1003 HW5

Long Chen
lc3424@nyu.edu

April 6th, 2021

Q1

We observe $\Delta(y_i, y)$ and $\Phi(x_i, y) - \Psi(x_i, y_i)$ are invariant (constant) w.r.t w, and thus,

$$\Delta(y_i, y) + \langle w, \Phi(x_i, y) - \Psi(x_i, y_i) \rangle \tag{1}$$

is an affine transformation of w. Thus, (1) is convex w.r.t w for $\forall i$. We can then conclude that the point-wise maximum for all $y \in \mathcal{Y}$ is convex. That is:

$$\max_{y \in \mathcal{Y}} \left[\Delta(y_i, y) + \langle w, \Phi(x_i, y) - \Psi(x_i, y_i) \rangle \right]$$

is convex. Also that the norm of$w$, $\|w\|^2$ is convex. Thus we conclude that the non-negative combination of convex functions, $J(w)$, is a convex function.

Q2

Let $\hat{y}_i = \arg\max_{y \in \mathcal{Y}} \left[\Delta(y_i, y) + \langle w, \Phi(x_i, y) - \Psi(x_i, y_i) \rangle \right]$. Then we can express $J(w)$ as:

$$J(w) = \lambda \|w\|^2 + \frac{1}{n} \sum_{i=1}^{n} \left[\Delta(y_i, \hat{y}_i) + \langle w, \Phi(x_i, \hat{y}_i) - \Psi(x_i, y_i) \rangle \right].$$

Therefore, the subgradient of $J(w)$ is:

$$\partial J(w) = 2\lambda w + \frac{1}{n} \sum_{i=1}^{n} \left[\Phi(x_i, \hat{y}_i) - \Psi(x_i, y_i) \right].$$

For convenience in the future, we set $g = J(w)$

Q3

$$g_{\text{SGD}} = 2\lambda w + \Phi(x_i, \hat{y}_i) - \Psi(x_i, y_i)$$

Q4

$$g_{\text{MINI-BATCH}} = 2\lambda w + \frac{1}{m} \sum_{j=1}^{i+m-1} \left[\Phi(x_j, \hat{y}_j) - \Psi(x_j, y_j) \right]$$

*Optional Question

$$\begin{aligned}
\ell(h,(x,y)) &= \max\{[\Delta(y,y) + h(x,y) - h(x,y)], [\Delta(y,-y) + h(x,-y) - h(x,y)]\} \\
&= \max\{\Delta(y,y), [\Delta(y,-y) + h(x,-y) - h(x,y)]\} \\
&= \max 0, 1 + \begin{cases} -\frac{g(x)}{2} - \frac{g(x)}{2} & \text{for } y = 1 \\ \frac{g(x)}{2} + \frac{g(x)}{2} & \text{for } y = -1 \end{cases} \\
&= \max\{0, 1 - yg(x)\}
\end{aligned}$$

Q5

```python
from sklearn.base import BaseEstimator, ClassifierMixin, clone

class OneVsAllClassifier(BaseEstimator, ClassifierMixin):
    def __init__(self, estimator, n_classes):
        self.n_classes = n_classes
        self.estimators = [clone(estimator) for _ in range(n_classes)]
        self.fitted = False

    def fit(self, X, y=None):
        for i in range(self.n_classes):
            y_cur = (y == i).astype(int)
            self.estimators[i].fit(X, y_cur)

        self.fitted = True
        return self

    def decision_function(self, X):
        if not self.fitted:
            raise RuntimeError("You must train classifer before predicting data.")

        if not hasattr(self.estimators[0], "decision_function"):
            raise AttributeError(
                "Base estimator doesn't have a decision_function attribute.")

        res = np.zeros((X.shape[0], self.n_classes))
        for i in range(self.n_classes):
            res[:, i] = self.estimators[i].decision_function(X)

        return res

    def predict(self, X):
```

```
        return self.decision_function(X).argmax(axis=1)
```

Q6

Output:

```
Coeffs 0
[[-1.05854163 -0.90295959]]
Coeffs 1
[[-0.27439972  0.45755914]]
Coeffs 2
[[ 0.89164476 -0.82601248]]

array([[ 99,   1,   0],
       [  0, 100,   0],
       [  0,   0, 100]])
```

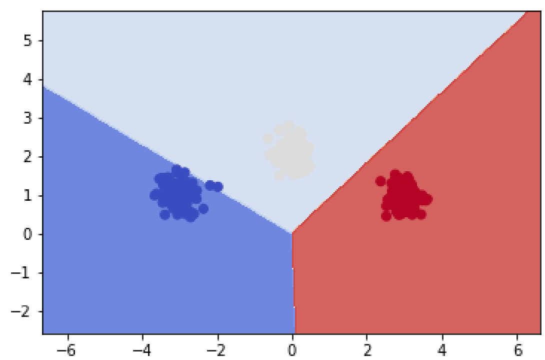

Figure 1: Q6 results.

Q7-Q9

```
def zeroOne(y,a) :
    return int(y != a)

def featureMap(X,y,num_classes) :
    num_samples, num_inFeatures = (1,X.shape[0]) if len(X.shape) == 1 else (X.shape[0],X.shape[1])
    n_outFeatures = num_inFeatures * num_classes

    # corner case: when we only have one datapoint
    if num_samples == 1:
        try:
            y = y[0]
```

```python
        except:
            y = y
        res = np.zeros(n_outFeatures)
        res[y * num_inFeatures : y * num_inFeatures + num_inFeatures] = X
        return res

    res = np.zeros((num_samples, n_outFeatures))

    for idx, xi in enumerate(X):
        temp = np.zeros(n_outFeatures)
        temp[y[idx] * num_inFeatures : y[idx] * num_inFeatures + num_inFeatures] = xi
        res[idx] = temp

    return res

def sgd(X, y, num_outFeatures, subgd, eta = 0.1, T = 10000):
    num_samples = X.shape[0]
    w = np.zeros(num_outFeatures)

    for cur_epoch in range(T):
        cur_idx = np.random.choice(num_samples, 1)
        # update
        w = w - eta * subgd(X[cur_idx], y[cur_idx], w)

    return w

class MulticlassSVM(BaseEstimator, ClassifierMixin):
    def __init__(self, num_outFeatures, lam=1.0, num_classes=3, Delta=zeroOne, Psi=featureMap):
        self.num_outFeatures = num_outFeatures
        self.lam = lam
        self.num_classes = num_classes
        self.Delta = Delta
        self.Psi = lambda X,y : Psi(X,y,num_classes)
        self.fitted = False

    def subgradient(self,x,y,w):
        res = []

        # compute class weights
        for y_prime in range(self.num_classes):
            res.append(self.Delta(y, y_prime) + np.dot(w, self.Psi(x, y_prime) - self.Psi(x, y)))

        # get argmax
        y_hat = np.argmax(res)

        return 2 * self.lam * w + self.Psi(x, y_hat) - self.Psi(x, y)

    def fit(self,X,y,eta=0.1,T=10000):
        self.coef_ = sgd(X,y,self.num_outFeatures,self.subgradient,eta,T)
        self.fitted = True
        return self
```

```
    def decision_function(self, X):
        if not self.fitted:
            raise RuntimeError("You must train classifer before predicting data.")

        res = np.zeros((X.shape[0], self.num_classes))

        # calculate scores for each classes
        for idx, xi in enumerate(X):
            res[idx, :] = [np.dot(self.coef_, self.Psi(xi, yi)) for yi in range(self.num_classes)]

        return res

    def predict(self, X):
    return self.decision_function(X).argmax(axis=1)
```

Q10

Note: using eta=0.01. For eta=0.1, sometimes the algorithm does not converge.

```
w:
[-0.2557413  -0.06645359  0.16196182  0.32185955  0.09377949
    -0.25540596]

array([[100,   0,   0],
       [  0, 100,   0],
       [  0,   0, 100]])
```

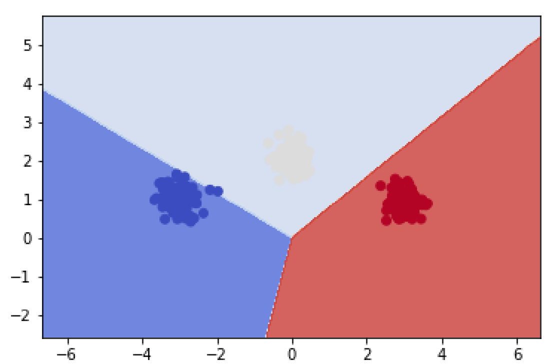

Figure 2: Q10 results.