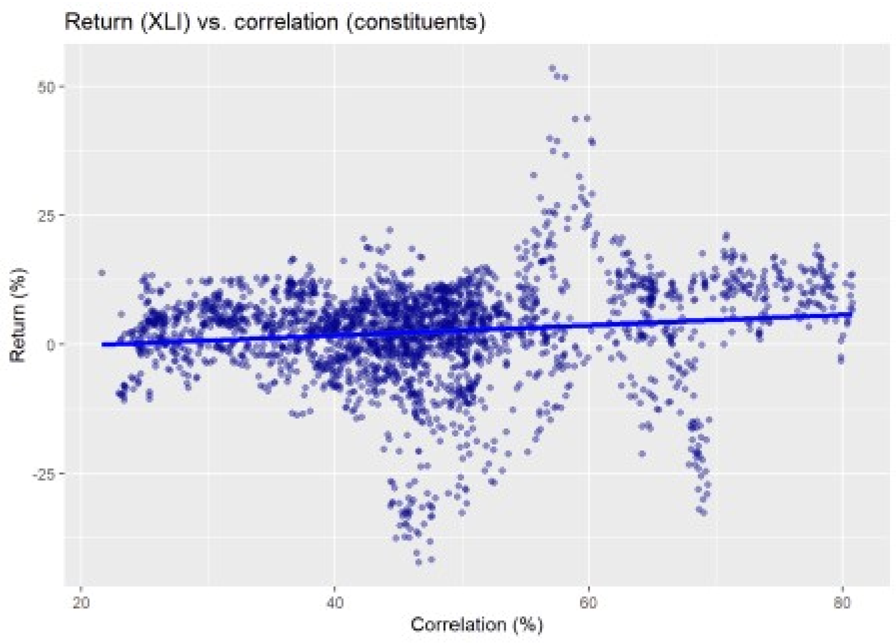
We proposed further analyses and were going to conduct one of them for this post, but then discovered the interesting R package generalCorr, developed by Professor H. Vinod of Fordham university, NY. While we can't do justice to all the package's functionality, it does offer ways to calculate non-linear dependence often missed by common correlation measures because such measures assume a linear relationship between the two sets of data. One particular function allows the user to identify probable causality between two pairs of variables. In other words, it tells you whether it is more likely x causes y or y causes x.

Why is this important? Our project is about exploring, and, if possible, identifying the predictive capacity of average rolling index constituent correlations on the index itself. If the correlation among the parts is high, then macro factors are probably exhibiting strong influence on the index. If correlations are low, then micro factors are probably the more important driver. Of course, other factors could cause rising correlations and the general upward trend of US equity markets should tend to keep correlations positive.

Additionally, if only a few stocks explain the returns on the index over a certain time frame, it might be possible to use the correlation of those stocks to predict future returns on the index. The notion is that the "memory" in the correlation could continue into the future. Then again, it might not!

But there's a bit of problem with this. If we're using a function that identifies non-linear dependence, we'll need to use a non-linear model to analyze the predictive capacity too. That means before we explore the generalCorr package we'll need some understanding of non-linear models.

In our previous post we analyzed the prior 60-trading day average pairwise correlations for all the constituents of the XLI and then compared those correlations to the forward 60-trading day return. Let's look at a scatter plot to refresh our memory. Recall, we split the data into roughly a 70/30 percent train-test split and only analyzed the training set. Since the data begins around 2005, the training set ends around mid-2015



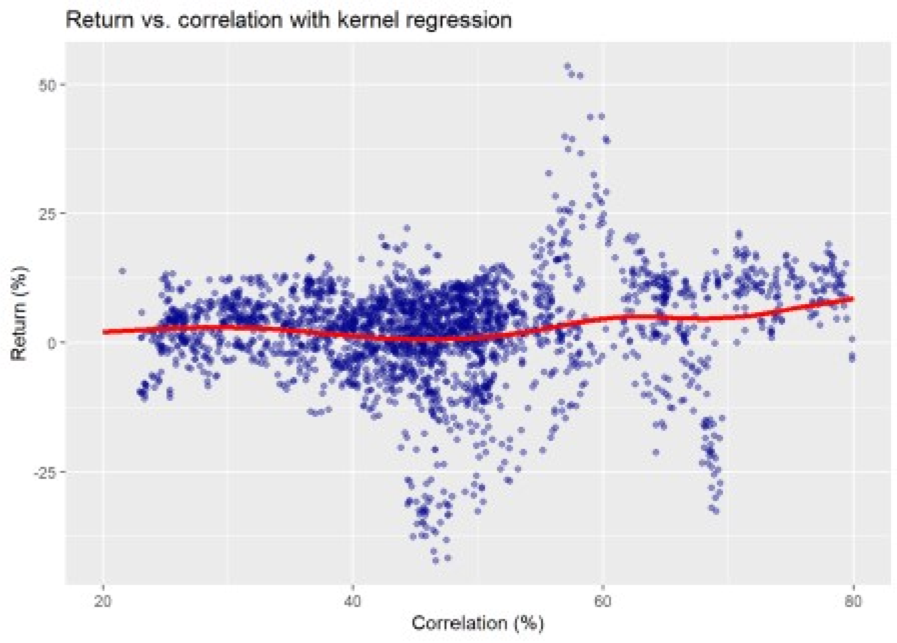
In the graph above, we see the rolling correlation doesn't yield a very strong linear relationship with forward returns. Moreover, there's clustering and apparent variability in the the relationship. Since our present concern is the non-linearity, we'll have to shelve these other issues for the moment.

The relationship between correlation and returns is clearly non-linear if one could call it a relationship at all. But where do we begin trying to model the non-linearity of the data? There are many algorithms that are designed to handle non-linearity: splines, kernels, generalized additive models, and many others. We'll use a kernel regression for two reasons: a simple kernel is easy to code—hence easy for the interested reader to reproduce—and the generalCorr package, which we'll get to eventually, ships with a kernel regression function.

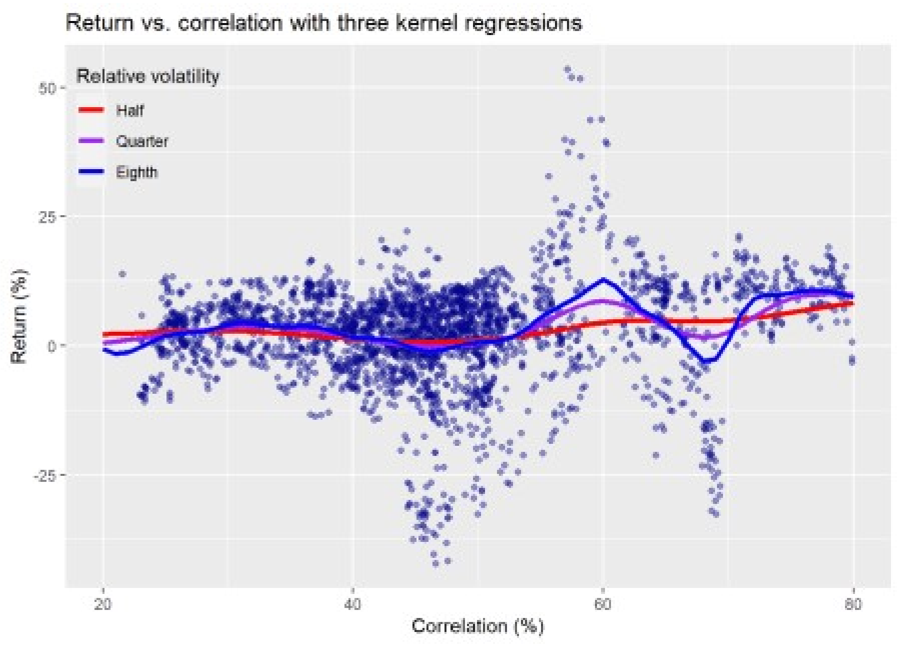
What is kernel regression? In simplistic terms, a kernel regression finds a way to connect the dots without looking like scribbles or flat lines. It assumes no underlying distribution. That is, it doesn't believe the data hails from a normal, lognormal, exponential, or any other kind of distribution. How does it do all this? The algorithm takes successive windows of the data and uses a weighting function (or kernel) to assign weights to each value of the independent variable in that window. Those weights are then applied to the values of the dependent variable in the window, to arrive at a weighted average estimate of the likely dependent value. Look at a section of data; figure out what the relationship looks like; use that to assign an approximate y value to the x value; repeat.

There are a bunch of different weighting functions: k-nearest neighbors, Gaussian, and eponymous multi-syllabic names. Window sizes trade off between bias and variance with constant windows keeping bias stable and variance inversely proportional to how many values are in that window. Varying window sizes—nearest neighbor, for example—allow bias to vary, but variance will remain relatively constant. Larger window sizes within the same kernel function lower the variance. Bias and variance being whether the model's error is due to bad assumptions or poor generalizability.

If all this makes sense to you, you're doing better than we are. Clearly, we can't even begin to explain all the nuances of kernel regression. Hopefully, a graph will make things a bit clearer; not so much around the algorithm, but around the results. In the graph below, we show the same scatter plot, using a weighting function that relies on a normal distribution (i.e., a Gaussian kernel) whose a width parameter is equivalent to about half the volatility of the rolling correlation.[1]



We see that there's a relatively smooth line that seems to follow the data a bit better than the straight one from above. How much better is hard to tell. What if we reduce the volatility parameter even further? We show three different parameters below using volatilities equivalent to a half, a quarter, and an eighth of the correlation.



This graph shows that as you lower the volatility parameter, the curve fluctuates even more. The smoothing parameter gives more weight to the closer data, narrowing the width of the window, making it more sensitive to local fluctuations.[2]

How does a kernel regression compare to the good old linear one? We run a linear regression and the various kernel regressions (as in the graph) on the returns vs. the correlation. We present the error (RMSE) and error scaled by the volatility of returns (RMSE scaled) in the table below.

| **Model** | **RMSE** | **RMSE scaled** |
|---|---|---|
| Linear | 0.097 | 0.992 |
| Kernel @ half volatility | 0.095 | 0.971 |
| Kernel @ quarter volatility | 0.092 | 0.944 |
| Kernel @ eighth volatility | 0.090 | 0.921 |

The table shows that, as the volatility parameter declines, the kernel regression improves from 2.1% points lower to 7.7% points lower error relative to the linear model. Whether or not a 7.7% point improvement in the error is significant, ultimately depends on how the model will be used. Is it meant to yield a trading signal? A tactical reallocation? Only the user can decide. Whatever the case, if improved risk-adjusted returns is the goal, we'd need to look at model-implied returns vs. a buy-and-hold strategy to quantify the significance, something we'll save for a later date.

For now, we could lower the volatility parameter even further. Instead, we'll check how the regressions perform using cross-validation to assess the degree of overfitting that might occur. We suspect that as we lower the volatility parameter, the risk of overfitting rises.

We run a four fold cross validation on the training data where we train a kernel regression model on each of the three volatility parameters using three-quarters of the data and then validate that model on the other quarter. We calculate the error on each fold, then average those errors for each parameter. We present the results below.

Table 1: Cross-validation errors and performance decline

| Parameter | Train | Validation | Decline (%) |
|---|---|---|---|
| Half | 0.090 | 0.103 | -12.6 |
| Quarter | 0.087 | 0.107 | -18.1 |
| Eighth | 0.085 | 0.110 | -22.4 |

As should be expected, as we lower the volatility parameter we effectively increase the sensitivity to local variance, thus magnifying the performance decline from training to validation set.

Let's compare this to the linear regression. We run the cross-validation on the same data splits. We present the results of each fold, which we omitted in the prior table for readability.

Table 2: Linear regression cross-validation errors and performance decline

| Train | Validation | Decline (%) |
|---|---|---|
| 0.106 | 0.045 | 137.0 |
| 0.099 | 0.087 | 14.3 |
| 0.099 | 0.095 | 4.7 |
| 0.067 | 0.177 | -62.3 |

What a head scratcher! The error rate improves in some cases! Normally, one wouldn't expect this to happen. A model trained on one set of data, shouldn't perform better on data it hasn't seen; it should perform worse! But that's the idiosyncratic nature of time series data. We believe this "anomaly" is caused by training a model on a period with greater volatility and less of an upward trend, than the period on which its validated. Given upwardly trending markets in general, when the model's predictions are run on the validation data, it appears more accurate since it is more likely to predict an up move anyway; and, even if the model's size effect is high, the error is unlikely to be as severe as in choppy markets because it won't suffer high errors due to severe sign change effects.

So which model is better? If we aggregate the cross-validation results, we find that the kernel regressions see a -18% worsening in the error vs. a 23.4% improvement for the linear model. But we know we can't trust that improvement. Clearly, we need a different performance measure to account for regime changes in the data. Or we could run the cross-validation with some sort of block sampling to account for serial correlation while diminishing the impact of regime changes. Not exactly a trivial endeavor.

These results beg the question as to why we didn't see something similar in the kernel regression. Same time series, why not the same effect? The short answer is we have no idea without looking at the data in more detail. We suspect there might be some data snooping since we used a range for the weighting function that might not have existed in the training set. We assume a range for the correlation values from zero to one on which to calculate the respective weights. But in the data, the range of correlation is much tighter— it doesn't drop much below ~20% and rarely exceeds ~80%.

Another question begging idea that pops out of the results is whether it is appropriate (or advisable) to use kernel regression for prediction? In many cases, it probably isn't advisable insofar as kernel regression could be considered a "local" regression. That is, it's deriving the relationship between the dependent and independent variables on values within a set window.

At least with linear regression it calculates the best fit using all of available data in the sample. But just as the linear regression will yield poor predictions when it encounters x values that are significantly different from the range on which the model is trained, the same phenomenon is likely to occur with kernel regression. Using correlation as the independent variable glosses over this somewhat problem since its range is bounded.[3]

Whatever the case, should we trust the kernel regression more than the linear? In one sense yes, since it performed—at least in terms of errors—exactly as we would expect any model to perform. That the linear model shows an improvement in error could lull one into a false sense of success. Not that we'd expect anyone to really believe they've found the Holy Grail of models because the validation error is better than the training error. But, paraphrasing Feynman, the easiest person to fool is the model-builder himself.

We've written much more for this post than we had originally envisioned. And we haven't even reached the original analysis we were planning to present! Nonetheless, as we hope you can see, there's a lot to unpack on the topic of non-linear regressions. We'll next look at actually using the generalCorr package we mentioned above to tease out any potential causality we can find between the constituents and the index. From there we'll be able to test out-of-sample results using a kernel regression. The suspense is killing us!

Until next time let us know what you think of this post. Did we fall down a rabbit hole or did we not go deep enough? And while you think about that here's the code.

R code:

```
# Built using R 3.6.2

## Load packages
suppressPackageStartupMessages({
  library(tidyverse)
  library(tidyquant)
  library(reticulate)
})

## Load data from Python
pd <- import("pandas")
prices <- pd$read_pickle("python/xli_prices.pkl")

xli <- pd$read_pickle("python/xli_etf.pkl")

# Create date index for xts
dates <- as.Date(rownames(prices))
price_xts <- as.xts(prices, order.by = dates)
price_xts <- price_xts[,!colnames(price_xts) %in% c("OTIS", "CARR")]

xli_xts <- as.xts(xli, order.by = dates)
names(xli_xts) <- "xli"
head(xli_xts)

prices_xts <- merge(xli_xts, price_xts)
prices_xts <- prices_xts[,!colnames(prices_xts) %in% c("OTIS", "CARR")]

# Create function for rolling correlation
```

```
mean_cor <- function(returns)
{
  # calculate the correlation matrix
  cor_matrix <- cor(returns, use = "pairwise.complete")

  # set the diagonal to NA (may not be necessary)
  diag(cor_matrix) <- NA

  # calculate the mean correlation, removing the NA
  mean(cor_matrix, na.rm = TRUE)
}

# Create data frame for regression
corr_comp <- rollapply(comp_returns, 60,
                       mean_cor, by.column = FALSE, align = "right")

xli_rets <- ROC(prices_xts[,1], n=60, type = "discrete")

total_60 <- merge(corr_comp, lag.xts(xli_rets, -60))[60:
(nrow(corr_comp)-60)]
colnames(total_60) <- c("corr", "xli")
split <- round(nrow(total_60)*.70)


train_60 <- total_60[1:split,]
test_60 <- total_60[(split+1):nrow(total_60),]

train_60 %>%
  ggplot(aes(corr*100, xli*100)) +
  geom_point(color = "darkblue", alpha = 0.4) +
  labs(x = "Correlation (%)",
       y = "Return (%)",
       title = "Return (XLI) vs. correlation (constituents)") +
  geom_smooth(method = "lm", se=FALSE, size = 1.25, color = "blue")


## Simple kernel

x_60 <- as.numeric(train_60$corr)
y_60 <- as.numeric(train_60$xli)


# Code derived from the following article https://towardsdatascience.com/kernel-
regression-made-easy-to-understand-86caf2d2b844

## Guassian kernel function
gauss_kern <- function(X, y, bandwidth, X_range = c(0,1)){

  pdf <- function(X,bandwidth){
    out = (sqrt(2*pi))^-1*exp(-0.5 *(X/bandwidth)^2)
    out
  }
  if(length(X_range) > 1){
    x_range <- seq(min(X_range), max(X_range), 0.01)
  } else {
    x_range <- X_range
  }

  xy_out <- c()

  for(x_est in x_range){
    x_test <-  x_est - X
    kern <- pdf(x_test, bandwidth)
    weight <- kern/sum(kern)
    y_est <- sum(weight*y)
    x_y <- c(x_est, y_est)
    xy_out <- rbind(xy_out,x_y)
  }

  xy_out

}

kernel_out <- gauss_kern(x_60, y_60, 0.06)

kernel_df <- data.frame(corr = as.numeric(train_60$corr),
                        xli = as.numeric(train_60$xli),
                        estimate = rep(NA, nrow(train_60)))


# Output y value
for(i in 1:nrow(kernel_df)){
  kernel_df$estimate[i] <- gauss_kern(x_60, y_60, 0.06,
kernel_df$corr[i])[2]
}

kernel_out <- kernel_out %>%  as.data.frame()
colnames(kernel_out) <- c("x_val", "y_val")

# Graph model and scatter plot
kernel_df %>%
  ggplot(aes(corr*100, xli*100)) +
  geom_point(color = "darkblue", alpha=0.4) +
  geom_path(data=kernel_out,
            aes(x_val*100, y_val*100),
            color = "red",
            size = 1.25) +
  xlim(20, 80) +
  labs(x = "Correlation (%)",
       y = "Return (%)",
       title = "Return vs. correlation, linear vs. kernel regression")

# Calculate errors
```

```r
rmse_kern <- sqrt(mean((kernel_df$xli - kernel_df$estimate)^2))
rmse_kernf_scale <- rmse_kern/sd(kernel_df$xli)

linear_mod <- lm(xli ~ corr, kernel_df)
rmse_linear <- sqrt(mean(kern_df_mod$residuals^2))
rmse_lin_scale <- rmse_linear/sd(kernel_df$xli)

## Multiple kernels

b_width <- c(0.0625, 0.03125, 0.015625)

kernel_out_list <- list()

for(i in 1:3){
  b_i <- b_width[i]
  val_out <- gauss_kern(x_60, y_60, b_i)
  val_out <- val_out %>%
    as.data.frame() %>%
    mutate(var = i) %>%
    `colnames<-`(c("x_val", "y_val", "var"))
  kernel_out_list[[i]] <- val_out
}

kernel_out_list <- do.call("rbind", kernel_out_list) %>%
as.data.frame()

# Graph multiple kernel regerssions
ggplot() +
  geom_point(data = kernel_df,
             aes(corr*100, xli*100),
             color = "darkblue", alpha=0.4) +
  geom_path(data=kernel_out_list,
            aes(x_val*100, y_val*100, color = as.factor(var)),
            size = 1.25) +
  xlim(20, 80) +
  scale_color_manual("Relative volatility", labels = c("1" = "Half",
"2" = "Quarter", "3" = "Eighth"),
                     values = c("1" = "red", "2" = "purple",
"3"="blue")) +
  labs(x = "Correlation (%)",
       y = "Return (%)",
       title = "Return vs. correlation with three kernel regressions")
+
  theme(legend.position = c(0.06,0.85),
        legend.background = element_rect(fill = NA))

# Create table for comparisons
kernel_df <- kernel_df %>%
  mutate(est_h = NA,
         est_q = NA,
         est_8 = NA)
```

```
for(j in 1:3){
  ests <- c()
  for(i in 1:nrow(kernel_df)){
    out <- gauss_kern(x_60, y_60, b_width[j], kernel_df$corr[i])[2]
    ests[i] <- out
  }
  kernel_df[,j+3] <- ests
}

kern_rmses <- apply(kernel_df[,4:6],2, function(x)
sqrt(mean((kernel_df$xli - x)^2))) %>%
  as.numeric()

kern_rmses_scaled <- kern_rmses/sd(kernel_df$xli)

rmse_df <- data.frame(Model = c("Linear", "Kernel @ half", "Kernel @
quarter", "Kernel @ eighth"),
                      RMSE = c(rmse_linear, kern_rmses),
                      `RMSE scaled` = c(rmse_lin_scale,
kern_rmses_scaled),
                      check.names = FALSE)

rmse_df %>%
  mutate_at(vars(RMSE, `RMSE scaled`), function(x) round(x,3)) %>%
  knitr::kable()


min_improv <- round(min(rmse_df[1,2]/rmse_df[2:4,2]-1),3)*100
max_improv <- round(max(rmse_df[1,2]/rmse_df[2:4,2]-1),3)*100


## Simple kernel cross-validation
c_val_idx <- round(nrow(train_60)/5)
c_val1 <- seq(1,c_val_idx*4)
c_val2 <- c(seq(1,c_val_idx*3), seq(c_val_idx*4, nrow(train_60)))
c_val3 <- c(seq(1,c_val_idx*2), seq(c_val_idx*3, nrow(train_60)))
c_val4 <- c(seq(1,c_val_idx), seq(c_val_idx*2, nrow(train_60)))
seqs <- list(c_val1, c_val2, c_val3, c_val4)

b_width <- c(0.0625, 0.03125, 0.015625)

kern_df <- c()

for(band in b_width){

  for(i in 1:length(seqs)){
    x_train <- as.numeric(train_60$corr)[seqs[[i]]]
    x_test <- as.numeric(train_60$corr)[!seq(1,nrow(train_60)) %in%
seqs[[i]]]

    y_train <- as.numeric(train_60$xli)[seqs[[i]]]
    y_test <- as.numeric(train_60$xli)[!seq(1,nrow(train_60)) %in%
seqs[[i]]]

    # kern_out <- gauss_kern(x_train, y_train, band )

    pred <- NULL
    for(xs in x_train){
        out <- gauss_kern(x_train, y_train, band, xs)[2]
        pred <- rbind(pred,out)
    }

    rmse_train_kern <- sqrt(mean((y_train-pred)^2, na.rm = TRUE))

    pred_test <- c()
    for(xs in x_test){
        out <- gauss_kern(x_train, y_train, band, xs)[2]
        pred_test <- rbind(pred_test,out)
    }

    rmse_test_kern <- sqrt(mean((y_test-pred_test)^2, na.rm = TRUE))

    rmses_kern <- cbind(band, rmse_train_kern, rmse_test_kern)
    kern_df <- rbind(kern_df, rmses_kern)

  }
}

# Print results table
kern_df %>%
  as.data.frame() %>%
  group_by(band) %>%
  summarise_all(mean) %>%
  mutate(decline = round((rmse_train_kern/rmse_test_kern-1)*100,1)) %>%
  arrange(desc(band)) %>%
  mutate_at(vars(-band, -decline),function(x) round(x,3)) %>%
  mutate(band = ifelse(band > 0.06, "Half", ifelse(band < 0.02,
"Eighth", "Quarer"))) %>%
  rename("Parameter" = band,
         "Train" = rmse_train_kern,
         "Validation" = rmse_test_kern,
         "Decline (%)" = decline) %>%
  knitr::kable(caption = "Cross-validation errors and performance
decline ")

mean_kern_decline <- kern_df %>%
  as.data.frame() %>%
  group_by(band) %>%
  summarise_all(mean) %>%
  mutate(decline = rmse_train_kern/rmse_test_kern-1) %>%
  summarise(decline = round(mean(decline),2)*100) %>%
  as.numeric()

## Linear cross validation
```

```
lm_df <- c()
lm_dat <- coredata(train_60[,c("xli", "corr")]) %>% as.data.frame()

seqs <- list(c_val1, c_val2, c_val3, c_val4)

for(i in 1:length(seqs)){
  train <- lm_dat[seqs[[i]], ]
  test <- lm_dat[!seq(1,nrow(lm_dat)) %in% seqs[[i]],]

  mod <- lm(xli ~ corr, train)
  pred_train <- predict(mod, train, type = "response")
  pred_test <- predict(mod, test, type = "response")

  rmse_train <- sqrt(mean((train$xli - pred_train)^2, na.rm=TRUE))
  rmse_test <- sqrt(mean((test$xli - pred_test)^2,na.rm=TRUE))

  rmse <- cbind(rmse_train, rmse_test)
  lm_df <- rbind(lm_df, rmse)

}

# Print results table
lm_df %>%
  as.data.frame() %>%
  mutate(decline = round((rmse_train/rmse_test-1)*100,1)) %>%
  mutate_at(vars(-decline),function(x) round(x,3)) %>%
  rename("Train" = rmse_train,
         "Validation" = rmse_test,
         "Decline (%)" = decline) %>%
  knitr::kable(caption = "Linear regression cross-validation errors and
performance decline")

mean_lin_decline <- lm_df %>%
  as.data.frame() %>%
  mutate(decline = rmse_train/rmse_test-1) %>%
  summarise(decline = round(mean(decline),3)*100) %>%
  as.numeric()
```

Python code:

```
# Built uisng Python 3.7.4

## Import libraries
import numpy as np
import pandas as pd
import pandas_datareader as dr
import matplotlib.pyplot as plt
import matplotlib
%matplotlib inline
matplotlib.rcParams['figure.figsize'] = (12,6)
plt.style.use('ggplot')
```

```
## See prior post for code to download prices

## Get rpices
prices = pd.read_pickle('xli_prices.pkl')
xli = pd.read_pickle('xli_etf.pkl')

returns = prices.drop(columns = ['OTIS', 'CARR']).pct_change()
returns.head()

## Create rolling correlation function
def mean_cor(df):
    corr_df = df.corr()
    np.fill_diagonal(corr_df.values, np.nan)
    return np.nanmean(corr_df.values)

## compile data and create train, test split
corr_comp = pd.DataFrame(index=returns.index[59:])
corr_comp['corr'] = [mean_cor(returns.iloc[i-59:i+1,:]) for i in
range(59,len(returns))]
corr_comp.head()

xli_rets = xli.pct_change(60).shift(-60)

total_60 = pd.merge(corr_comp, xli_rets, how="left",
on="Date").dropna()
total_60.columns = ['corr', 'xli']

split = round(len(total_60)*.7)
train_60 = total_60.iloc[:split,:]
test_60 = total_60.iloc[split:, :]

## Scatter plot with linear regression
# Note: could have done this with Seaborn, But wanted flexibility later
for other kernel regressions
from sklearn.linear_model import LinearRegression
X = train_60['corr'].values.reshape(-1,1)
y = train_60['xli'].values.reshape(-1,1)
lin_reg = LinearRegression().fit(X,y)
y_pred = lin_reg.predict(X)

plt.figure(figsize=(12,6))
plt.scatter(train_60['corr']*100, train_60['xli']*100, color='blue',
alpha = 0.4)
plt.plot(X*100, y_pred*100, color = 'darkblue')
plt.xlabel("Correlation (%)")
plt.ylabel("Return (%)")
plt.title("Return (XLI) vs. correlation (constituents)")
plt.show()


# Create gaussian kernel function
# The following websites were helpful in addition to the article
```

```
mentioned above
# https://github.com/kunjmehta/Medium-Article-Codes/blob/master/gaussian-kernel-regression-from-scratch.ipynb
# https://www.kaggle.com/kunjmehta/gaussian-kernel-regression-from-scratch

def gauss_kern(X,y, bandwidth, X_range=[0,1]):
    def pdf(X, bandwidth):
        return (bandwidth * np.sqrt(2 * np.pi))**-1 * np.exp(-0.5*
(X/bandwidth)**2)

    if len(X_range) > 1:
        x_range = np.arange(min(X_range), max(X_range), 0.01)
    else:
        x_range = X_range

    xy_out = []

    for x_est in x_range:
        x_test = x_est - X
        kern = pdf(x_test, bandwidth)
        weight = kern/np.sum(kern)
        y_est = np.sum(weight * y)
        xy_out.append([x_est, y_est])

    return np.array(xy_out)

## Run kernel regresssion
kernel_out = gauss_kern(train_60['corr'].values,
train_60['xli'].values, 0.06)

## Plot kernel regression over scatter plot
plt.figure(figsize=(12,6))
plt.scatter(train_60['corr']*100, train_60['xli']*100, color='blue',
alpha = 0.4)
plt.plot(kernel_out[:,0]*100, kernel_out[:,1]*100, color = 'red')
plt.xlim(20,80)
plt.xlabel("Correlation (%)")
plt.ylabel("Return (%)")
plt.title("Return (XLI) vs. correlation (constituents)")
plt.show()

## Run kernel regression on multiple bandwiths
b_width = [0.0625, 0.03125, 0.015625]

kernel_out_list = []

for i in range(3):
    b_i = b_width[i]
    val_out = gauss_kern(train_60['corr'], train_60['xli'], b_i)
    kernel_out_list.append(val_out)
```

```
## Plot multiple regressions
cols = ['red', 'purple', 'blue']
labs = ['Half', 'Quarter', 'Eighth']

plt.figure(figsize=(12,6))
plt.scatter(train_60['corr']*100, train_60['xli']*100, color='blue',
alpha = 0.4)

for i in range(3):
    plt.plot(kernel_out_list[i][:,0]*100, kernel_out_list[i][:,1]*100,
color = cols[i], label = labs[i])

plt.xlim(20,80)
plt.xlabel("Correlation (%)")
plt.ylabel("Return (%)")
plt.title("Return vs. correlation with kernel regressions")
plt.legend(loc='upper left', title = "Relative volatility")
plt.show()

## Print RMSE comparisons
ests = []
for j in range(3):
    est_out = []
    for i in range(len(train_60)):
        out =  gauss_kern(train_60['corr'], train_60['xli'],
b_width[j], [train_60['corr'][i]])[0][1]
        est_out.append(out)
    ests.append(np.array(est_out))

lin_rmse = np.sqrt(np.mean((train_60['xli'].values - y_pred)**2))

rmse = [lin_rmse]
for k in range(3):
    rmse_k = np.sqrt(np.mean((train_60['xli'].values - ests[k])**2))
    rmse.append(rmse_k)

rmse_scaled = [x/np.std(train_60['xli']) for x in rmse]
models = ["Linear", "Kernel @ half", "Kernel @ quarter", "Kernel @
eighth"]

for l in range(4):
    print(f'{models[l]} -- RMSE: {rmse[l]:0.03f} RMSE scaled:
{rmse_scaled[l]:0.03f}')

## Kernel cross-validation
c_val_idx = round(len(train_60)/5)
c_val1 = np.arange(0,c_val_idx*4)
c_val2 = np.concatenate((np.arange(0,c_val_idx*3),np.arange(c_val_
idx*4-1,len(train_60))))
c_val3 = np.concatenate((np.arange(0,c_val_idx*2),np.arange(c_val_
idx*3-1,len(train_60))))
c_val4 = np.concatenate((np.arange(0,c_val_idx),np.arange(c_val_idx*
2-1,len(train_60))))
seqs = [c_val1, c_val2, c_val3, c_val4]

kern_df = []
b_width = [0.065, 0.03125, 0.015625]

for band in b_width:
    for seq in seqs:
        test_val = [x for x in np.arange(len(train_60)) if x not in
seq]
        x_train = train_60['corr'][seq].values
        x_test = train_60['corr'][test_val].values

        y_train = train_60['xli'][seq].values
        y_test = train_60['xli'][test_val].values

        pred = []
        for xs in x_train:
            out = gauss_kern(x_train, y_train, band, [xs])[0][1]
            pred.append(out)

        rmse_train_kern = np.sqrt(np.mean((y_train - pred)**2))

        pred_test = []
        for xs in x_test:
            out = gauss_kern(x_train, y_train, band, [xs])[0][1]
            pred_test.append(out)

        rmse_test_kern = np.sqrt(np.mean((y_test - pred_test)**2))
        rmses_kern = [band, rmse_train_kern, rmse_test_kern]

        kern_df.append(rmses_kern)

## Print cross-validation results
kern_df = pd.DataFrame(kern_df, columns = ['Parameter', 'Train',
'Validation'])
kern_df
kern_df.groupby('Parameter')[['Train', 'Test']].apply(lambda x:
x.mean())

kern_df_out = kern_df.groupby('Parameter').mean()
kern_df_out['Decline'] = kern_df_out['Train']/kern_df_out['Test']-1
kern_df_out.apply(lambda x: round(x,3))

## Linear model cross-valdation
lm_df = []

for seq in seqs:
    test_val = [x for x in np.arange(len(train_60)) if x not in seq]
    x_train = train_60['corr'][seq].values.reshape(-1,1)
    x_test = train_60['corr'][test_val].values.reshape(-1,1)
    y_train = train_60['xli'][seq].values.reshape(-1,1)
    y_test = train_60['xli'][test_val].values.reshape(-1,1)

    lin_reg = LinearRegression().fit(x_train, y_train)

    pred_train = lin_reg.predict(x_train)
    rmse_train = np.sqrt(np.mean((y_train-pred_train)**2))

    pred_test = lin_reg.predict(x_test)
    rmse_test = np.sqrt(np.mean((y_test-pred_test)**2))
    lm_df.append([rmse_train, rmse_test])

##Print linear model results
lm_df = pd.DataFrame(lm_df, columns = ['Train', 'Test'])
lm_df['Decline'] = lm_df['Train']/lm_df['Test']-1
lm_df.apply(lambda x: round(x,3))
```