Machine Learning for Financial Market Prediction: Classification with Keras and sklearn

By Druce Vertes | September 4th, 2018 | Machine Learning, Research Insights

In our last post, we applied a machine learning approach to regression, a familiar algorithm to most finance practitioners. We showed how machine learning can improve forecasting versus linear regression, when underlying reality is nonlinear with complex interactions. (And we have adequate data!)

In this post, we'll review machine learning for classification, which is less familiar, but can have useful investing applications.

Regression predicts a numerical value: for example, the return on an asset.

Classification predicts a discrete value: for example, will a stock outperform next period? This is a *binary classification* problem, predicting a yes/no response.

Or, which quartile will a stock's performance fall into? This is *multinomial classification*, predicting which of 4 possible outcomes will be the correct one.

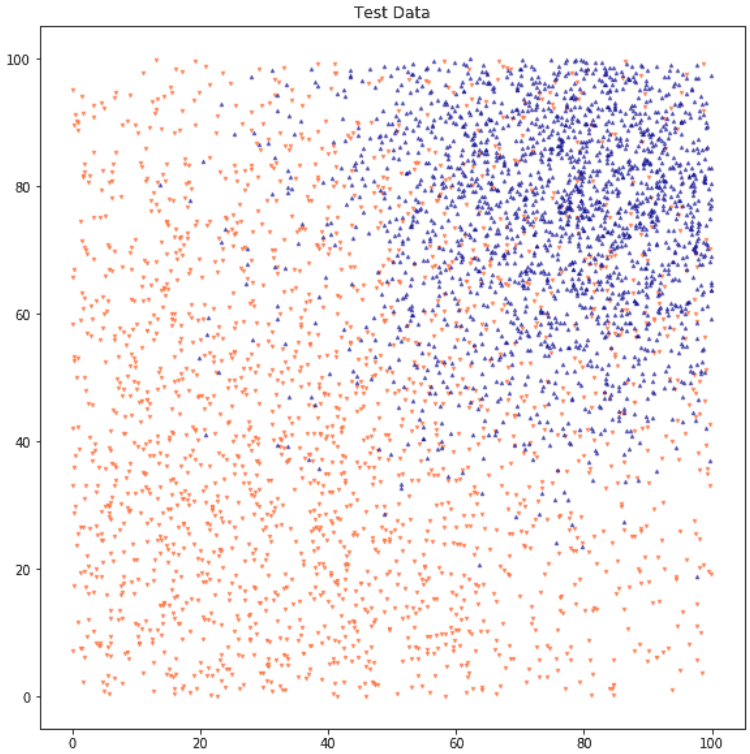
In this post:

1. We'll break down a classification example 'Barney-style' with Python code.
2. We'll do a high-level overview of classification algorithms.
3. Finally, we'll apply classification to the same problem we looked at last time to generate a portfolio strategy by classifying industries into long/short/flat.

From classification, it's a short step to deep learning problems like handwriting recognition, or image labeling. Classification is a fundamental building block that enables machine learning to perform what looks like magic.

Part 1. Classification essentials.

We start (code is here) by generating random data with two predictors (the x and y axes) and a variable with two labels:



2000 red dots are centered around 25, 25 with a standard deviation of 50, and assigned the label 0.

2000 blue dots are centered around 80, 80 with a standard deviation of 20, and assigned the label 1.

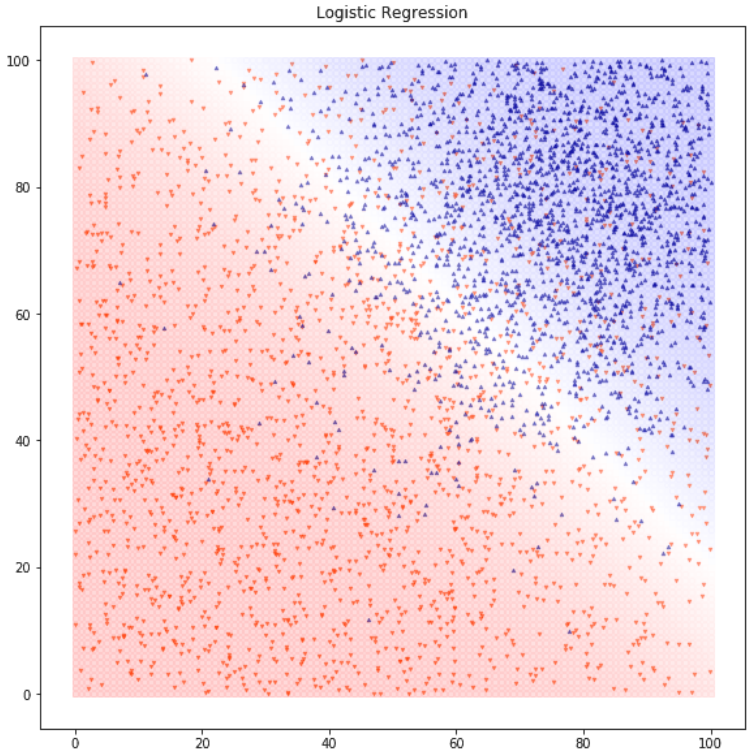
We want to create a function that takes the coordinates x and y, and returns a prediction of 0 (blue) or 1 (red).

Logistic regression is the simplest, most popular classification algorithm. How it works:

1. Apply a linear *decision function* of x and y that outputs a numeric variable: $Z(x, y) = ax + by + c$
2. Apply a *sigmoid* 'squashing' function to Z that maps large positive numbers to a probability estimate close to 1, and large negative numbers to a probability close to 0.
3. Apply a *loss function* that returns a value close to 0 when the predicted probability estimate matches the observed value. The loss should be close to 0 when the predicted probability is close to 1 for the blue observations (label=1), and also close to 0 if the predicted probability is close to 0 for the red observations (label=0).
4. Finally, find the parameters a, b, c of our linear function that minimize the loss function.

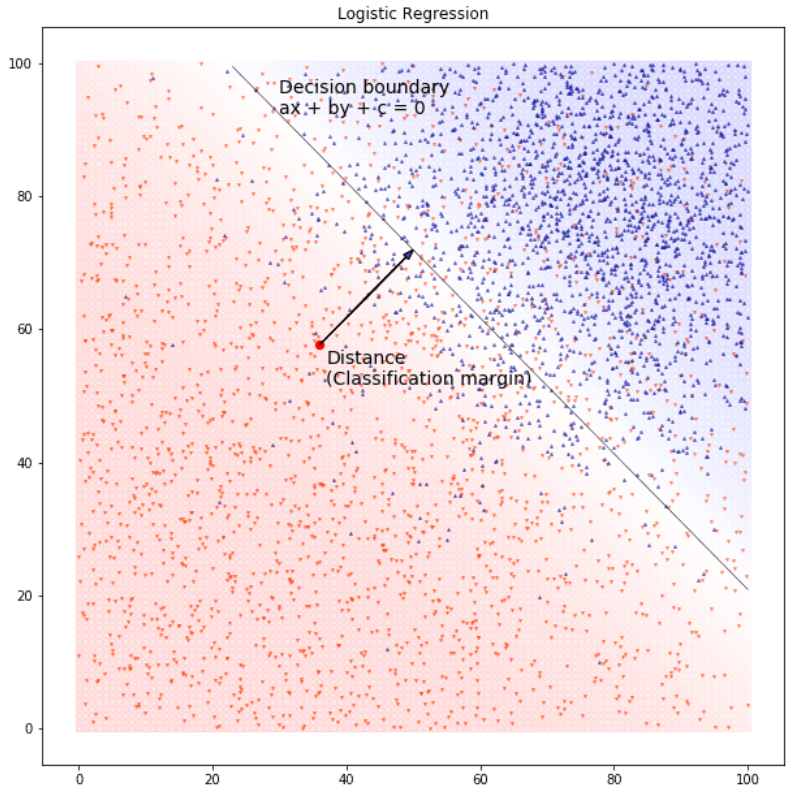
If we are able train our classifier so its average loss is close to zero, we obtain good matches between our predictions and observed values. We minimize the loss via *gradient descent*: choose random starting values for a, b, c, determine in which direction they need to be updated to reduce the loss, and iteratively update them until we find values that achieve a minimum loss.

When we do that, we obtain a classification that looks like this:



Now, let's look more closely at the math behind these four steps. (If you're not into the math, you can skip down, but it's not too difficult, and after all Alpha Architect is a quant finance blog!)

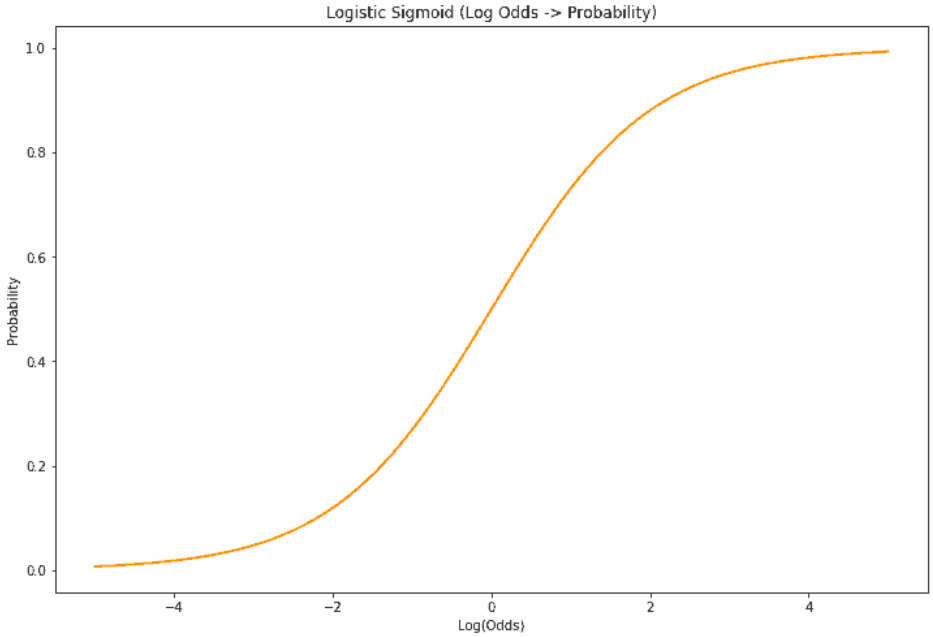
Decision function: Z(x,y) = *a*x + b*y* + c is proportional to the Euclidian distance from (x, y) to the line defined by *a*x + b*y* + c = 0. It represents how far a point is from the decision boundary, the *classification margin*.[1]



'Squashing' function: For logistic regression, the squashing function is the *logistic function*: $\frac{1}{1+e^{-Z(x,y)}}$

This type of function, which maps [-∞, +∞] to [0, 1], is known as a *sigmoid* function.

An intuitive explanation of the logistic function is that it converts *log odds* to probability.



Suppose you take Vegas-style odds like 3:1, which map to probability 0.25. Then:[2]

- $LogisticFunction(log(3)) = 0.75 \approx LogisticFunction(1.1)$
- $LogisticFunction(log(1)) = 0.5 = LogisticFunction(0)$
- $LogisticFunction(log(1/3)) = 0.25 \approx LogisticFunction(-1.1)$

In other words, logistic regression *models log odds as a linear function of the distance from the decision boundary*.[3]

Loss function: Suppose the predicted probability of blue or label 1 is 0.8. What is our measure of how bad a prediction this is, that we would like to minimize? For binary classification, we minimize *log loss* (also known as *binary cross-entropy*).

$$LogLoss = -log(p)y - log(1-p)(1-y)$$

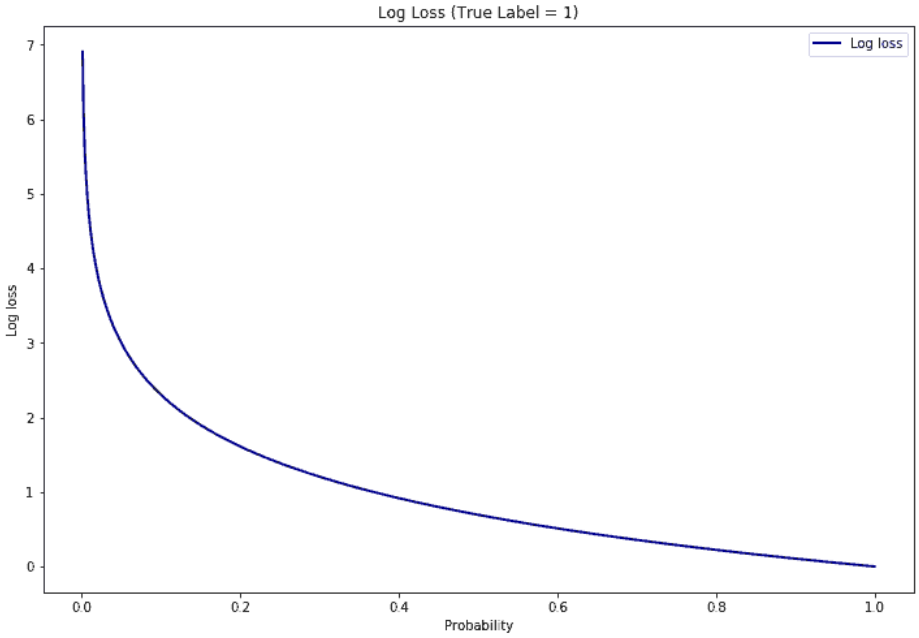
where *y* is our label (0 for red, 1 for blue) and *p* is our predicted probability of label = 1 (blue).

A good way to visualize log loss is as *-log(correctness).* If our prediction is 100% correct, log loss is 0. If our prediction is 0% correct, log loss is +∞.

To break it down further: Our observed label *y* is either 0 or 1. Our probability prediction *p* is between 0 and 1 exclusive.[4]

If the observed label *y* is 1, and our predicted probability is *p* then $LogLoss = -log(p)$. The second term goes away since 1-*y*=0. For a prediction close to 1, the log loss is close to 0. For a prediction close to 0, the log loss is very large.

Conversely: If the observed label *y* is 0, then $LogLoss = -log(1-p)$. The first term goes away since *y*=0. For a prediction close to 0, the log loss is 0. For a prediction close to 1, the log loss is very large.

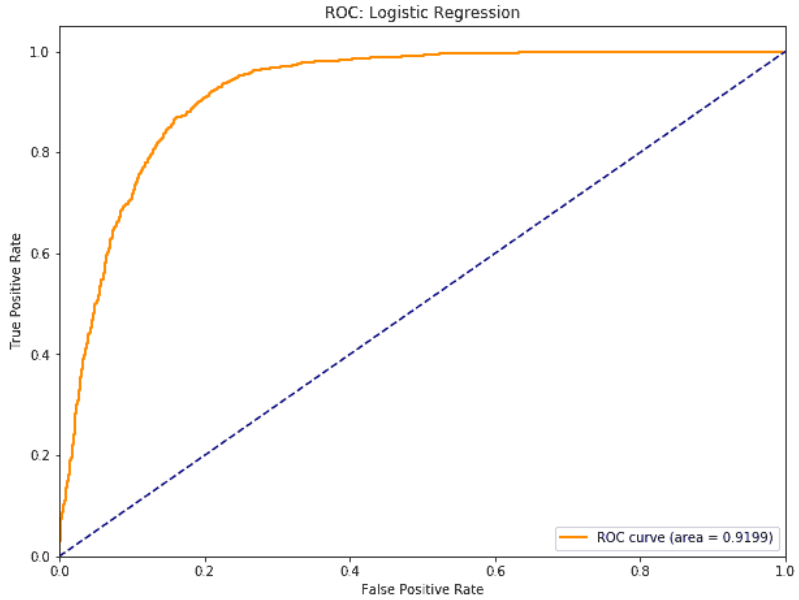


Using these four steps, we reduced our binary classification problem to, find *a*, *b*, *c* that minimize the average of $LogLoss(LogisticFunction(ax + by + c))$ over all our training observations.

That's it! The essence of binary classification: find a function of a vector of predictors that outputs a probability of the class label being 1, with minimal cross-entropy.

A final important point:

We predict the blue class when the predicted probability exceeds a probability threshold. In general, it should not be assumed that our decision boundary must be at the 50% probability line. In real-world problems it's important to consider the cost of false positives vs. false negatives. As we raise the probability threshold, we reduce the number of false positives and increase the number of false negatives. The ROC curve can help us visualize that. We want to choose a decision boundary threshold that maximizes real-world performance, where if we increase the threshold for a positive prediction, the marginal gain of fewer false positives equals the cost of additional false negatives.



This is a toy problem with no real-world costs of false positives and negatives, so let's just maximize accuracy. After selecting a threshold to maximize accuracy, we obtain out-of-sample test accuracy of 85.3%, and an area under the ROC curve of 92.0%.

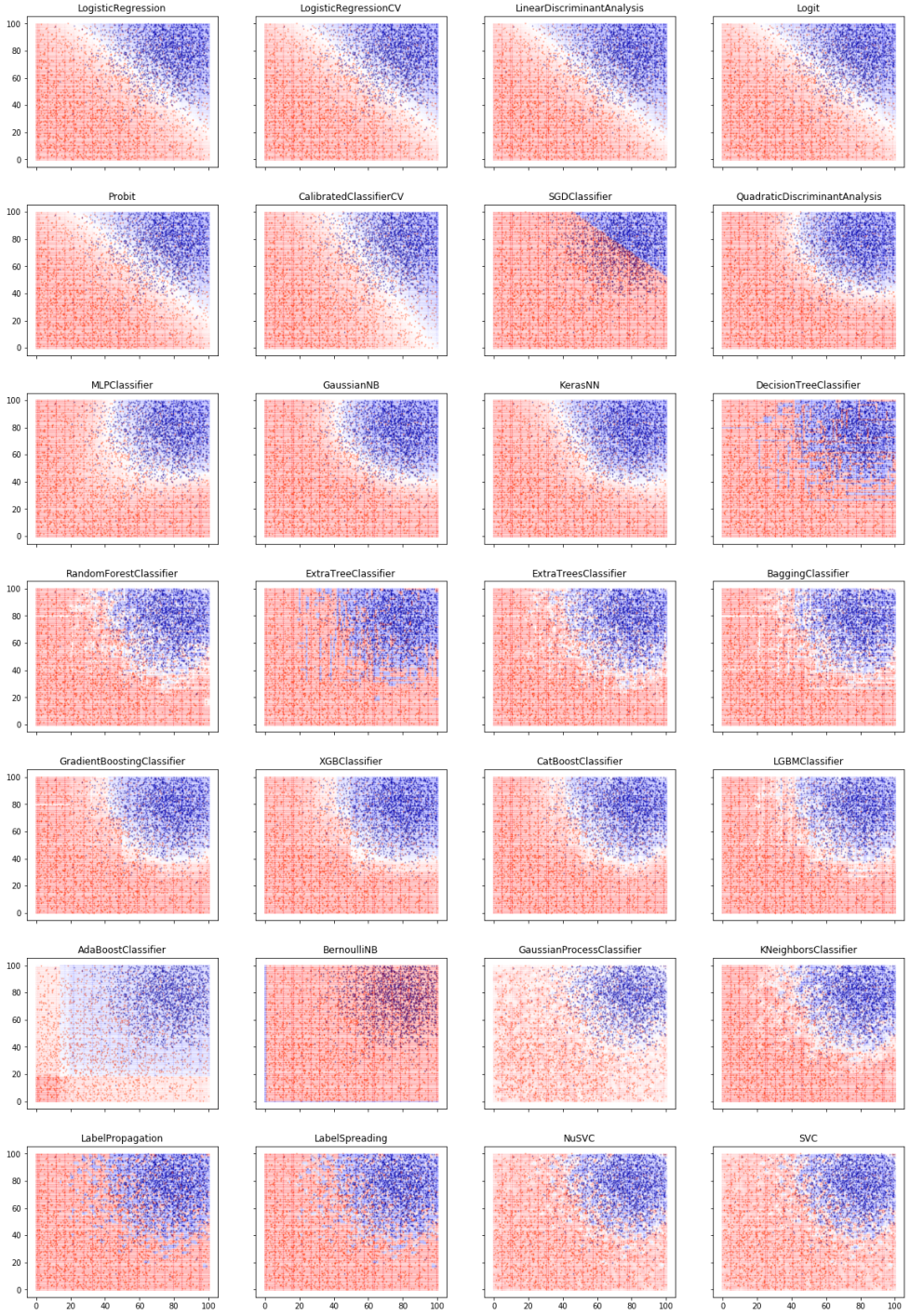
```
In-Sample Accuracy (50% threshold): 0.8478
CV Threshold 0.4390 Accuracy 0.8608 F1 0.8679
Out-Of-Sample Accuracy: 0.8528
Confusion Matrix:
```

```
[[1589 411]
[ 178 1822]]
(True negative 1589 True positive 1822 False positive 178 False negative 411)
precision recall f1-score support

0.0 0.8993 0.7945 0.8436 2000
1.0 0.8159 0.9110 0.8609 2000

avg / total 0.8576 0.8528 0.8522 4000

AUC: 0.9199
```

Coding up the above in our python notebook, we now have a nice template to perform classification using logistic regression (cell 11).

Part 2 — An Overview of Classification Algorithms.

In our classification code template, we can easily swap out logistic regression for any other sklearn classification algorithm. We can even enumerate all the available classification functions, and run them all on our data via our template:



That is a lot of classifiers. Let's go over some of their key concepts and characteristics:

We are always trying to classify a discrete response *y* given predictors *X*.

1. What *underlying distribution* of *y* given *X* are we modeling? (Logistic regression: linear relationship between predictors and log-odds.)
2. What is the shape of the *decision boundary*? Linear, piecewise linear, nonlinear, what broad functional specification? (Logistic regression: linear decision boundary)
3. What is the *loss function* we are minimizing over the underlying distribution and functional specification? Popular loss functions are mean-squared-error, cross-entropy loss, hinge loss, Huber loss. (Logistic regression: Log loss or cross-entropy)
4. What is the mechanism for controlling *overfitting* and optimizing the bias/variance tradeoff? (Logistic regression: Regularization can be used to apply a penalty to large coefficients and shrink them, biasing the model toward low coefficients[5])
5. And finally, what does the algorithm look like, how do you train it and how does it predict *y* given *X*? What is the *computational complexity* of the algorithm? Is the classifier *generative* or *discriminative*? A generative classifier models the full joint distribution of *X* and *y*; a discriminative classifier uses a less complete model. (Logistic regression is simple and fast, neural networks take a relatively long time to train and tune. Logistic regression just models P(*y*|*X*), so it is a discriminative classifier. [6])

The first 5 classifiers produce linear decision boundaries. On this data set, they produce essentially identical results. They have minor differences in assumptions about underlying distribution, loss functions, and controls for overfitting.

Other classifiers have smooth nonlinear decision boundaries: Quadratic Discriminant Analysis, Multi Layer Perceptron, Gaussian Naive Bayes, our Keras NN (which is similar to MLP).

Still others are piecewise linear or smooth: KNN, Trees, Bagging, Boosting.

K Nearest Neighbors works by finding e.g. the 5 nearest neighbors of any point (k=5). Assign the label of the majority of those 5 neighbors. Choose k based on best performance in cross-validation.

There is a lot of work on boosting models these days, and they win a lot of Kaggle machine learning contests. They generally use *ensembles* of *decision trees*.

Decision trees work like this: Find the rule of the form x or $y > k$ which labels the training observations and gives the lowest average loss. In other words, partition the graph into 2 parts and label one partition 0 and the other 1, using a rule that minimizes the loss function. Next take each partition, and partition it in the way that minimizes the loss function. Continue until no further loss reduction can be achieved in cross-validation. This tends to yield a patchwork of small squares (and will overfit training data if you don't cross-validate carefully).

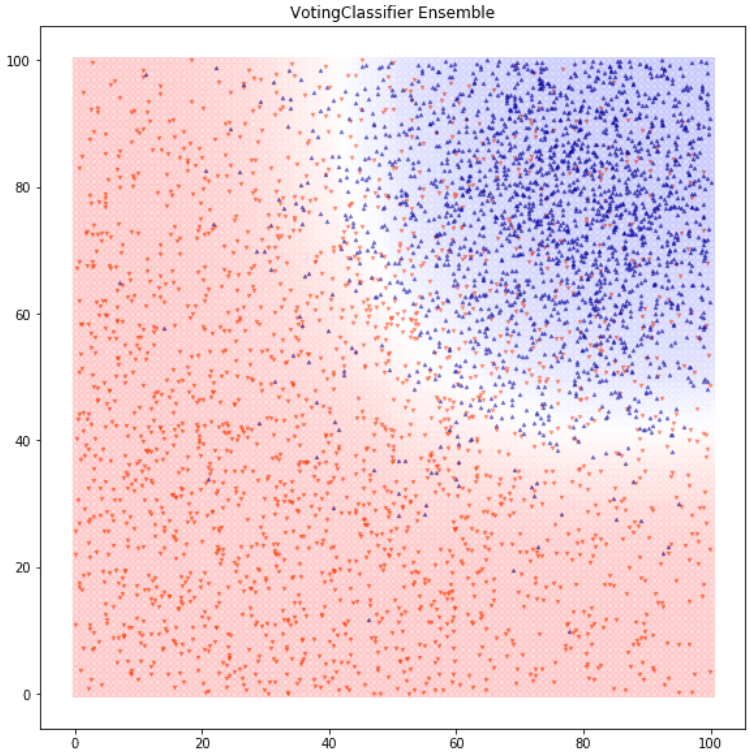
Bagging: instead of using the single best decision tree for your training data 1) build many decision trees using a randomly selected half of your data, and 2) have them vote on the prediction for each observation. This is 'bootstrap aggregation', hence *bagging*. It acts like regularization. A single decision tree will produce the best possible partition of your training data. Bagging will produce a slightly worse result in your training data, but will generalize better out of sample. Many weak rules are better than a single perfect rule, because they are less likely to overfit the training data. We can control overfitting by adjusting the number of trees, the depth of the trees, and the size of the random subsets, and choose the best combination of these *hyper-parameters* using cross-validation.

Random forest: Build trees using both random subsets of data and random subsets of predictors. This prevents the classifier from becoming over-reliant on any particular predictors, and generally works even better out of sample than bagging.

Boosting: Bagging and random forest are ensemble methods which generate many independent weak classifiers that are run in parallel and then aggregated by e.g. voting. Boosting is an ensemble method where you train many classifiers, but in sequence, at each step training new classifiers to work well *on the observations that were previously misclassified*.

Ensembles: after running all available classifiers, we can combine the ones that perform well into an ensemble that votes on the answer. We get > 86% accuracy:

```
In-Sample Accuracy: 0.8608
Out-Of-Sample Accuracy: 0.8648
Confusion Matrix:
[[1652 348]
[ 193 1807]]
(True negative 1652 True positive 1807 False positive 193 False negative 348)
AUC: 0.9263
```



Part 3. Creating a Portfolio Strategy Using Classification.

Now we can apply what we've learned about classification to generate a portfolio strategy.

We start with our dataset from last time: excess returns from 30 industries via Ken French.

We add a couple of additional series that might be predictive[7]:

1. Monthly change in the 3-month Treasury rate
2. Monthly change in the 10-year Treasury rate
3. Industrial production: change from a year ago
4. CPI: change from a year ago
5. Yield curve: 10-year – 3-month
6. Overall market excess return
7. Dummy variables for each month to allow the model to incorporate seasonality

Last time, we used the excess return of each industry as the response variable. We used regression to forecast each industry's excess return by minimizing mean squared error (MSE) using ordinary least squares (OLS) with Lasso predictor selection.

This time, we classify each month's industry returns into quartiles. Each month's worst performing 8 industries are assigned label 0, next 7 label 1, next 7 label 2, and finally the best performing 8 get label 3. This 4-level discrete label is our response variable: the quartile the industry will fall into.

We can now run a classification algorithm to classify the response variable using the previous month's predictors, and predict the quartile.

We produce forecasts for our forecast period using walk-forward cross-validation. We have 68 years of data. We divide our timespan into 5 folds. We train on the first fold (the first ~14 years) and predict the 2nd fold (the next ~14 years). Then we train on the 1st and 2nd folds (the first ~27 years) and predict the 3rd fold, etc. In this manner we get out of-sample predictions on most of our data and control better for serial correlation than picking our fold data at random.[8]

Our classifier returns probability estimates for each quartile. For each industry in each month it returns the probability it will belong to each quartile: 4 probabilities that sum to 1.

How do we turn those 120 forecasts each month into an optimal portfolio strategy?

Our chosen approach is to calculate the average predicted class weighted by probability. This gives us an 'expected quartile', which would be 1.5 if the prediction was 50% quartile 1 and 50% quartile 2.[9]

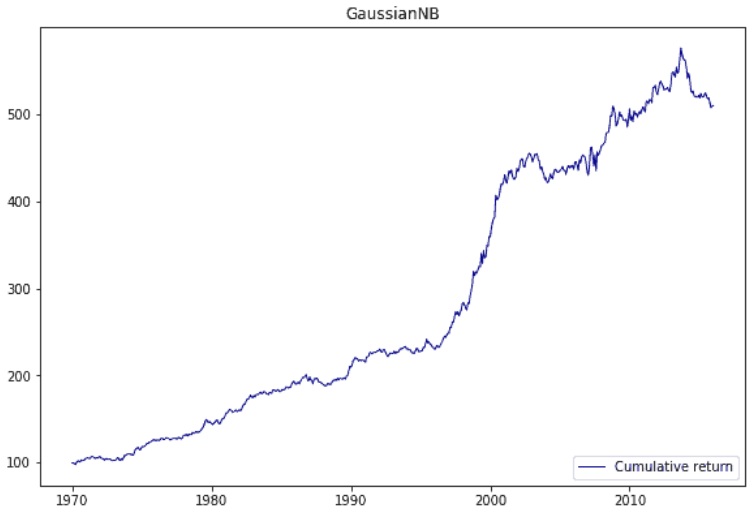
We create a long short portolio using top *n* long and bottom n short. We choose the *n* that gives highest Sharpe ratio in cross-validation.

Following this procedure, we cross-validate classification algorithms.

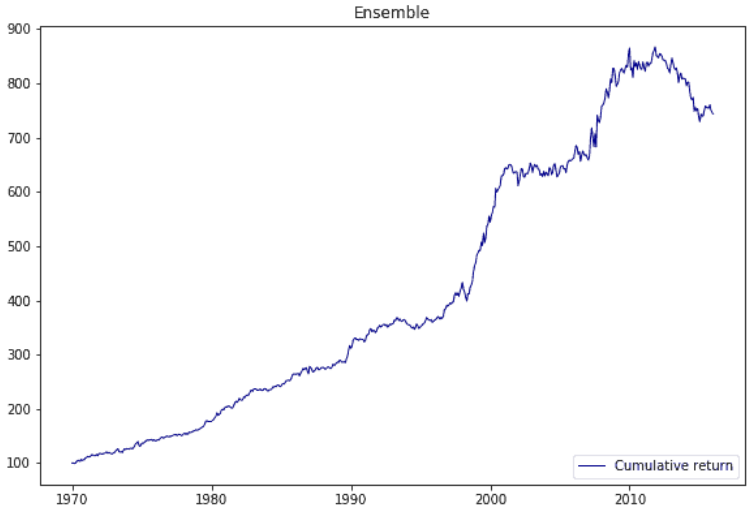
	Name	In-Sample Acc	OOS Directional Acc	Kendall's tau	Ann. Return	Vol	Sharpe
12	Gaussian Naive Bayes	0.342	0.365	0.036	0.033	0.040	0.828
15	Sklearn Ensemble	0.855	0.376	0.042	0.043	0.053	0.824
11	Bernoulli Naive Bayes	0.335	0.375	0.036	0.040	0.067	0.632
5	AdaBoost	0.466	0.361	0.023	0.017	0.027	0.631
0	Logistic Regression	0.412	0.377	0.036	0.020	0.033	0.628
2	Gradient Boost	0.854	0.380	0.025	0.028	0.047	0.627
10	ExtraTreesClassifier	0.855	0.377	0.026	0.024	0.040	0.610
4	LGBMClassifier	0.854	0.372	0.026	0.022	0.040	0.578
8	RandomForestClassifier	0.843	0.368	0.021	0.019	0.038	0.514
16	Keras Neural Network	0.305	0.378	0.032	0.030	0.063	0.511
13	KNeighborsClassifier	0.460	0.366	0.016	0.019	0.042	0.479
6	BaggingClassifier	0.842	0.371	0.016	0.017	0.040	0.435
7	DecisionTreeClassifier	0.852	0.372	0.010	0.015	0.038	0.419
1	Multilayer Perceptron	0.837	0.375	0.016	0.023	0.067	0.380
3	XGBClassifier	0.281	0.386	0.007	0.010	0.042	0.267
9	ExtraTreeClassifier	0.854	0.369	0.009	0.001	0.028	0.068
14	Support Vector Machine	0.856	0.347	-0.001	-0.000	0.039	0.016

Our best single algorithm is Gaussian Naive Bayes, which generates an annualized return of 3.3% with a monthly Sharpe ratio of 0.828.

In a full backtest, we re-estimate the model each month and forecast the following month, instead of estimating just 5 folds in walk-forward cross-validation, and we obtain an annualized return of 3.45% and a Sharpe of 0.794.



Finally, we construct an ensemble of the top algos. On a full backtest we obtain a 3.45% return with a monthly Sharpe ratio of 0.892. (Better accuracy metrics, same return, better Sharpe.)



Conclusion

What have we learned?

1. The fundamentals of classification
2. How to perform and interpret logistic regression
3. Key characteristics of different classification algos
4. How to create an ensemble that combines several classification algorithms for improved results
5. How to apply classification to create a portfolio strategy

We obtained better results than with last time's Lasso regression. This improvement is from adding additional predictors and doing cross-validation on *n*, the number of industries to go long/short. It shouldn't be interpreted as superiority of classification.

Our motivation for trying classification was the observation that improvements in MSE didn't translate directly to Sharpe, and that an equal-weighted portfolio selection is a classification problem: some securities are long, some are short, some are omitted.

But after implementing this classification approach, I would generally never recommend bucketing a continuous variable into quartiles or classes and forecasting via classification instead of regression.

First, you are removing information from your response variable when you transform it into quartiles. Bucketing doesn't generally help forecasting.

Second, classification doesn't care about the order. Vanilla categorical cross-entropy doesn't care if you misclassify 1st quartile as 2nd or 4th. It just cares how big a probability you predicted for the correct label.[10]

If the real-world variable you care about is performance, then forecast performance using regression.

However, you may come across a data set that has earnings beat/miss, quartiles, or some other discrete data, in which case classification is the way to go.

Classification is a fundamental building block of modern machine learning. When you chain many neural network layers and train them end-to-end with a categorical cross-entropy loss, you get deep learning and complex emergent behavior, like classifying audio patterns as spoken phonemes and words, classifying images as 'bird' or 'house', classifying objects as 'stop sign' or 'deer in the road' in autonomous driving.

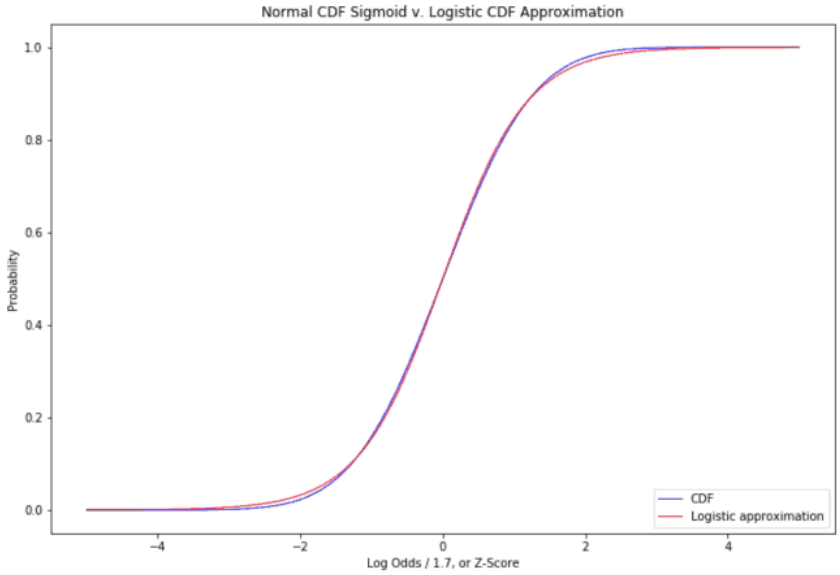
I hope this gives a sense of the power of modern classification algorithms and a practical roadmap for applying them to financial problems.

References

1. ↑ The distance from the line defined by $ax + by + c = 0$ to a point (x_0, y_0) is $ax_0 + by_0 + c/(\sqrt{a^2 + b^2})$. Since $\sqrt{a^2 + b^2}$ is constant with respect to x and y, $ax + by + c$ has a linear relationship to the

classification margin.
The red/blue shadings define a plane if you visualize color intensity in the z-axis, the blue areas are positive, the red areas are negative, with z = 0 at the decision boundary. (Although our colorings are based on logistic_function(Z), not Z itself)

2. ↑ To match Vegas odds, we have to take the inverse of the way they are normally quoted, or take $1 - LogisticFunction(log(VegasOdds))$ but you get the idea.
3. ↑ This is not the only possibly 'squashing' function. We can use any function that maps the interval [-∞, +∞] to the interval [0,1]. We could use sigmoid(straight odds) instead of sigmoid(log-odds). We could use the normal cumulative distribution function (CDF) which maps the z-score to the area under the normal distribution up to that z-score — our familiar statistical significance tables. If you take the derivative of the normal CDF function, you get the probability density of the normal distribution. If you take the derivative of the logistic function, you get the probability density of the logistic distribution: the logistic function is the CDF of the logistic distribution. If we use the normal CDF as our sigmoid function, we are doing a probit model. If we use our logistic sigmoid, we are doing a logit model. The choice of sigmoid reflects the distribution you are modeling. In machine learning, we generally use the logit model. If you have a good reason to believe you are modeling a normal distribution, and the distance from the decision boundary has a linear relationship to the z-score, then the probit model is technically correct (the best kind of correct). If you don't know your distribution is normal but have good reason to think log odds can be modeled as a linear function of predictors, the logit model is correct. The logit model is computationally simple to optimize, the log-odds relationship is a reasonable assumption even when the modeled distribution is not normal, and in practice the two are very similar:



4. ↑ Exclusive since our sigmoid is never exactly 0 or 1 except as a limit when the decision function goes to -∞ or +∞.
5. ↑ L2 regularization will minimize the sum of the log loss and the squared coefficients multiplied by some constant λ. It will tend to shrink the coefficients towards 0 and reduce the impact of outliers. It will also tend to equalize the magnitude of the coefficients, biasing the model toward a 45° line: since the derivative of x^2 is 2x, shrinking a much larger coefficient will have much more effect on the overall loss.
6. ↑ If it modeled P(*X*) and P(*y*|*X*), which are sufficient to fully specify P(*y*) and P(*X*|*y*) then it would be a generative classifier.
7. ↑ These are arbitrary but the idea was to give the model some indication of financial conditions and where we are in the business cycle. There is really no need to add additional data for demonstration purposes, but we do it to see if we can improve on our result from last time. By itself classification doesn't seem to obtain any improvement vs. regression. Adding these variables also gives us an example of how you can get this data from FRED and preprocess it.
8. ↑ Often in financial time series, we get n observations which represent fewer than n independent observations because they are serially correlated. In the extreme, consider if a factory manager decides weekly how many widgets to produce that week and spreads production out evenly over the days of the week. The daily widget production series is the same number each weekday. If we were to cross-validate by putting aside a random 20% of a year's observations, each of those observations would be duplicated in the training data. We would under-estimate the real-world error. Walk-forward cross-validation is closer to a real-world process where we only use data we have seen up to the present.
9. ↑ Why do we do this? You could argue this turns our classifier into a regressor. You would be correct.
We could assign each stock to the highest probability quartile. But suppose you get 0.41,0.1, 0.1, 0.39? Assigning to quartile 0 is not necessarily ideal. Furthermore, if you put the predicted top-quartile stocks into your portfolio, there is no guarantee you get 8 stocks each month. You might even get 0 stocks assigned to top quartile, the classifier could assign everything to the middle quartiles. Unfortunately, our classifiers are not order-aware, classes are just like blue and red. Ordinal regression is order-aware and can be used to predict rank, but unfortunately not included in the vanilla sklearn classifiers.
10. ↑ I attempted to remedy this issue in the Keras classifier with a custom loss function. It weights the log loss on

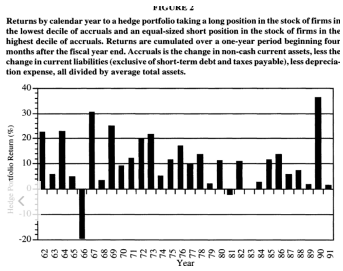
each class, assigning larger losses to large misses. One would expect slightly worse 4-quartile accuracy but better Kendall's tau and Sharpe. It performed slightly but not significantly better on those metrics. The Keras model is slow to train and hard to integrate into a voting classifier ensemble. Ordinal regression or directly optimizing performance may be better approaches to this problem than classification.

Learn Something? Share the Knowledge!

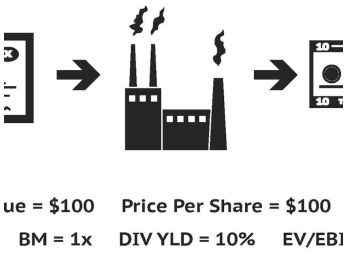
About the Author: Druce Vertes

Druce Vertes, CFA is founder of StreetEYE, a leading financial news aggregator, and previously worked as a consultant and IT executive for leading hedge funds.

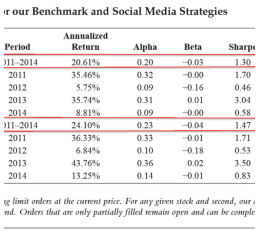
Related Posts



[Accruals Momentum as an Investment Strategy]
August 16th, 2018 | 0 Comments



[Buybacks: Why They Don't Matter, Why They Do, and Why You Should Care Yet Still Relax.]
August 30th, 2018 | 0 Comments



r our Benchmark and Social Media Strategies

Period	Annualized Return	Alpha	Beta	Sharpe
011–2014	20.61%	0.20	−0.03	1.30
2011	35.46%	0.32	−0.00	1.70
2012	5.75%	0.09	−0.16	0.46
2013	35.74%	0.31	0.01	3.04
2014	8.81%	0.09	−0.00	0.58
011–2014	24.10%	0.23	−0.04	1.47
2011	36.33%	0.33	−0.01	1.71
2012	6.84%	0.10	−0.18	0.53
2013	43.76%	0.36	0.02	3.50
2014	13.25%	0.14	−0.01	0.83

g limit orders at the current price. For any given stock and second, our
nd. Orders that are only partially filled remain open and can be comple

[Fintwit Might Matter for Mc and Mean Reversion in Stoc]
August 27th, 2018

GET UPDATES NOW

I have read and agree to the Terms & Privacy Policy. Alpha Architect will use your information to provide blog updates and for email marketing.

Your email address

Get Blog Updates

FOLLOW US AND LEARN

RECENT TWEETS

A diverse work group improves performance. See our latest post!

Popular | **Recent**

The Robust Asset Allocation (RAA) Index
December 2nd, 2014

Avoiding the Big Drawdown with Trend-Following Investment Strategies
August 13th, 2015

The Quantitative Momentum Investing Philosophy
December 1st, 2015

IMPORTANT DISCLOSURES

Performance figures contained herein are hypothetical, unaudited and prepared by Alpha Architect, LLC; hypothetical results are intended for illustrative purposes only. Past performance is not indicative of future results, which may vary. There is a risk of substantial loss associated with trading stocks, commodities, futures, options and other financial instruments. Full disclosures here.