 1 help

Text categorization: combining different kind of features

The problem I am tackling is categorizing short texts into multiple classes. My current approach is to use tf-idf weighted term frequencies and learn a simple linear classifier (logistic regression). This works reasonably well (around 90% macro F-1 on test set, nearly 100% on training set). A big problem are unseen words/n-grams.

I am trying to improve the classifier by adding other features, e.g. a fixed sized vector computed using distributional similarities (as computed by word2vec) or other categorical features of the examples. My idea was to just add the features to the sparse input features from the bag of words. However, this results in worse performance on the test and training set. The additional features by themselves give about 80% F-1 on the test set, so they aren't garbage. Scaling the features didn't help as well. My current thinking is that these kind of features don't mix well with the (sparse) bag of words features.

So the question is: assuming the additional features provide additional information, what is the best way to incorporate them? Could training separate classifiers and combining them in some kind of ensemble work (this would probably have the drawback that no interaction between the features of the different classifiers could be captured)? Are there other more complex models I should consider?

machine-learning classification feature-selection logistic-regression information-retrieval

asked Aug 17 '14 at 17:29

elmille
56 4

1 Some update: I was able to achieve acceptable results by l2-normalizing the additional dense vectors. I wrongfully assumed the sklearn StandardScaler would do that. I am still looking for more complex methods, though, that would allow me to model label dependencies or incorporate confidence of sub-classifiers. – elmille Aug 19 '14 at 22:11

I was doing basic the same experiment last year and encounter the exactly the same problem you have. Can your word2vec vector after l2-normalizing process beat BOW? I haven't done l2-normalizing, but even after testing many post processing method semantic vector is still 2-4 absolute percent behind BOW tf/idf features I wonder is that direction a deadend. My original sought is to combine a densely semantic vector with traditional BOW and see if it can enhance topic classification/modeling performances. BTW: what data set have you been working on, mine is 20newsgroup. – user15150 Jan 3 at 15:42

I was working with a dataset for the CIKM 2014 competition. For me, the vector representations were never able to beat BOW with tf-idf weights. My plan was to use them in addition to improve quality. In my experience (for text classification) some form of tf-idf + a linear model with n-grams is an extremely strong approach. I am currently experimenting with convolutional neural networks and even with these (more or less) complex models that approach hard to beat. – elmille Jan 4 at 16:56

To Mod: Sorry that I don't have 50 reputation, so I can't write in the comment area. Hi elmille: Yes, that is what I experience in all the test. However, do you find that word vec + BOW help? In my experience, when I concatenate word vec with BOW tf-idf(in my case this vec is actually a overall vector within entire article, its not word-vec but very similar), the performance get even lower. I originally think it should be BOW+vec > BOW > vec. Since they contain mutually assistant information. The actually result is BOW>vec>BOW+vec. Then I do standard scaling and normalization to bow and vec in – user15150 Jan 5 at 1:52

2 Answers

Linear models simply add their features multiplied by corresponding weights. If, for example, you have 1000 sparse features only 3 or 4 of which are active in each instance (and the others are zeros) and 20 dense features that are all non-zeros, then it's pretty likely that dense features will make most of the impact while sparse features will add only a little value. You can check this by looking at feature weights for a few instances and how they influence resulting sum.

One way to fix it is to go away from additive model. Here's a couple of candidate models.

SVM is based on separating hyperplanes. Though hyperplane is linear model itself, SVM doesn't sum up its parameters, but instead tries to split feature space in an optimal way. Given the number of features, I'd say that linear SVM should work fine while more complicated kernels may tend to overfit the data.

Despite its name, **Naive Bayes** is pretty powerful statistical model that showed good results for text classification. It's also flexible enough to capture imbalance in frequency of sparse and dense features, so you should definitely give it a try.

Finally, **random forests** may work as a good ensemble method in this case. Randomization will ensure that different kinds of features (sparse/dense) will be used as primary decision nodes in different trees. RF/decision trees are also good for inspecting features themselves, so it's worth to note their structure anyway.

Note that all of these methods have their drawbacks that may turn them into a garbage in your

case. Combing sparse and dense features isn't really well-studied task, so let us know what of these approaches works best for your case.

answered Aug 18 '14 at 13:35

ffriend
1,936 5 14

Thank you for your answer! I have two follow-up questions :) 1) How are SVM (with a linear kernel) and Naive Bayes different in that they do not sum up their features and corresponding weights (i.e. what you call an "additive model")? Both effectively create a separating hyperplane so isn't the result is always some kind of adding features multiplied by corresponding weights? 2) I'd like to try random forests, but unfortunately the feature space is too large to represent it in dense format (I'm using sklearn). Is there an implementation that can handle that? – elmille Aug 19 '14 at 22:07

1) In linear regression you are interested in points *on* hyperplane, thus you add up weighted features to get predicted point. In SVM, on other hand, you are looking for points *on the sides* of hyperplane. You do classification by simple checking on which side is your example, no summation involved during prediction. Naive Bayes may incorporate different kinds of models (e.g. binomial or multinomial), but basically you multiply probabilities, not add them. – ffriend Aug 20 '14 at 15:33

2) I have seen some research in this topic, but never encountered implementation (probably googling will give some links here). However, you can always go another way - reduce dimensionality with, say, PCA and then run random forest based on reduced dataset. – ffriend Aug 20 '14 at 15:35

If I understand correctly, you essentially have two forms of features for your models. (1) Text data that you have represented as a sparse bag of words and (2) more traditional dense features. If that is the case then there are 3 common approaches:

1. Perform dimensionality reduction (such as LSA via `TruncatedSVD`) on your sparse data to make it dense and combine the features into a single dense matrix to train your model(s).
2. Add your few dense features to your sparse matrix using something like scipy's `hstack` into a single sparse matrix to train your model(s).
3. Create a model using only your sparse text data and then combine its predictions (probabilities if it's classification) as a dense feature with your other dense features to create a model (ie: ensembling via stacking). If you go this route remember to only use CV predictions as features to train your model otherwise you'll likely overfit quite badly (you can make a quite class to do this all within a single `Pipeline` if desired).

All three approaches are valid and have their own pros and cons. Personally, I find (1) to typically be the worst because it is, relatively speaking, extremely slow. I also find (3) to usually be the best, being both sufficiently fast and resulting in very good predictions. You can obviously do a combination of them as well if you're willing to do some more extensive ensembling.

As for the algorithms you use, they can essentially all fit within that framework. Logistic regression performs surprisingly well most of the time, but others may do better depending on the problem at hand and how well you tune them. I'm partial to GBMs myself, but the bottom line is that you can try as many algorithms as you would like and even doing simple weighted ensembles of their predictions will almost always lead to a better overall solution.

answered Jan 6 at 4:20

David
419 2 7
