Supervised Learning: 5 Algorithms

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

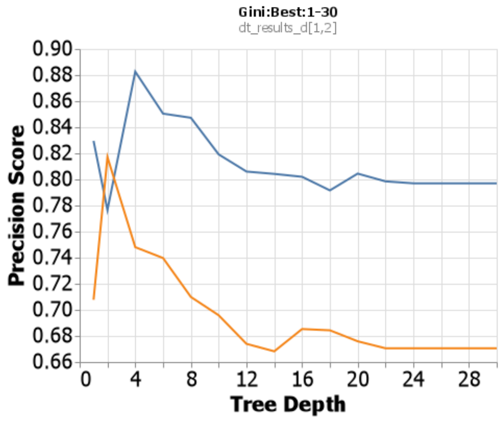
There are patterns and behaviors or "rules" to supervised learning algorithms. These must be understood before we can learn to "break them like an artist". As we go through this paper, we will be exploring each of these models across 2 different datasets. These two datasets come from Kaggle and are called 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.

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. This analysis was done in the python language and utilized the Sci-kit Learn (Sklearn), Altair, and Pandas packages and their dependencies found within.

Part 1: Model Behavior

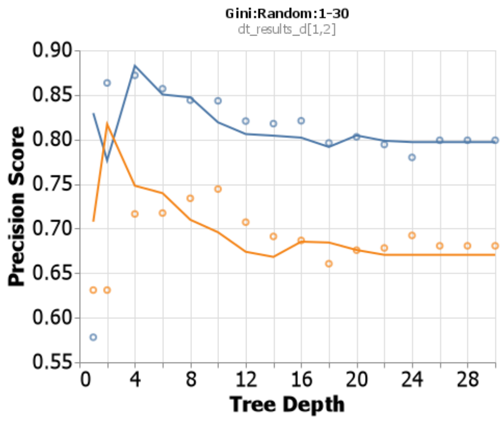
Decision Tree

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 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

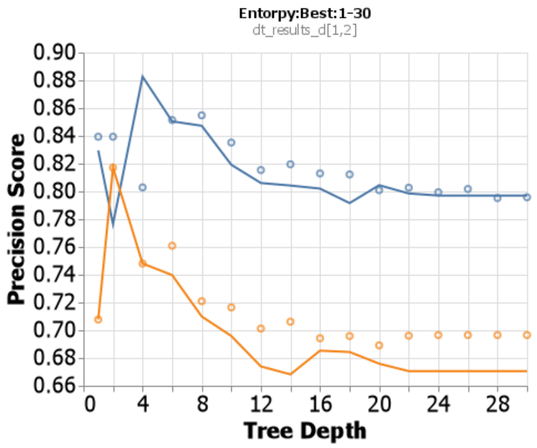


combination to be our base case with the blue line representing the first dataset and the orange line representing the second dataset and a line to denote the base case (see right).

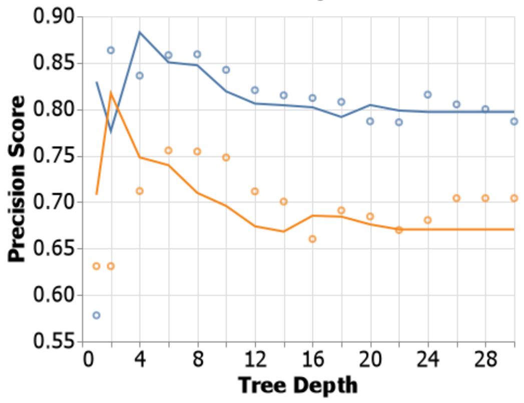
Now that we are familiar with our base case, we can put in another test case to compare, Gini:Random. When we look at depths 16 and on we see a little difference in the precision score, but no significant difference. When we look at 1-14 though, this is where the graph gets interesting. Here we changed 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 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 $\sum_{i=1}^{n} p_{ilog_2 p_i}$ and the equation for gini is $\sum_{i=1}^{n} p_i^2$. Due to the log in the entropy equation it does a good job at 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 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

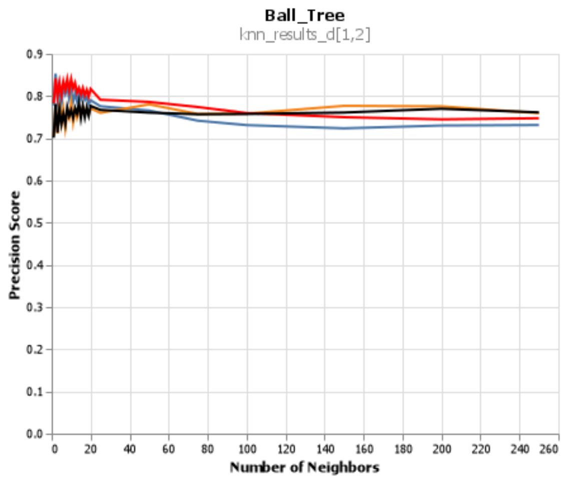


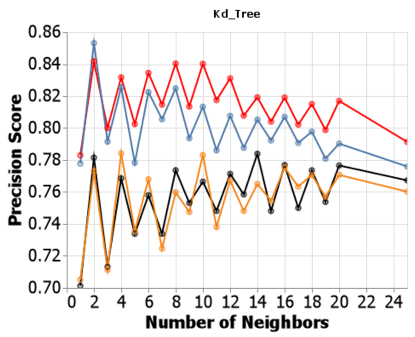
inconsistent types of splits. This adds on to the idea that entropy is trying to generalize, but it is being fed bad splits by the random splitter.

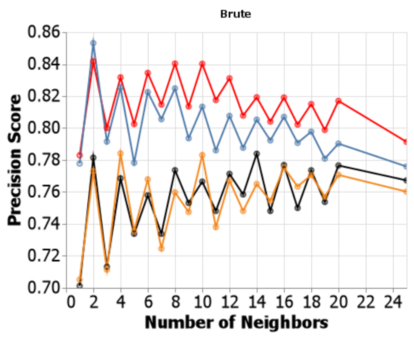
K-Nearest Neighbor (KNN)

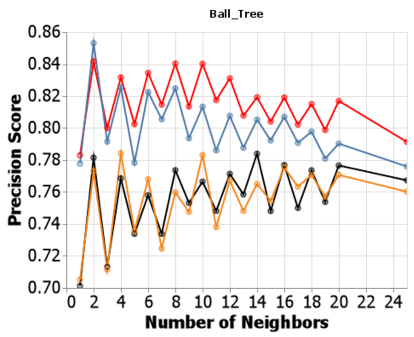
Following the same pattern, we need to establish our base case. The hyperparameters I chose to examine are n_neighbors, algorithm, and metric. The default for algorithm and metric are auto and Minkowski respectively. Even though I did not iterate over the Minkowski metric specifically, the Minkowski equation takes in a parameter p where when p=1 it is a Manhattan distance and when p=2 it is a Euclidean distance. Also knowing the default for p is 2, our base parameters will be auto and Euclidean. Like Decision Tree, for the first and second datasets the lines will be blue\orange for Euclidean and black\red 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 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 worth the extra computation time to search over the extra 220 neighboring points.





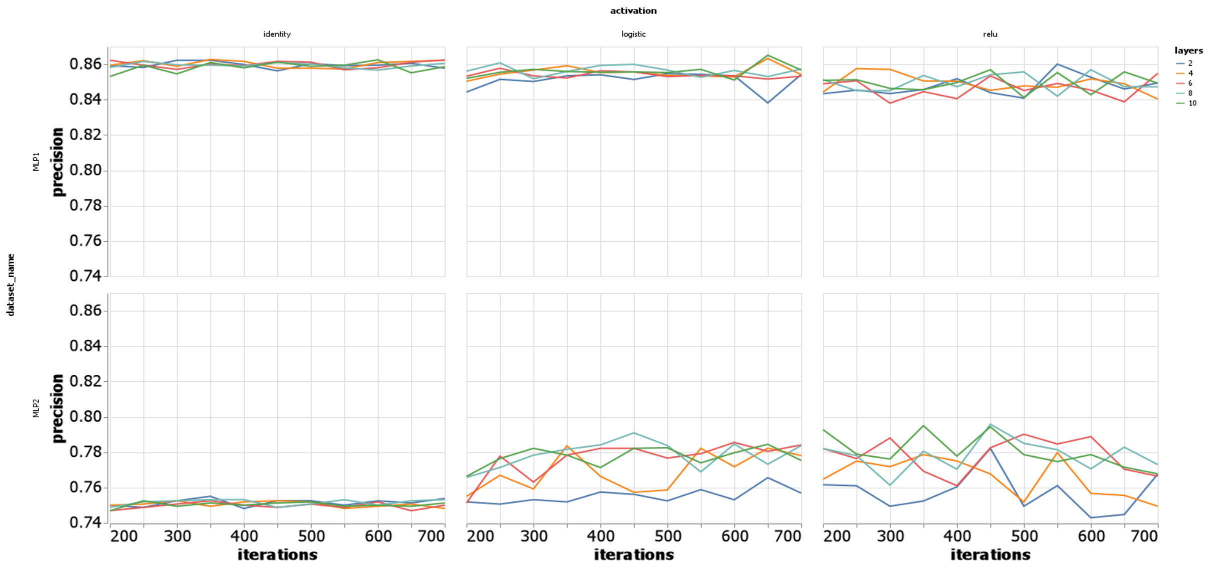




Above is a graph that plots the precision metric against our other KNN models. I know it is extremely fishy, but I've checked my code many times over and I couldn't find any reason this would be happening due to error in the code. This leads me to believe the data was well clustered, which I would try 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, 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.

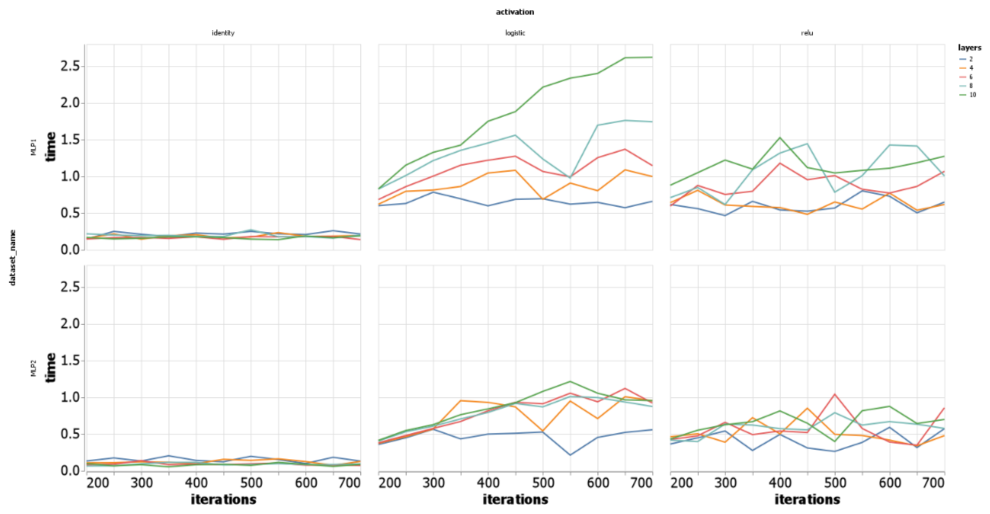
Artificial Neural Networkworks (ANN)

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



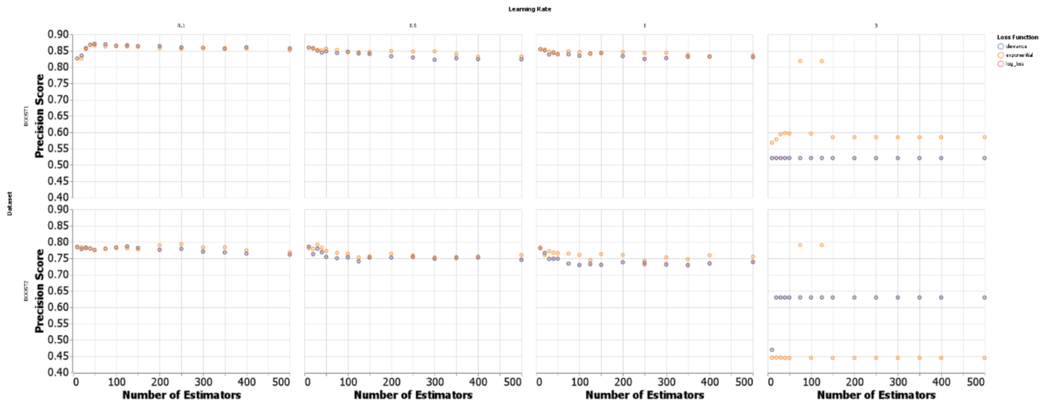
interesting. The identity function $f(x) = x$ is a linear activation function, thus making it proportional to the input received and making the predicted outputs more normalized. The logistic function $f(x) = \frac{1}{1+e^{-x}}$ shows us that as the value of x increases it slowly approaches 1 and as 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. 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.

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, 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

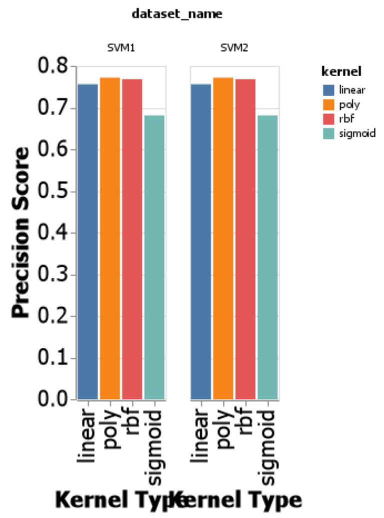
For this algorithm I chose to examine the loss function, learning rates and the number of estimators within the Boosting model. At first, when I started this analysis, I wanted to see what would happen when I do a wide range of estimators and I produced the chart below. It was interesting to find that 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 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 a local minima.



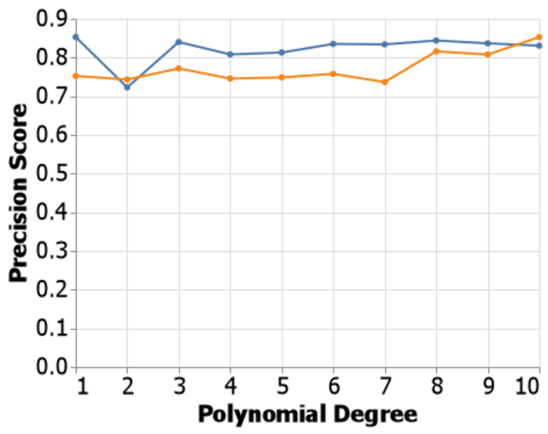
While exploring the results from my new tuning, only with a few estimators was I able to achieve a high precision score and stay there across any number of estimators and a learning rate of 1 or less. Once I reached a learning rate of 2 or 1.5 it was descending the gradient too quickly and would miss 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.

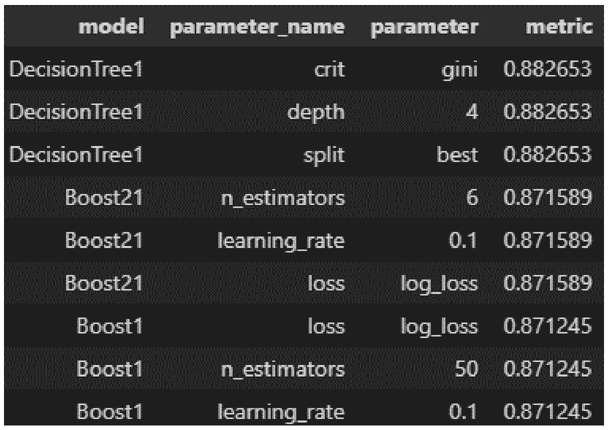


Once I saw that the polynomial kernel was best suited for this type of problem, I chose to go back and look at what would happen when I select only the polynomial kernel but change its degree. With the blue line being the first dataset and the orange being the second, we can learn some information about the dataset by 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 separator 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.

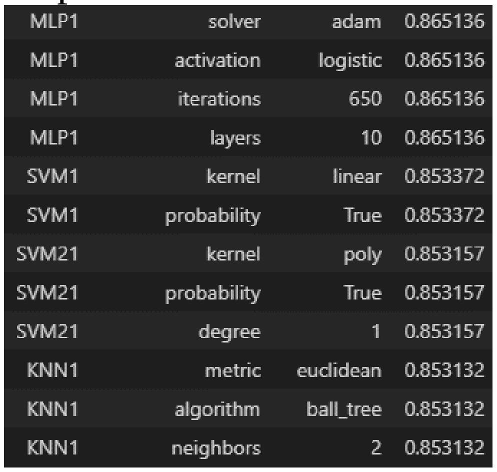


Part 2: Model Cross Comparison

Dataset 1: Subscriptions

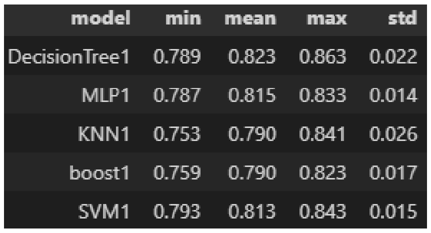


model	parameter_name	parameter	metric
DecisionTree1	crit	gini	0.882653
DecisionTree1	depth	4	0.882653
DecisionTree1	split	best	0.882653
Boost21	n_estimators	6	0.871589
Boost21	learning_rate	0.1	0.871589
Boost21	loss	log_loss	0.871589
Boost1	loss	log_loss	0.871245
Boost1	n_estimators	50	0.871245
Boost1	learning_rate	0.1	0.871245



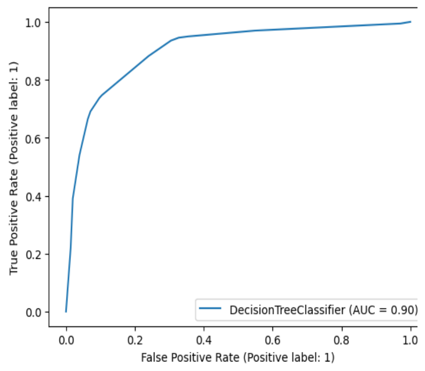
MLP1	solver	adam	0.865136
MLP1	activation	logistic	0.865136
MLP1	iterations	650	0.865136
MLP1	layers	10	0.865136
SVM1	kernel	linear	0.853372
SVM1	probability	True	0.853372
SVM21	kernel	poly	0.853157
SVM21	probability	True	0.853157
SVM21	degree	1	0.853157
KNN1	metric	euclidean	0.853132
KNN1	algorithm	ball_tree	0.853132
KNN1	neighbors	2	0.853132

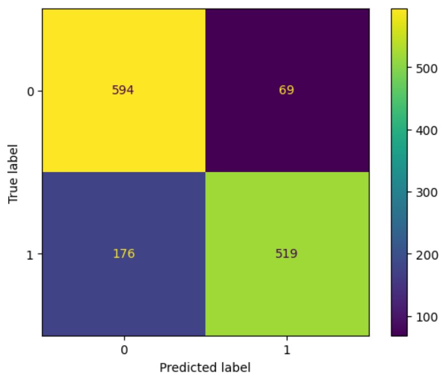
As mentioned earlier, the first dataset is based on the type of people who subscribe to a certain product. At this point, we have a long list of CSVs, the models and their hyperparameters with how they performed using various metrics to identify that performance. above is a table with all the best parameters for the precision metric. The Boost21 and SVM21 are the second versions of the Boost and SVM model I went back and ran more tests over because I realized the best parameters had more room for even finer tuning. The chart is ordered by greatest to least based on the metric column. From this we can see the Decision Tree was the best model for this dataset with a precision score of .86, this also shows us some of my models got lucky on the splitting and decisions with a normal train_test_split. To double check this I picked all of the best hyper parameters and ran them through a cross validation each with 10 folds focusing on general accuracy to get a broader view of how our model is behaving. Above are the results. Seeing as how the standard deviation is only .01 or .02 I feel very confident that the values we are seeing are accurate and reliable enough to make these parameters our main. As a special note, since the second versions of both the SVM and Boost models did better than the first version we will use those parameters in our final model.



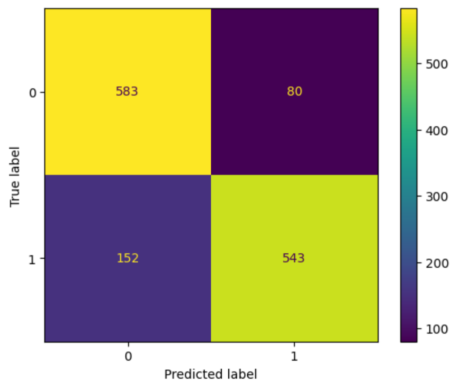
model	min	mean	max	std
DecisionTree1	0.789	0.823	0.863	0.022
MLP1	0.787	0.815	0.833	0.014
KNN1	0.753	0.790	0.841	0.026
boost1	0.759	0.790	0.823	0.017
SVM1	0.793	0.813	0.843	0.015

As we begin to cross compare the models and how well they performed with their best hyperparameters, we will be comparing them against the Decision Tree model as this model is the one that performed the best. But, before we can compare against the Decision Tree model, we need to examine it by itself. To the right is an ROC curve for the Decision Tree model just to establish a base line. Even though this isn't the best ROC curve, it is the best on our dataset. Note, all but one of the models had nearly identical curve as this one so I won't be putting it in the report except for the one that looked very different. Even though we chose the precision metric for our analysis, we see that our Decision Tree model wasn't well optimized for that metric, and it performed better with the recall metric.



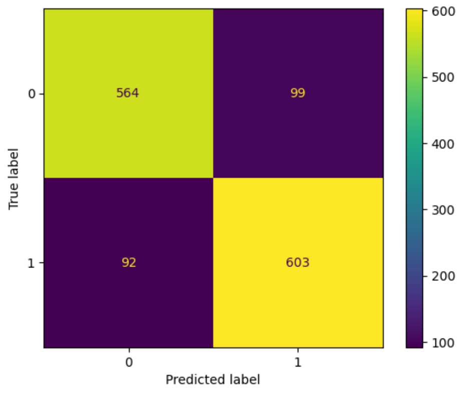


The first model we will explore will be the next runner up, our second revision of the Boosting model. Its parameters were n_estimators = 6, learning_rate = .1 and loss = log_loss, this gave us a precision score of .87 and a training time of .0069. Below you will see the

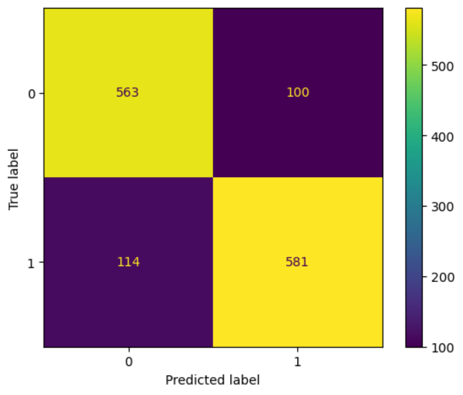


confusion matrix, roc and precision recall curves. As to why this model did just a little bit worse, I think it can be found in the ensemble portion. Because we had so many models going into the Boosting model, they can pick up on smaller trends but that may not always be the best idea. Sometimes an ensemble model can pick up on a rule that may be specific to only a few, but then it is applied to the whole. Not to mention the Boost model can create a tree that isn't pruned by what the user knows to be the best limit. Interestingly, it did better for reducing the number of type 2 errors than the Decision Tree. Goes to show that more heads aren't always better than one.

The next runner up is our Neural Network. With parameters at layers = 10, iterations = 650, and activation = logistic this model comes in with a precision score of .86 and a training time of .14. Something I find interesting for this model comparison is in the confusion matrix. When we compare them the Neural Network was almost equally weighted in the precision recall balance. This makes sense because the Neural Networkwork is designed to balance all outputs and make sure things are weighted properly to give us a very balanced model. This is good to know because if you are wanting a model that generally does well and has a high accuracy score, a Neural Network would be a good place to start.

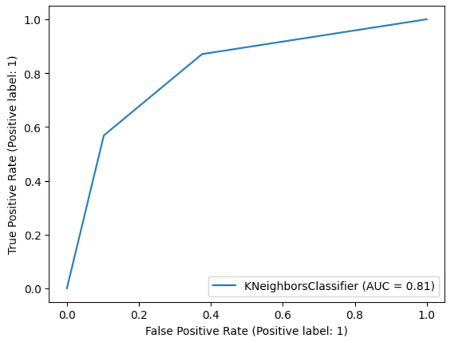


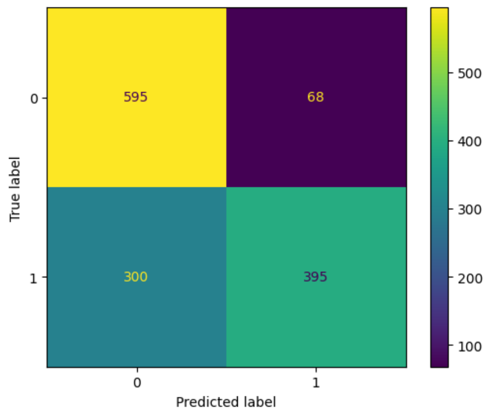
Our SVM model had parameters of kernel = poly, probability = True and degree = 1, the two metrics of time and precision were 1.84 and .85 respectively. Both the first version and the second version are essentially the same. When we have a degree of 1 in a polynomial kernel it is fundamentally the same as a linear kernel. We see only a small difference in the two, less than .001. Similarly, to the Neural Network, our confusion matrix for the SVM seems to be very balanced as well. With a degree of 1, this helps me to further believe that this data is



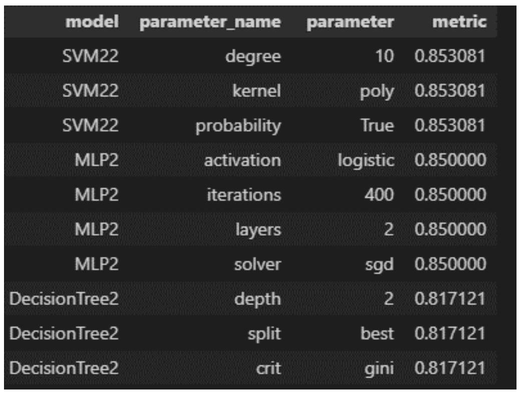
very well clustered and is best separated by linear means where the clusters have low contamination.

For our last model on the first dataset we come to KNN with parameters metric = Euclidean, algorithm = ball_tree, and neighbors = 2 with a precision score of .85 and a training time of basically 0. This isn't surprising though because we know KNNs to be lazy learners. Both the confusion matrix and ROC curve are interesting compared to the Decision Tree. Below you will see how well the model did in predicting type 1 errors. Out of all the models none of them did this poorly. The ROC curve as well is most bizarre, all the other curves smoothed out where this curve has a sharp turn. This tells me there was probably a cluster of points where most of them were 0, but there was enough 1s in there to bring down the score significantly. This is one of the drawbacks of a KNN compared to a Decision Tree.

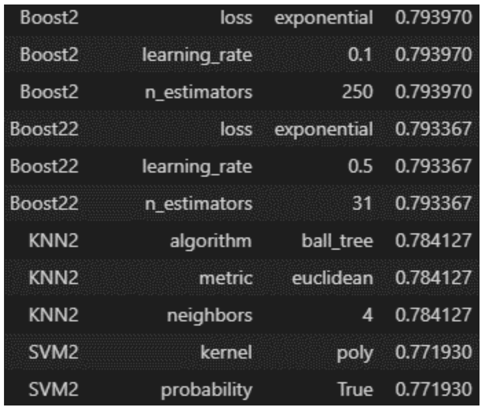




Dataset 2: Customer Churn

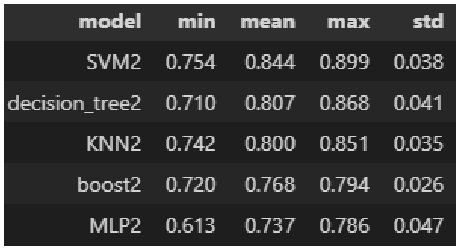


model	parameter_name	parameter	metric
SVM22	degree	10	0.853081
SVM22	kernel	poly	0.853081
SVM22	probability	True	0.853081
MLP2	activation	logistic	0.850000
MLP2	iterations	400	0.850000
MLP2	layers	2	0.850000
MLP2	solver	sgd	0.850000
DecisionTree2	depth	2	0.817121
DecisionTree2	split	best	0.817121
DecisionTree2	crit	gini	0.817121



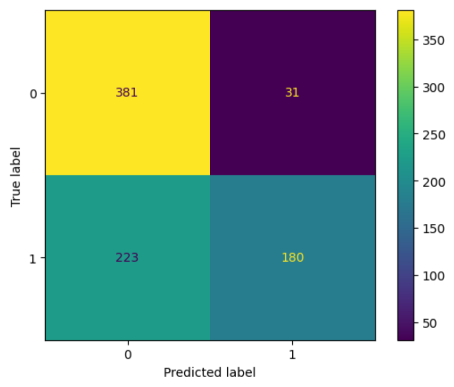
Boost2	loss	exponential	0.793970
Boost2	learning_rate	0.1	0.793970
Boost2	n_estimators	250	0.793970
Boost22	loss	exponential	0.793367
Boost22	learning_rate	0.5	0.793367
Boost22	n_estimators	31	0.793367
KNN2	algorithm	ball_tree	0.784127
KNN2	metric	euclidean	0.784127
KNN2	neighbors	4	0.784127
SVM2	kernel	poly	0.771930
SVM2	probability	True	0.771930

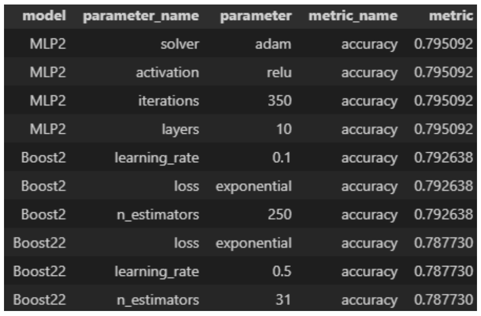
This dataset is all about customer churn. To the above is the list of each model, including the second versions of my Boosting and SVM models. Like before, this table is ordered by how well each model performed. Here we see the second version of the SVM model out performed every model, and even the first draft of our SVM model by roughly .08! To confirm



model	min	mean	max	std
SVM2	0.754	0.844	0.899	0.038
decision_tree2	0.710	0.807	0.868	0.041
KNN2	0.742	0.800	0.851	0.035
boost2	0.720	0.768	0.794	0.026
MLP2	0.613	0.737	0.786	0.047

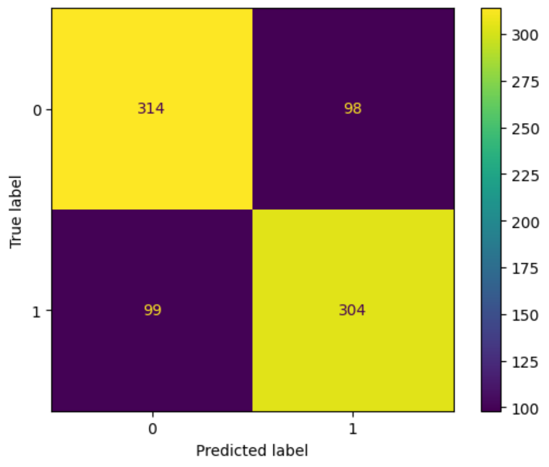
these numbers I ran a cross validation with 10 folds and found these metrics in the chart below. Our SVM model still comes out on top, but after that things tend to shift around. Again, this speaks to the importance of running a cross validation search instead of taking the word of one normal train_test_split.. We don’t have enough room to cover all models so we will only examine the top 3. Something I found interesting is how deceptive looking at one metric can be. If we look at the confusion matrix below, we will see in terms of a precision score, the SVM did really well, but in overall accuracy it did poorly. Infact, if we were to change the initial graph above to focus on accuracy instead of precision, this is what the top 3 would look like. It is very important when you pick a metric you want to optimize for, that you are willing to sacrifice performance in the other areas. It isn’t a guarentee you will loose performance, but each model is highly suceptible towards this and can be negatively effected as such. Now, onto our first model.



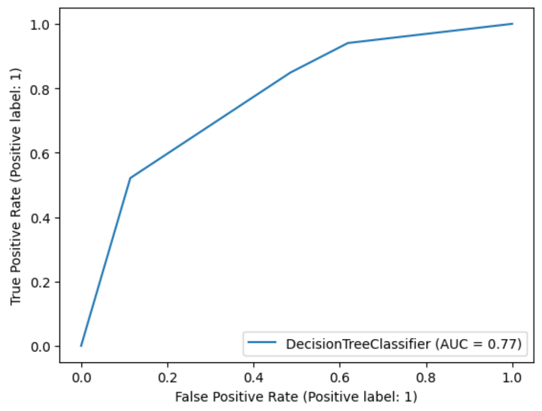


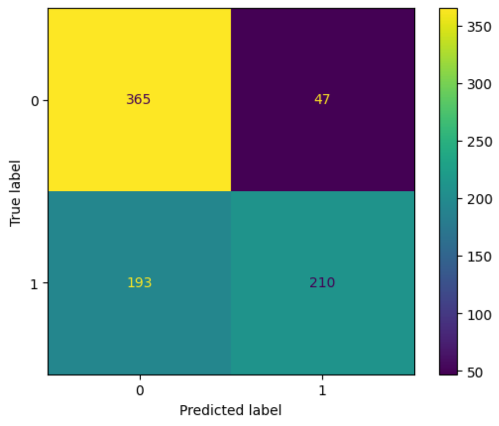
model	parameter_name	parameter	metric_name	metric
MLP2	solver	adam	accuracy	0.795092
MLP2	activation	relu	accuracy	0.795092
MLP2	iterations	350	accuracy	0.795092
MLP2	layers	10	accuracy	0.795092
Boost2	learning_rate	0.1	accuracy	0.792638
Boost2	loss	exponential	accuracy	0.792638
Boost2	n_estimators	250	accuracy	0.792638
Boost22	loss	exponential	accuracy	0.787730
Boost22	learning_rate	0.5	accuracy	0.787730
Boost22	n_estimators	31	accuracy	0.787730

The MLP model performed best overall as seen in the confusin matrix. Again, this is most likely due to the fact that a Neural Networkwork is aimed towards creating the best balanced model by adjusting the weights and layers within the model. As you tune your MLP model, you can be assured the resulting model will be a model best suited for any of the 3 metrics, but you do need to be aware of time when working with an MLP model as they can be one of the most labor intesive models out there.



Next up is the Decision Tree. Again, where a Decision Tree isn't biased or more prone to lean towards one score or another, it isn't hard to see why on paper this model seems so good. We sacrifice predicting they would churn when in reality they didn't so we could increase the number of people we target for an ad campaign, for example. Even though the tree is pretty shallow with a depth of 2, we know that this is the best combination for our learning curve, roc curve, and focusing on a precision score. Just by looking at the ROC curve, we know our model suffers from this type of optimization. A model that is well balanced with have a more gradual curve instead of the sharp points you see on the graph.





Coming to learn the rules of machine learning, especially in supervised learning can be extremely challenging. It involves lots of study, experiments, and practice. Knowing how Decision Trees split their nodes, or why KNN clusters certain points together, or why a Boosting model doesn't tend to over fit, or why a Neural Network will often come out balanced and have a better accuracy score, and why SVMs even need a polynomial kernel is what makes someone a pro. This is not something we come to overnight. The 10,000-hour rule may be understating how long it will take for someone to really reach this stature. Why then is it important to learn them? So we may be the change and use them in ways others have not even dreamed of, this is the art of machine learning.

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

Sievert, S. (2018). Altair: Interactive statistical visualizations for python. Journal of Open Source Software, 3(32), 1057