Factorization Machines

Written by Matthew B. Wilson, August 25, 2015

The data science team at AdRoll is constantly working to improve our programmatic bidding algorithm, BidIQ. One recent improvement to BidIQ has been the introduction of a novel modeling technique called Factorization Machines (FM). The FM model allows us to consider interactions between all predictive variables, rather than only certain manually selected interactions. Thanks to the FM model, we are able to use the exact same feature information to more accurately value every potential impression that comes our way. As a result, our advertisers are collecting almost 7% more clicks for only 2% more cost.

To motivate the FM model, let's consider a simplified ad bidding algorithm that bids using three variables: web domain, advertiser, and user location. In the first iteration of our modeling, we would learn a weight w_i for each web domain, advertiser, and user location. Our bid, B_{linear}, would then be based on a function f of a linear combination of the weight vector $\vec{w}$. That is:

$$B_{linear} = f\left(w_0 + \sum_{i=1}^{n} w_i x_i\right)$$

where w_0 is an intercept term and x_i is the value of the ith feature. This modeling technique proved extremely powerful, but the drawback was that the model only learned the effect of our three variables individually rather than in combination. But what if an advertiser's ads perform particularly well on some specific domains? What if users in San Francisco are much more interested in some advertisers than others? What if a particular domain tends to receive exceptionally valuable traffic from New York? One popular way of solving this problem at scale is to supplement the standard linear model equation with an additional term to model pairwise feature interactions.

The simplest such strategy is to learn a weight w_{ij} for each feature combination. We would then make our bid, $B_{quadratic}$, according to:

$$B_{quadratic} = f\left(w_0 + \sum_{i=1}^{n} w_i x_i + \sum_{i=1}^{n} \sum_{j=i+1}^{n} w_{ij} x_i x_j\right)$$

where $\mathbf{W} \in \mathbb{R}^{n \times n}$ are additional parameters to be learned. Unfortunately, this naïve approach will not work for two main reasons. First, the size of the model is now $\mathcal{O}(n^2)$, which has terrible implications for both the amount of memory needed to store the model, and the time it takes to train the model. Second, our dataset is too sparse for us to learn the all of the weights $\mathbf{W}$ reliably. That is, for almost all pairs (i, j) we would not have enough training examples to learn the weight w_{ij} well. To improve the pairwise feature interaction modeling of $B_{quadratic}$, we could include only the weights w_{ij} with corresponding features sufficiently dense and informative. In the second iteration of our modeling, we did just this. The problem is selecting the valuable weights w_{ij} is difficult to do algorithmically, and does not scale to a large number of possible feature combinations.

FM solves the problem of considering pairwise feature interactions. Indeed, it allows us to bid based on reliable information from every pairwise combination of variables in the model. Just as important, FM allows us to do this in a remarkably efficient way both in terms of both time and space complexity. So how exactly does FM work? FM models pairwise feature interactions as the inner product of low dimensional vectors. More precisely, our bid with the FM model, B_{FM}, becomes:

$$B_{FM} = f\left(w_0 + \sum_{i=1}^{n} w_i x_i + \sum_{i=1}^{n} \sum_{j=i+1}^{n} \langle \vec{v}_i, \vec{v}_j \rangle x_i x_j\right)$$

where $\mathbf{V} \in \mathbb{R}^{n \times k}$ are additional parameters to be learned, and $\vec{v}_i$ is the ith row of $\mathbf{V}$. Notice that the FM model replaces the weights w_{ij} by $\langle \vec{v}_i, \vec{v}_j \rangle$. From a modeling perspective, this is powerful because each feature ends up embedded in an inner product space, with similar features embedded near one another. As a result, the FM model is even able to learn the effect of interactions between features which appear together very infrequently in the training data. Furthermore, the size of the FM model is now a more reasonable $\mathcal{O}(kn)$, where the latent dimension k is a hyperparameter of the FM model.

However, the FM model is not an obvious improvement from a computation time perspective. In fact, computation of the pairwise feature interaction term now *appears* to require $\mathcal{O}(kn^2)$ operations rather than the $\mathcal{O}(n^2)$ of the naïve interaction modeling solution. Yet this is not the case; after some manipulation we may rewrite the nonlinear FM term as follows (1):

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} \langle \vec{v}_i, \vec{v}_j \rangle x_i x_j = \frac{1}{2} \sum_{j=1}^{k} \left(\left(\sum_{i=1}^{n} v_{ij} x_i \right)^2 - \sum_{i=1}^{n} v_{ij}^2 x_i^2 \right)$$

The right hand side of this equation can clearly be computed in $\mathcal{O}(kn)$ time. This is the magic of FM: *we are able to compute the term that models all pairwise interactions in linear time.* As a result, we are able to train the FM model in a time proportional to the time needed to train the linear model.

Apart from the optimization described above, we used several other strategies to reduce training time and improve convergence. First, the FM model is trained using stochastic gradient descent (SGD), an algorithm known for its speed. (2) Second, we parallelized SGD using the lock-free so-called "HOGWILD!" scheme [2]. This lock-free, parallel SGD is notable because it avoids blocking any threads by allowing race conditions during model updates. Additionally, we found AdaGrad to be a very effective learning-rate schedule for training an FM model with SGD [3]. Finally, we used single instruction multiple data (SIMD) computation to vectorize calculations involving the matrix $\mathbf{V}$, quartering FM model training time. See below for a D programming language (pseudocode) example of the vectorized calculation of the FM model equation. With the optimizations described we are able to train an FM model in approximately the same amount of time it took to train our previous model. This is outstanding as the FM model is significantly more predictive.

```
class FMModel
{
    uint k; // FM latent dimension
    float w0;
    float[] w_vector;
    float4[][] v_matrix;

    // ... other FM model variables and functions ...

    float model_equation(Observation obs)
    {
        float linear_term = w0;
        foreach(Feature feat; obs)
            linear_term += w_vector[feat.id];

        float4 non_linear_term4 = 0;
        for(int j = 0; j < (k/4); j++)
        {
            float4 first_term4 = 0.toFloat4();
            float4 second_term4 = 0.toFloat4();
            foreach(Feature feat; obs)
            {
                float4 update = v_matrix[j][feat.id] * feat.weight;
                first_term4  += update;
                second_term4 += (update * update);
            }
            first_term4 *= first_term4;

            non_linear_term4 += (first_term4 - second_term4);
        }

        float non_linear_term = non_linear_term4.sum_float4();
        return 0.5 * (linear_term + non_linear_term);
    }
}
```

Moving forward, we continue to search for modeling and model training improvements. If Factorization Machines or machine learning are the types of things that interest you, please consider applying to work at AdRoll!

References

1. Rendle, Steffen. *Factorization Machines.*
2. Recht, Benjamin and Re, Christopher and Wright, Stephen and Niu, Feng. *Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent.*
3. Duchi, John and Hazan, Elad and Singer, Yoram. *Adaptive Subgradient Methods for Online Learning and Stochastic Optimization.*

Footnotes

1. For details, please see [1].
2. The main downside to using FM is the resulting optimization problem is no longer convex. As a result, many effective optimization techniques are no longer at our disposal when learning the parameters $\vec{w}$ and $\mathbf{V}$ of the FM model. Fortunately, SGD still works quite well in this non-convex setting.