Learning Sparser Perceptron Models

Yoav Goldberg and **Michael Elhadad**
Ben Gurion University of the Negev
Department of Computer Science
POB 653 Be'er Sheva, 84105, Israel
{yoavg,elhadad}@cs.bgu.ac.il

Abstract

The averaged-perceptron learning algorithm is simple, versatile and effective. However, when used in NLP settings it tends to produce very dense solutions, while much sparser ones are also possible. We present a simple modification to the perceptron algorithm which allows it to produce sparser solutions while remaining accurate and computationally efficient. We test the method on a multiclass classification task, a structured prediction task, and a guided learning task. In all of the experiments the method produced models which are about 4-5 times smaller than the averaged perceptron, while remaining as accurate.

1 Introduction

Linear models are the most commonly used machine-learning (ML) techniques in natural-language-processing. Such models are based on linear scoring functions of the form $f(x) = \mathbf{w} \cdot \mathbf{x} = \sum_i w_i x_i$, that is a linear combination of various features x_i of the observed input $\mathbf{x}$, weighted according to a set of parameters $\mathbf{w}$. The task of the learning algorithm is to set the weights w_i to produce a good scoring function for the task at hand.

In NLP, features are usually binary indicator functions indicating the presence of some condition (i.e. a specific feature will take the value of 1 if the previous word is "dog" and 0 otherwise), or counts over several indicator functions (i.e. a feature will take the value of 4 if the word "cat" appears 4 times in a document). This results in a very high-dimensional feature space (the set of possible features is very large), yet only few of the features are active for each observed instance. Of course, not all of the features are relevant for classification. Ideally, such irrelevant features would get a weight of 0 in the training process. Alternatively, and as usually happens in practice, irrelevant features may get a weight which is different from 0 but which still does not affect the final predictor accuracy.

A classic algorithm for learning linear models is the perceptron (Rosenblatt, 1958). The algorithm is simple, efficient, easy to implement, and was extended to be used for structured-prediction (Collins, 2002) and guided-learning (Shen et al., 2007). A popular variant on the perceptron algorithm commonly used in practice is the averaged-perceptron (Freund and Schapire, 1999), which can be seen as a way of adding regularization to the perceptron algorithm. While the algorithm is empirically robust to irrelevant features, it tends to produce relatively dense models, in which many parameters have non-0 weights. Coupled with the large feature-sets used in NLP applications, model sizes can get very big. A model with many 0-valued features is said to be *sparse*. Sparse models have the advantage of requiring less space for memory and storage, they are faster to load from disk and are faster to transmit over a network.

This paper introduces a variant on the perceptron learning algorithm allowing it to produce sparser, smaller models.

2 Averaged Perceptron and Variants

We consider 3 variants of the perceptron algorithm.

In the **multiclass** case, the label set is assumed to be $\mathcal{Y} = \{1, 2, ..., k\}$. Here the goal is learning k scoring functions $f_1, ..., f_k$, one for each class, such that for instances of class L we get $f_L(\mathbf{x}) > f_{l \neq L}(\mathbf{x})$. Each scoring function f_k is parameterized by a vector $\mathbf{w}_{:\mathbf{k}}$ of weights. In the **structured** case (Collins, 2002; McDonald et al., 2005), the goal is learning a single scoring function $f(\phi(\mathbf{x}, \mathbf{y}))$ over features extracted from the observed data $\mathbf{x}$ together with a hidden structure $\mathbf{y}$, such for an instance $\mathbf{x}$ with a known structure $\mathbf{y}$ we get $f(\phi(\mathbf{x}, \mathbf{y})) > f(\phi(\mathbf{x}, \mathbf{y}'))$ for all $\mathbf{y}' \neq \mathbf{y}$.

The perceptron learning algorithm is error-driven. It works by maintaining a weight vector $\mathbf{w}$, and making several passes over the training data. Each pass is made of rounds. At each round t the algorithm is presented with a labeled training example $(\mathbf{x}^t, y^t)$ where $\mathbf{x}^t$ is an instance to be classified and y^t is its true label or structure. The algorithm uses its current weight vector $\mathbf{w}^t$ to calculate the score $\mathbf{w}^t \cdot \mathbf{x}^t = \sum_i w_i^t x_i^t$. If the score is good (leads to a correct prediction of y), there is no need to change the parameters and so $\mathbf{w}^{t+1} \leftarrow \mathbf{w}^t$. Otherwise, an update occurs. For the multiclass case, assume the algorithm predicted an incorrect label i while the correct label is c, the update is $\mathbf{w}_{:\mathbf{c}}{}^{t+1} \leftarrow \mathbf{w}_{:\mathbf{c}}{}^{t} + \mathbf{x^t}$ and $\mathbf{w}_{:\mathbf{i}}{}^{t+1} \leftarrow \mathbf{w}_{:\mathbf{i}}{}^{t} - \mathbf{x^t}$. For the structured case, assuming the incorrect structure $\mathbf{y}'$ got scored above the correct structure $\mathbf{y^t}$ the update is $\mathbf{w}^{t+1} \leftarrow \mathbf{w}^t + \phi(\mathbf{x^t}, \mathbf{y^t}) - \phi(\mathbf{x^t}, \mathbf{y}')$.

Both of these updates increase by 1 the weights of the features of items that got scored too low, and decrease by 1 the weights of the features of items that got scored too high.

Guided Learning A recent line of work attempts to integrate search into the training procedure (Daumé III et al., 2009; Shen et al., 2007; Shen and Joshi, 2008; Goldberg and Elhadad, 2010). In the guided-learning setting, we are faced with a task that can be decomposed into a series of smaller steps (such as tagging individual words in a sequence-labeling task, or attaching syntactic structures in a parsing task). There are many possible orders in which the steps could be taken, and each step can add information to be used in subsequent steps. The learner does not know in advance the correct order of steps to take. The task of the learning algorithm is twofold: to learn which steps are valid, and to learn a good ordering of steps. This can be done with a variant of the multiclass perceptron. While beyond the scope of this paper, details are available in (Shen et al., 2007; Goldberg and Elhadad, 2010). This is particularly challenging for methods that attempt to optimize the number of the learned parameters because: (1) the set of training examples and features is not fixed in advances, but is determined dynamically and interacts tightly with the progress of learning. (2) There is no natural, easily-computable notion of loss. These two factors contribute to very large and very dense parameter vectors, and prevent us from using other methods for achieving sparse parameter vectors, such as the rare-feature pruning and $L1$ regularization. Our approach is still relevant and effective in this setting as shown below.

Parameter Averaging The perceptron algorithm is prone to overfiting the training data. A common way around this is parameter averaging (Freund and Schapire, 1999): the final weight vector $\mathbf{w}$ is taken to be the average of all the parameter vectors $\mathbf{w^1}, ..., \mathbf{w^T}$ seen in training. Averaging reduces the variance between the different vectors, and produces a regularization effect. While hard to analyze theoretically, averaged perceptron are empirically shown to produce very accurate predictors. The averaged weight vector can be computed incrementally without storing all the intermediate vectors. For details, see, e.g. (Collins, 2002).

3 Learning Sparser Models

Why does the original perceptron model produce dense models? Note that at each update, all the features in the training example that triggered the update are added to the weight vector. Even if an example can be correctly scored without a given feature, the feature would still participate in the update and hence appear in the final parameters vector. Moreover, if it is a rare feature its chances of changing its weight back to 0 because of an opposite update are extremely small.

Our sparse variant of the perceptron algorithm is based on the intuition that only relevant features should end up in the final parameter vector, and that all features are irrelevant until proven otherwise. For a feature to prove its relevance, it needs to participate in a minimum number of updates. That is, a feature is relevant if our attempts to ignore it fail at-least $minUpdate$ times.[1]

The concrete algorithm is very simple. In addition to the weight vector maintained by the perceptron algorithm, we maintain another vector U of update counts, keeping track of the number of times each feature participated in an update. The scoring function is then instructed to ignore features which participated in less than $minUpdate$ updates.

Recall that in the multiclass classification case a perceptron is maintained for each class. In this case the update-counts vector U is shared between all the perceptrons. That is, a feature becomes relevant if it was updated at least $minUpdates$ times, in *any* of the parameter vectors.

Note that in both cases, the update-counts affect only the scoring function and not the update function. Updates to the weight vector w are done as usual, using all the features of the example that triggered the update. This means that once a feature becomes relevant (its count in U is high enough) it comes with its update-history – it behaves as if it was there from the beginning, and we do not need to waste extra time in learning a new weight for it.

Convergence Proof The averaged perceptron is not well understood theoretically, and is hard to analyze. However, the convergence proof of the non-averaged perceptron carries on to the modified version with very little adjustments, and our modified perceptron algorithm is guaranteed to converge after a finite number of passes over the training data. To see this, notice that for a minUpdate value of k we could always lump together k iterations of the algorithm and rearrange the order of examples internally so that the algorithm is presented with the same example for k times in a row, after which all the features in the example are updated, and the algorithm is essentially the same as the original perceptron. As the original proof does not care about the order of examples, it holds also for the modified version which is guaranteed to converge as well.

[1]In a sense this motivation is the opposite of the one for confidence-weighted learning (Dredze et al., 2008), in which parameter updates are more aggressive for rare features than for frequently seen ones – the confidence-weighted learning algorithm has extreme trust in the relevance of rare features.

Comparison to Feature-Pruning A commonly used method for achieving smaller models is rare-feature pruning: ignoring features which occur less than k times in the training data. This creates a smaller model by reducing the dimensionality of the data. In many cases such feature pruning does not hurt the algorithm performance while resulting in a much smaller model. We view this method as a crude way of doing feature selection. This method has three shortcomings compared to ours. (1) It may prune too little. Irrelevant features occurring k times or more still have a good chance of finding their way into the final model. We demonstrate in Section 4 that our method can achieve smaller models than the feature-pruned ones. (2) It may prune too much. Some rare feature can be important to classification, in the sense that without them some instances cannot be disambiguated. Our method will include these features eventually. (3) Finally, feature-pruning is not always feasible. Structured prediction problems tend to introduce enormous feature sets, and keeping track of feature counts over all possible features can get very computationally demanding. In the *guided learning* setting the training examples are dynamic and interact with the learning, making it infeasible to even enumerate the possible feature set, let alone count their occurrences.

Comparison to L1 Regularization Another popular method for achieving sparse models is L_1 regularization: requiring the L_1 norm of the final weight vector to be small. A parameter C controls the trade-off between minimizing the L_1 norm (effectively setting many of the parameter values to 0, resulting in sparser models) and minimizing the loss (resulting in accurate predictions). In an online training, setting the L_1 norm minimization can be achieved using a variation of SGD, see (Tsuruoka et al., 2009; Langford et al., 2009) for details. L_1 norm minimization is well understood and theoretically justified for models such as SVMs and log-linear conditional models. While it could also be applied to the 0-1 loss of the perceptron, its interpretation and application with regard to the averaged-perceptron is not as clear. Furthermore, L_1 regularization cannot be applied in the guided-learning setting, where the loss function is tricky to define.

Finally, both L_1 regularization and our method require setting a parameter to control the amount of sparsity. We find the semantics of our parameter ("don't include a feature if it took part in an update less than k times") to be much more intuitive than for L_1 regularization (either "small L_1 norm is C times more important than small loss", or "we believe the parameter values should behave according to a Laplace prior with mean 0 and variance σ^2"). This has the advantage of making the choice of parameter easier to interpret and to explain.

4 Experiments and Results

We present experiments with the three variants of the averaged-perceptron algorithm (multiclass, structured, guided) described above.

For each of the tasks we compare our sparse averaged-perceptrons (SAP) with minUpdate values of 5 and 10 (SAP_5 and SAP_{10} respectively) to the standard averaged percepton. For comparison with the feature-pruning baseline, for the multiclass task, we also train two feature-pruned averaged-perceptron models with rare-feature thresholds of 5 and 10 (that is, features appearing less than 5 or 10 times in the training set are removed prior to training). For each of the runs we report the task-specific accuracy, the model size in terms of non-0 features, and the size of the model file in megabytes. All models were trained for 10 passes over the training data.

Multiclass Classification: Text Chunking We consider the text-chunking task as defined in the CoNLL 2000 shared task (Sang and Buchholz, 2000). We use the common `BIO` encoding (Marcus and Ramshaw, 1995) in which each word is classified as being at the `B`eginning, `I`nside, or `O`utside of a chunk. With the 11 chunk types in the task this amounts to a 23-way classification problem (`B-C` and `I-C` for each of the 11 chunk type `C`, and `O`). We use the training and test sets of the shared task.

We use a feature set based on (Sha and Pereira, 2003; Zhang et al., 2001). All the features are binary indicator variables, and include the identities of words and pos-tags surrounding the word being classified, the two previous chunk tags[2], and various conjunctions of those. The accuracy metric we report for this task is per-word classification accuracy.

[2]These are fixed at training time and determined dynamically at test times based on the previous classifier decisions.

ALGORITHM	ACC	SIZE(# FEATS.)	SIZE(MB)
Avg.Percep	96.0	196,523	31M
pruned(5)	95.9	77,880	13M
pruned(10)	95.8	52,072	9.0M
SAP_5	95.9	47,906	8.4M
SAP_{10}	95.9	26,160	5.0M

Table 1: Multiclass Classification Results

ALGORITHM	ACC	SIZE(# FEATS.)	SIZE(MB)
Avg.Percep	89.7	5,265,583	209M
SAP_5	89.7	1,719.432	66M
SAP_{10}	89.6	1,018,336	38M

Table 2: Structured Prediction Results

ALGORITHM	ACC	SIZE(# FEATS.)	SIZE(MB)
Avg.Percep	89.5	2,988,660	121M
SAP_5	89.6	1,095,898	47M
SAP_{10}	89.4	644,264	28M

Table 3: Guided Learning Results

Structured Prediction: Edge-factored Dependency Parsing For the structured-prediction experiment, we take first-order edge-factored projective dependency parsing, using the Eisner algorithm for inference (Carreras et al., 2006; Eisner, 2000). As usual, the models are trained on sections 2-21 of the WSJ corpus, and evaluated on section 23. We use the feature-set of (McDonald, 2006). Both the train and test sentences are automatically pos-tagged.The accuracy metric we report for this task is unlabeled attachment accuracy, i.e. the percent of words in the evaluation set which got assigned the correct parent.

Guided Learning: Easy-first Dependency Parsing For the guided-learning experiment, we use the Easy-First dependency parsing implementation of (Goldberg and Elhadad, 2010), using the default English feature set[3]. We use the same dataset and accuracy metric as in the structured-prediction case.

Results are presented in Tables 1,2 and 3. They show the method is effective at reducing the size of the learned model while preserving accuracy.

5 Conclusions

We presented a simple modification of the perceptron algorithm, with the goal of producing sparser models. The method is effective: our experiments show that in three different settings the method produces models which are about 4-5 times smaller, with only a small, if any, drop in accuracy.

[3]http://www.cs.bgu.ac.il/∼yoavg/software/easyfirst/

References

X. Carreras, M. Surdeanu, and L. Marquez. 2006. Projective dependency parsing with perceptron. In *Proc. of CoNLL-2006*, pages 181–185. Association for Computational Linguistics.

M. Collins. 2002. Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms. In *Proceedings of the ACL-02 conference on Empirical methods in natural language processing-Volume 10*, pages 1–8. Association for Computational Linguistics.

Hal Daumé III, John Langford, and Daniel Marcu. 2009. Search-based structured prediction. *Machine Learning Journal (MLJ)*.

Mark Dredze, Koby Crammer, and Fernando Pereira. 2008. Confidence-weighted linear classification. In *Proc. of ICML-2008*.

J. Eisner. 2000. Bilexical grammars and their cubic-time parsing algorithms. *Advances in Probabilistic and Other Parsing Technologies*, pages 29–62.

Yoav Freund and Robert E. Schapire. 1999. Large margin classification using the perceptron algorithm. *Machine Learning*, 37(3):277–296.

Yoav Goldberg and Michael Elhadad. 2010. An efficient algorithm for easy-first non-directional dependency parsing. In *Proc. of NAACL*.

J. Langford, L. Li, and T. Zhang. 2009. Sparse online learning via truncated gradient. *The Journal of Machine Learning Research*, 10:777–801.

M. Marcus and L. Ramshaw. 1995. Text Chunking Using Transformation-Based Learning. In *Proc. of the 3rd ACL Workshop on Very Large Corpora*.

Ryan McDonald, Koby Crammer, and Fernando Pereira. 2005. Online large-margin training of dependency parsers. In *Proc of ACL*.

Ryan McDonald. 2006. *Discriminative Training and Spanning Tree Algorithms for Dependency Parsing*. Ph.D. thesis, University of Pennsylvania.

F. Rosenblatt. 1958. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386–408.

Erik F. Tjong Kim Sang and S. Buchholz. 2000. Introduction to the conll-2000 shared task: chunking. In *CoNLL-2000*.

F. Sha and F. Pereira. 2003. Shallow parsing with conditional random fields. In *Proc. HLT/NAACL 2003*, pages 134–141. Association for Computational Linguistics.

Libin Shen and Aravind K. Joshi. 2008. Ltag dependency parsing with bidirectional incremental construction. In *Proc of EMNLP*.

Libin Shen, Giorgio Satta, and Aravind K. Joshi. 2007. Guided learning for bidirectional sequence classification. In *Proc of ACL*.

Y. Tsuruoka, J. Tsujii, and S. Ananiadou. 2009. Stochastic gradient descent training for l1-regularized log-linear models with cumulative penalty. In *Proc. ACL 2009*, pages 477–485. Association for Computational Linguistics.

Tong Zhang, Fred Damerau, and David Johnson. 2001. Text chunking using regularized winnow. In *ACL '01: Proceedings of the 39th Annual Meeting on Association for Computational Linguistics*, pages 539–546, Morristown, NJ, USA. Association for Computational Linguistics.