# CS 224N: Assignment 1

Ryan McMahon Monday 6th February, 2017

## Contents

## Problem 1: Softmax (10 pts)

### 1.1 (a) Softmax Invariance to Constant (5 pts)

*Prove that softmax is invariant to constant offsets in the input, that is, for any input vector $x$ and any constant $c$, softmax($x$) = softmax($x + c$), where $x + c$ means adding the constant $c$ to every dimension of $x$. Remember that*

$$softmax(x)_i = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{1.1}$$

**Answer:**

We can show that softmax($x$) = softmax($x + c$) by factoring out $c$ and canceling:

$$\begin{aligned} softmax(x+c)_i &= \frac{e^{x_i+c}}{\sum_j e^{x_j+c}} = \frac{e^{x_i} \times e^c}{e^c \times \sum_j e^{x_j}} \\ &= \frac{e^{x_i} \times \cancel{e^c}}{\cancel{e^c} \times \sum_j e^{x_j}} = softmax(x)_i \end{aligned}$$

### 1.2 (b) Softmax Coding (5 pts)

*Given an input matrix of $N$ rows and $D$ columns, compute the softmax prediction for each row using the optimization in part (a). Write your implementation in* `q1_softmax.py`*. You may test by executing* `python q1_softmax.py`.

*Note: The provided tests are not exhaustive. Later parts of the assignment will reference this code so it is important to have a correct implementation. Your implementation should also be efficient and vectorized whenever possible (i.e., use numpy matrix operations rather than for loops). A non-vectorized implementation will not receive full credit!*

**Answer:**

See code: ~`/code/q1_softmax.py`.

# Problem 2: Neural Network Basics (30 pts)

## 2.1 (a) Sigmoid Gradient (3 pts)

*Derive the gradients of the sigmoid function and show that it can be rewritten as a function of the function value (i.e., in some expression where only (x), but not x, is present). Assume that the input x is a scalar for this question. Recall, the sigmoid function is*

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{2.1}$$

**Answer:**

$$\begin{aligned}
\sigma(x) &= \frac{1}{1+e^{-x}} \\
&= \frac{e^x}{1+e^x} \\
\frac{\partial}{\partial x}\sigma(x) &= \frac{e^x \times (1+e^x) - (e^x \times e^x)}{(1+e^x)^2} \\
&= \frac{e^x + \cancel{(e^x \times e^x)} - \cancel{(e^x \times e^x)}}{(1+e^x)^2} \\
&= \frac{e^x}{(1+e^x)^2} = \sigma(x) \times (1-\sigma(x))
\end{aligned}$$

Because $1-\sigma(x) = \sigma(-x)$ we can show that:

$$\begin{aligned}
\frac{\partial}{\partial x}\sigma(x) &= \frac{e^x}{(1+e^x)^2} \\
&= \sigma(x) \times \sigma(-x) \\
&= \frac{e^x}{1+e^x} \times \frac{1}{1+e^{+x}} \\
&= \frac{e^x}{(1+e^x)^2}
\end{aligned}$$

## 2.2 (b) Softmax Gradient w/ Cross Entropy Loss (3 pts)

*Derive the gradient with regard to the inputs of a softmax function when cross entropy loss is used for evaluation, i.e., find the gradients with respect to the softmax input vector* $\boldsymbol{\theta}$*, when the prediction is made by* $\hat{\mathbf{y}} = softmax(\boldsymbol{\theta})$*. Remember the cross entropy function is*

$$CE(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_i y_i \times log(\hat{y}_i) \tag{2.2}$$

*where* $\mathbf{y}$ *is the one-hot label vector, and* $\hat{\mathbf{y}}$ *is the predicted probability vector for all classes. (Hint: you might want to consider the fact many elements of* $\mathbf{y}$ *are zeros, and assume that only the* $k-th$ *dimension of* $\mathbf{y}$ *is one.)*

**Answer:**

Let $S$ represent the softmax function:

$$\begin{aligned}
f_i &= e^{\theta_i} \\
g_i &= \sum_{k=1}^{K} e^{\theta_k} \\
S_i &= \frac{f_i}{g_i} \\
\frac{\partial S_i}{\partial \theta_j} &= \frac{f_i' g_i - g_i' f_i}{g_i^2}
\end{aligned}$$

So if $i = j$:

$$\begin{aligned}
f_i' &= f_i; \;\; g_i' = e^{\theta_j} \\
\frac{\partial S_i}{\partial \theta_j} &= \frac{e^{\theta_i} \sum_k e^{\theta_k} - e^{\theta_j} e^{\theta_i}}{(\sum_k e^{\theta_k})^2} \\
&= \frac{e^{\theta_i}}{\sum_k e^{\theta_k}} \times \frac{\sum_k e^{\theta_k} - e^{\theta_j}}{\sum_k e^{\theta_k}} \\
&= S_i \times (1 - S_i)
\end{aligned}$$

And if $i \neq j$:

$$\begin{aligned}
\frac{\partial S_i}{\partial \theta_j} &= \frac{0 - e^{\theta_j} e^{\theta_i}}{(\sum_k e^{\theta_k})^2} \\
&= -\frac{e^{\theta_j}}{\sum_k e^{\theta_k}} \times \frac{e^{\theta_i}}{\sum_k e^{\theta_k}} \\
&= -S_j \times S_i
\end{aligned}$$

We can now use these when operating on our loss function (let $L$ represent the cross entropy function):

$$\begin{aligned}
\frac{\partial L}{\partial \theta_i} &= -\sum_k y_k \frac{\partial log S_k}{\partial \theta_i} \\
&= -\sum_k y_k \frac{1}{S_k} \frac{\partial S_k}{\partial \theta_i} \\
&= -y_i(1 - S_i) - \sum_{k \neq i} y_k \frac{1}{S_k}(-S_k \times S_i) \\
&= -y_i(1 - S_i) + \sum_{k \neq i} y_k S_i \\
&= -y_i + y_i S_i + \sum_{k \neq i} y_k S_i \\
&= S_i(\sum_k y_k) - y_i
\end{aligned}$$

And because we know that $\sum_k y_k = 1$:

$$\frac{\partial L}{\partial \theta_i} = S_i - y_i$$

## 2.3 (c) One Hidden Layer Gradient (6 pts)

*Derive the gradients with respect to the inputs $\boldsymbol{x}$ to a one-hidden-layer neural network (that is, find $\frac{\partial J}{\partial \boldsymbol{x}}$ where $J = CE(\mathbf{y}, \hat{\mathbf{y}})$ is the cost function for the neural network). The neural network employs sigmoid activation function for the hidden layer, and softmax for the output layer. Assume the one-hot label vector is $\mathbf{y}$, and cross entropy cost is used. (Feel free to use $\sigma'(x)$ as the shorthand for sigmoid gradient, and feel free to define any variables whenever you see fit.)*

*Recall that forward propoagation is as follows*

$$\mathbf{h} = sigmoid(\boldsymbol{x}\boldsymbol{W}_1 + \boldsymbol{b}_1) \qquad \hat{\boldsymbol{y}} = softmax(\boldsymbol{h}\boldsymbol{W}_2 + \boldsymbol{b}_2)$$

**Answer:**

Let $f_2 = \boldsymbol{x}\boldsymbol{W}_1 + \boldsymbol{b}_1$ and $f_3 = \boldsymbol{h}\boldsymbol{W}_2 + \boldsymbol{b}_2$;

$$\begin{aligned}
\frac{\partial J}{\partial f_3} &= \boldsymbol{\delta}_3 = \hat{\boldsymbol{y}} - \boldsymbol{y} \\
\frac{\partial J}{\partial \boldsymbol{h}} &= \boldsymbol{\delta}_2 = \boldsymbol{\delta}_3 \boldsymbol{W}_2^T \\
\frac{\partial J}{\partial f_2} &= \boldsymbol{\delta}_1 = \boldsymbol{\delta}_2 \circ \sigma'(f_2) \\
\frac{\partial J}{\partial \boldsymbol{x}} &= \boldsymbol{\delta}_1 \frac{\partial f_2}{\partial \boldsymbol{x}} \\
&= \boldsymbol{\delta}_1 \boldsymbol{W}_1^T
\end{aligned}$$

## 2.4 (d) No. Parameters (2 pts)

*How many parameters are there in this neural network* [from (**c**) above], *assuming the input is $D_x-$dimensional, the output is $D_y-$dimensional, and there are $H$ hidden units?*

**Answer:**

$$\begin{aligned}
n_{W_1} &= D_x \times H \\
n_{b_1} &= H \\
n_{W_2} &= H \times D_y \\
n_{b_2} &= D_y \\
N &= (D_x \times H) + H + (H \times D_y) + D_y
\end{aligned}$$

## 2.5 (e) Sigmoid Activation Code (4 pts)

*Fill in the implementation for the sigmoid activation function and its gradient in* `q2_sigmoid.py`*. Test your implementation using* `python q2_sigmoid.py`*. Again, thoroughly test your code as the provided tests may not be exhaustive.*

**Answer:**

See code: ~`/code/q2_sigmoid.py`.

## 2.6 (f) Gradient Check Code (4 pts)

*To make debugging easier, we will now implement a gradient checker. Fill in the implementation for* `gradcheck_naive` *in* `q2_gradcheck.py`*. Test your code using* `python q2_gradcheck.py`.

**Answer:**

See code: ~`/code/q2_gradcheck.py`.

## 2.7 (g) Neural Net Code (8 pts)

*Now, implement the forward and backward passes for a neural network with one sigmoid hidden layer. Fill in your implementation in* `q2_neural.py`*. Sanity check your implementation with* `python q2_neural.py`.

**Answer:**

See code: ~`/code/q2_neural.py`.

## Problem 3: Word2Vec (40 pts + 2 bonus)

### 3.1 (a) Context Word Gradients (3 pts)

*Assume you are given a predicted word vector $\boldsymbol{v}_c$ corresponding to the center word $\boldsymbol{c}$ for skipgram, and word prediction is made with the softmax function found in word2vec models*

$$\hat{\boldsymbol{y}}_o = p(\boldsymbol{o}|\boldsymbol{c}) = \frac{exp(\boldsymbol{u}_0^T \boldsymbol{v}_c)}{\sum\limits_{w=1}^{W} exp(\boldsymbol{u}_w^T \boldsymbol{v}_c)} \tag{3.1}$$

*where $\boldsymbol{w}$ denotes the w-th word and $\boldsymbol{u}_w$ ($w = 1, ..., W$) are the "output" word vectors for all words in the vocabulary. Assume cross entropy cost is applied to this prediction and word $\boldsymbol{o}$ is the expected word (the $\boldsymbol{o}$-th element of the one-hot label vector is one), derive the gradients with respect to $\boldsymbol{v}_c$,*

Hint: It will be helpful to use notation from question 2. For instance, letting $\hat{\boldsymbol{y}}$ be the vector of softmax predictions for every word, $\boldsymbol{y}$ as the expected word vector, and the loss function

$$J_{softmax-CE}(\boldsymbol{o}, \boldsymbol{v}_c, \boldsymbol{U}) = CE(\boldsymbol{y}, \hat{\boldsymbol{y}}) \tag{3.2}$$

*where $\boldsymbol{U}$ = [$\boldsymbol{u}_1$,$\boldsymbol{u}_1$,..., $\boldsymbol{u}_W$] is the matrix of all the output vectors. Make sure you state the orientation of your vectors and matrices.*

**Answer:**

From Problem 2.2 we know that $\frac{\partial J}{\partial \boldsymbol{\theta}} = (\hat{\boldsymbol{y}} - \boldsymbol{y})$. Given that, let $\boldsymbol{theta} = \boldsymbol{v}_c$. Then

$$\frac{\partial J}{\partial \boldsymbol{\theta}} = \boldsymbol{U}^T(\hat{\boldsymbol{y}} - \boldsymbol{y})$$

## 3.2 (b) Output Word Gradients (3 pts)

*As in the previous part, derive gradients for the output word vectors $\boldsymbol{u}_w$ 's (including $\boldsymbol{u}_o$).*

**Answer:**

Here we're going to do essentially the same thing, but instead transpose the error. So, from above and Problem 2.2, let $\boldsymbol{\theta} = \boldsymbol{U}$

$$\frac{\partial J}{\partial \boldsymbol{\theta}} = \boldsymbol{v}_c(\hat{\boldsymbol{y}} - \boldsymbol{y})^T$$

## 3.3 (c) Repeat Gradients with Negative Sampling Loss (6 pts)

*Repeat part **(a)** and **(b)** assuming we are using the negative sampling loss for the predicted vector $\boldsymbol{v}_c$, and the expected output word is $\boldsymbol{o}$. Assume that $\boldsymbol{K}$ negative samples (words) are drawn, and they are $\mathbf{1}, \ldots, \boldsymbol{K}$, respectively for simplicity of notation ($o \notin \{1, ..., K\}$). Again, for a given word, $o$,denote its output vector as $\boldsymbol{u}_o$. The negative sampling loss function in this case is*

$$J_{neg-sample}(\boldsymbol{o}, \boldsymbol{v}_c, \boldsymbol{U}) = log(\sigma(\boldsymbol{u}_o^T \boldsymbol{v}_c)) - \sum_{k=1}^{K} log(\sigma(-\boldsymbol{u}_k^T \boldsymbol{v}_c)) \tag{3.3}$$

where $\sigma(\cdot)$ is the sigmoid function.

*After youve done this, describe with one sentence why this cost function is much more efficient to compute than the softmax-CE loss (you could provide a speed-up ratio, i.e. the runtime of the softmax-CE loss divided by the runtime of the negative sampling loss).*

**Answer:**

Let $z_j = \boldsymbol{u}_j^T \boldsymbol{v}_c$:

$$\frac{\partial J}{\partial z_j} = \begin{cases} \sigma(\boldsymbol{u}_j^T \boldsymbol{v}_c) - 1 & \text{if } j = o \\ \sigma(\boldsymbol{u}_j^T \boldsymbol{v}_c) & \text{if } j \in \boldsymbol{K} \end{cases}$$

Then we can separate out the partials for $\boldsymbol{u}_j$ and $\boldsymbol{v}_c$.

$$\begin{aligned} \frac{\partial J}{\partial \boldsymbol{u}_o} &= \frac{\partial J}{\partial z_j} \times \frac{\partial z_j}{\partial \boldsymbol{u}_o} \\ &= (\sigma(\boldsymbol{u}_o^T \boldsymbol{v}_c) - 1)\boldsymbol{v}_c \\ \frac{\partial J}{\partial \boldsymbol{u}_k} &= \frac{\partial J}{\partial z_j} \times \frac{z_j}{\partial \boldsymbol{u}_k} \\ &= -(\sigma(-\boldsymbol{u}_k^T \boldsymbol{v}_c) - 1)\boldsymbol{v}_c \text{ for all } k \in \boldsymbol{K} \end{aligned}$$

$$\begin{aligned}\frac{\partial J}{\partial \boldsymbol{v}_c} &= \frac{\partial J}{\partial z_j} \times \frac{\partial z_j}{\partial \boldsymbol{v}_c} \\ &= (\sigma(\boldsymbol{u}_o^T \boldsymbol{v}_c) - 1)\boldsymbol{u}_o - \sum_{k=1}^{K} (\sigma(-\boldsymbol{u}_k^T \boldsymbol{v}_c) - 1)\boldsymbol{u}_k\end{aligned}$$

This is faster than the original cross entropy loss because we are no longer deriving the gradients $\forall w_j \in W$. Instead, we are only evaluating the gradients for $[w_o, w_k, \dots, w_K]$.

## 3.4 (d) Skip-gram and CBOW Gradients (8 pts)

*Derive gradients for all of the word vectors for skip-gram and CBOW given the previous parts and given a set of context words [word$_{cm}$ ,...,word$_{c1}$ ,word$_{c}$ ,word$_{c+1}$ ,...,word$_{c+m}$ ], where $m$ is the context size. Denote the input and output word vectors for word $k$ as $v_k$ and $u_k$ respectively.*

*Hint: Feel free to use $F(\boldsymbol{o}, \boldsymbol{v}_c)$ (where $\boldsymbol{o}$ is the expected word) as a placeholder for the $J_{softmax-CE}(\boldsymbol{o}, \boldsymbol{v}_c, \dots)$ or $J_{neg-sample}(\boldsymbol{o}, \boldsymbol{v}_c, \dots)$ cost functions in this part – you'll see that this is a useful abstraction for the coding part. That is, your solution may contain terms of the form $\frac{\partial F(\boldsymbol{o}, \boldsymbol{v}_c)}{\partial \dots}$.*
*Recall that for skip-gram, the cost for a context centered around $c$ is*

$$J_{skip-gram}(word_{c-m \dots c+m}) = \sum_{-m \leqslant j \leqslant m, j \neq 0} F(\boldsymbol{w}_{c+j}, \boldsymbol{v}_c) \tag{3.4}$$

*where $\boldsymbol{w}_{c+j}$ refers to the word at the $j-th$ index from the center.*

*CBOW is slightly different. Instead of using $\boldsymbol{v}_c$ as the predicted vector, we use $\hat{\boldsymbol{v}}$ defined below. For (a simpler variant of) CBOW, we sum up the input word vectors in the context*

$$\hat{\boldsymbol{v}} = \sum_{-m \leqslant j \leqslant m, j \neq 0} \boldsymbol{v}_{c+j} \tag{3.5}$$

then the CBOW cost is

$$J_{CBOW}(word_{c-m \dots c+m}) = F(\boldsymbol{w}_c, \hat{\boldsymbol{v}}) \tag{3.6}$$

*Note: To be consistent with the $\hat{\boldsymbol{v}}$ notation such as for the code portion, for skip-gram $\hat{\boldsymbol{v}} = \boldsymbol{v}_c$.*

**Answer:**

For the skip-gram model, we can show that (for a given context window) the gradients are equal to:

$$\begin{aligned}\frac{J_{skip-gram}(word_{c-m \dots c+m})}{\partial \boldsymbol{U}} &= \sum_{-m \leqslant j \leqslant m, j \neq 0} \frac{\partial F(\boldsymbol{w}_{c+j}, \boldsymbol{v}_c)}{\partial \boldsymbol{U}} \\ \frac{J_{skip-gram}(word_{c-m \dots c+m})}{\partial \boldsymbol{v}_c} &= \sum_{-m \leqslant j \leqslant m, j \neq 0} \frac{\partial F(\boldsymbol{w}_{c+j}, \boldsymbol{v}_c)}{\partial \boldsymbol{v}_c} \\ \frac{J_{skip-gram}(word_{c-m \dots c+m})}{\partial \boldsymbol{v}_j} &= 0, \forall j \neq c\end{aligned}$$

For the CBOW model, alternatively, the gradients are equal to:

$$
\begin{aligned}
\frac{J_{CBOW}(word_{c-m \dots c+m})}{\partial \boldsymbol{U}} &= \frac{\partial F(\boldsymbol{w}_c, \hat{\boldsymbol{v}})}{\partial \boldsymbol{U}} \\
\frac{J_{CBOW}(word_{c-m \dots c+m})}{\partial \boldsymbol{v}_j} &= \frac{\partial F(\boldsymbol{w}_c, \hat{\boldsymbol{v}})}{\partial \hat{\boldsymbol{v}}}, \forall (j \neq c) \in \{c - m \dots c + m\} \\
\frac{J_{CBOW}(word_{c-m \dots c+m})}{\partial \boldsymbol{v}_j} &= 0, \forall (j \neq c) \notin \{c - m \dots c + m\}
\end{aligned}
$$

The qualifying indexes for the latter two equations are essentially saying "where $j$ is in the range $\{c - m \dots c + m\}$, if the middle term $(c - 0)$ is removed"; and then the same thing, but "where $j$ is not in that range".

## 3.5 (e) Word2Vec with SGD Code (12 pts)

*In this part you will implement the word2vec models and train your own word vectors with stochastic gradient descent (SGD). First, write a helper function to normalize rows of a matrix in* `q3_word2vec.py`. *In the same file, fill in the implementation for the softmax and negative sampling cost and gradient functions. Then, fill in the implementation of the cost and gradient functions for the skip-gram model. When you are done, test your implementation by running* `python q3_word2vec.py`. *Note: If you choose not to implement CBOW (part h), simply remove the* `NotImplementedError` *so that your tests will complete.*

**Answer:**

See code: ~`/code/q3_word2vec.py`.

## 3.6 (f) SGD Code (4 pts)

*Complete the implementation for your SGD optimizer in* `q3_sgd.py`. *Test your implementation by running* `python q3_sgd.py`.

**Answer:**

See code: ~`/code/q3_sgd.py`.

## 3.7 (g) Train Vectors on Stanford Sentiment Treebank (4 pts)

*Show time! Now we are going to load some real data and train word vectors with everything you just implemented! We are going to use the Stanford Sentiment Treebank (SST) dataset to train word vectors, and later apply them to a simple sentiment analysis task. You will need to fetch the datasets first. To do this, run* `sh get_datasets.sh`. *Test your implementation by running* `python q3_sgd.py`. *There is no additional code to write for this part; just run* `python q3_run.py`

**Answer:**

See code: ∼`/code/q3_sgd.py`.

## 3.8 (h) CBOW (Extra Credit: 2 pts)

*Implement the CBOW model in* `q3_word2vec.py`. *Note: this part is optional but the gradient derivations for CBOW in part (d) are not!*

**Answer:**

Not implemented (currently).

## Problem 4: Sentiment Analysis (20 pts)

*Now, with the word vectors you trained, we are going to perform a simple sentiment analysis. For each sentence in the Stanford Sentiment Treebank dataset, we are going to use the average of all the word vectors in that sentence as its feature, and try to predict the sentiment level of the said sentence. The sentiment level of the phrases are represented as real values in the original dataset, here we'll just use five classes:*

"very negative" ($--$); "negative" ($-$); "neutral"; "positive" ($+$); "very positive" ($++$)

*which are represented by 0 to 4 in the code, respectively. For this part, you will learn to train a softmax classifier, and perform train/dev validation to improve generalization.*

### 4.1 (a) Sentence Features (2 pts)

*Implement a sentence featurizer. A simple way of representing a sentence is taking the average of the vectors of the words in the sentence. Fill in the implementation in* `q4_sentiment.py`.

**Answer:**

See code: $\sim$`/code/q4_sentiment.py`.

### 4.2 (b) Purpose of Regularization (1 pt)

*Explain in at most two sentences why we want to introduce regularization when doing classification (in fact, most machine learning tasks).*

**Answer:**

Regularization serves the purpose of increasing the generalizability of a trained model to new data. Constraining our parameter estimates using a(set of) penalty(penalties) helps mitigate issues related to over-fitting.

### 4.3 (c) Purpose of Regularization (1 pt)

*Fill in the hyperparameter selection code in* `q4_sentiment.py` *to search for the optimal regularization parameter. Attach your code for* `chooseBestModel` *to your written write-up. You should be able to attain at least 36.5% accuracy on the dev and test sets using the pre-trained vectors in part (d).*

**Answer:**

See code: ~`/code/q4_sentiment.py`.

### 4.4 (d) Pretrained vs New Vectors (3 pts)

*Run* `python q4_sentiment.py --yourvectors` *to train a model using your word vectors from q3. Now, run* `python q4_sentiment.py --pretrained` *to train a model using pretrained GloVe vectors (on Wikipedia data). Compare and report the best train, dev, and test accuracies. Why do you think the pretrained vectors did better? Be specific and justify with 3 distinct reasons.*

**Answer:**

*Note: Both tables rely on a "One-versus-Rest" logistic classifier. This is the default assignment code.*

With our newly trained vectors, our best out-of-sample accuracy is roughly 29.3%: for the fine-grained Stanford Sentiment Treebank dataset.

Table 4.1: Sentiment Accuracy – New Vectors

| **Inverse Regularization** | **Accuracy** | | |
|---:|---|---|---|
| **Strength** (L2) | *Train* | *Dev* | *Test* |
| 25 | **30.314** | **30.972** | **29.367** |
| 16 | 30.302 | **30.972** | 28.869 |
| 9 | 30.22 | 30.79 | 28.597 |
| 4 | 30.091 | 30.518 | 27.511 |
| 1 | 28.874 | 27.611 | 25.973 |

In comparison, when we move to using the pretrained GloVe vectors, our best out-of-sample accuracy jumps to approximately 37.5%. This constitutes a tremendous improvement over our newly trained vectors. Though, if this were done properly (i.e., not running on the test set each time) we would have concluded that the inverse regularization strength of 16 (row 2 in Table 4.2) was best – not 25.

Table 4.2: Sentiment Accuracy – Pretrained GloVe

| **Inverse Regularization** | **Accuracy** | | |
|---:|---|---|---|
| **Strength** (L2) | *Train* | *Dev* | *Test* |
| 25 | 39.817 | **36.331** | **37.511** |
| 16 | **39.864** | **36.331** | 37.376 |
| 9 | 39.747 | 36.240 | 37.195 |
| 4 | 39.654 | 36.603 | 37.195 |
| 1 | 39.525 | 36.603 | 37.330 |

There are lots of potential reasons for the GloVe vectors to outperform our newly trained ones. One is that we simply were training our vectors on (far) less data than was used to generate the GloVe vectors. Additionally, while our vectors are only 10-dimensional, the pretrained vectors are $D = 50$. This *should* result in "better" embeddings. And finally, the classifier gets 50 features with the pretrained vectors, but only 10 (less informative) features with our newly generated measurements.