Linear models with early stopping regularization

Toby Dylan Hocking

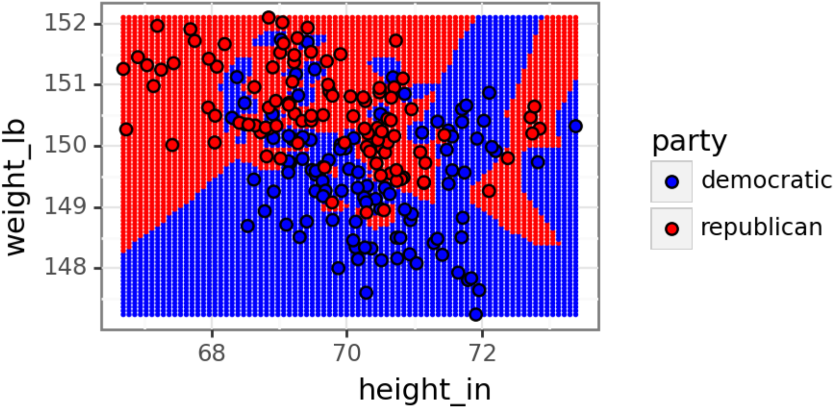
Supervised machine learning

- Goal is to learn a function $f(\mathbf{x}) = y$ where $\mathbf{x} \in \mathbb{R}^p$ is an input/feature vector and y is an output/label.
- $\mathbf{x} =$image of digit/clothing, $y \in \{0, \ldots, 9\}$ (ten classes).
- $\mathbf{x} =$vector of word counts in email, $y \in \{1, 0\}$ (spam or not).
- This week we will study linear models in depth, and we will focus on binary classification, with labels represented by $y \in \{-1, 1\}$.

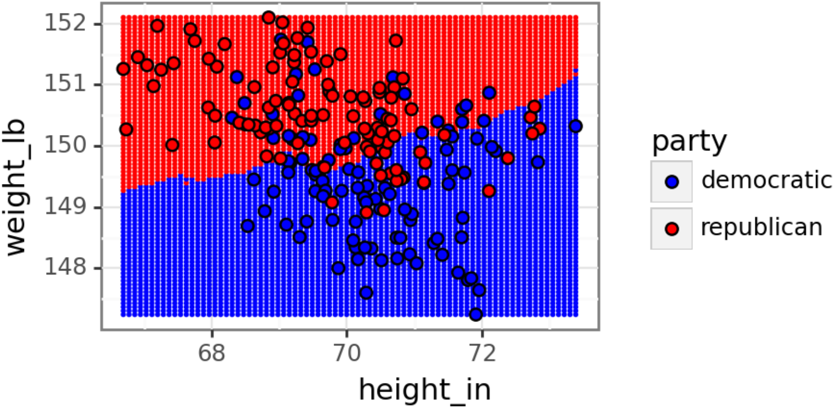
Mixture data table

```
##             party  height_in   weight_lb
## 0    democratic  71.741421  149.565034
## 1    democratic  69.582283  149.275446
## 2    democratic  69.983547  149.961470
## 3    democratic  69.908764  150.021178
## 4    democratic  69.195491  150.111237
## ..          ...        ...         ...
## 195  republican  69.472078  151.537588
## 196  republican  71.140501  149.409036
## 197  republican  70.517269  150.236183
## 198  republican  69.223459  151.486248
## 199  republican  69.019082  149.795387
##
## [200 rows x 3 columns]
```

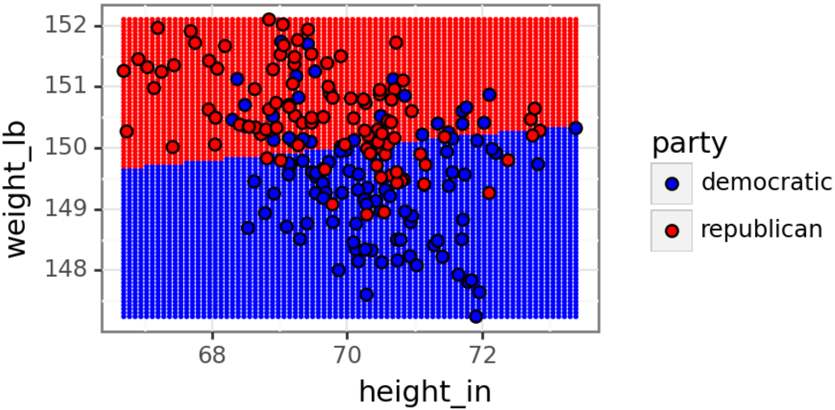
Visualize predictions of 1-nearest neighbor algorithm



Visualize predictions of 100-nearest neighbor algorithm



Linear model predictions



What is a linear model?

- Compute a real-valued score that measures how likely the feature vector is to be in the positive class. (predict positive class if score greater than zero)
- Parameters required in order to compute score must be learned from the train data: weights $\mathbf{w} \in \mathbb{R}^p$ and intercept $\beta \in \mathbb{R}$ (sometimes called regression coefficients in the statistics literature).
- The score is computed by multiplying each feature with a weight, then adding them to the intercept: $f(\mathbf{x}) = \beta + \mathbf{w}^T\mathbf{x}$. Pseudo-code:

```
def predict_score(feature_vec, weight_vec, intercept):
  score = intercept
  for weight, feature in zip(weight_vec, feature_vec):
    score += weight * feature
  return score
```

Using learned coefficients to compute predictions

```
from sklearn.linear_model import LogisticRegression
lr = LogisticRegression().fit(mix_features, mix_labels)
lr.intercept_
```

```
## array([-189.61454832])
```

```
lr.coef_
```

```
## array([[-0.14097561,  1.32973242]])
```

```
g_df = pd.DataFrame(grid_mat, columns=["height_in","weight_
lr.decision_function(g_df)
```

```
## array([-3.22155406, -3.23349517, -3.24543628, ...,
##         2.31570689,  2.30376579,  2.29182468])
```

```
(grid_mat * lr.coef_).sum(axis=1) + lr.intercept_
```

```
## array([-3.22155406, -3.23349517, -3.24543628, ...,
##         2.31570689,  2.30376579,  2.29182468])
```

Using learned coefficients to compute predictions 2

Vector of predictions for matrix of data is **Xw**,

```
lr.decision_function(g_df)
```

```
## array([-3.22155406, -3.23349517, -3.24543628, ...,
##         2.31570689,  2.30376579,  2.29182468])
```

```
np.matmul(grid_mat, lr.coef_.T) + lr.intercept_
```

```
## array([[-3.22155406],
##        [-3.23349517],
##        [-3.24543628],
##        ...,
##        [ 2.31570689],
##        [ 2.30376579],
##        [ 2.29182468]])
```

```
np.matmul(lr.coef_, grid_mat.T) + lr.intercept_
```

```
## array([[-3.22155406, -3.23349517, -3.24543628, ...,
##          2.31570689,  2.30376579,  2.29182468]])
```

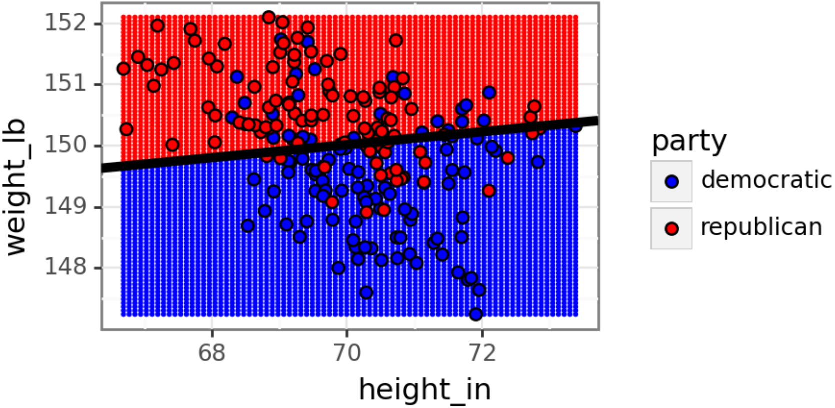
Equations for plotting the decision boundary

- The linear model decision boundary is defined by $f(\mathbf{x}) = \beta + \mathbf{w}^T \mathbf{x} = 0$.
- On one side of this line we predict positive, and on the other side we predict negative.
- For the mixture data set ($p = 2$) this implies $\beta + w_1 x_1 + w_2 x_2 = 0$, and $x_2 = -(\beta + w_1 x_1)/w_2 = -\beta/w_2 - (w_1/w_2)x_1$.

```
line_param_df = pd.DataFrame({
    "slope":-lr.coef_[:,0]/lr.coef_[:,1],
    "intercept":-lr.intercept_/lr.coef_[:,1]})
line_param_df
```

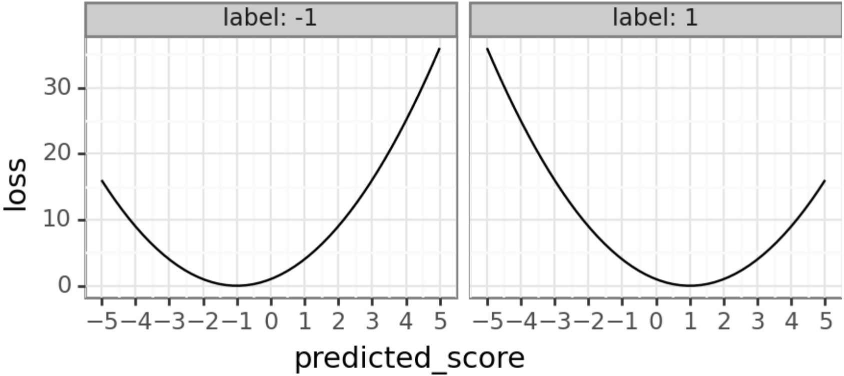
```
##        slope   intercept
## 0  0.106018  142.596018
```

Plotting the decision boundary with data and predictions



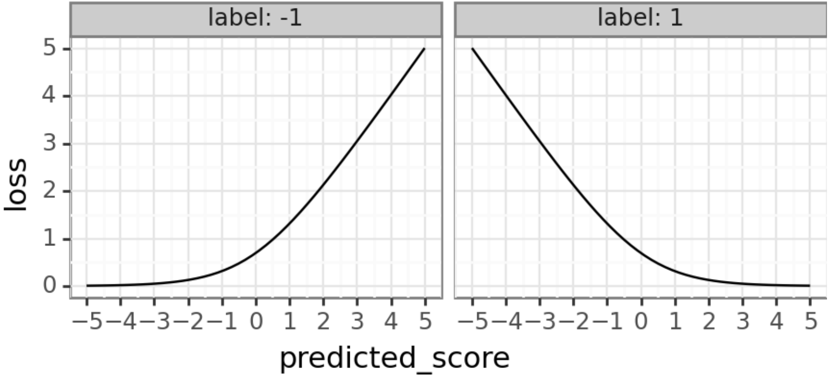
How are the coefficients learned?

- Typically we use some version of gradient descent.
- This algorithm requires definition of a differentiable loss function to minimize on the train set.
- For regression problems ($y \in \mathbb{R}$) we use the square loss, $\ell(\beta + \mathbf{w}^T\mathbf{x}, y) = (\beta + \mathbf{w}^T\mathbf{x} - y)^2$.



Logistic regression and loss function

- For binary classification problems ($y \in \{-1, 1\}$) we use the logistic loss, resulting in a model called logistic regression.
- Confusing name because it is actually for binary classification problems, not regression problems!
- Logistic loss is $\ell(\beta + \mathbf{w}^T \mathbf{x}, y) = \log[1 + \exp(-y(\beta + \mathbf{w}^T \mathbf{x}))]$.



Optimization Problem and Gradients

For any set of coefficients $\mathbf{w}, \beta$ we define the loss on the set of n train examples as:

$$\mathcal{L}(\mathbf{w}, \beta) = \sum_{i=1}^{n} \ell(\beta + \mathbf{w}^T \mathbf{x}_i, y_i)$$

- We want to compute coefficients $\mathbf{w}, \beta$ which minimize the train loss $\mathcal{L}(\mathbf{w}, \beta)$ — this is an optimization problem.
- Actually we would like to minimize the test loss, but we do not have access to the test set at train time, so we must assume that the train set is similar enough to be used as a surrogate.
- To minimize $\mathcal{L}$ we need to compute the gradients of the loss with respect to the optimization variables,
- $\nabla_\beta \mathcal{L} : \mathbb{R}^{p+1} \rightarrow \mathbb{R}$ tells us how β is changing at the input coefficients, $\nabla_\beta \mathcal{L}(\mathbf{w}, \beta) = \partial \mathcal{L}(\mathbf{w}, \beta)/\partial \beta$.
- $\nabla_{\mathbf{w}} \mathcal{L} : \mathbb{R}^{p+1} \rightarrow \mathbb{R}^p$ tells us how $\mathbf{w}$ is changing at the input coefficients, $\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}, \beta) = [\partial \mathcal{L}(\mathbf{w}, \beta)/\partial w_1 \cdots \partial \mathcal{L}(\mathbf{w}, \beta)/\partial w_p]$.

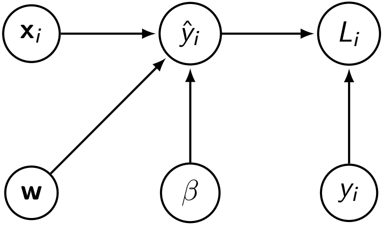
Brief review of some facts from calculus

- Why do we need to do calculus? Because we need to compute partial derivatives / gradients to implement our learning algorithm!
- $d/dx\ y = dy/dx$ is the derivative of y with respect to x.
- Constant: $d/dx\ a = 0$.
- Polynomial: $d/dx\ x^a = ax^{a-1}$. (including linear, $a = 1$)
- Logarithm: $d/dx \log x = 1/x$.
- Exponential: $d/dx \exp x = \exp x$.
- Linearity: $d/dx[g(x) + h(x)] = d/dx\ g(x) + d/dx\ h(x)$.
- Multiplicative constants: $d/dx[cf(x)] = c[d/dx\ f(x)]$.
- Chain rule: if $z = g(x)$ and $y = h(g(x)) = h(z)$ then we have

$$\frac{dy}{dx} = \frac{dy}{dz}\frac{dz}{dx} = h'(z)g'(x) = h'(g(x))g'(x).$$

Computation graph and chain rule for gradient

- Let $\hat{y}_i = \beta + \mathbf{w}^T \mathbf{x}_i$ be the predicted score for example i.
- Then loss for example i is $L_i = \ell(\beta + \mathbf{w}^T \mathbf{x}_i, y_i) = \ell(\hat{y}_i, y_i)$.



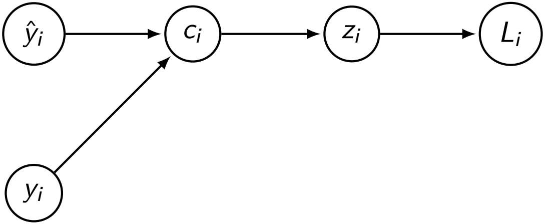
$$\nabla \mathcal{L}(\mathbf{w}, \beta) = \sum_{i=1}^{n} \nabla \ell(\beta + \mathbf{w}^T \mathbf{x}_i, y_i) = \sum_{i=1}^{n} \nabla L_i.$$

- This can simplify the gradient computation using the chain rule:

$$\nabla_\beta L_i = \frac{\partial L_i}{\partial \beta} = \frac{\partial L_i}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial \beta} = \frac{\partial L_i}{\partial \hat{y}_i}, \ \nabla_{\mathbf{w}} L_i = \frac{\partial L_i}{\partial \hat{y}_i} \nabla_{\mathbf{w}} \hat{y}_i = \frac{\partial L_i}{\partial \hat{y}_i} \mathbf{x}_i.$$

Gradient computation and chain rule for loss

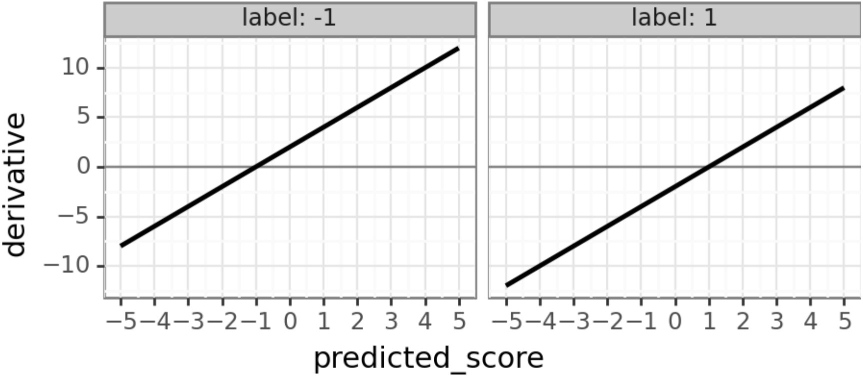
- Need to compute $\partial L_i / \partial \hat{y}_i$.
- Logistic loss is $L_i = \ell(\hat{y}_i, y_i) = \log[1 + \exp(-y_i \hat{y}_i)]$.
- Let $z_i = 1 + \exp(-y_i \hat{y}_i)$, then $L_i = \log[z_i]$.
- Let $c_i = -y_i \hat{y}_i$, then $z_i = 1 + \exp(c_i)$.



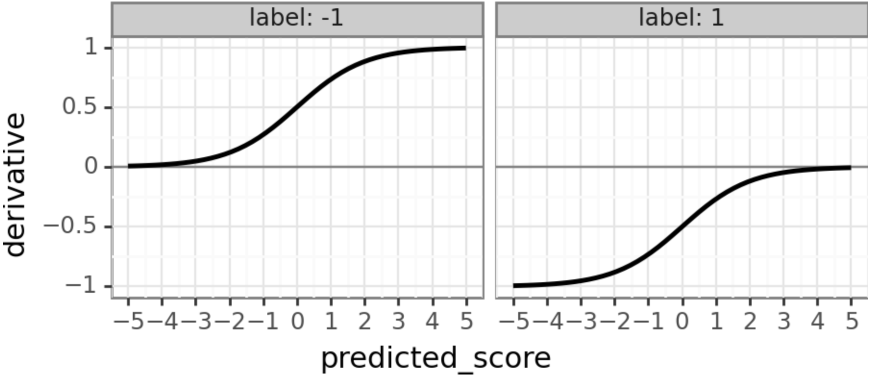
We can also compute this gradient using the chain rule:

$$\frac{\partial L_i}{\partial \hat{y}_i} = \frac{\partial L_i}{\partial z_i} \frac{\partial z_i}{\partial c_i} \frac{\partial c_i}{\partial \hat{y}_i} = (\frac{1}{z_i})\big(\exp[c_i]\big)(-y_i) = \frac{-y_i \exp(c_i)}{1 + \exp(c_i)} = \frac{-y_i}{1 + \exp(y_i \hat{y}_i)}.$$

Visualization of square loss gradient/derivative



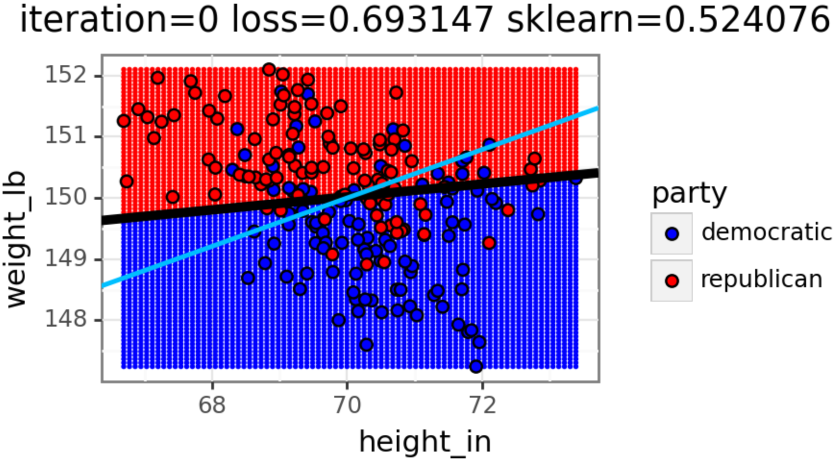
Visualization of logistic loss gradient/derivative



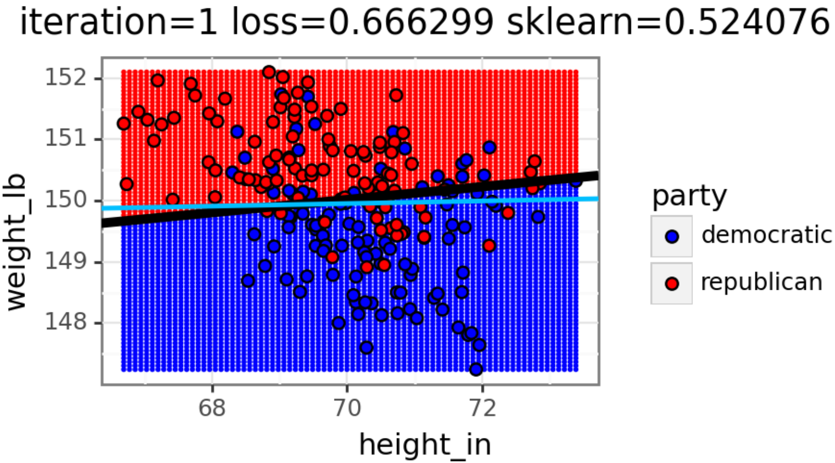
Gradient descent algorithm

- Start with random initial coefficients $\mathbf{w}^0, \beta^0$ near zero.
- Compute the gradient, which is the direction of steepest ascent.
- For each iteration $k \in \{0, 1, \dots\}$, take a step of size $\alpha > 0$ in the opposite direction: (because we want to minimize the loss)
- Step size α is sometimes called learning rate, and can be either constant or variable α^k with each iteration k.
- $\mathbf{w}^{k+1} = \mathbf{w}^k - \alpha \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}^k, \beta^k)$
- $\beta^{k+1} = \beta^k - \alpha \nabla_{\beta} \mathcal{L}(\mathbf{w}^k, \beta^k)$
- Optimizing a linear model with a convex loss function is a convex optimization problem, so a solution $\mathbf{w}^*, \beta^*$ which achieves the global minimum of the train loss $\mathcal{L}$ can be computed using gradient descent (if step size small enough).
- But do we want to minimize the train loss? (no, we want to minimize the test loss)

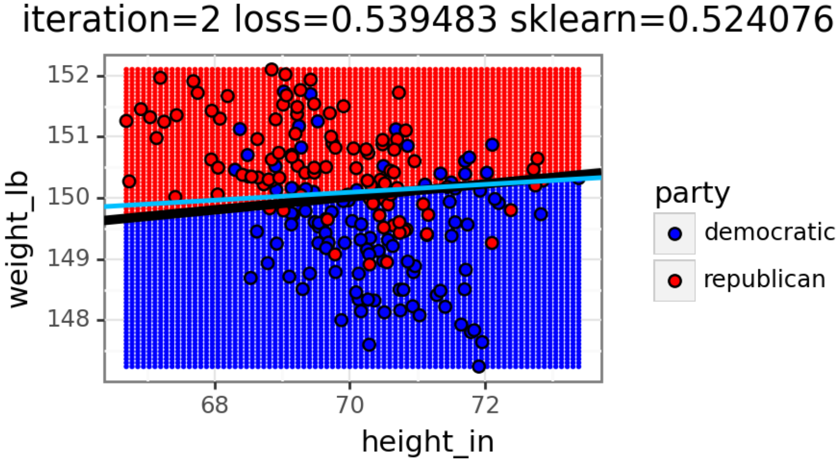
Iterations of gradient descent



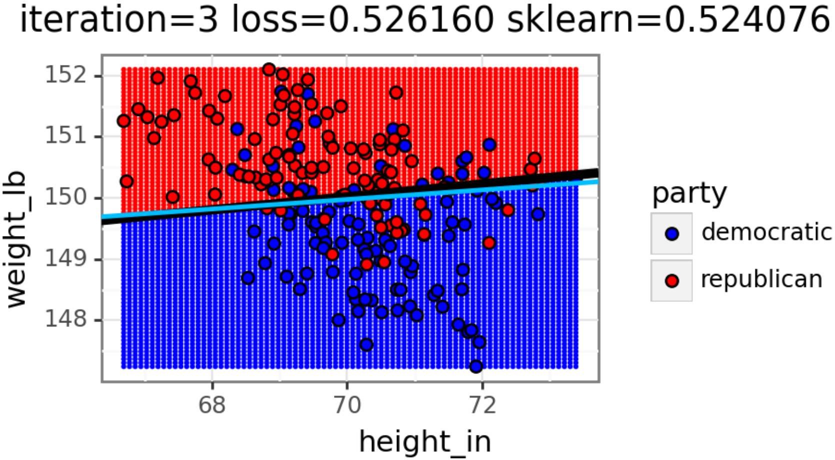
Iterations of gradient descent



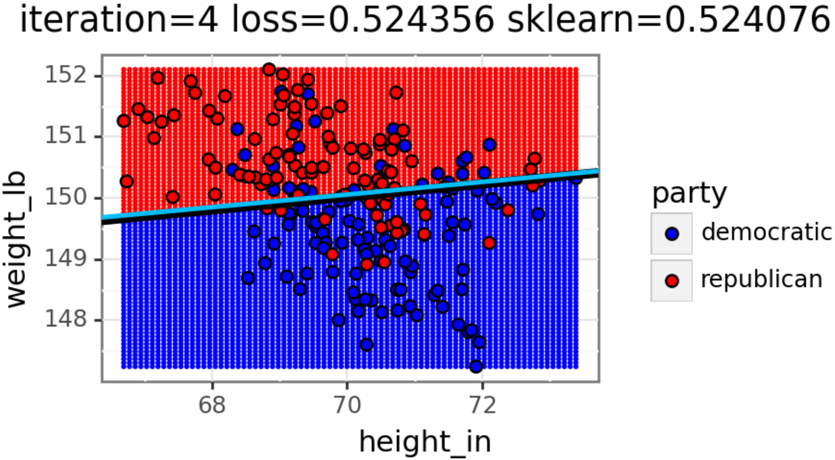
Iterations of gradient descent



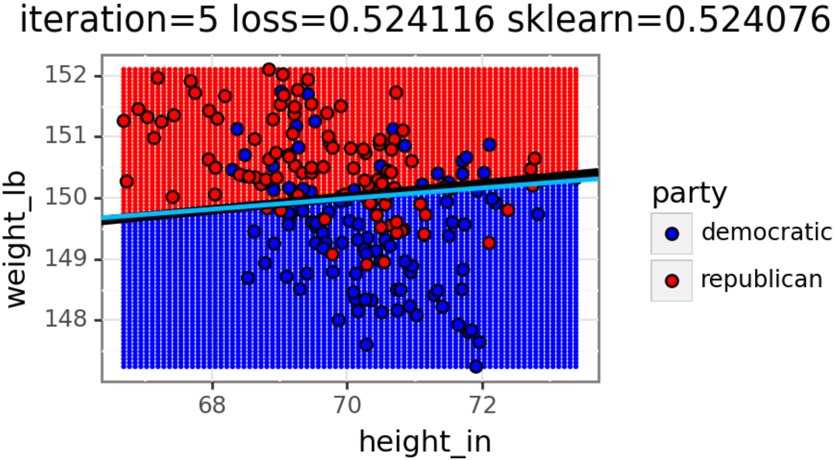
Iterations of gradient descent



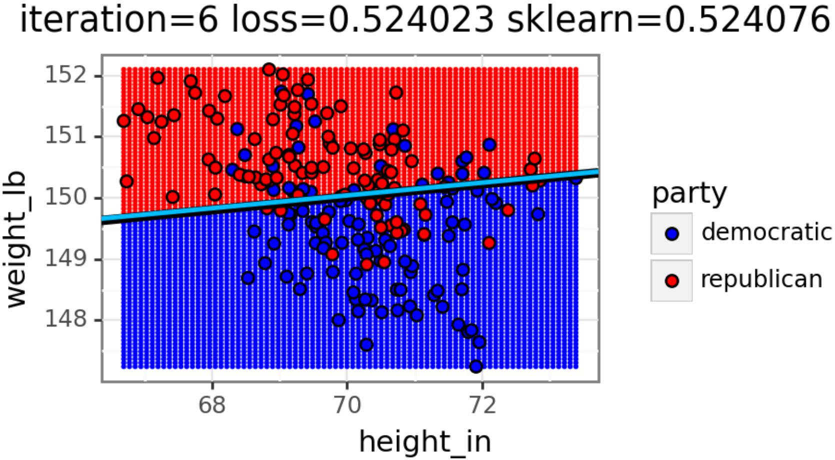
Iterations of gradient descent



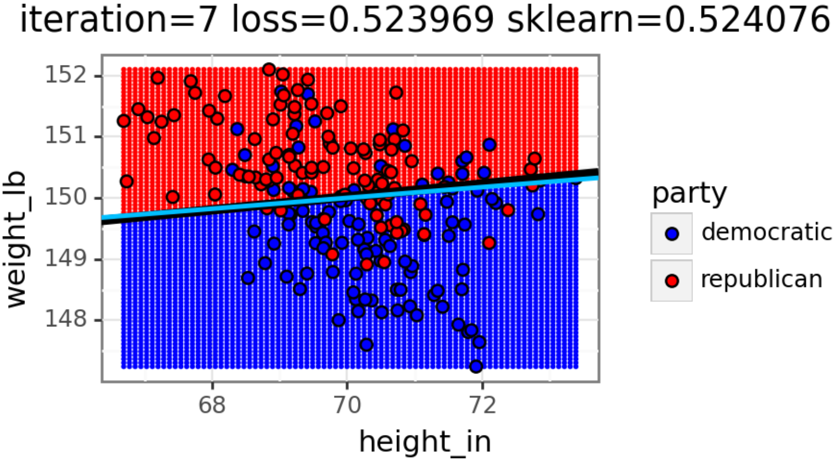
Iterations of gradient descent



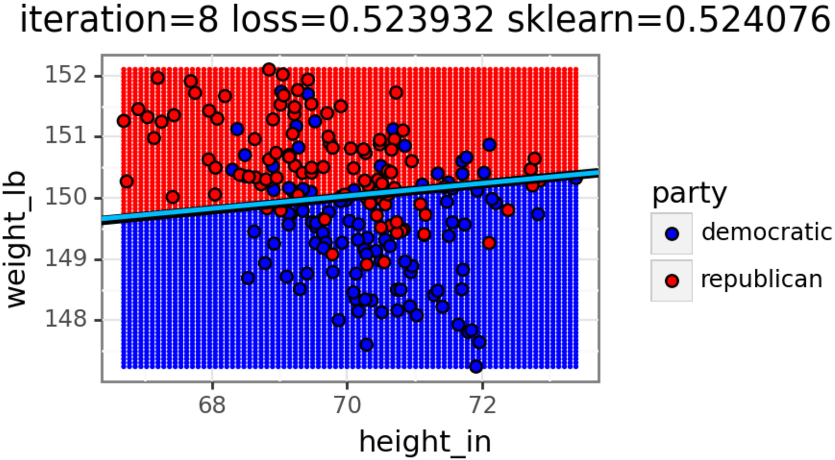
Iterations of gradient descent



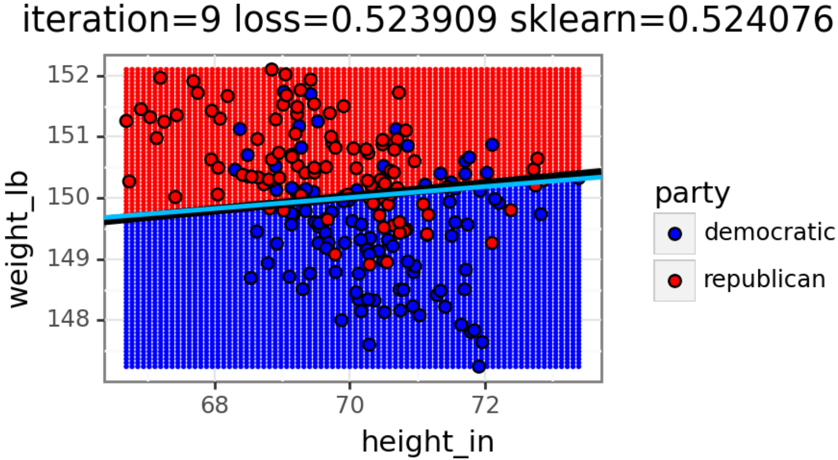
Iterations of gradient descent



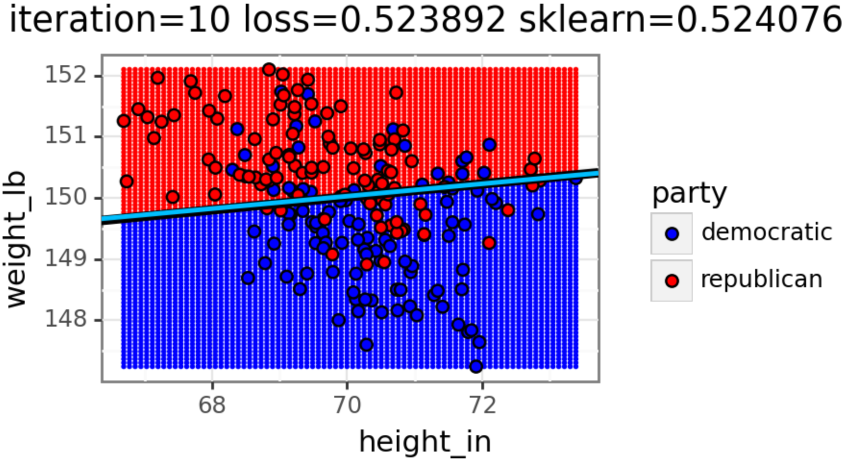
Iterations of gradient descent



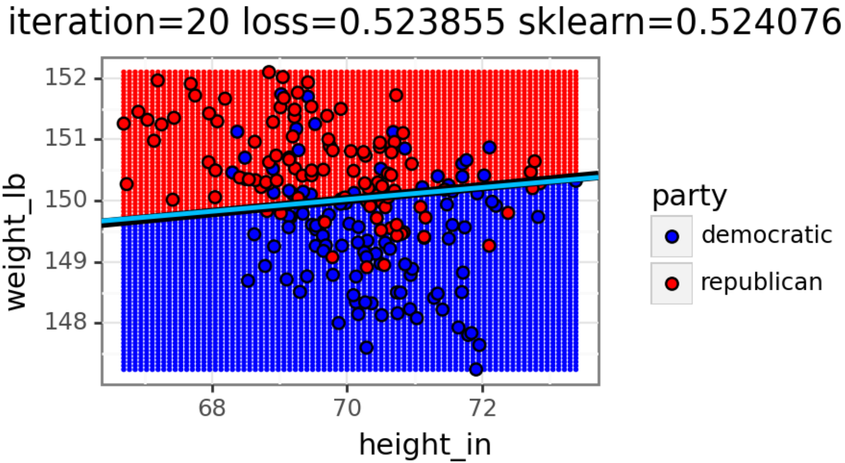
Iterations of gradient descent



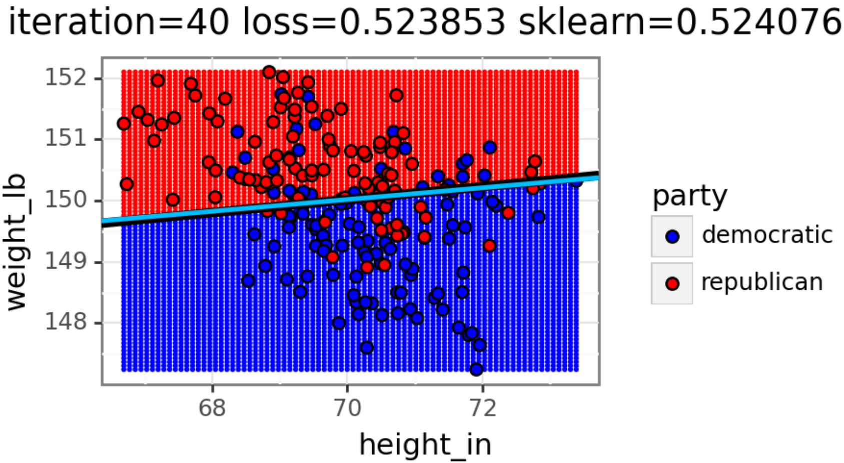
Iterations of gradient descent



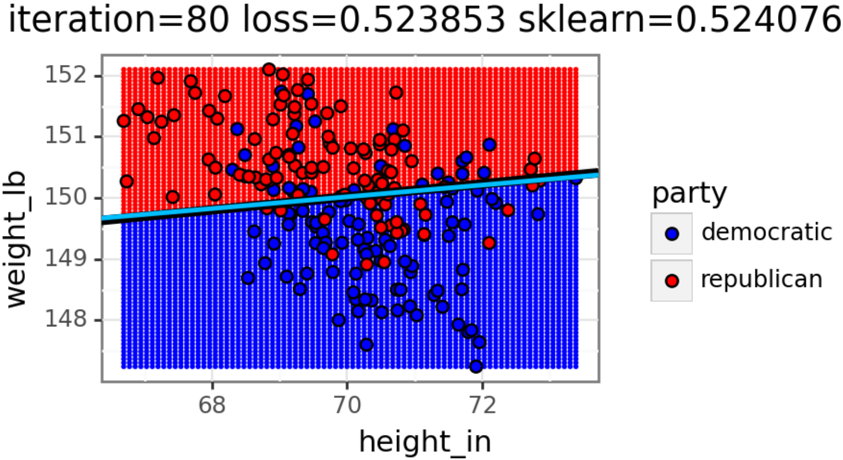
Iterations of gradient descent



Iterations of gradient descent

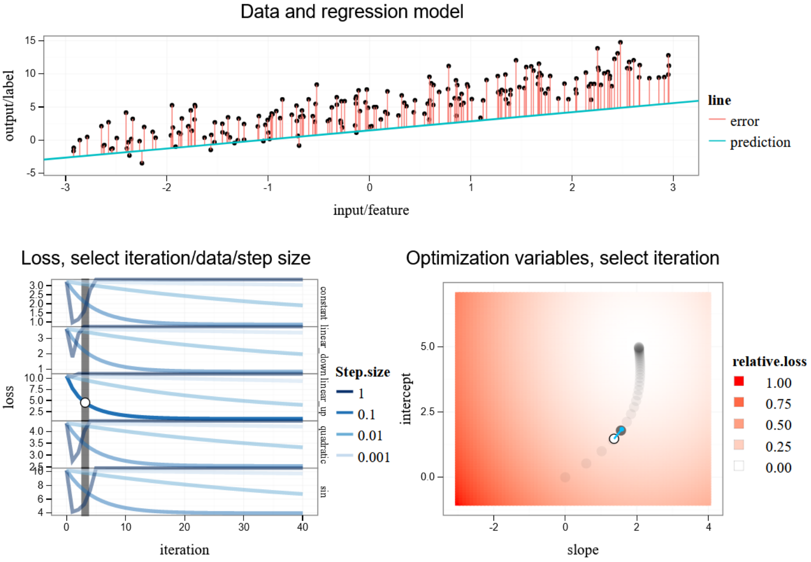


Iterations of gradient descent



Interactive visualization of gradient descent for regression

http://ml.nau.edu/viz/2022-02-02-gradient-descent-regression/



Early stopping regularization

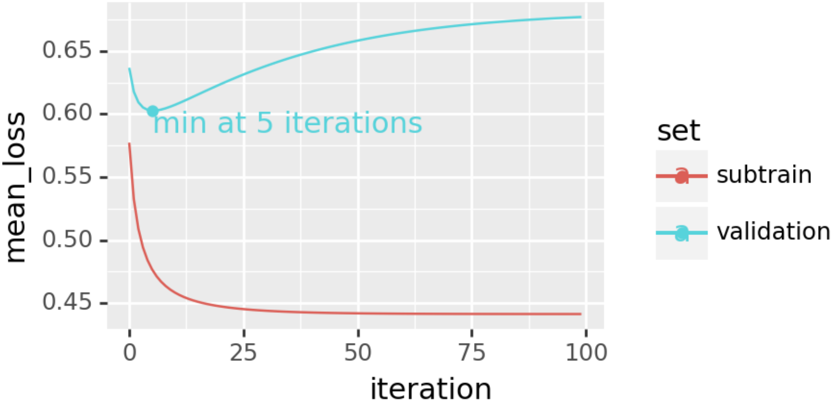
- Goal is to have good predictions on held-out data (test set).
- Computing parameters which achieve global minimum of train loss may not be desirable (overfitting noise features in the train set, not relevant in the test data).
- So then how many iterations of gradient descent should we do?
- We need to split the train set into subtrain and validation sets in order to avoid overfitting.
- Subtrain set used to compute gradients and learn model **parameters** $(\beta, \mathbf{w})$.
- Validation set used to learn **hyper-parameter**, number of iterations k. The best one minimizes error/loss (maximizes accuracy) with respect to validation set.
- Plot subtrain and validation loss as a function of iteration k: subtrain always decreasing, and validation U shape.

Add 20 noise features using np.random.randn

```
##            party  height_in  ...     noise18     noise19
## 0    democratic  71.741421  ...    7.942844   22.100326
## 1    democratic  69.582283  ...  219.020236  -34.046501
## 2    democratic  69.983547  ...  -13.489215  -57.774854
## 3    democratic  69.908764  ...   16.051798  -40.403156
## 4    democratic  69.195491  ...   44.269784  -60.328943
## ..          ...        ...  ...         ...         ...
## 195  republican  69.472078  ...   24.847164   57.340679
## 196  republican  71.140501  ...   23.306911   74.468363
## 197  republican  70.517269  ...  120.647915 -156.217900
## 198  republican  69.223459  ... -113.229565 -195.971788
## 199  republican  69.019082  ... -119.217509   73.226482
##
## [200 rows x 23 columns]
```

Demonstration of early stopping regularization

- Mixture data: 2 signal features, 20 noise features, gradient descent with constant step size $\alpha = 1$.



Commentary on regularization hyper-parameters

- The linear model learning algorithm is gradient descent, which computes the weight/intercept **parameters** (can be used to compute predictions).
- In every machine learning algorithm, we also need to learn one or more **hyper-parameters** which control regularization (and must be fixed prior to running the learning algorithm).
- In the linear model, we can use early stopping regularization, meaning the number of iterations is the **hyper-parameter** which needs to be learned using subtrain/validation splits.
- In the mixture data the mean loss with respect to validation set is minimized after only 5 iterations of gradient descent.
- Before 5 iterations we say the model **underfits**, since it is neither accurate on the subtrain nor on the validation data.
- After 5 iterations we say the model **overfits**, since it is memorizing the noise in the subtrain set which is not relevant for accurate prediction in the validation set.
- The optimal hyper-parameter values are specific to each data set (you always need to do a subtrain/validation split).

Implementation details / scaling

Before learning make sure to

- scale the input/feature matrix columns to each have mean=0 and sd=1.
- remove columns which are constant (sd=0).
- to learn intercept, can add a column of ones, and add an extra entry to weight vector (simpler to code).

If you do scaling as a part of your learning algorithm, then make sure to report the coefficients on the original scale. (easier for user)

- Let m_j, s_j be the mean/sd of each column j.
- Let $\tilde{\mathbf{w}}, \tilde{\beta}$ be the parameters learned using the scaled inputs.
- If $S^{-1} = \text{Diag}(1/s_1, \dots, 1/s_p)$ then predictions are $f(\tilde{\mathbf{x}}) = \tilde{\mathbf{w}}^T \tilde{\mathbf{x}} + \tilde{\beta} = \tilde{\mathbf{w}}^T S^{-1}(\mathbf{x} - \mathbf{m}) + \tilde{\beta}$.
- Then the parameters for unscaled inputs are $w_j = \tilde{w}_j / s_j$ for all features j with positive variance, and $\beta = \tilde{\beta} - \sum_{j=1}^p \tilde{w}_j m_j / s_j$.

Where does the loss function come from?

- We assume that the labels are generated from a probability distribution, then compute model parameters which maximize the log likelihood.
- For regression problems maximizing the normal log likelihood results in minimizing the square loss: $y_i \sim N(\beta + \mathbf{w}^T \mathbf{x}_i, \sigma^2)$.

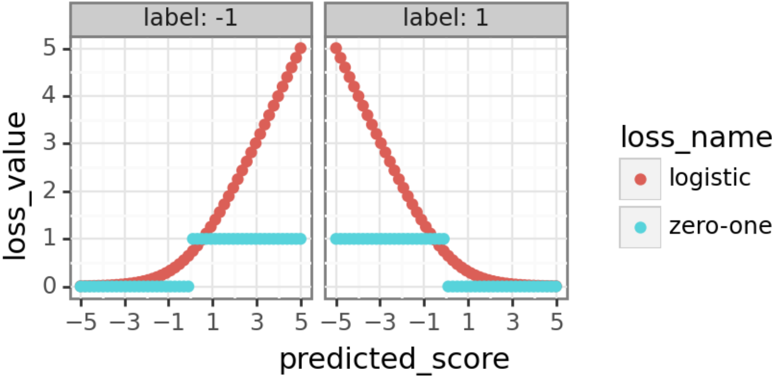
$$\max_{\beta, \mathbf{w}} \sum_{i=1}^{n} 0.5 \log(2\pi\sigma^2) - (\beta + \mathbf{w}^T \mathbf{x}_i - y_i)^2/(2\sigma^2).$$

- For binary classification problems maximizing the Bernoulli log likelihood results in minimizing the logistic loss: $y_i \sim \text{Bernoulli}[s(\beta + \mathbf{w}^T \mathbf{x}_i)]$, with the logistic link function: $s(\hat{y}_i) = 1/(1 + \exp(-\hat{y}_i))$ is probability of class 1.

$$\max_{\beta, \mathbf{w}} \sum_{i=1}^{n} \log[s(\beta + \mathbf{w}^T \mathbf{x}_i)] I[y_i = 1] + \log[1 - s(\beta + \mathbf{w}^T \mathbf{x}_i)] I[y_i = -1].$$

Why don't we just maximize the accuracy?

- Accuracy is number/percent of correctly predicted labels.
- Error is number/percent of incorrectly predicted labels (also called zero-one loss).
- Zero-one loss can not be used for learning (gradient is zero), but logistic loss is a decent approximation (and has non-zero gradient).



Possible exam questions

- How would the gradient computation change if we have a regression problem, so are using the square loss instead of the logistic loss?
- What regularization hyper-parameter(s) can be learned in nearest neighbors?
- Could the step size / learning rate be considered a regularization parameter? Do large or small values overfit/underfit?