Roy Howie and Lingyue Zheng Wednesday, October 12, 2016 EEN548 project 1

Task
Experimenting with selected domains from the UCI repository, compare the behavior of linear and polynomial classifiers using perceptron learning. Observe how the former wins in simple domains and the latter in highly non-linear domains.

Background
The perceptron algorithm is a form of linear or polynomial classifier guaranteed to find a solution, if one exists.

It works by initializing a set of random weights. A linear classifier will have a number of weights equal to one more than the number of attributes. For example, a linear classifier on the Iris data set (more info below) would have 5 weights: one per attribute plus the constant weight.

An instance in the dataset is evaluated for learning by taking the dot product of its attributes with the weight vector; the constant weight is always included. If the dot product is positive, perceptron hypothesizes that the instance *is* a member of whichever class perceptron is classifying. Similarly, if the dot product is non-positive, perceptron hypothesizes that the instance is *not* a member.

For example, consider the first instance of the Iris data set: `[5.1 3.5 1.4 0.2 Iris-setosa]` and the set of weights `[-0.5 0 0 0 0.5]`. Suppose perceptron is classifying the data into `Iris-setosa (true)` or `not Iris-setosa (false)`. The dot product of these two vectors is

```
1*-0.5 + 0*5.1 + 0*3.5 + 0*1.4 + 0.5*0.2 = -0.5 + 0.1 = -0.4
```

which is less than zero. Thus, perceptron would incorrectly hypothesize `false`.

If perceptron's hypothesis matches an instance's actual classification, nothing is done. If, however, perceptron is in disagreement with the data set, it updates its weights. This is done by adding or subtracting a *learning factor* to the weights which contributed to the dot product. In the above example, only `w0` and `w4` contributed to the sum, so they would be adjusted accordingly.

Perceptron can also be used as a polynomial classifier. The trick is to assign a weights not only for attributes but also for their pairings. For example, if there are 3 attributes `[x, y, z]` and we want to consider a polynomial of order 2, there would be 10 weights:

```
c, x, y, z, xx, xy, xz, yy, yz, zz
```

The number of weights corresponds to `(attributes + order) choose (order)`.

Goal

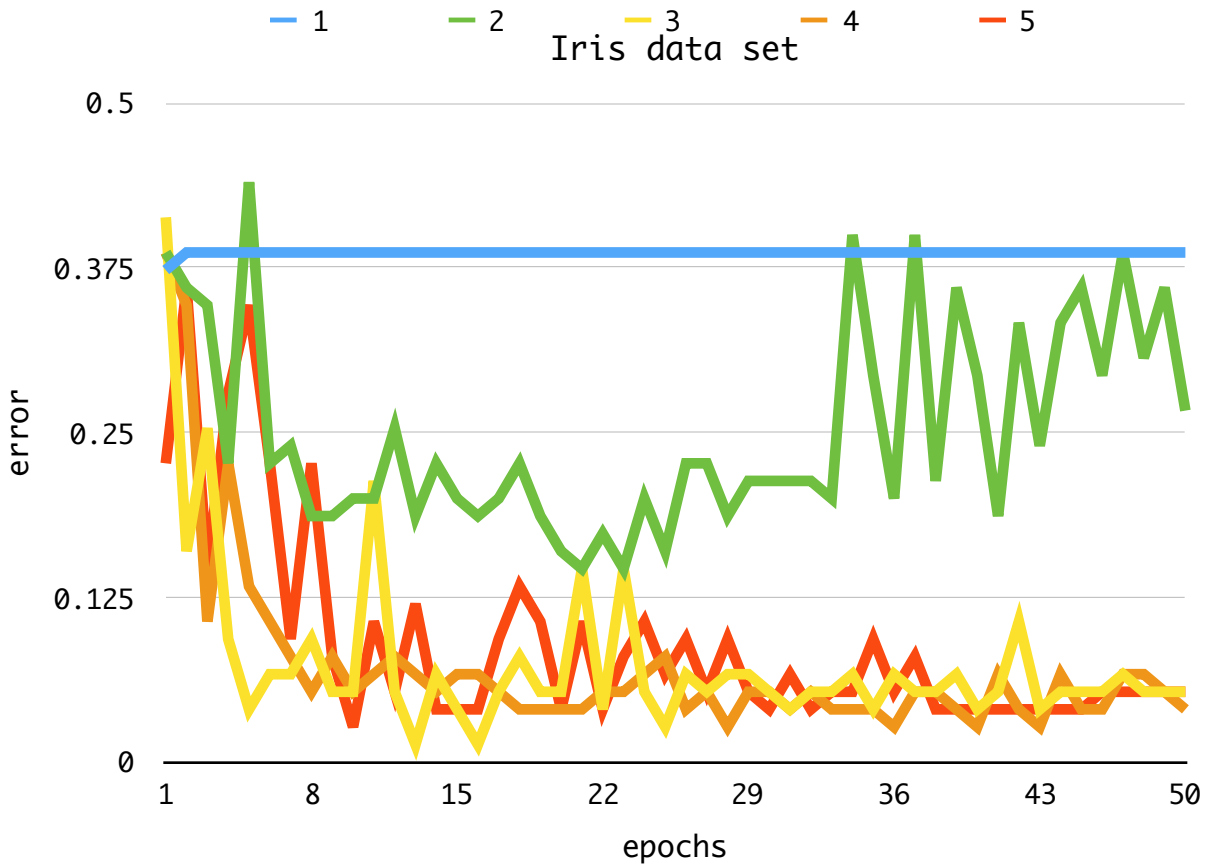
We will examine several data sets, namely the Iris data set, the skin segment data set, the fertility data set, and the banknote authentication data set. Each data set will be evaluated by perceptron using polynomials from the 1st to 5th order. We will comment on how different orders affected perceptron's ability to classify each set. We will generally not comment on speed, as perceptron was implemented in C++ and even for large data sets, e.g. the skin segment data set with 250k instances, the program was able to classify the data set over 50 for polynomials of high order in insignificant time periods. Most of the time running was actually spent printing the results. A learning factor of 0.2 was used for all tests.

Iris Data Set

The Iris data set is considered an exceedingly simple domain. It is one of the most widely cited and used because of how perfect it is. Here is the breakdown:

- 150 instances
- 4 attributes: sepal length, sepal width, petal length, petal width
- 3 classes (50 instances per class): Iris-setosa, Iris-versicolor, Iris-virginica
- no missing values

One class *is* linearly separable from the other two; the latter are *not* linearly separable from one another.



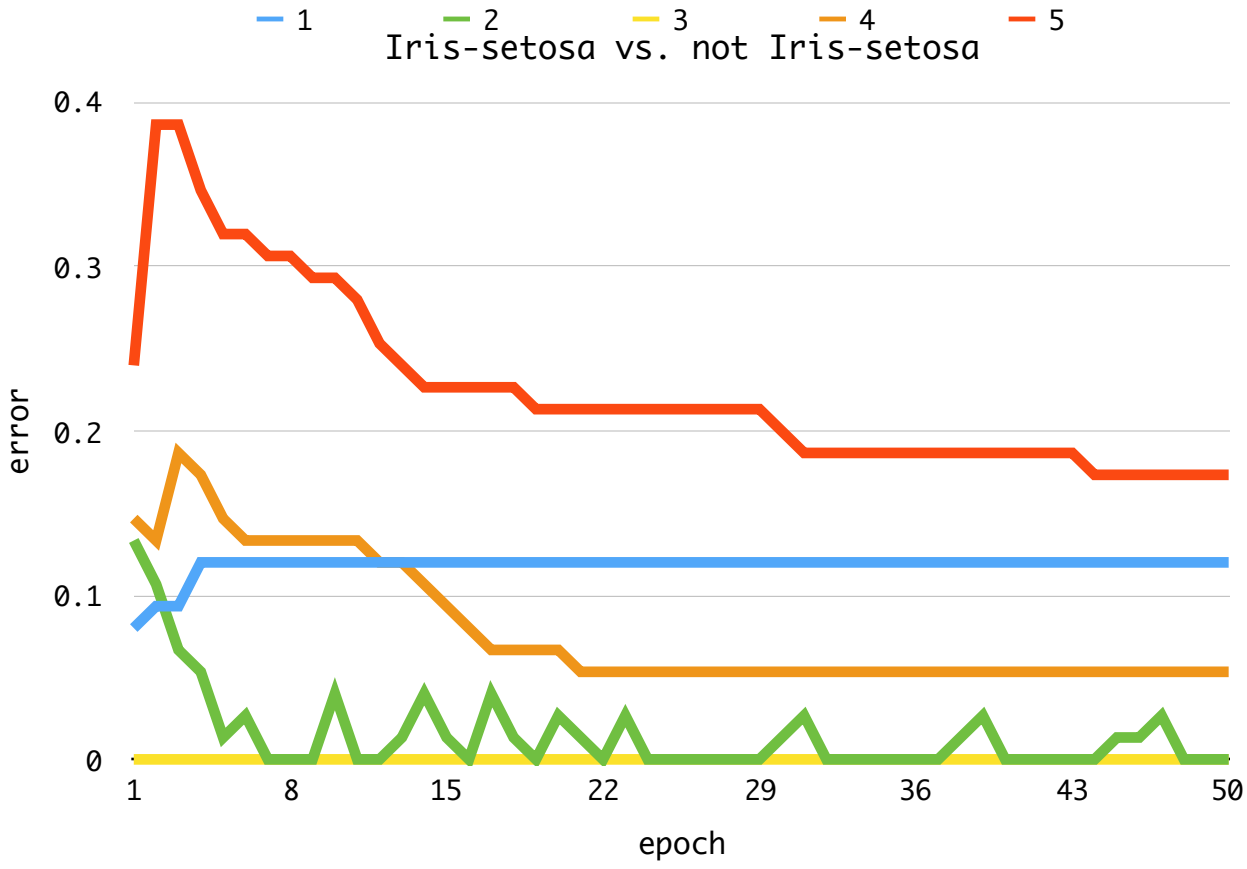
Here are the error rates after the 50th epoch:

	1	2	3	4	5
50	0.386667	0.266667	0.0533333	0.04	0.0533333

The linear classifier did not do so well, classifying over a third of the testing set incorrectly; the 2nd order polynomial did slightly better with only a quarter incorrect guesses. The 3rd, 4th, and 5th order polynomials, however, did exceedingly better, hovering around 5% error.

Interestingly, the 3rd order classifier had an even better testing result near the 15th epoch. In fact, it was able to achieve a mere 1.33% error rate after the 16th epoch.

The UCI machine learning repository noted that the first class was linearly separable from the other two, so let's have a look:

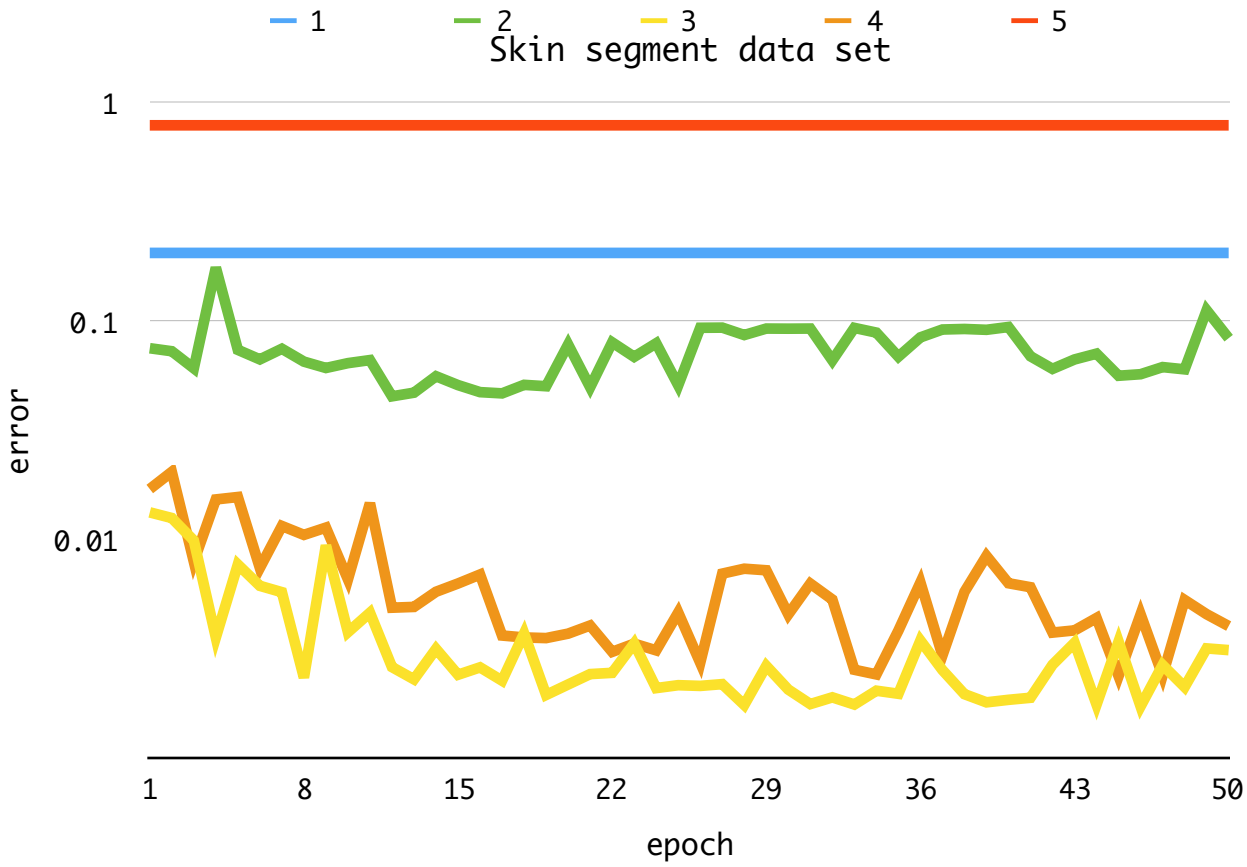


Indeed, Iris-setosa was able to be separated from the other two classes with a 0% error rate using either a 2nd or 3rd order polynomial. Our implementation of perceptron was unable to induce a separation using a linear classifier, however.

Skin segment data set

The skin segment data set is a complicated and very large data set. Here is the breakdown:

- 245,057 instances
- 3 attributes: blue, green, and red color values from images of faces
- 2 classes: 50,859 are skin samples, 194,198 are not skin samples
- no missing values



For reference, here are the last five rows:

	1	2	3	4	5
45	0.206669	0.0566315	0.00343592	0.00239127	0.793331
46	0.206669	0.0575537	0.00173836	0.00457851	0.793331
47	0.206669	0.0620425	0.00268508	0.00240759	0.793331
48	0.206669	0.0605816	0.00213011	0.00532935	0.793331
49	0.206669	0.112977	0.00319924	0.00458667	0.793331
50	0.206669	0.0841352	0.00314211	0.00404802	0.793331

The linear classifier performed mediocrely, incorrectly classifying over a fifth of the data set. The 2nd order polynomial was passable, hovering just below a 10% error rate.

The 3rd and 4th order polynomials, on the other hand, were beyond impressive. They both managed to classify the immense data set with under 1% error. In fact, the 3rd order polynomial had a minimum error of 0.21% after the 48th epoch; the 4th order polynomial had a minimum error of 0.23%.

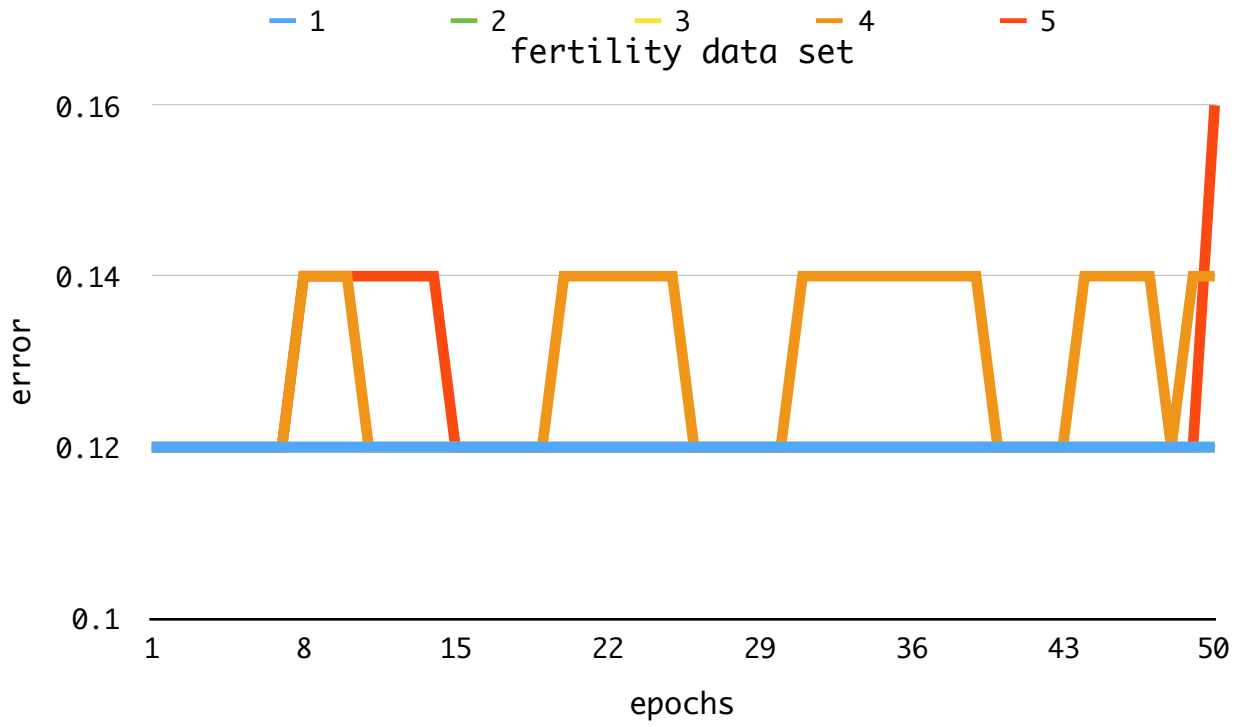
The 5th order polynomial was dreadful with its 79.3% error rate, yielding a classifier worse than a coin toss. Perhaps this occurred because of overfitting. As there are only 3 attributes, maybe it does not make sense to have a 5th order polynomial. This may have caused the classifier to get stuck in a slump, i.e., caught up in a local minimum of which the learning factor could not push it out.

We would consider this data set to be separable. The very small error achieved by the 3rd and 4th order polynomials may have been due to noise, such as inconsistent data or incorrect data recordings. Regardless, the error was so minuscule as to be essentially irrelevant.

Fertility data set

The fertility data set was a simple domain. Here is the breakdown:

- 100 instances
- 10 attributes (* = binary, real otherwise): season, childhood diseases*, accident or serious trauma*, surgical intervention*, high fevers in the last year*, frequency of alcohol consumption, smoking habit, hours spent sitting per day
- 2 classes: normal diagnosis and altered diagnosis



Here is the last row:

	1	2	3	4	5
50	0.12	0.12	0.12	0.14	0.16

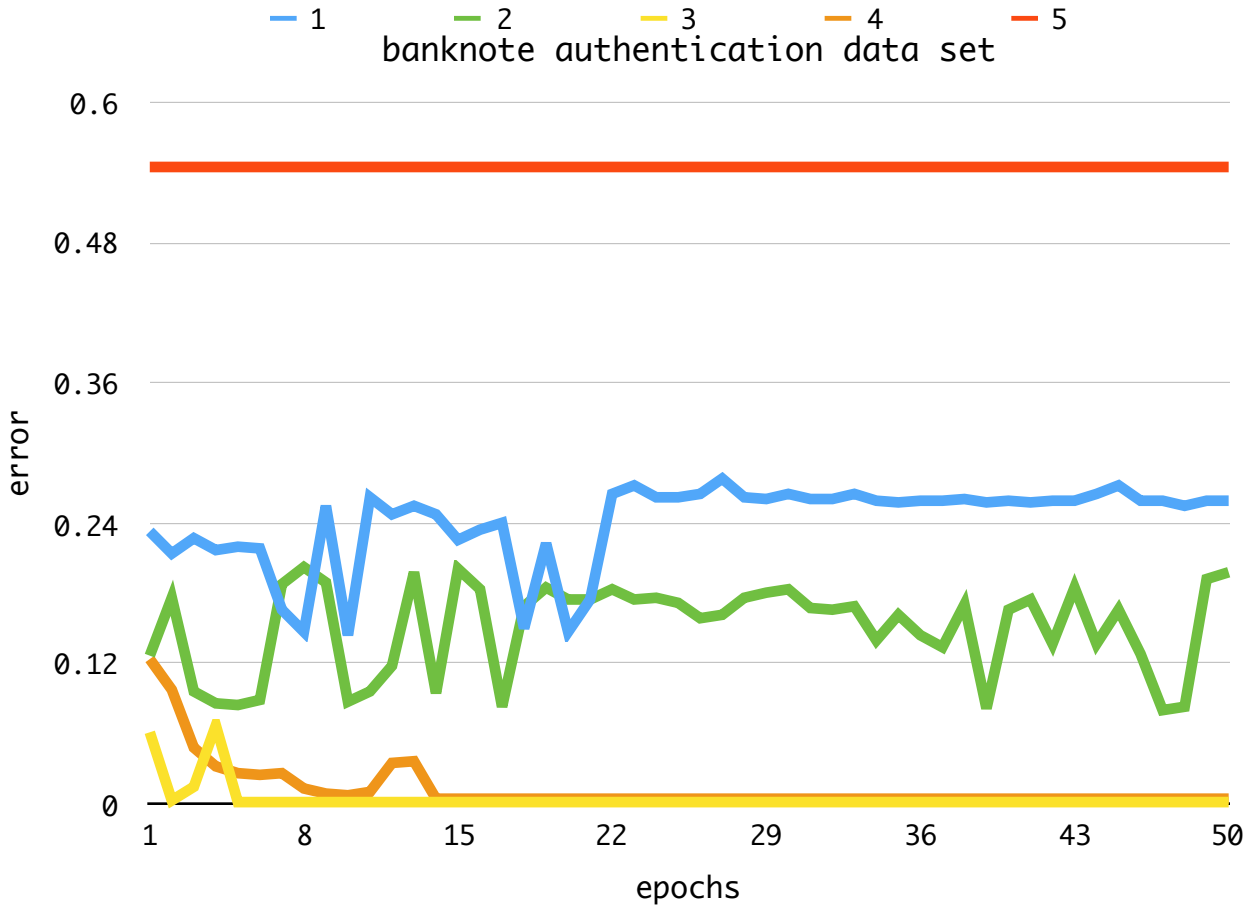
Because this was a simple domain, the linear classifier reigned supreme. The minimum error achieved was 12%. Indeed, perceptron was run on the domain (data not included in graph) for 6th through 10th order polynomials and was unable to achieve an error rate lower than the one given by the linear classifier.

Thus, the linear classifier would ultimately be most desirable, as it assumes the least, runs the fastest, and achieves the lowest possible error. Yes, the 2nd and 3rd order polynomials achieved the same error rate, but there is no reason to use a more complex classifier if a simple gets the job done.

Banknote Authentication data set

The banknote data set was more complex. It consists of data extracted from images of forged and genuine banknotes, which were extracted using wavelet transformed images (WTI).

- 1372 instances
- 4 attributes: variances, skewness, and curtosis of WTI; entropy of the image
- 2 classes: forged (762) or genuine (610)



Here are the last five rows:

	1	2	3	4	5
46	0.259475	0.12828	0.00145773	0.00437318	0.54519
47	0.259475	0.0801749	0.00145773	0.00437318	0.54519
48	0.255102	0.0830904	0.00145773	0.00437318	0.54519
49	0.259475	0.19242	0.00145773	0.00437318	0.54519
50	0.259475	0.198251	0.00145773	0.00437318	0.54519

As expected, the linear classifier did not fare so well on this data set, incorrectly classifying a quarter of the testing examples.

The 2nd order polynomial did slightly better, achieving a minimum error rate of about 8%. While it consistently hovered around 19% error, the 2nd order polynomial could be improved with a slight change of algorithm. The perceptron algorithm could record the weights of the lowest error rate achieved throughout the epochs and ultimately report those as the solution.

The 3rd and 4th order polynomials did extremely well, achieving under 1% error. In fact, the 3rd order polynomial was able to classify the testing set with only 0.14% error. Similarly, the 4th order polynomial settled on a set of weights which achieved a 0.43% error rate. This has incredibly useful real-world applications. Being able to determine whether a banknote is forged after a wavelet transform test with such low error would be extremely valuable to banks, almost eliminating the problems caused by forgery.

Simply put, the 5th order polynomial was terrible with its 54% error rate. This is about the same as guessing, as the data set was split 762 to 610, or 55.5% to 44.5%. A bank would not be wise to implement a 5th order polynomial classifier using perceptron.

Conclusion

Perceptron was highly effective in classifying data sets. On simple domains, such as the fertility data set, the linear classifier was most effective, in part because of its simplicity compared to higher order polynomials. On the Iris data set, which is considered to be exceedingly simple, our implementation of perceptron was not able to achieve a reasonable linear classifier. When considering a single class in the data set, however, perceptron was able to separate reasonably well using a linear classifier, but only found a perfect separation using 2nd and 3rd order polynomials.

On more complex domains, such as the skin segment data set and the banknote authentication data set, perceptron performed best using polynomial classifiers of higher order, i.e., 3rd and 4th order polynomials.

As detailed above, one slight modification to the perceptron algorithm would be well worth exploring: recording the weights from the epoch in which the minimum error rate was achieved. While perceptron is guaranteed to converge if a solution exists, that might depend on the learning factor. Hence, it would also be worth exploring whether a changing learning factor would produce better results.

Data Set sources

https://archive.ics.uci.edu/ml/datasets/Iris
https://archive.ics.uci.edu/ml/datasets/Skin+Segmentation
https://archive.ics.uci.edu/ml/datasets/Fertility
https://archive.ics.uci.edu/ml/datasets/banknote+authentication