, is available
#  from the sklearn package.
# Let's use the features identified in the paper, as those to use
#  for a linear regression model, and see if our outcome matches.
features_paper=[
```

```
    'NSM', 'lights', 'Press_mm_hg', 'RH_5', 'T3', 'RH_3'
]

x_train_paper = x_train[features_paper]

x_scaler_paper = StandardScaler()
x_scaler_paper.fit(x_train_paper)
x_tmp = x_scaler_paper.transform(x_train_paper)
print('DEBUG: x_train_scaled', type(x_train_paper), x_train_paper.shape)
x_train_paper = pd.DataFrame(data=x_tmp, index=x_train_paper.index, columns=x_train_paper.co

model_paper = LinRegModel(intercept=False)

fit_and_eval(model_paper, x_train_paper, y_train_scaled, y_train, y_scaler)

# This behaves similarly to to the way described in paper
#   I intend to evaluate both models against the test set.
```

A.6 regression_model_evaluation.py

```
# This file will recreate the final models built from
#   feature_selection_regression, and evaluate them on
#   the test dataset.

# Core Python Dependencies
from os.path import sep # attempts to maintain OS portability

# Import Third-part Libraries
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from sklearn.preprocessing import StandardScaler

# Import things developed by me either
#   throughout the coursework, or
#   specifically for the term project
#   (provided in additional python files)
from utils.data import load_y_train, load_x_train, load_y_test, load_x_test
from utils.regression import LinRegModel
import utils.stats as stats
from utils.conf import tmp_graphics_folder

# Model based on filtering by F-test, then incrementally removing
#  insignificant features based on the t-tests
```

```
features1 = ['lights', 'T2', 'RH_8', 'T3', 'RH_1', 'T5']
# Model based on the papers choices (uses all the features)
features2 = [
    'NSM', 'lights', 'Press_mm_hg', 'RH_5', 'T3', 'RH_3'
]

# Let's setup StandardScaler Transforms for normalizing the data
#   we'll use the same one transform for the dependent variables
#   but we'll need two different scalers for the inputs, as
#   we have two different feature lists.
x_scaler1 = StandardScaler()
x_scaler2 = StandardScaler()
y_scaler = StandardScaler()

# Let's get the data setup
x_train = load_x_train()
y_train = load_y_train()
x_test = load_x_test()
y_test = load_y_test()

# apply feature selection
x_train1 = x_train[features1]
x_test1 = x_test[features1]
x_train2 = x_train[features2]
x_test2 = x_test[features2]

# fit scalers using only the test data
x_scaler1.fit(x_train1)
x_scaler2.fit(x_train2)
y_scaler.fit(y_train)

# apply scalers
#   decided to setup a little helper function for this
def _apply_scaler(scaler, df):
    return pd.DataFrame(
        scaler.transform(df),
        index=df.index,
        columns=df.columns
    )

x_train1 = _apply_scaler(
    x_scaler1,
    x_train1
)
x_train2 = _apply_scaler(
    x_scaler2,
```

```
    x_train2
)
y_train_scaled = _apply_scaler(
    y_scaler,
    y_train
)

# Setup and fit Linear Regression Models
#  in keeping with our finding during feature selection
#  we'll have no interecept values.
model1 = LinRegModel(intercept=False)
model2 = LinRegModel(intercept=False)
model1.fit(x_train1, y_train_scaled)
model2.fit(x_train2, y_train_scaled)

# Let's print the model summary results
print('MODEL 1 RESULTS (on transformed data):')
model1.print_summary()

print('MODEL 2 RESULTS (on transformed data):')
model2.print_summary()

# OK, let's transform the test data
x_test1 = _apply_scaler(
    x_scaler1,
    x_test1
)

x_test2 = _apply_scaler(
    x_scaler2,
    x_test2
)

# and get our predicted y data
#  y_pred1 and y_pred2 are numpy arrays,
#  note dataframes
y_pred1 = model1.predict(x_test1)
y_pred2 = model2.predict(x_test2)

# now we need to invert the scaler transform
y_pred1 = y_scaler.inverse_transform(y_pred1)
y_pred2 = y_scaler.inverse_transform(y_pred2)

# let's convert the y_test actual data to numpy arrays
#   column vectors for everything!!
y_test = y_test.to_numpy().reshape(-1, 1)
```

```
# evaluate models
#  another helper function
print()
print('Model 1 Test Set Regression Analysis:')
residuals1 = stats.print_metrics(y_test, y_pred1, x_test1.shape[1], x_train1.shape[0])
stats.acf_plot(residuals1, 'Model 1 Residuals', 20)
plt.savefig(f'{tmp_graphics_folder}{sep}regression-model-1-residuals-acf')
plt.figure()

print()
print('Model 2 Test Set Regression Analysis:')
residuals2 = stats.print_metrics(y_test, y_pred2, x_test2.shape[1], x_train2.shape[0])
stats.acf_plot(residuals2, 'Model 2 Residuals', 20)
plt.savefig(f'{tmp_graphics_folder}{sep}regression-model-2-residuals-acf')
plt.figure()

y_train = y_train.to_numpy().reshape(-1, 1)
y_test = y_test.reshape(-1, 1)
y_actuals = np.append(y_train, y_test)
plt.plot(y_actuals)
xs = list(range(y_train.shape[0], y_actuals.shape[0]))
plt.plot(xs, y_pred1)
plt.title('Regression Model 1 Prediction vs Time')
plt.legend(['Actual', 'Predicted'])
plt.savefig(f'{tmp_graphics_folder}{sep}final-regression-model-1-actual-pred-vs-time')
plt.figure()

plt.plot(y_actuals)
plt.plot(xs, y_pred2)
plt.title('Regression Model 2 Prediction vs Time')
plt.legend(['Actual', 'Predicted'])
plt.savefig(f'{tmp_graphics_folder}{sep}final-regression-model-2-actual-pred-vs-time')
plt.figure()
```

A.7 arma_model_identification.py

```
# This file responsible for generating GPAC code
#   and exploring various ARMA models for consideration

# This file generates the content for the Introduction
#   and performs the train-test-split
# Core Python Dependencies
from os.path import sep # attempts to maintain OS portability
```

```
# Import Third-part Libraries
import matplotlib.pyplot as plt
from statsmodels.tsa.stattools import adfuller

# Import things developed by me either
#   throughout the coursework, or
#   specifically for the term project
#   (provided in additional python files)
from utils.data import load_y_train
from utils.gpac import gpac
import utils.visualizations as viz
from utils.conf import tmp_graphics_folder
from utils import arma
from utils import stats

# load x and y train datasets
y_train = load_y_train()

# setup our numpy array/matrix content
#  need dependent variable as column vector
y = y_train.to_numpy().reshape(-1, 1)

# we'll arbitrarily start with 8 for now.
table = gpac(
    y,
    8, 8
)

# and plot the table
viz.plot_GPAC(table, 'y-train')
plt.savefig(f'{tmp_graphics_folder}{sep}gpac-y-train')
plt.figure()

# acf of training sig
viz.acf_plot(y, 'y-train', 20)
plt.savefig(f'{tmp_graphics_folder}{sep}acf-20-y-train')

# I have my doubts, but the most obvious ARMA Identification from
#   the GPAC appears to be an ARMA(1, 0) model.
#   Let's give that a try.
# Failed to converge on initial attempt, let's bump μ_max by 5
#   orders of magnitude and try again.
# Still failing to convert, let's attempt shrining the gradient/hessian
#   approximation step size
# WOW.. ok... uping μ_max by another 5 orders of magnitude.
# Still not converging.. let's increase convergence threshold
```

```
# We've gone to far... Let's increase the number of iterations
#   and reset to all the defaults on everything else.
# Luckily the iterations are happening rather quickly.
#   but i'm betting the signal is simply NOT a well identified ARMA model,
#   if we can't achieve convergence within 200 iterations.
# Actually looking at model results, and continuing to experiment... I
#   believe our criteria for convergence is simply to small
# Yep, LMA converges in 1 iteration if ϵ is 1, and doesn't otherwise.
#   I suspect that the mean/variance of the errors is high enough it
#   can't converg to the same delta of SSE we'd expected for prev.
#   datasets covered in class.
print()
print('ARMA(1,0):')
res = arma.lm(y, 1, 0, ϵ = 1, verbose=False)
res.summary()

# Since it'll converge with less restrictions...
#   is it useful. Actuall... not bad based on this graph.
res.plot_pred_vs_true('arma-1-0-train', y)
plt.savefig(f'{tmp_graphics_folder}{sep}arma-1-0-train')
plt.figure()

# So, despite it failing to converge, we did have an interesting outcome.
#  a_1 has p-value of 0.. i'm thinking that may have something to do with
#  the large number of samples.
# Let's go back to the GPAC table, and see if a more complicated model
#  can be identified if we lower our expectation a bit.
# From the GPAC, I'm interesting in trying ARMA(1, 1), ARMA(1, 2), ARMA(2, 1), and an ARMA(2
print()
print('ARMA(1,1):')

res = arma.lm(y, 1, 1, verbose=False)
res.summary()
res.plot_pred_vs_true('arma-1-1-train', y)
plt.savefig(f'{tmp_graphics_folder}{sep}arma-1-1-train')
plt.figure()

print()
print('ARMA(1,2):')

res = arma.lm(y, 1, 2, verbose=False)
res.summary()
res.plot_pred_vs_true('arma-1-2-train', y)
plt.savefig(f'{tmp_graphics_folder}{sep}arma-1-2-train')
plt.figure()
```

```
print()
print('ARMA(2,1):')

res = arma.lm(y, 2, 1, verbose=False)
res.summary()
res.plot_pred_vs_true('arma-2-1-train', y)
plt.savefig(f'{tmp_graphics_folder}{sep}arma-2-1-train')
plt.figure()

print()
print('ARMA(2,2):')

res = arma.lm(y, 2, 2, verbose=False)
res.summary()
res.plot_pred_vs_true('arma-2-2-train', y)
plt.savefig(f'{tmp_graphics_folder}{sep}arma-2-2-train')
plt.figure()

# OK.. so highly significant parameters for every model..
#  I'm thinking we'd get massively better results with something
#  similar to the normalization or logarithm transform we've used in the
#  labs and homeworks before.
# Higher MA orders appears to add cyclic behavior we don't want.
# AR seems more useful, BUT, isn't doing a good job of capturing
#  the skewed nature of the signal. I'm thinking to attempt
#  some transforms, as we did with homework9.
# log and norm seem fairly resonably behaved
# I attempted to take advantage of seasonality (daily) that
#   we identified in the decomp using the lagged difference transform
#   but did not have much luck. issue is the lagged difference inverse
#   manages to
y_trans, inverter = arma.compose_transform(
    arma.logarithmic_transform,
    arma.normalization_transform,
)(y)
print('DEBUG: y_trans.shape', y_trans.shape)

# But.. What did that do, let's do some quick plots
#   and check we're still stationary
plt.plot(y_trans)
plt.title('y training after Transform')
plt.xlabel('Time Steps')
plt.ylabel('Tranformed y')
plt.savefig(f'{tmp_graphics_folder}{sep}y-training-trans-vs-time')
plt.figure()
```

```
plt.hist(y_trans)
plt.title('y training hist after Transform')
plt.savefig(f'{tmp_graphics_folder}{sep}y-training-trans-hist')
plt.figure()

print('ADFTest for y transform:', adfuller(y_trans.reshape(-1)))

viz.acf_plot(y_trans, 'y-train-trans', 20)
plt.savefig(f'{tmp_graphics_folder}{sep}y-train-trans-acf')
plt.figure()

trans_table = gpac(
    y_trans, 8, 8
)

viz.plot_GPAC(trans_table, 'y-train-trans')
plt.savefig(f'{tmp_graphics_folder}{sep}y-train-trans-gpac')
plt.figure()

# Cool, based on the new GPAC
print()
ar_order = 4
ma_order = 3
print(f'ARMA({ar_order},{ma_order}):')

res = arma.lm(y_trans, ar_order, ma_order, verbose=False)
res.summary()
res.plot_pred_vs_true(f'arma-{ar_order}-{ma_order}-train-trans', y_trans)
plt.savefig(f'{tmp_graphics_folder}{sep}arma-{ar_order}-{ma_order}-train-trans')
plt.figure()

# undo transform and check RMSE (against train data)
_, y_pred = res.dlsim(
    num_samples=y_trans.shape[0],
    #  provide first (ar_order + 1) values of data to base
    #    prediction on, as this is a recursive prediction
    #    mechanism, it does need a starting point
    y0=y_trans[:ar_order+1]
)
# making this a column vector makes print_metrics A LOT faster
y_pred = y_pred.reshape(-1, 1)
y_pred = inverter(y_pred)

plt.plot(y)
plt.plot(y_pred)
plt.legend(['Actual', 'Predicted'])
```

```
plt.savefig(f'{tmp_graphics_folder}{sep}arma-{ar_order}-{ma_order}-actual-pred-vs-time')
plt.figure()

stats.print_metrics(y_pred, y, ar_order+ma_order, y.shape[0])
```

A.8 arma_model_evaluation.py

```
# let's evaluate the chosen arma model on the test set

# This file generates the content for the Introduction
#   and performs the train-test-split
# Core Python Dependencies
from os.path import sep # attempts to maintain OS portability

# Import Third-part Libraries
import matplotlib.pyplot as plt
import numpy as np

# Import things developed by me either
#   throughout the coursework, or
#   specifically for the term project
#   (provided in additional python files)
from utils.data import load_y_train, load_y_test
from utils.conf import tmp_graphics_folder
from utils import arma
from utils import stats

# load the data
y_test = load_y_test()
y_train = load_y_train()

# convert to column vectors
y_test = y_test.to_numpy().reshape(-1, 1)
y_train = y_train.to_numpy().reshape(-1, 1)

# setup transform
transform = arma.compose_transform(
    arma.logarithmic_transform,
    arma.normalization_transform
)

# transform data
#  NOTE: The transform is being created based on the training
```

```
#  set, as the nomalization transform saves mean and var
#  for use in the inverter.
y_train_trans, y_train_inverter = transform(y_train)

# train ARMA(4,3) model
model = arma.lm(y_train_trans, 4, 3)

model.summary()

# predict using last 4 values of y_train_trans as initial
#   values, let's see how close we get to the y_test data
_, y_test_pred = model.dlsim(
    num_samples=y_test.shape[0] + 4, # as first 4 values are coming from y_train
    #  provide first (ar_order + 1) values of data to base
    #    prediction on, as this is a recursive prediction
    #    mechanism, it does need a starting point
    #    (last 4 values in training set)
    y0=y_train_trans[y_train_trans.shape[0] - 4 - 1:, 0]
)
# reshape, and drop vals from y_train

y_test_pred = y_test_pred.reshape(-1, 1)[4:, 0]

# invert transform
#   NOTE: again... so much faster using column vectors
y_test_pred = y_train_inverter(y_test_pred).reshape(-1, 1)

# plot results
print('INFO: y_test.shape:', y_test.shape)
y_actual = np.append(y_train, y_test)
plt.plot(y_actual)
xs = list(range(y_train.shape[0], y_actual.shape[0]))
plt.plot(xs, y_test_pred)
plt.savefig(f'{tmp_graphics_folder}{sep}final-arma-4-3-test')
plt.figure()

residuals = stats.print_metrics(y_test_pred, y_test, 7, y_train.shape[0])
stats.acf_plot(residuals, 'ARMA Model Residuals', 20)
plt.savefig(f'{tmp_graphics_folder}{sep}final-arma-4-3-test-residuals-acf')
plt.figure()

# train ARMA(1, 0) model
model = arma.lm(y_train_trans, 1, 0)

model.summary()
```

```
# predict using last 4 values of y_train_trans as initial
#   values, let's see how close we get to the y_test data
_, y_test_pred = model.dlsim(
    num_samples=y_test.shape[0] + 1, # as first values are coming from y_train
    #  provide first (ar_order + 1) values of data to base
    #    prediction on, as this is a recursive prediction
    #    mechanism, it does need a starting point
    #    (last 4 values in training set)
    y0=y_train_trans[y_train_trans.shape[0] - 1:, 0]
)
# reshape, and drop vals from y_train
y_test_pred = y_test_pred.reshape(-1, 1)[1:, 0]

# invert transform
#   NOTE: again... so much faster using column vectors
y_test_pred = y_train_inverter(y_test_pred).reshape(-1, 1)

# plot results
print('INFO: y_test.shape:', y_test.shape)
y_actual = np.append(y_train, y_test)
plt.plot(y_actual)
xs = list(range(y_train.shape[0], y_actual.shape[0]))
plt.plot(xs, y_test_pred)
plt.savefig(f'{tmp_graphics_folder}{sep}final-arma-1-0-test')
plt.figure()

residuals = stats.print_metrics(y_test_pred, y_test, 1, y_train.shape[0])
stats.acf_plot(residuals, 'ARMA Model Residuals', 20)
plt.savefig(f'{tmp_graphics_folder}{sep}final-arma-1-0-test-residuals-acf')
plt.figure()
```

A.9 test_linreg.py

```
# used this to debug some issues with linreg model
#  relating to Confidence interval/covariance matrix compute
import pandas as pd
import numpy as np
import statsmodels.api as sm
from utils.regression import LinRegModel

# c is target variable
#  1a + 1b + 1 = c
num_samples = 1000
```

```
a = np.random.normal(size=1000)
b = np.random.normal(size=1000)
B = np.array([2, .5, .1]).reshape(-1, 1)
# noise = np.random.normal(scale=0, size=1000).reshape(-1, 1)

X = np.zeros((1000, 3))
X[:, 0] = 1
X[:, 1] = a
X[:, 2] = b
Y = np.dot(X, B) # + noise

data = pd.DataFrame({
    'a': a,
    'b': b,
    'c': Y.reshape(-1)
})

model = LinRegModel(intercept=True)
model.fit(data[['a', 'b']], data['c'])
model.print_summary()
print(model.covTable())

# ok.. now I'm wondering if my regression model is broke, let's confirm i get similar
#  behaviour from sklearn's linearregression setup the same way.
model = sm.OLS(data['c'], sm.add_constant(data[['a', 'b']]))
res = model.fit()
# print(res.cov_params())
print(res.summary())
```

A.10 Utils Package

A.10.1 conf.py

```
# This file to provide general configuration information and helper
#   routines, used throughout the term-project code.

# Core Python Dependencies
from pathlib import Path # used to setup folders
from os.path import sep

# folder config/setup
# This file is provided, but can be downloaded from
#   the github project associated with the original
#   research paper.
```

```
#   https://github.com/LuisM78/Appliances-energy-prediction-data/blob/master/energydata_comp
data_source_file=f'data{sep}energydata_complete.csv'
tmp_graphics_folder='tmp_graphics'
tmp_data_folder='tmp_data'

# function to initialize folders as needed
def init_tmp_folders():
    Path(tmp_graphics_folder).mkdir(parents=True, exist_ok=True)
```

A.10.2 data.py

```
# This file is the landing place for a number of data
#   specific settings and utilities used through
#   the term-project code.

# Core Python Dependencies
from os.path import sep

# Third Party Imports
import pandas as pd

# Import things developed by me either
#   throughout the coursework, or
#   specifically for the term project
#   (provided in additional python files)
from .conf import tmp_data_folder, data_source_file

# custom dateparser
dateparser = lambda x: pd.datetime.strptime(x, "%Y-%m-%d %H:%M:%S")

# assumes dataset_info_split.py has been ran
def __common_load_df(filename):
    df = pd.read_csv(filename, parse_dates=['date'], date_parser=dateparser)
    df = df.set_index('date')
    return df

def load_original_data():
    return __common_load_df(data_source_file)

def load_y_test():
```

```
    return __common_load_df(f'{tmp_data_folder}{sep}y_test.csv')

def load_y_train():
    return __common_load_df(f'{tmp_data_folder}{sep}y_train.csv')

def load_x_test():
    return __common_load_df(f'{tmp_data_folder}{sep}x_test.csv')

def load_x_train():
    return __common_load_df(f'{tmp_data_folder}{sep}x_train.csv')

# used in decomposition.py and holt-winter.py
#   seemed a reasonable place for it
__data_freq = 10 # minutes

# minutes per week over data freq in minutes
weekly_freq = int(60*24*7/__data_freq)

# minutes per day over data freq in minutes
daily_freq = int(60*24/__data_freq)

# minutes per hour over data freq in minutes
hourly_freq = int(60/__data_freq)
```

A.10.3 arma.py

```
# This file defines our arma models and utility functions

# Core Python Imports
from os.path import sep # attempts to maintain OS portability

# Import Third-part Libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.stats import distributions

# Import things developed by me either
#   throughout the coursework, or
#   specifically for the term project
#   (provided in additional python files)
from . import stats

# dlsim function mimicing api for statsmodel dlsim,
#   except that we always assume 1
```

```
#   TODO: make this more column vector based?
def dlsim(sys, e, y0=None):
    # drop first entry from sys[1] to get a, give it dimensions
    a = np.array(sys[1])[1:].reshape(1, -1)
    # sys[0] is b
    b = np.array(sys[0]).reshape(1, -1)

    # wrapper for getting initialization right with coef_vec usage
    def coef_vec(v, i):
        if i < 0:
            return np.empty((0, v.shape[1]))
        if  i < v.shape[1]:
            return v[:, :i + 1]
        else:
            return v

    # wrapper for getting initialization right with e and y "vals" vectors
    def vals_vec(vals, i, order):
        if i < 0:
            return np.empty((0, 1))
        if i < order:
            return vals[i::-1]
        else:
            return vals[i:i - order:-1]

    e = np.array(e).reshape(-1, 1)

    # normal case
    if y0 is None:
        y = np.zeros(e.shape)
        i_start = 0
    else:
        # when starting y values have been provided.
        y0 = y0.reshape(-1, 1)
        y = np.append(y0, np.zeros((e.shape[0] - y0.shape[0], 1)), axis=0)
        print('DEBUG: y shape', y.shape)
        i_start = y0.shape[0]

    for i in range(i_start, e.shape[0]):
        # get a and b, as function of i
        b_temp = coef_vec(b, i)
        a_temp = coef_vec(a, i - 1)

        # get e and y, as function of i
        e_temp = vals_vec(e, i, b.shape[1])
        y_temp = vals_vec(y, i - 1, a.shape[1])
```

```
        assert(e_temp.shape[0] > 0)

        if y_temp.shape[0] > 0:
            ar_comp = np.dot(
                a_temp,
                y_temp
            )
            ma_comp = np.dot(
                b_temp,
                e_temp
            )

            y[i, 0] = ma_comp - ar_comp
        else:
            y[i, 0] = np.dot(
                b_temp,
                e_temp
            )

    return np.array(range(1, e.shape[0])), y.reshape(1, -1)

# generate normally distributed white noise
#   mu is desired mean
#   sigma2 is desired variance
#   samples is number of samples
#   random_seed (optional) is the random seed to use.
def generate_e(mu, sigma2, samples, random_seed=42):
    np.random.seed(random_seed)
    return np.random.randn(samples)*np.sqrt(sigma2) + mu


# Added an LMAResult Class (inspired by statsmodel/scipy/sklearn apis)
#  to simplify model result aggregation/interpretation
class LMAResult:
    def __init__(self, ar_coefficients, ma_coefficients,
                 cov_θ, var_e, sse_accum, num_samples, e):
        self.ar_coefficients = ar_coefficients
        self.ma_coefficients = ma_coefficients
        self.cov_θ = cov_θ
        self.var_e = var_e
        self.sse_accum = sse_accum
        self.num_samples = num_samples
        self.e = e
        self.ar_order = len(self.ar_coefficients[1:])
        self.ma_order = len(self.ma_coefficients[1:])
```

```
# calulcated ci, based on slides, and web resources
#   https://kite.com/python/examples/702/scipy-compute-a-confidence-interval-from-a-data
#   https://stats.stackexchange.com/questions/241449/matrix-and-regression-model
def _ci_calc(self, val, se, confidence):
    h = distributions.norm.ppf((1 - (1-confidence)/2))
    low = val - h*se
    high = val + h*se
    return low, high

# prints header for Table
def _printCIHeader(self):
    print('\tlabel\tcoeff\tlower\tupper\tt-test\tp-value')

# prints coefficients and their low/high intervals
def _printCIs(self, label, coefs, ses, confidence):
    # verify sanity
    assert(len(coefs) == len(ses))

    # build labels list
    labels = [f'{label}_{i}' for i in range(1, len(coefs) + 1)]

    # print line per coef/CI
    for label, coef, se in zip(labels, coefs, ses):
        low, high = self._ci_calc(coef, se, confidence=confidence)
        t = stats.ttest_1sample(coef, se * np.sqrt(self.num_samples), self.num_samples)
        p = stats.ttest_pvalue(t, self.num_samples - self.cov_θ.shape[0])
        print('\t%s\t%.4g\t%.4g\t%.4g\t%.4g\t%.4g' % (label, coef, low, high, t, p))

# prints table of coefficients with Confidence Intervals
def printCoeffsTable(self, confidence=0.95):
    # all standard errors
    ses = np.sqrt(
        self.cov_θ[np.diag_indices_from(self.cov_θ)]
    )

    self._printCIHeader()

    # print('ar_coefs and confidence intervals:')
    self._printCIs(
        label='a',
        coefs=self.ar_coefficients[1:],
        ses=ses[:self.ar_order],
        confidence=confidence
    )
```

```
    # print('ma_coefs and confidence intervals:')
    self._printCIs(
        label='b',
        coefs=self.ma_coefficients[1:],
        ses=ses[self.ar_order:],
        confidence=confidence
    )

# returns estimated ARMA in transform function form
#   can be directly passed into dlsim as system tuple
def _system(self):
    def fix_coeffs(coeffs, order):
        return np.append(coeffs, np.zeros(order + 1 - len(coeffs)))

    max_order = max(self.ar_order, self.ma_order)

    return (
        fix_coeffs(self.ma_coefficients, max_order),
        fix_coeffs(self.ar_coefficients, max_order),
        1
    )

# returns generated noise based on summary stats from error
#   collected by LMAResult on creation
def _generate_e(self, num_samples):
    return generate_e(
        np.mean(self.e),
        np.var(self.e),
        num_samples
    )

# returns simulated ARMA based on result coefficients
#   and error characterists
def dlsim(self, num_samples, y0=None):
    return dlsim(
        self._system(),
        self._generate_e(num_samples),
        y0=y0
    )

# helper for taking care of 2 through 4
def summary(self):
    # 1-ish
    # print('predicted transfer function (as dlsim system):')
    # print(self.system())
```

```
            # 2 - confidence intervals
            print('ARMA coefficients:')
            self.printCoeffsTable()

            # 3 - cov matrix
            # updated to pretty-print using numpy
            coef_labels = [f'a_{i}' for i in range(1, self.ar_order + 1)]
            coef_labels.extend([f'b_{i}' for i in range(1, self.ma_order + 1)])
            cov_df = pd.DataFrame(self.cov_θ, index=coef_labels, columns=coef_labels)
            print('covariance matrix:', cov_df, sep='\n')

            # 4 - variance of error
            # unwapping dimensions around this... it's always scalar though
            print('estimated variance of error', self.var_e[0][0])
            print('estimated mean of error', np.mean(self.e))

        # sse vs iterations
        def plot_sse_vs_iter(self):
            plt.plot(range(1, len(self.sse_accum) + 1), self.sse_accum)
            plt.title('sse vs iteration')
            plt.xlabel('iterations')
            plt.ylabel('sse')

        # helper for taking care of true vs pred plot
        def plot_pred_vs_true(self, label, original_data):
            _, pred = self.dlsim(
                num_samples=len(original_data),
                y0=original_data[:self.ar_order + 1])
            pred = pred.reshape(-1)

            # pyplot stuffs
            plt.plot(original_data)
            plt.plot(pred)
            plt.title(f'True and Pred. vs Time Step\n{label}')
            plt.xlabel('Time Step')
            plt.ylabel(label)
            plt.legend(['True', 'Pred.'])

# Welp... this got pretty long
#  returns ar_coefs, ma_coefs
#  Note: heavy use of optional params for easy tuning/adjustments
#  Note: generally see convergence well within 50 iterations
#  TODO: Generalize/sepparate from ARMA application
#    sounds like an interesting thing to do, but i'm not
#    sure I'll have time.
def lm(sig, ar_order, ma_order,
```

```
δ = 1e-6, # jacobian matrix approximation (dθ)
μ = 0.01, # Gaus-Newton vs Gradient Descent weighting
μ_max = 1e10, # maximum
ϵ = 1e-4, # convergence threshold
max_iterations = 50,
verbose=True):
# hey! python supports symbols... let's use them... it'll match the slides...
# N is number of samples
N = sig.shape[0]

# confirm sig is a column vector
assert(sig.shape[0] == N and sig.shape[1] == 1)

# Step 0
#   setup θ and dimension properly as column vector
θ = np.zeros((ar_order + ma_order, 1)).reshape(-1, 1)
n = θ.shape[0]

# this turned out to be very similar to dlsim... but reversed...
#   probably could use common code in some way... mey... maybe another day
#   returns column vector of Nx1 for [e(1), e(2), ... e(N-1)].T
def _e(θ):
    # Note: uses sig, ar_order, and ma_order 'implicitly'
    # Note: unlike dlsim, this time decided to use column vectors consistently
    # unless i'm way mistaken... this should pretty much be the reverse of dlsim process
    #   pretty much working backways to get the 'estimated' noise array, and then minimi
    #   sum-squared noise array... kinda makes sense.
    # so...
    #   y(t) + a_1y(t - 1) + ... + a_na y(t - na) = e(t) + b_1 e(t - 1) + ... + b_na y(t
    #   a.T * y = e(t) + b.T * e'
    #   e(t) = a.T * y - b.T * e'
    # where:
    #   a = [1, a_1, ..., a_na].T
    #   b = [b_1, ..., b_nb].T
    #   e' = [e(t-1), e(t-2), ... , e(t-nb)].T
    #   y = [y(t), y[t-1], ... , y[t-na]].T
    # ar and ma coefficient column vectors
    ar_coefficients = np.append(np.array([1]), θ[:ar_order]).reshape(-1, 1)
    ma_coefficients =  θ[ar_order:].reshape(-1, 1)

    # copied from dlsim above
    #   return column vec of correct size/order
    def coef_vec(v, i):
        if i < 0:
            return np.empty((0, 1))
        if  i < v.shape[0]:
```

```
        return v[:i + 1, :]
    else:
        return v

# copied from dlsim above
#   return column vec of correct size/order
def vals_vec(vals, i, order):
    # need reverse order, line up a1y(t-1), a2 y(t-2), etc..
    if i < 0:
        return np.empty((0, 1))
    if i < order:
        return vals[i::-1, :]
    else:
        return vals[i:i - order:-1, :]

e = np.zeros((N, 1))
for i in range(N):
    a = coef_vec(
        ar_coefficients,
        i,
    )

    y_temp = vals_vec(sig, i, ar_order + 1)

    # always looking one time-delay back noise wise, as we're
    #  returning e(t) per iteration... hence i-1
    b = coef_vec(
        ma_coefficients,
        i - 1
    )

    e_temp = vals_vec(e, i - 1, ma_order)

    assert(y_temp.shape[0] > 0)

    if (e_temp.shape[0] > 0):
        e[i, 0] = np.dot(
            a.T,
            y_temp
        ) - np.dot(
            b.T,
            e_temp
        )
    else:
        e[i, 0] = np.dot(
            a.T,
```

```
                y_temp
            )
    return np.nan_to_num(e)

def _sse(e):
    return np.dot(e.T, e)

# appears to approximate jacobian? really not sure
#   Note: uses δ
#   returns matrix with each column x_i as described in lecture slides
def _X(e, θ):
    # return column vector with ith θ updated
    #   to θ + δ
    def θ_new(i):
        tmp = θ.copy()
        tmp[i, 0] = tmp[i, 0] + δ
        return tmp

    X = np.zeros((N, n))
    for i in range(n):
        X[:, i] = ((e - _e(θ_new(i)))/δ).reshape(-1)
    return X

def step1(θ, e=None, sse=None):
    if e is None:
        e = _e(θ)
    if sse is None:
        sse = _sse(e)
    X = _X(e, θ)
    A = np.dot(X.T, X)
    g = np.dot(X.T, e)
    return e, sse, A, g

def step2(A, g, θ):
    delta_θ = np.dot(np.linalg.inv(
            A + μ*np.identity(n)
        ),
        g
    )

    θ_new = θ + delta_θ
    e_new = _e(θ_new)
    sse_new = _sse(e_new)
    if np.isnan(sse_new):
        sse_new[0, 0] = 1e8
    return delta_θ, θ_new, e_new, sse_new
```

```
# running initial step1 operation outside loop
e, sse, A, g = step1(θ)
num_iterations = None
cov_θ = None
var_e = None

sse_accum = []
# assumes initial θ is set
for i in range(max_iterations):
    sse_accum.append(sse[0,0])
    # Step 1 - after initialization ran at bottom of loop
    # Step 2
    #   Update step math and results
    #   aww... delta isn't valid variable name symbol... oh well
    delta_θ, θ_new, e_new, sse_new = step2(A, g, θ)

    # Step 3
    #   Update Logic
    if sse_new < sse:
        # indicates that we've converged... set vars and break from loop
        if np.linalg.norm(delta_θ) < ϵ:
            θ = θ_new
            e = e_new
            num_iterations = i + 1
            var_e = sse_new/(N-n)
            cov_θ = var_e*np.linalg.inv(A)
            break
        else:
            θ = θ_new
            μ = μ/10

    while sse_new >= sse:
        μ = μ*10
        if μ > μ_max:
            if verbose:
                print(f'INFO: μ has exceeded μ_max in iteration {i + 1}')
            # initially (for lab 9) i was off by literally 10 orders of
            #  magnitude for what an appropriate μ_max value was.
            #  with it set more reasonably, happiness abounds
            break
        _, θ_new, e_new, sse_new = step2(A, g, θ)

    # allows us to skip an e calculation
    θ = θ_new
    e, sse, A, g = step1(θ, e=e_new, sse=sse_new)
```

```
    ar_coefficients = np.append([1], θ[:ar_order]).reshape(-1)
    ma_coefficients =  np.append([1], θ[ar_order:]).reshape(-1)

    # check for convergence, and compute esimated standard_error and cov_θ
    if num_iterations is None or cov_θ is None or var_e is None:
        print('WARNING: LMA failed to converge')
        var_e = sse_new/(N-n)
        cov_θ = var_e*np.linalg.inv(A)
    else:
        print('INFO: lma converged in', num_iterations, 'iterations.')

    return LMAResult(
        ar_coefficients,
        ma_coefficients,
        cov_θ,
        var_e,
        sse_accum,
        sig.shape[0], # num samples,
        e
    )

# let's pull in all the transforms from hw 9
# identity transform
#   haha! the inverse of the identity transform... IS the identify transform
#   doh! not quite. the inverse isn't supposed to return a tuple
#   returns transformed data, and inverse transform function
def identity_transform(data):
    def _identity_inverse(pred):
        return pred
    return data, _identity_inverse

# difference transform
#   return transformed data, and inverse transform function
def lagged_difference_transform(lag):
    _lag = lag
    def difference_transform(data):
        c = data.tolist()[:_lag]

        # define difference inverse in scope where it will
        #   have a copy of c
        def difference_inverse(pred):
            print('inverting lagdif', _lag)
            inv_accum = c.copy()
            for i in range(len(pred)):
                inv_accum.append(inv_accum[i] + pred[i])
```

```
            return np.array(inv_accum)

        # perform transform
        accum = []
        for i in range(_lag, len(data)):
            accum.append(data[i] - data[i-_lag])
        return np.array(accum), difference_inverse
    return difference_transform

# difference transform
#   return transformed data, and inverse transform function
def difference_transform(data):
    c = data[0]

    # define difference inverse in scope where it will
    #   have a copy of c
    def difference_inverse(pred):
        inv_accum = [c]
        for i in range(len(pred)):
            inv_accum.append(inv_accum[i] + pred[i])
        return np.array(inv_accum)

    # perform transform
    accum = []
    for i in range(1, len(data)):
        accum.append(data[i] - data[i-1])
    return np.array(accum), difference_inverse

def logarithmic_transform(data):
    def logarithmic_inverse(pred):
        return np.exp(pred)

    return np.log(data), logarithmic_inverse

def normalization_transform(data):
    variance = np.var(data)
    mean = np.mean(data)

    def nomalization_inverse(pred):
        return pred * variance + mean

    return (data - mean)/variance, nomalization_inverse

# function for contructing multi-layer transform
#   welp... that's an eyesore.. but it works :)
#   returns transform function
```

```
def compose_transform(*transforms):
    inverse_accum = []
    def _compose_transform(data):
        def _compose_inverse(pred):
            for i in inverse_accum[::-1]:
                pred = i(pred)
            return pred

        for t in transforms:
            data, i = t(data)
            inverse_accum.append(i)
        return data, _compose_inverse
    return _compose_transform

# got some inspiration from this
#   https://fmwww.bc.edu/repec/bocode/t/transint.html
def reciprocal_transform(data):
    def reciprocal_inverse(pred):
        return np.reciprocal(pred)

    # inverse of the reciprocal tranform, is itself :shrug:
    return np.reciprocal(data), reciprocal_inverse

# why not...
#  https://en.wikipedia.org/wiki/Hyperbolic_functions
def htan_transform(data):
    def htan_inverse(pred):
        return np.arctanh(np.maximum(np.minimum(pred, 1-1e-100), 1e-100-1))

    return np.tanh(data), htan_inverse
```

A.10.4 gpac.py

```
# This file defines functions related to generating GPAC tables
#  for ARMA Model Identification

# Core Python Dependencies

# Import Third-part Libraries
import numpy as np

# Import things developed by me either
#   throughout the coursework, or
#   specifically for the term project
```

```
#   (provided in additional python files)
from .stats import autocorrelation_estimation
from .optimization import memoize

# function for defining an Ry function as described in class
def _R(y):
    y_mean = np.mean(y)

    def Ry(k):
        return autocorrelation_estimation(y, k, y_mean=y_mean)

    # add memoization, and vectorization to autocorrelation function for y
    return np.vectorize(memoize(Ry))

# B, denominator of cramer rule fraction
#   Eq. 2.3 from paper, uisng Ry instead of p
def _B(s, t, Ry):
    retVal = np.zeros((s, s))
    for i in range(s):
        retVal[:, i] = Ry(np.array(range(t - i, t + s - i)))
    return retVal

# A, numerator of cramer rule fraction
#   paragraph after Eq. 2.3 in paper, using Ry instead of p
def _A(s, t, Ry):
    retVal = _B(s, t, Ry)
    retVal[:, s - 1] = Ry(np.array(range(t + 1, t + s + 1)))
    return retVal

# kth autoregressive coefficient of ARMA(k, j)
#   partially based on paper:
#     https://www.smu.edu/-/media/Site/Dedman/Departments/Statistics/TechReports/TR-222.pdf
#   k is autoregression coefficient number, and autoregression order of ARMA model
#   j is moving average order of ARMA model
#   Ry is autocorrelation function for y
#       expected to be vectorized (see numpy vectorize)
#       performance improves when also memoized (see function above)
def _phi(k, j, Ry):
    # sanity check
    assert(k > 0)

    # equation 2.2 from paper, using Ry instead of p
    if k is 1:
        return Ry(j + 1)/Ry(j)
    else:
        return np.linalg.det(_A(k, j, Ry))/np.linalg.det(_B(k, j, Ry))
```

```
def gpac(sig, ar_order_max, ma_order_max):
    Ry = _R(sig)

    retVal = np.zeros((ma_order_max, ar_order_max))

    # let's go row in outer loop, that seems to be how
    #   numpy organizes the ndarray types, so should
    #   go a bit faster *shrug*
    for i in range(ma_order_max):
        for j in range(ar_order_max):
            retVal[i, j] = _phi(j + 1, i, Ry)
    return retVal
```

A.10.5 models.py

```
# This file contains code spcifically related to the holt-winter
#   and simpler related models types. Largely in support of the
#   holt-winter.py file for building holt-winter models of the
#   dataset. The contents of this file are based on code
#   developed for lab-4.

# The functions implemented here take a dataframe/series as input
#   and then return a forecasting function for that dataset.

# Core Python Dependencies

# Third-Party Dependencies
import numpy as np
import statsmodels.tsa.holtwinters as ets

# average method
#  train function returns the forecast function.
def avg_trainer(df):
    y_data = np.asarray(df)
    mean = np.mean(y_data)
    def avg_forecast(h):
        return np.repeat(mean, h)
    return avg_forecast

# naive method
#   train function returns the forecast function.
def naive_trainer(df):
    y_data = np.asarray(df)
```

```
    y_T = y_data[len(y_data)-1]
    def naive_forecast(h):
        return np.repeat(y_T, h)
    return naive_forecast

# drift method
#   train function returns the forecast function.
def drift_trainer(df):
    y_data = np.asarray(df)
    y_T = y_data[len(y_data)-1]
    y_0 = y_data[0]
    T = len(y_data)
    slope = (y_T - y_0)/(T - 1)
    def drift_forecast(h):
        hs = np.array(range(1, h + 1))
        return y_T + hs*slope
    return drift_forecast

# Simple Exponential Smoothing
#   train function returns the forecast function
#   we're assuming alpha is 0.5, and initial condition is 0 as instructed.
def ses_trainer(
    df,
    alpha=0.5,
    l_0=0):

    def l(t):
        _l = alpha * y_data[t] + (1 - alpha) * l_0
        for i in range(t):
            _l = alpha * y_data[i] + (1 - alpha) * _l
        return _l

    y_data = np.asarray(df)
    l_T = l(len(y_data)-1)

    def ses_forecast(h):
        return np.repeat(l_T, h)
    return ses_forecast

# Holt Linear Train
#   train function returns the predict function
#   we're assuming alpha is 0.5, and initial condition is 0 as instructed.
# Parameters below were selected by hand after numerous attempts.
def holt_linear_trainer(
    df,
    alpha = 0.5,
```

```
    beta = 0.0,
    l_0 = 0.0,
    b_0 = 0.0
):

    # Below is a for loop which performs the recursive operation iteratively
    def l_b(t):
        l = l_0
        b = b_0
        for i in range(t):
            l_new = np.nan_to_num(alpha * y_data[i] + (1 - alpha) * (l * b))
            b = np.nan_to_num(beta * (l_new - l) + (1 - beta) * b)
            l = l_new
            #   print(i, l, b)
        return l, b

    y_data = np.asarray(df)
    l_T, b_T = l_b  (len(y_data) - 1)

    def holt_linear_predict(h):
        hs = np.array(range(1, h + 1))
        return l_T + hs * b_T
    return holt_linear_predict

# proxying the statsmodel api to the api i came up with.
#   Ouch... This takes a long time to fit
def holt_winters_trainer_full(
    df,
    seasonal='additive',
    trend='additive',
    seasonal_periods=None):

    # optimized ExponentialSmoothing on y_data
    print('WARNING: This takes a long time. Recommend a coffee break.')
    model = ets.ExponentialSmoothing(
        df,
        seasonal=seasonal,
        trend=trend,
        seasonal_periods=seasonal_periods
    ).fit()
    def holt_winters_predict(h):
        return model.forecast(h)
    return holt_winters_predict
```

A.10.6 optimization.py

```
# This file the collection of optimization algorithms
#   and tools used throughout the course
# Core Python Dependencies

# Import Third-part Libraries
import numpy as np


def memoize(func):
    cache = {}
    def cache_func(x):
        if not x in cache:
            cache[x] = func(x)
        return cache[x]
    return cache_func

# Perform Least Square Esimator Batch Compute
#   X Matrix of independent data, features in columns, obs in rows.
#   Y vector of depenedent data, (column vector)
def LSE(X, Y):
    return np.dot(
        np.dot(
            np.linalg.inv(
                np.dot(
                    np.transpose(X),
                    X
                )
            ),
            np.transpose(X)
        ),
        Y
    )
```

A.10.7 regression.py

```
# This file is a landing place for various optimization algorithms
#   Specifically LSE and LMA as developed in class.

# Core Python Dependencies
from os.path import sep # attempts to maintain OS portability
```

```
# Import Third-part Libraries
import numpy as np # used lots
import pandas as pd # used for pretty print, and input structuring
from scipy.stats import distributions # used for p-values, and confidence intervals

# Import things developed by me either
#   throughout the coursework, or
#   specifically for the term project
#   (provided in additional python files)
from .optimization import LSE
from . import stats

def _convertToNumpy(input):
    retVal = input.to_numpy()
    if len(retVal.shape) == 1:
        # force single variable input to be
        #   column vector
        retVal = retVal.reshape(-1, 1)
    return retVal

def _prepXInputForIntercept(x_input):
        tmp = _convertToNumpy(x_input)
        x = np.zeros((tmp.shape[0], tmp.shape[1] + 1))
        x[:, 0] = 1
        x[:, 1:] = tmp
        return x

def predict(X, B):
    return np.dot(X, B)

# calulcated ci, based on slides, and web resources
#   https://kite.com/python/examples/702/scipy-compute-a-confidence-interval-from-a-dataset
#   https://stats.stackexchange.com/questions/241449/matrix-and-regression-model
def _ci_calc(val, se, confidence):
    h = distributions.norm.ppf((1 - (1 - confidence)/2))
    low = val - h*se
    high = val + h*se
    return low, high


class LinRegModel():
    def __init__(self, intercept=True):
        self.intercept=intercept
        self._reset_state()
```

```
def _reset_state(self):
    # parameters set in fit routine
    self.X = None
    self.Y = None
    self.B = None
    self._p = None
    self._n = None
    self._df = None
    self.x_cols = None

    # cache params that are set elsewhere
    #  and need be reset when fit is called
    self._residuals = None
    self._cov = None
    self._coeffTable = None
    self._r2 = None
    self._adj_r2 = None

def _preprocessX(self, x_input):
    if self.intercept is True:
        return _prepXInputForIntercept(x_input)
    else:
        return _convertToNumpy(x_input)

def fit(self, x_input, y_input):
    self._reset_state()

    self.X = self._preprocessX(x_input)

    self.Y = _convertToNumpy(y_input)
    self.B = LSE(self.X, self.Y)
    self._p = self.B.shape[0] # num parameters
    self._n = self.Y.shape[0] # num samples
    self._df = self._n - self._p # degrees of freedom

    if isinstance(x_input, pd.DataFrame):
        self.x_cols = x_input.columns
    else:
        self.x_cols = [x_input.name]

def predict(self, x_input):
    x_in = self._preprocessX(x_input)

    y_pred = predict(x_in, self.B)

    return predict(x_in, self.B)
```

```
# predict residuals given same X used to fit
def residuals(self):
    if self._residuals is None:
        self._residuals = self.Y - predict(self.X, self.B)
    return self._residuals

# give covariance estimate per fitted regression problem.
#  https://stats.stackexchange.com/questions/68151/how-to-derive-variance-covariance-mat
#
def cov(self):
    if self._cov is None:
        # MLE estimate for variance given in wikipedia article
        #   https://en.wikipedia.org/wiki/Ordinary_least_squares#Estimation
        s2 = np.dot(self.residuals().T, self.residuals()) / self._df
        est_var = s2 * (self._df/self._n)

        # Simple covariance matrix
        #   https://en.wikipedia.org/wiki/Ordinary_least_squares#Covariance_matrix
        self._cov = est_var * np.linalg.inv(np.dot(self.X.T, self.X))
    return self._cov

def covTable(self):
    labels = self._labelList()

    return pd.DataFrame(self.cov(), columns=labels, index=labels)

# prints coefficients and their low/high intervals
def _compute_CITableRow(self, labels, coefs, ses, confidence):
    # verify sanity
    assert(len(coefs) == len(ses))

    # print line per coef/CI
    row_accum = []
    for label, coef, se in zip(labels, coefs, ses):
        low, high = _ci_calc(coef, se, confidence=confidence)
        t = stats.ttest_1sample(coef, se)
        p = stats.ttest_pvalue(t, self._df)
        row_accum.append((label, coef, se, low, high, t, p))
    return row_accum

def _labelList(self):
    if self.intercept is True:
        labels = ['intercept']
        labels.extend(self.x_cols)
    else:
```

```
            labels = list(self.x_cols)
        return labels

    # prints table of coefficients with Confidence Intervals
    def coeffTable(self, confidence=0.95):
        # all standard errors
        if self._coeffTable is None:
            ses = np.sqrt(
                self.cov()[np.diag_indices_from(self.cov())]
            )

            labels = self._labelList()

            columns = ['label','coeff', 'std err', 'lower', 'upper','t-test','p-value']
            rows = self._compute_CITableRow(
                labels= labels,
                coefs=self.B.reshape(-1),
                ses=ses,
                confidence=confidence
            )

            self._coeffTable = pd.DataFrame(rows, columns=columns)
        return self._coeffTable


    def print_summary(self):
        # print coefficient, low, high, t-test, p-value for coefficients involved
        print()
        print('#########################################################################')
        print('## Linear Regression Model Summary                                     ##')
        print('#########################################################################')
        print('Coefficients Table:')
        print(self.coeffTable())
        print()
        print('Model Metrics:')
        stats.print_metrics(
            y_pred = predict(self.X, self.B).reshape(-1),
            y_actual = self.Y.reshape(-1),
            num_params = self._p,
            sample_size = self._n
        )
```

A.10.8 stats.py

```
# This file defines various statistics related functions
```

```
# Third Party Imports
import numpy as np
from scipy.stats import distributions, chi2
import matplotlib.pyplot as plt # supports acf_plot function

# assumes 2 sided t-test
#   uses the survival function of t distribution to
#   figure out the p-value given a t-statistic
#   and the degrees of freedom.
def ttest_pvalue(t, df, sides=2):
    # sides should be 1 or 2
    assert(sides == 1 or sides == 2)

    # adapted from scipy stats code
    #   https://github.com/scipy/scipy/blob/adc4f4f7bab120ccfab9383aba272954a0a12fb0/scipy/s
    return distributions.t.sf(np.abs(t), df) * sides

# t-Test for computing the statistical significance of a given correlation r, and
#   dataset size n.
#   https://stats.stackexchange.com/questions/344006/understanding-t-test-for-linear-regress
#   https://en.wikipedia.org/wiki/Pearson_correlation_coefficient
def ttest(r, n):
    t = r * np.sqrt(
        (n-2)/(1 - r**2)
    )

    prob = ttest_pvalue(t, n-2)

    return (t, prob)

# differs from lab 8/lab 9
#   but based on that work, and wikipedia
#   https://en.wikipedia.org/wiki/Standard_error
def ttest_1sample(sample_mean, std_err, target_mean=0):
    return (sample_mean - target_mean)/std_err

# Pearson's correlation coefficient estimation as developed
#   in lab 2. Renamed
# def correlation_coefficient_cal(x, y):
def corr(x, y):
    # assumes that datasets are of the same length
    assert(len(x) == len(y))

    x_mean = np.mean(x)
    y_mean = np.mean(y)
```

```
    numerator = np.sum(
        np.multiply(
            np.subtract(x, x_mean),
            np.subtract(y, y_mean)
        )
    )

    denominator = np.sqrt(np.sum(np.power(
        np.subtract(x, x_mean),
        2
    ))) * np.sqrt(np.sum(np.power(
        np.subtract(y, y_mean),
        2
    )))

    return numerator/denominator


# 11 - Autocorrelation Function (ACF)
# python code for estimating AutoCorrelation Function
#   copied from lab 3
def autocorrelation_estimation(y, k, y_mean = None):
    if y_mean == None:
        y_mean = np.mean(y)

    # absolute value, as negative k values are symmetric with positive ones
    #  this makes it much easier to deal with below.
    k = np.abs(k)

    # length of y
    T = len(y)

    # initial t of 1st part of numerator summation
    t0 = k

    # final t of 2nd part of numerator summation
    t1 = T - k

    return np.sum (
        np.multiply(
            (y[t0:] - y_mean),
            (y[:t1] - y_mean)
        )
    ) / np.sum (
        np.power(
```

```
            y - y_mean,
            2
        )
    )

# copied and adapted from lab 3
#  updated to allow caller to manage saving plot
def acf_plot(residuals, label, k_max):
    residuals_mean = np.mean(residuals)
    acf = [(k, autocorrelation_estimation(residuals, k, y_mean=residuals_mean))
        for k in range(-k_max, k_max + 1)]

    # setting use_line_collection to true hides a deprecation warning
    plt.stem([a for a, b in acf],
        [b for a, b in acf])
    plt.title(f'ACF Plot of Generated Signal\n{label}')


# compute r_ab, t-test, and p-value
#   r_ab is Pearson's correlation coefficient between
#      a and b.
#   t-test is the t-statistic value computed based on
#      r_ab and the length of a and b.
#   p-value is a measure of the statistical significance
#      of the r_ab estimation.
def cor_and_ttest(a, b):
    n = len(a)

    # verify sanity
    assert(len(a) == len(b))
    assert(n > 2)
    r_ab = corr(a, b)
    t, prob = ttest(r_ab, n)

    return r_ab, t, prob

# assuming h based on this guidance:
#  https://robjhyndman.com/hyndsight/ljung-box-test/
def _h(T):
    # assumes that the timeseries data is not seasonal
    #   seems appropriate as we really hope our residuals don't
    #   show seasonal bahavior
    return min(10, int(T/5))


# based on reviewing lecture slides/notes
```

```
# and referring to scipy source code
def q_value(residuals, T, h=None):
    if h is None:
        h = _h(T)
    res_mean = np.mean(residuals)
    acfs = [autocorrelation_estimation(residuals, k, res_mean) for k in range(1, h + 1)]
    return T * np.sum(np.power(acfs, 2)[:h])

# Ljung-Box Q* from lectures
def q_value2(residuals, T, h=None):
    if h is None:
        h = _h(T)
    res_mean = np.mean(residuals)
    ks = range(1, h + 1)
    sum_tmp = sum([((T - k)**(-1)) * (autocorrelation_estimation(residuals, k, res_mean)**2)
    return T * (T + 2) * sum_tmp

# convert q to a p-value based on the survival function of the chi2 distribution
def q_to_pvalue(q, T, h=None):
    if h is None:
        h = _h(T)
    # adapted from https://www.statsmodels.org/stable/_modules/statsmodels/tsa/stattools.htm
    #   and
    return chi2.sf(q, h)

# adjusted r2 given
#   r2, pearson r squared
#   T, number of observations
def adj_r2(r2, T):
    return 1 - (1 - r2) * (T - 1)/(T - 1)

# sse given
#  residuals
def sse(residuals):
    return np.sum(residuals**2)

# rmse given
#   residuals
def rmse(residuals):
    return np.sqrt(np.mean(residuals**2))

# AIC based on lecture notes
#  T is number of observations
#  SSE is sum of squared error
#  k is number of parameters
#  TODO: uncertain if hardcoded 2,2 values need
```

```
#    adjustment per degrees of freedom or something?
def AIC(T, residuals, k):
    return T*np.log(sse(residuals)/T) + 2*(k + 2)

# AICc based on lecture notes
#  T is number of observations
#  SSE is sum of squared error
#  k is number of parameters
#  TODO: uncertain if hardcoded 2,2,3,3 values need
#    adjustment per degrees of freedom or something?
def AICc(T, residuals, k):
    return AIC(T, residuals, k) + (2 * (k + 2) * (k + 3)) / (T - k - 3)

# BIC based on lecture notes
#  T is number of observations
#  SSE is sum of squared error
#  k is number of parameters
#  TODO: uncertain if hardcoded 2 needs
#     adjustment per degrees of freedom or something?
def BIC(T, residuals, k):
    return T * np.log(sse(residuals) / T) + (k + 2) * np.log(T)

# included as the paper for the dataset uses it.
#   Mean Absolute Error
def mae(residuals):
    return np.sum(np.abs(residuals))/len(residuals)


# included as the paper for the dataset uses it.
#   Mean Absolute Percentate Error
def mape(residuals, y):
    return np.sum(np.divide(np.absolute(residuals), y))

def print_metrics(y_pred, y_actual, num_params, sample_size):
    r2 = np.nan_to_num(corr(y_actual, y_pred)**2)
    residuals = y_actual - y_pred
    print('\tR2:', r2)
    print('\tAdj R2:', np.nan_to_num(adj_r2(r2, y_pred.shape[0])))
    # Had trouble figuring out how to compute this or associated p-value without
    # just building an OLS model from statsmodels library and getting it from the summary
    # TODO: fix this if time allows.
    # print('\tF-Statistic:', )
    # print('\tF-Statistic (probability):', )
    print('\tAIC:', AIC(sample_size, residuals, num_params))
    print('\tAICc:', AICc(sample_size, residuals, num_params))
    print('\tBIC:', BIC(sample_size, residuals, num_params))
```

```
    q = q_value(residuals, sample_size)
    print('\tQ:', q)
    print('\tQ (p-value):', q_to_pvalue(q, sample_size))
    q2 = q_value2(residuals, sample_size)
    print('\tQ*:', q2)
    print('\tQ* (p-value):', q_to_pvalue(q, sample_size))
    print('\tSSE:', sse(residuals))
    print('\tRMSE:', rmse(residuals))
    print('\tMAE:', mae(residuals))
    print('\tMAPE:', mape(residuals, y_actual))
    print('\tResidual Mean:', np.mean(residuals))
    print('\tResidual Var:', np.var(residuals))
    print('\tResidual std dev:', np.std(residuals))
    return residuals
```

A.10.9 visualizations.py

```
# this file defines several graphics utilities
#  for assisting with plotting ACF, GPAC, etc..

# Core Python Dependencies

# Import Third-part Libraries
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Import things developed by me either
#   throughout the coursework, or
#   specifically for the term project
#   (provided in additional python files)
from . import stats


def acf_plot(residuals, label, k_max):
    residuals_mean = np.mean(residuals)
    acf = [(k, stats.autocorrelation_estimation(residuals, k, y_mean=residuals_mean))
        for k in range(-k_max, k_max + 1)]

    # setting use_line_collection to true hides a deprecation warning
    plt.stem([a for a, b in acf],
        [b for a, b in acf])
    plt.title(f'ACF Plot of Generated Signal\n{label}')
    plt.tight_layout()
```

```
def plot_GPAC(table, label):
    sns.heatmap(table, robust=True, annot=True)
    plt.title("GPAC Table\n" + label)
    plt.xscale
    plt.xlabel('Autoregressive Order (na-1)')
    plt.ylabel('Moving Avg Order (nb)')
    plt.tight_layout()
```