absorp_ratio_m2_ex5

October 25, 2018

1 Course Project: Trading Strategy based on PCA

Welcome to your course project. This exercise gives you a hands-on experience to use PCA to:

- construct eigen-portfolios
- implement a measure of market systemic risk
- develop simple trading strategy

Instructions: - You will be using Python 3. - Avoid using for-loops and while-loops, unless you are explicitly told to do so. - Do not modify the (# GRADED FUNCTION [function name]) comment in some cells. Your work would not be graded if you change this. Each cell containing that comment should only contain one function. - After coding your function, run the cell right below it to check if your result is correct.

After this assignment you will: - Be able to use PCA to construct eigen-portfolios - Be able to use PCA to calculate a measure of market systemic risk - Be able to implement and analyze performance of portfolio strategy

Let's get started!

1.1 About iPython Notebooks

iPython Notebooks are interactive coding environments embedded in a webpage. You will be using iPython notebooks in this class. You only need to write code between the ### START CODE HERE ### and ### END CODE HERE ### comments. After writing your code, you can run the cell by either pressing "SHIFT"+"ENTER" or by clicking on "Run Cell" (denoted by a play symbol) in the upper bar of the notebook.

We will often specify "(X lines of code)" in the comments to tell you about how much code you need to write. It is just a rough estimate, so don't feel bad if your code is longer or shorter.

1.2 1 - Packages

First, let's run the cell below to import all the packages that you will need during this assignment. - numpy is the fundamental package for scientific computing with Python. - pandas Python data analysis library - pandas scikit-learn - machine learning in Python. - matplotlib is a famous library to plot graphs in Python.

```
In [1]: import pandas as pd
        import numpy as np
```

```
import sklearn.decomposition
import tensorflow as tf
from tensorflow.contrib.layers import fully_connected
import sys

import matplotlib.pyplot as plt
%matplotlib inline

print("Package Versions:")
print("  scikit-learn: %s" % sklearn.__version__)
print("  tensorflow: %s" % tf.__version__)

sys.path.append("..")
import grading

try:
    import sklearn.model_selection
    import sklearn.linear_model
except:
    print("Looks like an older version of sklearn package")

try:
    import pandas as pd
    print("  pandas: %s"% pd.__version__)
except:
    print("Missing pandas package")
```

```
Package Versions:
  scikit-learn: 0.18.1
  tensorflow: 1.8.0
  pandas: 0.19.2
```

In [2]:
```
### ONLY FOR GRADING. DO NOT EDIT ###
submissions=dict()
assignment_key="LztgGBBtEeiaYgrsftMrjA"
all_parts=["oZXnf", "ahjZa", "9tUbW","wjLiO"]
### ONLY FOR GRADING. DO NOT EDIT ###
```

In [75]:
```
# COURSERA_TOKEN = # the key provided to the Student under his/her email on submission
# COURSERA_EMAIL = # the email
COURSERA_TOKEN="Y3jMwrYGzVHMxG7K"
COURSERA_EMAIL="cilsya@yahoo.com"
```

Dataset: daily prices of stocks from S&P 500 index For this exercise we will be working with S&P 500 Index stock prices dataset. The following cell computes returns based for a subset of S&P 500 index stocks. It starts with stocks price data:

```
In [4]: import os
        # load dataset

        asset_prices = pd.read_csv(os.getcwd() + '/data/spx_holdings_and_spx_closeprice.csv',
                                   date_parser=lambda dt: pd.to_datetime(dt, format='%Y-%m-%d'),
                                   index_col = 0).dropna()
        n_stocks_show = 12
        print('Asset prices shape', asset_prices.shape)
        asset_prices.iloc[:, :n_stocks_show].head()
```

```
Asset prices shape (3493, 419)
```

```
Out[4]:                   A       AA     AAPL     ABC      ABT     ADBE      ADI  \
        2000-01-27  46.1112  78.9443  3.9286  4.5485  13.7898  15.6719  48.0313
        2000-01-28  45.8585  77.8245  3.6295  4.5485  14.2653  14.3906  47.7500
        2000-01-31  44.5952  78.0345  3.7054  4.3968  14.5730  13.7656  46.7500
        2000-02-01  47.8377  80.7640  3.5804  4.5333  14.7128  13.9688  49.0000
        2000-02-02  51.5434  83.4934  3.5290  4.5788  14.7968  15.3281  48.1250

                        ADM      ADP    ADSK      AEE      AEP
        2000-01-27  10.8844  39.5477  8.1250  32.9375  33.5625
        2000-01-28  10.7143  38.5627  7.7188  32.3125  33.0000
        2000-01-31  10.6576  37.3807  7.6406  32.5625  33.5000
        2000-02-01  10.8844  37.9717  7.9219  32.5625  33.6875
        2000-02-02  10.6576  35.9032  7.9688  32.5625  33.6250
```

1.2.1 Calculate daily log-returns

```
In [5]: asset_returns = np.log(asset_prices) - np.log(asset_prices.shift(1))
        asset_returns = asset_prices.pct_change(periods=1)
        asset_returns = asset_returns.iloc[1:, :]
        asset_returns.iloc[:, :n_stocks_show].head()
```

```
Out[5]:                    A        AA      AAPL       ABC       ABT      ADBE  \
        2000-01-28 -0.005480 -0.014185 -0.076134  0.000000  0.034482 -0.081758
        2000-01-31 -0.027548  0.002698  0.020912 -0.033352  0.021570 -0.043431
        2000-02-01  0.072710  0.034978 -0.033735  0.031045  0.009593  0.014761
        2000-02-02  0.077464  0.033795 -0.014356  0.010037  0.005709  0.097310
        2000-02-03  0.016340 -0.031014  0.045537 -0.006617  0.005670  0.126402

                         ADI       ADM       ADP      ADSK       AEE       AEP
        2000-01-28 -0.005857 -0.015628 -0.024907 -0.049994 -0.018975 -0.016760
        2000-01-31 -0.020942 -0.005292 -0.030651 -0.010131  0.007737  0.015152
        2000-02-01  0.048128  0.021281  0.015810  0.036816  0.000000  0.005597
        2000-02-02 -0.017857 -0.020837 -0.054475  0.005920  0.000000 -0.001855
        2000-02-03  0.098701  0.000000  0.067217  0.035288  0.011516  0.033457
```

```
In [6]: def center_returns(r_df):
            """
```

```
    Normalize, i.e. center and divide by standard deviation raw asset returns data

    Arguments:
    r_df -- a pandas.DataFrame of asset returns

    Return:
    normed_df -- normalized returns
    """
    mean_r = r_df.mean(axis=0)
    sd_r = r_df.std(axis=0)
    normed_df = (r_df - mean_r) / sd_r
    return normed_df
```

In [7]:
```
normed_r = center_returns(asset_returns)
normed_r.iloc[:, :n_stocks_show].head()
```

Out[7]:

	A	AA	AAPL	ABC	ABT	ADBE
2000-01-28	-0.190054	-0.513710	-2.714709	-0.049779	2.182933	-2.684131
2000-01-31	-0.898232	0.096888	0.688156	-1.757230	1.355644	-1.438899
2000-02-01	2.319164	1.264327	-1.227995	1.539597	0.588289	0.451774
2000-02-02	2.471738	1.221529	-0.548494	0.464060	0.339454	3.133764
2000-02-03	0.510174	-1.122380	1.551619	-0.388563	0.336944	4.078966

	ADI	ADM	ADP	ADSK	AEE	AEP
2000-01-28	-0.212461	-0.766996	-1.540731	-1.803947	-1.372991	-0.994169
2000-01-31	-0.720771	-0.279098	-1.891884	-0.391433	0.547298	0.871919
2000-02-01	1.606541	0.975244	0.948126	1.272129	-0.008894	0.313197
2000-02-02	-0.616815	-1.012898	-3.348109	0.177340	-0.008894	-0.122594
2000-02-03	3.310577	-0.029293	4.090396	1.217955	0.818989	1.942390

Now we are ready to compute Absorption Ratio(AR). We do so by defining a moving look back window over which we collect returns for computing PCA. We start off from the earliest historical data and march forward moving by step_size, which we also choose arbitrary. For each such window we compute PCA and AR, fixing in advance number of components in the enumerator. Specifically, for we use the following hyper-parameters:

Part 1 (Implement exponentially-weighted) Instructions: Implement exponent_weighting function which returns a sequence of w_j as np.array. See below:

Define sequence of X_j where $j \subset [N, 0]$, an integer taking all values in the interval from 0 to N

$$X_j = e^{-\frac{log(2)}{H} \times j}$$

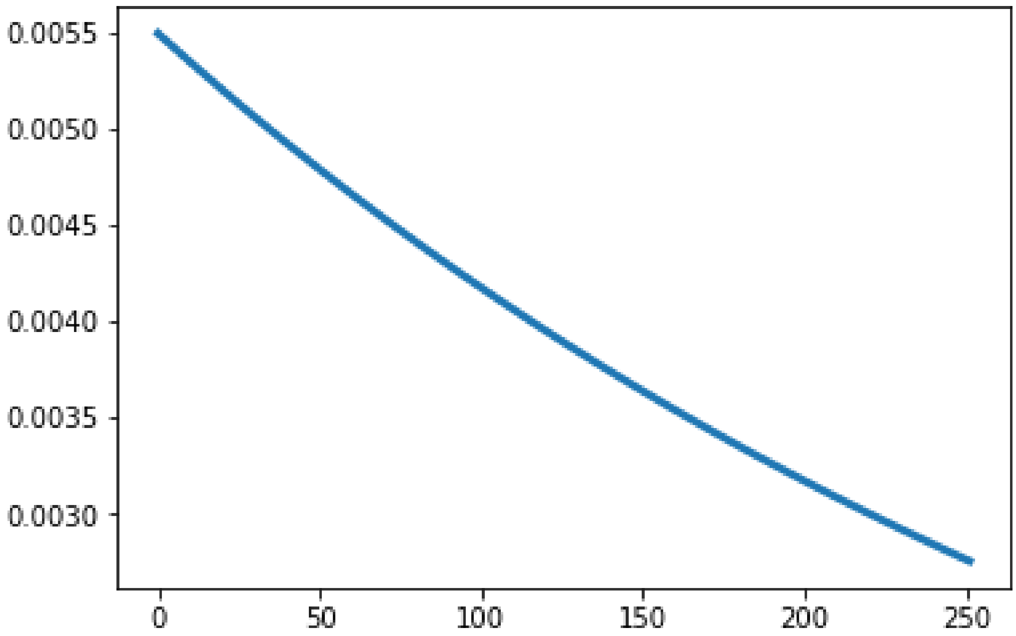
where H is half-life which determines the speed of decay, and log is natural log function Then a sequence of exponentially decaying weights w_j is defined as

$$w_j = \frac{X_j}{\sum\limits_{j=0}^{N} X_j}$$

In [13]:
```
# GRADED FUNCTION: exponent_weighting# GRADE
def exponent_weighting(n_periods, half_life = 252):
    """
    Calculate exponentially smoothed normalized (in probability density function sense)

    Arguments:
    n_periods -- number of periods, an integer, N in the formula above
    half_life -- half-life, which determines the speed of decay, h in the formula

    Return:
    exp_probs -- exponentially smoothed weights, np.array
    """

    exp_probs = np.zeros(n_periods) # do your own calculation instead of dummy zero arr
    ### START CODE HERE ### ( 3 lines of code)
    ### ...
    for j in range(n_periods):
        frac_temp = -1*(np.log(2)/half_life)*j
        xj = np.exp(frac_temp)
        exp_probs[j] = xj

    denominator = exp_probs.sum()
    exp_probs = exp_probs / denominator


    return exp_probs
    ### END CODE HERE ###
```

In [14]:
```
exp_probs = exponent_weighting(252*1)
plt.plot(exp_probs, linewidth=3)
```

Out[14]:
```
[<matplotlib.lines.Line2D at 0x7f8aa038cb70>]
```



In [15]:
```
### GRADED PART (DO NOT EDIT) ###
part_1=list(exp_probs[:100])
try:
    part1 = " ".join(map(repr, part_1))
except TypeError:
    part1 = repr(part_1)
submissions[all_parts[0]]=part1
grading.submit(COURSERA_EMAIL, COURSERA_TOKEN, assignment_key,all_parts[:1],all_parts,s
exp_probs[:100]
### GRADED PART (DO NOT EDIT) ###
```

```
Submission successful, please check on the coursera grader page for the status
```

Out[15]:
```
array([ 0.00549361,  0.00547852,  0.00546347,  0.00544846,  0.0054335 ,
        0.00541857,  0.00540369,  0.00538885,  0.00537404,  0.00535928,
        0.00534456,  0.00532988,  0.00531524,  0.00530064,  0.00528608,
        0.00527156,  0.00525708,  0.00524264,  0.00522824,  0.00521388,
        0.00519956,  0.00518528,  0.00517103,  0.00515683,  0.00514267,
        0.00512854,  0.00511445,  0.0051004 ,  0.00508639,  0.00507242,
        0.00505849,  0.00504459,  0.00503074,  0.00501692,  0.00500314,
        0.0049894 ,  0.00497569,  0.00496202,  0.00494839,  0.0049348 ,
        0.00492125,  0.00490773,  0.00489425,  0.00488081,  0.0048674 ,
        0.00485403,  0.0048407 ,  0.0048274 ,  0.00481414,  0.00480092,
        0.00478773,  0.00477458,  0.00476146,  0.00474838,  0.00473534,
        0.00472233,  0.00470936,  0.00469643,  0.00468353,  0.00467066,
```

```
                0.00465783,  0.00464504,  0.00463228,  0.00461956,  0.00460687,
                0.00459421,  0.00458159,  0.00456901,  0.00455646,  0.00454394,
                0.00453146,  0.00451901,  0.0045066 ,  0.00449422,  0.00448188,
                0.00446957,  0.00445729,  0.00444505,  0.00443284,  0.00442066,
                0.00440852,  0.00439641,  0.00438433,  0.00437229,  0.00436028,
                0.0043483 ,  0.00433636,  0.00432445,  0.00431257,  0.00430072,
                0.00428891,  0.00427713,  0.00426538,  0.00425367,  0.00424198,
                0.00423033,  0.00421871,  0.00420712,  0.00419557,  0.00418404])
```

```
In [16]: def absorption_ratio(explained_variance, n_components):
             """
             Calculate absorption ratio via PCA. absorption_ratio() is NOT to be used with Auto-

             Arguments:
             explained_variance -- 1D np.array of explained variance by each pricincipal compone

             n_components -- an integer, a number of principal components to compute absorption

             Return:
             ar -- absorption ratio
             """
             ar = np.sum(explained_variance[:n_components]) / np.sum(explained_variance)
             return ar
```

Part 2 (Implement Linear Auto-Encoder) LinearAutoEncoder class has two fully connected layers and no activation functions between the layers.

Instructions: - fill missing code within LinearAutoEncoder class - in init() method of LinearAutoEncoder setup neural network - **self.codings_layer** is a fully connected layer with **n_codings** neurons and no activation function - **self.outputs** is a fully connected layer with **n_outputs** neurons and no activation function - define loss function as Mean Square Error between the outputs and inputs referenced by **self.X** in the code - use AdamOptimizer to optimize model parameters

```
In [23]: def reset_graph(seed=42):
             tf.reset_default_graph()
             tf.set_random_seed(seed)
             np.random.seed(seed)

         class LinearAutoEncoder:
             """
             To perform simple PCA, we set activation_fn=None
             i.e., all neurons are linear and the cost function is the Mean-Square Error (MSE)
             """
             def __init__(self, n_inputs, n_codings, learning_rate=0.01):
                 self.learning_rate = learning_rate
                 n_outputs = n_inputs
                 self.destroy()
                 reset_graph()
```

```python
        # the inputs are n_inputs x n_inputs covariance matrices
        self.X = tf.placeholder(tf.float32, shape=[None, n_inputs, n_inputs])
        with tf.name_scope("lin_ae"):
            self.codings_layer = None
            self.outputs = None
            ### START CODE HERE ### ( 2 lines of code)
            self.codings_layer = tf.layers.dense(self.X, n_codings)
            self.outputs = tf.layers.dense(self.codings_layer, n_outputs)
            ### END CODE HERE ###

        with tf.name_scope("loss"):
            self.reconstruction_loss = None
            self.training_op = None
            ### START CODE HERE ### ( 4-5 lines of code)
            self.reconstruction_loss = tf.losses.mean_squared_error(self.X, self.output
            self.training_op = tf.train.AdamOptimizer(self.learning_rate).minimize(self
            ### END CODE HERE ###
            self.init = tf.global_variables_initializer()

    def destroy(self):
        if hasattr(self, 'sess') and self.sess is not None:
            self.sess.close()
            self.sess = None

    def absorption_ratio(self, test_input):
        """
        Calculate absorption ratio based on already trained model
        """
        if self.outputs is None:
            return test_input, 0.

        with self.sess.as_default():  # do not close session
            codings = self.codings_layer.eval(feed_dict={self.X: test_input})

            # calculate variance explained ratio
            result_ = self.outputs.eval(feed_dict={self.X: test_input})
            var_explained = np.sum(np.diag(result_.squeeze())) / np.sum(np.diag(test_in

        return codings[0, :, :], var_explained

    def next_batch(self, X_train, batch_size):
        """
        X_train - np.array of double of size K x N x N, where N is dimensionality of th
        batch_size - an integer, number of training examples to feed through the nwtwor
        """
        y_batch = None

        selected_idx = np.random.choice(tuple(range(X_train.shape[0])), size=batch_size
```

```
        X_batch = X_train[selected_idx, :, :]
        return X_batch, y_batch

    def train(self, X_train, X_test, n_epochs=5, batch_size=2, verbose=False):
        """
        train simple auto-encoder network
        :param X_train:
        :param X_test:
        :param n_epochs: number of epochs to use for training the model
        :param batch_size:
        :return:
        """
        if self.outputs is None:
            return X_test, 0.

        n_examples = len(X_train)  # number of training examples
        self.sess = tf.Session()

        # as_default context manager does not close the session when you exit the conte
        # and you must close the session explicitly.
        with self.sess.as_default():
            self.init.run()
            for epoch in range(n_epochs):
                n_batches = n_examples // min(n_examples, batch_size)
                for _ in range(n_batches):
                    X_batch, y_batch = self.next_batch(X_train, batch_size)
                    self.sess.run(self.training_op, feed_dict={self.X: X_batch})

                if verbose:
                    # last covariance matrix from the training sample
                    if X_train.shape[0] == 1:
                        mse_train = self.reconstruction_loss.eval(feed_dict={self.X: X_
                    else:
                        mse_train = self.reconstruction_loss.eval(feed_dict={self.X: np
                    mse_test = self.reconstruction_loss.eval(feed_dict={self.X: X_test}
                    print('Epoch %d. MSE Train %.4f, MSE Test %.4f' % (epoch, mse_train

            # calculate variance explained ratio
            test_input = np.array([X_train[-1, :, :]])
            result_ = self.outputs.eval(feed_dict={self.X: test_input})
            var_explained = np.sum(np.diag(result_.squeeze())) / np.sum(np.diag(test_in
            # print('Linear Auto-Encoder: variance explained: %.2f' % var_explained)

            codings = self.codings_layer.eval(feed_dict={self.X: X_test})
            # print('Done training linear auto-encoder')

        return codings[0, :, :], var_explained
```

In [24]:
```
### GRADED PART (DO NOT EDIT) ###
ix_offset = 1000
stock_tickers = asset_returns.columns.values[:-1]
assert 'SPX' not in stock_tickers, "By accident included SPX index"

step_size = 60
num_samples = 5
lookback_window = 252 * 2   # in (days)
num_assets = len(stock_tickers)
cov_matricies = np.zeros((num_samples, num_assets, num_assets)) # hold training data

# collect training and test data
ik = 0
for ix in range(ix_offset, min(ix_offset + num_samples * step_size, len(normed_r)), ste
    ret_frame = normed_r.iloc[ix_offset - lookback_window:ix_offset, :-1]
    print("time index and covariance matrix shape", ix, ret_frame.shape)
    cov_matricies[ik, :, :] = ret_frame.cov()
    ik += 1

# the last covariance matrix determines the absorption ratio
lin_ae = LinearAutoEncoder(n_inputs=num_assets, n_codings=200)
np.array([cov_matricies[-1, :, :]]).shape
lin_codings, test_absorp_ratio = lin_ae.train(cov_matricies[ : int((2/3)*num_samples),
                                                np.array([cov_matricies[-1, :, :]]),
                                                n_epochs=10,
                                                batch_size=5)
lin_codings, in_sample_absorp_ratio = lin_ae.absorption_ratio(np.array([cov_matricies[0
### GRADED PART (DO NOT EDIT) ###
```

```
time index and covariance matrix shape 1000 (504, 418)
time index and covariance matrix shape 1060 (504, 418)
time index and covariance matrix shape 1120 (504, 418)
time index and covariance matrix shape 1180 (504, 418)
time index and covariance matrix shape 1240 (504, 418)
```

In [25]:
```
### GRADED PART (DO NOT EDIT) ###
part_2=[test_absorp_ratio, in_sample_absorp_ratio]
try:
    part2 = " ".join(map(repr, part_2))
except TypeError:
    part2 = repr(part_2)
submissions[all_parts[1]]=part2
grading.submit(COURSERA_EMAIL, COURSERA_TOKEN, assignment_key,all_parts[:2],all_parts,s
[test_absorp_ratio, in_sample_absorp_ratio]
### GRADED PART (DO NOT EDIT) ###
```

```
Submission successful, please check on the coursera grader page for the status
```

```
Out[25]: [0.39359208195496026, 0.39359208195496026]
```

```
In [26]: stock_tickers = asset_returns.columns.values[:-1]
         assert 'SPX' not in stock_tickers, "By accident included SPX index"

         half_life = 252             # in (days)
         lookback_window = 252 * 2   # in (days)
         num_assets = len(stock_tickers)
         step_size = 1          # days : 5 - weekly, 21 - monthly, 63 - quarterly

         # require of that much variance to be explained. How many components are needed?
         var_threshold = 0.8

         # fix 20% of principal components for absorption ratio calculation. How much variance d
         absorb_comp = int((1 / 5) * num_assets)

         print('Half-life = %d' % half_life)
         print('Lookback window = %d' % lookback_window)
         print('Step size = %d' % step_size)
         print('Variance Threshold = %d' % var_threshold)
         print('Number of stocks = %d' % num_assets)
         print('Number of principal components = %d' % absorb_comp)
```

```
Half-life = 252
Lookback window = 504
Step size = 1
Variance Threshold = 0
Number of stocks = 418
Number of principal components = 83
```

```
In [28]: # indexes date on which to compute PCA
         days_offset = 4 * 252
         num_days = 6 * 252 + days_offset
         pca_ts_index = normed_r.index[list(range(lookback_window + days_offset, min(num_days, l

         # allocate arrays for storing absorption ratio
         pca_components = np.array([np.nan]*len(pca_ts_index))
         absorp_ratio = np.array([np.nan]*len(pca_ts_index))
         lae_ar = np.array([np.nan]*len(pca_ts_index))  # absorption ratio computed by Auto-Enco

         # keep track of covariance matricies as we would need them for training Auto-Encoder
         buf_size = 5
         cov_matricies = np.zeros((buf_size, num_assets, num_assets))

         exp_probs = exponent_weighting(lookback_window, half_life)
         assert 'SPX' not in normed_r.iloc[:lookback_window, :-1].columns.values, "By accident i
```

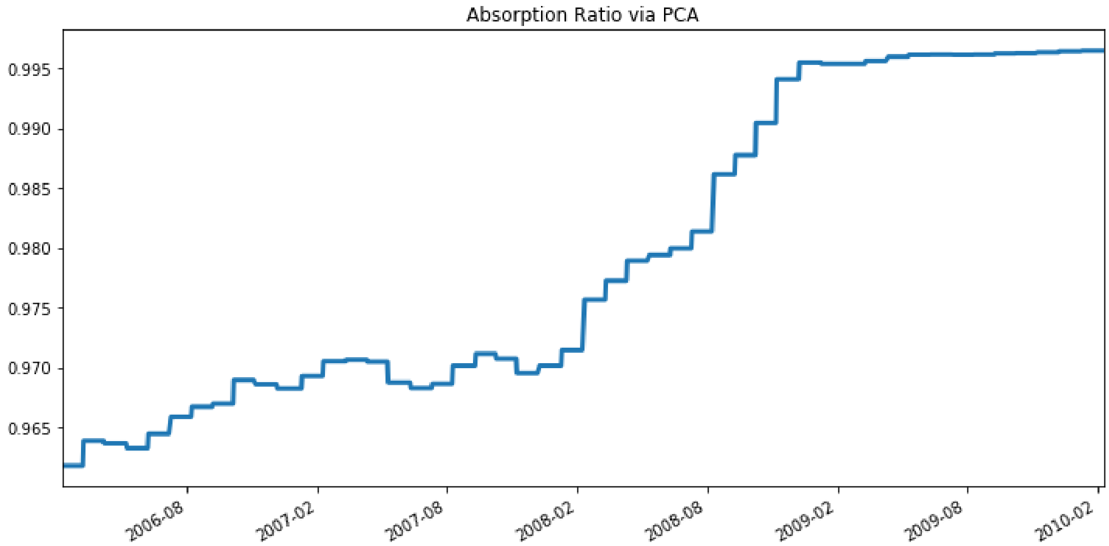
Instructions: - on each loop iteration: - fit PCA to **cov_mat** - use fitted pca model to pass values to absorption_ratio(). The result of absorption ratio calculation goes into **absorp_ratio** - compute number of principal components it takes to explain at least **var_threshold** of variance. The result of this calculation goes into **pca_components**

In [29]:
```
# run the main loop computing PCA and absorption at each step using moving window of re
# run this loop using both exponentially weighted returns and equally weighted returns
import time

ik = 0
use_ewm = False
lin_ae = None
time_start = time.time()
for ix in range(lookback_window + days_offset, min(num_days, len(normed_r)), step_size)
    ret_frame = normed_r.iloc[ix - lookback_window:ix, :-1]  # fixed window
    # ret_frame = normed_r.iloc[:ix, :-1]  # ever-growing window
    if use_ewm:
        ret_frame = (ret_frame.T * exp_probs).T

    cov_mat = ret_frame.cov()
    circ_idx = ik % buf_size
    cov_matricies[circ_idx, :, :] = cov_mat.values

    if ik == 0 or ik % 21 == 0:
        ### START CODE HERE ### ( 4-5 lines of code)
        ### fit PCA, compute absorption ratio by calling absorption_ratio()
        ### store result into pca_components for grading

        # https://www.coursera.org/learn/fundamentals-machine-learning-in-finance/discu
        # http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.ht
        pca=sklearn.decomposition.PCA().fit(cov_mat)
        absorp_ratio[ik] = absorption_ratio(pca.explained_variance_, absorb_comp)

        ### END CODE HERE ###
    else:
        absorp_ratio[ik] = absorp_ratio[ik-1]
        pca_components[ik] = pca_components[ik-1]

    if ik == 0 or ik % 252 == 0:
        if lin_ae is not None:
            lin_ae.destroy()

        print('Trainging AE', normed_r.index[ix])
        lin_ae = LinearAutoEncoder(cov_mat.shape[0], absorb_comp)
        lin_codings, lae_ar[ik] = lin_ae.train(cov_matricies[:circ_idx + 1, :, :],
                                               np.array([cov_mat.values]),
                                               batch_size=2)
    else:
        lin_codings, lae_ar[ik] = lin_ae.absorption_ratio(np.array([cov_mat.values]))
```

```
            ik += 1

        print ('Absorption Ratio done! Time elapsed: {} seconds'.format(time.time() - time_star
        ts_pca_components = pd.Series(pca_components, index=pca_ts_index)
        ts_absorb_ratio = pd.Series(absorp_ratio, index=pca_ts_index)
        ts_lae_absorb_ratio = pd.Series(lae_ar, index=pca_ts_index)
```

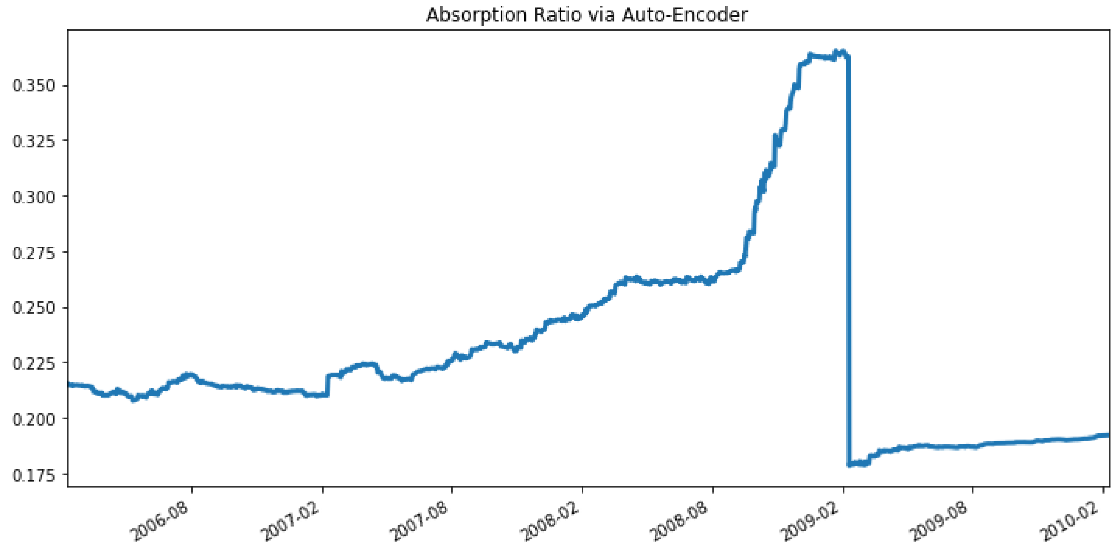
```
Trainging AE 2006-02-07 00:00:00
Trainging AE 2007-02-09 00:00:00
Trainging AE 2008-02-11 00:00:00
Trainging AE 2009-02-10 00:00:00
Absorption Ratio done! Time elapsed: 35.51445770263672 seconds
```

In [30]:
```
ts_absorb_ratio.plot(figsize=(12,6), title='Absorption Ratio via PCA', linewidth=3)
plt.savefig("Absorption_Ratio_SPX.png", dpi=900)
```



In [31]:
```
ts_lae_absorb_ratio.plot(figsize=(12,6), title='Absorption Ratio via Auto-Encoder', lin
```

Out[31]: <matplotlib.axes._subplots.AxesSubplot at 0x7f8a2d360550>



Having computed daily (this means the step size is 1) Absorption Ratio times series, we further follow M. Kritzman to make use of AR to define yet another measure: AR Delta. In particular:

$$AR\delta = \frac{AR_{15d} - AR_{1y}}{AR\sigma_{1y}}$$

We use $AR\delta$ to build simple portfolio trading strategy

```
In [32]: # following Kritzman and computing AR_delta = (15d_AR -1yr_AR) / sigma_AR
         ts_ar = ts_absorb_ratio
         ar_mean_1yr = ts_ar.rolling(252).mean()
         ar_mean_15d = ts_ar.rolling(15).mean()
         ar_sd_1yr = ts_ar.rolling(252).std()
         ar_delta = (ar_mean_15d - ar_mean_1yr) / ar_sd_1yr    # standardized shift in absorptio

         df_plot = pd.DataFrame({'AR_delta': ar_delta.values, 'AR_1yr': ar_mean_1yr.values, 'AR_
                                 index=ts_ar.index)
         df_plot = df_plot.dropna()
         if df_plot.shape[0] > 0:
             df_plot.plot(figsize=(12, 6), title='Absorption Ratio Delta', linewidth=3)
```