COMP 557: Assignment #7

Due on Mond, December 2, 2016

Misiura Mikita & Lee Call

Problem 1. Naive Bayes, Perceptrons, Decision Trees, Neural Networks

Part a. Naive Bayes

- Probabilities:

$$P(Income \mid \leq 50k) = 0.6 \tag{1}$$

$$P(Income \mid > 50k) = 0.4 \tag{2}$$

Education	P(Education$\mid\leq 50k$)	P(Education$\mid> 50k$)
BS	1.0	0.0
MS	0.0	1.0
PhD	0.5	0.5

Gender	P(Gender$\mid\leq 50k$)	P(Gender$\mid> 50k$)
male	0.6	0.4
female	0.6	0.4

Citizenship	P(Citizenship$\mid\leq 50k$)	P(Citizenship$\mid> 50k$)
US	0.50	0.50
nonUS	0.75	0.25

- Result is:

$$\begin{aligned} &P(\leq 50k \mid PhD, male, US) = \\ &\frac{P(PhD \mid\leq 50k)\cdot P(male \mid\leq 50k)\cdot P(US \mid\leq 50k)\cdot P(\leq 50k)}{\sum_{Income} P(PhD \mid Income)\cdot P(male \mid Income)\cdot P(US \mid Income)\cdot P(Income)} = \\ &\frac{0.5\cdot 0.6\cdot 0.5\cdot 0.6}{0.5\cdot 0.6\cdot 0.5\cdot 0.6 + 0.5\cdot 0.4\cdot 0.5\cdot 0.4} = 0.70 \end{aligned} \tag{3}$$

etc.

Education	Gender	Citizenship	Income
PhD	male	US	$\leq 50k$
PhD	male	nonUS	$\leq 50k$
MS	female	nonUS	$> 50k$

Part b. The perceptron algorithm

- We need to convert categorical data into numerical and to add column of ones for bias term. Example encoding scheme is as follows:

Education:

BS	MS	PhD
-1	0	1

Gender:

male	female
-1	1

Citizenship:

US	nonUS
-1	1

Income:

$> 50k$	$\leq 50k$
-1	1

So the data looks like this:

Bias	Education	Gender	Citizenship	Income
1	-1	-1	-1	1
1	0	-1	1	-1
1	-1	1	-1	1
1	1	-1	1	1
1	0	1	-1	-1
1	1	1	1	1
1	-1	-1	-1	1
1	1	-1	-1	-1
1	-1	1	1	1
1	1	1	-1	-1

- Result:

Init	1	2	3	4	5	6	7	8	9	10
0	0	-1	-1	0	-1	-1	0	-1	-1	-1
0	0	0	0	1	1	1	0	-1	-1	-1
0	0	1	1	0	-1	-1	-2	-1	-1	-1
0	0	-1	-1	0	1	1	0	1	1	1

- It's hard to say, but this data does not look like linearly separable. So no. (We wrote the code to test it and yes, it is not converging, weights are stuck jumping between two values)

Part c. Decision Trees

- Calculations:

$$Entropy(Income) = -\frac{6}{10}log_2\frac{6}{10} - \frac{4}{10}log_2\frac{4}{10} = 0.97 \tag{4}$$

Gender:

$$Entropy(Income \mid Gender) = \frac{1}{2}\left[-\frac{6}{10}log_2\frac{6}{10} - \frac{4}{10}log_2\frac{4}{10}\right] + \frac{1}{2}\left[-\frac{6}{10}log_2\frac{6}{10} - \frac{4}{10}log_2\frac{4}{10}\right] = 0.97 \tag{5}$$

$$IG(Income \mid Gender) = 0 \tag{6}$$

Citizenship:

$$Entropy(Income \mid Citizenship) = \frac{6}{10}\cdot 1 + \frac{4}{10}\left[-\frac{3}{4}log_2\frac{3}{4} - \frac{1}{4}log_2\frac{1}{4}\right] = 0.93 \tag{7}$$

$$IG(Income \mid Citizenship) = 0.04 \tag{8}$$

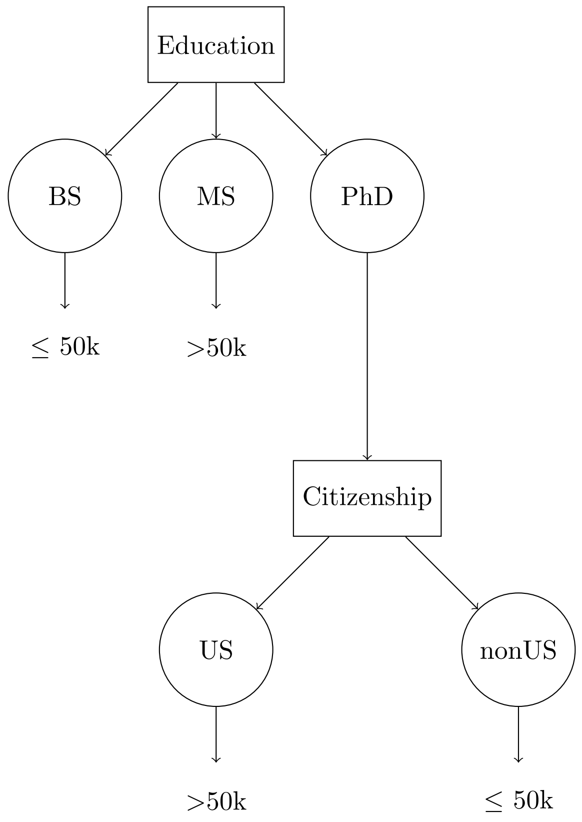
Education:

$$Entropy(Income \mid Education) = \frac{4}{10} \cdot 0 + \frac{2}{10} \cdot 0 + \frac{4}{10} \cdot 1 = 0.40 \tag{9}$$

$$IG(Income \mid Education) = 0.57 \tag{10}$$

So the first split should be on Education.

- Calculations for the rest of a tree are obvious: split on Gender gives us IG=0, split on Citizenship gives us IG=1. So the next and final split is on Citizenship. Our tree:



- Result:

Education	Gender	Citizenship	Income
PhD	male	US	$> 50k$
PhD	male	nonUS	$\leq 50k$
MS	female	nonUS	$> 50k$

Part d. Neural networks

- We should convert all categorical features into numerical ones, rescale the data to be between 0 and 1 and we should add bias term:

Education:

BS	MS	PhD
0	0.5	1

Gender:

male	female
0	1

Citizenship:

US	nonUS
0	1

Income:

$> 50k$	$\leq 50k$
-1	1

So the data looks like this:

Bias	Education	Gender	Citizenship	Income
1	0	0	0	1
1	0.5	0	1	-1
1	0	1	0	1
1	1	0	1	1
1	0.5	1	0	-1
1	1	1	1	1
1	0	0	0	1
1	1	0	0	-1
1	0	1	1	1
1	1	1	0	-1

- The code can be found in the file ”nnet.py”. Cross-validation was made by using 20% to test (2 points) and 80% to train. Resulting accuracies reported as averages over all possible choices of test points:

Hidden units	Test acuracy
2	0.46
3	0.50
4	0.53
5	0.53
6	**0.57**
7	0.50
8	0.53
9	0.53

- Predictions:

Education	Gender	Citizenship	Income
PhD	male	US	$> 50k$
PhD	male	nonUS	$\leq 50k$
MS	female	nonUS	$\leq 50k$

Comparison:

Well, the dataset is very small, so no wonder that all methods give a different predictions. The second example is the easiest, so it is being classified always the same, but the other two are being classified into different categories by different classifiers.

Part e. Using the full data set

The code for this part can be found in the file "compare.py". In all cases we used all the features. Some comments on what we did and observed:

1. For Naive Bayes we just converted all categorical data into numerical and threw it in. We tried splitting continuous data into categories, but it did not improve the result, even lowered a bit our accuracy. Normalization, obviously, does not matter here.

2. For perceptron we added bias term and converted categorical data into numerical. Normalization is not really very important here: it boosts test accuracy from 78.3% to 78.6% which is not much.

3. For decision trees we just threw the data in. This method just off the box did a good job with this data - 82%.

4. For neural network we converted categorical data into numerical and threw the data in. To choose our net structure we used cross-validation with varying number of hidden layers and units in them (takes a while of course, commented out in the current code). The best structure we found is with two hidden layers with 100 units in each. These settings can be significantly lowered to make a faster code, but this gave us just a bit higher accuracy. We could use a single layer, but two is better, and third layer does not give rise in accuracy. Normalization is really important here: it raises test accuracy from 77% to 85%! We also tried different kinds of nonlinearities (ReLU, Sigmoin, Tanh, no nonlinearity), but ReLU is the best (others are commented out in the code to speed up the script).

Overall, neural network gave us the best accuracy (85%) for additional computational and optimization cost. Other methods gave easy result, but with about 78%-82% accuracy. This in general in agreement with supplemental information on the site, which gives the same range for accuracies showed by a range of methods.

Interesting that all the methods, except for decision trees (they were simply overfitted because there were no pruning, they can very easily fit any data, unlike other methods), have very close train and test accuracies. We think, that this means that we are getting close to the maximum accuracy reachable on this dataset.

The accuracy of Naive Bayes and Decision Trees can be probably increased by accurate binning of continuous features (with roughly equivalent number of examples in every bin, we tried only naive binning), but we think (hope) that this is beyond necessary work for this problem.

Results are:

Method	Features	Train acc	Test acc
Naive bayes	All	0.796	0.796
Perceptron	All	0.780	0.783
Decision trees	All	0.999	0.815
Neural network	All	0.853	0.851

Problem 2. Text classification

Part 2.1.1

Dev errors:

	k = 10,000	k = 20,000	k = 30,000
n = 1	0.13	0.16	0.49
n = 2	0.20	0.12	0.47
n = 3	0.25	0.12	0.46

It seems like the best choice is around $n = 2$, $k = 20,000$.

Part 2.1.3

We got a dev error of 3.7% with unigram features when we were replacing newline symbols with spaces before splitting the example into words. Without this, our dev error was 4.9%.

Dependence on number of examples:

- With unigram features:

#	500	1000	1500	2000	2500	3000	3500	4000	4500	5000
TR, %	0.00	0.00	0.07	0.05	0.44	0.10	0.43	0.18	0.16	0.50
DEV, %	8.41	6.73	5.11	3.99	3.55	3.80	3.74	3.68	3.24	3.80

- With bigram features with removed punctuation symbols:

#	500	1000	1500	2000	2500	3000	3500	4000	4500	5000
TR, %	0.00	0.00	0.00	0.00	0.08	0.00	0.00	0.00	0.00	0.38
DEV, %	11.53	9.22	6.79	6.04	4.98	4.92	4.67	4.05	3.99	4.36

- With bigram features and with all the symbols:

#	500	1000	1500	2000	2500	3000	3500	4000	4500	5000
TR, %	0.00	0.00	0.00	0.00	0.56	0.00	0.00	0.00	0.00	0.00
DEV, %	11.03	8.85	7.17	6.17	6.48	4.92	4.55	4.55	4.42	3.99

The result is not surprising - the more examples we get, the higher the training error in general (less overfitting) and the lower the dev error. Although for bigram features it almost always zero anyway. Unigram features are better than bnigram (this is a real surprise). As to the removing of punctuation symbols, it is really hard to say whether it worth it or not - for some number of iterations this works better, for some it works worse, but the best combination we found is: unigram features, 4500 training examples with dev error 3.24%.

Part 2.2

In this part of the problem we tried one additional modification to the procedure: we tried not to use different punctuation formatting signs that were in the texts. The idea behind that was that the information needed for classification is contained in the words, but not in the punctuation, that punctuation only increases number of features and slows down the classification and decreases accuracy.

Our errors after 20 iterations:

1. With unigram features: 3.8% training error, 17.4% dev error.
2. With bigram features and all punctuation in place: 0.1% training error, 18.0% dev error.
3. With bigram features and no punctuation: 3.5% training error, 25.3% dev error.

Looks like keeping all punctuation symbols really helps with classification. This is maybe just because it gives more features, or because positive and negative reviews have a somewhat different punctuation. More surprizing result is that unigram features work even better than bigram!

Dependence of the test and training errors on the number of iterations:

# iters	Train (no punct)	Dev (no punct)	Train (w punct)	Dev (w punct)
1	0.50	0.51	0.48	0.50
2	0.46	0.50	0.50	0.51
3	0.09	0.21	0.09	0.26
4	0.26	0.40	0.06	0.20
5	0.05	0.19	0.27	0.40
6	0.08	0.28	0.50	0.51
7	0.08	0.22	0.07	0.23
8	0.02	0.15	0.01	0.17
9	0.50	0.51	0.03	0.19
10	0.03	0.22	0.04	0.20
11	0.01	0.17	0.05	0.19
12	0.49	0.51	0.01	0.16
13	0.01	0.19	0.00	0.17
14	0.01	0.15	0.00	0.18
15	0.00	0.16	0.48	0.50
16	0.00	0.16	0.00	0.16
17	0.00	0.17	0.00	0.16
18	0.00	0.20	0.00	0.16
19	0.00	0.18	0.00	0.16
20	0.00	0.18	0.00	0.16

These results are also presented in the figures 1 and 2.

These results show non-monotonic dependence of the errors on the number of iterations used to train perceptron. This is most likely because our data is actually not separable and we use way too small number of iterations. This leads to the fact that our weights still change significantly every iteration. This seems like weights are bouncing between three close values for some time (the period of that peaks on the graph). This also explains odd result that unigram features work better than bigram - it is probably just fluctuation and doesn't mean than unigrams will work better in general.

Overall there is no clear difference between two ways to treat punctuation signs. Perhaps, we should do many more iterations to get more meaningful results and compare. More training examples will also help.

Part 2.3

Our errors after 20 iterations:

1. With unigram features: 0.0% training error, 10.8% dev error.
2. With bigram features and all punctuation in place: 0.0% training error, 12.0% dev error.
3. With bigram features and no punctuation: 0.0% training error, 12.4% dev error.

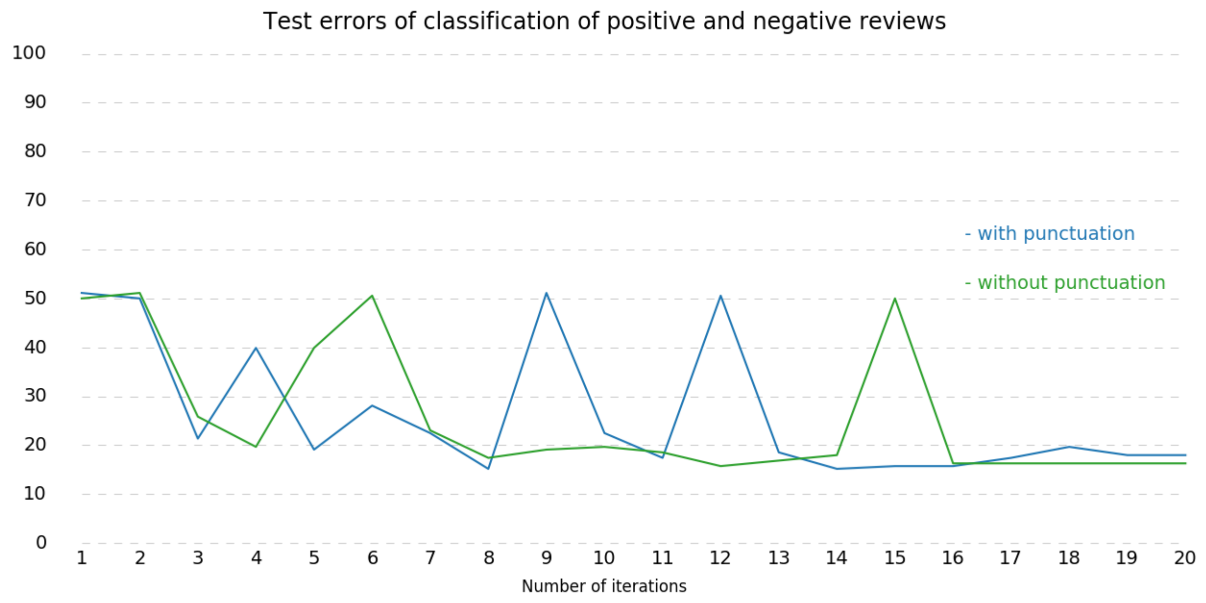


Figure 1: Dependence of the test (dev) errors of classification of the positive and negative movie reviews on the number of iterations for perceptron training.

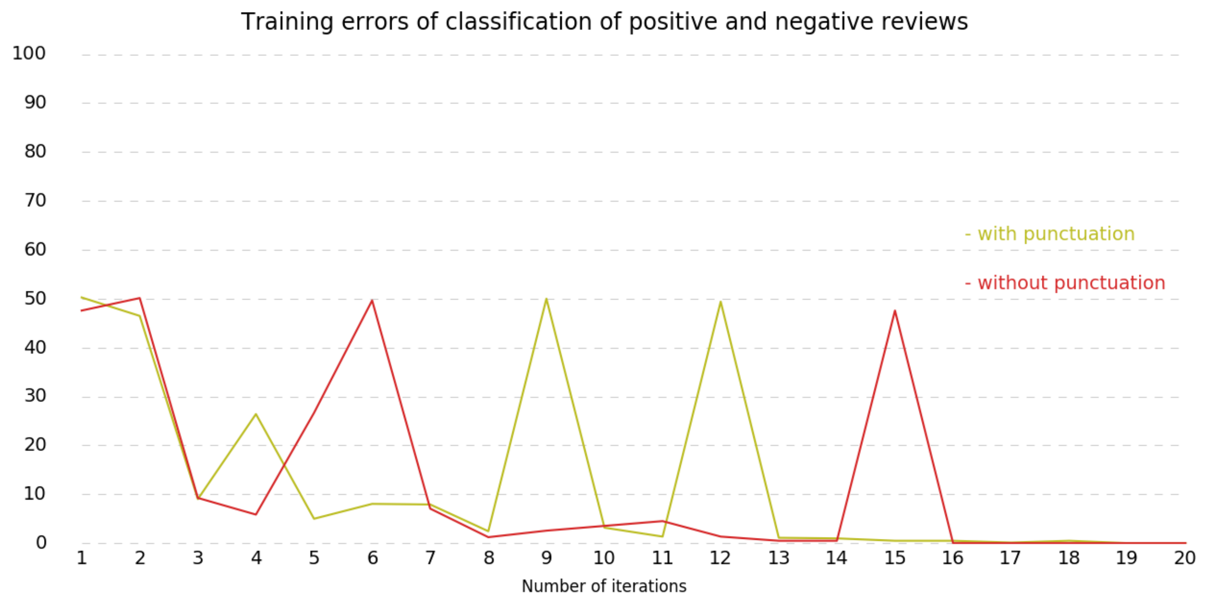


Figure 2: Dependence of the training errors of classification of the positive and negative movie reviews on the number of iterations for perceptron training.

Strange enough, unigram features seem to work better than bigram features.

Problem 3. Image classification

Part a. Warmup

When we train the perceptron classifier on the raw pixels using 500 images by running

```
python run.py --pixels
```

we see the error rate drops to near 0.0 after about 30 iterations. However, the test accuracy is only about 0.54. Pixel intensity values are not so meaningful it turns out.

Part b. K-means

Our strategy is to break all the images into "patches" of 8 x 8 pixels, then use k-means clustering on the training set to define centroids. We will then use the relationship between these centroids and the patches which comprise an image in order to generate more meaningful features. The features thus generated will hopefully be more meaningful than pixel intensities and will improve our classifier's accuracy. After filling out the `runKMeans` method we are able to check that our algorithm is learning sensible features by running

```
python run.py --view
```

which shows us the following 25 centroids:

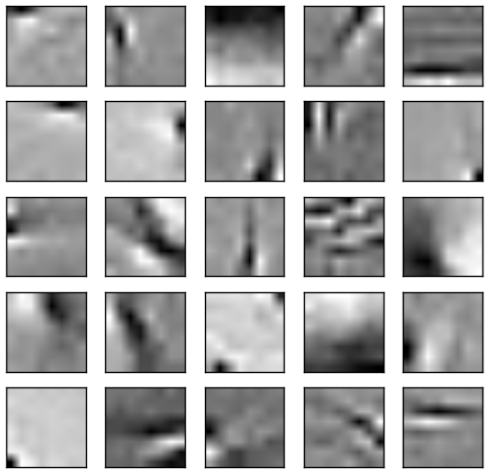


Figure 3: First 25 learned centroids using K-means clustering.

Part c. Feature Extraction

Next, we use the centroids found by our K-means algorithm to generate features that will be more useful for classification than the raw pixel intensities. We do this by implementing the `extractFeatures` method in `submission3.py`. This method defines a new feature for each patch in an image based on the distance between this patch and the centroids found previously using K-means. These new features will be used in the next part when we do supervised training.

Part d. Supervised Training

Now when we run

```
python run.py --gradient=perceptron
```

we see that using our new features in place of the pixel intensities we have improved the test accuracy from 0.54 to 0.65. However, after computing the gradient of the logistic loss function with respect to w,

$$Loss_{logistic}(w, \phi(x), yy) = -\frac{\phi(x) * yy}{e^{w^T \phi(x) * yy} + 1} \tag{11}$$

and implementing the `logisticGradient` method, we run

```
python run.py --gradient=logistic
```

and see that the test accuracy has improved further from 0.65 to 0.73. This despite the fact that after 300 training iterations the training set accuracy remains at 0.82. So we see we have avoided some overfit by using the new centroid-based patch features and our logistic loss gradient.

We also computed the sub-gradient of the hinge loss function, with respect to `w`, as

$$Loss_{hinge}(w, \phi(x), yy) = \begin{cases} -\phi(x) * yy, & \text{if } w^T \phi(x) * yy < 1 \\ 0, & \text{if } w^T \phi(x) * yy > 1 \end{cases} \tag{12}$$

Now when we run

```
python run.py --gradient=hinge
```

where we train using 500 images, we see that the test accuracy improves from the 0.65 when using perceptron gradient to 0.71 with the hinge gradient. If we then run

```
python run.py --gradient=hinge -m
```

where we train K-means using 1000 images, we improve the accuracy to 0.72.

Problem 4. Relating Naive Bayes classifiers and perceptrons

A naive Bayes classifier with binary features has conditional probabilities of *P(y)*, *P(f_i| y)*, and *P(f_i| y)*. The naive Bayes classification rule is

$$argmax_y P(y) \prod_{i=1}^{n} P(f_i|y) \tag{13}$$

which, by definition of the argmax and logarithm functions (and taking advantage of the fact the log of a product is the sum of the logs), can be equivalently stated as

$$argmax_y logP(y) + \sum_{i=1}^{n} logP(f_i|y)$$

which, can be rewritten and simplified as follows

$$= argmax_y logP(y) + \sum_{i=1}^{n} [(f_i) logP(F_i = 1|y) + (1 - f_i) logP(F_i = 0|y)]$$

$$= argmax_y logP(y) + \sum_{i=1}^{n} logP(F_i = 0|y) + \sum_{i=1}^{n} f_i log \frac{P(F_i = 1|y)}{P(F_i = 0|y)} \tag{14}$$

Now, the linear classifier rule of a perceptron is

$$argmax_y \sum_{i=0}^{n} w_i f_i \tag{15}$$

where f_0 is a bias term equal to 1.

It can be seen that equation 14 is equivalent to a linear classifier of the same form as equation 15, with weights as follows:

$$w_0 = logP(y) + \sum_{i=1}^{n} logP(F_i = 0|y)$$

$$w_i = \sum_{i=1}^{n} log \frac{P(F_i = 1|y)}{P(F_i = 0|y)} \text{ For } i = 1, ..., n$$