Vowpal Wabbit

http://hunch.net/~vw/

git clone
git://github.com/JohnLangford/vowpal_wabbit.git

What is Vowpal Wabbit?

1. fast/efficient/scalable learning algorithm.
2. vehicle for rule-breaking tricks. Progressive validation, Hashing, Log-time prediction, Allreduce, ...
3. combinatorial learning algorithm.
4. Open Source project. BSD license, ~ 10 contributors in the last year, >100 mailing list. Used by (at least) Amazon, AOL, eHarmony, Facebook, IBM, Microsoft, Twitter, Yahoo!, Yandex.
5. Used for Ad prediction, document classification, spam detection, etc...

Combinatoric design of VW

1. Format {binary, text}
2. IO { File, Pipe, TCP, Library }
3. Features {sparse, dense}
4. Feature {index, hashed} with namespaces
5. Feature manipulators {ngrams, skipgrams, ignored, quadratic, cubic}
6. Optimizers {online, CG, LBFGS} parallelized
7. Representations {linear, MF, LDA}
8. Sparse Neural Networks by reduction.
9. Losses {squared, hinge, logistic, quantile}
10. Multiclass {One-Against-All, ECT}
11. Cost-sensitive {One-Against-All, WAP}
12. Contextual Bandit {Ips, Direct, Double Robust}
13. Structured { Imperative Searn, Dagger}
14. Understanding { l1, audit, Prog. Validation}

An example application might use

1. Format {binary, text}
2. IO { File, Pipe, TCP, Library }
3. Features {sparse, dense}
4. Feature {index, hashed} with namespaces
5. Feature manipulators {ngrams, skipgrams, ignored, quadratic, cubic}
6. Optimizers {online, CG, LBFGS} parallelized
7. Representations {linear, MF, LDA}
8. Sparse Neural Networks by reduction.
9. Losses {squared, hinge, logistic, quantile}
10. Multiclass {One-Against-All, ECT}
11. Cost-sensitive {One-Against-All, WAP}
12. Contextual Bandit {Ips, Direct, Double Robust}
13. Structured { Imperative Searn, Dagger}
14. Understanding { l1, audit, Prog. Validation}

An example

An adaptive, scale-free, importance invariant update rule.

Example: `vw -c rcv1.train.raw.txt -b 22 --ngram 2 --skips 4 -l 0.25 --binary` provides stellar performance in 12 seconds.

Learning Reductions

The core idea: reduce complex problem A to simpler problem B then use solution on B to get solution on A.

Problems:

1. How do you make it efficient enough?
2. How do you make it natural to program?

The Reductions Interface

```
void learn(void* d, learner& base, example* ec)
{
  base.learn(ec); // The recursive call
  if ( ec->final_prediction > 0) //Thresholding
    ec->final_prediction = 1;
  else
    ec->final_prediction = -1;
  label_data* ld = (label_data*)ec->ld;//New loss
  if (ld->label == ec->final_prediction)
    ec->loss = 0.;
  else
    ec->loss = 1.;
}
```

```
learner* setup(vw& all,
                    std::vector<std::string>&opts,
                    po::variables_map& vm,
                    po::variables_map& vm_file)
{ //Parse and set arguments
   if (!vm_file.count("binary"))
   {
      std::stringstream ss;
      ss « " –binary ";
      all.options_from_file.append(ss.str());
   }
   all.sd->binary_label = true;
   //create new learner
   return new learner(NULL, learn, all.l);
}
```

Searn/Dagger: Structured prediction algorithms

The basic idea: Define a search space, then learn which steps to take in it.

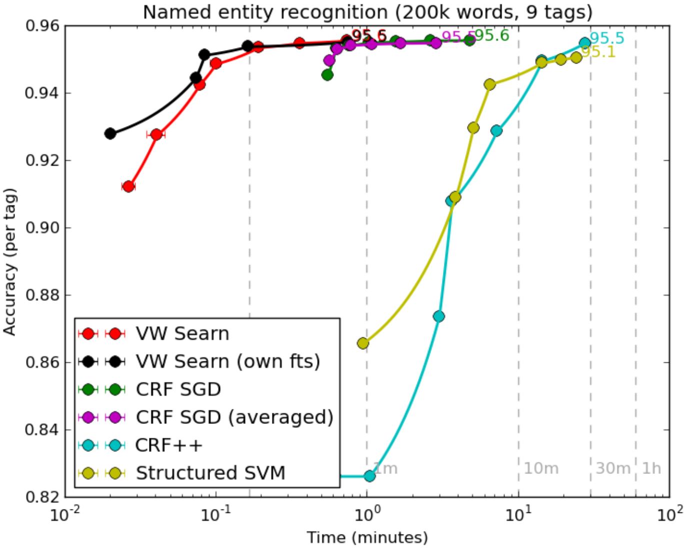
1. A method for compiling global loss into local loss.
2. A method for transporting prediction information from adjacent predictions.

Demonstration: wget

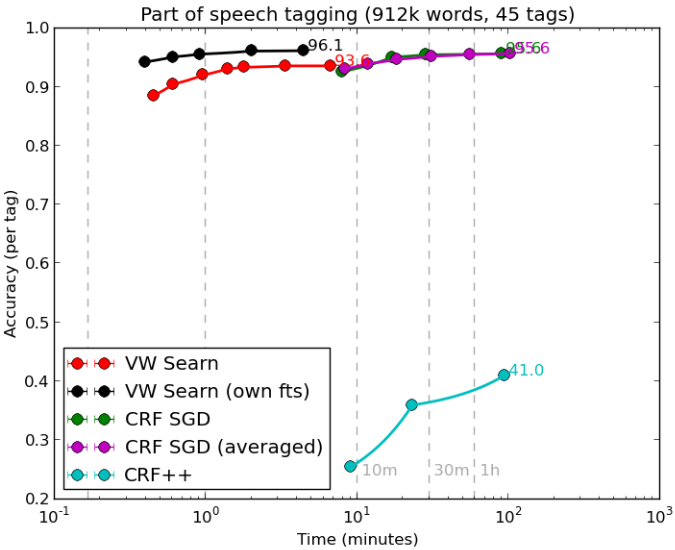
http://hal3.name/tmp/pos.gz

vw -b 24 -k -c -d pos.gz --passes 4 --searn_task sequence --searn 45 --searn_as_dagger 1e-8 --holdout_after 38219 --searn_neighbor_features -2:w,-1:w,1:w,2:w --affix -3w,-2w,-1w,+3w,+2w,+1w

This really works



This really works, part II



Imperative Searn (or Dagger)

```
void structured_predict(searn& srn, example**ec, size_t len)
{
  v_array<uint32_t> * y_star = srn.task_data;

  for (size_t i=0; i<len; i++)
    {


    //Prediction with advice.
    label_to_array(ec[i]->ld, *y_star);
    size_t pred = srn.predict(ec[i], NULL, y_star);




    }

}
```

Imperative Searn (or Dagger)

```
void structured_predict(searn& srn, example**ec, size_t len)
{
  v_array<uint32_t> * y_star = srn.task_data;
  float total_loss = 0;
  for (size_t i=0; i<len; i++)
    {



      //Prediction with advice.
      label_to_array(ec[i]->ld, *y_star);
      size_t pred = srn.predict(ec[i], NULL, y_star);
      //track loss
      if (y_star->size() > 0)
        total_loss += (pred != y_star->last());
    }//declare loss
  srn.declare_loss(len, total_loss);
}
```

Imperative Searn (or Dagger)

```
void structured_predict(searn& srn, example**ec, size_t len)
{
  v_array<uint32_t> * y_star = srn.task_data;
  float total_loss = 0;
  for (size_t i=0; i<len; i++)
    { //save state for optimization
    srn.snapshot(i, 1, &i, sizeof(i), true);
    srn.snapshot(i, 2, &total_loss, sizeof(total_loss), false);
    //Prediction with advice.
    label_to_array(ec[i]->ld, *y_star);
    size_t pred = srn.predict(ec[i], NULL, y_star);
    //track loss
    if (y_star->size() > 0)
      total_loss += (pred != y_star->last());
    }//declare loss
  srn.declare_loss(len, total_loss);
}
```

The Rest

1. Zhen Qin
2. Paul Mineiro
3. Nikos Karampatziakis