Ben Fuqua

Supervised Learning: 5 Algorithms

“Learn the rules like a pro, so you can break them like an artist” -Pablo Picasso

You might have heard of the 10,000-hour rule, it goes like this: “It takes 10,000 hours of intensive practice to achieve mastery of complex skills and materials” (Gladwell). Learning to not only understand the rules artificial intelligence but learning to make it into an art will most definitely follow this 10,000-hour rule. This means diving deep into the algorithms behind the “intelligence”, seeing how they behave in certain scenarios, and, most definitely, trying it out for yourself. The idea of learning how to mold, shape, and perfect the art of artificial intelligence is the essence of this paper.

There are patterns and behaviors or “rules” to supervised learning algorithms. These must be understood before we can learn to “break them like an artist”. We will be examining 5 supervised algorithms so we can begin to learn and understand their rules. They are Decision Trees, Neural Networks, Boosting, Support Vector Machines (SVMs) and K-Nearest Neighbors (KNN). As we go through this paper, we will be exploring each of these models across 2 different datasets. These two datasets come from Kaggle and are called the “Product Subscription”, which we will call the 1st dataset, and “Bank Customer Churn” or the 2nd dataset. In order to evaluate how well a model is performing, we need to have a metric which is normally derived from a business case. For these two datasets we will propose a business case of “it is really bad if we predict more customers who churn/subscribe than those who didn’t” which would lead us to using the evaluation metric of precision. This case can be seen in companies like Hello Fresh who may over order food because they overestimated the number of people who would subscribe and now, they have food that will spoil.

Now that we have defined our models, datasets and a performance metric, we can talk about the areas of analysis. We will start by examining the different parameters used in each of the models and how they tend to behave. Next, we will compare models within each dataset to determine which model was best at optimizing for our business case or the precision metric. Then, we will compare the same model across our 2 datasets to see how each model behaves with different data. This analysis was done in the python language and utilized the Sci-kit Learn (Sklearn), Altair, and Pandas packages found within.

**Part 1: Model Behavior**

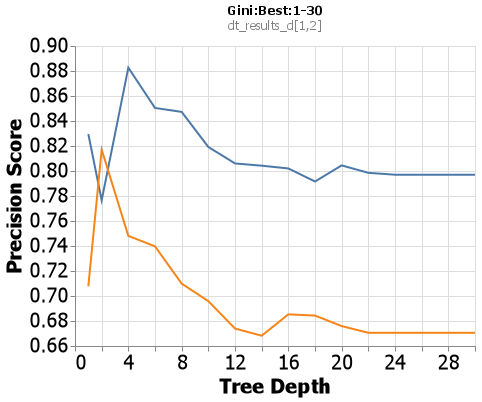
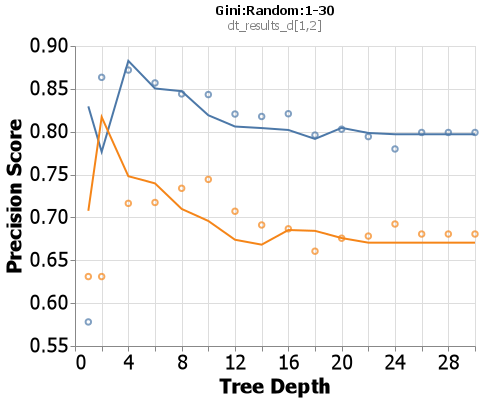
**Chart, scatter chart

Description automatically generated**Chart

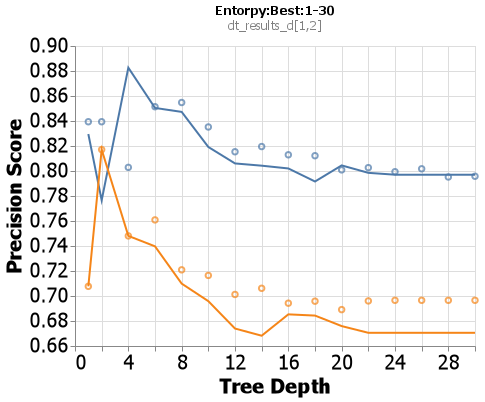
Description automatically generatedBalanced vs Unbalanced

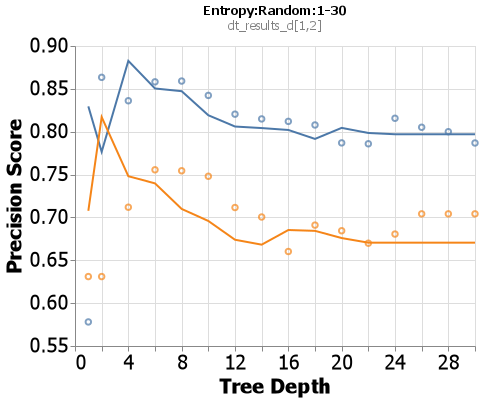
Before I go into the specific models, I want to touch on the importance of balancing a dataset. I feel this is important because it can drastically influence the outcome of how successful a model is at classification. On the graph to the right, we have a decision tree model that was trained on two datasets, the line on the bottom represents dataset 1 and it has a class balance of 90-10 and the top line represents dataset 2 and it had a class balance of 80-20. As you can see, when the model is more evenly split we tend to do better at classification. Of course, a 50-50 class balance isn’t always appropriate. You have to apply domain knowledge to your model and ask your self “Is it reasonable to say that these categories have the same probability of occurring?” and if the answer is no, a 50-50 split most likely won’t be your best course of action.

Decision Tree

 When examining the Sklearn decision tree algorithm I chose to manipulate 3 different hyperparameters: criterion with values of gini and entropy, splitter with parameters of best and random, and max\_depth with parameters 1 through 30 by increments of 2. This allows us to look at how a tree with varying lengths can learn when it is constrained to a small network vs a large network, and how the other parameters affect this ability. The default parameter combination for sklearn’s decision tree is criterion = gini, splitter = best and max\_depth = None. Now, a quick side note, max depth being set to None may not make sense, but it is essentially saying continue until all the final nodes are 100% one class, or until all the final nodes have 2 or less data points. With these default parameters being set we will treat the Gini:Best combination to be our base case with the blue line representing the first dataset and the orange line representing the second dataset and a line to denote the base case (see right).

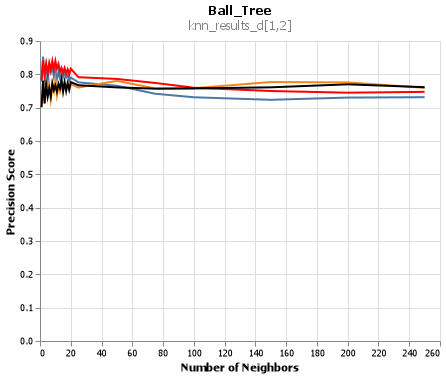
Now that we are familiar with our base case, we can put in another test case to compare, Gini:Random. When we look at depths 16 and on we see a little difference in the precision score, but no significant difference. When we look at 1-14 though, this is where the graph gets interesting. Here we changed the splitter parameter from Best to Random, this has caused the algorithm to not search for the best possible split on every single node. Once a node has been split well enough, it moves on. Within a depth of 1-6 we see a drastic difference, that is because the nodes on the trees haven’t had a chance to be optimized yet. But, when we look at 6-14 the Random splitter actually does a better job, most likely due to the better generalization of a random split then trying to go for the best gini impurity reduction.

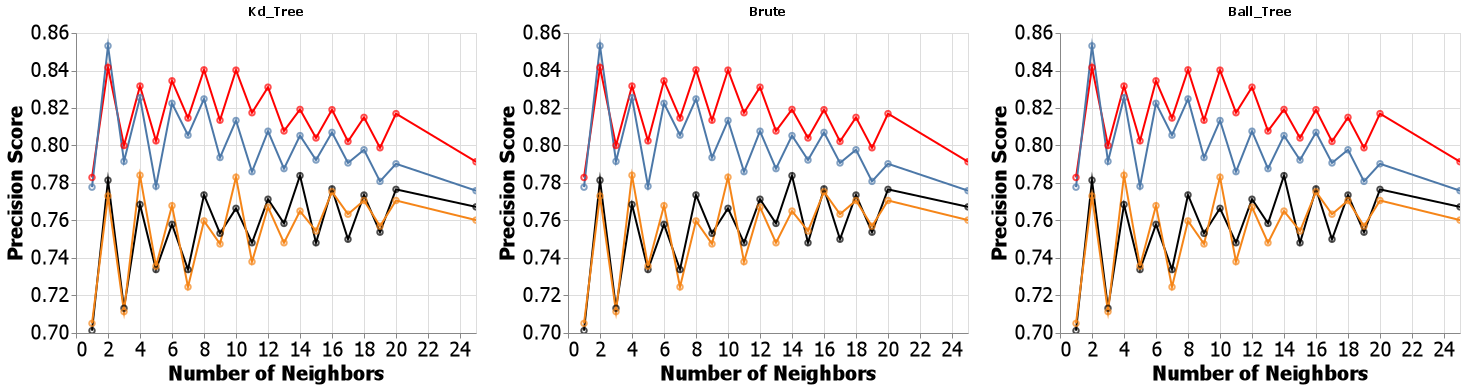
On to a new criterion: entropy. When compared to the base case, we see the same plateau after a depth of about 12. But what differs is between 2&6, if you look closely, you will see entropy does a better job at being more generalized earlier on. This is because the entropy formula is and the equation for gini is . Due to the log in the entropy equation, it makes it more generalized because when you take the log of a big number it becomes much smaller, thus leveling out the drastic differences and providing a better generalization.

For this last combination we see a much more sporadic chart. There isn’t a clear plateau like in the previous 3, and the beginning starts off a little bit more extreme as well. This is most likely caused by the fact that we are using a random splitter so our entropy impurity values aren’t optimized, thus giving us inconsistent types of splits. This adds on to the idea that entropy is trying to generalize, but it is being fed bad splits by the random splitter.

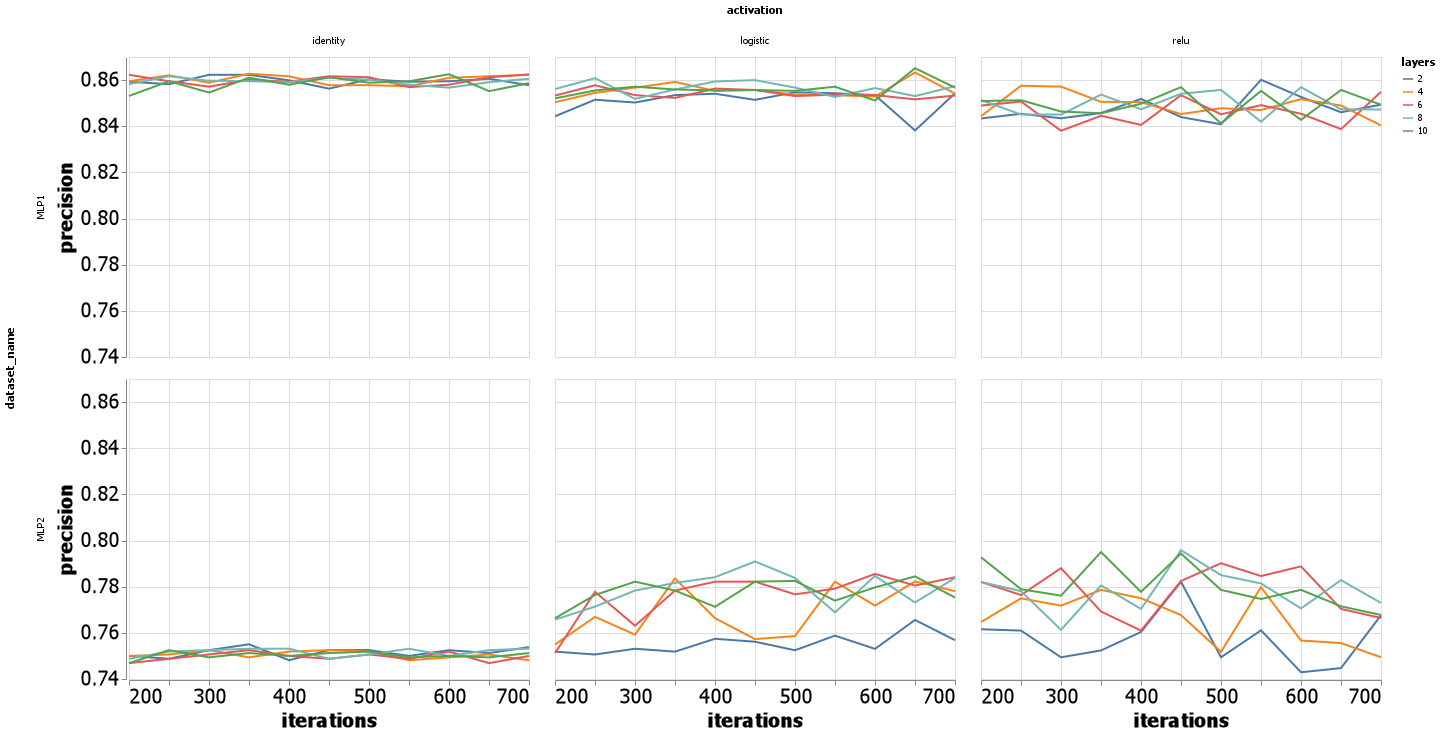
K-Nearest Neighbor (KNN)

Following the same pattern, we need to establish our base case. The hyperparameters I chose to examine are n\_neighbors, algorithm, and metric. The default for algorithm and metric are auto and Minkowski respectively. Even though I did not iterate over the Minkowski metric specifically, the Minkowski equation takes in a parameter p where when p=1 it is a Manhattan distance and when p=2 it is a Euclidean distance. Due to this fact, our base parameters will be auto and both Euclidean and Manhattan. Like decision tree, for the first and second datasets the lines will be blue and orange for Euclidean and black red and black for Manhattan respectively.

 Before we go in, I want to show a behavior that I found interesting. To the right is a chart ranging from 1 neighbor to 250 neighbors. As you can see, around 40 neighbors the graph begins to plateau. This was specifically for the ball\_tree metric, but all of the other metrics performed in this same manner. This is interesting because it tells us that anywhere past 30-40 neighbors and the information gained will only result in roughly a .01-.02 change. Even though this is specific to our model, it is interesting to know that there comes a point where no matter how many neighbors you want your model to look at, the model will perform the same. It isn’t work the extra computation time to search over the extra 220 neighboring points.

I know I said we would be examining the precision metric, but below this is a graph that plots the precision metric against our other KNN models. I know it is extremely fishy, but I’ve checked my code many times over and I could find any reason this would be happening due to an error on my side. This leads me to believe the data was well clustered, which I would try to show but we cannot enter the 23rd dimension. It makes sense why this would occur, the ball\_tree and kd\_tree algorithms are very similar in the sense that they both work like little decision trees. The ball\_tree algorithm creates hyperspheres around certain clusters of points and tries to make those hyperspheres as pure as possible, then from within each sphere more spheres are formed while trying to maximize the distance between each sphere (hence the name “ball”). The kd\_tree splits the data on the mean of each feature for each class, thus creating something that might look like a bunch of boxes in a multi-dimensional space. The brute algorithm goes through the data and attempts to make predictions only by guessing and checking. It is a little rudimentary, but that is why they call that parameter “brute”. If the data was well partitioned and separated, it makes sense that these 3 algorithms would come to the same conclusion. If anything, this would tell you the model you have created has examined all possible outcomes and is tuned very well.

Artificial Neural Networks (ANN)

Below in the chart you will see how the variance of each of the lines increase with different activation functions. This was something that was seen across both of the datasets, starting with the identity function the variance is very small bouncing between .85 and .87. Then as we go through the logistic and relu functions we see the variance bounce between .75 and .8. In all reality the difference we see here isn’t big, but it is the spread of variance that makes it interesting. The identity function is a linear activation function, thus making it proportional to the input received and making the predicted outputs more normalized. The logistic function shows us that as the value of x increases it slowly approaches 1 and as x decreases it slowly reaches .5. The relu function is a value of 0 if the input is less than 0 thus essentially canceling out that value. So, it completely disregards all other inputs that are less than 0, thus would create more variation in the outputs if a portion of the inputs are less than 0. This wouldn’t be a problem though if all your inputs are greater than 0. With these 3 activation functions identified, we can begin to decern why the graph is behaving the way it is. When we append multiple layers to the identity activation function, any combination of linear equations is going to be linear, thus not allowing any of the nodes to behave differently and creating a constant small variance. As for the logistic and relu functions, they allow each layer to take on a nonlinearity behavior. This allows for some layers and nodes to go higher much quicker and some to stay stagnant. Once the output is reached for these two activation functions, the numbers would be across board as to what the prediction would be thus creating a wider range across the precision score metric.

Chart, line chart

Description automatically generated Since this is an iteration model, time is also something that should be examined. Below is the same chart from up above just with time as the y-axis. Because the identity function is linear, we don’t see any significant fluctuations in the training time across any number of layers. As for logistic and relu, it is not surprising to see their times fluctuate. Due to the logistic function working with exponentials, it is not surprising to see it grow quickly as the computation time needed for each node goes up at an ever-increasing rate. As for relu, we see it have a slight increase as the number of iterations increase, but not nearly as fast as the logistic function.

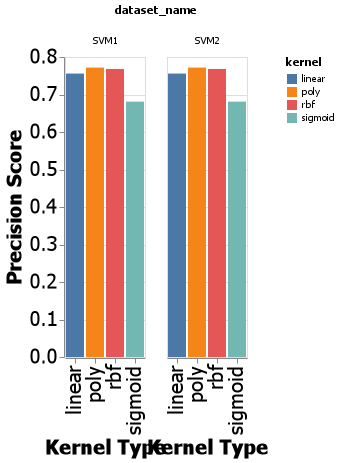
Boosting

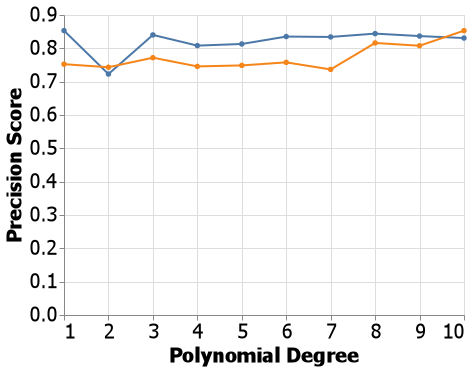
For this algorithm I chose to examine the loss function, learning rates and the number of estimators within the boosting model. At first, when I started this analysis I wanted to see what would happen when I do a wide range of estimators and I produced the chart below. It was interesting to find that even though they had different precision scores, there came a point in the number of estimators where the information gained was insignificant. So I went back and re-ran the algorithm to show more information from 0 to 50 as this seems to be where the charts leveled *Chart, scatter chart

Description automatically generated*off. I also decided to decrease my max learning rate as a learning rate of 3 was leaping too far and looks like it got stuck in some local minima where the precision was very bad.

While exploring the results from my new tuning, I found that boosting does well very quickly. Only with a few estimators was I able to achieve a high precision score and stay there across any number of estimators and a learning rate of 1 or less. Once I reached a learning rate of 2 or 1.5 it was descending the gradient too quickly and would miss what most likely was the global minima.

Support Vector Machines (SVM)

 For the support vector machine, I chose to look at the kernel and degree parameters. Once I saw how the KNN model performed and the results were algorithm agnostic, I wondered how the SVM would do under different kernels. Even though they are all extremely similar, the sigmoid function stood out. Where the sigmoid function is essentially a two layer perceptron activation function, it creates values that are 0s or 1s which can give it the potential to over or underfit the data because it looses some of the complexity behind the data.

 Once I saw that the polynomial kernel was best suited for this type of problem, I chose to go back and look at what would happened when I select only the polynomial kernel but change its degree. With the blue line being the first dataset and the orange being the second, we can learn some information about the dataset by just looking at this graph. In the high dimensional space that the model for the first dataset lives in, it would be save to assume the data can be separated by only using a straight line. Notice how when the polynomial degree is “2”, our precision score drops significantly. When the highest degree is 2, it is impossible to make the line bend back down in the same relative direction before it started to bend back up again. Once we get into the 3rd order and higher it is possible to make the line quickly turn up and down to fit the data. We also see a huge jump in precision after the 8th order degree is reached for the second dataset. This tells us there is some complex relationship between many of these variables in which the data needs to be able to bend and curve at varying intervals and this can only be achieved when we work with a kernel of “poly and a degree of “8” or more with 10 producing some of the best results we have seen for the second dataset.

**Part 2: Model Comparison (Same Data)**

Dataset 1: Subscriptions

Attributions

Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011

Gladwell, Malcolm. Outliers: The Story of Success. New York: Little, Brown and Co., 2008