Penn Engineering ONLINE LEARNING

CIS 521-Artificial Intelligence

Homework 9: Perceptrons [105 points]

Instructions

In this assignment, you will gain experience working with binary and multiclass perceptrons.

A skeleton file homework9.py containing empty definitions for each question has been provided. Since portions of this assignment will be graded automatically, none of the names or function signatures in this file should be modified. However, you are free to introduce additional variables or functions if needed.

You may import definitions from any standard Python library, and are encouraged to do so in case you find yourself reinventing the wheel.

You will find that in addition to a problem specification, most programming questions also include a pair of examples from the Python interpreter. These are meant to illustrate typical use cases, and should not be taken as comprehensive test suites.

Once you have completed the assignment, you should submit your file on Gradescope. You may submit as many times as you would like before the deadline, but only the last submission will be saved.

0. Style [5 points]

Your code should follow the proper Python style guidelines set forth in PEP 8, which was written in part by the creator of Python. Our autograders will automatically scan your submission for style errors using the `pycodestyle` library on default settings. If your submission contains **any** style errors, the autograder will show you some of them and you will not receive these 5 points. You can use `pycodestyle` or any other tool you like to make sure that your submission conforms to PEP 8 guidelines.

1. Perceptrons [35 points]

In this section, you will implement two varieties of the standard perceptron: one which performs binary classification, distinguishing between positive and negative instances, and one which performs multiclass classification, distinguishing between an arbitrary number of labeled groups.

As in previous assignments, your use of external code should be limited to built-in Python modules, which excludes packages such as NumPy and NLTK.

1. **[15 points]** A binary perceptron is one of the simplest examples of a linear classifier. Given a set of data points each associated with a positive or negative label, the goal is to learn a vector $\vec{w}$ such that $\vec{w} \cdot \vec{x}_+ > 0$ for positive instances $\vec{x}_+$ and $\vec{w} \cdot \vec{x}_- \leq 0$ for negative instances $\vec{x}_-$.

One learning algorithm for this problem initializes the weight vector $\vec{w}$ to the zero vector, then loops through the training data for a fixed number of iterations, adjusting the weight vector whenever a sample is misclassified.

> **Input:**
>
> A list T of training examples $(\vec{x_1}, y_1), \cdots, (\vec{x_n}, y_n)$, where $y_i \in \{+, -\}$; the number of passes N to make over the data set.
>
> **Output:**
>
> The weight vector $\vec{w}$.
>
> **Procedure:**
>
> 1. Initialize the weight vector as $\vec{w} \leftarrow 0$.
> 2. **for** iteration $= 1$ **to** N **do**
> 3. **for** each example $(\vec{x_i}, y_i) \in T$ **do**
> 4. Compute the predicted class as $\hat{y_i} = \text{sign}(\vec{w} \cdot \vec{x_i})$.
> 5. **if** $\hat{y_i} \neq y_i$ **then**
> 6. Set $\vec{w} \leftarrow \vec{w} + \vec{x_i}$ if y_i is positive, or $\vec{w} \leftarrow \vec{w} - \vec{x_i}$ if y_i is negative.
> 7. **end if**
> 8. **end for**
> 9. **end for**

Implement the initialization and prediction methods `__init__(self, examples, iterations)` and `predict(self, x)` in the `BinaryPerceptron class` according to the above specification.

During initialization, you should train the weight vector $\vec{w}$ on the input data using iterations passes over the data set, then store $\vec{w}$ as an internal variable for future use. Each example in the examples list will be a 2-tuple $(\vec{x}, y)$ of a data point paired with its binary label `True` or `False`.

The prediction method should take as input an unlabeled example $\vec{x}$ and compute the predicted label as sign($\vec{w} \cdot \vec{x}$), returning `True` if $\vec{w} \cdot \vec{x} > 0$ or `False` if $\vec{w} \cdot \vec{x} \leq 0$.

In this assignment, we will represent vectors such as $\vec{w}$ and $\vec{x}$ in Python as dictionary mappings from feature names to values. If a feature is absent from a vector, then its value is assumed to be zero. This allows for efficient storage of sparse, high-dimensional vectors, and encourages efficient implementations which consider only non-zero elements when computing dot products. Note that in general, the weight vector $\vec{w}$ will have a non-zero value associated with every feature, whereas individual instances $\vec{x}$ will have only a handful of non-zero values. You are encouraged to keep this asymmetry in mind when writing your code, as it can impact performance if not taken into account.

```
# Define the training and test data
>>> train = [({"x1": 1}, True), ({"x2": 1}, True),
... ({"x1": -1}, False), ({"x2": -1}, False)]

>>> test = [{"x1": 1}, {"x1": 1, "x2": 1},
... {"x1": -1, "x2": 1.5}, {"x1": -0.5, "x2": -2}]

# Train the classifier for one iteration
>>> p = BinaryPerceptron(train, 1)

# Make predictions on the test data
>>> [p.predict(x) for x in test]
[True, True, True, False]
```

2. **[20 points]** A multiclass perceptron uses the same linear classification framework as a binary perceptron, but can accommodate an arbitrary number of classes rather than just two. Given a set of data points and associated labels, where the labels are assumed to be drawn from some set $\{\ell_1, \cdots, \ell_m\}$, the goal is to learn a collection of weight vectors $\vec{w}_{\ell_1}, \cdots, \vec{w}_{\ell_m}$ such that $\text{argmax}_{\ell_k}(\vec{w}_{\ell_k} \cdot \vec{x})$ equals the correct label ℓ for each input pair $(\vec{x}, \ell)$.

The learning algorithm for this problem is similar to the one for the binary case. All weight vectors are first initialized to zero vectors, and then several passes are made over the training data, with the appropriate weight vectors being adjusted whenever a sample is misclassified.

Input:

A list T of training examples $(\vec{x_1}, y_1), \cdots, (\vec{x_n}, y_n)$, where the labels y_i are drawn from the set $\{\ell_1, \cdots, \ell_m\}$; the number of passes N to make over the data set.

Output:

The weight vectors $\vec{w_{\ell_1}}, \cdots, \vec{w_{\ell_m}}$.

Procedure:

1. Initialize the weight vectors as $\vec{w_{\ell_k}} \leftarrow 0$ for $k = 1, \cdots, m$.
2. **for** iteration $= 1$ **to** N **do**
3. **for** each example $(\vec{x_i}, y_i) \in T$ **do**
4. Compute the predicted label as $\hat{y_i} = \text{argmax}_{\ell_k}(\vec{w_{\ell_k}} \cdot \vec{x_i})$.
5. **if** $\hat{y_i} \neq y_i$ **then**
6. Increase the score for the correct class by setting $\vec{w_{y_i}} \leftarrow \vec{w_{y_i}} + \vec{x_i}$.
7. Decrease the score for the predicted class by setting $\vec{w_{\hat{y_i}}} \leftarrow \vec{w_{\hat{y_i}}} - \vec{x_i}$.
8. **end if**
9. **end for**
10. **end for**

Implement the initialization and prediction methods `__init__(self, examples, iterations)` and `predict(self, x)` in the `MulticlassPerceptron` class according to the above specification.

During initialization, you should train the weight vectors $\vec{w}_{\ell_k}$ on the input data using iterations passes over the data set, then store them as internal variables for future use. Each example in the examples list will be a 2-tuple $(\vec{x}, y)$ of a data point paired with its label. You will have to perform a single pass over the data set at the beginning to determine the set of labels which appear in the training examples. Do not make any assumptions about the form taken on by the labels; they might be numbers, strings, or other Python values.

The prediction method should take as input an unlabeled example $\vec{x}$ and return the predicted label $\ell = \text{argmax}_{\ell_k}(\vec{w}_{\ell_k} \cdot \vec{x})$.

```
''' Define the training data to be the corners and
    edge midpoints of the unit square '''
>>> train = [({"x1": 1}, 1), ({"x1": 1, "x2": 1}, 2), ({"x2": 1}, 3),
... ({"x1": -1, "x2": 1}, 4), ({"x1": -1}, 5), ({"x1": -1, "x2": -1}, 6),
... ({"x2": -1}, 7), ({"x1": 1, "x2": -1}, 8)]

# Train the classifier for 10 iterations so that it can learn each class
>>> p = MulticlassPerceptron(train, 10)

# Test whether the classifier correctly learned the training data
```

```
>>> [p.predict(x) for x, y in train]
[1, 2, 3, 4, 5, 6, 7, 8]
```

2. Applications [60 points]

In this section, you will use the general-purpose perceptrons implemented above to create classification systems for a number of specific problems. In each case, you will be responsible for creating feature vectors from the raw data, determining which type of perceptron should be used, and deciding how many passes over the training data should be performed. You will likely require some experimentation to achieve good results. The requisite data sets have been provided as Python objects in homework9_data.py, which has been pre-imported under the module name data in the skeleton file.

1. **[10 points]** Ronald Fisher's iris flower data set has been a benchmark for statistical analysis and machine learning since it was first released in 1936. It contains 50 samples from each of three species of the iris flower: iris setosa, iris versicolor, and iris virginica. Each sample consists of four measurements: the length and width of the sepals and petals of the specimen, in centimeters.

   Your task is to implement the `__init__(self, data)` and `classify(self, instance)` methods in the `IrisClassifier` class, which should perform training and classification on this data set. Example data will be provided as a list of 2-tuples $(\vec{x}, y)$, where $\vec{x}$ is a 4-tuple of real-valued measurements and y is the name of a species. Test instances will be provided in the same format as the $\vec{x}$ components of the training examples.

   ```
   >>> c = IrisClassifier(data.iris)
   >>> c.classify((5.1, 3.5, 1.4, 0.2))
   'iris-setosa'
   ```

   ```
   >>> c = IrisClassifier(data.iris)
   >>> c.classify((7.0, 3.2, 4.7, 1.4))
   'iris-versicolor'
   ```

2. **[10 points]** The National Institute of Standards and Technology has released a collection of bitmap images depicting thousands of handwritten digits from different authors. Though originally presented as 32×32 blocks of binary pixels, the data has been pre-processed by dividing the images into nonoverlapping blocks of 4×4 pixels and counting the number of activated pixels in each block. This reduces the dimensionality of the data, making it easier to work with, and also provides some robustness against minor distortions. Each processed image is therefore represented by a list of $8 \times 8 = 64$ values between 0 and 16 (inclusive), along with a label between 0 and 9 corresponding to the digit which was origially written.
   Your task is to implement the `__init__(self, data)` and `classify(self, instance)` methods in the `DigitClassifier` class, which should perform training and classification on this data set. Example data will be provided as a list of 2-tuples $(\vec{x}, y)$, where $\vec{x}$ is a 64-tuple of pixel counts between 0 and 16 and y is the digit represented by the image. Test instances will be provided in the same format as the $\vec{x}$ components of the training examples.

```
>>> c = DigitClassifier(data.digits)
>>> c.classify((0,0,5,13,9,1,0,0,0,0,13,15,10,15,5,0,0,3,
... 15,2,0,11,8,0,0,4,12,0,0,8,8,0,0,5,8,0,0,9,8,0,0,4,11,
... 0,1,12,7,0,0,2,14,5,10,12,0,0,0,0,6,13,10,0,0,0))
0
```

3. **[10 points]** A simple data set of one-dimensional data is given in `data.bias`, where each example consists of a single positive real-valued feature paired with a binary label. Because the binary perceptron discussed in the previous section contains no bias term, a classifier will not be able to directly distinguish between the two classes of points, despite them being linearly separable. To see why, we observe that if the weight vector (consisting of a single component) is positive, then all instances will be labeled as positive, and if the weight vector is negative, then all instances will be labeled as negative. It is therefore necessary to augment the input data with an additional feature in order to allow a constant bias term to be learned.

   Your task is to implement the `__init__(self, data)` and `classify(self, instance)` methods in the `BiasClassifier` class to perform training and classification on this data set. Example data will be provided as a list of 2-tuples $(\vec{x}, y)$ of real numbers paired with binary labels. Test instances will be single numbers, and should be classified as either `True` or `False`. As discussed above, instances will have to be augmented with an additional feature before being fed into a perceptron in order for proper learning and classification to take place.

```
>>> c = BiasClassifier(data.bias)
>>> [c.classify(x) for x in (-1, 0, 0.5, 1.5, 2)]
[False, False, False, True, True]
```

4. **[15 points]** A mystery data set of two-dimensional data is given in `data.mystery1`, where each example consists of a pair of real-valued features and a binary label. As in the previous problem, this data set is not linearly separable on its own, but each instance can be augmented with one or more additional features derived from the two original features so that linear separation is possible in the new higher-dimensional space.

   Your task is to implement the `__init__(self, data)` and `classify(self, instance)` methods in the `MysteryClassifier1` class to perform training and classification on this data set. Example data will be provided as a list of 2-tuples $(\vec{x}, y)$ of pairs of real numbers and their associated binary labels. Test instances will be pairs of real numbers, and should be classified as either `True` or `False`. Instances will have to be augmented with one or more additional features before being fed into a perceptron in order for proper learning and classification to take place. We recommend first visualizing the data using your favorite plotting software in order to understand its structure, which should help make the appropriate transformation(s) more apparent.

Square both feature value

```
>>> c = MysteryClassifier1(data.mystery1)
>>> [c.classify(x) for x in ((0, 0), (0, 1),
    ... (-1, 0), (1, 2), (-3, -4))]
[False, False, False, True, True]
```

5. **[15 points]** Another mystery data set of three-dimensional data is given in `data.mystery2`, where each example consists of a triple of real-valued features paired with a binary label. As in the previous few problems, this data set is not linearly separable on its own, but each instance can be augmented with one or more additional features so that linear separation is possible in the new higher-dimensional space.

   Your task is to implement the `__init__(self, data)` and `classify(self, instance)` methods in the `MysteryClassifier2` class to perform training and classification on this data set. Example data will be provided as a list of 2-tuples $(\vec{x}, y)$ of triples of real numbers paired with binary labels. Test instances will be triples of real numbers, and should be classified as either `True` or `False`. Instances will again have to be augmented with one or more additional features before being fed into a perceptron in order for proper learning and classification to take place. As before, we recommend first visualizing the data using your favorite plotting software in order to understand its structure, then thinking about what transformation(s) might help separate the two classes of data.

```
>>> c = MysteryClassifier2(data.mystery2)
>>> [c.classify(x) for x in ((1, 1, 1), (-1, -1, -1),
    (1, 2, -3), (-1, -2, 3))]
[True, False, False, True]
```

3. Feedback [5 points]

1. **[1 point]** Approximately how many hours did you spend on this assignment?

2. **[2 point]** Which aspects of this assignment did you find most challenging? Were there any significant stumbling blocks?

3. **[2 point]** Which aspects of this assignment did you like? Is there anything you would have changed?