Learner Implementation

Tuesday, June 7, 2016 2:00 PM

Constructor parameters:

1. Lin Reg Learner ()
2. Polynomial Learner (degree)
3. KNN Learner (K)
4. Kernel Regression Learner (K, weighting func)
   (basically a Weighed KNN)
5. Decision Tree Learner (threshold value, max_depth, min_split/max_leaf_size

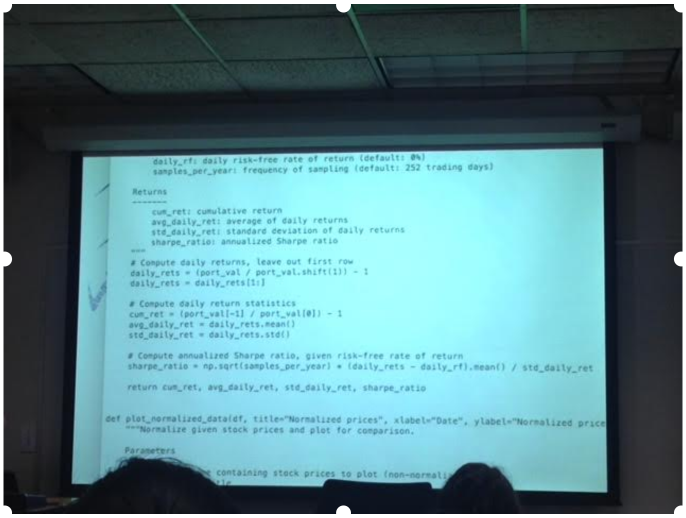
API:

```
learner = learner()
Learner.addEvidence(Xtrain, Ytrain)
y = learner.query(Xtest)
```

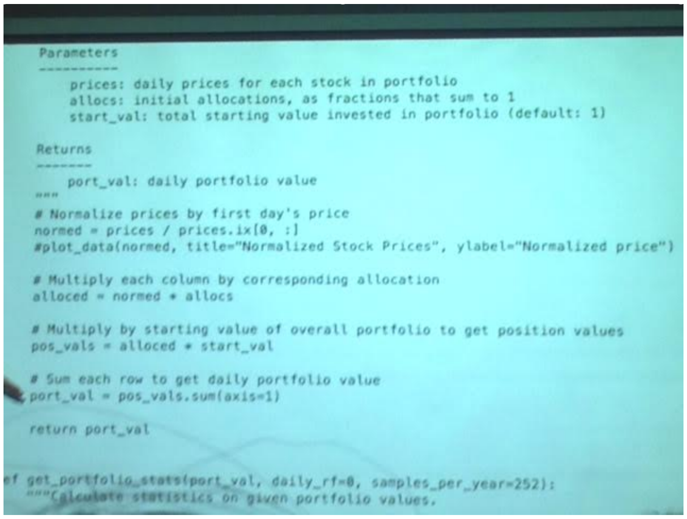
About Project 1

Pandas indexing change

Before 0.15, df[0:2] works as numpy index inside
df[0] gives the first row
After 0.15, df[0] works differently returns that index equals to 0. df.ix[0] gives the same results as older version



```
        daily_rf: daily risk-free rate of return (default: 0%)
        samples_per_year: frequency of sampling (default: 252 trading days)

    Returns
    -------
        cum_ret: cumulative return
        avg_daily_ret: average of daily returns
        std_daily_ret: standard deviation of daily returns
        sharpe_ratio: annualized Sharpe ratio
    """
    # Compute daily returns, leave out first row
    daily_rets = (port_val / port_val.shift(1)) - 1
    daily_rets = daily_rets[1:]

    # Compute daily return statistics
    cum_ret = (port_val[-1] / port_val[0]) - 1
    avg_daily_ret = daily_rets.mean()
    std_daily_ret = daily_rets.std()

    # Compute annualized Sharpe ratio, given risk-free rate of return
    sharpe_ratio = np.sqrt(samples_per_year) * (daily_rets - daily_rf).mean() / std_daily_ret

    return cum_ret, avg_daily_ret, std_daily_ret, sharpe_ratio


def plot_normalized_data(df, title="Normalized prices", xlabel="Date", ylabel="Normalized price
    """Normalize given stock prices and plot for comparison.

    Parameters
    ----------
        e containing stock prices to plot (non-normali
```



```
    Parameters
    ----------
        prices: daily prices for each stock in portfolio
        allocs: initial allocations, as fractions that sum to 1
        start_val: total starting value invested in portfolio (default: 1)

    Returns
    -------
        port_val: daily portfolio value
    """
    # Normalize prices by first day's price
    normed = prices / prices.ix[0, :]
    #plot_data(normed, title="Normalized Stock Prices", ylabel="Normalized price")

    # Multiply each column by corresponding allocation
    alloced = normed * allocs

    # Multiply by starting value of overall portfolio to get position values
    pos_vals = alloced * start_val

    # Sum each row to get daily portfolio value
    port_val = pos_vals.sum(axis=1)

    return port_val


ef get_portfolio_stats(port_val, daily_rf=0, samples_per_year=252):
    """Calculate statistics on given portfolio values.
```

1. **LinRegLearner.py**

```
class LinRegLearner::
    def __init__():
        pass
    def addEvidence(Xtrain, Ytrain):
        self.m, self.b = (yflrf).linreg(Xtrain, Ytrain)
    def query(Xtest):
        return (self.m * Xtest + self.b)
```

Linear
We already know the result should be in form of
y = mx + b

```
Loop until delta_J < threshold:
    sse = 0
    for x, y in zip(XTrain, YTrain):
        ypred = mx + b
        sse += (ypred - y)**2

    J = 1/2 * sse

    m_new = m - (ypred-y)*x*LearningRate
    b_new = b - (ypred - y)*LearningRate
```

J is the function I want to minimize here, as the 'loss function'

If we do the derivative, which is:
dJ/dm = (ypred - y)x
dJ/db = (ypred - y)

Without normalization, oscillation around optimum point is possible, therefore we need to adjust by learning rate.

We can set a threshold that if change of J is smaller than it we can quit the program.

2. **KNNLearner.py**

```
Class KNNLearner::
    def __init__(K=3):
        self.K = K
    def addEvidence(Xtrain, Ytrain):
        self.Xtrain = Xtrain
        self.Ytrain = Ytrain
    def query(Xtest):
        for i in range(0, self.Xtrain.shape[0]):
            dist[i] = metric(self.Xtrain[i], Xtest)
        nn = dist.argsort()

        for idx in nn:
            ysum += Ytrain[idx]

        return ysum/K
```

pt #	9	4	3	10	5
nn	2	1	4	0	3

In a classification problem, the only difference lays here. We need a Votes to give classification. E.g.

Votes[Ytrain[idx]] += 1

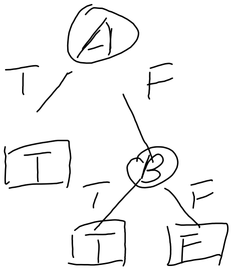
return (the most voted class)

3. **Decision Tree: CART**

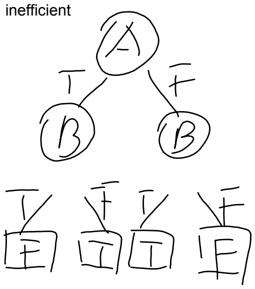
Bad: hard to learn "optimal decision tree"
1. Greedy
2. Depth/Size
3. NP-complete (cannot be proven to be computable in polynomial time)
4. Degenerate cases

Good:
1. Easy to understand
2. No preprocessing of data
3. Do both classification and regression
4. 'White box' model

'A or B' example



'A Xor B' case would be a problem, we have to create complete tree; if factors increases, we need 2^N nodes; becomes inefficient


A
T
F
B
B
T
F
T
F
F
T
T
F