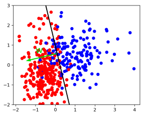
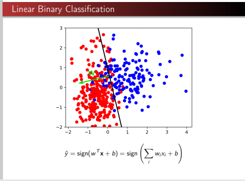
Linear Binary Classification



$$\hat{y} = \text{sign}(w^T\mathbf{x} + b) = \text{sign}\left(\sum_i w_i x_i + b\right)$$

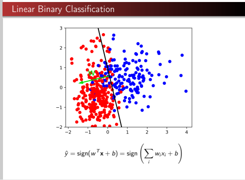
We'll first start with linear models for binary classification, so if there are only two classes. That makes the models much easier to understand.
Similar to the regression case, basically all linear models for classification have the same way to make predictions. As with regression, they compute an inner product of a weight vector w with the feature vector x, and add some bias b. The result of that is a real number, as in regression. For classification, however, we only look at the sign of the result, so whether it is negative or positive. If it's positive, we predict one class, usually called $+1$, if it's negative, we predict the other class, usually called -1. If the result is 0, by convention the positive class is predicted, but because it's a floating point number that doesn't really happen in practice. You'll see that sometimes in my notation I will not have a b. That's because you can always add a constant feature to x to achieve the same effect (thought you would then need to leave that feature out of the regularization). So when I write $w^T x$ without a b assume that there is a constant feature added that is not part of any regularization.



Linear Binary Classification

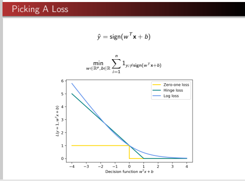
$$\hat{y} = \text{sign}(w^T\mathbf{x} + b) = \text{sign}\left(\sum_i w_i x_i + b\right)$$

Geometrially, what the formula means is that the decision boundary of a linear classifier will be a hyperplane in the feature space, where w is the normal vector of that plane. In the 2d example here, it's just a line separating red and blue. Everything on the right hand side would be classified as blue by this classifier, and everything on the left-hand side as red.


Linear Binary Classification

$$\hat{y} = \text{sign}(w^T\mathbf{x} + b) = \text{sign}\left(\sum_i w_i x_i + b\right)$$

So again, the learning here consists of finding parameters w and b based on the training set, and that is where the different algorithms differ. There are quite a lot of algorithms out there, and there are also quite a lot in scikit-learn, but we'll only discuss the most common ones.
The most straight-forward way to approach finding w and b is to use the framework of empirical risk minimization that we talked about last time, so finding parameters that minimize some loss on the training set. Where classification differs quite a bit from regression is on how we want to measure misclassifications.


Picking A Loss

$$\hat{y} = \text{sign}(w^T\mathbf{x} + b)$$

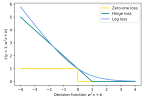
$$\min_{w\in\mathbb{R}^p, b\in\mathbb{R}} \sum_{i=1}^{n} 1_{y_i \neq \text{sign}(w^T\mathbf{x}+b)}$$

So we need to define a loss function for given w and b that tell us how well they fit the training set. Obvious Idea: Minimize number of misclassifications aka 0-1 loss but this loss is non-convex, not continuous and minimizing it is NP-hard. So we need to relax it, which basically means we want to find a convex upper bound for this loss. This is not done on the actual prediction, but on the inner product $w^T x$, which is also called the decision function. So this graph here has the inner product on the x axis, and shows what the loss would be for class 1. The 0-1 loss is zero if the decision function is positive, and one if it's negative. Because a positive decision function means a positive prediction, means correct classification in the case of $y = 1$. A negative prediction means a wrong classification, which is penalized by the 0-1 loss with a loss of 1, i.e. one mistake.

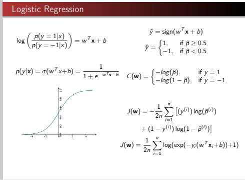
Picking A Loss

$$\hat{y} = \text{sign}(w^T\mathbf{x} + b)$$

$$\min_{w \in \mathbb{R}^p, b \in \mathbb{R}} \sum_{i=1}^{n} 1_{y_i \neq \text{sign}(w^T\mathbf{x}+b)}$$



The other losses we'll talk about are mostly the hinge loss and the log loss. You can see they are both upper bounds on the 0-1 loss but they are convex and continuous. Both of these losses care not only that you make a correct prediction, but also "how correct" your prediction is, i.e. how positive or negative your decision function is. We'll talk a bit more about the motivation of these two losses, starting with the logistic loss.



Logistic Regression

$$\log\left(\frac{p(y=1|x)}{p(y=-1|x)}\right) = w^T\mathbf{x} + b$$

$$\hat{y} = \text{sign}(w^T\mathbf{x} + b)$$

$$\hat{y} = \begin{cases} 1, & \text{if } \hat{p} \geq 0.5 \\ -1, & \text{if } \hat{p} < 0.5 \end{cases}$$

$$p(y|\mathbf{x}) = \sigma(w^Tx+b) = \frac{1}{1+e^{-w^T\mathbf{x}-b}}$$

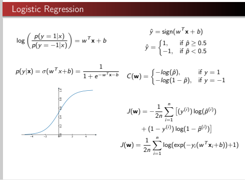
$$C(\mathbf{w}) = \begin{cases} -log(\hat{p}), & \text{if } y = 1 \\ -log(1-\hat{p}), & \text{if } y = -1 \end{cases}$$

$$J(\mathbf{w}) = -\frac{1}{2n}\sum_{i=1}^{n}\left[(y^{(i)})\log(\hat{p}^{(i)}) + (1-y^{(i)})\log(1-\hat{p}^{(i)})\right]$$

$$J(\mathbf{w}) = \frac{1}{2n}\sum_{i=1}^{n}\log(\exp(-y_i(w^T\mathbf{x}_i+b))+1)$$

Logistic regression is probably the most commonly used linear classifier, maybe the most commonly used classifier overall. The idea is to model the log-odds, which is $logp(y=1|x) - logp(y=0|x)$ as a linear function, as shown here. Rearranging the formula, you get a model of $p(y|\mathbf{x}) = \frac{1}{1+e^{-w^T\mathbf{x}-b}}$This function is called the logistic sigmoid, and is drawn to the right here. Basically it squashed the linear function w^Tx between 0 and 1, so that it can model a probability.

Given this equation for $p(y|x)$, what we want to do is maximize the probability of the training set under this model. This approach is known as maximum likelihood. Basically you want to find w and b such that they assign maximum probability to the labels observed in the training data. You can rearrange that a bit and end up with this equation here, which contains the log-loss as seen on the last slide.



Logistic Regression

$$\log\left(\frac{p(y=1|x)}{p(y=-1|x)}\right) = w^T\mathbf{x} + b$$

$$\hat{y} = \text{sign}(w^T\mathbf{x} + b)$$

$$\hat{y} = \begin{cases} 1, & \text{if } \hat{p} \geq 0.5 \\ -1, & \text{if } \hat{p} < 0.5 \end{cases}$$

$$p(y|\mathbf{x}) = \sigma(w^T x + b) = \frac{1}{1 + e^{-w^T\mathbf{x} - b}}$$

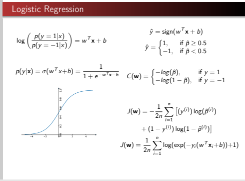
$$C(\mathbf{w}) = \begin{cases} -log(\hat{p}), & \text{if } y = 1 \\ -log(1-\hat{p}), & \text{if } y = -1 \end{cases}$$

$$J(\mathbf{w}) = -\frac{1}{2n}\sum_{i=1}^{n}\left[(y^{(i)})\log(\hat{p}^{(i)}) + (1-y^{(i)})\log(1-\hat{p}^{(i)})\right]$$

$$J(\mathbf{w}) = \frac{1}{2n}\sum_{i=1}^{n}\log(\exp(-y_i(w^T\mathbf{x}_i + b)) + 1)$$

The prediction is the class with the higher probability. In the binary case, that's the same as asking whether the probability of class 1 is bigger or smaller than .5. And as you can see from the plot of the logistic sigmoid, the probability of the class +1 is greater than .5 exactly if the decision function $w^T x$ is greater than 0. So predicting the class with maximum probability is the same as predicting which side of the hyperplane given by w we are on.

Ok so this is logistic regression. We minimize this loss and get a w which defines a hyper plane. But if you think back to last time, this is only part of what we want. This formulation tries to fit the training data, but it doesn't care about finding a simple solution.



Logistic Regression

$$\log\left(\frac{p(y=1|x)}{p(y=-1|x)}\right) = w^T\mathbf{x} + b$$

$$\hat{y} = \text{sign}(w^T\mathbf{x} + b)$$

$$\hat{y} = \begin{cases} 1, & \text{if } \hat{p} \geq 0.5 \\ -1, & \text{if } \hat{p} < 0.5 \end{cases}$$

$$p(y|\mathbf{x}) = \sigma(w^T x + b) = \frac{1}{1 + e^{-w^T\mathbf{x} - b}}$$

$$C(\mathbf{w}) = \begin{cases} -log(\hat{p}), & \text{if } y = 1 \\ -log(1-\hat{p}), & \text{if } y = -1 \end{cases}$$

$$J(\mathbf{w}) = -\frac{1}{2n}\sum_{i=1}^{n}\left[(y^{(i)})\log(\hat{p}^{(i)}) + (1-y^{(i)})\log(1-\hat{p}^{(i)})\right]$$

$$J(\mathbf{w}) = \frac{1}{2n}\sum_{i=1}^{n}\log(\exp(-y_i(w^T\mathbf{x}_i + b)) + 1)$$

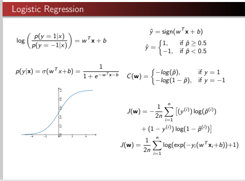
$$\ell_i = y_i \log\left(\frac{1}{1+\exp(-w^T x_i)}\right) + (1-y_i)\log\left(1 - \frac{1}{1+\exp(-w^T x_i)}\right) \tag{1}$$

$$= y_i \log\left(\frac{1}{1+\exp(-w^T x_i)}\right) + (1-y_i)\log\left(\frac{1+\exp(-w^T x_i)}{1+\exp(-w^T x_i)} - \frac{1}{1+\exp(-w^T x_i)}\right) \tag{2}$$

$$= y_i \log\left(\frac{1}{1+\exp(-w^T x_i)}\right) + (1-y_i)\log\left(\frac{\exp(-w^T x_i)}{1+\exp(-w^T x_i)}\right) \tag{3}$$

$$= y_i \log\left(\frac{1}{1+\exp(-w^T x_i)}\right) + (1-y_i)\log\left(\frac{\exp(-w^T x_i)}{1+\exp(-w^T x_i)} \times \frac{\exp(w^T x_i)}{\exp(w^T x_i)}\right) \tag{4}$$

$$= y_i \log\left(\frac{1}{1+\exp(-w^T x_i)}\right) + (1-y_i)\log\left(\frac{1}{\exp(w^T x_i)+1}\right) \tag{5}$$


Logistic Regression

$$\log\left(\frac{p(y=1|x)}{p(y=-1|x)}\right) = w^T\mathbf{x} + b$$

$$\hat{y} = \text{sign}(w^T\mathbf{x} + b)$$

$$\hat{y} = \begin{cases} 1, & \text{if } \hat{p} \geq 0.5 \\ -1, & \text{if } \hat{p} < 0.5 \end{cases}$$

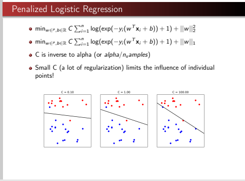
$$p(y|\mathbf{x}) = \sigma(w^T x + b) = \frac{1}{1 + e^{-w^T\mathbf{x} - b}}$$

$$C(\mathbf{w}) = \begin{cases} -log(\hat{p}), & \text{if } y = 1 \\ -log(1-\hat{p}), & \text{if } y = -1 \end{cases}$$

$$J(\mathbf{w}) = -\frac{1}{2n}\sum_{i=1}^{n}\left[(y^{(i)})\log(\hat{p}^{(i)}) + (1 - y^{(i)})\log(1 - \hat{p}^{(i)})\right]$$

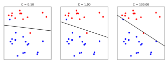
$$J(\mathbf{w}) = \frac{1}{2n}\sum_{i=1}^{n}\log(\exp(-y_i(w^T\mathbf{x}_i + b)) + 1)$$

$$\begin{aligned}
\ell_i &= \log\left(\frac{1}{1 + \exp\left(\begin{cases} -w^T x_i & y_i = 1 \\ w^T x_i & y_i = -1 \end{cases}\right)}\right) && (6) \\
&= \log\left(\frac{1}{1 + \exp\left(-y_i' w^T x_i\right)}\right) && (7) \\
&= -\log\left(1 + \exp\left(-y_i' w^T x_i\right)\right) && (8)
\end{aligned}$$

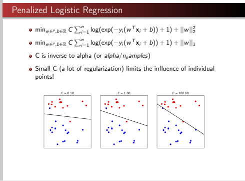


Penalized Logistic Regression

- $\min_{w \in ^p, b \in \mathbb{R}} C \sum_{i=1}^{n} \log(\exp(-y_i(w^T \mathbf{x}_i + b)) + 1) + ||w||_2^2$
- $\min_{w \in ^p, b \in \mathbb{R}} C \sum_{i=1}^{n} \log(\exp(-y_i(w^T \mathbf{x}_i + b)) + 1) + ||w||_1$
- C is inverse to alpha (or $alpha/n_samples$)
- Small C (a lot of regularization) limits the influence of individual points!

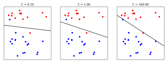


So we can do the same we did for regression: we can add regularization terms using the L1 and L2 norm. The effects are the same as for regression: both push the coefficients towards zero, but the l1 norm encourages coefficients to be exactly zero, for the same reasons we discussed last time.
You could also use a mixed penalty to get something like the elasticnet. That's not implemented in the logisticregression class in scikit-learn right now, but it's certainly a sensible thing to do.
Here I used a slightly different notation as last time, though. I'm not using alpha to multiply the regularizer, instead I'm using C to multiply the loss. That's mostly because that's how it's done in scikit-learn and it has only historic reasons. The idea is exactly the same, only now C is 1 over alpha. So large C means heavy weight to the loss, means little regularization, while small C means less weight on the loss, means strong regularization.

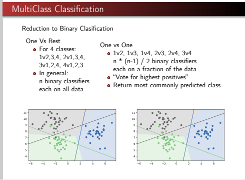


Penalized Logistic Regression

- $\min_{w \in ^p, b \in \mathbb{R}} C \sum_{i=1}^{n} \log(\exp(-y_i(w^T \mathbf{x}_i + b)) + 1) + ||w||_2^2$
- $\min_{w \in ^p, b \in \mathbb{R}} C \sum_{i=1}^{n} \log(\exp(-y_i(w^T \mathbf{x}_i + b)) + 1) + ||w||_1$
- C is inverse to alpha (or $alpha/n_samples$)
- Small C (a lot of regularization) limits the influence of individual points!



Depending on the model, there might be a factor of n$_s$*amplesintheresomewhere.Usuallywetrytomaketheobjectiveasindependentofthenumber*
learn, *butthatmightleadtosurprisesifyou′renotawareofit*.
Some side-notes on the optimization problem: here, as in regression, having more regularization makes the optimization problem easier. You might have seen this in your homework already, if you decrease C, meaning you add more regularization, your model fits more quickly.
One particular property of the logistic loss, compared to the hinge loss we'll discuss next is that each data point contributes to the loss, so each data point has an effect on the solution. That's also true for all the regression models we saw last time. So I spared you with coefficient plots, because they looks the same as for regression. All the things I said about model complexity and dependency on the number of features and samples is as true for classification as it is for regression.
There is another interesting way to thing about regularization that I found helpful, though. I'm not going to walk through the math for this, but you can reformulate the optimization problem and find that what the C parameter does is actually limit the influence of individual data points. With very large C, we said we have no



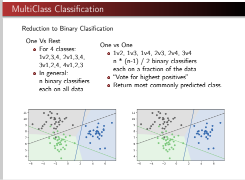
MultiClass Classification

Reduction to Binary Clasification

One Vs Rest
- For 4 classes: 1v2,3,4, 2v1,3,4, 3v1,2,4, 4v1,2,3
- In general: n binary classifiers each on all data

One vs One
- 1v2, 1v3, 1v4, 2v3, 2v4, 3v4 n * (n-1) / 2 binary classifiers each on a fraction of the data
- "Vote for highest positives"
- Return most commonly predicted class.

Next I want to look at how to go from binary classification to multi-class classification. Basically there is a simple but hacky way, and there's a slightly more complicated but theoretically sound way.
Let's start with One vs Rest. here, we learn one binary classifier for each class against the remaining classes. So let's say we have 4 classes, called 1 to 4. First we learn a binary classifier of the points in class 1 vs the points in the classes 2, 3 and 4. Then, we do the same for class 2, and so on. The way we end up building as many classifiers as we have classes. To make a prediction, we compute the decision function of all classifiers, say 4 in the example, on a new data point. The one with the highest score for the positive class, the single class, wins, and that class is predicted.



MultiClass Classification

Reduction to Binary Clasification

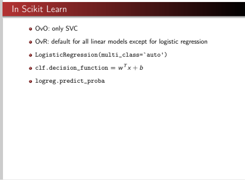
One Vs Rest
- For 4 classes: 1v2,3,4, 2v1,3,4, 3v1,2,4, 4v1,2,3
- In general: n binary classifiers each on all data

One vs One
- 1v2, 1v3, 1v4, 2v3, 2v4, 3v4 n * (n-1) / 2 binary classifiers each on a fraction of the data
- "Vote for highest positives"
- Return most commonly predicted class.

The other method of reduction is called one vs one. In one vs one, we build one binary model for each pair of classes. In the example of having four classes that is one for 1 vs 2, one for 1v3 and so on. So we end up with $n * (n-1)/2$ binary classifiers. And each is trained only on the subset of the data that belongs to these classes.

To make a prediction, we again apply all of the classifiers. For each class we count how often one of the classifiers predicted that class, and we predict the class with the most votes.

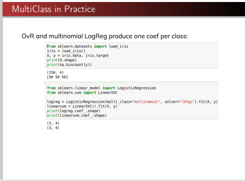
Again, this is just a heuristic and there's not really a good theoretical explanation why this should work.



In Scikit Learn

- OvO: only SVC
- OvR: default for all linear models except for logistic regression
- `LogisticRegression(multi_class=`auto')`
- `clf.decision_function` $= w^T x + b$
- `logreg.predict_proba`

All the models in scikit-learn have multi-class built-in and most of them use One versus Rest. There's only one model that uses One versus One which is SVC. The Kernel SVM uses One versus One because of the authors of the SVM like that. For historical reasons, logistic regression also uses One versus Rest by default in scikit-learn. That's probably not a good idea, but we can't really change the default easily. Usually, if you do multiclass you probably want to use multinomial logistic regression and so you set multinomial to true and then it does multinomial logistic regression. Question: Does that make it run faster? Answer: Its unlikely to make it run faster, but it makes it more theoretically sound and gives you better probabilities. The probabilities that come out of it, if you do OvR, are a complete hack in the multi-class case but in the binary case, it's the same. Logistic regression also has a predict$_p$*robmethod.Thismethodgivestheprobabilityestimates, soyougetavectoroflengthnu*
incross−validation.Thiswilltakeforeverandtheoutcomewillprobablybenotthatgreat.Don′t



MultiClass in Practice

OvR and multinomial LogReg produce one coef per class:

```
from sklearn.datasets import load_iris
iris = load_iris()
X, y = iris.data, iris.target
print(X.shape)
print(np.bincount(y))
```

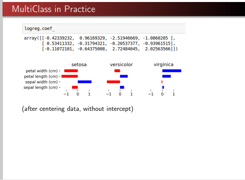
```
(150, 4)
[50 50 50]
```

```
from sklearn.linear_model import LogisticRegression
from sklearn.svm import LinearSVC

logreg = LogisticRegression(multi_class="multinomial", solver="lbfgs").fit(X, y)
linearsvm = LinearSVC().fit(X, y)
print(logreg.coef_.shape)
print(linearsvm.coef_.shape)
```

```
(3, 4)
(3, 4)
```

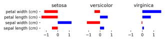
In practice, I'm using logistic regression on the iris dataset. We have 50 data points, 4 features, 3 classes, each class has 50 samples, and we're trying to classify different kinds of irises. Here, we're looking at logistic regression and linear SVM. After building the model, the coefficient *w* is stored in *coef* on this score. Everything in scikit-learn that's estimated from the data ends with an underscore. If it doesn't, it wasn't learned from the data. So *coef* are the coefficient that is learned by the model. They're for logistic regression and linear Support Vector Machine, they're the same shape, three classes times four features but they have different semantics.



MultiClass in Practice

```
logreg.coef_
```

```
array([[-0.42339232,  0.96169329, -2.51946669, -1.0860205 ],
       [ 0.53411332, -0.31794321, -0.20537377, -0.93961515],
       [-0.11072101, -0.64375008,  2.72484045,  2.02563566]])
```



(after centering data, without intercept)

In practice, I'm using logistic regression on the iris dataset. We have 50 data points, 4 features, 3 classes, each class has 50 samples, and we're trying to classify different kinds of irises. Here, we're looking at logistic regression and linear SVM. After building the model, the coefficient *w* is stored in *coef* on this score. Everything in scikit-learn that's estimated from the data ends with an underscore. If it doesn't, it wasn't learned from the data. So *coef* are the coefficient that is learned by the model. They're for logistic regression and linear Support Vector Machine, they're the same shape, three classes times four features but they have different semantics.