CIS 520 Project: Sentiment Analysis

The SVMen:

Gil Landau Ryan Hub Jakob Heckelmann

Background:

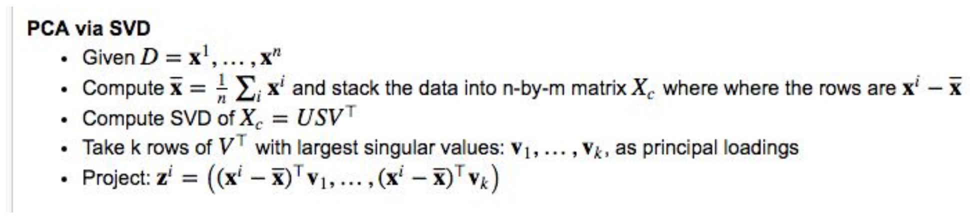
The goal of this assignment was to correctly label the sentiment in a set of tweets. Each tweet was given one of five emotions: joy, sadness, surprise, fear, or anger. The data was comprised of a labelled training set of roughly 18,000 tweets and an unlabelled validation set of about 9,000 tweets. Each tweet had an associated 10,000 word-frequency vector and the raw text was readily available for further feature extraction. The model accuracy is measured by average cost of a prediction, because different predictions have different costs and the average costs reflects that better than a more traditional precision ranking. The general strategy was to find individual methods and ideas that worked well in practice and to create a ensemble that optimized the weight of each method. Using this strategy, we were able to reach **4th place** on the final test rankings.

Preprocessing:

For this project we had a total of 10,000 features, which would take many machine learning methods too much time and too much memory to use practically. Therefore, we tried to use two different methods to decrease the number of features. However, in the end, we did not use any dimensionality reduction in our final model.

Dimensionality Reduction: PCA

The first way was to conduct PCA via SVD and more specifically svds in Matlab because we had a sparse matrix. Per the course wiki notes we used the following algorithm.


PCA via SVD

- Given $D = \mathbf{x}^1, \ldots, \mathbf{x}^n$
- Compute $\bar{\mathbf{x}} = \frac{1}{n}\sum_i \mathbf{x}^i$ and stack the data into n-by-m matrix X_c where where the rows are $\mathbf{x}^i - \bar{\mathbf{x}}$
- Compute SVD of $X_c = USV^\top$
- Take k rows of $V^\top$ with largest singular values: $\mathbf{v}_1, \ldots, \mathbf{v}_k$, as principal loadings
- Project: $\mathbf{z}^i = \left((\mathbf{x}^i - \bar{\mathbf{x}})^\top \mathbf{v}_1, \ldots, (\mathbf{x}^i - \bar{\mathbf{x}})^\top \mathbf{v}_k\right)$

We calculated V on the training data and saved it to a V.mat file, in order to apply it on the test data. A problem we noticed here is the V.mat file takes up a lot of memory. For example, the top 400 features would create a file of 30 MB; this became burdensome because our models would be close to 20 MB without the PCA data, and it became difficult to stay below the memory limit. Ultimately, we saw no better performance when using the PCA data.

Dimensionality Reduction: Feature Selection

The other method we used was to improve the feature selection. We removed stop words, as well as words that appeared infrequently. We preferred this way to PCA via SVD because it took less time and memory to generate and use.. The drawback of this method was that it could possibly remove a word that would be helpful in distinguishing the sentiment of a given tweet.

Base Methods:

K-Nearest Neighbors

For our instance-based model we utilized the K-NN algorithm, using the fitcknn MATLAB function. The first thing that stood out to us is that it took a very long time to build the model and predict the labels. This became a problem when trying to choose the best K for the model. We tried to use 10-fold cross-validation to find the optimal K, but found that it took too much time, to the point of being impractical. Even with with K=5, it would take longer than the ten minute time limit to predict all the labels when using all 10000 features. By using feature selection, we cut the number of features down to the top 800 words that appeared the most. Using that approach, K=5 yielded a cost of 1.4334 and K=20 yielded a cost of 1.3313. Increasing K lowered the test error, but took a longer time to compute. We considered lowering the number of features so we could increase the number of neighbors, but we realized K-NN would not be able compete with other methods, such as Naive Bayes. K-NN is not an efficient algorithm for this project, nor is it for large data sets with many features, in general. It requires a good deal of pre-processing. A recommended starting K is square-root of n, where n is number of features, which would have taken too long to predict the labels given the ten minute time limit.

Decision Trees/Random Forests

For our discriminative-based model, we used MATLAB's fitctree and TreeBagger functions to build decision trees and random forests, respectively. On principle, we did not expect a decision tree model to be incredibly successful. Decision trees have a tendency to overfit, which can hinder language models, and we did not believe that word frequency lends itself well for decision trees. Decision trees make more local decisions rather than looking at the overall context, especially when compared to something like a Naive-Bayes model. The model took a relatively long time to build and predict, but performed surprisingly well, with a cost of 1.3405.

We then tried to improve on the decision tree model by using random forests, which could hopefully highlight the more important words, and attempted to contain both the size of the model and the time it took to build and predict. We found an acceptable middle ground of 50 trees, which performed at a cost of slightly over 1 on our 10-fold cross-validation. Unfortunately, that improvement was not reflected in the validation set results and the model returned a cost of 1.3324. Due to the poor performance, we quickly moved away from this model.

Naive Bayes

For our generative-based model, we implemented Naive Bayes using MATLAB's fitcnb(X_train, Y_train, 'distribution', 'mn'), where 'mn' accounts for a multinomial distribution. Naive Bayes without any preprocessing gave us a cost of 0.95379, which was one of the best results from a basic model. We expected Naive Bayes to perform well, since it tends to do pretty well in text-processing contexts. We then added additional features to try and improve the model. We used 10-fold cross-validation to determine which features worked and which did not .

- <u>Punctuation:</u> We counted the words in a tweet that included some form of punctuation and the words in the same tweet without punctuation separately and add these two features to the input matrix. So if a tweet had 18 words, the original row adds up to 18 and the row in the new input matrix adds up to 36. The new input matrix has the dimensions m by 10,002. This approach gave us a cost of 0.9471 in our cross-validation.

- Word length: This is the same as punctuation approach, but with counting words that are longer than 8 characters. The optimal length was found using our cross-validation, which returned a cost of 0.9489.

- Numbers: This is the same as punctuation approach, but counting words that include numbers. This returned a cross-validated cost of 0.95169.

Using the additional punctuation features gave us the best results (words featuring punctuation such as smileys are highly predictive) and so we used this for our final model. If we had more time, we would have chosen to implement Naive Bayes with bigrams, since that can yield better results.

Logistic Regression

We used the liblinear package to implement logistic regression. L2-regularized logistic regression gave us the best results and it has a probability distribution as an output. This is a huge advantage over linear regression since we have a cost sensitive problem. The cost for this method was 0.96032 on the validation set. Using binary data (word exists or doesn't) improved the cost to 0.94788, which was expected since logistic regression performs better on binary data, and further adding a constant feature improved the cost to 0.94599. We used the latter method in our final model.

Linear Regression

We used the liblinear package for this as well. The output is only the class label, so there is no probability distribution. In order to make linear regression usable for our ensemble we changed the prediction matrix to a categorical matrix. A label "2" would become "0 1 0 0 0". In this way we can add all the probability distribution matrices from the different models for our ensemble in the end. It gave us a lackluster cost of 1.0898.

Ensemble Model:

Final Model - Ensemble of Naive Bayes and Logistic Regression

Logistic regression, Naive Bayes and linear regression gave us promising results. So we built an ensemble using these methods. For the ensemble, we added up all the probability distributions from our different methods. In order to find the best weights, we used fminsearchbnd, a function that searches for minima inside a given boundary for the variables. This model trained amazingly quickly, especially when compared to something like K-NN. It was also very lightweight in terms of size.

```
%% find the best weights
fun = @(x)loss_function(probability_to_class(prob_nb*x(1) + prob_lr * x(2) + prob_lin * x(3)), Y_test_prop);
best_weights = fminsearchbnd(fun, [.5, .5, .5], [0, 0, 0], [1, 1, 1]);
```

Including linear regression did not improve our cost. This is probably due to the similarity between the predictions from logarithmic and linear regression and because it does not return a real probability distribution (see above). The code below shows our final ensemble.

```
%% ensemble
prob_total = 0.4*prob_nb + 0.6*prob_lr + 0*prob_lin;
```

We needed to convert the probability distributions to classes. Our code picks the class with the lowest expected cost. "prob_total" is a m by 5 matrix and "Y_hat" is a m by 1 matrix.

```
%% probability to class
costs = [0 3 1 2 3; 4 0 2 3 2; 1 2 0 2 1; 2 1 2 0 2; 2 2 2 1 0];
[~, Y_hat] = min(prob_total*costs, [], 2);
```

The most surprising part of this ensemble is how well it worked. On the validation set, the model performed just outside the top 20 models, but we jumped to 4th place on the test set and were just .00011 off from being in third. We were probably this successful because we were very careful not to overfit the training set and to only include approaches that enhanced the model rather than simply augment it. Also, luck probably played a role.

Experimental Approaches:

In one of our failed experimental approaches, we attempted to take advantage of the validation set given and utilize a semi-supervised approach to help create a more robust model. First, we created a Naive Bayes model with the training data and then we would predict the class probabilities on the validation tweets and if the predicted probability for a class was above 0.7, we were confident enough to label that tweet with that class and add it to the training data to create a new model. However, this did not benefit us and negatively affected our test error, due to the fact that the labels we predicted might have been wrong and we are now introducing bad data into our training set. It also seemed to overfit the training data.

Another model we briefly experimented with was SVM. At first we created an SVM model in Matlab using fitcecoc with the default values and all 10,000 features. We received a test error of 1.0374 from the leaderboard. To help optimize this model we then tried to create an SVM model on Matlab with a Gaussian kernel. This took a much longer time to train with the 10,000 features. We then cut the number of features down to 400 and trained the model. However, after trying to predict the model locally it still took longer than the 10-minute time limit and at this point we thought it would be best to abandon this method because we were having more success with other methods.

We also experimented with a variety of distance metrics including cosine similarity and Jaccard similarity. We generated "sum" vectors, which summed the feature vectors of tweets of the same class. We then ended up with "sum" vectors for each class and calculated the distance between a test tweet and every "sum" vector (using Cosine similarity or Jaccard similarity). The test tweet would be classified by the closest "sum" vector. Jaccard performed better than Cosine, both in terms of time to calculate and average cost. Cross-validated results, when added to our ensemble model, frequently averaged a cost of below .91. Unfortunately, it did not perform as well as other models on the validation set, due to overfitting.

Finally, we tried to experiment with different ensemble methods. One idea was to cluster and reclassify results using K-means or K-NN. However, these models often yielded much worse results, so we did not end up using them.

Method	**Validation Cost**
Decision Tree	1.3405
Random Forest (50 trees)	1.3324
KNN K =5	1.4334
KNN K = 20	1.3313
SVM via Matlab's standard fitcecoc	1.0374
Logistic Regression with binary data and additional constant feature	0.9459*
Naive Bayes with additional punctuation features	0.9471
Naive Bayes + Logistic Regression	0.9268
Linear Regression	1.0898
Unweighted Jaccard Distance + Naive Bayes + Logistic Regression	0.9266
0.4 * Naive Bayes with punctuation features + 0.6 * logistic regression with binary data and additional constant feature	**0.9176**

***Cross-Validated Result**