Name That Language: *Ajilabal Rech Eta'manik*[*]

Matthew R. Gormley
mgormley@cs.cmu.edu

1 Introduction

Machine learning is an exciting discipline that is at the intersection of probability, statistics, computer science, optimization, and linear algebra—to name a few. This lab explores the diversity of avenues-for-improvement that any machine learning researcher or practitioner faces when attempting to tackle a real-world problem. Our focus here is on supervised multi-class classification. We choose as our application, the problem of language identification: given a snippet of text, we predict which language it is written in.

In recent years, the availability of high quality open source libraries and toolkits for machine learning has blossomed. Throughout this summer school you will be exposed to many of these, and asked to apply them to challenging speech and language problems. Your *goal* in this lab is to obtain the highest accuracy on a language identification benchmark. The easiest way to achieve this goal would undoubtedly be to apply one of the aforementioned ML toolkits. However, we impose the *constraint* that no such libraries are allowed—if you want a fancy ML technique, you'll have to implement it from scratch![1]

We expect you to work in small teams.

2 Datasets

We will consider two datasets as our benchmarks for language identification.

Family	Language (Dialect)	ISO Code
A	Bosnian	bs
	Croatian	hr
	Serbian	sr
B	Indonesian	id
	Malay	my
C	Czech	cz
	Slovak	sk
D	Portuguese (Brazilian)	pt-BR
	Portuguese (European)	pt-PT
E	Spanish (Argentine)	es-AR
	Spanish (Castilian)	es-ES
F	Bulgarian	bg
	Macedonian	mk
	Others	xx

Table 1: Languages from DSL 2015 Shared Task

DSL 2015 The first dataset comes from the Discriminating Similar Languages (DSL) Shared Task 2015 (Zampieri et al., 2015) and is comprised of sentences from 13 languages across 6 language families. Discriminating between languages in different families is often easy; however, learning to differentiate between similar language varieties and dialects can be difficult. Each sentence consists of 20-100 tokens and was sampled from a journalistic source. Table 1 shows the languages and their corresponding ISO codes. The dataset is publicly available[2]. Following the shared task design, we provide you with the labeled data (i.e. language ISO codes included) for the train and development splits, but only the sentences from the test data.

[*] Seriously, can you name it?

[1] The astute reader would question whether it is feasible to implement an end-to-end ML system from scratch in just 2 hours. Not to worry: we'll permit the starter code described in Section 3

[2] `https://github.com/Simdiva/DSL-Task`

	DSL 2015	Tatoeba
# languages	14	321
avg. tok/sent	35.4	8.2
# sent in train	252k	4,625k
# sent in dev	28k	578k
# sent in test	14k	578k
size on disk	71 Mb	306 Mb

Table 2: Summary statistics for two datasets: the DSL 2015 Shared Task, and Tatoeba. Sentence counts are given in thousands (k).

Tatoeba[3] The second dataset comes from Tatoeba[4], a collaboratively edited set of sentences and their translations in 300+ languages. Because of their source, the sentences tend to be shorter in length. However, they exist in great abundance: as of the time of collection there were over 5 million sentences available. Table 2 summarizes various statistics about the two datasets.

Format The source formats of these two datasets differ, so we have preprocessed the data into a simple two column format, where the language code and the sentence are separated by a TAB character, and the words in the sentence are space separated.

```
<lang><TAB><sentence>
```

The lines in the training files have been shuffled in order to facilitate stochastic gradient descent in a streaming setting.

3 Starter Code

The starting point for this lab are some basic implementations of feature extraction (`features.py`) and online gradient descent for logistic regression (`classify.py`), plus a few handy make commands (`Makefile`). The code is available on GitHub (`http://github.com/mgormley/jsalt-lab`).

features.py The feature extraction code reads the provided "simple" files formatted in `<lang><TAB><sentence>` and outputs the "features" files in a slightly different format: the label (in this case the language code string) is separated from the features by a TAB character. The features are separated by SPACE characters.

```
<label><TAB><feat1><SPACE><feat2><SPACE>
    ... <SPACE><featN>
```

The initial version of `features.py` is quite simple: For Tatoeba's Mandarin and Japanese data, it outputs character unigram features. For all other languages, it outputs word unigram features—the definition of a "word" here is quite naive as it simply splits the sentence on spaces.

classify.py We also provide an implementation of multinomial logistic regression where learning is done by online gradient descent (aka. stochastic gradient descent (SGD)). Much of the boilerplace code for the experimental setup is already in place, leaving you to improve upon the existing (rather basic) implementation. Though simple, it includes quite a few nice tricks:

- The data can be read either in a streaming setting, or cached in memory—both have advantages. If the number of epochs for SGD (`--num_epochs`) is greater than 1, it will automatically cache the training data. Dev data is always cached.
- Features are read in as strings and immediately hashed to integers modulo the maximum number of features (`--num_features`). This requires no bookkeeping and, while there may be collisions, they are rare provided the modulus is sufficiently large. This is called the feature hashing trick (Weinberger et al., 2009). Another advantage is that the model parameters have a fixed capacity before reading any data—convenient for online learning.
- The full label set is read from a file at the start of the program (`--labels`). This ensures we can easily restrict the total number of training examples read in, again with minimal bookkeeping (`--train_max`). The model keeps track of the mapping from labels to `int`s for internal use.
- After each epoch, the accuracy on dev data is reported. The frequency of reporting can be changed as well (`--dev_iters`).
- Internally, each instance is represented as a sparse feature vector. The sparse vector is

[3] Can you name the language source for this name? *Hint:* don't be deceived by the UTF-8 bytecodes.

[4] `http://tatoeba.org`

stored as a list of feature indices (i.e. the hashed, modulo-ed values). For each index found in the list, its value is assumed to be the number of occurrences of that index. Indices that are absent have value 0. This ensures that the SGD only needs to perform sparse updates. We default to a maximum of 1 million features; for DSL 2015's 14 languages, this means the parameter vector consists of 14 million `float`s. As such, only updating the handful of parameters is quite convenient.

Makefile The Makefile is a collection of useful commands for preprocessing the data and running experiments. The commands are briefly described below.

setup: Install the required packages, download the data, and extract it.
dslf: Convert the raw DSL 2015 data to its feature representation.
dslc: Train a classifier on the DSL 2015 data.
dslcfast: Train a classifier on a subset of the DSL 2015 data.
dsl: `dslf` followed by `dslc`.
tatf: Convert the raw Tatoeba data to its feature representation.
tatc: Train a classifier on the Tatoeba data.
tatcfast: Train a classifier on a subset of the Tatoeba data.
tat: `tatf` followed by `tatc`.
prof: Profile the classifier with cProfile.

Be sure to run `make setup` before any of the others.

3.1 PSC Bridges

It may be convenient to collaborate on this exercise using the PSC Bridges compute nodes. You can log into the cluster via `ssh <username>@bridges.psc.edu`. Once logged into, the head nodes have full internet access. You should not run jobs directly on the head nodes. Instead, start an interactive session on a compute node via the command `interact`. Compute nodes do not have web access. If you are familiar with the Slurm queueing system you could even use `sbatch` to submit jobs to the grid.

4 Submission Instructions

The goal of this lab is to achieve the highest possible accuracy on the test set (`test-blind.txt`), for which we've provided the sentences, but not the true labels. For convenience of processing, we've substituted the true labels in that file with a single dummy label.

Using the `--test_out` option on `classify.py`, you should output your best model's predictions on the test data to a file. The file should be named `test-dsl.txt` for DSL 2015 data, and `test-tat.txt` for Tatoeba data. Copy this file to the PSC Bridges cluster for us to evaluate it as below.

1. Login to the PSC Bridges cluster:
   `ssh <username>@bridges.psc.edu`
2. Create a directory for your team in the shared submission directory on Bridges:
   `mkdir /pylon2/ci560op/mgormley/submissions/<team>`
3. Copy the test output file to this subdirectory. If you are working on your laptop, you can use:

   ```
   scp test-{tat,dsl}.txt <username>@bridges.psc.edu:/pylon2/ci560op/mgormley/submissions/<team>/
   ```

   Running `make submit` will also give you the appropriate command for submission.

5 Improving the Online Learner

There are a variety of ways that the existing implementations of feature engineering and learning could be improved. Your team will have to select a collection of improvements to try. The ones below were selected because of their promise; though this list is hardly comprehensive and may even include some ideas that won't pan out as well for this particular Language Identification task.

5.1 Feature Engineering

The current feature set is quite minimal. The winners of the DSL 2015 Shared Task used word unigram and bigram features and character n-grams for $n \in \{1, \ldots, 6\}$ (Malmasi and Dras, 2015). Other possible features include bytecode features or sentence length features. Further improvements could also be made to the tokenization, which currently

has no special handling of punctuation. TF-IDF weighting of features has also been applied to this task (Zampieri et al., 2015). While the current `classify.py` is designed to support only binary indicator features it could be extended to support feature weights.

5.2 Regularization

The provided implementation of SGD does not use explicit regularization.

Early Stopping Early-stopping regularizes the training objective by stopping the learning algorithm after performance on development data has stopped improving. Early-stopping has been shown to be equivalent to L2 regularization in certain settings. Implementing this would require storing the best model seen so far, and using it to make predictions on test data.

L2 Regularization Explicit L2 regularization adds the sum of the squares of the parameters as an extra term in the objective function.

$$J_i(\boldsymbol{\theta}) + \lambda \sum_{m=1}^{M} \theta_m^2 \tag{1}$$

While the gradient of $J_i(\boldsymbol{\theta})$ will be sparse, the gradient of the L2 penalty will not be. For this reason, it's important to implement carefully. See Bottou (2012), Section 5.1 for full details. The key trick it to represent the parameter vector as the product of a scalar v and a vector $phiv$, so that $\boldsymbol{\theta} = v\boldsymbol{\phi}$.

Lazy L2 Regularization Implement lazy L2 regularization updates. Instead of updating each parameter at every iteration, lazily update only the parameters which have non-zero feature weights in the current example i. Note that this requires bookkeeping of when the last update to that parameter was made and all the L2 regularizer updates that were skipped, should be made at once. This is similar to the trick used for RMSProp above.

5.3 Optimization

Stochastic gradient descent uses an update of the form:

$$\boldsymbol{\theta} \rightarrow \boldsymbol{\theta} - \eta_t \nabla_{\boldsymbol{\theta}} J_i(\boldsymbol{\theta}) \tag{2}$$

where t is the current time step, η_t is the learning rate, i is a randomly chosen training example, $J_i(\boldsymbol{\theta})$ is the loss on that example. For logistic regression, this loss is the negative log-likelihood $-\log p(y_i|\mathbf{x}_i)$.

Better Selection of Learning Rate The current implementation of SGD uses a fixed learning rate, i.e. $\eta_t = \eta_0 \; \forall t$. Several improvements are given below.

- Instead, keep the learning rate fixed until accuracy on development data stops improving. Then divide the learning rate in half and repeat.
- Use a learning rate schedule. Bottou (2012) suggests setting $\eta_t = \frac{\eta_0}{1+\eta_0 \lambda t}$. The value λ is the weight of the L2 regularizer if used. Otherwise, it should be 1. The value of η_0 should be tuned on a small subset of the training data.
- Report the average progressive training loss on the examples. While it would be somewhat computationally expensive to compute the full training accuracy frequently, one can progressively evaluate the predictions that the classifier would make on the *next* example and report the average. This gives a clearer indication of how well the optimizer is optimizing the objective $\sum_i J_i(\boldsymbol{\theta})$.

AdaGrad One extremely effective technique is to learn with per-feature learning rates. One such adaptive learning rate technique is AdaGrad (Duchi et al., 2011). Instead of the vectorized update rule in Eq. (2), we can consider the per-parameter update that SGD performs:

$$\theta_m \rightarrow \theta_m - \eta_{tm} \frac{\partial J_i(\boldsymbol{\theta})}{\partial \theta_m} \tag{3}$$

where η_{tm} is the learning rate at time step t for parameter m. AdaGrad uses the following learning rates:

$$\eta_{tm} = \frac{\eta_0}{\sqrt{G_m + \epsilon}} \tag{4}$$

where η_0 is tuned by hand, ϵ is a small number that avoids division by zero (e.g. 1e-8), and G_m is the sum of the squares of the gradients for parameter m

over time steps $1, \ldots, t-1$. Thus, after each SGD step, we update G_m as below:

$$G_m \rightarrow G_m + \left(\frac{\partial J_i(\boldsymbol{\theta})}{\partial \theta_m} \right)^2 \quad (5)$$

RMSProp The AdaGrad learning rates decrease over time and can become quite small later in training. RMSProp deals with this issue by defining g_m to be an exponentially decaying sum of the squares of the gradients seen for parameter m.

$$G_m \rightarrow \gamma G_m + (1-\gamma) \left(\frac{\partial J_i(\boldsymbol{\theta})}{\partial \theta_m} \right)^2 \quad (6)$$

Geoff Hinton introduced this unpublished technique and suggests $\gamma = 0.9$ and $\eta_0 = 0.001$. Note that in our sparse setting, it would be preferable only to update G_m lazily whenever we have a non-zero feature m in our current example i. Doing so requires that we keep track of when G_m was last updated and then apply the updates we skipped all at once. For example, if it has been s iterations since we last updated θ_m, then we would first update $G_m \rightarrow \gamma^s G_m$ since the other terms in the updates we skipped were all zero.

Normalized Gradient Descent Ross et al. (2013) introduce Normalized online Gradient Descent (NG). This provides a technique for online normalization of features that can even be successful when binary features are used. It has also been shown to be complementary to AdaGrad, despite their similarities. Algorithm 1 of Ross et al. (2013) describes the algorithm in detail.

Other Optimization Algorithms Sebastian Rudder's blog[5] provides a very concise summary of a variety of other optimization algorithms such as Adam, SGD with Momentum, and Nesterov Accelerated Gradient (NAG). These effective techniques could be considered, but their application to our setting with sparse updates could be quite tricky given the time constraints.

5.4 Experiments

Successful approaches to this task will require tuning of parameters and a variety of experiments. It may be helpful to devote one team member's time to running experiments and analyzing the results.

[5] `http://sebastianruder.com/optimizing-gradient-descent/`

6 Discussion

Supervised classification is one of the simplest forms of a learning problem that we'll discuss during the summer school. Even in this case, however, the learning problem is not always easy and requires careful empirical work in order to get good results.

References

Léon Bottou. 2012. Stochastic gradient descent tricks. In *Neural Networks: Tricks of the Trade*. Springer.

John Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research (JMLR)*.

Shervin Malmasi and Mark Dras. 2015. Language identification using classifier ensembles. In *Proceedings of the Joint Workshop on Language Technology for Closely Related Languages, Varieties and Dialects (LT4VarDial)*, pages 35–43.

Stephane Ross, Paul Mineiro, and John Langford. 2013. Normalized Online Learning. In *UAI*. arXiv: 1305.6646.

Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. 2009. Feature hashing for large scale multitask learning. In *Proceedings of the International Conference on Machine Learning (ICML)*.

Marcos Zampieri, Liling Tan, Nikola Ljubeic, Jörg Tiedemann, and Preslav Nakov. 2015. Overview of the dsl shared task 2015. In *Proceedings of the Joint Workshop on Language Technology for Closely Related Languages, Varieties and Dialects (LT4VarDial)*, pages 1–9.