# HW3_pattern recognition

Name: Jordan Wu Section R3 CRN: 69112
Shih-Jie Chang Section Q3 CRN:31425
date:3/30/2019

## 1.1:

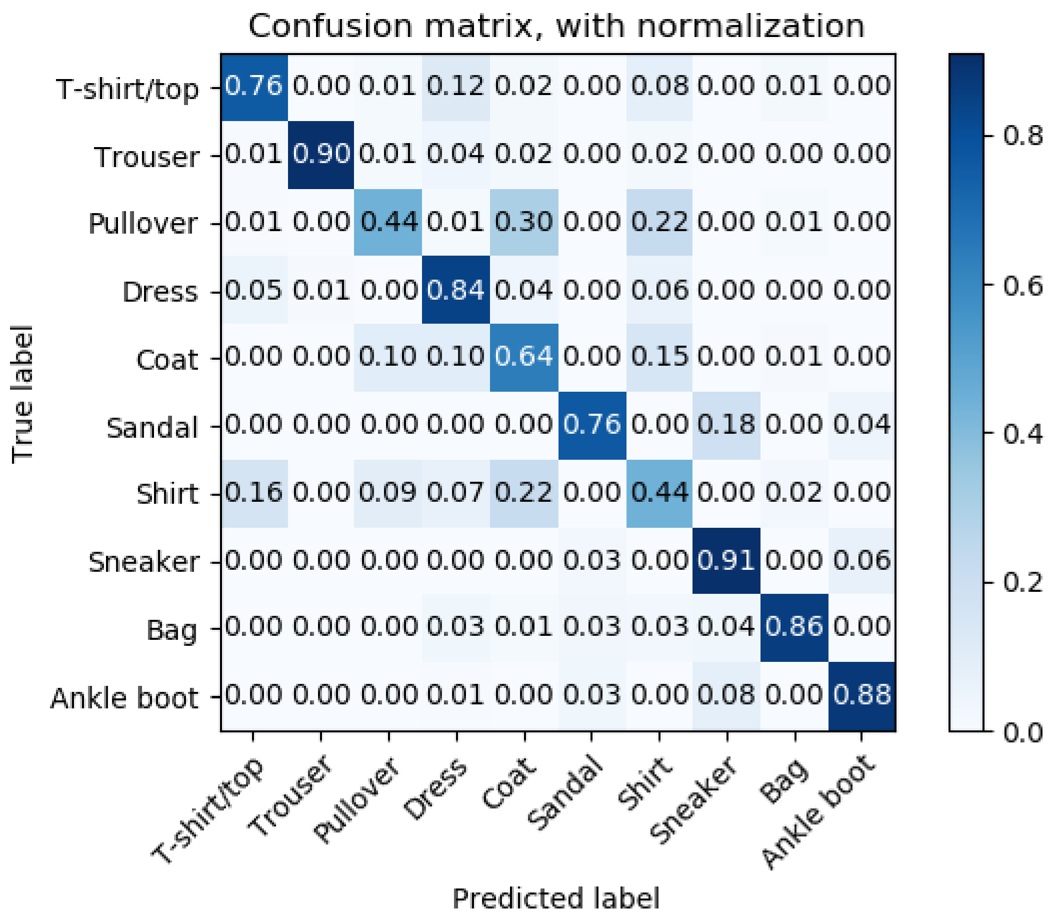
### Avg Classification Rate / Accuracy

0.7435

### Classification Rate Per Item

[0.758, 0.902, 0.442, 0.843, 0.639, 0.764, 0.441, 0.909, 0.857, 0.88]

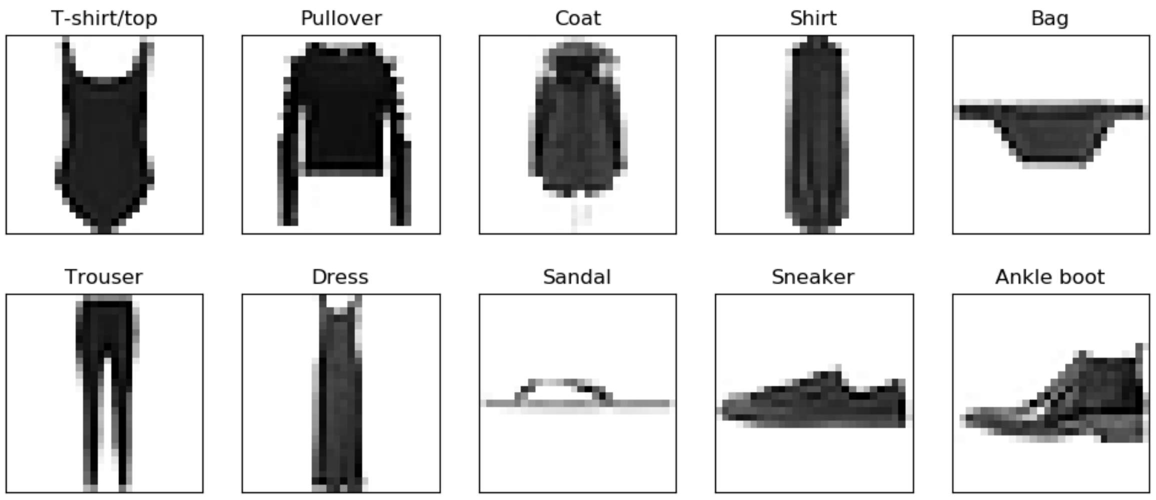
### Confusion Matrix



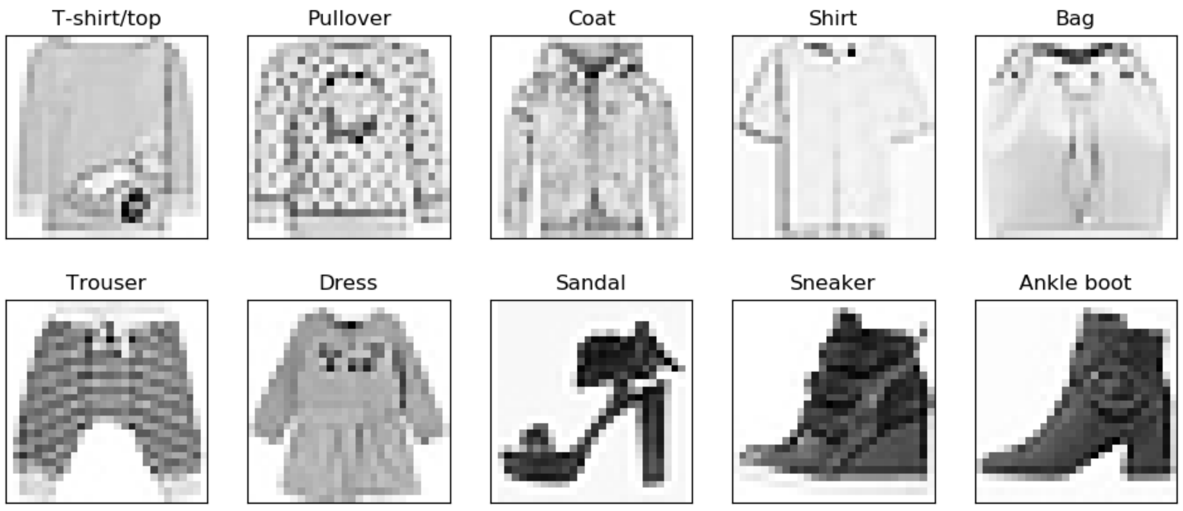
| | T-shirt/top | Trouser | Pullover | Dress | Coat | Sandal | Shirt | Sneaker | Bag | Ankle boot |
|---|---|---|---|---|---|---|---|---|---|---|
| T-shirt/top | 0.76 | 0.00 | 0.01 | 0.12 | 0.02 | 0.00 | 0.08 | 0.00 | 0.01 | 0.00 |
| Trouser | 0.01 | 0.90 | 0.01 | 0.04 | 0.02 | 0.00 | 0.02 | 0.00 | 0.00 | 0.00 |
| Pullover | 0.01 | 0.00 | 0.44 | 0.01 | 0.30 | 0.00 | 0.22 | 0.00 | 0.01 | 0.00 |
| Dress | 0.05 | 0.01 | 0.00 | 0.84 | 0.04 | 0.00 | 0.06 | 0.00 | 0.00 | 0.00 |
| Coat | 0.00 | 0.00 | 0.10 | 0.10 | 0.64 | 0.00 | 0.15 | 0.00 | 0.01 | 0.00 |
| Sandal | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.76 | 0.00 | 0.18 | 0.00 | 0.04 |
| Shirt | 0.16 | 0.00 | 0.09 | 0.07 | 0.22 | 0.00 | 0.44 | 0.00 | 0.02 | 0.00 |
| Sneaker | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.03 | 0.00 | 0.91 | 0.00 | 0.06 |
| Bag | 0.00 | 0.00 | 0.00 | 0.03 | 0.01 | 0.03 | 0.03 | 0.04 | 0.86 | 0.00 |
| Ankle boot | 0.00 | 0.00 | 0.00 | 0.01 | 0.00 | 0.03 | 0.00 | 0.08 | 0.00 | 0.88 |

## Test examples from that class that have the highest and lowest posterior probabilities

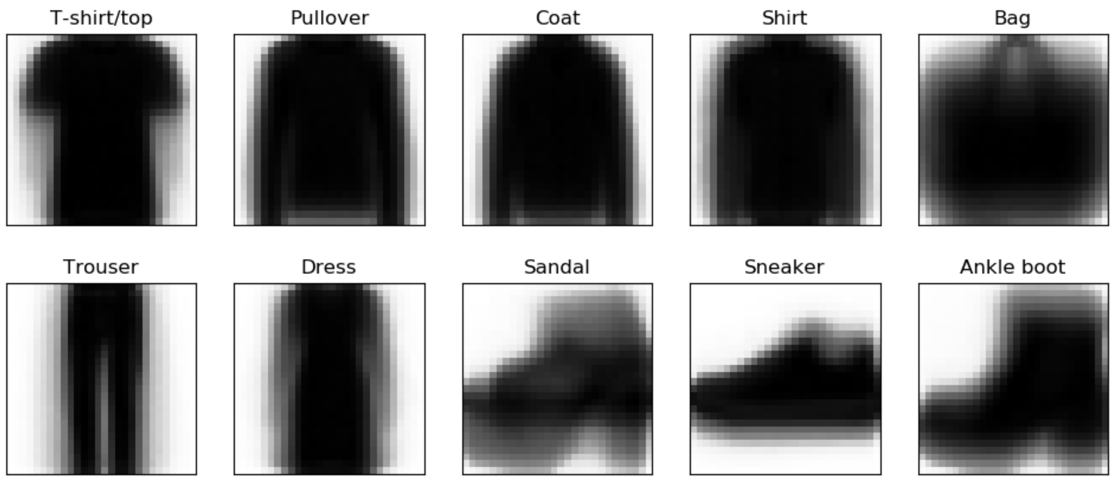
Highest:



Lowest:



## Feature Likelihood Visualization



## Implementation Details

Basic training Procedure Pseudocode:

```
For Each Class (0-10):
    Select out from training set all examples in class as slice
    For each pixelIdx (0-784):
        Count the unique values across the slice at this pixelIdx
        Store Count(feature|class)
```

Doing this slicing procedure allows for a highly vectorized implementation, meaning we never really have to iterate through all 50,000 examples at the “python” level.

The rest of the procedure is pretty obvious, just some divisions.

For example. P(feature|class) = Count(feature|class) / Count(class)

We played around with various laplace smoothing term, and found k = 0.4 to give pretty good accuracy.

For testing, we made use numpy fancy indexing to retrieve the likelihoods for each pixel in the image quickly. So we just had to iterate through the 10 classes.

## 1.2

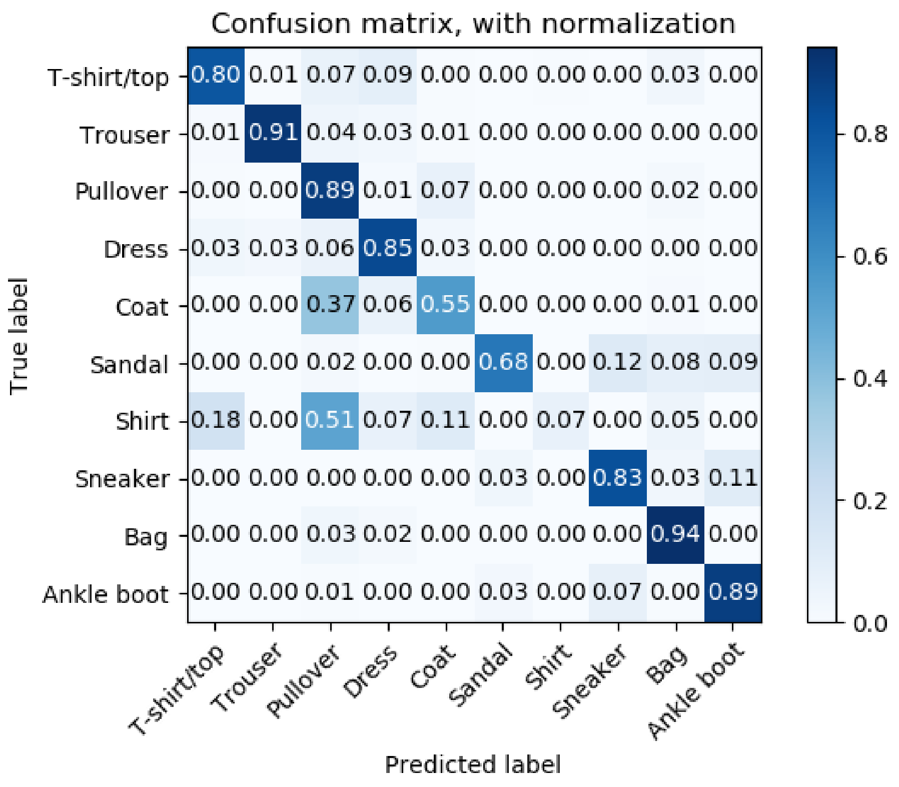
### Avg Accuracy

0.7409

### Classification Rate Per Item

[0.802, 0.912, 0.892, 0.847, 0.548, 0.683, 0.07, 0.827, 0.941, 0.887]

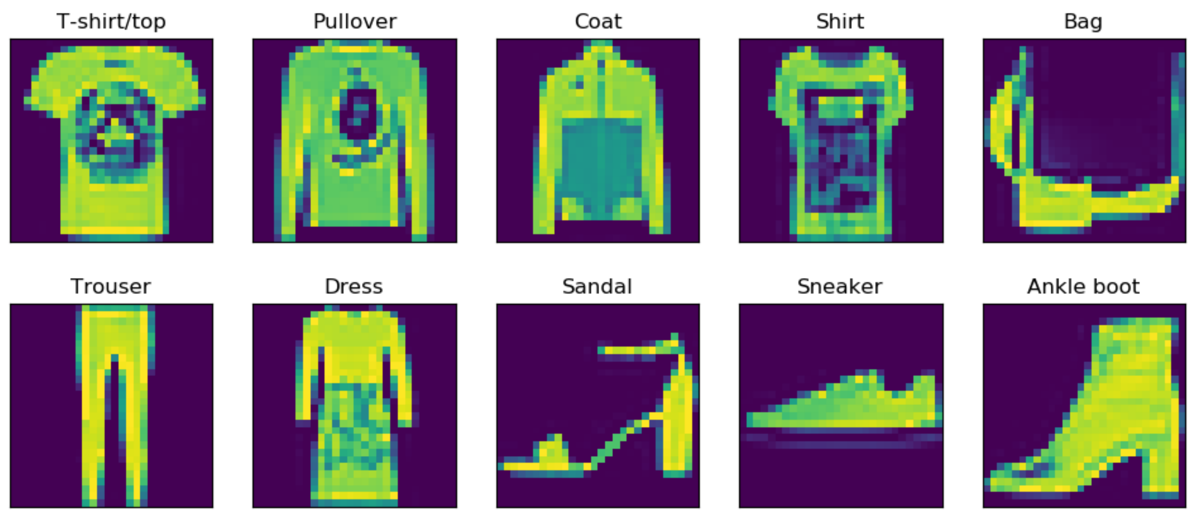
### Confusion Matrix



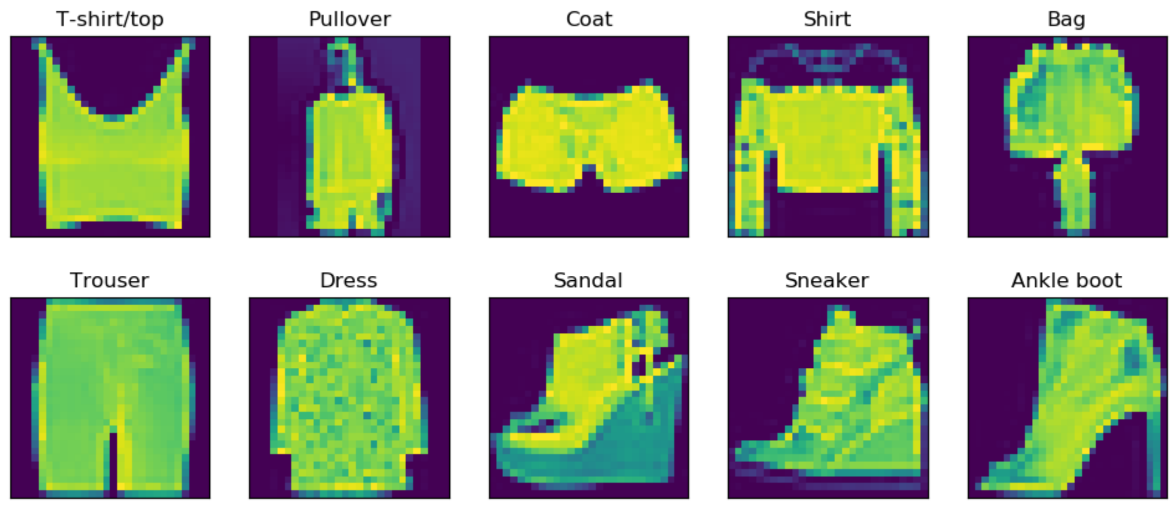
| | T-shirt/top | Trouser | Pullover | Dress | Coat | Sandal | Shirt | Sneaker | Bag | Ankle boot |
|---|---|---|---|---|---|---|---|---|---|---|
| T-shirt/top | 0.80 | 0.01 | 0.07 | 0.09 | 0.00 | 0.00 | 0.00 | 0.00 | 0.03 | 0.00 |
| Trouser | 0.01 | 0.91 | 0.04 | 0.03 | 0.01 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| Pullover | 0.00 | 0.00 | 0.89 | 0.01 | 0.07 | 0.00 | 0.00 | 0.00 | 0.02 | 0.00 |
| Dress | 0.03 | 0.03 | 0.06 | 0.85 | 0.03 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| Coat | 0.00 | 0.00 | 0.37 | 0.06 | 0.55 | 0.00 | 0.00 | 0.00 | 0.01 | 0.00 |
| Sandal | 0.00 | 0.00 | 0.02 | 0.00 | 0.00 | 0.68 | 0.00 | 0.12 | 0.08 | 0.09 |
| Shirt | 0.18 | 0.00 | 0.51 | 0.07 | 0.11 | 0.00 | 0.07 | 0.00 | 0.05 | 0.00 |
| Sneaker | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.03 | 0.00 | 0.83 | 0.03 | 0.11 |
| Bag | 0.00 | 0.00 | 0.03 | 0.02 | 0.00 | 0.00 | 0.00 | 0.00 | 0.94 | 0.00 |
| Ankle boot | 0.00 | 0.00 | 0.01 | 0.00 | 0.00 | 0.03 | 0.00 | 0.07 | 0.00 | 0.89 |

## Test examples from that class that have the highest and lowest perceptron scores

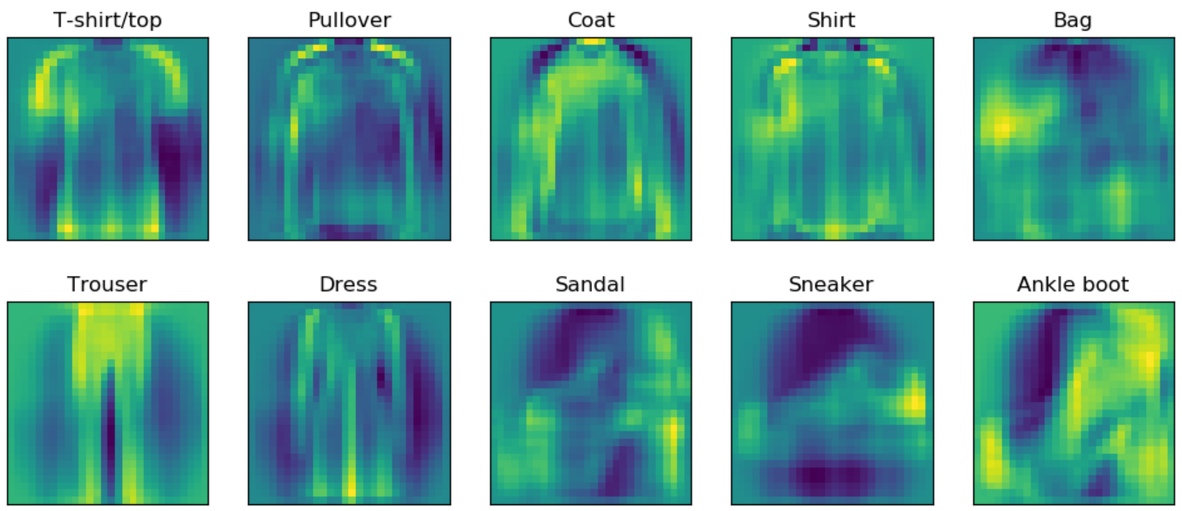
Highest:



Lowest:



## Perceptron Weight Visualization



## Implementation Details:

The implementation is pretty standard, albeit we treated the procedures more like a logistics regression classifier to exploit vectorization:

To predict:
Y_hat (predict) = sgn( train_set x weights )

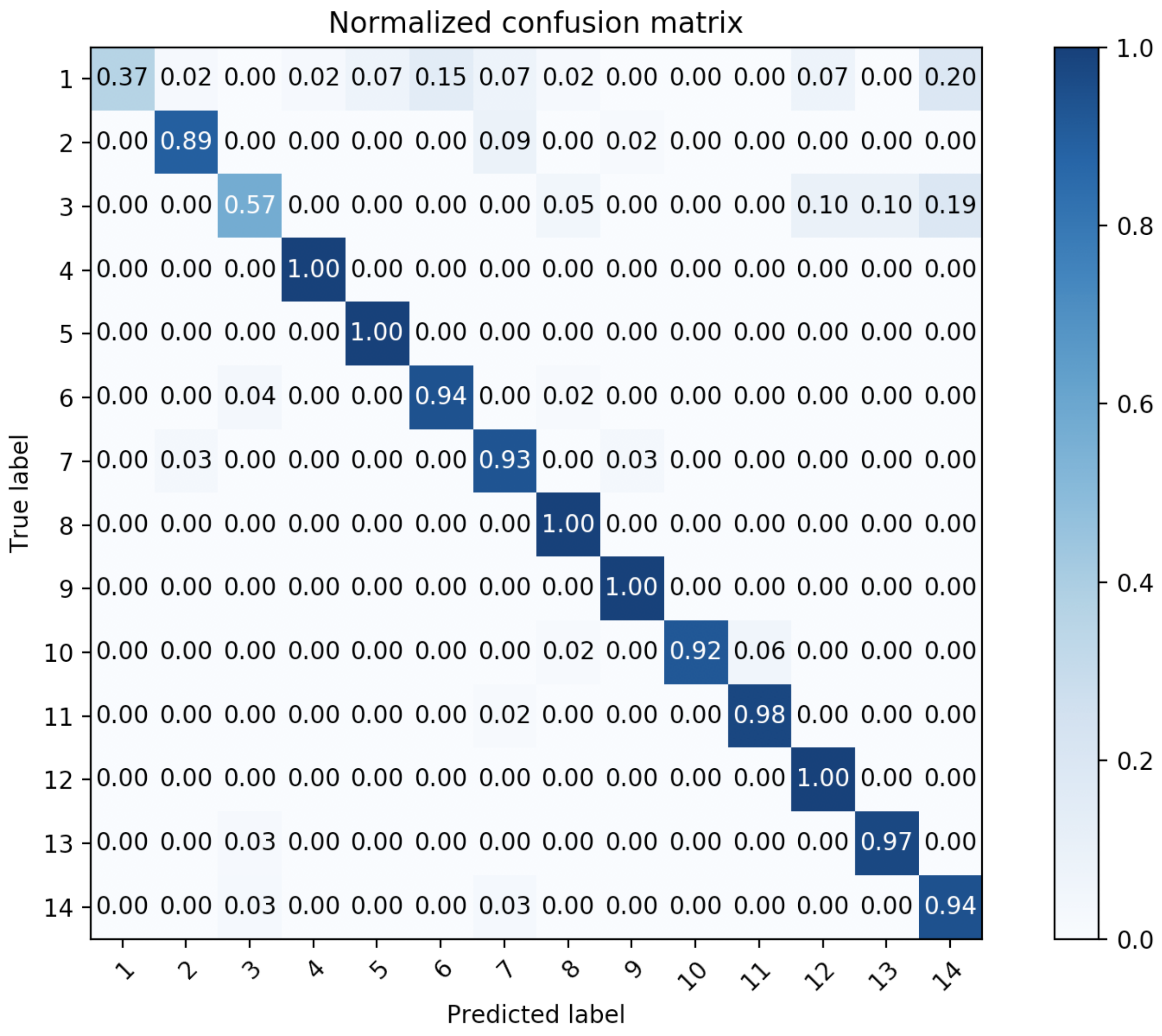
To calculate the "error function", we used constructed a one-hot matrix for Y (given labels), so then we can just subtract the two matrix to find mislabeled examples (Y - Y_hat).

To update weights, the procedure then becomes just:
Weights += train_set^T x (Y - Y_hat) * learning_rate

The learning rate was the most interesting part of our solution. We played around with a constant learning rate, but was unable to dial in a rate that achieves accuracy > .7 while keeping the epochs reasonable (<100) due to time constraints.
We also tried a linearly decreasing rate of 1/epoch, but it also didn't work that well, because the training_rate decreased too quickly before convergence. By the 100th epoch, the rate would be 0.01, which is negligible.
I then came up with a rate of 1/sqrt(epoch), so that by the 100th epoch, the rate is still at 0.1. This ultimately gave the best accuracy. With 200 epochs, we were getting close to 80% accuracy.
However, in the end, we stayed with ~100 epochs due to EWS being too slow and concerns about time limit.

## 2:

### Confusion matrix



| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0.37 | 0.02 | 0.00 | 0.02 | 0.07 | 0.15 | 0.07 | 0.02 | 0.00 | 0.00 | 0.00 | 0.07 | 0.00 | 0.20 |
| 2 | 0.00 | 0.89 | 0.00 | 0.00 | 0.00 | 0.00 | 0.09 | 0.00 | 0.02 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| 3 | 0.00 | 0.00 | 0.57 | 0.00 | 0.00 | 0.00 | 0.00 | 0.05 | 0.00 | 0.00 | 0.00 | 0.10 | 0.10 | 0.19 |
| 4 | 0.00 | 0.00 | 0.00 | 1.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| 5 | 0.00 | 0.00 | 0.00 | 0.00 | 1.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| 6 | 0.00 | 0.00 | 0.04 | 0.00 | 0.00 | 0.94 | 0.00 | 0.02 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| 7 | 0.00 | 0.03 | 0.00 | 0.00 | 0.00 | 0.00 | 0.93 | 0.00 | 0.03 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| 8 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 1.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| 9 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 1.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| 10 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.02 | 0.00 | 0.92 | 0.06 | 0.00 | 0.00 | 0.00 |
| 11 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.02 | 0.00 | 0.00 | 0.00 | 0.98 | 0.00 | 0.00 | 0.00 |
| 12 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 1.00 | 0.00 | 0.00 |
| 13 | 0.00 | 0.00 | 0.03 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.97 | 0.00 |
| 14 | 0.00 | 0.00 | 0.03 | 0.00 | 0.00 | 0.00 | 0.03 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.94 |

### Classification Accuracy for all class (diagonal of confusion matrix):

[0.36585365853658536, 0.8913043478260869, 0.5714285714285714, 1.0, 1.0, 0.9375, 0.9333333333333333, 1.0, 1.0, 0.92, 0.9777777777777777, 1.0, 0.9736842105263158, 0.9428571428571428]

## Precision for all classes

[1.0, 0.9534883720930233, 0.8, 0.9583333333333334, 0.88, 0.8823529411764706, 0.7567567567567568, 0.8947368421052632, 0.8, 1.0, 0.9361702127659575, 0.8936170212765957, 0.9487179487179487, 0.7391304347826086]

## Recall for all classes

[0.36585365853658536, 0.8913043478260869, 0.5714285714285714, 1.0, 1.0, 0.9375, 0.9333333333333333, 1.0, 1.0, 0.92, 0.9777777777777777, 1.0, 0.9736842105263158, 0.9714285714285714]

## F1 Score for all classes

[0.5357142857142857, 0.9213483146067417, 0.6666666666666666, 0.9787234042553191, 0.9361702127659575, 0.9090909090909091, 0.835820895522388, 0.9444444444444444, 0.888888888888889, 0.9583333333333334, 0.9565217391304347, 0.9438202247191011, 0.9610389610389611, 0.8395061728395062]

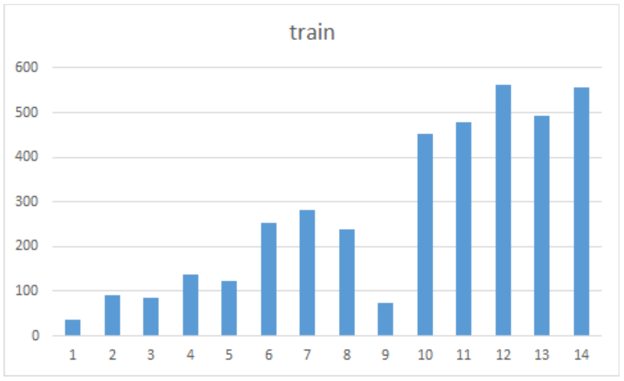
## Top 20 Feature Words for Each Class

1 : ['company', 'based', 'business', 'founded', 'record', 'records', 'bergen', 'systems', 'services', 'products', 'office', 'buses', 'located', 'distribution', 'national', 'health', 'virgin', 'established', 'also', 'regional']
2 : ['school', 'high', 'located', 'university', 'college', 'public', 'schools', 'students', 'education', 'district', 'county', 'founded', 'one', 'new', 'united', 'established', 'independent', 'city', 'part', 'catholic']
3 : ['born', 'american', 'known', 'new', 'band', 'writer', 'best', 'rock', 'musician', 'music', 'work', 'also', 'singer', 'york', 'books', 'author', 'album', 'university', 'series', 'united']
4 : ['born', 'football', 'played', 'league', 'professional', 'player', 'plays', 'footballer', 'former', 'national', 'american', 'also', 'hockey', 'currently', 'rugby', 'team', 'australian', 'november', 'world', 'new']
5 : ['born', 'member', 'district', 'politician', 'state', 'senate', 'democratic', 'house', 'party', 'served', 'former', 'county', 'since', 'representatives', 'republican', 'united', 'elected', 'american', 'representing', 'national']
6 : ['navy', 'built', 'war', 'ship', 'uss', 'united', 'class', 'aircraft', 'world', 'states', 'launched', 'service', 'named', 'first', 'designed', 'royal', 'commissioned', 'american', 'ii', 'us']
7 : ['historic', 'house', 'built', 'located', 'church', 'building', 'national', 'register', 'places', 'listed', 'county', 'street', 'united', 'known', 'museum', 'also', 'states', 'designed', 'added', 'hospital']
8 : ['river', 'lake', 'mountain', 'located', 'south', 'km', 'north', 'county', 'near', 'tributary', 'west', 'range', 'lies', 'creek', 'crater', 'east', 'ft', 'state', 'flows', 'pass']
9 : ['village', 'district', 'population', 'province', 'located', 'census', 'municipality', 'nepal', 'state', 'india', 'county', 'km', 'people', 'within', '2010', '1991', 'south', 'township', 'central', 'southern']
10 : ['family', 'species', 'found', 'genus', 'moth', 'gastropod', 'sea', 'known', 'marine', 'described', 'tropical', 'snail', 'mollusk', 'endemic', 'subtropical', 'habitat', 'natural', 'forests', 'snails', 'moist']
11 : ['species', 'family', 'plant', 'genus', 'native', 'endemic', 'flowering', 'known', 'found', 'common', 'plants', 'leaves', 'habitat', 'tree', 'grows', 'name', 'orchid', 'south', 'bulbophyllum', 'perennial']
12 : ['album', 'released', 'band', 'records', 'first', 'studio', 'american', 'songs', 'music', 'second', 'release', 'recorded', 'rock', 'debut', 'live', 'tracks', 'label', 'albums', 'new', 'ep']
13 : ['film', 'directed', 'starring', 'american', 'stars', 'released', 'written', 'based', 'drama', 'comedy', 'produced', 'also', 'films', 'silent', 'first', 'movie', 'roles', 'novel', 'name', 'documentary']
14 : ['published', 'book', 'novel', 'first', 'journal', 'written', 'series', 'newspaper', 'american', 'story', 'author', 'new', 'magazine', 'fiction', 'books', 'peerreviewed', 'also', 'science', 'publication', 'life']

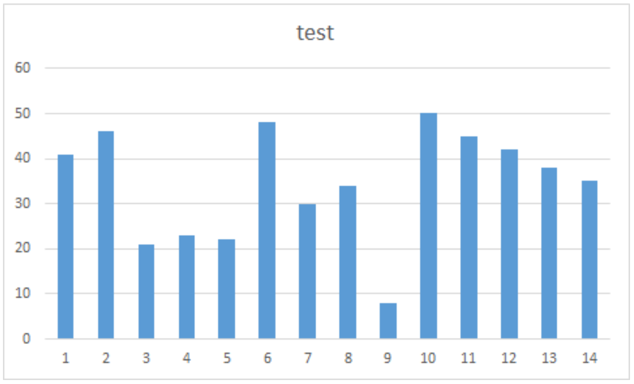
## Prior Changes

Accuracy without uniform distribution in prior (default):.8903
Accuracy with uniform distribution in prior: .8944
Accuracy without priors: .8944

This change may result from the difference between training set and the testing set in the occurrences of each class. If we look at the prior, we can see prior contains the values:
prior: {1: 37, 2: 90, 3: 86, 4: 136, 5: 124, 6: 253, 7: 281, 8: 238, 9: 75, 10: 453, 11: 479, 12: 563, 13: 493, 14: 557}
But if we look at the testing set, the occurrences of each class is:
{1: 41, 2: 46, 3: 21, 4: 23, 5: 22, 6: 48, 7: 30, 8: 34, 9: 8, 10: 50, 11: 45, 12: 42, 13: 38, 14: 35}
The charts are shown below. We can see that the distributions of two sets seem to be irrelevant to each other. Because of this situation, the action of taking prior into account may result in worse accuracy.



.



## Implementation Details

To calculate likelihoods, we used the “One Bit Per Word Token” approach described in class:
P(w|class) = (count(w|class) + smoothing) / (sum of counts of all w’s in class + smoothing * |V|)

To store count(w|class), we used a 2 layer dictionary keyed by class->word->count.

We then summed the log(P(w|class)) + log(P(class)) to find the posterior P(class|w1...wn), then used the MAP decision rule.

We also looked online for more details on how to do smoothing, and we found:
http://www.cs.cornell.edu/courses/cs4740/2014sp/lectures/smoothing+backoff.pdf
to be extremely helpful

We played around with various smoothing terms, and found 0.1 to give the best accuracy

## Extra Credit

We also did the extra credit suggested by implementing a bigram model.

To calculate the bigram likelihoods, we used a “One Bit Per Bigram” approach

P(w2|w1) = (count(w1w2) + smoothing) / (count(w1) + V * smoothing)
Note: w2 = w_k, w1 = w_k-1

Then we just implemented the equation given on the assignment page for P(w1,w2..,wn)

To store count(w1w2), we used the 2 layer dictionary approach, keyed by class->tuple(w1,w2), count(w1) is just the unigram count.

Again, the smoothing term of 0.1 for the bigram model also gave the best accuracy.

Standalone, (mixture = 1.0), the bigram model was able to achieve accuracy = .78

Since the posterior outputted by the bigram model was often orders of magnitude smaller than the unigram’s posterior, we had to use a mixture > .9 for the bigram model to come into play.

We could not find any mixture that achieved better than the unigram.
Though curiously, when mix = 0.99, accuracy = 0.8861
compared to a mix of 0.9, accuracy = 0.8841, which could imply the bigram model is actually boosting the accuracy.

Responses to questions posed:

1) By relaxing the naive assumptions of the model, we are saying that adjacent words are no longer independent of each other. This in turn increases the feature space of our model, where adjacency is also a feature now. Increasing the feature space allows us to decrease the bias, but could also lead to higher variance and overfitting.
2) When N is a really large number, then the probability of a sequence in the test set being out-of-vocabulary (where a feature was never seen in the train set) also goes up. This requires special consideration and modeling, which leads to further complexity. In addition, as stated previously, larger N also leads to high variance and overfitting.

## Statement of Contribution

Jordan did all the code.

Jordan did Section I and Extra Credit of the report. Shih-Jie did section II of the report.