#h1 Chapter 15 - Classification

#h2 Or why logistic regression is not regression

#h3 15.1 - Introduction

#pg Congratulations, we've worked through how to find the best possible parameters for a linear regression model, given our training data. We can use the most basic statistics approach, which is straightforward to implement, even in Excel. The ordinary least squares matrix closed-form solution is fast, easy to compute with the right numerical toolboxes (e.g., NumPy), and deterministic. In addition, it has many use cases beyond machine learning. My first exploration of any data set will typically be using that equation. It gives a good sense of what your data is about, even though more complex algorithms outperform it substantially. Finally, the gradient descent solution we build here is a trivial use of gradient descent. It does not show off its elegance and strength as an algorithm. Gradient descent and backpropagation are still the core algorithms in training machine learning models. Now that we have introduced these new concepts, we can move from regression to classification. Most of the methods remain the same, but we need to make a few minor tweaks to our algorithms to make them suitable for classification problems. Specifically, we need different metrics of success, loss, error, and accuracy. To explore all that, we will implement the most well-known classification algorithm: logistic regression (and yes, it is a somewhat unfortunate name).

#h3 15.2 - A new activation function

#pg In the previous chapter, we used linear algebra to predict an outcome value from inputs, transforming this input using trained parameters or weights. The input and parameters form a linear system, where the resulting output is a weighted sum of the inputs with the parameters providing this weighting. Because of this, the resulting output was a continuous value. The first step in transitioning to classification is transforming such a continuous output into something discreet and binary. Several functions will accomplish this, but the most commonly used one is the sigmoid function. At zero, this function yields a value of 0.5, which we can interpret as the predicted probability of class A (and, by extension, class B, since pA = 1 - pB because pA + pB = 1). Values greater than 0 will increase the probability of one class over the other, but this asymptotes out at 1. Similarly, negative values will eventually asymptote to 0. The sigmoid function conveniently maps the linear outputs with a domain R (all real numbers, negative and positive) to a probability between 0 and 1. What is left is to simply threshold the value. Anything input yielding a probability >= 0.5 is predicted as class 1 (positive class), and anything smaller as class 0 (negative class). However, since we are now outputting probabilities rather than continuous values, we also need to rethink our loss function: the measure of a discrepancy between predicted and actual outcomes.

#im ../assets/figures/015/015-01.png 50 256

#im ../assets/equations/015/png/015-01.png 50 64 Equation 15.1 - The sigmoid function equation. The sigmoid function maps any continuous variable R to an output bound by [0,1]. We use it to transform the linear component (the inputs weighted by their parameters) into probabilities. Thresholding the resulting probabilities yields a binary prediction.

#h3 15.3 – A new loss function

#pg In the previous chapter, we defined our loss function as the squared error between a model's prediction and some ground truth value. The underlying assumption for this was that the error had a normal distribution. When we change our problem statement to classification, it is easy to see why this normal distribution no longer makes sense. Classification algorithms produce probabilities that are thresholded to yield classifications from input data. Unlike regression, where we have a single output that can be compared directly with the ground truth by way of its difference, in binary classification, we have two probabilities, one for each prediction. For example, in regression, we might predict the asking price for a house based on its square footage, the number of rooms, and location. The difference in the asking price predicted by our algorithm and the actual asking price defines how well the algorithm performs during training. The more it deviates, the more we adjust the weights to correct for it. However, if we were to use classification to predict whether a house has more than four rooms based on its asking price, we obtain two probabilities for each scenario: the house has more than four rooms (pA), or it does not (pB). By thresholding these output probabilities, we obtain an actual prediction. But the absolute difference between prediction and actual outcome in classification is a bit meaningless: 0 for correct predictions and 1 for incorrect predictions. First, these errors are not normally distributed, as they are discreet. Second, we would prefer a loss function that tells us not just that the algorithm was incorrect but also by how much it got it wrong. In this way, when we update the parameters during learning, they are updated in proportion to how poor the prediction was. Imagine that our model predicts pA = 0.6 and pB = 0.4. In this case, the algorithm favors one decision over another, but it isn't necessarily all that sure about it, as the two probabilities are close to 0.5. Outputting probabilities close to 0.5 is a classification model's way of saying 'I am essentially guessing'. Suppose that the actual outcome in this scenario was that the house had more than four rooms. If we had thresholded the two probabilities, our model would have predicted this was the case, and it would have been correct. Our model would have also been correct if the probabilities it outputted were pA = 0.9 and pB = 0.1. And both would have been incorrect if the house did not have more than four rooms. Institutively, I hope you see how being correct isn't necessarily the whole story of how well our model did. We should factor in, somehow, the model's confidence in its prediction as well. Getting a prediction wrong when the algorithm is essentially guessing is not all that bad. We can't fault it for not having enough signal in the data if there isn't any. But suppose it seems very certain about its prediction (probabilities close to 1). In that case, it should be penalized more strongly if it predicted the outcome incorrectly and rewarded more heavily if it did get it right. There is a way to compute this confidence-weighted error: the log loss function.

#im ../assets/figures/015/015-02.png 50 256

#im ../assets/equations/015/png/015-02.png 50 16 Equation 15.2 - The log loss function. In this figure, we consider two possible ground truth scenarios in which our classifier's actual predicted output should have been 0 (gray curve) or 1 (black curve). If the probability of us predicting 1 is low (and thus high for predicting 0), our log loss function returns a substantial loss when the actual outcome should have been 1 but almost no loss when the actual outcome should have been 0. As the probability of predicting 1 increases (towards the right of the graph), this scenario reverses. The loss obtained using this function is thus proportional to our error (i.e., the difference between our prediction, our confidence in it, and the actual outcome).

#br

#pg It might look a little complicated at first but plotting its behavior as a function of both the probabilities of the model's output and the ground truth shows its favorable properties. Each curve represents one of the two ground truth scenarios: the actual outcome was 1 (black) or 0 (gray). When both the predicted and actual outcome are the same, the resulting loss for that data point is 0. However, the more the probability produced by the model points towards the opposite outcome (0), the higher the associated loss. The logloss function thus guarantees we penalize the model proportionally to how incorrect it was, not just that it was incorrect.

#h3 15.4 - Introducing Logistic Regression

#pg These two changes to the gradient descent algorithm we built in the previous chapter are all that is needed to move from regression to classification. We can and should use all the same techniques previously discussed. We divide our data into training and testing data to prevent overfitting. Finally, we need to decide on the learning rate, the amount of regularization, and other hyperparameters we need to fine-tune to find the best possible model. However, one crucial difference between this algorithm and the linear models we worked with in the previous chapters is that the loss landscape is not convex. This nonconvexity means there is no guarantee that there is 1) only one optimal solution and 2) that we will find it. There is the possibility that our gradient descent algorithm will get stuck in local minima. In addition, each random set of initial weights can lead to very different models with varying performance levels. More importantly, since the statistical and algebraic methods need the loss landscape to be convex, we cannot use them here. With convexity, only one of the three methods we used in the previous chapter is applicable here: gradient descent. The combination of gradient descent and our new log loss function transform our linear regression algorithm from the previous chapter into a classification algorithm: logistic regression. As you can tell, it is a bit of an unfortunate misnomer since it is not a regression algorithm. It is, however, one of the core algorithms in machine learning and the first any textbook on machine learning will discuss in the context of classification. This course is no exception. This isn't just because it is a robust yet easy-to-implement algorithm. Mathematically, it has many similarities to deep belief neural networks, which we will explore later, so it provides a useful starting point. One way to imagine a modern DBNN is a neural network in which each node is essentially a tiny logistic regression algorithm. Training a newly developed logistic regression model follows the approach outlined in Chapter 14 for linear regression. The only difference is in how we define our loss function and the extra step we take to transform the output of the linear operation on our data by the model using the sigmoid function. We have included a few demo notebooks that implement logistic regression at various levels of coding complexity and with different learning strategies (conventional, batch, and stochastic gradient descent).

#h3 15.5 - Measures of success

#pg Another important difference between regression and classification is our measure of success. Understanding how well a classification algorithm performs is a bit more complicated than you might think. Biased algorithms often result from a focus on the wrong performance or success metrics. Models can perform exceptionally well, or so it appears, using one metric but not another. When we release such models into the wild, they might be faulty, unreliable, and even dangerous since we measure their performance with the wrong yardstick. If you are not keen on all the mathematics we have used so far in this book, and you do not think you will find yourself using or implementing any of the methods, you can skip many sections that work through these technical details. But if you are interested in machine learning and AI, work closely with people who build and deploy them, and want a better understanding of how AI can introduce dangerous biases that have real-world consequences, we recommend working through the following few sections in particular. We want to encourage critical thinking when anyone makes bold claims about their new AI solution. You will begin to question flashy headlines like: 'new algorithm accurately predicts the presence of an extremely rare cancer doctors were unable to diagnose before!'. And for a good reason.

#h3 15.6 - Measures of success - When accuracy falls short

#pg So, what might we get wrong in trying to understand model performance? There is a natural tendency to evaluate a classification algorithm based on its accuracy. Surely something with an accuracy close to 100% must be a good algorithm. Similarly, if it doesn't do better than chance, what is the point of having it? Things are far more subtle. Consider that the before-mentioned algorithm is in charge of detecting an extremely rare form of cancer through some form of measurement (medical imaging, biopsy, etc.). Suppose I was particularly lazy and nefarious at the same time. In that case, I could build an algorithm that returns the same answer no matter the input: the diagnosis is negative. And the algorithm would be right almost all the time since this cancer is so rare, it is often not present to begin with.

#h3 15.7 - Measures of success - False, True, Negative and Positive

#pg This imbalance between classes, where one class occurs with a much higher frequency than the other, is a problem for classification algorithms, but there are ways around it. But we should start by unpacking what it means to make a 'correct' or 'incorrect' prediction. On closer inspection, a binary classification algorithm can be correct in two ways and wrong in two ways. A true positive is when the algorithm correctly predicts the presence of something, like a tumor in an MRI scan. But it can also correctly predict the absence of that same thing: a true negative. Likewise, an algorithm can get it wrong by predicting something that is not present (a false positive) and get it wrong by failing to do when it is present: a false negative). This creates a 2x2 matrix of these four possibilities in a binary classification task. For a multiclass problem, it is called a confusion matrix, where the matrix is n x n, with n being the number of classes. In the binary case, we can compute two additional measures that give us a better insight into the algorithm's performance compared to accuracy: precision and recall.

#im ../assets/figures/015/015-03.png 50 256 Figure 15.3 - Precision and Recall. In a binary Boolean classification task, there are four possible outcomes, two correct and two incorrect. True positives indicate the classifier correctly predicts the presence of an outcome, whereas true negatives correctly predict the absence. False positives mean the classifier incorrectly indicates the presence of the outcome, and a false negative when the classifier wrongly assumes the absence of the outcome. We can determine our classifier's precision and recall by combining these outcomes.

#h3 15.8 - Precision and recall

#pg By dividing correct and incorrect predictions into these four possibilities (true and false positives, true and false negatives), we can compute two new metrics: precision and recall. Imagine I am writing a paper on topic X. To do so, I need to find existing papers on that topic to reference them. Using some search engine, I ask for all papers on topic X. It could be that topic X might be a little vaguely defined or broadly scoped, so the search engine will have to decide which to include and which to exclude. We can evaluate the returned list in two ways. First, how complete is the list? Did the search engine accidentally leave some articles out that are on topic X? This ratio, between the list of articles I got and the entire list of articles on topic X is called recall: the total number of existing relevant items retrieved. In contrast, the search engine may have accidentally returned some articles that are not actually on topic X. This ratio between all the articles returned and those that were retrieved and were actually about topic X is called the precision. The difference is subtle but important, especially in healthcare. The CDC recommends yearly mammograms for women over a certain age, but the problem is that mammograms often 'cry wolf'. Their precision is low: many false positives are reported, which, as you can imagine, can cause extreme anxiety in people before additional testing proves it was indeed a false positive. The opposite approach is perhaps best phrased as 'better safe than sorry'. Yes, we do end up scaring patients by predicting many false positives. But it also decreases the odds of predicting a false negative: a negative diagnosis even though the cancer is present. You can tell both by the math and probably also by intuition that a shift towards better recall will always degrade our precision and vice versa. It is up to you and the domain experts to decide what is more important. Luckily shifting the importance of either measure is easily achieved by changing the threshold we apply to the probabilities our algorithm puts on events. We can demand at least a probability of 0.8 before we decide the event is indeed present, which increases our precision. Or we can loosen our criteria and set the threshold to 0.2 to make sure we do not miss any relevant events, which gives us an increase in recall.

#h3 15.9 - F1-Score

#pg Although there are several metrics that we can compute from the 2x2 matrix of predicted and actual outcomes, the most commonly used one worth mentioning here is the F-score. The F1-score combines the precision and recall of a classifier into a single metric by taking their harmonic mean. We primarily use it to compare the performance of two classifiers. If that classifier A has a higher recall and classifier B has higher precision, which one performs the best? By weighing precision and recall, the F-score provides a way to answer that question to some extent.

#im ../assets/equations/015/png/015-03.png 50 32

#im ../assets/equations/015/png/015-04.png 50 32

#im ../assets/equations/015/png/015-05.png 50 32 Equations 15.3-5 - Various measures of success associated with classification

#h3 15.10 – Area under the ROC curve

#pg A final metric that summarizes a classification algorithm's performance is the so-called area under the ROC curve. ROC stands for receiver-operator curve and has its roots in information theory. Electrical and radar engineers first developed it during World War II. ROC analyses since then have been used in experimental psychology, medicine, radiology, biometrics, the forecasting of natural hazards, meteorology, model performance assessment, and machine learning. It aims to measure and illustrate a binary classifier system's diagnostic ability as its discrimination threshold varies. Imagine that we have two phenotypes that differ in whether they are at risk for a particular cancer. In addition, we have some other data we think might predict whether you are from one of these phenotypes, say BMI. Nature being nature, we expect intrinsic noise, where people form a continuum of the predictive signal, but they still represent two distributions. Imagine that population B is at risk for cancer while population A is not. For example, we might assume that their height predicts the cancer phenotype. But when we plot these two distributions of height, they overlap completely. That means that we will be at chance no matter where we place our threshold that will predict whether someone belongs to group A or B.

#im ../assets/figures/015/015-04.png 50 256 Figure 15.4 - Area under the ROC curve. (A) We aim to build a classifier that uses height to predict whether an individual is at increased risk for a particular cancer. This can work if the distribution of heights for the group at increased risk is statistically different from the one not at increased risk. Panel (A) shows that with a greater difference in mean, the two distributions start to separate, allowing for increasingly more accurate predictions. (B) By varying the threshold for predicting whether an individual is at risk based on their height, we can create ROC curves for each of the four levels of separation seen in (A). Indeed, as the two distributions begin to separate more clearly, the area under the curve increases, indicating true predictive power.

#br

#pg An ROC curve goes a bit further. Remember that we use the sigmoid function to transform the linear output of our classification model into an output probability and thresholding that probability to a binary output. However, by varying our decision threshold in small increments, we can compute two metrics at each threshold: the true positive (TPR) and false positive rates (FPR). The true positive rate is the ratio between the true positives (ypred = 1 and y = 1) and the total number of positive outcomes (y = 1). The false positive rate is the ratio of the false positives (ypred = 1 and y = 0) and the total number of negative outcomes (y = 0). If we plot these two values for a range of thresholds between 0 and 1, we will get a curve. What does this curve look like when we have no predictive power at any threshold? At low thresholds, we will always predict 1, so we have high false and true positive rates because we always predict a positive outcome. At a high threshold, our positive rates (false or true) drop to zero, as we now seldom predict a positive outcome. This is true whether or not we achieved some accuracy in discriminating between these two populations. The key is to look at the threshold at 0.5. This is where we would ideally like to see our algorithm perform well and produce many true positives and few false positives. But if there is no predictive power, this ratio will remain constant. And this is true no matter where we place the threshold. Graphically, this means that all points fall on the diagonal of TPR = FPR, creating a straight line from (0,0) to (1,1). Now imagine we have two distributions that are separated, indicating they do differ in terms of phenotype based on our metric. In this scenario, we still expect the two edge points where the threshold is either 0 or 1 to be at (0,0) and (1,1). However, for a threshold of 0.5, we would now see what we had hoped for: the ratio between true and false positives is leaning heavily towards true positives. The more this is true, the more our resulting curve across many thresholds deviates from the diagonal, its curvature more pronounced and stretched into the left-upper corner where we have the high ratio of true positive vs. false positives. We quantify this as the area under the curve. And as our algorithm improves, this area will increase in size. In other words, as the separation in the mean between the two distributions increases, the area under the resulting ROC curve also increases.

#h3 15.11 - Underfitting and Overfitting

#pg Like regression, we should take care to prevent both underfitting and overfitting. Classification is not different from regression in how it mitigates these dangers. The best approach across all machine learning algorithms is to have independent datasets for training, testing, and validation. What constitutes test and validation isn't always clear, and the terms are used loosely. We adhere to the definition outlined by Russell and Norvig: A training data set is used to compute gradients and optimize our model's parameters (weights). On each iteration, whether this is an update from a single data point, a mini-batch, or a pass through the entire training data, we use a validation data set to compute the loss function that tells us how learning is progressing. And depending on our approach, what constitutes training and test data can change. If we divide the data beforehand: one bit of our data will go into changing our parameters, and the other bit will determine the necessary performance metrics like loss and accuracy. Finally, when applying the algorithm to the validation data yields acceptable results, we once more run that algorithm, but now on the completely untouched and independent test data set. This will yield a final and objective measure of success and quality without any of the test data having been exposed or used during optimization.

#h3 15.12 - Imbalanced classes

#pg A common problem when training a classification algorithm is the imbalance between the number of training examples you have for each class. Some events we are interested in predicting might indeed be quite rare. As we have discussed before, we should always be careful interpreting the output of these models and consider the apriori probability (the probability of the things we are trying to predict independent of the input data the model uses). But there is an additional issue: our learning algorithms might also get confused. Remember that we are trying to minimize the loss: the overall error of our algorithm. But in an imbalanced data set, that error will be driven by the dominant class, and important but infrequent signals to a rare event are lost in the computations to optimize the parameters. This can lead to algorithms that underperform or even worse. The most stringent approach is to balance your data set using the class with the lowest number of examples. This is a sound approach but potentially removes a lot of useful data from your data set. Other methods can weigh data points according to their count or adjust the loss function to account for the imbalances. We have seen instances where data sets were balanced by creating copies of the outnumbered class to match the other classes. This is not correct. Yes, you are adding data points. But you are not adding new independent data. As a result, the statistics of this expanded data set are misleading to both you and the algorithm. We should not trust the resulting model.

#h3 15.13 - Multiclass problems

#pg So far, we have only dealt with examples of binary classification tasks. In real-world situations, we often classify inputs into more than two categories. We can accomplish this by modifying the algorithm itself, but this will have our discussion encroach on much of what we will want to discuss in the context of neural networks. Neural network architectures can easily handle multiclass classification tasks. It only requires adding nodes to the output layer, one for each class in our data. However, when we use logistic regression to predict one of multiple classes, we have to approach it a little differently. We can do multiclass logistic regression using two strategies: one vs. all and one vs. one. The first method has us build as many logistic regression models as there are classes. And for each model, we pick one out of n classes as the positive example in a binary classification task, while all the classes represent negative examples. In a three-class situation, we thus create three models: Class A vs. Class B and C, Class B vs. Class A and C, and Class C vs. Class A and B. Training the models is not different from how we implemented it earlier. The only thing to note is that this approach will create an imbalance between classes, at least for some of the models we are training. Prediction is where we must do things a little differently. One method is to pick the model that assigns the highest probability to its class. In this scenario, we might end up with a model that by itself might not even predict our class but compared to the other models, still assigns it the highest probability. In another approach, referred to as one-vs-one, we train as many models as there are unique pairs of classes. In a four-way classification problem, we thus create six models (A vs. B, A vs. B, A vs. C, B vs. C, C vs. D, and B vs. D). To identify the predicted class in this situation, we now use a majority vote: which class was predicted as the most probable class most often?

#h3 15.14 - Other Classification Algorithms

#pg Some other classification methods you might come across are ridge regression, k-nearest neighbors, and support vector machines. Support vector machines (SVMs) were all the rage a few years ago because they could handle classification problems that seemed out of reach for logistic regression. Whereas logistic regression tries to find a decision boundary that yields the lowest loss (and thus the best accuracy), SVMs go one step further and map training examples to points into a space to maximize the width of the gap between the two categories. This mapping can be linear, making the SVM linear, but it can also utilize what is known as the 'kernel trick'. The kernel trick is a nonlinear mapping of inputs into high-dimensional feature spaces. K-nearest neighbors is a good algorithm if you suspect or know there is no global separation between your classes, but there might be local clustering of data belonging to the same class. K-nearest classifies a point by finding its closest neighbors and setting the class of the new data point to whatever class is most common amongst its nearest neighbors. Finally, ridge regression is a useful tool when you have many regressors compared to the number of available training data and in situations where your data shows multicollinearity (the regressors show high correlations between them). Although technically a regression method, we can use it for classification.

#h3 15.15 - Conclusion

#pg Despite neural networks gradually taking over most of machine learning, logistic regression is still widely used. First and foremost, as an educational tool to teach the fundamentals of machine learning, such as loss functions and gradient descent. But also because it just performs so well for so many different datasets. It is easier to implement, faster to train, and doesn't require the amount of data needed by a neural network.

#h3 15.16 - Demo Notebooks

#pg You can find a few notebook to accompany this chapter, implementing a variety of classification machine learning algorithms

#h3 15.17 - References

#bs

#be

#bp Russell, S., Norvig, P. (2016). Artificial Intelligence: A Modern Approach. CreateSpace Independent Publishing Platform.

#bp Gurnsey, R. (2017). Statistics for Research in Psychology: A Modern Approach Using Estimation. United States: SAGE Publications.

#bp Michalski, R. S., Banerji, R. B., Anderson, J. R. (1983). Machine Learning: An Artificial Intelligence Approach. Germany: Tioga Publishing Company.

#bp Murphy, K. P. (2012). Machine Learning: A Probabilistic Perspective. United Kingdom: MIT Press.

#bp Hespanha, J. P. (2018). Linear Systems Theory: Second Edition. United Kingdom: Princeton University Press.

#bp Provost, F., Fawcett, T. (2013). Data Science for Business: What You Need to Know about Data Mining and Data-Analytic Thinking. United States: O'Reilly Media.

#bp Rieke, F., Warland, D., Bialek, W., De Ruyter Van Steveninck, R. (1999). Spikes: Exploring the Neural Code. United Kingdom: MIT Press.

#bp Van der Plas, J. (2016). Python Data Science Handbook: Essential Tools for Working with Data. United States: O'Reilly Media.

#be