The VW FAQ

Pushpendre Rastogi
v1: 2 Feb 2021
last update: February 4, 2021

Contents

1 Which ML problems does VW handle?

VW can handle the following tasks for online learning and contextual bandits:

- Importance weighted Multiclass classification: Here the label is an integer encoded class along with a weight. the weight has default value of 1. when not specified.
- Multilabel classification: Here the label is a comma separated list of integer encoded class variables.
- Cost sensitive classification: Here the label is a space separated list of pairs. Each pair has two values and a colon in between. The class and the cost of predicting that class. Higher costs are worse.
- Contextual Bandits: The label is a space-delimited-list of colon-delimited-triples. Each triple specified the action taken, the cost incurred, and the probability which the exploration policy used to choose that action.
- Contextual Bandit Evaluation: In evaluation mode we are evaluating a new policy against the logging data collected by a logging policy. Here the label is `action (<action>:<cost>:<probability>)*`
- Conditional Contextual Bandits (CCB):
- Continuous arms
- Slates

2 What are Filter Trees / Error Correcting Tournaments? How can I train a tree online?

2.1 ECT

The filter tree/error correcting tournament was designed by Beygelzimer, Langford, and Ravikumar et al. in https://arxiv.org/pdf/0902.3176v4.pdf. They are useful (only) if you have a very big number of output labels (classes), let's say N=1000. You can imagine Filter trees (which are the basis of ECT) as a decision tree where in each node you ask whether the example belongs to one set of labels or another set of labels (using all the features, unlike original decision trees). With N¿1000, OAA is too slow (and even the accuracy is low), you should try ECT (or `--log_multi` or `--csoaa_ldf` in VW, if you can preselect a smaller number of labels which are relevant for each example). Three really useful references are

1. Slides
2. SO
3. arxiv paper

2.2 Logarithmic Time Online Multiclass prediction: LOMTree

https://arxiv.org/pdf/1406.1822.pdf

2.3 Contextual Memory Trees

https://arxiv.org/pdf/1807.06473.pdf

3 What is a learning reduction?

A learner - i.e. a reduction - is a akin to a transformer in a scikit learn pipeline, but its more general. A reduction in VW can do feature pre-processing, but it can also transform the labels, the link function, and the loss function to change one problem into another.

DETOUR: Meanings of some common abbreviations

csoaa Cost sensitive One Against All

adf Action dependent feature

ldf Label dependent feature

See https://github.com/VowpalWabbit/vowpal_wabbit/wiki/What-is-a-learner%3F

For example, here's the pseudo-code for the csoaa algorithm.

Initial predictor $f_1(x)$
for t in 1 to T **do**
  Observe $\{x_{t,i}\}_{i=1}^K$. The i indexes over actions, so basically these are action dependent features.
  Predict class $i_t = \arg\min_{i=1}^K f_t(x_{t,i})$
  Observe costs $\{c_{t,i}\}_{i=1}^K$
  Update f_t using online least-squares regression on data $\{x_{t,i}, c_{t,i}\}_{i=1}^K$
end for

4 How does Learning to Search and the Credit Assignment work in VW?

The learning to search sub-system basically is a system for imitation learning where we have an episodic sequential decision problem, and annotated gold-standard outputs are available for every step of the sequential decision problem, and we want to avoid the gap between training-and-testing where at training time we provide perfect trajectories to the imitation learner, but at test time, the classifier will have to rely on possibly wrong intermediate predictions.

The L2S system depends on some base algorithms for imitation learning.

4.1 Imitation Learning Base algorithms

The oldest one is Searn (2006), and then Dagger (2011), and then Aggrevate (2014), and finally Lols(2015). references are linked here. For example, the LOLS algorithm works as follows in a slightly unrealistic setting.

given a dataset of labeled examples and $\beta \geq 0$ which is a mixture parameter. The labels suggest a reference policy called π^{ref}. For example, when we are doing POS-tagging the reference policy outputs whatever label was assigned to a word.
for instance i in 1 to N **do**
  Initialize Γ. It is an example specific "augmented" dataset used to improve our current learnt estimate of the behavior policy.
  Let $\hat{\pi}_1$ be our guess of some good initial policy.
  for all t in 0 to $T-1$ **do**
    Roll-in by executing $\hat{\pi}_i$ for t rounds to reach s_t
    Generate a feature vector $\phi_{i,t}$ that describes the ith example at step t.
    for all actions a that are possible at this step **do**
      take action a
      pick whether to use the reference policy π^{ref} or $\hat{\pi}_i$ with probability β.
      Execute the picked policy for the remaining $T-t-1$ steps. and note does the cost $c_{i,t}(a)$
    end for
    Now we have a vector of observed costs $c_{i,t}(a)$ and feature vector $\phi_{i,t}$.
    Add this full-feedback example to Γ.
  end for
  Update $\hat{\pi}_i$ using examples from Γ to generate $\hat{\pi}_{i+1}$
end for
Return an ensemble of policies from $\hat{\pi}_1, \ldots, \hat{\pi}_{N+1}$

The LOLS algorithm can be modified to the structured contextual bandit setting as well. See Algorithm 2 in the LOLS paper. It's a small modification where bandit-feedback is collected for an example, i.e. instead of gathering T full-feedback examples, only 1 bandit-feedback example is gathered.

The gist is that "roll-in, roll-out" will learned policy is bad, and "roll-in" with reference policy is bad, therefore the only choice is "roll-in" with learned policy and "roll-out" with a mix of reference policy and learned policy.

4.2 Credit Assignment Compiler

The Credit assignment compiler is just a general interface for implementing imitation learning algorithms, its like an api, and a few common optimizations that are packaged together.

The basic idea is that if we have a policy that executes episodically and we have true-reference labels available then we can do imitation learning of such a sequential execution using the following tricks. Basically it wraps two method *PREDICT* and *LOSS* and passes it to the runtime function *MYRUN*.

For every time step, it generate a single cost-sensitive classification example; its features are $ex[t0]$, and there are $M(ex[t0])$ possible labels (=actions).

General Learn To Search: The Credit Compiler

function MYRUN(X, PREDICT, LOSS)
for t in 1 to $len(X)$ **do**
  $Y[t] := PREDICT(x, y, tag, condition)$
end for
return LOSS, Y

function LEARN
$T := 0$
$PREDICT2(x, y) := \{T++; ex[T-1] := x; cache[T-1] = F(x, y, rollin)\}$
eexcute $MYRUN(X, PREDICT2, LOSS)$
for t' in 1 to T **do**
  Let $t = 0$
  for a_0 in 1 to $A(ex[t_0])$ **do**
    $PREDICT2 = \{t++; return...\}$
    $LOSS2 := \{losses[a_0] += val$
    $MYRUN(X, PREDICT2, LOSS2)$
  end for
end for

5 How do interactively run VW?

6 How do I inspect the trained model?

7 How do I run VW for a specific ML problem?

https://github.com/VowpalWabbit/vowpal_wabbit/wiki/Input-format

8 How can I use VW for online learning?

9 What model classes does VW implement for online learning?

10 What are invariant, normalized and adaptive learning rates in VW?

10.1 Invariant

Invariant means that the function learnt using importance weights will be the same as duplicating the samples in the input. For example, if we have a sample and an importance weight of 2 $(x, y), 2$ then the model that we learn should be the same as feeding the example twice to the SGD learning algorithm

11 What is the easiest way to use VW? Some example commands?

Here are some examples commands from http://www.machinedlearnings.com/2013/02/one-louder.html

```
% less Makefile
SHELL=/bin/zsh
CXXFLAGS=-O3

.SECONDARY:

all:

%.check:
        @test -x "$$(which $*)" || {                              \
          echo "ERROR: you need to install $*" 1>&2;              \
          exit 1;                                                 \
        }

dna_train.%.bz2: wget.check
        wget ftp://largescale.ml.tu-berlin.de/largescale/dna/
    dna_train.$*.bz2

quaddna2vw: quaddna2vw.cpp

quaddna.model.nn%: dna_train.lab.bz2 dna_train.dat.bz2 quaddna2vw
    vw.check
        time paste -d' '                                          \
            <(bzcat $(word 1,$^))                                 \
            <(bzcat $(word 2,$^) | ./quaddna2vw) |                \
          tail -n +1000000 |                                      \
        vw -b 24 -l 0.05 --adaptive --invariant                   \
          --loss_function logistic -f $@                          \
          $$([ $* -gt 0 ] && echo "--nn $* --inpass")

quaddna.test.%: dna_train.lab.bz2 dna_train.dat.bz2 quaddna.model.%
     quaddna2vw vw.check
        paste -d' '                                               \
          <(bzcat $(word 1,$^))                                   \
          <(bzcat $(word 2,$^) | ./quaddna2vw) |                  \
        head -n +1000000 |                                        \
        vw -t --loss_function logistic -i $(word 3,$^) -p $@

quaddna.perf.%: dna_train.lab.bz2 quaddna.test.% perf.check
        paste -d' '                                               \
          <(bzcat $(word 1,$^))                                   \
          $(word 2,$^) |                                          \
        head -n +1000000 |                                        \
        perf -ROC -APR
```

12 What are the VW input options

An even more comprehensive list of arguments is here https://github.com/VowpalWabbit/vowpal_wabbit/wiki/Command-line-arguments

Option	Description
Input Options	
`-d, -data <f>`	The file to read data from, If multiple flags are given then all files will be read.
`-daemon`	Run as a server
`-port <p>`	The port for the server.
`-passes <n>`	Number of passes to use for online learning.
`-c [-cache]`	Tell VW to build a cache of input features (or to use one if it exists)
`-cache_file <fc>`	Use the fc cache file. Multiple: Use all, Missing: Create. Multiple+Missing: Concatenate
`-compressed`	Gzip compress the cache file.
Output Options	
`-p <po>`	`predictions` File to dump predictions into.
`-r <ro>`	File to output unnormalized predictions into.
`-sendto <host:port>`	Relay the input examples to host:port
`-audit`	Print detailed information about (feature name, feature index, feature value, weight value)
Example Featurization and Ommission Options	
`-t [-testonly]`	
`-q <ab>`	`[-quadratic]` Cross every feature in namespace $a*$ ($*$ is a wildcard) with every feature in namespace $b*$
`-ignore <a>`	Remove a namespace and all features in it.
`-noconstant`	
`-sort_features`	Sort features for small cache files.
`-ngram <N>`	Generate $N-$grams.
`-skips <S>`	Generate N-grams with S skips.
`-hash all`	hash even integer features.
Learning Rate/Update Rule Options. See gd.cc	
`<d=1>`	`decay_learning_rate`
`-initial_t <i=1>`	The initial time step
`-power_t <p=0.5>`	The power used to anneal the learning rate. If we are trying to track a changing system then use `power_t = 0` if its really iid then `power_t = 1` else 0.5.
`-l <l=10>`	The learning rate
`-loss_fuction`	squared, logistic, hinge etc. The "classic" loss function is actually squared but it does basic "sgd". This option has been renamed to a flag like "sgd".
Learn Weight Options.	
`-b <b=18>`	`bit_precision`. Essentially the number of bits used for hashing. This can be thought of as the log of the total number of features. But if we actually start filling this completely then there will be a lot of collisions.
`-i <ri>`	The initial regressor weight value. Multiple $\implies$ Average. This seems to be the same as `-initial_weight`
`-f`	file to store final weight values in.
`-readable_model`	As $-f$, but in text.
`-save_per_pass`	Save the model after every pass over data.
`-random_weights <r>`	Make initial weights random. Particularly this is useful for LDA.
`passes`	
`holdout_off`	
`holdout_period`	
`noconstant`	
Contextual Bandit Options, including warm-start. Detail wiki	
`cb_type`	These specify the policy evaluation approach. E.g. `ips, dr, dm, mtr`. `mtr` works with `cb_adf` but hasnt been implemented for `cb` `mtr` may be the best method for learning a policy.
CB Learning	`cb, cb_explore, cb_explore_adf, cb_adf`. The `cb` contextual bandit module allows us to optimize predictor based on already collected data, or contextual bandits without exploration. The basic `cb` algorithm reduces the problem to cost-sensitive multiclass classification when we only observe costfor 1 action/example. OTOH the cb_explore method is used when the maximum number of actions is known ahead of time. The cb_explore_adf is most general and allows the set of actions to change and to have rich action dependent features for each action. The ADF algorithm builds upon the label dependent features learner inside VW. See https://hunch.net/~jl/interact.pdf. The explore flags are online learning and the non-explore flags are off-policy learning.
CB Exploration	`first, epsilon, bag, cover, softmax`. The default is epsilon-greedy. The cover algorithm is not supported in the ADF mode.
Settings for Basic and Cost-sensitive Multiclass Classification	
`oaa`	one against all
`ect`	error correcting tournament / filter tree.
`csoaa`	**Default cost-sensitive classifier**. Cost sensitive one against all (reduction to regression)
`wap`	Weighted all pairs (reduction to weighted binary classification)

13 What is the multiline LDF format?

Copied from http://users.umiacs.umd.edu/~hal/tmp/multiclassVW.html

suppose you have four document labels (inspired by a dataset due to Andrew McCallum) that talk about automobiles or airplanes. But some are talking about real automobiles (or airplanes) and some are talking about toy automobiles (or airplanes). Thus the four labels are: RealAuto ToyAuto RealAir ToyAir (call them 1, 2, 3, 4 in that order). You might believe that it's actually useful to share parameters across the classes. For example, when you have a basic feature "f" on an example with label RealAuto, you might want to have three version of "f" available for classification: "Real_f", "Auto_f" and "RealAuto_f". If you use the normal support for cost-sensitive classification, you'd only get the last of these.

The ldf format was invented to cope with problems like this. The easiest way to specify ldf example is in a multiline format. This means that each label gets its own line in the input file, and blank lines separate examples. For instance, you could have something like the following as a valid two-example dataset for the above problem:

```
1:0 | Real_f Auto_f RealAuto_f
2:1 | Toy_f  Auto_f ToyAuto_f
3:1 | Real_f Air_f  RealAir_f
4:1 | Toy_f  Air_f  ToyAir_f

1:1 | Real_g Auto_g RealAuto_g
2:1 | Toy_g  Auto_g ToyAuto_g
3:0 | Real_g Air_g  RealAir_g
4:1 | Toy_g  Air_g  ToyAir_g
```

If this data is in csldf.txt, you can train with:

`vw --csoaa_ldf multiline < csldf.txt`

If you get a warning about ring size when running in ldf mode, it means that you have a multiline example that contains more lines than VW is willing to hold in memory at once. You can fix this by adding "–ring_size 1024" or something like that (the default is 256). Just make sure this number is greater than the maximum number of labels for any given example. In ldf mode, having shared features for WAP or classifier-based OAA is useless: the predictions are all based on differences of feature vectors, and shared features will cancel each other out.

14 How to warm-start VW?

https://github.com/VowpalWabbit/vowpal_wabbit/wiki/Warm-starting-contextual-bandits https://arxiv.org/pdf/1901.00301.pdf

15 How do Normalized, Adaptive, Invariant updates in VW work?

Normalized, adaptive, and invariant updates are improvements over classic SGD for online learning that were built over the course of two papers.

1. Normalized online learning by Ross, Mineiro and Langford
2. Online Importance Weight Aware Updates by Karampatziakis and Langford

The basic normalized SGD works as follows

NORMALIZED SGD

Initially $w_i = 0, s_i = 0, N = 0$
for each timestep t **do**
  for each i, if $|x_i| > s_i$ **do**
    $s'_i := |x_i|; w_i := w_i s_i^2 / s'^2_i; s_i := s'_i$
  end for
  $\hat{y} = \sum_i w_i x_i$
  $N := N + \sum_i \frac{s_i^2}{s_i^2}$
  for each i **do**
    $w_i := w_i - \eta_t \frac{t}{N} \frac{1}{s_i^2} \frac{\partial L(\hat{y}, y)}{\partial w_i}$
  end for
end for

The adaptive variant of this update rule works by accumulating the square of the gradient, also it linearly scales the weights instead of as a square of the ratio.

NORMALIZED ADAPTIVE SGD

Initially $w_i = 0, s_i = 0, G_i = 0, N = 0$
for each timestep t **do**
  for each i, if $|x_i| > s_i$ **do**
    $s'_i := |x_i|; w_i := w_i s_i / s'_i; s_i := s'_i$
  end for
  $\hat{y} = \sum_i w_i x_i$
  $N := N + \sum_i \frac{s_i^2}{s_i^2}$
  for each i **do**
    $g_i := G_i + \left(\frac{\partial L(\hat{y}, y)}{\partial w_i}\right)^2$
    $w_i := w_i - \eta_t \sqrt{\frac{t}{N}} \frac{1}{s_i \sqrt{G_i}} \frac{\partial L(\hat{y}, y)}{\partial w_i}$
  end for
end for

DETOUR

Reasoning behind dividing by square of the scale
The input features have units. Let x_i be the base unit of feature i. To maintain the same prediction, when we double the feature i, we must halve the weight w_i, therefore w_i has unit $1/x_i$. But the gradient has the unit of x_i (this is most easily seen for linear predictors $\langle w \cdot x \rangle$. Therefore when we do the update we are adding $1/x_i + x_i$ unitwise which is not good.
Therefore to handle this we do normalization since x_i become close to 1 so units are less of an issue. And we choose x_i on a similar scale to x_j so unit mismatch across features doesn't kill us. But the basic fix to this is to create per feature learning rates that divide by the square of the average gradients. Now the square of the average gradients create their own problem so we again have to normalize.

The importance weighted updates with invariance property were defined in the paper "Online Importance Weight Aware Updates" https://arxiv.org/pdf/1011.1576.pdf and they work as follows. The assumption is that our predictor is linear – maybe followed by a link function – but mainly linear. The loss function maybe "Squared" or "Logistic", "Hinge", "Logarithmic" or "Expoenntial" or some other loss function. But the main thing is that the loss function takes the hypothesis $p = x^t w$ and a true value y. Let $g(p) = \left.\frac{\partial l}{\partial p}\right|_{p,y}$ Because of this composition the gradient of the loss for a single example will be $g(p) \times x$. Now say that we present the same example (x, y) two times starting from parameter w_1 then the final value w_3 will be computed as

$$w_2 = w_1 - \eta g(w_1^T x) x \quad (1)$$
$$w_3 = w_2 - \eta g(w_2^T x) x \quad (2)$$

Therefore if we repeat this process k times then $w_{k+1} = w_k - \eta g(w_k^T x) x = w_1 - x\eta \sum_{i=1}^k g(w_i^T x)$.

Let $s(k) = \eta \sum_{i=1}^k g(w_i^T x)$. Therefore we can say that

$$s(k+1) = s(k) + \eta g\big((w_1 - x s(k))^T x\big)$$

. This is the core difference equation at the heart of the technique. They then generalize this difference equation to an ODE and also propose to handle varying learning rates to get

$$s'(h) = \eta_t(h) g\big((w_t - s(h) x_t)^T x\big)$$

15.1 How does everything work together?

The two types of updates are intricately merged to work together. The basic idea is that the importance weight aware methods do not ultimately change the direction of the gradient. They just make sure that the updates for large importance weights satisfy the invariance property and that they do not lineary scatter away.

```cpp
struct norm_data
{
  float grad_squared;
  float pred_per_update;
  float norm_x;
  power_data pd;
  float extra_state[4];
};

float compute_update(gd& g, example& ec) {
    float pred_per_update = sensitivity<sqrt_rate, feature_mask_off
    , adaptive, normalized, spare, false>(g, ec);
    float update_scale = get_scale<adaptive>(g, ec, ec.weight);
    if(invariant)
      update = all.loss->getUpdate(ec.pred.scalar, ld.label,
    update_scale, pred_per_update);
}

float get_scale(gd& g, example& /* ec */, float weight)
{
  float update_scale = g.all->eta * weight;
}
// For squared loss.
float getUpdate(float prediction, float label, float update_scale,
    float pred_per_update)
  {
    return (label - prediction) * (1.f - correctedExp(-2.f *
    update_scale * pred_per_update)) / pred_per_update;
  }
```

16 How to benchmark contextual bandits

COBA

Bakeoff paper

CB Bakeoff

16.1 Example of benchmarking REGCB and SquareCB

https://github.com/VowpalWabbit/vowpal_wabbit/wiki/Contextual-Bandit-Exploration-with-SquareCB https://arxiv.org/pdf/1803.01088.pdf

http://proceedings.mlr.press/v119/foster20a/foster20a.pdf

https://arxiv.org/pdf/2003.12699.pdf

17 How to modify VW?

VW is written in CPP and it is written using a some common idioms with performance in mind. Things like shared pointers, destructors and move constructors, adn the boost library are heavily used.

17.1 Common coding patterns in VW

TODO

17.1.1 How does the reduction stack work?

Stack of reductions for every vw run is defined by 2 things: 1) DAG of dependencies that are defined in setup function for every reduction. E.g. if we have cb_explore_adf reduction included, we also include cb_adf one link. 2) Topological ordering of the DAG of all reductions defined here, so the final stack of reduction for each VW run is actually sub-stack of 2) that contains a) reductions provided in the command line, b) reductions defined in input model file (if any), c) reductions populated as dependencies.

all setup functions of all "reductions" call into setup_base themselves (with the exception of the reductions that are base reductions, i.e. they know they should be the last one on the stack, like gd and ftrl). So if you browse around on most setup functions you should see a call into setup_base and then that result gets used to construct the next learner layer.

```
vw -d debug.txt --foreground --ccb_explore_adf \
  --cb_type mtr --epsilon 0.01 --ftrl -f debug.model

[Total Enabled reductions]
ftrl, scorer, csoaa_ldf, cb_adf, cb_explore_adf_greedy,\
  cb_sample, shared_feature_merger, ccb_explore_adf

[Provided explicitly]
ccb_explore_adf, ftrl

[Dependencies]
ccb_explore_adf -> cb_sample
ccb_explore_adf -> cb_explore_adf_greedy -> \
   cb_adf -> csoaa_ldf
```

Files

ccb_explore_adf

17.2 Illustrative PRs

18 How are the policies in VW parameterized?

From the Section 2, paragraph on parameterization in the Bakeoff arxiv version https://arxiv.org/pdf/1802.04064.pdf

We consider linearly parameterized policies taking the form $\pi(x) = \arg\min_a \theta_a^T x$ or in the case of IWR reduction, regressors $f(x, a) = \theta_a^T x$. For DR loss estimator, the loss estimators is also linear, and this can be extended to more complex action dependent features for different actions. in which case the parameterization is of the form $f(x, a) = \theta^T x_a, \hat{l}(x, a) = \phi^T x_a$. We consider linearly parameterized policies