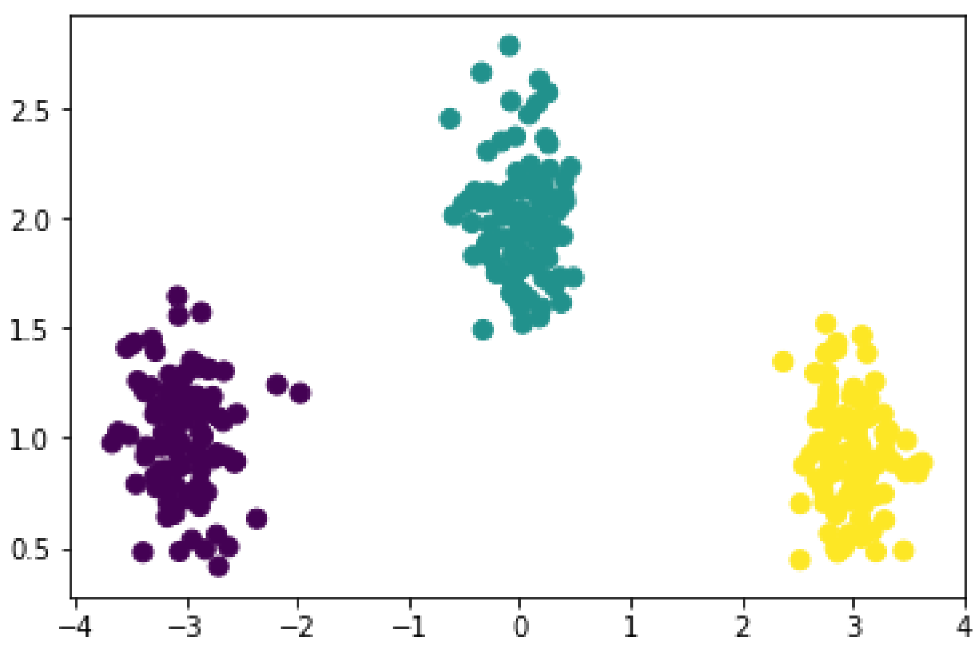
multiclass-skeleton-code

April 29, 2019

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        from sklearn.datasets.samples_generator import make_blobs
        import pdb
        from collections.abc import Iterable
        from tqdm import tqdm

        %matplotlib inline
```

```
In [2]: # Create the  training data
        np.random.seed(2)
        X, y = make_blobs(n_samples=300,cluster_std=.25, centers=np.array([(-3,1),(0,2),(3,1)]))
        plt.scatter(X[:, 0], X[:, 1], c=y, s=50)
```

```
Out[2]: <matplotlib.collections.PathCollection at 0x7fa982119470>
```



```python
In [3]: from sklearn.base import BaseEstimator, ClassifierMixin, clone

        class OneVsAllClassifier(BaseEstimator, ClassifierMixin):
            """
            One-vs-all classifier
            We assume that the classes will be the integers 0,..,(n_classes-1).
            We assume that the estimator provided to the class, after fitting, has a "decision_f
            returns the score for the positive class.
            """
            def __init__(self, estimator, n_classes):
                """
                Constructed with the number of classes and an estimator (e.g. an
                SVM estimator from sklearn)
                @param estimator : binary base classifier used
                @param n_classes : number of classes
                """
                self.n_classes = n_classes
                self.estimators = [clone(estimator) for _ in range(n_classes)]
                self.fitted = False

            def fit(self, X, y=None):
                """
                This should fit one classifier for each class.
                self.estimators[i] should be fit on class i vs rest
                @param X: array-like, shape = [n_samples,n_features], input data
                @param y: array-like, shape = [n_samples,] class labels
                @return returns self
                """
                for class_n, estimator in enumerate(self.estimators):
                    labels = (y == class_n).astype(int)
                    estimator.fit(X, labels)
                self.fitted = True
                return self

            def decision_function(self, X):
                """
                Returns the score of each input for each class. Assumes
                that the given estimator also implements the decision_function method (which skl
                and that fit has been called.
                @param X : array-like, shape = [n_samples, n_features] input data
                @return array-like, shape = [n_samples, n_classes]
                """
                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.")
```

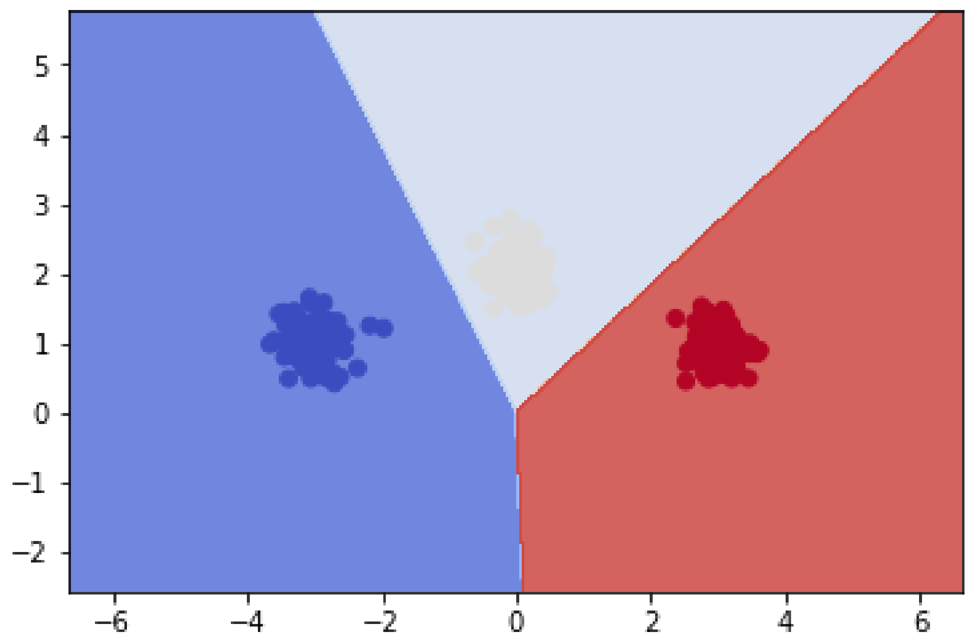
```
            return np.array([e.decision_function(X) for e in self.estimators]).T

        def predict(self, X):
            """
            Predict the class with the highest score.
            @param X: array-like, shape = [n_samples,n_features] input data
            @returns array-like, shape = [n_samples,] the predicted classes for each input
            """
            return np.argmax(self.decision_function(X), axis=1)
```

```
In [4]: #Here we test the OneVsAllClassifier
        from sklearn import svm
        svm_estimator = svm.LinearSVC(loss='hinge', fit_intercept=False, C=200)
        clf_onevsall = OneVsAllClassifier(svm_estimator, n_classes=3)
        clf_onevsall.fit(X,y)

        for i in range(3) :
            print("Coeffs %d"%i)
            print(clf_onevsall.estimators[i].coef_) #Will fail if you haven't implemented fit ye

        # create a mesh to plot in
        h = .02  # step size in the mesh
        x_min, x_max = min(X[:,0])-3,max(X[:,0])+3
        y_min, y_max = min(X[:,1])-3,max(X[:,1])+3
        xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                             np.arange(y_min, y_max, h))
        mesh_input = np.c_[xx.ravel(), yy.ravel()]

        Z = clf_onevsall.predict(mesh_input)
        Z = Z.reshape(xx.shape)
        plt.contourf(xx, yy, Z, cmap=plt.cm.coolwarm, alpha=0.8)
        # Plot also the training points
        plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.coolwarm)


        from sklearn import metrics
        metrics.confusion_matrix(y, clf_onevsall.predict(X))
```

```
Coeffs 0
[[-1.05852964 -0.90293998]]
Coeffs 1
[[ 0.31047003 -0.19010007]]
Coeffs 2
[[ 0.8916347  -0.82599479]]
```

```
/home/cfizette/anaconda3/envs/ml/lib/python3.6/site-packages/sklearn/svm/base.py:931: Convergenc
  "the number of iterations.", ConvergenceWarning)
```

```
Out[4]: array([[100,   0,   0],
               [  0, 100,   0],
               [  0,   0, 100]])
```



Multiclass SVM

```
In [16]: def zeroOne(y,a) :
             '''
             Computes the zero-one loss.
             @param y: output class
             @param a: predicted class
             @return 1 if different, 0 if same
             '''
             return int(y != a)

         def featureMap(X,y,num_classes) :
             '''
             Computes the class-sensitive features.
             @param X: array-like, shape = [n_samples,n_inFeatures] or [n_inFeatures,], input fe
             @param y: a target class (in range 0,..,num_classes-1)
             @return array-like, shape = [n_samples,n_outFeatures], the class sensitive features
             '''
             #The following line handles X being a 1d-array or a 2d-array
             num_samples, num_inFeatures = (1,X.shape[0]) if len(X.shape) == 1 else (X.shape[0],
             if num_samples == 1:
                 X = [X]

             # n_out_features = n_classes * n_in_features
             # feature_map[y*n_in_features : y*n_in_features + n_in_features] = X

             feature_map = np.zeros((num_samples, num_inFeatures * num_classes))

             for i, xi in enumerate(X):
                 start = y*num_inFeatures
                 end = start + num_inFeatures
                 feature_map[i,start:end] = xi

             '''
             for i in range(num_samples):
                 xi = X[i]
                 yi = y
                 start = yi*num_inFeatures
                 end = start + num_inFeatures
                 feature_map[i,start:end] = xi
             '''
             return feature_map

         def sgd(X, y, num_outFeatures, subgd, eta = 0.1, T = 10000):
             '''
             Runs subgradient descent, and outputs resulting parameter vector.
             @param X: array-like, shape = [n_samples,n_features], input training data
             @param y: array-like, shape = [n_samples,], class labels
```

```
    @param num_outFeatures: number of class-sensitive features
    @param subgd: function taking x,y and giving subgradient of objective
    @param eta: learning rate for SGD
    @param T: maximum number of iterations
    @return: vector of weights
    '''
    num_samples = X.shape[0]
    w = np.zeros(num_outFeatures)
    for _ in tqdm(range(T)):
        for xi, yi in zip(X, y):
            w -= eta * subgd(xi, yi, w)
    return w

class MulticlassSVM(BaseEstimator, ClassifierMixin):
    '''
    Implements a Multiclass SVM estimator.
    '''
    def __init__(self, num_outFeatures, lam=1.0, num_classes=3, Delta=zeroOne, Psi=feat
        '''
        Creates a MulticlassSVM estimator.
        @param num_outFeatures: number of class-sensitive features produced by Psi
        @param lam: l2 regularization parameter
        @param num_classes: number of classes (assumed numbered 0,..,num_classes-1)
        @param Delta: class-sensitive loss function taking two arguments (i.e., target
        @param Psi: class-sensitive feature map taking two arguments
        '''
        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 generalized_hinge_loss(self, x, y, w):
        return max([self.class_hinge_loss(x,y,w,y_pred) for y_pred in range(self.n_clas

    def class_hinge_loss(self, x, y, w, y_pred):
        return self.Delta(y, y_pred) + (self.Psi(x, y_pred) - self.Psi(x, y))@w

    def subgradient(self,x,y,w):
        '''
        Computes the subgradient at a given data point x,y
        @param x: sample input
        @param y: sample class
        @param w: parameter vector
        @return returns subgradient vector at given x,y,w
        '''
        y_hat = np.argmax([self.class_hinge_loss(x,y,w,y_pred) for y_pred in range(self
```

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

        def fit(self,X,y,eta=0.1,T=10000):
            '''
            Fits multiclass SVM
            @param X: array-like, shape = [num_samples,num_inFeatures], input data
            @param y: array-like, shape = [num_samples,], input classes
            @param eta: learning rate for SGD
            @param T: maximum number of iterations
            @return returns self
            '''
            self.coef_ = sgd(X,y,self.num_outFeatures,self.subgradient,eta,T)
            self.fitted = True
            return self

        def decision_function(self, X):
            '''
            Returns the score on each input for each class. Assumes
            that fit has been called.
            @param X : array-like, shape = [n_samples, n_inFeatures]
            @return array-like, shape = [n_samples, n_classes] giving scores for each sampl
            '''
            if not self.fitted:
                raise RuntimeError("You must train classifer before predicting data.")

            return np.array([self.Psi(X, y)@self.coef_ for y in range(self.num_classes)]).T

        def predict(self, X):
            '''
            Predict the class with the highest score.
            @param X: array-like, shape = [n_samples, n_inFeatures], input data to predict
            @return array-like, shape = [n_samples,], class labels predicted for each data
            '''
            raw_pred = self.decision_function(X)
            pred = np.argmax(raw_pred, axis=1)
            return pred
```

In [17]:
```
#the following code tests the MulticlassSVM and sgd
#will fail if MulticlassSVM is not implemented yet
est = MulticlassSVM(6,lam=1)
est.fit(X,y, T=500, eta=0.01)
print("w:")
print(est.coef_)
Z = est.predict(mesh_input)
Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, cmap=plt.cm.coolwarm, alpha=0.8)
# Plot also the training points
```

```
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.coolwarm)


from sklearn import metrics
metrics.confusion_matrix(y, est.predict(X))
```

```
100%|| 500/500 [00:10<00:00, 46.90it/s]


w:
[-0.34533686 -0.04675874  0.02345881  0.06871364  0.32187805 -0.02195491]
```

```
Out[17]: array([[100,   0,   0],
                [  0, 100,   0],
                [  0,   0, 100]])
```