Pegasos SVM implementation

- Siddharth Bhat (20161105)
- Link to video showing the code running on fashion-MNIST
- Link to repo (will be made public after deadline)

Scroll down to `how to run` to see running instructions.

Non-kernalized pegasos (linear model)

Training code

- Implement the straightforward formula given in `Figure 1` of the pegasos paper.
- Also contains debugging code to display the hyperplane if `debug=True` is passed. Helps debug code.

```
# d = dimension
# lam=lambda
# ts = training samples. List of (xi, yi)
# T = number of iterations
# Fig 1. Pegasos algorithm
def train_linear(d, lam, ts, T=None, debug=False):
    if T is None:
        T = len(ts) * 2

    w = np.zeros(d) # weight vector
    t = 1 # current iteration
    ixs = randint(0, len(ts), size=T + 1) # generate random indeces

    # loop for the samples
    while t <= T:
        # calculate eta
        eta = 1.0 / (float(lam) * float(t))
        # current sample (x, y)
        (x, y) = ts[ixs[t]]

        if y * np.dot(w, x) < 1:
            w = (1 - eta * lam) * w + eta * y * x
        else:
            w = (1 - eta * lam) * w

        # Debugging code to plot the hyperplanes
        if debug and t % (T//50) == 0:
            # code elided
```

```
            ...

        t += 1
    return w
```

Testing code

- Classify a point `x` given the weight vector `w`:

```
def classify_linear(w, x):
return 1 if np.dot(w, x) >= 1 else -1
```

```
def train_test_linreg_linear():
    print("LINEAR REGRESSION (with linear SVM): ")
    # generate these many training points
    NTRAIN = 100000
    ts = []
    for _ in range(NTRAIN):
        x1 = np.random.rand() * 10
        x2 = np.random.rand() * 10
        t = bool2y(x1 > 3 *  x2)
        ts.append((np.asarray([x1, x2]), t))

    # call the training
    w = train_linear(2, 0.01, ts, debug=False)

    # count total loss and print
    loss = 0
    ts = []
    NTEST = 1000
    for _ in range(NTEST):
        x1 = np.random.rand()
        x2 = np.random.rand()
        t = bool2y(x1 > 3 * x2)
        if classify_linear(w, np.asarray([x1, x2])) != t:
            loss += 1

    print("total loss: ", loss)
    print("avg loss: ", loss / NTEST)
```

Kernalized Pegasos

- We use the polynomial kernel, and we implement the kernalized pegasos algorithm presented in `Fig 3`.

```
# The polynomial kernel for a given power
```

```
# https://en.wikipedia.org/wiki/Polynomial_kernel
# d: dimension
# p: power to raise
def polynomialK(x1, x2, d, p):
    s = np.dot(x1, x2)
    return (s + 1) ** p


# Figure 3
# lam = lambda
# ts = training samples. List of (xi, yi)
# T = number of timesteps
# pow = polynomial to raise the kernel: (1 + x_i . x_j)^pow
def train_kernel_poly(lam, ts, T=None, pow=3):
    if T is None:
        T = len(ts)*3

    print ("training poly kernel. #samples: %d | lambda: %4.3f | T: %d | pow: %4.2f" %
                (len(ts), lam, T, pow))
    lam = float(lam) # lambda
    n = len(ts) # total number of training samples
    a = np.zeros(len(ts)) # alpha vector
    d = len(ts[0][0]) # find out dimension of the training vectors
    t = 1 # training count
    ixs = randint(0, n, size=(T+2)) # generate random indeces
    print(" ")
    while t <= T:
        i = ixs[t]
        # current training sample
        (xi, yi) = ts[i]

        # score
        s = 0
        j = 0
        while j < n:
            (xj, _) = ts[j]
            s += a[j] * yi * polynomialK(xi, xj, d, pow)
            j += 1

        s *=  yi * 1.0 / (lam * float(t))

        # if score < 1, bump up the a[i] index
        if s < 1:
            a[i] = a[i] + 1
        t += 1
```

```
        # debug print
        if t % (T // 100) == 0:
            print("\rtraining: %4.2f %%" %(float(t)/T * 100.0), end='')
    return a
```

Testing code: Cubic equation fitting

We generate random points `(x1, x2)` and classify based on whether `x1 >= x2^3`.

```
def train_test_cube_kernel():
    NTRAIN = 500
    print("CUBIC with kernel (#training: %d):" % NTRAIN)


    # generate training samples
    ts = []
    for _ in range(NTRAIN):
        x1 = np.random.rand() * 10
        x2 = np.random.rand() * 10
        t = bool2y(x1 >=  x2 * x2  * x2)
        ts.append((np.asarray([x1, x2]), t))

    # generate alpha vector
    a = train_kernel_poly(0.01, ts, T=len(ts)*10)


    # calculate losses
    loss = 0
    NTEST = 100
    print("\n\nrunning tests (total %d)" % NTEST)
    for i in range(NTEST):
        x1 = np.random.rand()
        x2 = np.random.rand()
        t = bool2y(x1 >=  x2 * x2 * x2)
        print("\rtesting: %4.2f %%" %(100.0 * i / NTEST), end='')
        if classify_kernel_poly(a, np.asarray([x1, x2]), ts) != t:
            loss += 1

    print(" ")
    print("total loss: ", loss)
    print("avg loss: ", loss / NTEST)
```

Testing code: Fashion-MNIST

The fashion-MNIST code is divided into three parts: loading the dataset, running

the training, and running the testing. The training and testing are separate so that we can batch the training.

To convert the training set to a SVM-friendly format, we need to change the class labels to `+1, -1` from the dataset which is multi-class.

```
# return list of tuples of (x, y) from fashion-MNIST dataset
def load_mnist(path, kind):
    # use mnist_reader from the fashion dataset codebase.
    (xs, ys) = mnist_reader.load_mnist(path, kind=kind)

    ts = []
    for i in range(len(xs)):
        # only keep samples that have y={0, 1}
        if ys[i] > 1: continue
        # transform y={0, 1} to y={1, -1}
        y = 1 if ys[i] == 0 else -1
        ts.append((xs[i], y))

    # return the cleaned up and SVM'd training set.
    return ts

# train the fashion kernel on the large dataset.
# N=number of training points.
def train_fashion_kernel(N):
    print("FASHION (first %s): " % N)
    ts = load_mnist(".",kind="train")
    ts = ts[:N]
    print("fashion dataset sample: ", ts[0])

    a = train_kernel_poly(0.01, ts, T=len(ts)*10)
    return a

def test_fashion_kernel(a):
    # take the first 'a' lenght sample from the traning set
    # since the vector 'a' describes their weights
    ts = load_mnist(".", kind="train")[:len(a)]

    tests = load_mnist(".", kind="t10k")
    tests = tests[:300]

    print("\n\n")
    print("#train : ", len(a))
    print("#test samples: ", len(tests))
    loss = 0
    i = 0
    N = len(tests)
    for (x,y) in tests:
```

```
        if classify_kernel_poly(a, x, ts) != y:
            loss += 1
        i += 1
        if i % (N // 100) == 0:
            print("\rtesting: %4.2f %%" % (100.0 * float(i)/N), end='')
    print("\n")
    print("total loss: ", loss)
    print("avg loss: ", loss / len(tests))
```

Performance metrics

The variables are:

- `d`: dimension of each training sample
- `N`: number of training samples
- `T`: number of training iterations
- Complexity of dot product: `d^2`.
- Complexity of the polynomial kernel: `O(dot product ~= d^2`.
- Complexity for non-kernelized: `O(#iterations) * O(dot product) ~= Td^2`.
- Complexity for kernelized: `O(#iterations) * O(#samples) * O(kernel) ~= TNd^2`.

How to run

To run the sample models, run:

- `./pegasos.py --demofashion`
- `./pegasos.py --demolinear`
- `./pegasos.py --democubic`

These trains and tests the model and prints the results. Useful to get a feel for how the model trains.

- `./pegasos.py --trainfashion`
- `./pegasos.py --testfashion`

The `--trainfashion` command trains the model for `(N=10000)` and dumps out a binary file which `--testfashion` picks up.