Reference

Machine learning for finance: DataCamp course

Preparing data and a linear model

Explore the data with some EDA

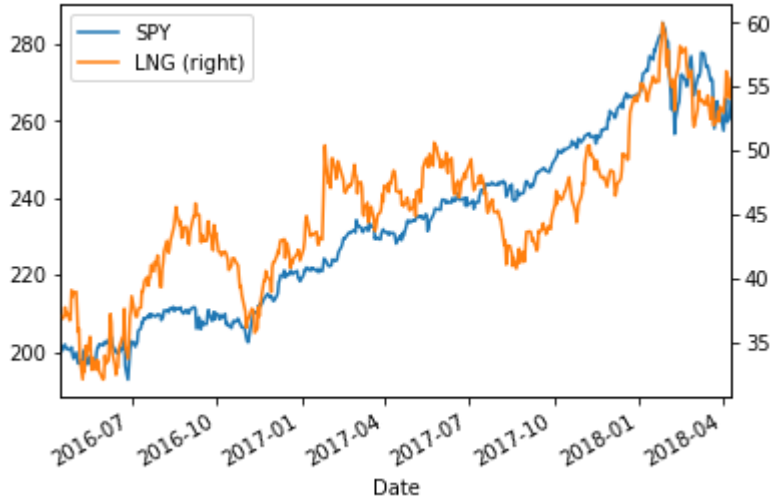
Beginning with raw data plots and histograms allows us to understand our data's distributions. **If it's a normal distribution, we can use things like parametric statistics.**

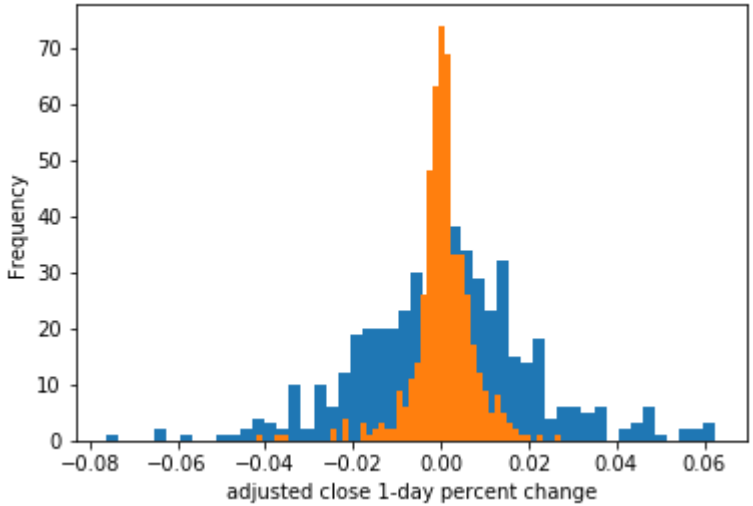
Comments:

- Parametric machine learning algorithms make population distribution assumption.
- For linear regression, although a normal distribution can give a linear regression formula, it is not a necessary condition. See cs229 notes.

In [3]:
```
import pandas as pd
import matplotlib.pyplot as plt

lng_df = pd.read_csv("LNG.csv", index_col = 0, parse_dates = True)
spy_df = pd.read_csv("spy.csv", index_col = 0, parse_dates = True)

lng_df = lng_df.loc['2016-Apr-15':'2018-Apr-10',:]
spy_df = spy_df.loc['2016-Apr-15':'2018-Apr-10',:]

print(lng_df.head())
print(spy_df.head())

spy_df['Adj_Close'].plot(label='SPY', legend=True)
lng_df['Adj_Close'].plot(label='LNG', legend=True, secondary_y=True)
plt.show()
plt.clf()

# Histogram of the daily price change percent of Adj_Close for LNG
lng_df['Adj_Close'].pct_change().plot.hist(bins=50)
spy_df['Adj_Close'].pct_change().plot.hist(bins=50)
plt.xlabel('adjusted close 1-day percent change')
plt.show()

print(lng_df.tail())
print(spy_df.tail())
```

```
            Adj_Close  Adj_Volume
Date
2016-04-15      37.13   4293775.0
2016-04-18      36.90   3445852.0
2016-04-19      37.12   3748050.0
2016-04-20      37.77   2470384.0
2016-04-21      37.21   2043988.0
             Adj_Close  Adj_Volume
Date
2016-04-15  199.760673  75761600.0
2016-04-18  201.164330  75277700.0
2016-04-19  201.798846  88316100.0
2016-04-20  201.991139  81100300.0
2016-04-21  200.904747  85695000.0
```





	Adj_Close	Adj_Volume
Date		
2018-04-04	54.30	1816771.0
2018-04-05	56.21	2533635.0
2018-04-06	54.36	1754446.0
2018-04-09	54.10	1203998.0
2018-04-10	55.63	3430268.0

	Adj_Close	Adj_Volume
Date		
2018-04-04	263.56	123574054.0

```
2018-04-05     265.64   80993290.0
2018-04-06     259.72  179483634.0
2018-04-09     261.00  105442932.0
2018-04-10     265.15  105383213.0
```

The above figures are different from those of DataCamp as the data used are different.

Correlations

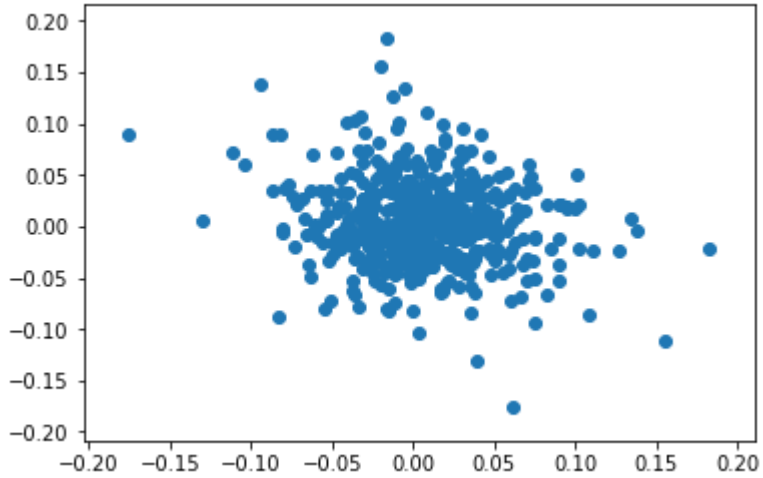
Correlations are nice to check out before building machine learning models, because we can see which features correlate to the target most strongly. Pearson's correlation coefficient is often used, which only **detects linear relationships**.

If we use the same time periods for previous price changes and future price changes, we can see if the stock price is mean-reverting (bounces around) or trend-following (goes up if it has been going up recently).

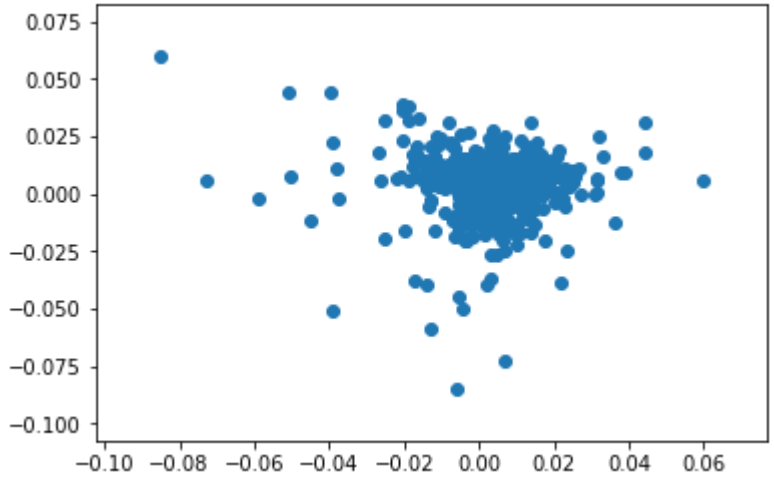
This is similar to the analysis for ACF in other course, except here we only calculate ACF for one lag. The negative value for one lag in ACF in other course here corresponds to a 2D scattered plot, which shows negative correlation, a north-west to south-east bound cloud.

In [8]:
```
# Create 5-day % changes of Adj_Close for the current day, and 5 days in the future
lng_df['5d_future_close'] = lng_df['Adj_Close'].shift(-5)
#shift to the left, and thus the final five days has no value.
#understand why this is called future close. See the print code later.
#It would be clear if we draw a number line to show how the pct_change(5) is calculated for the two cases.
lng_df['5d_close_future_pct'] = lng_df['5d_future_close'].pct_change(5)
lng_df['5d_close_pct'] = lng_df['Adj_Close'].pct_change(5)

# print(lng_df['Adj_Close'])
# print("--------------------")
# print(lng_df['5d_future_close'])
# print(lng_df['5d_close_pct'])
# print('---------------------------')
# print(lng_df['5d_close_future_pct'])


# Calculate the correlation matrix between the 5d close pecentage changes (current and future)
corr = lng_df[['5d_close_pct', '5d_close_future_pct']].corr()
print(corr)

# Scatter the current 5-day percent change vs the future 5-day percent change
plt.scatter(lng_df['5d_close_pct'], lng_df['5d_close_future_pct'])
plt.show()

# Create 5-day % changes of Adj_Close for the current day, and 5 days in the future
spy_df['5d_future_close'] = spy_df['Adj_Close'].shift(-5)
spy_df['5d_close_future_pct'] = spy_df['5d_future_close'].pct_change(5)
spy_df['5d_close_pct'] = spy_df['Adj_Close'].pct_change(5)

# Calculate the correlation matrix between the 5d close pecentage changes (current and future)
corr = spy_df[['5d_close_pct', '5d_close_future_pct']].corr()
print(corr)

# Scatter the current 5-day percent change vs the future 5-day percent change
plt.scatter(spy_df['5d_close_pct'], spy_df['5d_close_future_pct'])
plt.show()
```

```
                   5d_close_pct  5d_close_future_pct
5d_close_pct           1.000000            -0.164861
5d_close_future_pct   -0.164861             1.000000
```



	5d_close_pct	5d_close_future_pct
5d_close_pct	1.000000	-0.081765
5d_close_future_pct	-0.081765	1.000000



We can see the 5-day change is slightly negatively correlated to the change in the last 5 days -- an example of overall mean reversion!

Create moving average and RSI features

We want to add historical data to our machine learning models to make better predictions, but adding lots of historical time steps is tricky. Instead, we can condense information from previous points into a single timestep with indicators.

A moving average is one of the simplest indicators - it's the average of previous data points. This is the function talib.SMA() from the TAlib library.

Another common technical indicator is the relative strength index (RSI). This is defined by:

RSI=100−100/(1+RS) RS=average gain over n periods / (average loss over n periods) The n periods is set in talib.RSI() as the timeperiod argument.

A common period for RSI is 14, so we'll use that as one setting in our calculations.

Comments

- For very high dimensional data such as time-series data or images, normally we need ways to reduce the dimension.
- For images, convolutional neural network provides a nice way to reduce dimension while still keep the underlying features.
- For time series, one way is to use time window to reduce the dimensions.

There is problem installing ta-lib Using: conda install -c Quantopian ta-lib will have a problem of conlict.
Using: conda install -c masdeseiscaracteres ta-lib works.

In [10]:
```
import talib
feature_names = ['5d_close_pct']  # a list of the feature names for later

# Create moving averages and rsi for timeperiods of 14, 30, 50, and 200
for n in [14, 30, 50, 200]:

    # Create the moving average indicator and divide by Adj_Close
    lng_df['ma' + str(n)] = talib.SMA(lng_df['Adj_Close'].values,
                                      timeperiod=n) / lng_df['Adj_Close']
    # Create the RSI indicator
    lng_df['rsi' + str(n)] = talib.RSI(lng_df['Adj_Close'].values, timeperiod=n)

    # Add rsi and moving average to the feature name list
    feature_names = feature_names + ['ma' + str(n), 'rsi' + str(n)]

print(feature_names)
# print(lng_df.head(50))
```

```
['5d_close_pct', 'ma14', 'rsi14', 'ma30', 'rsi30', 'ma50', 'rsi50', 'ma200', 'rsi200']
```

Create features and targets

We have **features from current price changes (5d_close_pct) and indicators (moving averages and RSI), and we created targets** of future price changes (5d_close_future_pct). Now we need to break these up into separate numpy arrays so we can feed them into machine learning algorithms. Note the following print out the correlation among different features.

In [11]:
```
# Drop all na values from calculations.
lng_df = lng_df.dropna() #From here, we do not need to worry about the NAN generated before

# Create features and targets
# use feature_names for features; 5d_close_future_pct for targets
features = lng_df[feature_names]
targets = lng_df['5d_close_future_pct']

# Create DataFrame from target column and feature columns
# The following is only used to calucate correlation. For machine learning regression models, we don't need put f
# and target together. However, calculate correlation below might help select features by eliminating strongly co
feat_targ_df = lng_df[['5d_close_future_pct'] + feature_names]
# A nice way to create a DataFrame. Summarize the other two ways of adding new columns to Data Frame.

corr = feat_targ_df.corr()
print(corr)
```

```
                    5d_close_future_pct  5d_close_pct      ma14     rsi14  \
5d_close_future_pct             1.000000     -0.047183  0.096373 -0.068888
5d_close_pct                   -0.047183      1.000000 -0.827699  0.683973
ma14                            0.096373     -0.827699  1.000000 -0.877566
rsi14                          -0.068888      0.683973 -0.877566  1.000000
ma30                            0.102744     -0.609573  0.848778 -0.964795
rsi30                          -0.106279      0.518748 -0.713427  0.935711
ma50                            0.113444     -0.475081  0.692689 -0.916540
rsi50                          -0.138946      0.426045 -0.601849  0.845788
ma200                           0.230860     -0.220690  0.346457 -0.551087
rsi200                         -0.221029      0.284021 -0.416221  0.639057

                        ma30     rsi30      ma50     rsi50     ma200  \
5d_close_future_pct  0.102744 -0.106279  0.113444 -0.138946  0.230860
5d_close_pct        -0.609573  0.518748 -0.475081  0.426045 -0.220690
ma14                 0.848778 -0.713427  0.692689 -0.601849  0.346457
rsi14               -0.964795  0.935711 -0.916540  0.845788 -0.551087
ma30                 1.000000 -0.900934  0.925715 -0.805506  0.527767
rsi30               -0.900934  1.000000 -0.962825  0.975608 -0.761846
ma50                 0.925715 -0.962825  1.000000 -0.915729  0.693863
rsi50               -0.805506  0.975608 -0.915729  1.000000 -0.871883
ma200                0.527767 -0.761846  0.693863 -0.871883  1.000000
rsi200              -0.600068  0.834532 -0.750857  0.930507 -0.976110
```

```
                       rsi200
5d_close_future_pct -0.221029
5d_close_pct         0.284021
ma14                -0.416221
rsi14                0.639057
ma30                -0.600068
rsi30                0.834532
ma50                -0.750857
rsi50                0.930507
ma200               -0.976110
rsi200               1.000000
```

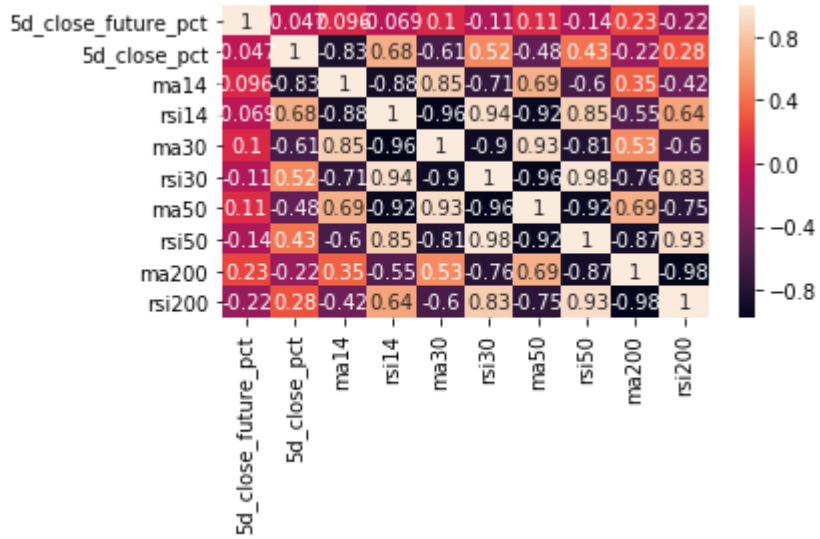
now we've got features and targets ready for machine learning!

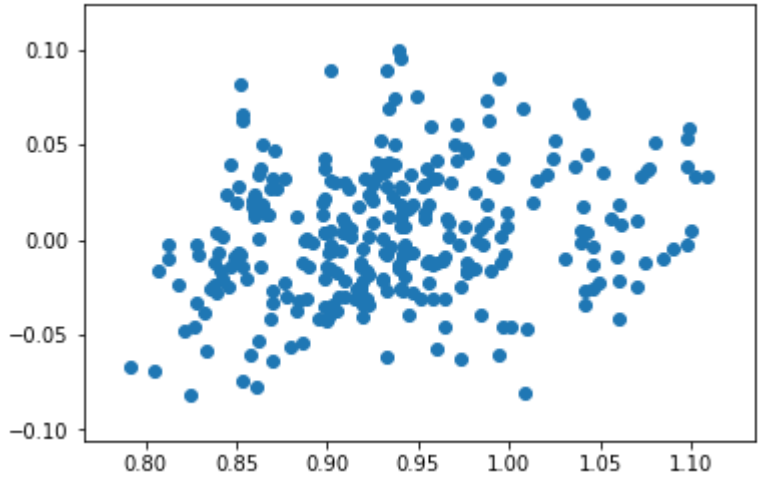
Check the correlations

Before we fit our first machine learning model, let's look at the correlations between features and targets. Ideally we want large (near 1 or -1) correlations between features and targets. Examining correlations can help us tweak features to maximize correlation (for example, altering the timeperiod argument in the talib functions). It can also help us remove features that aren't correlated to the target. **Comments**. It can also help us to eliminate some of those strongly correlated features.

To easily plot a correlation matrix, we can use seaborn's heatmap() function. This takes a correlation matrix as the first argument, and has many other options. Check out the annot option -- this will help us turn on annotations.

In [12]:
```
import seaborn as sns
# Plot heatmap of correlation matrix
sns.heatmap(corr, annot=True)
plt.yticks(rotation=0); plt.xticks(rotation=90)  # fix ticklabel directions
plt.tight_layout()  # fits plot area to the plot, "tightly"
plt.show()  # show the plot
plt.clf()  # clear the plot area

# Create a scatter plot of the most highly correlated variable with the target
plt.scatter(lng_df['ma200'], lng_df['5d_close_future_pct'])
plt.show()
```





We can see a few features have some correlation to the target! **Comments. Here we should plot multiple data as in the visualization course**.

Create train and test features

Before we fit our linear model, we want to add a constant to our features, so we have an intercept for our linear model.

We also want to create train and test features. This is so we can fit our model to the train dataset, and evaluate performance on the test dataset. We always want to check performance on data the model has not seen to make sure we're not overfitting, which is memorizing patterns in the training data too exactly.

With a time series like this, we typically want to use the oldest data as our training set, and the newest data as our test set. This is so we can evaluate the performance of the model on the most recent data, which will more realistically simulate predictions on data we haven't seen yet. **There is a simple way to do this in scikit-learn**.

In [24]:
```
# Use other way is simpler. The following is an old way.
# Import the statsmodels library with the alias sm
import statsmodels.api as sm

# Add a constant to the features
linear_features = sm.add_constant(features) #This is not necessary with other packages.

# Create a size for the training set that is 85% of the total number of samples
train_size = int(0.85 * features.shape[0])
train_features = linear_features[:train_size]
train_targets = targets[:train_size]
test_features = linear_features[train_size:]
test_targets = targets[train_size:]
print(linear_features.shape, train_features.shape, test_features.shape)
print(linear_features.head())
```

```
(295, 10) (250, 10) (45, 10)
            const  5d_close_pct      ma14      rsi14      ma30      rsi30  \
Date
2017-01-31    1.0      0.043812  0.950697  62.968946  0.918657  60.655749
2017-02-01    1.0     -0.023429  0.967486  60.156019  0.932427  59.159791
2017-02-02    1.0     -0.021618  0.932220  66.659314  0.895486  62.988478
2017-02-03    1.0      0.007336  0.938976  66.920059  0.899461  63.148741
2017-02-06    1.0      0.022129  0.964080  62.059184  0.921608  60.623276

                ma50      rsi50     ma200     rsi200
Date
2017-01-31  0.895731  57.951677  0.839267  53.695377
2017-02-01  0.910110  57.041713  0.850150  53.487663
2017-02-02  0.873379  59.662227  0.812956  54.209475
2017-02-03  0.875497  59.774509  0.812490  54.241479
2017-02-06  0.895860  58.236185  0.829234  53.889442

C:\Users\ljyan\Anaconda3\lib\site-packages\numpy\core\fromnumeric.py:52: FutureWarning: Method .ptp is deprecat
ed and will be removed in a future version. Use numpy.ptp instead.
  return getattr(obj, method)(*args, **kwds)
```

Fit a linear model

We'll now fit a linear model, because they are simple and easy to understand. Once we've fit our model, we can see which predictor variables appear to be meaningfully linearly correlated with the target, as well as their magnitude of effect on the target. Our judgment of whether or not predictors are significant is based on the p-values of coefficients. This is using a t-test to statistically test if the coefficient significantly differs from 0. The p-value is the percent chance that the coefficient for a feature does not differ from zero. Typically, we take a p-value of less than 0.05 to mean the coefficient is significantly different from 0.

In [25]:
```
# Create the linear model and complete the least squares fit
model = sm.OLS(train_targets, train_features)
results = model.fit()  # fit the model
print(results.summary())

# examine pvalues
# Features with p <= 0.05 are typically considered significantly different from 0
print(results.pvalues)

# Make predictions from our model for train and test sets
train_predictions = results.predict(train_features)
test_predictions = results.predict(test_features)
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:     5d_close_future_pct   R-squared:                       0.273
Model:                             OLS   Adj. R-squared:                  0.246
Method:                  Least Squares   F-statistic:                     10.01
Date:                 Wed, 29 May 2019   Prob (F-statistic):           4.92e-13
Time:                         11:39:58   Log-Likelihood:                536.49
No. Observations:                  250   AIC:                            -1053.
Df Residuals:                      240   BIC:                            -1018.
Df Model:                            9
Covariance Type:             nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          6.8197      1.169      5.832      0.000       4.516       9.123
5d_close_pct  -0.0944      0.114     -0.830      0.408      -0.319       0.130
ma14           0.3473      0.230      1.512      0.132      -0.105       0.800
rsi14          0.0261      0.004      6.520      0.000       0.018       0.034
ma30           0.2200      0.206      1.067      0.287      -0.186       0.626
rsi30         -0.1789      0.025     -7.111      0.000      -0.228      -0.129
ma50          -2.0856      0.374     -5.578      0.000      -2.822      -1.349
rsi50          0.2410      0.032      7.458      0.000       0.177       0.305
ma200          0.5639      0.220      2.567      0.011       0.131       0.997
rsi200        -0.1999      0.029     -6.999      0.000      -0.256      -0.144
==============================================================================
Omnibus:                        3.594   Durbin-Watson:                   0.560
Prob(Omnibus):                  0.166   Jarque-Bera (JB):                2.482
Skew:                          -0.038   Prob(JB):                        0.289
Kurtosis:                       2.518   Cond. No.                     6.92e+04
```

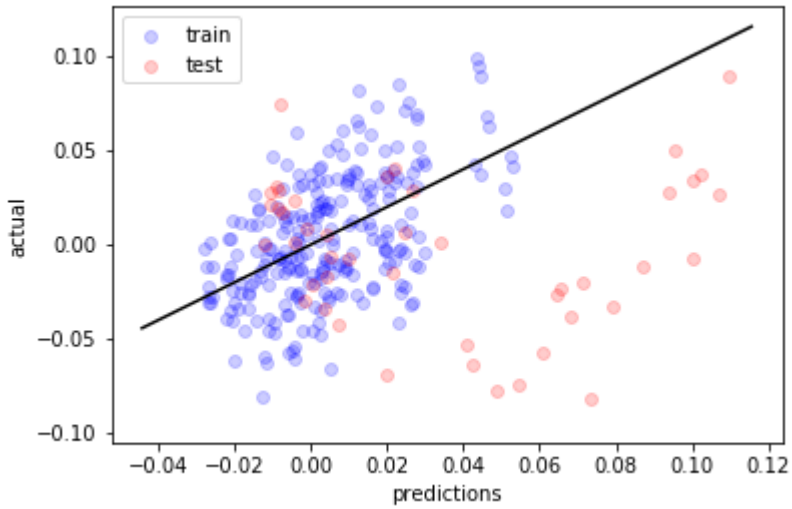
```
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 6.92e+04. This might indicate that there are
strong multicollinearity or other numerical problems.
const           1.764767e-08
5d_close_pct    4.075985e-01
ma14            1.317652e-01
rsi14           4.119023e-10
ma30            2.870964e-01
rsi30           1.315491e-11
ma50            6.542888e-08
rsi50           1.598367e-12
ma200           1.087610e-02
rsi200          2.559536e-11
dtype: float64
```

Evaluate our results

Once we have our linear fit and predictions, we want to see how good the predictions are so we can decide if our model is any good or not. Ideally, we want to back-test any type of trading strategy. However, this is a complex and typically time-consuming experience.

A quicker way to understand the performance of our model is looking at regression evaluation metrics like R2, and plotting the predictions versus the actual values of the targets. Perfect predictions would form a straight, diagonal line in such a plot, making it easy for us to eyeball how our predictions are doing in different regions of price changes. We can use matplotlib's .scatter() function to create scatter plots of the predictions and actual values.

In [26]:
```
import numpy as np
# Scatter the predictions vs the targets with 80% transparency
plt.scatter(train_predictions, train_targets, alpha=0.2, color='b', label='train')
plt.scatter(test_predictions, test_targets, alpha=0.2, color='r', label='test')

# Plot the perfect prediction line
xmin, xmax = plt.xlim()
plt.plot(np.arange(xmin, xmax, 0.01), np.arange(xmin, xmax, 0.01), c='k')

# Set the axis labels and show the plot
plt.xlabel('predictions')
plt.ylabel('actual')
plt.legend()  # show the legend
plt.show()
```



The following is from DataCamp. Note the data used here is very different from that of Datacamp and thus the results are very different. In the DataCamp results, we see the predictions are ok, but not very good yet. We need non-linearity!

Machine learning tree methods

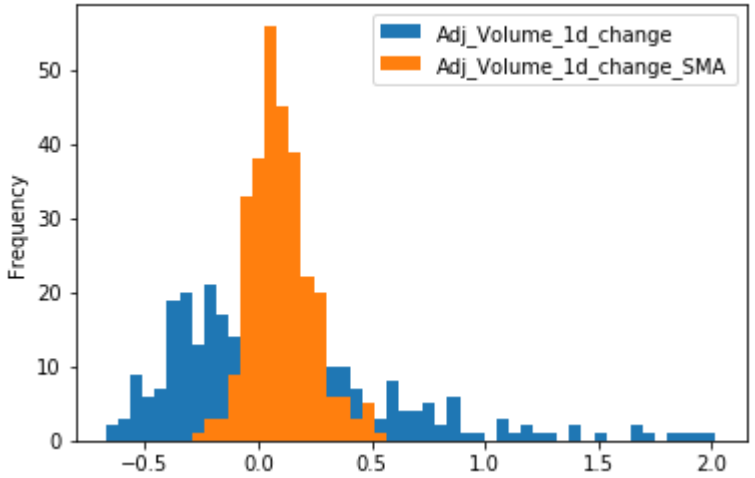
Learn how to use tree-based machine learning models to predict future values of a stock's price, as well as how to use forest-based machine learning methods for regression and feature selection.

Feature engineering from volume

We're going to use non-linear models to make more accurate predictions. With linear models, **features must be linearly correlated to the target**. Other machine learning models can combine features in non-linear ways. For example, what if the price goes up when the moving average of price is going up, and the moving average of volume is going down? The only way to capture those interactions is to either multiply the features, or to use a machine learning algorithm that can handle non-linearity (e.g. random forests).

To incorporate more information that may interact with other features, we can add in weakly-correlated features. First we will add volume data, which we have in the lng_df as the Adj_Volume column.

In [27]:
```
# Create 2 new volume features, 1-day % change and 5-day SMA of the % change
new_features = ['Adj_Volume_1d_change', 'Adj_Volume_1d_change_SMA']
feature_names.extend(new_features)
lng_df['Adj_Volume_1d_change'] = lng_df['Adj_Volume'].pct_change()
lng_df['Adj_Volume_1d_change_SMA'] = talib.SMA(lng_df['Adj_Volume_1d_change'].values,
                                               timeperiod=5)

# Plot histogram of volume % change data
lng_df[new_features].plot(kind='hist', sharex=False, bins=50)
plt.show()
```



We can see the moving average of volume changes has a much smaller range than the raw data.

Create day-of-week features

We can engineer datetime features to add even more information for our non-linear models. Most financial data has datetimes, which have lots of information in them -- year, month, day, and sometimes hour, minute, and second. But we can also get the day of the week, and things like the quarter of the year, or the elapsed time since some event (e.g. earnings reports).

We are only going to get the day of the week here, since our dataset doesn't go back very far in time. The dayofweek property from the pandas datetime index will help us get the day of the week. Then we will dummy dayofweek with pandas' get_dummies(). This creates columns for each day of the week with binary values (0 or 1). We drop the first column because it can be inferred from the others.

In [28]:
```
# Use pandas' get_dummies function to get dummies for day of the week
days_of_week = pd.get_dummies(lng_df.index.dayofweek,
                              prefix='weekday',
                              drop_first=True)

# Set the index as the original DataFrame index for merging
days_of_week.index = lng_df.index

# Join the dataframe with the days of week DataFrame
lng_df = pd.concat([lng_df, days_of_week], axis=1)

# Add days of week to feature names
feature_names.extend(['weekday_' + str(i) for i in range(1, 5)])
lng_df.dropna(inplace=True)  # drop missing values in-place
print(lng_df.head())
```

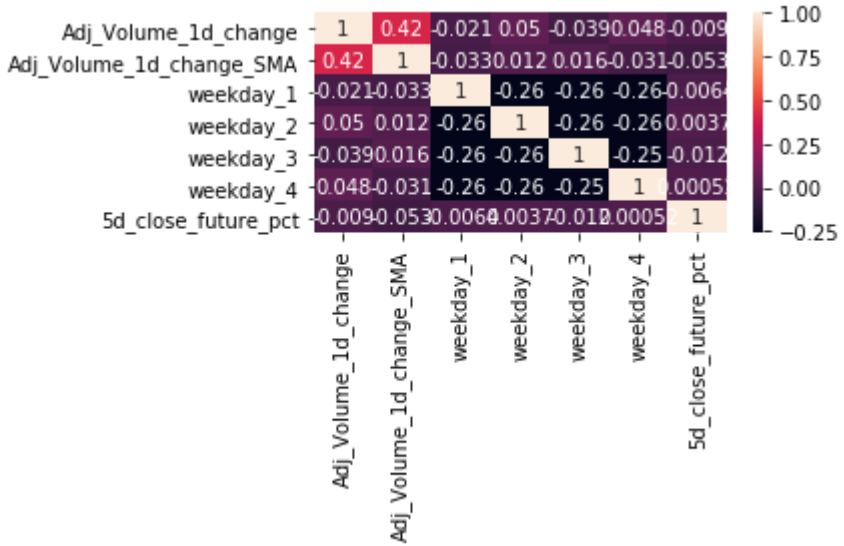
```
            Adj_Close  Adj_Volume  5d_future_close  5d_close_future_pct  \
Date
2017-02-07      47.81   2522644.0            47.88             0.001464
2017-02-08      48.01   1521339.0            47.69            -0.006665
2017-02-09      49.19   1717838.0            46.84            -0.047774
2017-02-10      48.93   1747989.0            46.71            -0.045371
2017-02-13      48.14   2321977.0            47.36            -0.016203

            5d_close_pct      ma14      rsi14      ma30      rsi30      ma50  \
Date
2017-02-07      0.003358  0.985448  58.654992  0.939476  58.817869  0.911801
2017-02-08      0.019321  0.989333  59.351015  0.940047  59.182360  0.911289
2017-02-09     -0.002838  0.974211  63.278848  0.922383  61.274321  0.893214
2017-02-10     -0.010115  0.987957  61.860468  0.931821  60.566775  0.902031
2017-02-13     -0.007423  1.007864  57.633333  0.951371  58.445466  0.919859

                rsi50     ma200     rsi200  Adj_Volume_1d_change  \
Date
2017-02-07  57.123212  0.842290  53.629901              0.206482
2017-02-08  57.364214  0.839954  53.694867             -0.396927
2017-02-09  58.759765  0.821018  54.076387              0.129162
2017-02-10  58.330493  0.826385  53.977901              0.017552
2017-02-13  57.038536  0.840935  53.679362              0.328370

            Adj_Volume_1d_change_SMA  weekday_1  weekday_2  weekday_3  \
Date
```

```
2017-02-07                 0.104112          1          0          0
2017-02-08                 0.032752          0          1          0
2017-02-09                -0.041798          0          0          1
2017-02-10                -0.108177          0          0          0
2017-02-13                 0.056928          0          0          0

            weekday_4
Date
2017-02-07          0
2017-02-08          0
2017-02-09          0
2017-02-10          1
2017-02-13          0
```

Examine correlations of the new features

Now that we have our volume and datetime features, we want to check the correlations between our new features (stored in the new_features list) and the target (5d_close_future_pct) to see how strongly they are related. Recall pandas has the built-in .corr() method for DataFrames, and seaborn has a nice heatmap() function to show the correlations.

In [29]:
```
import seaborn as sns
# Add the weekday labels to the new_features list
new_features.extend(['weekday_' + str(i) for i in range(1, 5)])

# Plot the correlations between the new features and the targets
sns.heatmap(lng_df[new_features + ['5d_close_future_pct']].corr(), annot=True)
plt.yticks(rotation=0)  # ensure y-axis ticklabels are horizontal
plt.xticks(rotation=90)  # ensure x-axis ticklabels are vertical
plt.tight_layout()
plt.show()
```



Even though the correlations are weak, they may improve our predictions via interactions with other features.

Fit a decision tree

We can use sklearn to fit a decision tree with DecisionTreeRegressor and .fit(features, targets).

Without limiting the tree's depth (or height), it will keep splitting the data until each leaf has 1 sample in it, which is the epitome of overfitting. We'll learn more about overfitting in the coming chapters.

In [30]:
```
from sklearn.tree import DecisionTreeRegressor

# Create a decision tree regression model with default arguments
decision_tree = DecisionTreeRegressor()

# Fit the model to the training features and targets
decision_tree.fit(train_features, train_targets)

# Check the score on train and test
print(decision_tree.score(train_features, train_targets))
print(decision_tree.score(test_features, test_targets))
```

```
0.9999950331816782
-1.5004856430060705
```

Try different max depths

For regular decision trees, probably the most important hyperparameter is max_depth. This limits the number of splits in a decision tree. Let's find the best value of max_depth based on the R2 score of our model on the test set, which we can obtain using the score() method of our decision tree models.

In [31]:
```
# Loop through a few different max depths and check the performance
for d in [3, 5, 10]:
    # Create the tree and fit it
    decision_tree = DecisionTreeRegressor(max_depth=d)
    decision_tree.fit(train_features, train_targets)

    # Print out the scores on train and test
    print('max_depth=', str(d))
    print(decision_tree.score(train_features, train_targets))
    print(decision_tree.score(test_features, test_targets), '\n')
```

```
max_depth= 3
0.31221074754988065
-0.34425820543875796

max_depth= 5
0.5300236519892909
-1.143575755704576

max_depth= 10
0.896998296863188
-1.4996391746861657
```

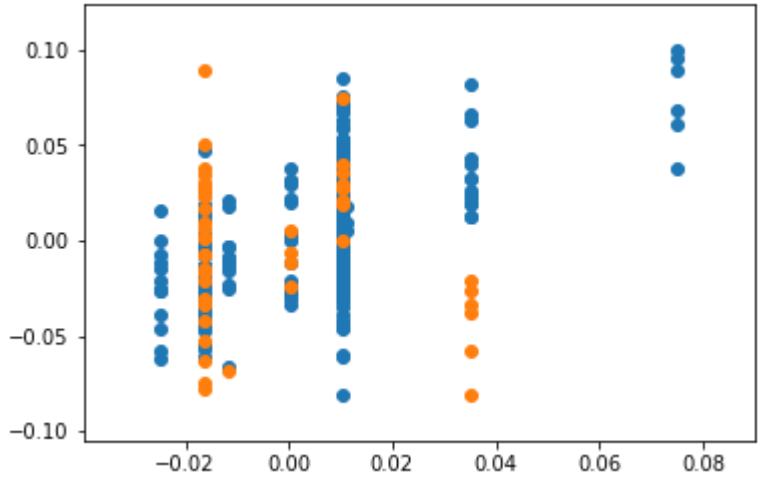
Remember what value of max_depth got the highest test score for the next exercise!

Check our results

Once we have an optimized model, we want to check how it is performing in more detail. We already saw the R2 score, but it can be helpful to see the predictions plotted vs actual values. We can use the .predict() method of our decision tree model to get predictions on the train and test sets.

Ideally, we want to see diagonal lines from the lower left to the upper right. However, due to the simplicity of decisions trees, our model is not going to do well on the test set. But it will do well on the train set.

In [36]:
```
# Use the best max_depth of 3 from last exercise to fit a decision tree
decision_tree = DecisionTreeRegressor(max_depth=3)
decision_tree.fit(train_features, train_targets)

# Predict values for train and test
train_predictions = decision_tree.predict(train_features)
test_predictions = decision_tree.predict(test_features)

# Scatter the predictions vs actual values
plt.scatter(train_predictions, train_targets, label='train')
plt.scatter(test_predictions, test_targets, label='test')
plt.show()
```



Fit a random forest

Data scientists often use random forest models. They perform well out of the box, and have lots of settings to optimize performance. Random forests can be used for **classification or regression**; we'll use it for regression to predict the future price change of LNG.

We'll create and fit the random forest model similarly to the decision trees using the .fit(features, targets) method. With sklearn's RandomForestRegressor, there's a built-in .score() method we can use to evaluate performance. This takes arguments (features, targets), and returns the R2 score (the coefficient of determination).

In [37]:
```
from sklearn.ensemble import RandomForestRegressor

# Create the random forest model and fit to the training data
rfr = RandomForestRegressor(n_estimators=200)
rfr.fit(train_features, train_targets)

# Look at the R^2 scores on train and test
print(rfr.score(train_features, train_targets))
print(rfr.score(test_features, test_targets))
```

```
0.9168878190141329
0.003090359848654911
```

Tune random forest hyperparameters

As with all models, we want to optimize performance by tuning hyperparameters. We have many hyperparameters for random forests, but the most important is often **the number of features we sample at each split**, or max_features in RandomForestRegressor from the sklearn library. For models like random forests that have randomness built-in, we also want to set the **random_state**. This is set for our results to be reproducible.

Usually, we can use **sklearn's GridSearchCV()** method to search hyperparameters, but **with a financial time series, we don't want to do cross-validation due to data mixing**. We want to fit our models on the oldest data and evaluate on the newest data. So we'll use **sklearn's ParameterGrid to create combinations of hyperparameters to search. Figure out the details in the future.**

In [41]:
```
from sklearn.model_selection import ParameterGrid

# Create a dictionary of hyperparameters to search
grid = {'n_estimators': [200], 'max_depth': [3], 'max_features': [4, 8], 'random_state': [42]}
test_scores = []

# Loop through the parameter grid, set the hyperparameters, and save the scores
for g in ParameterGrid(grid):
    rfr.set_params(**g)  # ** is "unpacking" the dictionary
    rfr.fit(train_features, train_targets)
    test_scores.append(rfr.score(test_features, test_targets))

# Find best hyperparameters from the test score and print
best_idx = np.argmax(test_scores)
print(test_scores[best_idx], ParameterGrid(grid)[best_idx])
```

```
0.03463037963408222 {'random_state': 42, 'n_estimators': 200, 'max_features': 4, 'max_depth': 3}
```

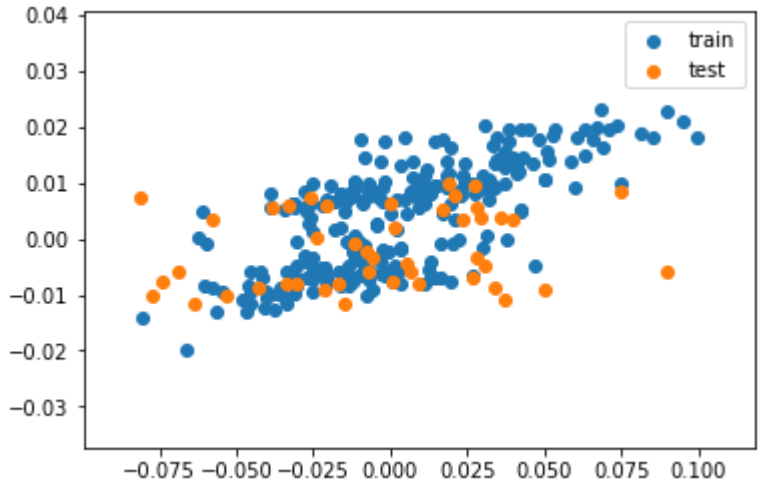
Our test score (R^2) isn't great, but it's greater than 0!

Evaluate performance

Lastly, and as always, we want to evaluate performance of our best model to check how well or poorly we are doing. Ideally **it's best to do back-testing**, but that's an involved process we don't have room to cover in this course.

We've already seen the R2 scores, but let's take a look at the scatter plot of predictions vs actual results using matplotlib. Perfect predictions would be a diagonal line from the lower left to the upper right.

In [17]:
```
# Use the best hyperparameters from before to fit a random forest model
rfr = RandomForestRegressor(n_estimators=200, max_depth=3, max_features=4, random_state=42)
rfr.fit(train_features, train_targets)

# Make predictions with our model
train_predictions = rfr.predict(train_features)
test_predictions = rfr.predict(test_features)

# Create a scatter plot with train and test actual vs predictions
plt.scatter(train_targets, train_predictions, label='train')
plt.scatter(test_targets, test_predictions, label='test')
plt.legend()
plt.show()
```



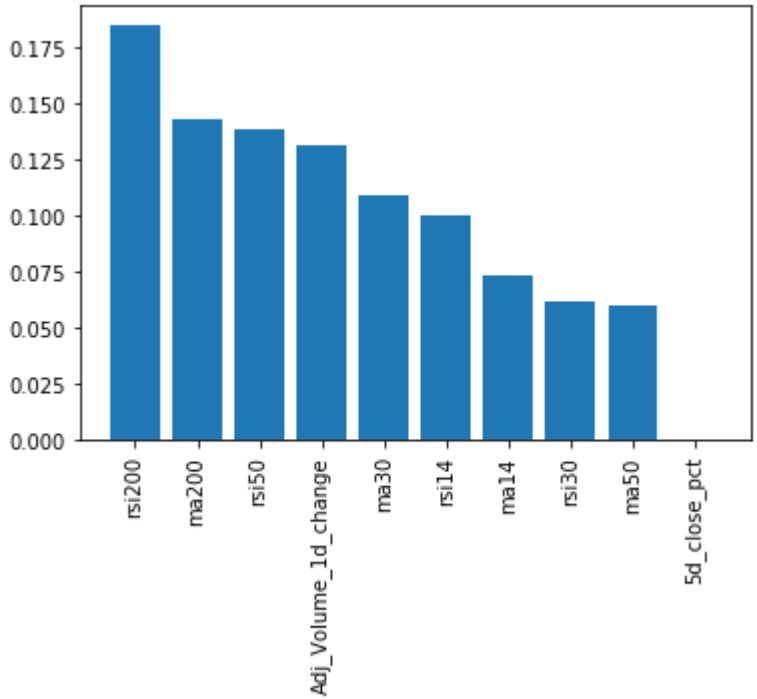
We can see our train predictions are good, but test predictions (generalization) are not great.

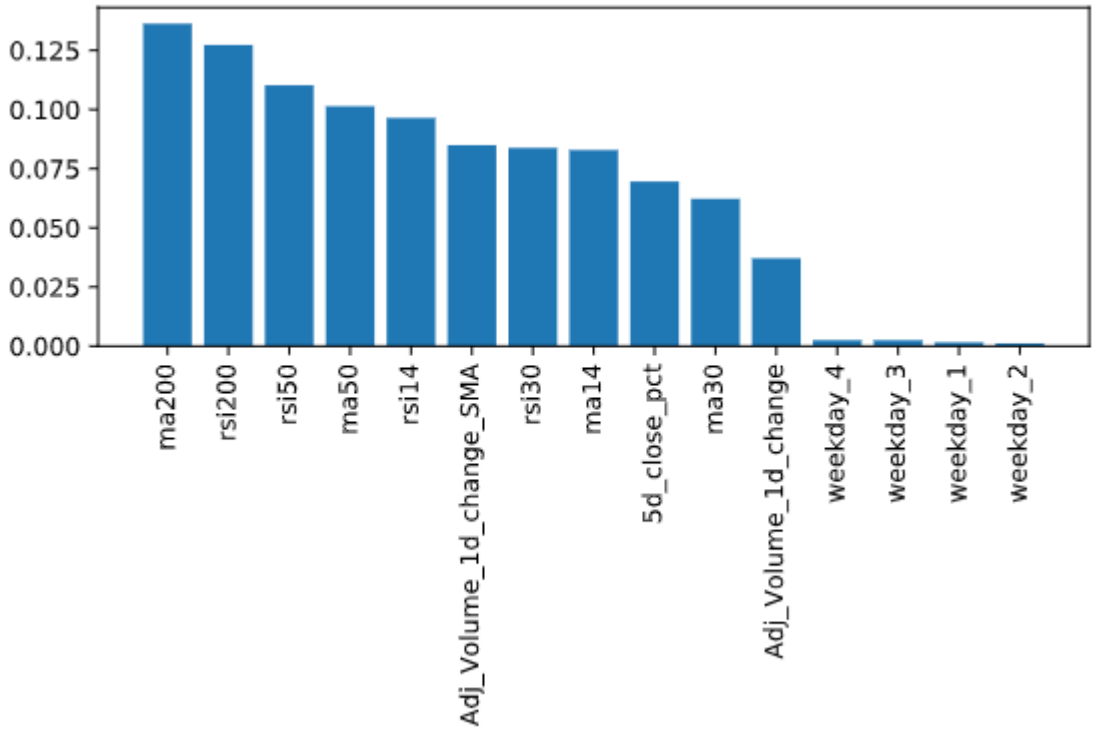
Random forest feature importances

One useful aspect of tree-based methods is the ability to extract feature importances. This is a quantitative way to measure how much each feature contributes to our predictions. It can help us focus on our best features, possibly enhancing or tuning them, and can also help us get rid of useless features that may be cluttering up our model.

Tree models in sklearn have a .feature*importances* property that's accessible after fitting the model. This stores the feature importance scores. We need to get the indices of the sorted feature importances using np.argsort() in order to make a nice-looking bar plot of feature importances (sorted from greatest to least importance).

In [44]:
```
# Get feature importances from our random forest model
importances = rfr.feature_importances_

# Get the index of importances from greatest importance to least
sorted_index = np.argsort(importances)[::-1]
x = range(len(importances))

# Create tick labels
labels = np.array(feature_names)[sorted_index]
plt.bar(x, importances[sorted_index], tick_label=labels)

# Rotate tick labels to vertical
plt.xticks(rotation=90)
plt.show()
```





Unsurprisingly, it looks like the days of the week should be thrown out. In the figure weekday_1 does not exist due to the drop column action from independence.

A gradient boosting model

Now we'll fit a gradient boosting (GB) model. It's been said **a linear model is like a Toyota Camry, and GB is like a Black Hawk helicopter.** GB has potential to outperform random forests, but doesn't always do so. This is called the no free lunch theorem, meaning we should always try lots of different models for each problem.

GB is similar to random forest models, but the difference is that trees are built successively. With each iteration, the next tree fits the residual errors from the previous tree in order to improve the fit.

For now we won't search our hyperparameters -- they've been searched for you.

In [45]:
```
from sklearn.ensemble import GradientBoostingRegressor

# Create GB model -- hyperparameters have already been searched for you
gbr = GradientBoostingRegressor(max_features=4,
                                learning_rate=0.01,
                                n_estimators=200,
                                subsample=0.6,
                                random_state=42)
gbr.fit(train_features, train_targets)

print(gbr.score(train_features, train_targets))
print(gbr.score(test_features, test_targets))
```

```
0.4412629798604665
0.009669282780346067
```

DataCamp resutls

```
0.4063115061547039
0.03992305163583343
```

In this case the gradient boosting model isn't that much better than a random forest, but you know what they say -- no free lunch!

Gradient boosting feature importances

As with random forests, we can extract feature importances from gradient boosting models to understand which features are the best predictors. **Sometimes it's nice to try different tree-based models and look at the feature importances from all of them.** This can help average out any peculiarities that may arise from one particular model.

The feature importances are stored as a numpy array in the .feature*importances* property of the gradient boosting model. We'll need to get the sorted indices of the feature importances, using np.argsort(), in order to make a nice plot. We want the features from largest to smallest, so we will use Python's indexing to reverse the sorted importances like feat_importances[::-1].

In [ ]:
```
# Extract feature importances from the fitted gradient boosting model
feature_importances = gbr.feature_importances_

# Get the indices of the largest to smallest feature importances
sorted_index = np.argsort(feature_importances)[::-1]
x = range(features.shape[1])

# Create tick labels
labels = np.array(feature_names)[sorted_index]

plt.bar(x, feature_importances[sorted_index], tick_label=labels)

# Set the tick lables to be the feature names, according to the sorted feature_idx
plt.xticks(rotation=90)
plt.show()
```

Notice the feature importances are not exactly the same as the random forest model's...but they're close.

Neural networks and KNN

We will learn how to normalize and scale data for use in KNN and neural network methods. Then we will learn how to use KNN and neural network regression to predict the future values of a stock's price (or any other regression problem).

Standardizing data

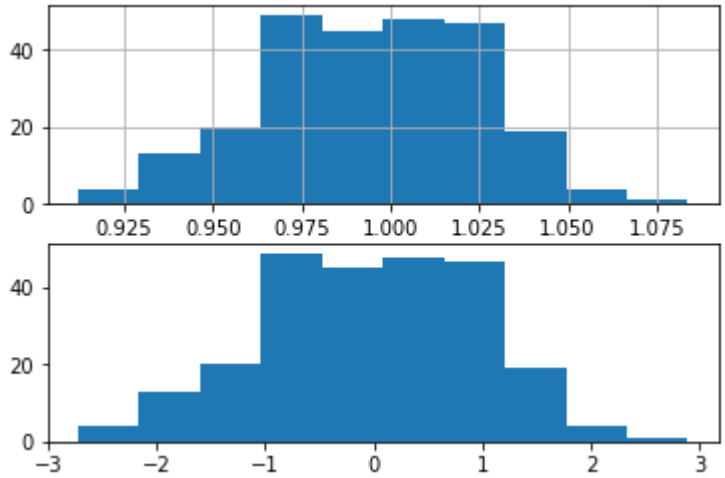
Some models, like K-nearest neighbors (KNN) & neural networks, work better with scaled data -- so we'll standardize our data.

We'll also remove unimportant variables (day of week), according to feature importances, by indexing the features DataFrames with .iloc[]. KNN uses distances to find similar points for predictions, so big features outweigh small ones. Scaling data fixes that.

sklearn's scale() will standardize data, which sets the mean to 0 and standard deviation to 1. Ideally we'd want to use StandardScaler with fit_transform() on the training data, and fit() on the test data, but we are limited to 15 lines of code here.

Once we've scaled the data, we'll check that it worked by plotting histograms of the data.

In [21]:
```
from sklearn.preprocessing import scale

# Remove unimportant features (weekdays)
train_features = train_features.iloc[:, :-4]
test_features = test_features.iloc[:, :-4]

# Standardize the train and test features
scaled_train_features = scale(train_features)
scaled_test_features = scale(test_features)

# Plot histograms of the 14-day SMA RSI before and after scaling
f, ax = plt.subplots(nrows=2, ncols=1)
train_features.iloc[:, 2].hist(ax=ax[0])
ax[1].hist(scaled_train_features[:, 2])
plt.show()
```



Optimize n_neighbors

Now that we have scaled data, we can try using a KNN model. To maximize performance, we should tune our model's hyperparameters. For the k-nearest neighbors algorithm, we only have one hyperparameter: n, the number of neighbors. We set this hyperparameter when we create the model with KNeighborsRegressor. The argument for the number of neighbors is n_neighbors.

We want to try a range of values that passes through the setting with the best performance. Usually we will start with 2 neighbors, and increase until our scoring metric starts to decrease. We'll use the R2 value from the .score() method on the test set (scaled_test_features and test_targets) to optimize n here. We'll use the test set scores to determine the best n.

In [22]:
```python
from sklearn.neighbors import KNeighborsRegressor

for n in range(2, 13):
    # Create and fit the KNN model
    knn = KNeighborsRegressor(n_neighbors=n)

    # Fit the model to the training data
    knn.fit(scaled_train_features, train_targets)

    # Print number of neighbors and the score to find the best value of n
    print("n_neighbors =", n)
    print('train, test scores')
    print(knn.score(scaled_train_features, train_targets))
    print(knn.score(scaled_test_features, test_targets))
    print()  # prints a blank line
```

```
n_neighbors = 2
train, test scores
0.7399807954377584
-0.2740672226942691

n_neighbors = 3
train, test scores
0.5824623875084005
-0.1181802758587529

n_neighbors = 4
train, test scores
0.46535493280420653
-0.15112267167378457

n_neighbors = 5
train, test scores
0.40037988429002047
-0.09383063949462533

n_neighbors = 6
train, test scores
0.3356478081872952
-0.034500892715922715

n_neighbors = 7
```

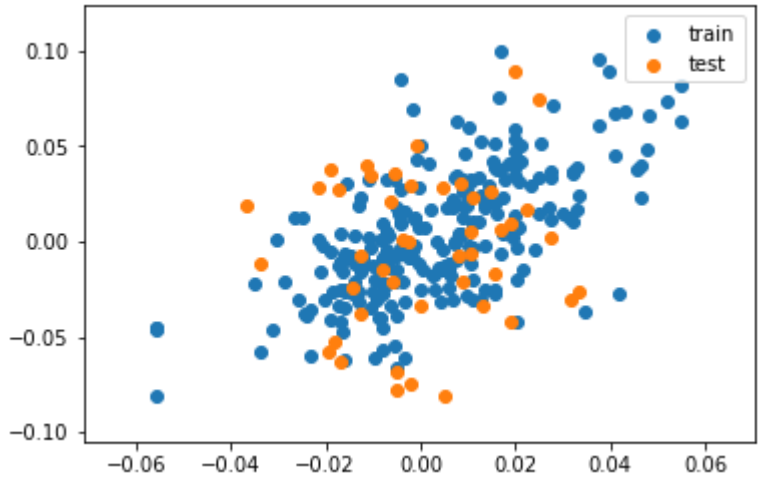
```
train, test scores
0.29425581102769294
0.005022709061472175

n_neighbors = 8
train, test scores
0.2847234128033601
3.995633668785192e-05

n_neighbors = 9
train, test scores
0.2690193196709867
0.012443040356902357

n_neighbors = 10
train, test scores
0.26768882984746856
0.0218643652727748

n_neighbors = 11
train, test scores
0.2420615649564759
0.005342723766639912

n_neighbors = 12
train, test scores
0.21366040665719777
-0.010402295557673469
```

Evaluate KNN performance

We just saw a few things with our KNN scores. For one, the training scores started high and decreased with increasing n, which is typical. The test set performance reached a peak at 5 though, and we will use that as our setting in the final KNN model.

As we have done a few times now, we will check our performance visually. This helps us see how well the model is predicting on different regions of actual values. We will get predictions from our knn model using the .predict() method on our scaled features. Then we'll use matplotlib's plt.scatter() to create a scatter plot of actual versus predicted values.

In [23]:
```
# Create the model with the best-performing n_neighbors of 5
knn = KNeighborsRegressor(n_neighbors=5)

# Fit the model
knn.fit(scaled_train_features, train_targets)

# Get predictions for train and test sets
train_predictions = knn.predict(scaled_train_features)
test_predictions = knn.predict(scaled_test_features)

# Plot the actual vs predicted values
plt.scatter(train_predictions, train_targets, label='train')
plt.scatter(test_predictions, test_targets, label='test')
plt.legend()
plt.show()
```



Build and fit a simple neural net

The next model we will learn how to use is a neural network. Neural nets can capture complex interactions between variables, but are difficult to set up and understand. Recently, they have been beating human experts in many fields, including image recognition and gaming (check out AlphaGo) -- so they have great potential to perform well.

To build our nets we'll use the keras library. This is a high-level API that allows us to quickly make neural nets, yet still exercise a lot of control over the design. The first thing we'll do is create almost the simplest net possible -- a 3-layer net that takes our inputs and predicts a single value. Much like the sklearn models, keras models have a .fit() method that takes arguments of (features, targets).

Instructions

In [24]:
```
from keras.models import Sequential
from keras.layers import Dense

# Create the model
model_1 = Sequential()
model_1.add(Dense(100, input_dim=scaled_train_features.shape[1], activation='relu'))
model_1.add(Dense(20, activation='relu'))
model_1.add(Dense(1, activation='linear'))

# Fit the model
model_1.compile(optimizer='adam', loss='mse')
history = model_1.fit(scaled_train_features, train_targets, epochs=25)
```

```
Using TensorFlow backend.

Epoch 1/25
250/250 [==============================] - 1s 5ms/step - loss: 0.0114
Epoch 2/25
250/250 [==============================] - 0s 48us/step - loss: 0.0048
Epoch 3/25
250/250 [==============================] - 0s 32us/step - loss: 0.0028
Epoch 4/25
250/250 [==============================] - 0s 41us/step - loss: 0.0021
Epoch 5/25
250/250 [==============================] - 0s 32us/step - loss: 0.0016
Epoch 6/25
250/250 [==============================] - 0s 32us/step - loss: 0.0014
Epoch 7/25
250/250 [==============================] - 0s 32us/step - loss: 0.0013
Epoch 8/25
250/250 [==============================] - 0s 48us/step - loss: 0.0011
Epoch 9/25
250/250 [==============================] - 0s 48us/step - loss: 0.0010
Epoch 10/25
250/250 [==============================] - 0s 64us/step - loss: 9.6056e-04
Epoch 11/25
250/250 [==============================] - 0s 80us/step - loss: 8.7928e-04
Epoch 12/25
250/250 [==============================] - 0s 64us/step - loss: 8.5683e-04
Epoch 13/25
250/250 [==============================] - 0s 64us/step - loss: 8.1794e-04
Epoch 14/25
```

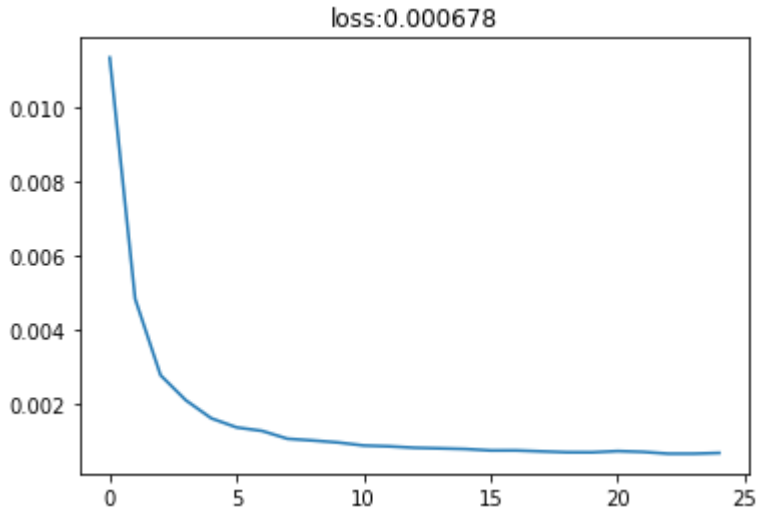
```
250/250 [==============================] - 0s 64us/step - loss: 8.0201e-04
Epoch 15/25
250/250 [==============================] - 0s 48us/step - loss: 7.8484e-04
Epoch 16/25
250/250 [==============================] - 0s 32us/step - loss: 7.4531e-04
Epoch 17/25
250/250 [==============================] - 0s 32us/step - loss: 7.4578e-04
Epoch 18/25
250/250 [==============================] - 0s 32us/step - loss: 7.2034e-04
Epoch 19/25
250/250 [==============================] - 0s 32us/step - loss: 6.9687e-04
Epoch 20/25
250/250 [==============================] - 0s 48us/step - loss: 6.9500e-04
Epoch 21/25
250/250 [==============================] - 0s 64us/step - loss: 7.2835e-04
Epoch 22/25
250/250 [==============================] - 0s 32us/step - loss: 7.0623e-04
Epoch 23/25
250/250 [==============================] - 0s 64us/step - loss: 6.5823e-04
Epoch 24/25
250/250 [==============================] - 0s 80us/step - loss: 6.5840e-04
Epoch 25/25
250/250 [==============================] - 0s 80us/step - loss: 6.7787e-04
```

Now we need to check that our training loss has flattened out and the net is sufficiently trained.

Plot losses

Once we've fit a model, we usually check the training loss curve to make sure it's flattened out. The history returned from model.fit() is a dictionary that has an entry, 'loss', which is the training loss. We want to ensure this has more or less flattened out at the end of our training.

In [25]:
```
# Plot the losses from the fit
plt.plot(history.history['loss'])

# Use the last loss as the title
plt.title('loss:' + str(round(history.history['loss'][-1], 6)))
plt.show()
```

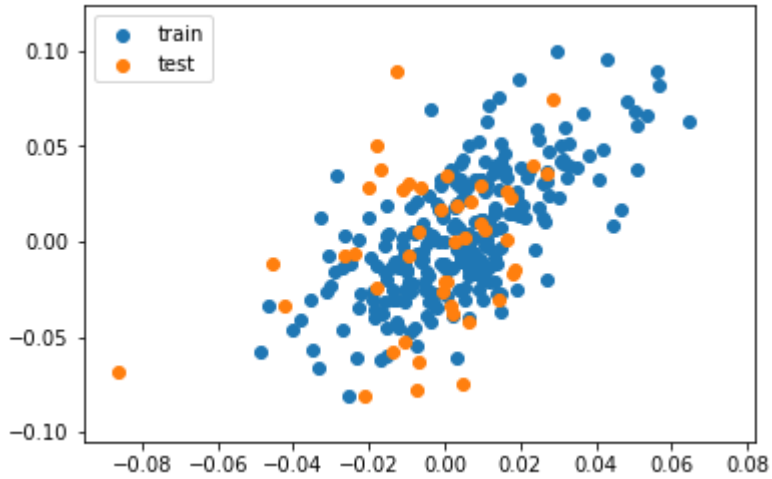


Measure performance

Now that we've fit our neural net, let's check performance to see how well our model is predicting new values. There's not a built-in .score() method like with sklearn models, so we'll use the r2_score() function from sklearn.metrics. This calculates the R2 score given arguments (y_true, y_predicted). We'll also plot our predictions versus actual values again. This will yield some interesting results soon (once we implement our own custom loss function).

In [26]:
```
from sklearn.metrics import r2_score

# Calculate R^2 score
train_preds = model_1.predict(scaled_train_features)
test_preds = model_1.predict(scaled_test_features)
print(r2_score(train_targets, train_preds))
print(r2_score(test_targets, test_preds))

# Plot predictions vs actual
plt.scatter(train_preds, train_targets, label='train')
plt.scatter(test_preds, test_targets, label='test')
plt.legend()
plt.show()
```

```
0.45720268295653965
0.06717148526550754
```



It doesn't look too much different from our other models at this point.

Custom loss function

Up to now, we've used the mean squared error as a loss function. This works fine, but with stock price prediction it can be useful to implement a custom loss function. A custom loss function can help improve our model's performance in specific ways we choose. For example, we're going to create a custom loss function with a large penalty for predicting price movements in the wrong direction. This will

help our net learn to at least predict price movements in the correct direction.

To do this, we need to write a function that takes arguments of (y_true, y_predicted). We'll also use functionality from the backend keras (using tensorflow) to find cases where the true value and prediction don't match signs, then penalize those cases.

In [27]:
```
import keras.losses
import tensorflow as tf

# Create loss function
def sign_penalty(y_true, y_pred):
    penalty = 100.
    loss = tf.where(tf.less(y_true * y_pred, 0), \
                    penalty * tf.square(y_true - y_pred), \
                    tf.square(y_true - y_pred))

    return tf.reduce_mean(loss, axis=-1)

keras.losses.sign_penalty = sign_penalty  # enable use of loss with keras
print(keras.losses.sign_penalty)
```

```
<function sign_penalty at 0x0000017CB22C67B8>
```

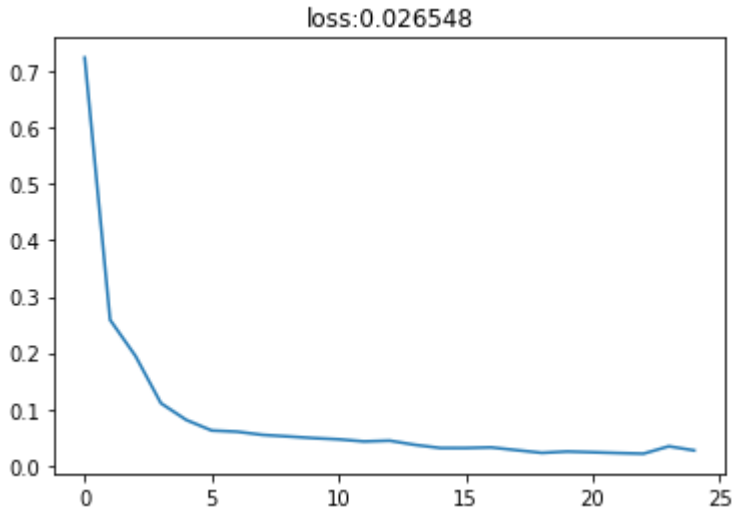
Fit neural net with custom loss function

Now we'll use the custom loss function we just created. This will enable us to alter the model's behavior in useful ways particular to our problem -- it's going to try to force the model to learn how to at least predict price movement direction correctly. All we need to do now is set the loss argument in our .compile() function to our function name, sign_penalty. We'll examine the training loss again to make sure it's flattened out.

In [28]:
```
# Create the model
model_2 = Sequential()
model_2.add(Dense(100, input_dim=scaled_train_features.shape[1], activation='relu'))
model_2.add(Dense(20, activation='relu'))
model_2.add(Dense(1, activation='linear'))

# Fit the model with our custom 'sign_penalty' loss function
model_2.compile(optimizer='adam', loss=sign_penalty)
history = model_2.fit(scaled_train_features, train_targets, epochs=25)
plt.plot(history.history['loss'])
plt.title('loss:' + str(round(history.history['loss'][-1], 6)))
plt.show()
```

```
Epoch 1/25
250/250 [==============================] - 0s 925us/step - loss: 0.7249
Epoch 2/25
250/250 [==============================] - 0s 32us/step - loss: 0.2586
Epoch 3/25
250/250 [==============================] - 0s 48us/step - loss: 0.1944
Epoch 4/25
250/250 [==============================] - 0s 48us/step - loss: 0.1105
Epoch 5/25
250/250 [==============================] - 0s 48us/step - loss: 0.0807
Epoch 6/25
250/250 [==============================] - 0s 48us/step - loss: 0.0620
Epoch 7/25
250/250 [==============================] - 0s 32us/step - loss: 0.0598
Epoch 8/25
250/250 [==============================] - 0s 32us/step - loss: 0.0542
Epoch 9/25
250/250 [==============================] - 0s 32us/step - loss: 0.0514
Epoch 10/25
250/250 [==============================] - 0s 32us/step - loss: 0.0485
Epoch 11/25
250/250 [==============================] - 0s 64us/step - loss: 0.0463
Epoch 12/25
250/250 [==============================] - 0s 64us/step - loss: 0.0425
Epoch 13/25
250/250 [==============================] - 0s 32us/step - loss: 0.0439
Epoch 14/25
250/250 [==============================] - 0s 32us/step - loss: 0.0365
Epoch 15/25
```

```
250/250 [==============================] - 0s 32us/step - loss: 0.0309
Epoch 16/25
250/250 [==============================] - 0s 64us/step - loss: 0.0308
Epoch 17/25
250/250 [==============================] - 0s 32us/step - loss: 0.0319
Epoch 18/25
250/250 [==============================] - 0s 67us/step - loss: 0.0270
Epoch 19/25
250/250 [==============================] - 0s 64us/step - loss: 0.0223
Epoch 20/25
250/250 [==============================] - 0s 48us/step - loss: 0.0246
Epoch 21/25
250/250 [==============================] - 0s 32us/step - loss: 0.0233
Epoch 22/25
250/250 [==============================] - 0s 48us/step - loss: 0.0219
Epoch 23/25
250/250 [==============================] - 0s 48us/step - loss: 0.0207
Epoch 24/25
250/250 [==============================] - 0s 32us/step - loss: 0.0339
Epoch 25/25
250/250 [==============================] - 0s 48us/step - loss: 0.0265
```

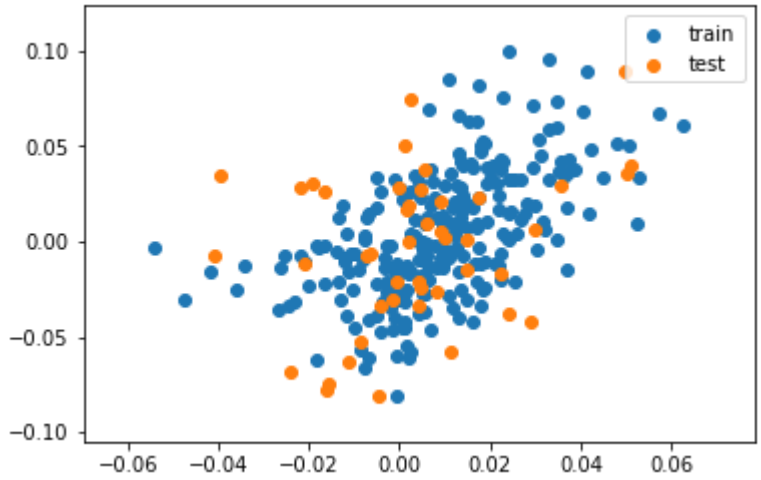


Visualize the results

We've fit our model with the custom loss function, and it's time to see how it is performing. We'll check the R2 values again with sklearn's r2_score() function, and we'll create a scatter plot of predictions versus actual values with plt.scatter(). This will yield some interesting results!

In [29]:
```
# Evaluate R^2 scores
train_preds = model_2.predict(scaled_train_features)
test_preds = model_2.predict(scaled_test_features)
print(r2_score(train_targets, train_preds))
print(r2_score(test_targets, test_preds))

# Scatter the predictions vs actual -- this one is interesting!
plt.scatter(train_preds, train_targets, label='train')
plt.scatter(test_preds, test_targets, label='test')  # plot test set
plt.legend(); plt.show()
```

```
0.2893463616506856
0.010626613271319973
```



Combatting overfitting with dropout

A common problem with neural networks is they tend to overfit to training data. What this means is the scoring metric, like R2 or accuracy, is high for the training set, but low for testing and validation sets, and the model is fitting to noise in the training data.

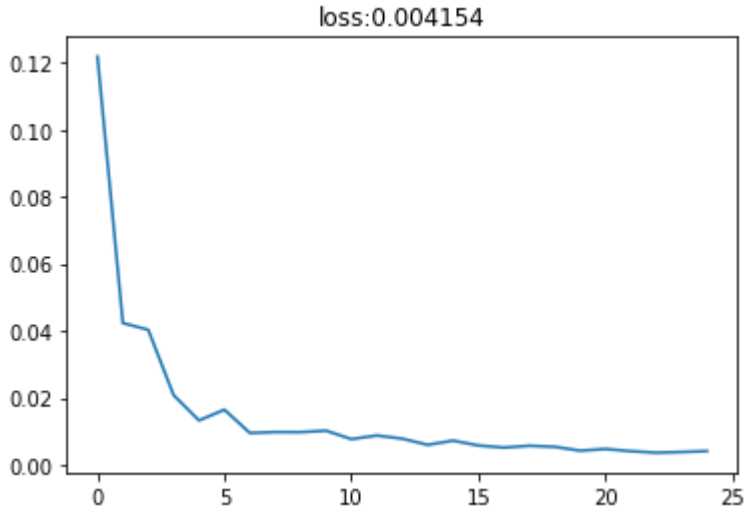
We can work towards preventing overfitting by using dropout. This randomly drops some neurons during the training phase, which helps prevent the net from fitting noise in the training data. keras has a Dropout layer that we can use to accomplish this. We need to set the dropout rate, or fraction of connections dropped during training time. This is set as a decimal between 0 and 1 in the Dropout() layer.

We're going to go back to the mean squared error loss function for this model.

In [30]:
```
from keras.layers import Dropout

# Create model with dropout
model_3 = Sequential()
model_3.add(Dense(100, input_dim=scaled_train_features.shape[1], activation='relu'))
model_3.add(Dropout(0.2))
model_3.add(Dense(20, activation='relu'))
model_3.add(Dense(1, activation='linear'))

# Fit model with mean squared error loss function
model_3.compile(optimizer='adam', loss='mse')
history = model_3.fit(scaled_train_features, train_targets, epochs=25)
plt.plot(history.history['loss'])
plt.title('loss:' + str(round(history.history['loss'][-1], 6)))
plt.show()
```

```
Epoch 1/25
250/250 [==============================] - 0s 1ms/step - loss: 0.1218
Epoch 2/25
250/250 [==============================] - 0s 67us/step - loss: 0.0423
Epoch 3/25
250/250 [==============================] - 0s 32us/step - loss: 0.0403
Epoch 4/25
250/250 [==============================] - 0s 53us/step - loss: 0.0208
Epoch 5/25
250/250 [==============================] - 0s 52us/step - loss: 0.0133
Epoch 6/25
250/250 [==============================] - 0s 48us/step - loss: 0.0165
Epoch 7/25
250/250 [==============================] - 0s 54us/step - loss: 0.0096
Epoch 8/25
250/250 [==============================] - 0s 32us/step - loss: 0.0098
Epoch 9/25
250/250 [==============================] - 0s 48us/step - loss: 0.0098
Epoch 10/25
250/250 [==============================] - 0s 48us/step - loss: 0.0102
Epoch 11/25
250/250 [==============================] - 0s 48us/step - loss: 0.0077
Epoch 12/25
250/250 [==============================] - 0s 48us/step - loss: 0.0088
Epoch 13/25
250/250 [==============================] - 0s 48us/step - loss: 0.0078
```

```
Epoch 14/25
250/250 [==============================] - 0s 48us/step - loss: 0.0060
Epoch 15/25
250/250 [==============================] - 0s 64us/step - loss: 0.0073
Epoch 16/25
250/250 [==============================] - 0s 64us/step - loss: 0.0058
Epoch 17/25
250/250 [==============================] - 0s 48us/step - loss: 0.0052
Epoch 18/25
250/250 [==============================] - 0s 48us/step - loss: 0.0057
Epoch 19/25
250/250 [==============================] - 0s 64us/step - loss: 0.0054
Epoch 20/25
250/250 [==============================] - 0s 57us/step - loss: 0.0043
Epoch 21/25
250/250 [==============================] - 0s 60us/step - loss: 0.0048
Epoch 22/25
250/250 [==============================] - 0s 48us/step - loss: 0.0041
Epoch 23/25
250/250 [==============================] - 0s 64us/step - loss: 0.0037
Epoch 24/25
250/250 [==============================] - 0s 48us/step - loss: 0.0039
Epoch 25/25
250/250 [==============================] - 0s 64us/step - loss: 0.0042
```



Dropout helps the model generalized a bit better to unseen data. Comments: This is like the random forest which makes the samples more

independent?

Ensembling models

One approach to improve predictions from machine learning models is ensembling. A basic approach is to average the predictions from multiple models. A more complex approach is to feed predictions of models into another model, which makes final predictions. Both approaches usually improve our overall performance (as long as our individual models are good). If you remember, random forests are also using ensembling of many decision trees.

To ensemble our neural net predictions, we'll make predictions with the 3 models we just created -- the basic model, the model with the custom loss function, and the model with dropout. Then we'll combine the predictions with numpy's .hstack() function, and average them across rows with np.mean(predictions, axis=1).

In [31]:
```
# Make predictions from the 3 neural net models
train_pred1 = model_1.predict(scaled_train_features)
test_pred1 = model_1.predict(scaled_test_features)

train_pred2 = model_2.predict(scaled_train_features)
test_pred2 = model_2.predict(scaled_test_features)

train_pred3 = model_3.predict(scaled_train_features)
test_pred3 = model_3.predict(scaled_test_features)

# Horizontally stack predictions and take the average across rows
train_preds = np.mean(np.hstack((train_pred1, train_pred2, train_pred3)), axis=1)
test_preds = np.mean(np.hstack((test_pred1, test_pred2, test_pred3)), axis=1)
print(test_preds[-5:])
```

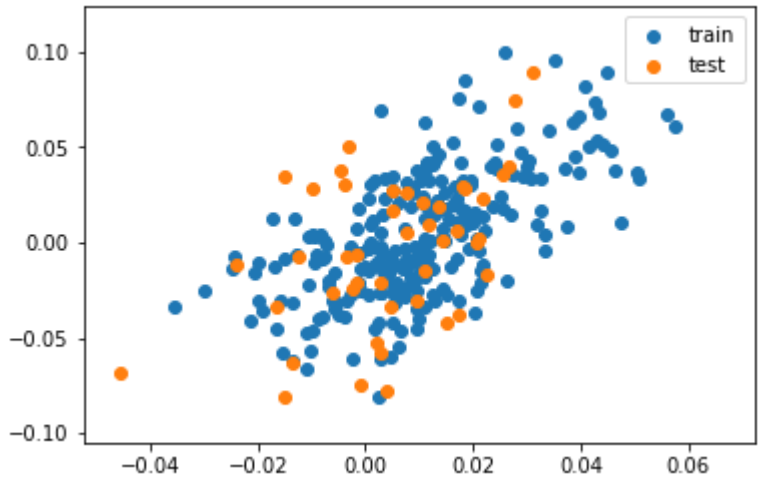
```
[ 0.02188003  0.02787314  0.00514324  0.01063638 -0.00382689]
```

See how the ensemble performed

Let's check performance of our ensembled model to see how it's doing. We should see roughly an average of the R2 scores, as well as a scatter plot that is a mix of our previous models' predictions. The bow-tie shape from the custom loss function model should still be a bit visible, but the edges near x=0 should be softer.

In [32]:
```
from sklearn.metrics import r2_score

# Evaluate the R^2 scores
print(r2_score(train_targets, train_preds))
print(r2_score(test_targets, test_preds))

# Scatter the predictions vs actual -- this one is interesting!
plt.scatter(train_preds, train_targets, label='train')
plt.scatter(test_preds, test_targets, label='test')
plt.legend(); plt.show()
```

```
0.3430407365008251
0.16745414447588025
```



Our R^2 values are around the average of the 3 models we ensembled. Notice the plot also looks like the bow-tie shape has been softened a bit.

Machine learning with modern portfolio theory

- Learn how to use modern portfolio theory (MPT) and the Sharpe ratio to plot and find optimal stock portfolios.
- Use machine learning to predict the best portfolios.
- Evaluate performance of the ML-predicted portfolios.

Join stock DataFrames and calculate returns

Our first step towards calculating modern portfolio theory (MPT) portfolios is to get daily and monthly returns. Eventually we're going to get the best portfolios of each month based on the Sharpe ratio. The easiest way to do this is to put all our stock prices into one DataFrame, then to resample them to the daily and monthly time frames. We need daily price changes to calculate volatility, which we will use as our measure of risk.

Below is a numerical way of calculating best portfolio. Going through these steps will give a clear understanding of MPT.

In [1]:
```
#Prepare data
import pandas as pd
import matplotlib.pyplot as plt

lng_df = pd.read_csv("LNG.csv", index_col = 0, parse_dates = True)
spy_df = pd.read_csv("spy.csv", index_col = 0, parse_dates = True)
smlv_df = pd.read_csv("smlv.csv", index_col = 0, parse_dates = True)

lng_df = lng_df.loc['1994-Apr-04':'2018-May-31',:]
spy_df = spy_df.loc['1993-Jan-29':'2018-May-31',:]
smlv_df = smlv_df.loc['2013-Feb-21':'2018-May-31',:]

lng_df = lng_df.drop(['Adj_Volume'], axis = 1)
lng_df.columns = ['LNG'] #This change the original Adj_Close to LNG

spy_df = spy_df.drop(['Adj_Volume'], axis = 1)
spy_df.columns = ['SPY']
smlv_df = smlv_df.drop(['Adj_Volume'], axis = 1)
smlv_df.columns = ['SMLV']
```

In [2]:
```
# Join 3 stock DataFrame together
full_df = pd.concat([lng_df, spy_df, smlv_df], axis=1).dropna()

# Resample the full DataFrame to monthly timeframe
monthly_df = full_df.resample('BMS').first() #business month start frequency

# Calculate daily returns of stocks
returns_daily = full_df.pct_change()
print(len(returns_daily))

# Calculate monthly returns of the stocks
returns_monthly = monthly_df.pct_change().dropna()
print(len(returns_monthly))
print(returns_monthly.tail())
```

```
1297
62
                 LNG       SPY      SMLV
Date
2017-12-01  0.019558  0.027069  0.029058
2018-01-01  0.128300  0.021450 -0.010725
2018-02-01  0.057770  0.047662 -0.003823
2018-03-01 -0.103353 -0.049293 -0.048131
2018-04-02  0.021396 -0.034367  0.009406
```

Calculate covariances for volatility

In MPT, we quantify risk via volatility. The math for calculating portfolio volatility is complex, and it requires daily returns covariances. We'll now loop through each month in the returns_monthly DataFrame, and calculate the covariance of the daily returns.

With pandas datetime indices, we can access the month and year with df.index.month and df.index.year. We'll use this to create a mask for returns_daily that gives us the daily returns for the current month and year in the loop. We then use the mask to subset the DataFrame like this: df[mask]. This gets entries in the returns_daily DataFrame which are in the current month and year in each cycle of the loop. Finally, we'll use pandas' .cov() method to get the covariance of daily returns.

Comments

- I implicitly employ the MPT to reduce my risk of trading all the time.
- The derivation of MPT can be understood easily with the derivation of SVM.

In [37]:
```
# Daily covariance of stocks (for each monthly period)
covariances = {} #Each element is a matrix.
rtd_idx = returns_daily.index
# print(rtd_idx)

for i in returns_monthly.index:
    # Mask daily returns for each month and year, and calculate covariance
    mask = (rtd_idx.month == i.month) & (rtd_idx.year == i.year)

    # Use the mask to get daily returns for the current month and year of monthy returns index
    covariances[i] = returns_daily[mask].cov()
#     print(returns_daily[mask]) #This shows returns_daily[mask] gives one month's daily data for three stocks.
#     print(i)
#     print(covariances[i])
#     print('--------------------------')

print(i)
print('----------------------------')
print(covariances[i]) #One covariance matrix for each month
print('----------------------------')
print(len(covariances))
```

```
2018-04-02 00:00:00
----------------------------
           LNG       SPY      SMLV
LNG   0.000366  0.000192  0.000146
SPY   0.000192  0.000173  0.000127
SMLV  0.000146  0.000127  0.000103
----------------------------
62
```

The covariances will allow us to calculate volatility in our next step.

Comments

- Because it is variance not the normalized correlation, so the diagonal elements are not one.
- Covariance matrix will be used in calculating portfolio variance (or square root version, volatility).
- Distinguish precisely the covariance, correlation, and volatility.

Answers can be found in https://en.wikipedia.org/wiki/Modern_portfolio_theory (https://en.wikipedia.org/wiki/Modern_portfolio_theory)

Calculate portfolios

We'll now generate portfolios to find each month's best one. numpy's random.random() generates random numbers from a **uniform distribution"**, then we normalize them so they sum to 1 using the /= operator. We use these weights to calculate returns and volatility. Returns are sums of weights times individual returns. Volatility is more complex, and involves the covariances of the different stocks.

Finally we'll store the values in dictionaries for later use, with months' dates as keys.

In this case, we will only **generate 10 portfolios for each date** so the code will run faster (I used 100), but in a real-world use-case you'd want to use more like 1000 to 5000 randomly-generated portfolios for a few stocks.

In [55]:
```
import numpy as np
portfolio_returns, portfolio_volatility, portfolio_weights = {}, {}, {}
# print(covariances.keys())

# Get portfolio performances at each month
for date in sorted(covariances.keys()):
    cov = covariances[date]

    for portfolio in range(100):
        weights = np.random.random(3) # We have only three stocks.
        weights /= np.sum(weights) # /= divides weights by their sum to normalize. We obtain three values that su
        returns = np.dot(weights, returns_monthly.loc[date]) #total return
        #This is a single value return, which is the sum of three weighted monthly returns of three stocks.

        volatility = np.sqrt(np.dot(weights.T, np.dot(cov, weights))) #total volatility
        # https://en.wikipedia.org/wiki/Modern_portfolio_theory

        portfolio_returns.setdefault(date, []).append(returns)
        portfolio_volatility.setdefault(date, []).append(volatility)
        portfolio_weights.setdefault(date, []).append(weights)

print(portfolio_weights[date][0])
# print(portfolio_weights.keys())
```

```
[0.7397415  0.22410883 0.03614966]
```

Comments:

- For each date, we calculate 100 possible portfolios, each with a return and volatility.
- Later we will choose the best one, which has best return per volatility.
- Note for each date (one for each month), we have many replicates of weights, and calculate the portfolio volatility from each weights.

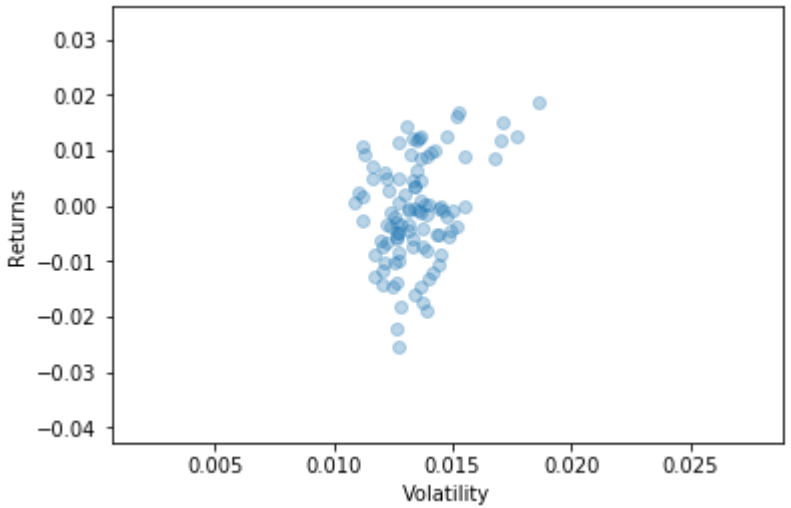
Plot efficient frontier

We can finally plot the results of our MPT portfolios, which shows the "efficient frontier". This is a plot of the volatility vs the returns. This can help us visualize our risk-return possibilities for portfolios. The upper left boundary of the points is the best we can do (**highest return for a given risk**), and that is the efficient frontier.

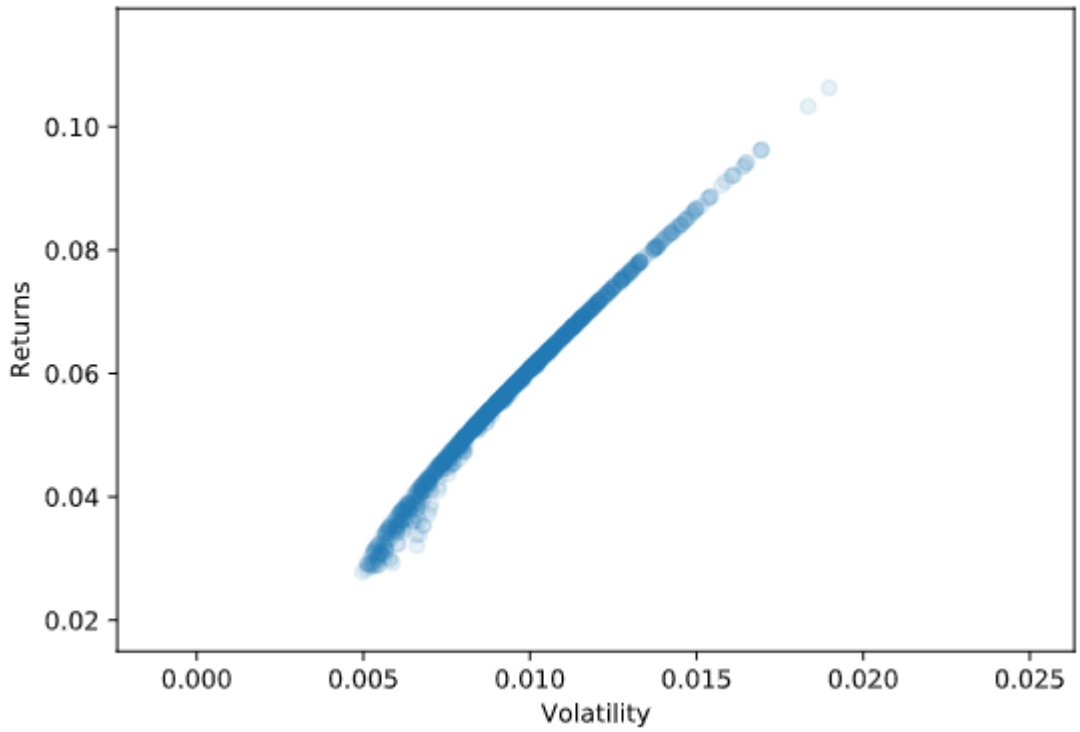
To create this plot, we will use the latest date in our covariances dictionary which we created a few exercises ago. This has dates as keys, so we'll get the sorted keys using sorted() and .keys(), then get the last entry with Python indexing ([-1]). Lastly we'll use matplotlib to scatter variance vs returns and see the efficient frontier for the latest date in the data.

In [58]:
```
# Get latest date of available data
date = sorted(covariances.keys())[-1]
print(date)

# Plot efficient frontier
# warning: this can take at least 10s for the plot to execute...
plt.scatter(x=portfolio_volatility[date], y=portfolio_returns[date], alpha=0.3)
plt.xlabel('Volatility')
plt.ylabel('Returns')
plt.show()
```

```
2018-04-02 00:00:00
```



The following is from DataCamp. Why so different from mine (Note I used 200 in the code, while the original is 10 for parameter. This number determines the number of dots in the figure). Often the efficient frontier will be a bullet shape, but if the returns are all positive then it may look like this. The data I used also have negative return while in the DataCamp all positive.



Get best Sharpe ratios

We need to find **the "ideal" portfolios for each date** so we can use them **as targets for machine learning**. We'll loop through each date in portfolio_returns, then loop through the portfolios we generated with portfolio_returns[date]. We'll then calculate the **Sharpe ratio, which is the return divided by volatility (assuming a no-risk return of 0)**.

We use enumerate() to loop through the returns for the current date (portfolio_returns[date]) and keep track of the index with i. Then we use the current date and current index to get the volatility of each portfolio with portfolio_volatility[date][i]. Finally, we get the index of the best Sharpe ratio for each date using np.argmax(). We'll use this index to get the ideal portfolio weights soon.

Comments
Note the way of keeping track of a special index with enumerate and argmax.

In [60]:
```
# Empty dictionaries for sharpe ratios and best sharpe indexes by date
sharpe_ratio, max_sharpe_idxs = {}, {}

# Loop through dates and get sharpe ratio for each portfolio
for date in portfolio_returns.keys():
    for i, ret in enumerate(portfolio_returns[date]):

        # Divide returns by the volatility for the date and index, i
        sharpe_ratio.setdefault(date, []).append(ret / portfolio_volatility[date][i])

    # Get the index of the best sharpe ratio for each date
    max_sharpe_idxs[date] = np.argmax(sharpe_ratio[date])
#     print(max_sharpe_idxs[date]) #output: 13, 77, .....

print(portfolio_returns[date][max_sharpe_idxs[date]])
```

```
0.016842900598001916
```

DataCamp Results
0.05053609644892129
We've got our **best Sharpe ratios, which we'll use to create targets for machine learning.**

Calculate EWMAs

We will now work towards creating some features to be able to predict our ideal portfolios. We will simply use the price movement as a feature for now. To do this we will create a daily exponentially-weighted moving average (EWMA), then resample that to the monthly timeframe. Finally, we'll shift the monthly moving average of price one month in the future, so we can use it as a feature for predicting future portfolios.

Comments:

- Note in the earlier chapters, we shift the price to the left for future target.
- Here we shift to the right. Figure out why. Shifting feature right and shifting target left are equivalent.

In [66]:
```
# Calculate exponentially-weighted moving average of daily returns
print(returns_daily.head())
ewma_daily = returns_daily.ewm(span=30).mean()

print(ewma_daily.head())
print('**********************************')

# Resample daily returns to first business day of the month with the first day for that month
ewma_monthly = ewma_daily.resample('BMS').first()
print(ewma_monthly.head())
print(ewma_monthly.tail())

# Shift ewma for the month by 1 month forward so we can use it as a feature for future predictions
ewma_monthly = ewma_monthly.shift(1).dropna()

# print(ewma_monthly.iloc[-1])
print('-----------------------------------------------')
print(ewma_monthly.head())
print(ewma_monthly.tail())
```

```
                 LNG       SPY      SMLV
Date
2013-02-21       NaN       NaN       NaN
2013-02-22  0.038595  0.009773  0.008058
2013-02-25 -0.026203 -0.019027 -0.008826
2013-02-26  0.034736  0.006846 -0.004872
2013-02-27  0.019858  0.012598  0.004559
                 LNG       SPY      SMLV
Date
2013-02-21       NaN       NaN       NaN
2013-02-22  0.038595  0.009773  0.008058
2013-02-25  0.005116 -0.005107 -0.000666
2013-02-26  0.015655 -0.000854 -0.002162
2013-02-27  0.016813  0.002852 -0.000310
**********************************
                 LNG       SPY      SMLV
Date
2013-02-01  0.038595  0.009773  0.008058
2013-03-01  0.006870  0.002056  0.000039
2013-04-01  0.013170  0.001171  0.000966
2013-05-01  0.004331  0.000800  0.000598
2013-06-03  0.000820  0.000626  0.000044
```

```
                 LNG       SPY      SMLV
Date
2017-12-01  0.001067  0.001504  0.001337
2018-01-01  0.006037  0.001228  0.000157
2018-02-01  0.003666  0.001327 -0.000560
2018-03-01 -0.004304 -0.001003 -0.001670
2018-04-02 -0.000751 -0.002635 -0.001181
-----------------------------------------------
                 LNG       SPY      SMLV
Date
2013-03-01  0.038595  0.009773  0.008058
2013-04-01  0.006870  0.002056  0.000039
2013-05-01  0.013170  0.001171  0.000966
2013-06-03  0.004331  0.000800  0.000598
2013-07-01  0.000820  0.000626  0.000044
                 LNG       SPY      SMLV
Date
2017-12-01  0.002789  0.000925 -0.000038
2018-01-01  0.001067  0.001504  0.001337
2018-02-01  0.006037  0.001228  0.000157
2018-03-01  0.003666  0.001327 -0.000560
2018-04-02 -0.004304 -0.001003 -0.001670
```

Make features and targets

To use machine learning to pick the best portfolio, we need to generate features and targets. Our features were just created in the last exercise – the exponentially weighted moving averages of prices. Our targets will be the best portfolios we found from the highest Sharpe ratio.

We will use pandas' .iterrows() method to get the index, value pairs for the ewma_monthly DataFrame. We'll set the current value of ewma_monthly in the loop to be our features. Then we'll use the index of the best Sharpe ratio (from max_sharpe_idxs) to get the best portfolio_weights for each month and set that as a target.

In [27]:
```
targets, features = [], []

# Create features from price history and targets as ideal portfolio
for date, ewma in ewma_monthly.iterrows():

    # Get the index of the best sharpe ratio
    best_idx = max_sharpe_idxs[date]
    targets.append(portfolio_weights[date][best_idx])
    features.append(ewma)  # add ewma to features


print(len(targets),len(features))
print(features[0:3])
print('----------------')
print(targets[0:3])
print('----------------')
targets = np.array(targets)
features = np.array(features)
```

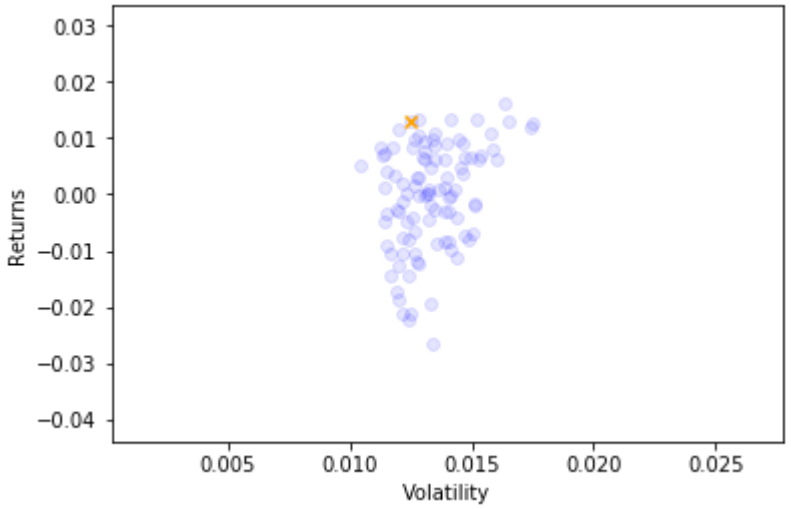
```
62 62
[LNG     0.038595
SPY     0.009773
SMLV    0.008058
Name: 2013-03-01 00:00:00, dtype: float64, LNG     0.006870
SPY     0.002056
SMLV    0.000039
Name: 2013-04-01 00:00:00, dtype: float64, LNG     0.013170
SPY     0.001171
SMLV    0.000966
Name: 2013-05-01 00:00:00, dtype: float64]
----------------
[array([2.87410727e-01, 7.11934373e-01, 6.54900272e-04]), array([0.84048976, 0.02357857, 0.13593167]), array
([0.06069156, 0.89976979, 0.03953865])]
----------------
```

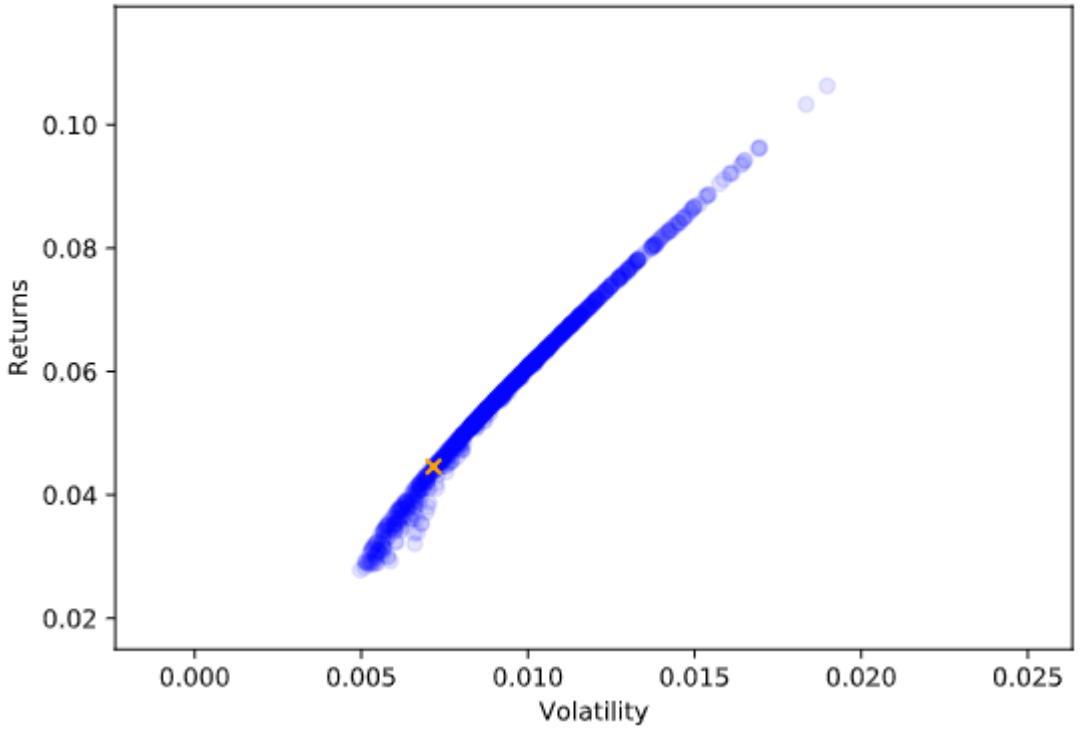
Plot efficient frontier with best Sharpe ratio

Let's now plot the efficient frontier again, but add a marker for the portfolio with the best Sharpe index. Visualizing our data is always a good idea to better understand it.

Recall the efficient frontier is plotted in a scatter plot of portfolio volatility on the x-axis, and portfolio returns on the y-axis. We'll get the latest date we have in our data from covariances.keys(), although any of the portfolio_returns, etc, dictionaries could be used as well to get the date. Then we get volatilities and returns for the latest date we have from our portfolio_volatility and portfolio_returns. Finally we get the index of the portfolio with the best Sharpe index from max_sharpe_idxs[date], and plot everything with plt.scatter().

In [157]:
```
# Get most recent (current) returns and volatility
date = sorted(covariances.keys())[-1]
cur_returns = portfolio_returns[date]
cur_volatility = portfolio_volatility[date]

# Plot efficient frontier with sharpe as point
plt.scatter(x=cur_volatility, y=cur_returns, alpha=0.1, color='blue')
best_idx = max_sharpe_idxs[date]

# Place an orange "X" on the point with the best Sharpe ratio
plt.scatter(x=cur_volatility[best_idx], y=cur_returns[best_idx], marker='x', color='orange')
plt.xlabel('Volatility')
plt.ylabel('Returns')
plt.show()
```





The best portfolio according to Sharpe is usually somewhere in that area where the orange x is.

Make predictions with a random forest

In order to fit a machine learning model to predict ideal portfolios, we need to create train and test sets for evaluating performance. We will do this as we did in previous chapters, where we take our features and targets arrays, and split them based on a train_size we set. Often the train size may be around 70-90% of our data.

We then fit our model (a random forest in this case) to the training data, and evaluate the R2 scores on train and test using .score() from our model. In this case, the hyperparameters have been set for you, but usually you'd want to do a search with ParameterGrid like we did in previous chapters.

In [158]:
```
# Make train and test features
train_size = int(0.85 * features.shape[0])
train_features = features[:train_size]
test_features = features[train_size:]
train_targets = targets[:train_size]
test_targets = targets[train_size:]

# Fit the model and check scores on train and test
rfr = RandomForestRegressor(n_estimators=300, random_state=42)
rfr.fit(train_features, train_targets)
print(rfr.score(train_features, train_targets))
print(rfr.score(test_features, test_targets))
```

```
0.8347443966179858
-0.32884626102590336
```

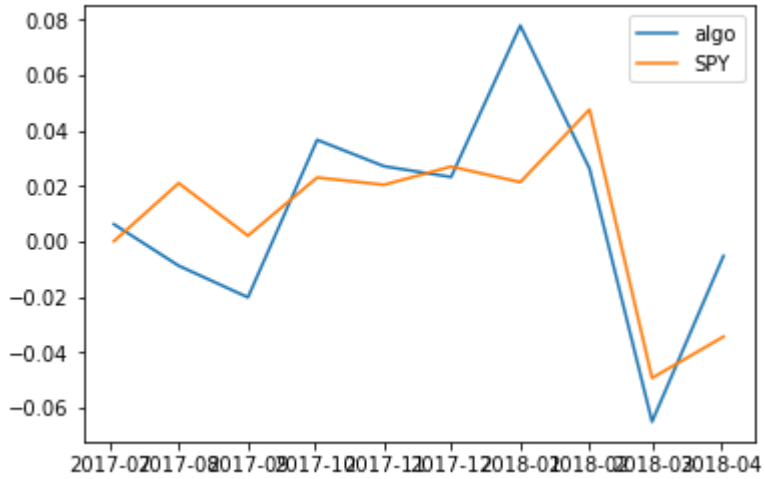
```
0.8216455876218289
-0.5641150131777833
```

Get predictions and first evaluation

Now that we have a trained random forest model (rfr), we want to use it to get predictions on the test set. We do this to evaluate our model's performance – at a basic level, is it doing as well or better than just buying the index, SPY?

We'll use the typical sklearn .predict(features) method, then multiply our monthly returns by our portfolio predictions. We sum these up with np.sum() since this will have 3 rows for each month. Then we plot both the monthly returns from our predictions, as well as SPY and compare the two.

In [159]:
```
# Get predictions from model on train and test
train_predictions = rfr.predict(train_features)
test_predictions = rfr.predict(test_features)

# Calculate and plot returns from our RF predictions and the SPY returns
test_returns = np.sum(returns_monthly.iloc[train_size:] * test_predictions, axis=1)
plt.plot(test_returns, label='algo')
plt.plot(returns_monthly['SPY'].iloc[train_size:], label='SPY')
plt.legend()
plt.show()
```



We're doing a little better than SPY sometimes, and other times not. Let's see how it adds up...

Evaluate returns

Let's now see how our portfolio selection would perform as compared with just investing in the SPY. We'll do this to see if our predictions are promising, despite the low R2 value.

We will set a starting value for our investment of $1000, then loop through the returns from our predictions as well as from SPY. We'll use the monthly returns from our portfolio selection and SPY and apply them to our starting cash balance. From this we will get a month-by-month picture of how our investment is doing, and we can see how our predictions did overall vs the SPY. Next, we can plot our portfolio from our predictions and compare it to SPY.

In [160]:
```
# Calculate the effect of our portfolio selection on a hypothetical $1k investment
cash = 1000
algo_cash, spy_cash = [cash], [cash]  # set equal starting cash amounts
for r in test_returns:
    cash *= 1 + r
    algo_cash.append(cash)

# Calculate performance for SPY
cash = 1000  # reset cash amount
for r in returns_monthly['SPY'].iloc[train_size:]:
    cash *= 1 + r
    spy_cash.append(cash)

print('algo returns:', (algo_cash[-1] - algo_cash[0]) / algo_cash[0])
print('SPY returns:', (spy_cash[-1] - spy_cash[0]) / spy_cash[0])
```

```
algo returns: 0.09583243810394675
SPY returns: 0.07811831329855999
```

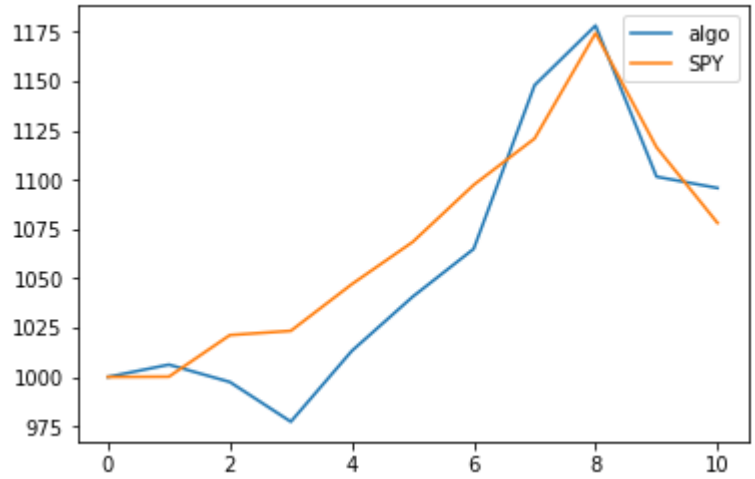
```
algo returns: 0.1360523316630688
SPY returns: 0.10942393961703784
```

Plot returns

Lastly, we'll plot the performance of our machine-learning-generated portfolio versus just holding the SPY. We can use this as an evaluation to see if our predictions are doing well or not.

Since we already have algo_cash and spy_cash created, all we need to do is provide them to plt.plot() to display. We'll also set the label for the datasets with legend in plt.plot().

In [161]:
```
# Plot the algo_cash and spy_cash to compare overall returns
plt.plot(algo_cash, label='algo')
plt.plot(spy_cash, label='SPY')
plt.legend()  # show the legend
plt.show()
```



Tools for bigger data

Python 3 multiprocessing
Dask
Spark
AWS or other cloud solutions

Get more and better data

Many experts and Nobel laureats demonstrate that it is very difficult, if not impossible, to predict stock future price by the historic price alone. So we need more data.

Data in this course: From Quandl.com/EOD (free subset available)
Alternative and other data:
satellite images

sentiment analysis (e.g. PsychSignal)
analyst predictions
fundamentals data

It is a very good advice to invest in low-cost ETF.