Perceptron Learning Algorithm

August 21, 2019

0.1 # Perceptron Learning Algorithm

David Gillis
CS4340 - Machine Learning
Due: September 5, 2019

0.2 Assignment Description

Implement any variant of the perceptron learning algorithm for binary classification.

Generate 80 data points total, 40 from each class. Use 50 for training, and 30 for testing. Consider two cases, one where the data is linearly seperable, and one where the data is not. Submit a single pdf file with the training data, source code, and results.

In addition, write responses to the following:

1. how (that is, following what logic)you generated the data points (training and test),
2. whether the training data points are linearly separable,
3. whether the test points are linearly separable,
4. your initial choice of the weights and constants,
5. the final solution equation of the line (decision boundary),
6. the total number of weight vector updates that your algorithm made,
7. the total number of iterations made over the training set(an iteration involves checking each of the training points and determining its class as given by the line), and
8. the final misclassification error, if any(expressed as a percentage), on the training data as well as on the test data.

Also, re-run your code (for both the above cases: linearly separable training set and not linearly separable training set) by varying the following and describe the effect, if any, that each of the three variations had on the final solution equation (test each of the three variations separately and summarize your resultsin a numbered list):

1. the initial choice of the weights,
2. the initial choice of the step size constant (c),
3. the order in which you consider the points in the training set.

```
In [1]: import numpy as np
        import warnings
        import matplotlib.pyplot as plt
```

0.3 Preliminaries

We need a way to perform multiple initializations. I can think of two relatively good ways to write initializers - the way I have done it here, and closures. I'd likely agree to the argument that closures are easier to read. I digress . . .

We create an `Initializer` base class which takes in `**kwargs` in the `__init__` method. The keyword args are then saved in a class dictionary. Finally, when we `__call__` the initializer, we pass off the arguments we passed in. This allows us to essentially create a function the same way we would in a closure.

Now, we can subclass the `Initializer`, and just provide any default arguments to `param_dict`, as well as, a function for initialization. Might be worth researching how this is done in Keras. In general, relying on subclassing is not maintainable.

Update: I just read the Keras source, and I was super close. I'm going to keep the old code in here because documenting my learning is beneficial. If this were production code, I'd remove it. I like the way Keras does this - and it follows my thinking about closures - to use closures, you have to specify the arguments in the definition of the closure, which is substantially more clear to the end user.

For Reference: Keras Initializer

```
In [2]: '''
        class Initializer(object):
            def __init__(self, **kwargs):
                self.param_dict = kwargs
                self._init_fn = None

            def __call__(self, *args, **kwargs):
                return self._init_fn(**self.param_dict, **kwargs)


        class StandardNormalInitializer(Initializer):
            def __init__(self, **kwargs):
                super(StandardNormalInitializer, self).__init__(**kwargs)
                self.param_dict['loc'] = 0.0
                self.param_dict['scale'] = 1.0
                self._init_fn = np.random.normal


        class RandomNormalInitializer(Initializer):
            def __init__(self, **kwargs):
                super(RandomNormalInitializer, self).__init__(**kwargs)
                self._init_fn = np.random.normal


        class ZeroInitializer(Initializer):
            def __init__(self, **kwargs):
                super(ZeroInitializer, self).__init__(**kwargs)
                self._init_fn = np.zeros
        '''
```

```
Out[2]: "\nclass Initializer(object):\n    def __init__(self, **kwargs):\n        self.param_d
```

```
In [3]: class Initializer(object):
            def __call__(self, shape):
                raise NotImplementedError()


        class ZeroInitializer(Initializer):
            def __call__(self, shape):
                return np.zeros(shape)


        class RandomNormalInitializer(Initializer):
            def __init__(self, mean, std):
                self.mean = mean
                self.std = std

            def __call__(self, shape):
                return np.random.normal(size=shape, loc=self.mean, scale=self.std)


        class StandardNormalInitializer(Initializer):
            def __call__(self, shape):
                return np.random.normal(size=shape, loc=0.0, scale=1.0)


        class ConstantInitializer(Initializer):
            def __init__(self, value):
                self.value = value

            def __call__(self, shape):
                return self.value * np.ones(shape=shape)
```

We also need a few utility functions. The first is to shuffle two arrays in unison - we need a way to be able to shuffle our predictors and targets together.

Next, we need the sign function. This will be used in the prediction loop. Ideally, there is a way to have this not applied element wise . . . I don't know what that is, I'll look it up later.

```
In [4]: # TODO
        def shuffle_arrays(a, b):
            """Shuffle two arrays with indicies."""
            if len(a) != len(b):
                raise ValueError('len(a) != len(b)')
            indices = np.random.permutation(len(a))
            return a[indices], b[indices]


        def sign(s):
```

```
        if s > 0:
            return 1
        else:
            return -1
```

For the perceptron itself, I'll explain the code in comments. I think it will be easier to follow that way. With that said, the algorithm is as follows:

1. Initialize the weights based on `initializer`. Default to zeros.
2. Until convergence, or maximum iterations:

   - Calculate the output $o := sign(w^T x)$
   - If the sample is misclassified, update the weights according to $w_{t+1} = w_t + \eta y_t x_t$

As of right now, I've only implemented the PLA version without a pocket. I may implement the pocket variant later, based on time.

Worth noting, matrix notation would also be ideal here. TODO

```
In [5]: class Perceptron(object):
            def __init__(self,
                         learning_rate=1.0,
                         maximum_iterations=1000,
                         tolerance=0.001,
                         shuffle_on_epoch=True,
                         initializer=None,
                         **kwargs):
                """
                Initialize a perceptron learner.

                learning_rate : float
                    Rate to multiply the update by. Default 1.0.
                maximum_iterations : int
                    Maximum number of iterations before stopping.
                tolerance : float
                    Error to reach before stopping.
                shuffle_on_epoch : bool
                    Whether to shuffle the training data on each pass through. Default True.
                initializer : str
                    How to initialize the weights. If not 'standard', arguments
                    for the respective initialization method must be supplied.

                Notes:
                    This is not a performant learner, for a multitude of reasons,
                    most notably that the `fit` method will make a copy of the
                    training data. While this is acceptable for this example, for
                    large data sets, you'll start to hose your RAM. Only use on
                    toy datasets.

                    It is also assumed that your targets are in the set {-1, +1}.
                    This is a large assumption, and it is one which would break
                    the algorithm as stated in other texts. However, we choose
                    to use the algorithm in the course text, which assumes the data
                    is in this form. DO NOT feed data in any other form. If you do,
                    the model IS NOT GUARANTEED to converge to a good solution, even
                    for linearly seperable data.
                """
                if initializer is None:
                    self._initializer = ZeroInitializer()
                else:
                    self._initializer = initializer
                self._learning_rate = learning_rate
                self._maximum_iterations = maximum_iterations
                self._tolerance = tolerance
                self._shuffle_on_epoch = shuffle_on_epoch
                self._weights = None
                self._validate_input()

            def fit(self, predictors, targets):
                """
                Fit the perceptron learner.
                predictors : numpy.ndarray (or array like, but don't blame me if it doesn't wo
                    Predictors to train on.
                """
                if len(predictors.shape) != 2:
                    raise ValueError(f'Invalid shape predictors.shape == {predictors.shape}')
                if len(targets.shape) != 1:
                    raise ValueError(f'Invalid shape targets.shape == {targets.shape}')

                # We only need to copy the dataset if we shuffle each epoch
                if self._shuffle_on_epoch:
                    targets = np.copy(targets)

                # Copying the predictors is handled in the call. Not sure I like
                # that . . . TODO
                predictors = Perceptron._pad_first_column_ones(predictors)

                # Recall that we have a weight for the bias. This is already included
                # in the shape, because we padded the ones at the front.
                cols = predictors.shape[1]
                self._weights = self._initializer(shape=(cols))

                lr = self._learning_rate
                # Iterate until we reach the maximum iteration, or until the error
                # is low enough to break.
                iterator, error = 0, np.inf
                while iterator < self._maximum_iterations and error > self._tolerance:
                    # Iterate over every sample. Randomization is handled by shuffling.
```

```python
            for predictor, target in zip(predictors, targets):
                # Compute the output
                prediction = sign(np.dot(self._weights, predictor))
                # If misclassed, update according to the update rule.
                if prediction != target:
                    update = lr * target * predictor
                    self._weights = self._weights + update
                iterator += 1

            # Compute the error for termination.
            predictions = self.predict(predictors, has_weights=True)
            error = self.evaluate(predictions, targets)
            # Shuffle as needed.
            if self._shuffle_on_epoch:
                predictors, targets = shuffle_arrays(predictors, targets)

        # Warn the caller if we didn't converge.
        if error > self._tolerance:
            warnings.warn('error > tolerance. Consider updating'
                          'maximum_iterations and running again.')

    def predict(self, predictors, has_weights=False):
        if not has_weights:
            predictors = Perceptron._pad_first_column_ones(predictors)
        predictions = []
        for predictor in predictors:
            predictions.append(sign(np.dot(self._weights, predictor)))
        return np.array(predictions)

    def evaluate(self, predicted, actual):
        # Again, likely a better way to do this.
        if len(predicted) != len(actual):
            raise ValueError('len(predicted) != len(actual)')
        correct = sum(predicted == actual) * 1.0
        return 1.0 - (correct / len(actual))

    @staticmethod
    def _pad_first_column_ones(predictors):
        rows, cols = predictors.shape
        predictors = np.concatenate([np.ones((rows, 1)), predictors], axis=1)
        return predictors

    def _validate_input(self):
        if self._learning_rate <= 0:
            raise ValueError('learning_rate must be > 0')
        if self._maximum_iterations <= 0:
            raise ValueError('maximum_iterations must be > 0')
        if self._tolerance <= 0:
```

```
                    raise ValueError('tolerance must be > 0')
```

```
In [6]: i = ConstantInitializer(value=20)
        i(shape=10)
```

```
Out[6]: array([20., 20., 20., 20., 20., 20., 20., 20., 20., 20.])
```

```
In [7]: p = Perceptron(initializer=StandardNormalInitializer())
        pos = np.random.sample((40, 2)) + 2
        neg = np.random.sample((40, 2))
        pos_targets = np.ones(40)
        neg_targets = np.zeros(40) - 1

        x = np.concatenate([pos, neg], axis=0)
        y = np.concatenate([pos_targets, neg_targets], axis=0)

        x, y = shuffle_arrays(x, y)

        print(x)
        print(y)
```

```
[[2.20889971 2.035565  ]
 [0.85721175 0.75619912]
 [2.88019393 2.91583929]
 [2.87949621 2.42915899]
 [2.93469967 2.04809626]
 [2.86990036 2.89558995]
 [0.90829449 0.04290223]
 [2.88218545 2.75819801]
 [2.5490934  2.1639312 ]
 [0.50793966 0.84321601]
 [0.77874422 0.64260657]
 [2.04356839 2.63953466]
 [0.0134899  0.24274582]
 [0.64520896 0.69545464]
 [0.48496653 0.80450537]
 [0.94932559 0.9538364 ]
 [0.79485416 0.27526242]
 [2.05350896 2.47109983]
 [2.29851623 2.05602852]
 [0.89509984 0.95818801]
 [2.05982688 2.18630969]
 [0.09767327 0.9559069 ]
 [0.15991682 0.57254497]
 [2.85442424 2.97783095]
 [2.50060374 2.01260035]
 [0.39336685 0.52801539]
 [0.4850978  0.50303004]
```

```
[0.03672223 0.26365145]
[0.6192916  0.18029216]
[2.59736358 2.85649832]
[0.33446367 0.49231055]
[2.34931531 2.8473107 ]
[2.17385141 2.06736888]
[2.07863404 2.38155956]
[2.73045664 2.13196118]
[2.58823517 2.32327126]
[2.71129256 2.54901938]
[2.35632422 2.03952326]
[0.73201469 0.26924852]
[2.04685575 2.78976374]
[0.99766596 0.46492411]
[0.07814515 0.25236089]
[2.48565025 2.26967513]
[0.22276367 0.41001418]
[2.40761792 2.38660789]
[0.53384568 0.5695088 ]
[2.62434315 2.95701676]
[0.46720614 0.06564927]
[2.79847331 2.71566784]
[0.88868163 0.78194496]
[0.06215215 0.83406317]
[2.90820949 2.41478254]
[0.30843155 0.21219654]
[0.10880384 0.30642112]
[0.64008654 0.20631188]
[0.78376865 0.17590933]
[2.33352008 2.48813503]
[0.45423856 0.48839688]
[2.54509156 2.19458213]
[2.28828937 2.40856654]
[2.35081362 2.68051275]
[2.04566489 2.58464027]
[0.06745626 0.43232474]
[0.66257292 0.67306871]
[2.97766283 2.64480276]
[2.0133938  2.91466711]
[0.91595301 0.46999975]
[0.07056588 0.08643395]
[2.59818102 2.71452405]
[0.59529781 0.89751177]
[2.3825545  2.45990771]
[0.01052646 0.51201515]
[0.51453311 0.19377042]
[0.68896558 0.4193379 ]
[2.67357594 2.8975072 ]
```