In [1]:

```
# Data Management
import pandas as pd
import tables as tb
from datetime import datetime
from datetime import timedelta
```

In [2]:

```
# Numerical Methods and Machine Learning Models
import numpy as np
import numba
import statsmodels
import statsmodels.api as sm
from scipy.stats import norm
from sklearn import metrics
from sklearn.preprocessing import MinMaxScaler
from sklearn.decomposition import KernelPCA

from keras.models import Sequential
from keras.layers import LSTM,Dense,Dropout
import torch
import torch.nn as nn
from torch.autograd import Variable
```

In [3]:

```
# Data Acquisition
import pandas_datareader.data as web
from joblib import Parallel, delayed
```

In [4]:

```
# Plotting
import plotly.graph_objects as go
from plotly.subplots import make_subplots
import matplotlib
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
from pandas.plotting import autocorrelation_plot
```

In [5]:

```
import warnings
warnings.filterwarnings('ignore')
```

Here we acquire Lazard's stock data over the course of 2020

In [6]:

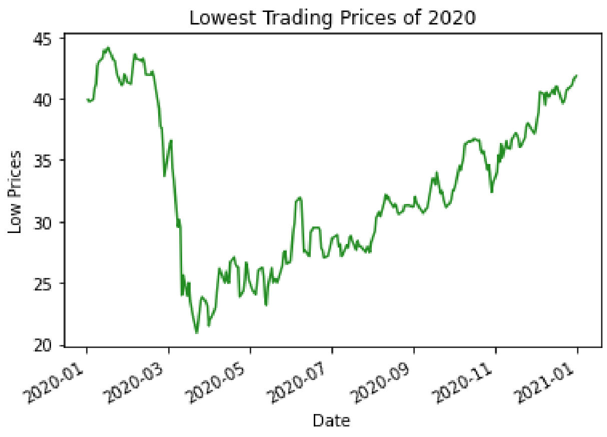
```
lazardData = web.DataReader('LAZ',data_source='yahoo',start='1/1/2020',end='1/1/2021')
```

EDA

Lows and Highs Plotted for Lazard

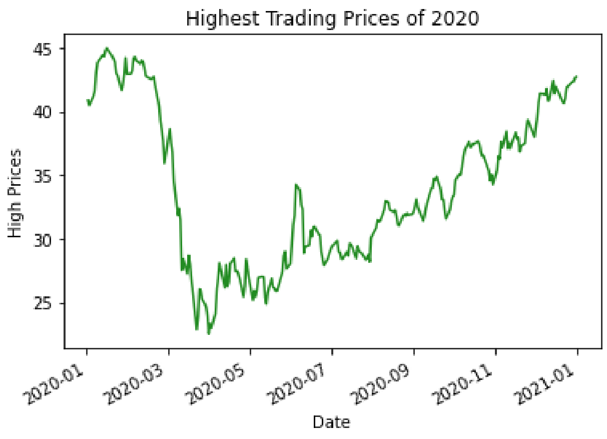
In [7]:

```
lazardData['Low'].plot(color='forestgreen')
plt.title("Lowest Trading Prices of 2020")
plt.xlabel("Date")
plt.ylabel("Low Prices")
plt.show()
```



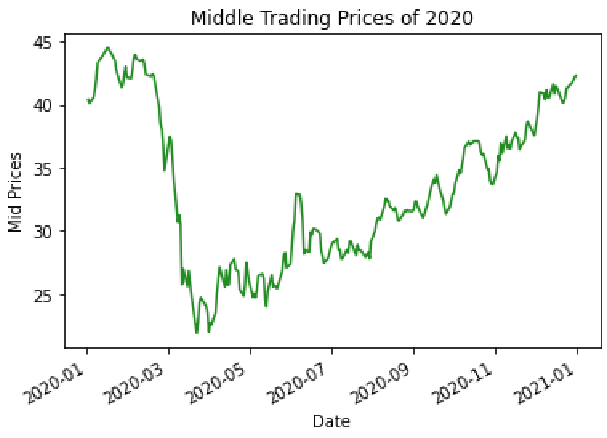
In [8]:

```
lazardData['High'].plot(color='forestgreen')
plt.title("Highest Trading Prices of 2020")
plt.xlabel("Date")
plt.ylabel("High Prices")
plt.show()
```



In [9]:

```
# we take the mid trading prices of the stock
((lazardData["High"]+lazardData["Low"])/2).plot(color='forestgreen')
plt.title("Middle Trading Prices of 2020")
plt.ylabel("Mid Prices")
plt.show()
```



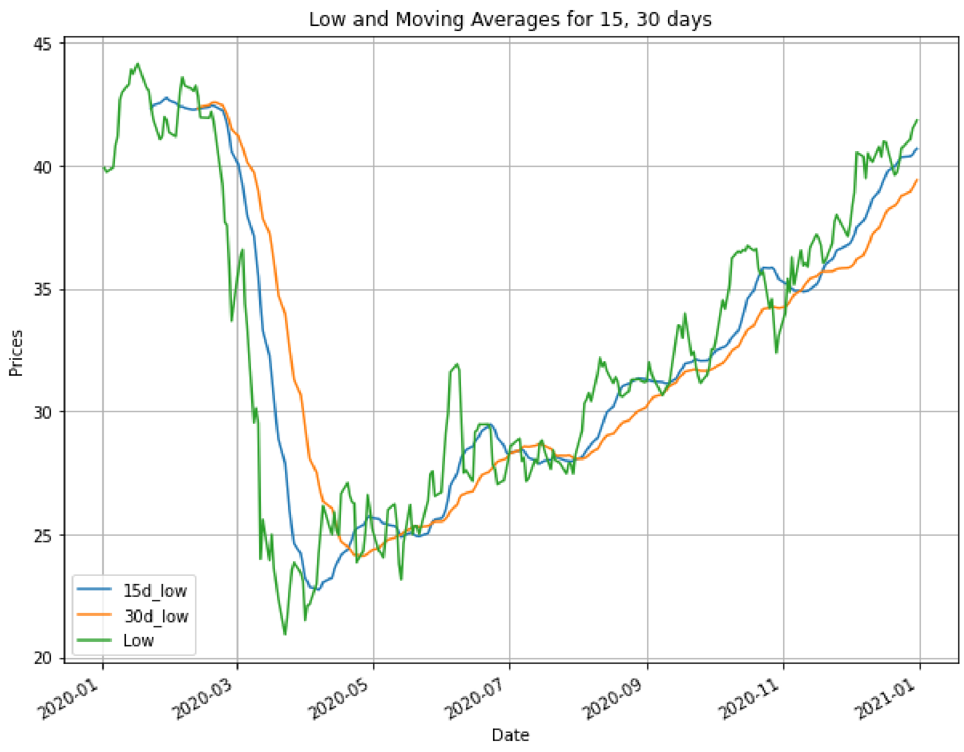
We can look into general trendlines over the course of "x" days in a rolling or moving average

In [10]:

```
# for the lowest trading prices
lazardData['15d_low'] = lazardData['Low'].rolling(15).mean()
lazardData['30d_low'] = lazardData['Low'].rolling(30).mean()
```

In [11]:

```
lazardData[['15d_low','30d_low',"Low"]].plot(grid=True,figsize=(10,8))
plt.title("Low and Moving Averages for 15, 30 days")
plt.xlabel("Date")
plt.ylabel("Prices")
plt.show()
```



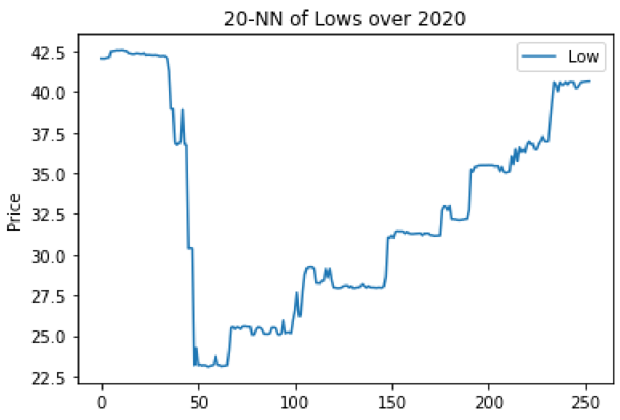
Other non-parametric methods such as general linear smoothers can be applied. Here, K-nearest neighbors nonparametric estimation is done.

In [12]:

```
def KNNEstimation(kNeighbors,data):
    L = len(data)
    if kNeighbors > L:
        print("Neighbors exceed data length.")
        return
    resultCol = []
    for i in range(L):
        currNeighbors = 0
        neighbors = np.array([])
        lower,upper = i,i
        while currNeighbors < kNeighbors:
            if lower == -1:
                neighbors = np.append(neighbors,[data.iloc[upper]])
                upper += 1
            elif upper == L:
                neighbors = np.append(neighbors,[data.iloc[lower]])
                lower -= 1
            else:
                lowerDist = abs(data.iloc[lower]-data.iloc[i])
                upperDist = abs(data.iloc[upper]-data.iloc[i])
                if lowerDist > upperDist:
                    neighbors = np.append(neighbors,[data.iloc[upper]])
                    upper += 1
                else:
                    neighbors = np.append(neighbors,[data.iloc[lower]])
                    lower -= 1
            currNeighbors += 1
        resultCol.append(np.average(neighbors))

    return resultCol
```

In [13]:

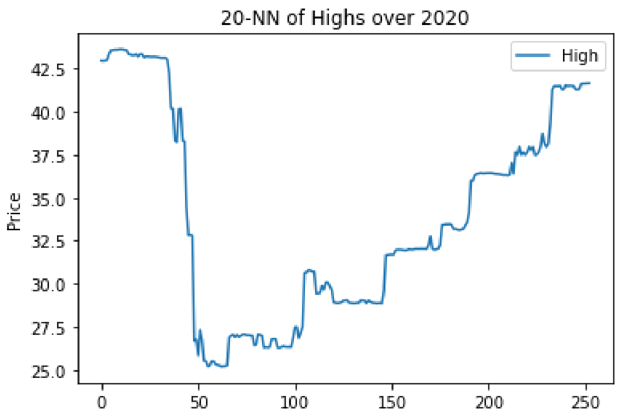
```
%%time
lowKNNData = KNNEstimation(20,lazardData["Low"])
pd.DataFrame(lowKNNData,columns=["Low"]).plot()
plt.title("20-NN of Lows over 2020")
plt.ylabel("Price")
plt.show()
```



```
CPU times: user 306 ms, sys: 35.6 ms, total: 342 ms
Wall time: 328 ms
```

In [14]:

```
%%time
highKNNData = KNNEstimation(20,lazardData["High"])
pd.DataFrame(highKNNData,columns=["High"]).plot()
plt.title("20-NN of Highs over 2020")
plt.ylabel("Price")
plt.show()
```



```
CPU times: user 302 ms, sys: 28 ms, total: 330 ms
Wall time: 322 ms
```

In [15]:

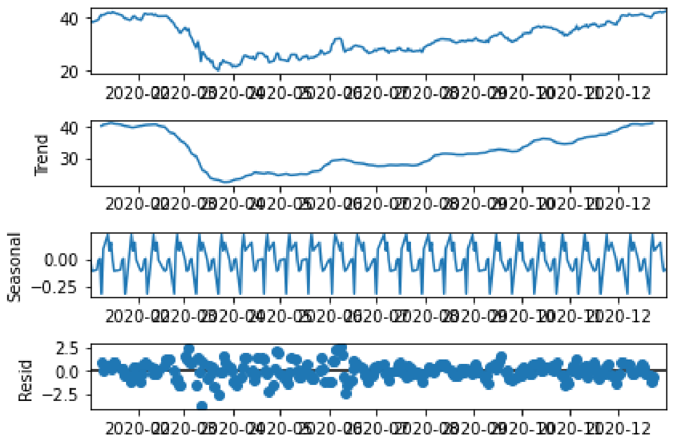
```
###
```

In [16]:

```
lazardPrices = pd.DataFrame(lazardData['Adj Close'])
```

In [17]:

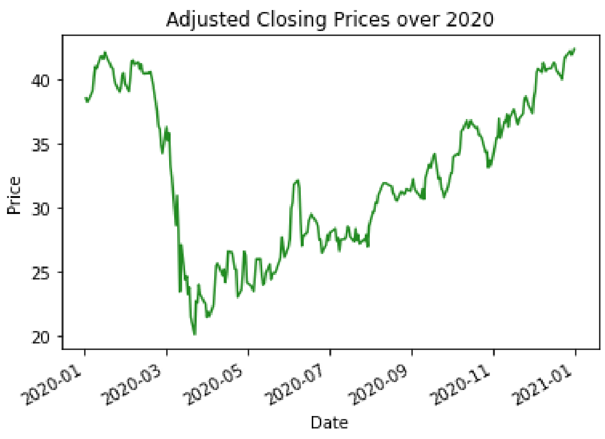
```
decompLazard = sm.tsa.seasonal_decompose(lazardPrices, model='additive',period=10)
decompLazard.plot()
plt.show()
```



Adjusted Closing Price Time Series Visualization

In [18]:

```
lazardData['Adj Close'].plot(color='forestgreen')
plt.title("Adjusted Closing Prices over 2020")
plt.ylabel("Price")
plt.show()
```



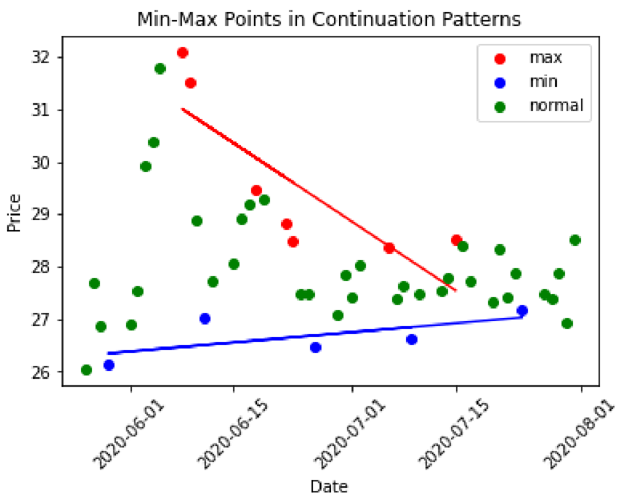
Forex Triangle Continuation Patterns

The sensitivity of how many highs or lows we want to use in the construction of the lower and upper boundaries of the triangle continuation patterns in stock prices impacts the final boundaries used in our hypothesis tests. For an ascending triangle, we determine whether the upper boundary may not be statistically significant from 0 and the lower boundary as statistically significant from 0 and positive. Similar tests apply to descending and symmetric triangles. Certainly visual inspection can be done, but numerical verification is another option in analytics.

In [19]:

```
class ForexTriangles:
    def __init__(self,data,prop):
        # input data
        self.data = data
        # proportion of points to be sampled.
        self.prop = prop
    # for relatively flat surfaces, we may still not reach convergence.
    def sgdRegression(self, points, learningRate, epochs=1000):
```

```
    def sgdRegression(self, points, learningRate, epochs=1000):
        params = [0,-100]
        for epoch in range(epochs):
            totalError = 0
            np.random.shuffle(points)
            gradW,gradB = 0,0
            for point in points:
                loss = (params[0]+point[0]*params[1]) - point[1]
                totalError += loss**2
                params[0] = params[0] - 2*point[0]*loss*learningRate
                params[1] = params[1] - learningRate*2*loss
            print("Epoch:",epoch,"Error:",totalError)
        return params
    # we can use an adaptive learning called Adam - Adaptive Moment Estimation.
    # Moment-based gradient descent was proposed, but has an issue of overshooting in a "valley".
    # Nesterov-Accelerated Gradient Descent makes movement from previous momentum and assigns
    # temporary gradients as a fix. Adam deals with adaptive learning rates and adaptive momentum.
    def adamRegression(self,points,learningRate,epochs=1000):
        params = [0,0]
        beta1,beta2 = 0.9,0.999
        eps = 10**-9
        mw,mb,vw,vb = 0,0,0,0
        for epoch in range(epochs):
            totalError = 0
            for i,point in enumerate(points):
                loss = (params[0]+point[0]*params[1]) - point[1]
                totalError += loss**2
                gradW = 2*point[0]*loss
                gradB = 2*loss
                mw = beta1*mw + (1-beta1)*gradW
                mb = beta1*mb + (1-beta1)*gradB
                vw = beta1*vw + (1-beta2)*gradW**2
                vb = beta1*vb + (1-beta2)*gradB**2
                mw_hat = mw/(1-(beta1**(i+1) ))
                mb_hat = mb/(1-(beta1**(i+1) ))
                vw_hat = vw/(1-(beta2**(i+1)))
                vb_hat = vb/(1-(beta2**(i+1)))
                params[0] = params[0] - (learningRate/(np.sqrt(vb_hat)+eps))*mb_hat
                params[1] = params[1] - (learningRate/(np.sqrt(vw_hat)+eps))*mw_hat
            print("Epoch:",epoch,"Error:",totalError)
        return params
    def getNeighbourhood(self,i,data,kNeighbors):
        currNeighbors = 0
        neighbors = np.array([])
        lower,upper = i,i
        while currNeighbors < kNeighbors:
            if lower == -1:
                neighbors = np.append(neighbors,[data.iloc[upper]])
                upper += 1
            elif upper == len(data):
                neighbors = np.append(neighbors,[data.iloc[lower]])
                lower -= 1
            else:
                lowerDist = abs(data.iloc[lower]-data.iloc[i])
                upperDist = abs(data.iloc[upper]-data.iloc[i])
                if lowerDist > upperDist:
                    neighbors = np.append(neighbors,[data.iloc[upper]])
                    upper += 1
                else:
                    neighbors = np.append(neighbors,[data.iloc[lower]])
                    lower -= 1
            currNeighbors += 1
        return neighbors
    def getMinMax(self,data,kNeighbors):
        maxI = []
        minI = []
        data = data['Adj Close']
        for i in range(len(data)):
            currNeighbors = 0
            neighbors = np.array([],dtype=float)
            lower,upper = i,i
            neighbors = self.getNeighbourhood(i,data,kNeighbors)
            meanNeighbors = np.average(neighbors)
            if data.iloc[i] > meanNeighbors:
                maxI.append(i)
            elif data.iloc[i] < meanNeighbors:
                minI.append(i)
        maxIds = set()
        minIds = set()
        for i in maxI:
            if i > kNeighbors and i < len(data)-kNeighbors:
                maxIds.add(data.iloc[i-kNeighbors:i+kNeighbors].idxmax())
        for j in minI:
            if j > kNeighbors and j < len(data)-kNeighbors:
                minIds.add(data.iloc[j-kNeighbors:j+kNeighbors].idxmin())
        return maxIds,minIds
    def olsParams(self,X,Y):
        smX = sm.add_constant(X)
        model = sm.OLS(Y,smX)
        results = model.fit()
        return results.pvalues,results.params
    def turnToDays(self,DateIndexValues,firstDate,form='%Y-%m-%dT%H:%M:%S.%f'):
        return list(map(lambda date: (datetime.strptime(str(date)[:-3],form)-firstDate).days,
                        DateIndexValues))
    def triangleContinuation(self,pattern,startDate,endDate,method="min-max",neighbors=6):
        dataPartition = self.data[(startDate <= self.data.index) & (self.data.index <= endDate)]
        firstDate = datetime.strptime(str(dataPartition.index.values[0])[:-3],
                                      '%Y-%m-%dT%H:%M:%S.%f')
        X = self.turnToDays(dataPartition.index.values,firstDate)
        if method == "high-low":
            highY = dataPartition["High"].values
            lowY = dataPartition["Low"].values
            highPValues, highParams = self.olsParams(X,highY)
            lowPValues, lowParams = self.olsParams(X,lowY)
            highLine = [highParams[1] * i + highParams[0] for i in X]
            lowLine = [lowParams[1] * i + highParams[0] for i in X]
            plt.plot(X,highY,'ro')
            plt.plot(X,lowY,'bo')
            plt.plot(X, highLine, 'r')
            plt.plot(X, lowLine, 'b')
            plt.title("Price vs. Days after {}".format(startDate))
            plt.ylabel("Price")
            plt.xlabel("Days after {}".format(startDate))
            plt.show()
            if pattern == 'ascending':
                return (highPValues[1] > 0.05 and highParams[1] == 0) and \
                       (lowPValues[1] <= 0.025 and lowParams[1] > 0)
            elif pattern == 'descending':
                return (lowPValues[1] > 0.05 and lowParams[1] == 0) and \
                       (highPValues[1] <= 0.025 and highParams[1] < 0)
            elif pattern == 'symmetrical':
                return (lowPValues[1] <= 0.05 and lowParams[1] > 0) and \
                       (highPValues[1] <= 0.025 and highParams[1] < 0)
            else:
                print("Not implemented.")
                return
        elif method == "min-max":
            maxIds,minIds = self.getMinMax(dataPartition,neighbors)
            notMax = set(dataPartition.index).difference(maxIds)
            notMinMax = list(notMax.difference(minIds))
            maxIds = list(maxIds)
            minIds = list(minIds)
            XMax = self.turnToDays(maxIds,firstDate,'%Y-%m-%d %H:%M')
            XMin = self.turnToDays(minIds,firstDate,'%Y-%m-%d %H:%M')
            YMax = list(dataPartition['Adj Close'].loc[maxIds])
            YMin = list(dataPartition['Adj Close'].loc[minIds])
            maxPValues, maxParams = self.olsParams(XMax,YMax)
            maxLine = [maxParams[1] * i + maxParams[0] for i in XMax]
            minPValues, minParams = self.olsParams(XMin,YMin)
            minLine = [minParams[1] * i + minParams[0] for i in XMin]
            plt.plot(maxIds, YMax,'ro',label='max')
            plt.plot(minIds, YMin,'bo',label='min')
            plt.plot(maxIds, maxLine, 'r')
            plt.plot(minIds, minLine, 'b')
            plt.plot(notMinMax, list(dataPartition['Adj Close'].loc[notMinMax]),'go',label='normal'
)
            plt.title("Min-Max Points in Continuation Patterns")
            plt.ylabel("Price")
            plt.xlabel("Date")
            plt.xticks(rotation=45)
            plt.legend()
            plt.show()
            if pattern == 'ascending':
                return (maxPValues[1] > 0.05 and maxParams[1] == 0) and \
                        (minPValues[1] <= 0.025 and minParams[1] > 0)
            elif pattern == 'descending':
                return (minPValues[1] > 0.05 and minParams[1] == 0) and \
                        (maxPValues[1] <= 0.025 and maxParams[1] < 0)
            elif pattern == 'symmetrical':
                return (minPValues[1] <= 0.05 and minParams[1] > 0) and \
                        (maxPValues[1] <= 0.025 and maxParams[1] < 0)
            else:
                print("Not implemented.")
                return
fxTriangles = ForexTriangles(lazardData,1)
fxTriangles.triangleContinuation('descending','2020-05-25','2020-08-01')
```

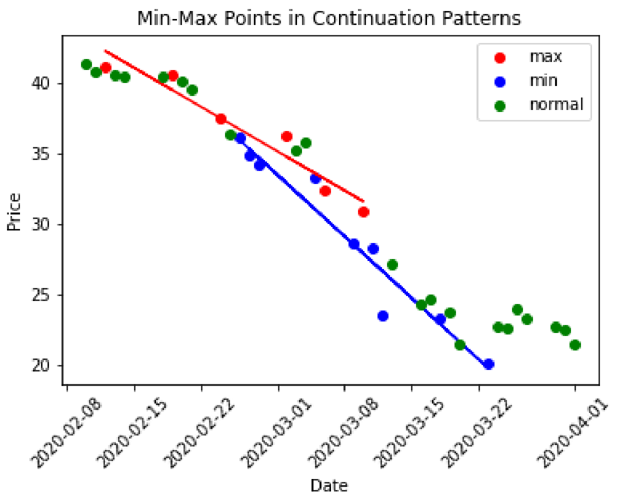


Out[19]:

```
False
```

In [20]:

```
fxTriangles.triangleContinuation('descending','2020-02-10','2020-04-01',method='min-max',neighbors
=5)
```

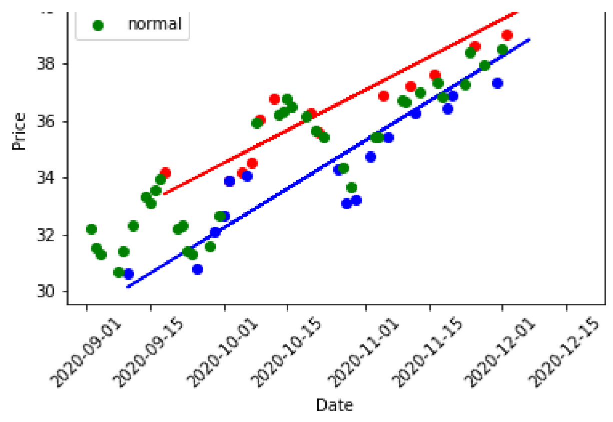


Out[20]:

```
False
```

In [21]:

```
fxTriangles.triangleContinuation('descending','2020-09-02','2020-12-20',method='min-max',neighbors
=5)
```



Out[21]:

```
False
```

Based on the linear regression done on the maximum and minimum points out of 5 neighbors, there appears to be price convergence with a descending triangle from adjusted closing prices from the end of May to start of July 2020. Although the red line plotted is negatively sloped, it's not statistically significant from 0 at the 10% alpha level. However, we can visually confirm the descending triangle continuation pattern and may take on a short position (sell stock then rebuy later on) following high volume breakdown, or downward movement of the stock price. Earlier in the year, there was a downward trend that didn't appear to follow general price convergence patterns. Towards the end of 2020, there was an upward trend with slight convergence of maximum and minimum prices.

A general candlestick plot is plotted and volatility analytics are done: A type of error bar/confidence interval-related statistical chart is Bollinger Bands. Bollinger Bands use N-period moving averages (MA), where the upper band is $MA + K\sigma$ and lower band is $MA - K\sigma$.

In [22]:

```
def plotBollingerBands(data,N=20,K=2):
    maData = data.rolling(N).mean()
    stdevData = data.rolling(N).std() # rolling standard deviations
    lowerBand = maData-(K*stdevData)
    upperBand = maData+(K*stdevData)
    return lowerBand,upperBand
```

In [23]:

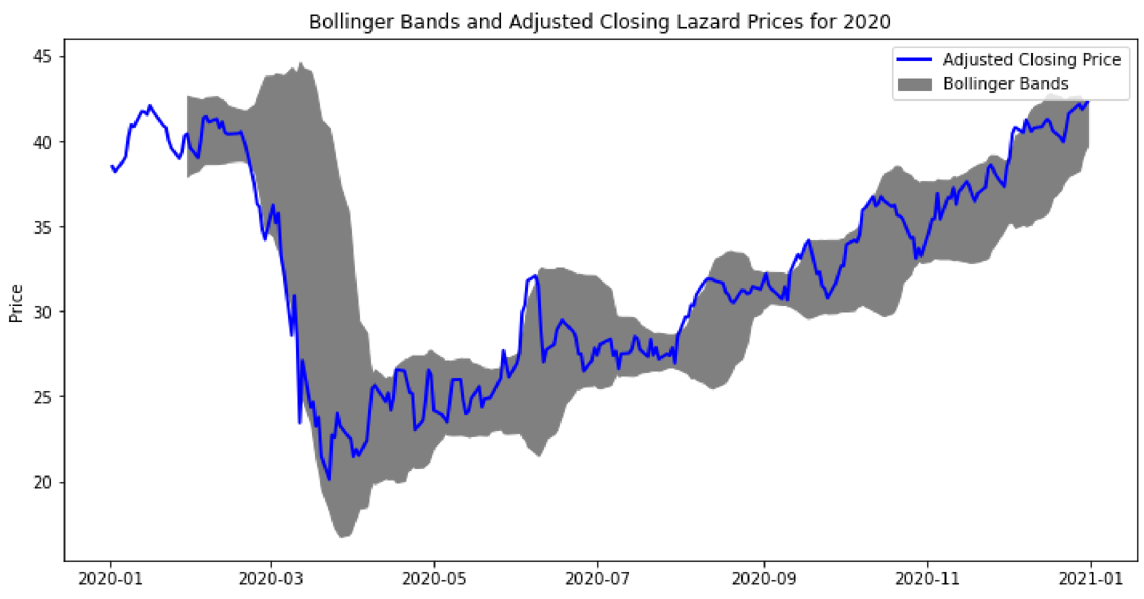
```
fig = go.Figure(data=[go.Candlestick(x=lazardData.index,
                open=lazardData['Open'],
                high=lazardData['High'],
                low=lazardData['Low'],
                close=lazardData['Close'])])

fig.update_layout(title="Lazard's Stock during 2020",
                  yaxis_title='LAZ Stock',
                  xaxis_title="Dates",
                 annotations=[dict(x='2020-01-21', xref='x',yref='paper',y=0.05,textangle=-90,
                                   showarrow=False, xanchor='left', text='COVID-19 First Reported')
                              dict(x='2020-03-11', xref='x',yref='paper',y=0.95,
                                   showarrow=False, xanchor='left', text='COVID-19 Declared Pandemi
'),
                              dict(text='2 Million COVID-19 Cases in US',x='2020-06-10',xref='x',
                                  yref='paper',y=0.75,showarrow=False, xanchor='left'),
                              dict(text='1st COVID-19 Drug Approved',x='2020-10-22',xref='x',
                                  yref='paper',y=0.05,showarrow=False,xanchor='left')],
                 shapes = [dict(x0='2020-01-21',x1='2020-01-21', y0=0,y1=1, xref='x',yref='paper',l
ne_width=2),
                            dict(x0='2020-03-11',x1='2020-03-11', y0=0,y1=1, xref='x',yref='paper',l
ne_width=2),
                            dict(x0='2020-06-10',x1='2020-06-10', y0=0,y1=1, xref='x',yref='paper',l
ne_width=2),
                            dict(x0='2020-10-22',x1='2020-10-22', y0=0,y1=1, xref='x',yref='paper',l
ne_width=2)])
```

```
fig.show()
```

As expected, after the onset of COVID-19, the stock market went into a temporary bear market. This was a sudden global stock market crash called the 2020 stock market crash. An interesting pattern prior to the 2 Million COVID-19 Cases in US was the evening star, where we see a green bar followed by a narrower red bar. Fresh buyers failed to show up. This bearish pattern indicates future declines in prices, which was what happened as COVID-19 cases increased and 2 Million cases were reported.

In [24]:

```
lowerBand,upperBand = plotBollingerBands(lazardData['Adj Close'])
fig = plt.figure(figsize=(12,6))
ax = fig.add_subplot(111)
ax.fill_between(lazardData.index, lowerBand, upperBand, color='grey',label='Bollinger Bands')
ax.plot(lazardData.index, lazardData['Adj Close'], color='blue', lw=2,label='Adjusted Closing
Price')
plt.title("Bollinger Bands and Adjusted Closing Lazard Prices for 2020")
plt.ylabel("Price")
plt.legend()
plt.show()
```

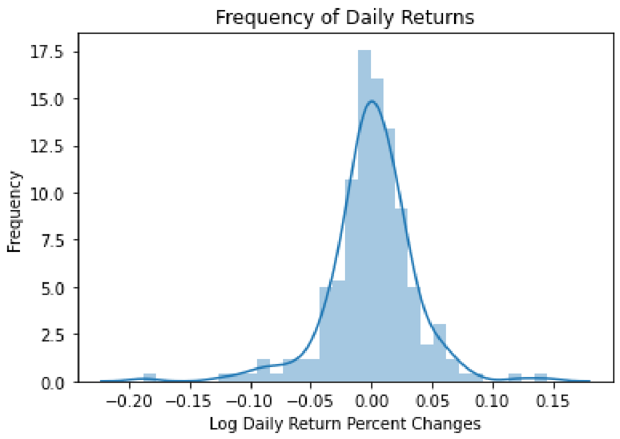


Based on the Bollinger Bands, the price doesn't appear to be extremely volatile given the closing prices fall within the range. Among various interpretations, one is that stocks may be bought when the price touches the lower bollinger band such as late February 2020.

Simulating Adjusted Closing Prices

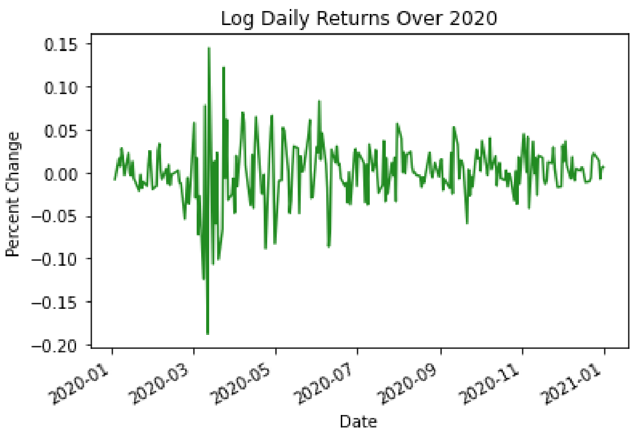
In [25]:

```
logReturns = np.log(1+lazardData['Adj Close'].pct_change())[1:]
sns.distplot(logReturns)
plt.xlabel("Log Daily Return Percent Changes")
plt.ylabel("Frequency")
plt.title("Frequency of Daily Returns")
plt.show()
```



In [26]:

```
logReturns.plot(color="forestgreen")
plt.title("Log Daily Returns Over 2020")
plt.ylabel("Percent Change")
plt.show()
```



Compute Drift - when predicting the daily return of the stock, we use brownian motion as a stochastic process for modeling random behavior over time. We have that

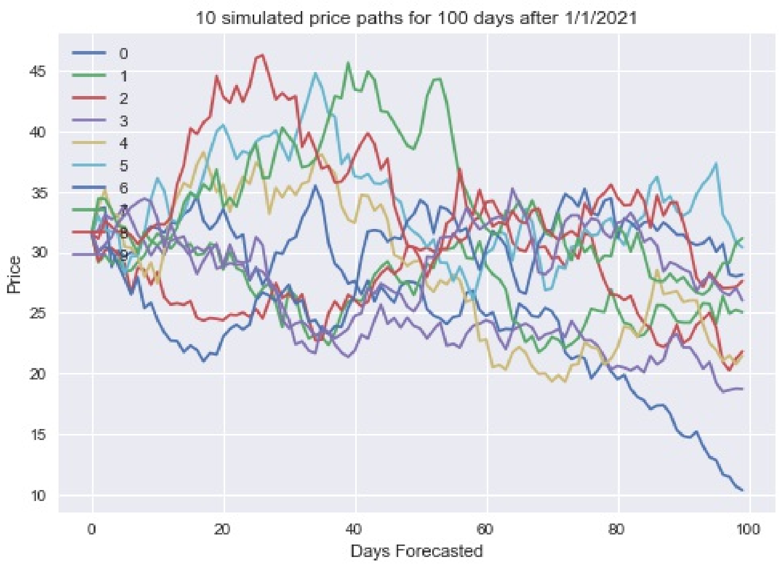
$$drift = \mu - \frac{1}{2}\sigma^2, volatility = \sigma Z[rand(0, 1)], r = (\mu - \frac{1}{2}\sigma) + \sigma Z[rand(0, 1)], S_t = S_{t-1} * e^{(\mu - \frac{1}{2}\sigma^2) + \sigma Z[Rand(0,1)]}$$

In [27]:

```
import time
import pandas as pd
import numpy as np
from scipy.stats import norm
import matplotlib
import matplotlib.pyplot as plt


def plotMCPricePaths(data,days,trials):
    logReturns = np.log(1+data['Adj Close'].pct_change())[1:]
    mu = logReturns.mean()
    var = logReturns.var()
    drift = mu-(0.5*var)

    stdev = logReturns.std()
    Z = norm.ppf(np.random.rand(days,trials))
    dailyReturns = np.exp(drift+stdev*Z)
    # we have 50 random variables for each day from every one of the 10,000 trials.
    pricePaths = np.zeros((days,trials))
    pricePaths[0] = data['Adj Close'].iloc[-1]
    for t in range(1,days):
        # S_t = S_{t-1}*e^r
        pricePaths[t] = pricePaths[t-1] * dailyReturns[t]
    paths = pd.DataFrame(pricePaths)
    return paths
```

In [28]:

```
%%time
bigLazardData = web.DataReader('LAZ',data_source='yahoo',start='1/1/2020',end='9/3/2020')
days = 100
paths = plotMCPricePaths(bigLazardData,days,200000)
matplotlib.style.use('seaborn')
paths[range(10)].plot()
plt.title("10 simulated price paths for {} days after 1/1/2021".format(days))
plt.xlabel("Days Forecasted")
plt.ylabel("Price")
plt.legend(loc='upper left')
plt.show()
```

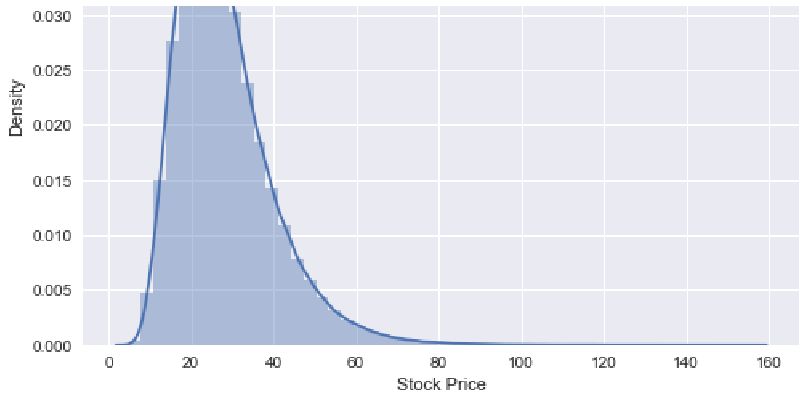


```
CPU times: user 2.52 s, sys: 964 ms, total: 3.48 s
Wall time: 3.94 s
```

In [29]:

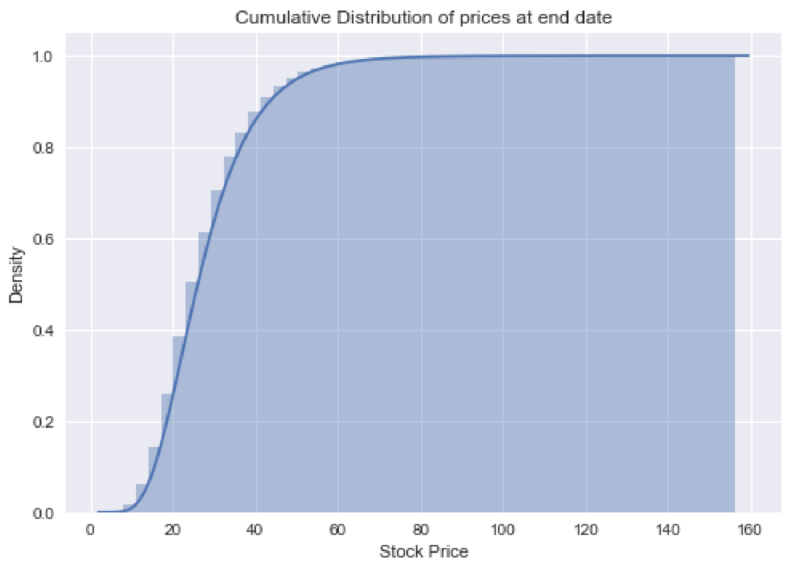
```
sns.distplot(paths.iloc[-1])
plt.title("Distribution of prices at end date")
plt.xlabel("Stock Price")
plt.show()
```

Distribution of prices at end date
0.040
0.035



In [30]:

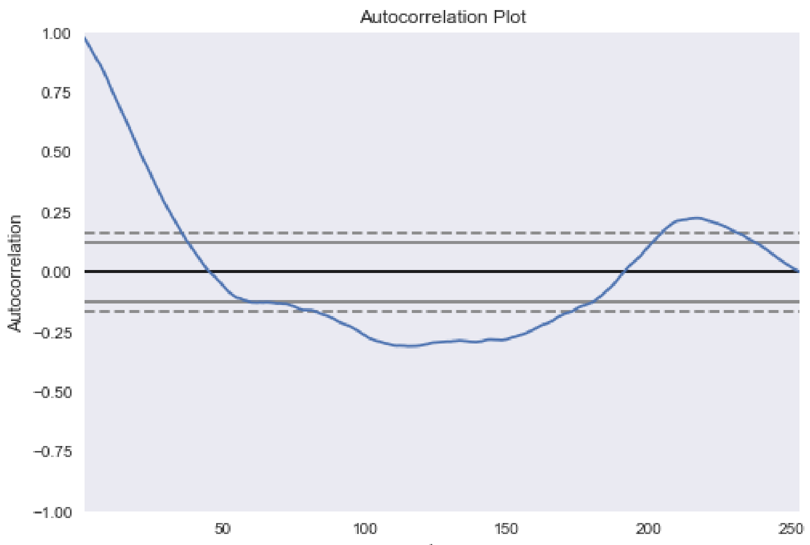
```
sns.distplot(paths.iloc[-1], hist_kws={'cumulative':True},kde_kws={'cumulative':True})
plt.title("Cumulative Distribution of prices at end date")
plt.xlabel("Stock Price")
plt.show()
```



Auto-regressive Moving Average Model

In [31]:

```
autocorrelation_plot(lazardData["Adj Close"])
plt.title("Autocorrelation Plot")
plt.show()
```



Lag

In [32]:

```
arimaLazard = sm.tsa.ARMA(lazardData['Adj Close'], order=(10,1,0))
arimaLazardFit = arimaLazard.fit()
```

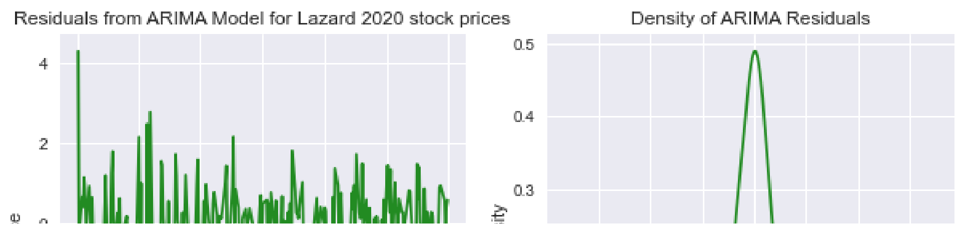
In [33]:

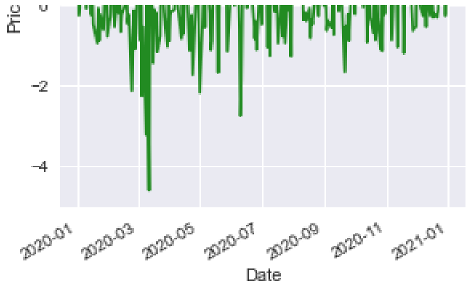
```
print(arimaLazardFit.summary())
```

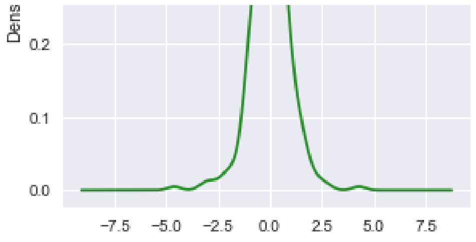
```
                              ARMA Model Results
==============================================================================
Dep. Variable:              Adj Close   No. Observations:                  253
Model:                    ARMA(10, 1)   Log Likelihood                -340.009
Method:                       css-mle   S.D. of innovations              0.919
Date:                Wed, 27 Jan 2021   AIC                            706.018
Time:                        23:40:02   BIC                            751.952
Sample:                             0   HQIC                           724.499

====================================================================================
                       coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------------
const               34.2126      3.414     10.021      0.000      27.521      40.904
ar.L1.Adj Close      0.8723      0.328      2.663      0.008       0.230       1.514
ar.L2.Adj Close      0.1754      0.322      0.544      0.586      -0.456       0.807
ar.L3.Adj Close     -0.0285      0.090     -0.318      0.751      -0.204       0.147
ar.L4.Adj Close     -0.0978      0.083     -1.174      0.240      -0.261       0.065
ar.L5.Adj Close      0.0399      0.084      0.473      0.637      -0.125       0.205
ar.L6.Adj Close      0.0128      0.080      0.160      0.873      -0.144       0.169
ar.L7.Adj Close      0.2812      0.079      3.575      0.000       0.127       0.435
ar.L8.Adj Close     -0.3002      0.123     -2.448      0.014      -0.540      -0.060
ar.L9.Adj Close      0.1576      0.142      1.110      0.267      -0.121       0.436
ar.L10.Adj Close    -0.1298      0.067     -1.932      0.053      -0.261       0.002
ma.L1.Adj Close      0.1033      0.328      0.315      0.753      -0.539       0.746
                                    Roots
==============================================================================
                  Real          Imaginary           Modulus         Frequency
------------------------------------------------------------------------------
AR.1           -1.0112           -0.4361j            1.1013           -0.4352
AR.2           -1.0112           +0.4361j            1.1013            0.4352
AR.3            1.0272           -0.0000j            1.0272           -0.0000
AR.4            1.1379           -0.0000j            1.1379           -0.0000
AR.5            0.8271           -0.8540j            1.1888           -0.1275
AR.6            0.8271           +0.8540j            1.1888            0.1275
AR.7           -0.4743           -1.1350j            1.2301           -0.3130
AR.8           -0.4743           +1.1350j            1.2301            0.3130
AR.9            0.1831           -1.5836j            1.5941           -0.2317
AR.10           0.1831           +1.5836j            1.5941            0.2317
MA.1           -9.6812           +0.0000j            9.6812            0.5000
------------------------------------------------------------------------------
```

In [34]:

```
fig, ax = plt.subplots(1,2)
fig.set_size_inches(10,5)
arimaLazardFit.resid.plot(title="Residuals from ARIMA Model for Lazard 2020 stock prices",
                          color="forestgreen",ylabel="Price",
                          ax=ax[0])
arimaLazardFit.resid.plot(kind='kde',
                          title='Density of ARIMA Residuals',
                          color='forestgreen',
                          ax=ax[1])
plt.show()
```







Based on the initial ARIMA model, residuals appear to be centered around 0, but a considerable number of them are above or below 0.

In [35]:

```
# Grid Search on the best hyperparameters.
minMAE = float('inf')
bestTup = None
for p in range(1,4):
    for d in range(1,4):
        for q in range(1,4):
            arimaLazardBack = sm.tsa.ARMA(lazardData['Adj Close'][:-15], order=(p,d,q))
            arimaLazardBackFit = arimaLazardBack.fit()
            forecasted, _, _ = arimaLazardBackFit.forecast(15, alpha=0.05)
            currMAE = metrics.mean_absolute_error(lazardData['Adj Close'][-15:],forecasted)
            if currMAE < minMAE:
                minMAE = currMAE
                bestTup = (p,d,q)
print(bestTup)
```

```
(2, 1, 1)
```

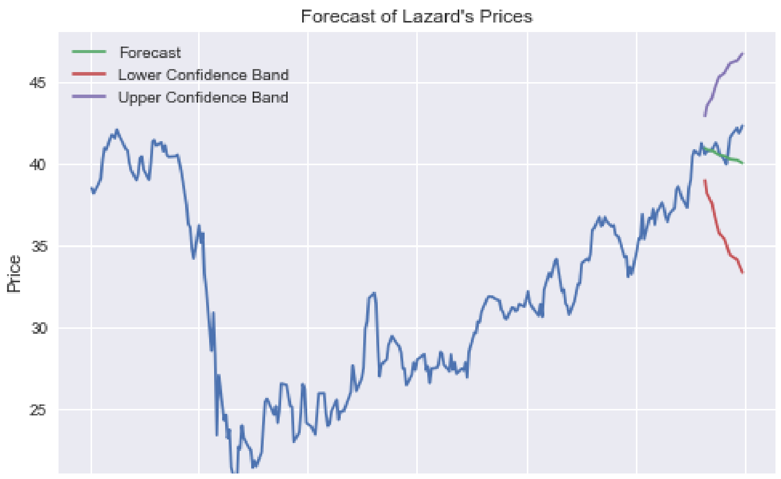
In [36]:

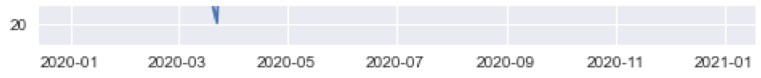
```
print(minMAE)
```

```
0.8222870784446423
```

In [37]:

```
arimaLazardBack = sm.tsa.ARMA(lazardData['Adj Close'][:-15], order=(2,1,1))
arimaLazardBackFit = arimaLazardBack.fit()
forecasted, _, conf = arimaLazardBackFit.forecast(15, alpha=0.05)
plt.plot(lazardData.index,lazardData['Adj Close'])
plt.plot(lazardData.index[-15:],forecasted,label='Forecast')
plt.plot(lazardData.index[-15:],conf[:, 0],label='Lower Confidence Band')
plt.plot(lazardData.index[-15:],conf[:,1], label='Upper Confidence Band')
plt.title("Forecast of Lazard's Prices")
plt.ylabel("Price")
plt.legend()
plt.show()
```





Given the erratic and unpredictable of stock prices, we could focus on the fact that forecasts inform us about general directions in the price. Also, the price is within the 95% confidence bands as shown, which is a good sign. The forecast matches the general direction of the price decreasing later on.

LSTM model for stock price predictions

In [38]:

```
dataScaler = MinMaxScaler(feature_range=(-1, 1))
# important to normalize the data in time series predictions
def splitTrainTest(trainProportion,data,timeStepsBack):
    trainSize = int(len(data)*trainProportion)
    XTrain,XTest,YTrain,YTest = [],[],[],[]
    train = data['Adj Close'].iloc[:trainSize].values
    test = data['Adj Close'].iloc[trainSize:].values
    train = dataScaler.fit_transform(train.reshape(-1,1))
    test = dataScaler.fit_transform(test.reshape(-1,1))
    for i in range(len(train)):
        if i >= timeStepsBack:
            XTrain.append(np.array(train[i-timeStepsBack:i]))
            YTrain.append(train[i])
    for j in range(len(test)):
        if j >= timeStepsBack:
            XTest.append(np.array(test[j-timeStepsBack:j]))
            YTest.append(test[j])
    return np.array(XTrain),np.array(XTest),np.array(YTrain),np.array(YTest)
```

In [39]:

```
def reshapeArray(X):
    return np.reshape(X, (X.shape[0], 1, X.shape[1]))
```

Here we have keras:

In [40]:

```
# structing X,Y data as X is over N time steps then Y is the next one.
def makeLSTMModel(data,timeStepsBack):
    XTrain,XTest,YTrain,YTest = splitTrainTest(0.75,data,timeStepsBack)
    XTrain = reshapeArray(XTrain)
    XTest = reshapeArray(XTest)
    model = Sequential()
    model.add(LSTM(20,input_shape=(1, timeStepsBack)))
    model.add(Dropout(0.2))
    model.add(Dense(50,activation='relu'))
    model.add(Dropout(0.2))
    model.add(Dense(1))
    model.compile(loss='mean_squared_error', optimizer='adam')
    model.fit(XTrain, YTrain, epochs=90, batch_size=1, verbose=2)
    model.summary()
    return model,XTest,YTest
model,XTest,YTest = makeLSTMModel(lazardData,3)
```

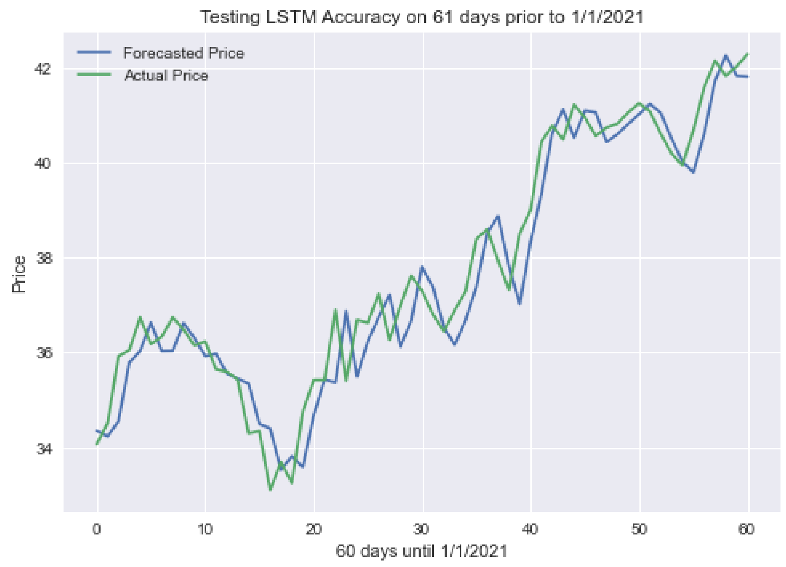
```
Epoch 1/90
186/186 - 1s - loss: 0.1386
Epoch 2/90
186/186 - 0s - loss: 0.0329
Epoch 3/90
186/186 - 0s - loss: 0.0308
Epoch 4/90
186/186 - 0s - loss: 0.0260
Epoch 5/90
186/186 - 0s - loss: 0.0237
Epoch 6/90
186/186 - 0s - loss: 0.0239
```

```
186/186 - 0s - loss: 0.0259
Epoch 7/90
186/186 - 0s - loss: 0.0234
Epoch 8/90
186/186 - 0s - loss: 0.0177
Epoch 9/90
186/186 - 0s - loss: 0.0192
Epoch 10/90
186/186 - 0s - loss: 0.0199
Epoch 11/90
186/186 - 0s - loss: 0.0191
Epoch 12/90
186/186 - 0s - loss: 0.0196
Epoch 13/90
186/186 - 0s - loss: 0.0171
Epoch 14/90
186/186 - 0s - loss: 0.0175
Epoch 15/90
186/186 - 0s - loss: 0.0216
Epoch 16/90
186/186 - 0s - loss: 0.0188
Epoch 17/90
186/186 - 0s - loss: 0.0196
Epoch 18/90
186/186 - 0s - loss: 0.0150
Epoch 19/90
186/186 - 0s - loss: 0.0187
Epoch 20/90
186/186 - 0s - loss: 0.0167
Epoch 21/90
186/186 - 0s - loss: 0.0168
Epoch 22/90
186/186 - 0s - loss: 0.0189
Epoch 23/90
186/186 - 0s - loss: 0.0156
Epoch 24/90
186/186 - 0s - loss: 0.0164
Epoch 25/90
186/186 - 0s - loss: 0.0172
Epoch 26/90
186/186 - 0s - loss: 0.0194
Epoch 27/90
186/186 - 0s - loss: 0.0151
Epoch 28/90
186/186 - 0s - loss: 0.0231
Epoch 29/90
186/186 - 0s - loss: 0.0175
Epoch 30/90
186/186 - 0s - loss: 0.0197
Epoch 31/90
186/186 - 0s - loss: 0.0181
Epoch 32/90
186/186 - 0s - loss: 0.0171
Epoch 33/90
186/186 - 0s - loss: 0.0186
Epoch 34/90
186/186 - 0s - loss: 0.0145
Epoch 35/90
186/186 - 0s - loss: 0.0204
Epoch 36/90
186/186 - 0s - loss: 0.0168
Epoch 37/90
186/186 - 0s - loss: 0.0165
Epoch 38/90
186/186 - 0s - loss: 0.0142
Epoch 39/90
186/186 - 0s - loss: 0.0145
Epoch 40/90
186/186 - 0s - loss: 0.0171
Epoch 41/90
186/186 - 0s - loss: 0.0181
Epoch 42/90
186/186 - 0s - loss: 0.0177
Epoch 43/90
186/186 - 0s - loss: 0.0181
Epoch 44/90
186/186 - 0s - loss: 0.0162
Epoch 45/90
Epoch 45/90
186/186 - 0s - loss: 0.0168
Epoch 46/90
186/186 - 0s - loss: 0.0189
Epoch 47/90
186/186 - 0s - loss: 0.0200
Epoch 48/90
186/186 - 0s - loss: 0.0142
Epoch 49/90
186/186 - 0s - loss: 0.0165
Epoch 50/90
186/186 - 0s - loss: 0.0187
Epoch 51/90
186/186 - 0s - loss: 0.0184
Epoch 52/90
186/186 - 0s - loss: 0.0163
Epoch 53/90
186/186 - 0s - loss: 0.0169
Epoch 54/90
186/186 - 0s - loss: 0.0171
Epoch 55/90
186/186 - 0s - loss: 0.0160
Epoch 56/90
186/186 - 0s - loss: 0.0172
Epoch 57/90
186/186 - 0s - loss: 0.0160
Epoch 58/90
186/186 - 0s - loss: 0.0143
Epoch 59/90
186/186 - 0s - loss: 0.0170
Epoch 60/90
186/186 - 0s - loss: 0.0166
Epoch 61/90
186/186 - 0s - loss: 0.0169
Epoch 62/90
186/186 - 0s - loss: 0.0149
Epoch 63/90
186/186 - 0s - loss: 0.0193
Epoch 64/90
186/186 - 0s - loss: 0.0151
Epoch 65/90
186/186 - 0s - loss: 0.0135
Epoch 66/90
186/186 - 0s - loss: 0.0160
Epoch 67/90
186/186 - 0s - loss: 0.0209
Epoch 68/90
186/186 - 0s - loss: 0.0195
Epoch 69/90
186/186 - 0s - loss: 0.0168
Epoch 70/90
186/186 - 0s - loss: 0.0164
Epoch 71/90
186/186 - 0s - loss: 0.0141
Epoch 72/90
186/186 - 0s - loss: 0.0166
Epoch 73/90
186/186 - 0s - loss: 0.0151
Epoch 74/90
186/186 - 0s - loss: 0.0149
Epoch 75/90
186/186 - 0s - loss: 0.0153
Epoch 76/90
186/186 - 0s - loss: 0.0148
Epoch 77/90
186/186 - 0s - loss: 0.0172
Epoch 78/90
186/186 - 0s - loss: 0.0155
Epoch 79/90
186/186 - 0s - loss: 0.0157
Epoch 80/90
186/186 - 0s - loss: 0.0161
Epoch 81/90
186/186 - 0s - loss: 0.0181
Epoch 82/90
186/186 - 0s - loss: 0.0158
Epoch 83/90
186/186 - 0s - loss: 0.0169
```

```
186/186 - 0s - loss: 0.0169
Epoch 84/90
186/186 - 0s - loss: 0.0154
Epoch 85/90
186/186 - 0s - loss: 0.0148
Epoch 86/90
186/186 - 0s - loss: 0.0155
Epoch 87/90
186/186 - 0s - loss: 0.0153
Epoch 88/90
186/186 - 0s - loss: 0.0153
Epoch 89/90
186/186 - 0s - loss: 0.0139
Epoch 90/90
186/186 - 0s - loss: 0.0164
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
lstm (LSTM)                  (None, 20)                1920      
_________________________________________________________________
dropout (Dropout)            (None, 20)                0         
_________________________________________________________________
dense (Dense)                (None, 50)                1050      
_________________________________________________________________
dropout_1 (Dropout)          (None, 50)                0         
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 51        
=================================================================
Total params: 3,021
Trainable params: 3,021
Non-trainable params: 0
_________________________________________________________________
```

In [41]:

```
plt.plot(dataScaler.inverse_transform(model.predict(XTest)),label='Forecasted Price')
plt.plot(dataScaler.inverse_transform(YTest),label='Actual Price')
plt.title("Testing LSTM Accuracy on {} days prior to 1/1/2021".format(len(YTest)))
plt.xlabel("60 days until 1/1/2021")
plt.ylabel("Price")
plt.legend()
plt.show()
```



In [42]:

```
forecasted = model.predict(XTest)
actual = YTest
mape = (1/len(actual))*(np.sum(abs((actual-forecasted)/actual)))
print("LSTM mean absolute percent error: {}%".format(mape))
rmse = np.sqrt((1/len(actual))*(np.sum((actual-forecasted)**2 )))
print("LSTM root mean-squared error: {}".format(rmse))
```

```
LSTM mean absolute percent error: 0.6809432038478334%
LSTM root mean-squared error: 0.14422690491431886
```

In Pytorch:

In [43]:

```
# Here we use tensors of the raw data to be fed into the LSTM

class LSTMStock(nn.Module):
    def __init__(self, numClasses, inSize, hiddenSize, numLayers):
        super(LSTMStock,self).__init__()
        self.numClasses=numClasses
        self.hiddenSize=hiddenSize
        self.numLayers = numLayers
        self.lstm = nn.LSTM(input_size=inSize,
                            hidden_size=hiddenSize,
                            num_layers=numLayers,
                            dropout=0.2,batch_first=True)
        self.fullyConnected = nn.Linear(hiddenSize,numClasses)
    def forward(self,inputVec):
        initialHiddenState = Variable(torch.zeros(self.numLayers, inputVec.size(0), self.hiddenSize
))

        initialCellState = Variable(torch.zeros(self.numLayers, inputVec.size(0), self.hiddenSize))
        _, (outHiddenState, _) = self.lstm(inputVec, (initialHiddenState, initialCellState))
        outHiddenState = outHiddenState.view(-1, self.hiddenSize)
        return self.fullyConnected(outHiddenState)
```

In [44]:

```
def tensorData(trainProportion, data, timeStepsBack):
    XTrain,XTest,YTrain,YTest = splitTrainTest(trainProportion,data,timeStepsBack)
    XTrain = Variable(torch.Tensor(XTrain))
    XTest = Variable(torch.Tensor(XTest))
    YTrain = Variable(torch.Tensor(YTrain))
    YTest = Variable(torch.Tensor(YTest))
    return XTrain,XTest,YTrain,YTest
XTrain,XTest,YTrain,YTest = tensorData(0.75, lazardData, 3)
```

In [45]:

```
def trainLSTMPytorch(XTrain,YTrain, epochs=1000,learningRate=0.01):
    lstmLazard = LSTMStock(1, 1, 3, 1)
    lossFunction = torch.nn.MSELoss()
    adamOptimizer = torch.optim.Adam(lstmLazard.parameters(),
                                     lr=learningRate)
    for epoch in range(1,epochs+1):
        output = lstmLazard(XTrain)
        adamOptimizer.zero_grad()
        loss = lossFunction(output, YTrain)
        # teacher forcing
        loss.backward()
        adamOptimizer.step()
        if epoch % 10 == 0:
            print("Epoch:",epoch,"Loss:",loss.item())
    return lstmLazard
```

In [46]:

```
lstmLazard = trainLSTMPytorch(XTrain,YTrain)
```

```
Epoch: 10 Loss: 0.40738940238952637
Epoch: 20 Loss: 0.3230709135532379
Epoch: 30 Loss: 0.2852208614349365
Epoch: 40 Loss: 0.26259005069732666
Epoch: 50 Loss: 0.23947089910507202
Epoch: 60 Loss: 0.19791144132614136
Epoch: 70 Loss: 0.11706162244081497
Epoch: 80 Loss: 0.03464185446500778
Epoch: 90 Loss: 0.027044227346777916
```

```
Epoch: 100 Loss: 0.02471667155623436
Epoch: 110 Loss: 0.020003562793135643
Epoch: 120 Loss: 0.019156452268362045
Epoch: 130 Loss: 0.017337989062070847
Epoch: 140 Loss: 0.0161819439381361
Epoch: 150 Loss: 0.015064094215631485
Epoch: 160 Loss: 0.01416902244091034
Epoch: 170 Loss: 0.013373675756156445
Epoch: 180 Loss: 0.012707575224339962
Epoch: 190 Loss: 0.012154310941696167
Epoch: 200 Loss: 0.0116957388818264
Epoch: 210 Loss: 0.011318718083202839
Epoch: 220 Loss: 0.011008888483047485
Epoch: 230 Loss: 0.010753553360700607
Epoch: 240 Loss: 0.010542608797550201
Epoch: 250 Loss: 0.010367287322878838
Epoch: 260 Loss: 0.010220315307378769
Epoch: 270 Loss: 0.010095643810927868
Epoch: 280 Loss: 0.009988384321331978
Epoch: 290 Loss: 0.00989473145455122
Epoch: 300 Loss: 0.0098118269816041
Epoch: 310 Loss: 0.009737606160342693
Epoch: 320 Loss: 0.009670608676970005
Epoch: 330 Loss: 0.009609827771782875
Epoch: 340 Loss: 0.009554548189043999
Epoch: 350 Loss: 0.009504244662821293
Epoch: 360 Loss: 0.00945849996060133
Epoch: 370 Loss: 0.009416949935257435
Epoch: 380 Loss: 0.009379264898598194
Epoch: 390 Loss: 0.009345123544335365
Epoch: 400 Loss: 0.009314213879406452
Epoch: 410 Loss: 0.009286229498684406
Epoch: 420 Loss: 0.00926087237894535
Epoch: 430 Loss: 0.009237856604158878
Epoch: 440 Loss: 0.00921691581606865
Epoch: 450 Loss: 0.009197797626256943
Epoch: 460 Loss: 0.009180277585983276
Epoch: 470 Loss: 0.009164145216345787
Epoch: 480 Loss: 0.009149221703410149
Epoch: 490 Loss: 0.009135340340435505
Epoch: 500 Loss: 0.009122364223003387
Epoch: 510 Loss: 0.00911017321050167
Epoch: 520 Loss: 0.009098662063479424
Epoch: 530 Loss: 0.00908774882555008
Epoch: 540 Loss: 0.009077351540327072
Epoch: 550 Loss: 0.0090674152597785
Epoch: 560 Loss: 0.00905788503587246
Epoch: 570 Loss: 0.009048717096447945
Epoch: 580 Loss: 0.00903987791389227
Epoch: 590 Loss: 0.00903133675456047
Epoch: 600 Loss: 0.009023066610097885
Epoch: 610 Loss: 0.009015049785375595
Epoch: 620 Loss: 0.009007265791296959
Epoch: 630 Loss: 0.008999698795378208
Epoch: 640 Loss: 0.008992337621748447
Epoch: 650 Loss: 0.008985171094536781
Epoch: 660 Loss: 0.008978188037872314
Epoch: 670 Loss: 0.008971380069851875
Epoch: 680 Loss: 0.008964740671217442
Epoch: 690 Loss: 0.008958266116678715
Epoch: 700 Loss: 0.008951946161687374
Epoch: 710 Loss: 0.00894577894359827
Epoch: 720 Loss: 0.008939757011830807
Epoch: 730 Loss: 0.008933881297707558
Epoch: 740 Loss: 0.00892814714461565
Epoch: 750 Loss: 0.008922548964619637
Epoch: 760 Loss: 0.00891708955168724
Epoch: 770 Loss: 0.00891176052391529
Epoch: 780 Loss: 0.00890656840056181
Epoch: 790 Loss: 0.008901502937078476
Epoch: 800 Loss: 0.008896569721400738
Epoch: 810 Loss: 0.008891761302947998
Epoch: 820 Loss: 0.00888708233833313
Epoch: 830 Loss: 0.008882525376975536
Epoch: 840 Loss: 0.008878096006810665
Epoch: 850 Loss: 0.008873787708580494
Epoch: 860 Loss: 0.008869597688317299
```

```
Epoch: 870 Loss: 0.008865526877343655
Epoch: 880 Loss: 0.008861570619046688
Epoch: 890 Loss: 0.008857730776071548
Epoch: 900 Loss: 0.00885399803519249
Epoch: 910 Loss: 0.008850373327732086
Epoch: 920 Loss: 0.00884685106575489
Epoch: 930 Loss: 0.008843428455293179
Epoch: 940 Loss: 0.008840099908411503
Epoch: 950 Loss: 0.00883686263114214
Epoch: 960 Loss: 0.008833709172904491
Epoch: 970 Loss: 0.008830636739730835
Epoch: 980 Loss: 0.008827641606330872
Epoch: 990 Loss: 0.00882471539080143
Epoch: 1000 Loss: 0.008821853436529636
```

In [47]:

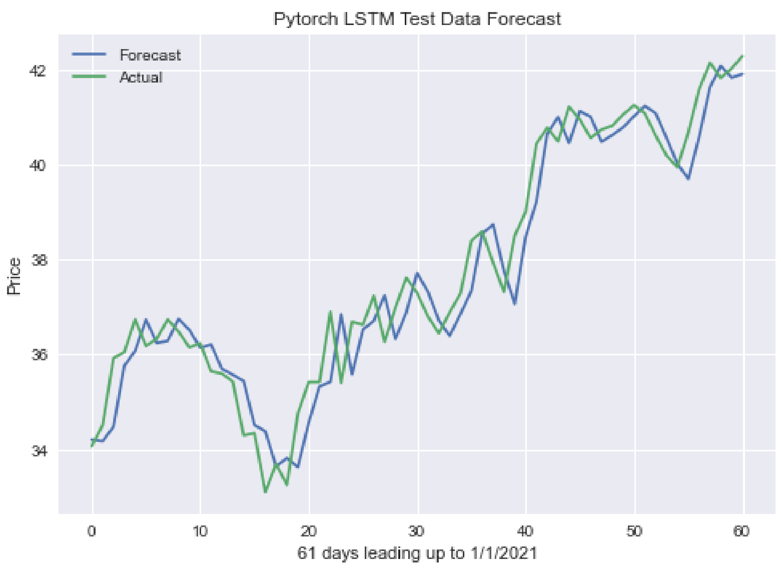
```
forecasted = lstmLazard(XTest)
predicted = dataScaler.inverse_transform(forecasted.data.numpy())
```

In [48]:

```
truePrice = dataScaler.inverse_transform(YTest)
```

In [49]:

```
plt.plot(predicted,label="Forecast")
plt.plot(truePrice,label="Actual")
plt.title("Pytorch LSTM Test Data Forecast")
plt.ylabel("Price")
plt.xlabel("{} days leading up to 1/1/2021".format(len(YTest)))
plt.legend()
plt.show()
```



In [50]:

```
mape = (1/len(truePrice))*(np.sum(abs((truePrice-predicted)/truePrice)))
print("LSTM Pytorch mean absolute percent error: {}%".format(mape))
rmse = np.sqrt((1/len(truePrice))*(np.sum((truePrice-predicted)**2 )))
print("LSTM Pytorch root mean-squared error: {}".format(rmse))
```

```
LSTM Pytorch mean absolute percent error: 0.014461386179170328%
LSTM Pytorch root mean-squared error: 0.6762026526779622
```

Long-short term memory neural networks is a type of gated recurrent neural network architecture. This resolves the issue of x values or gradients growing extremely large or small based on the recurrent nature of the weights in RNNs. And as shown above, LSTMs are good at identifying patterns in time series (data with an ordering) and follow out-of-sample price

identifying patterns in time series (data with an ordering) and follow out-of-sample price patterns quite closely.

The NASDAQ Financial-100 is an index that's composed of financial service stocks. Out of the 100 stocks, we can look for the most important components driving stock price movements of the index with PCA, specifically during 2020.

In [51]:

```
tags = ['AGNC','ANAT','ABCB','ACGL','AUB',
        'OZK','BANR','BOKF','BHF','CFFN',
        'CTRE','CATY','CINF','CME','CIGI',
        'COLB','CLBK','CBSH','CSGP','CACC',
        'CVBF','CONE','EWBC','EHTH','ESGR',
        'EQIX','ERIE','FITB','FCNCA','FFBC',
        'FFIN','FHB','FRME','FMBI','FCFS',
        'FSV','FULT','GLPI','GBCI','HLNE',
        'HWC','HOMB','HBAN','INDB','IBTX',
        'IBKR','IBOC','ISBC','KNSL','LAMR',
        'TREE','LPLA','MKTX','NDAQ','NAVI',
        'NTRS','ONB','PPBI','PACW','PBCT',
        'PNFP','BPOP','PCH','PFG','RDFN',
        'REG','RNST','ROIC','SBRA','SBAC',
        'SEIC','SIGI','SFBS','SBNY','SFNC',
        'SLM','SSB','SIVB','TROW','TCF',
        'TCBI','TFSL','CG','TOWN','TRMK',
        'UMBF','UMPQ','UBSI','UCBI','UNIT',
        'VLY','VIRT','WAFD','WSBC','WABC',
        'WTFC','WSFS','XP','ZG','ZION']
```

Let's compare the runtimes of a sequential, then a parallel implementation.

In [52]:

```
%%time
def readTag(tag,startDate,endDate):
    data = web.DataReader(tag,data_source='yahoo',start=startDate,end=endDate)
    data = data["Adj Close"]
    return data
def getPriceFromList(lst,startDate='1/1/2020',endDate='1/1/2021'):
    allData = pd.DataFrame()
    for tag in lst:
        data = readTag(tag,startDate,endDate)
        allData[tag] = data
    return allData

allData = getPriceFromList(tags)
```

```
CPU times: user 5.01 s, sys: 401 ms, total: 5.42 s
Wall time: 48 s
```

In [53]:

```
%%time

class dataCollector:
    def __init__(self,allData):
        self.allData = allData
    def readTag(self,tag,startDate,endDate):
        data = web.DataReader(tag,data_source='yahoo',start=startDate,end=endDate)
        data = data["Adj Close"]
        self.allData[tag] = data
    def getPriceFromList(self,lst,startDate='1/1/2020',endDate='1/1/2021'):
        _ = Parallel(n_jobs=10,prefer="threads")(delayed(self.readTag)(tag,startDate,endDate)\
                                                 for tag in lst)
allData = pd.DataFrame()
dc = dataCollector(allData)
dc.getPriceFromList(tags)
```

```
CPU times: user 4.27 s, sys: 429 ms, total: 4.7 s
Wall time: 13.7 s
```

Joblib does a good job parallelizing the data acquisition. We go from 55.7 seconds in the wall time to 16.3 seconds.

In [54]:

```
scale = lambda val: (val-val.mean())/val.std()
```

In [55]:

```
scaledAllData = dc.allData.apply(scale)
pca1 = KernelPCA(n_components=1).fit(scaledAllData)
pca5 = KernelPCA(n_components=5).fit(scaledAllData)
```

In [56]:

```
nasdaq100indices = pd.DataFrame(pca1.transform(-dc.allData),columns=['PCA1'])
pcaComponents = pca5.transform(-dc.allData)
scaleProp = lambda vec: vec/vec.sum()
weights = scaleProp(pca5.lambdas_)
nasdaq100indices['PCA5'] = np.dot(pcaComponents,weights)
```

In [57]:

```
nasdaq100stocks = web.DataReader('IXF',data_source='yahoo',start='1/1/2020',end='1/1/2021')
```

In [58]:

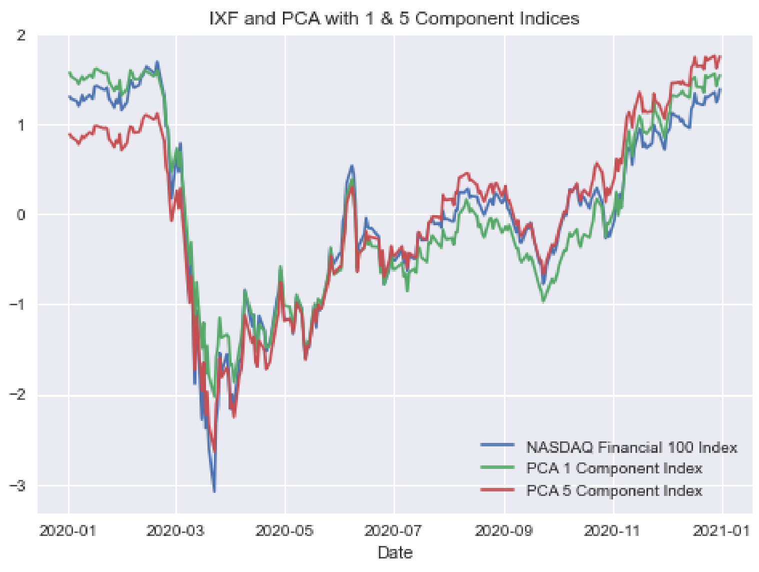
```
nasdaq100indices['IXF'] = nasdaq100stocks['Adj Close'].values
```

In [59]:

```
scaledData = nasdaq100indices.apply(scale)
```

In [60]:

```
plt.plot(nasdaq100stocks.index,scaledData['IXF'],label='NASDAQ Financial 100 Index')
plt.plot(nasdaq100stocks.index,scaledData['PCA1'],label='PCA 1 Component Index')
plt.plot(nasdaq100stocks.index,scaledData['PCA5'],label='PCA 5 Component Index')
plt.title("IXF and PCA with 1 & 5 Component Indices")
plt.xlabel("Date")
plt.legend()
plt.show()
```



Analytics Summary: Using data from Lazard and other companies that are within the NASDAQ

Analytics Summary: Using data from Lazard and other companies that are within the NASDAQ Financial 100, *exploratory data analytics* of the Lazard time series was done. Lazard's stock price trends were *estimated* using k-NN non-parametric regression and moving averages. Then triangle continuation patterns in price were investigated numerically using hypothesis testing and min-max retrieval using rolling mean. The candlestick plot revealed turning points in price when there were associated COVID-19 events. Volatility analytics was done on 2020 price data from Lazard. Simulations were done on possible paths 61 days after January 1, 2021. ARIMA(2,1,1) model performance was evaluated on the 15 days leading up to January 1, 2021, where it was able to capture the decline in price. Predictive analytics were investigated further with LSTM neural networks. Key out-of-sample performance indicators were calculated and LSTM did well. Then PCA indices were constructed from the NASDAQ Financial 100, where costly data acquisition of the companies was sped up with joblib parallelism.

Future Work: We can look into integration with the Hadoop architecture when aggregating stock data. Bayesian regression can be investigated as a predictive analytics method on the NASDAQ Financial 100 companies and Lazard. Beyond stocks, ETFs - collections of individual stocks - can be analyzed from the aforementioned companies and others.