Linear and Logistic Regression

CS 780/880 Natural Language Processing Lecture 10

Samuel Carton, University of New Hampshire

Last lecture

Key idea: Hidden Markov Models

Concepts:

- Bayes Networks
- Generative story
- HMMs
  - Inference
    - Likelihood: **Forward algorithm**
    - Decoding: **Viterbi algorithm**
  - Learning
    - Labeled: Counting
    - Unlabeled: **Forward-backward algorithm**
  - Generation

Concepts:

- POS tagging
- Dynamic programming
- Expectation-maximization

Linear regression

Linear regression

Basic idea: given some points in N-dimensional space, find a "line of best fit" that is as close as possible to those points.

When the points are text:

- N = vocabulary size
- Examples:
  - Grading essays 0-100
  - Scoring text complexity

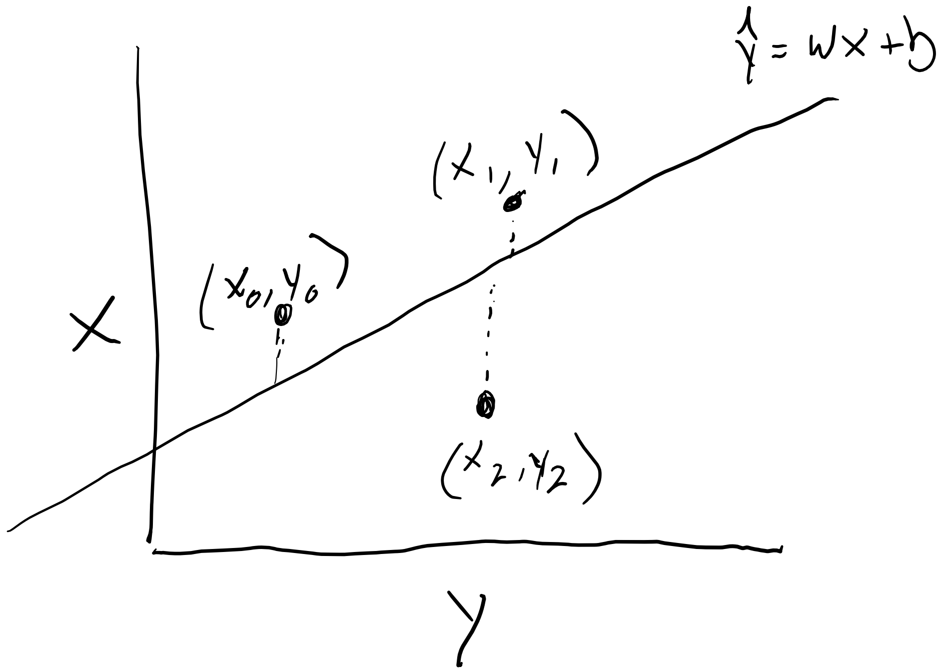
Linear regression

Mathematically, what we're trying to do is figure out some function:

$\hat{y} = Wx + b$

…where W and b are values such that $\hat{y}$ tends to be close to y for any given x.

Very common in ML to refer to predicted output as $\hat{y}$ and true output as y.



Loss function

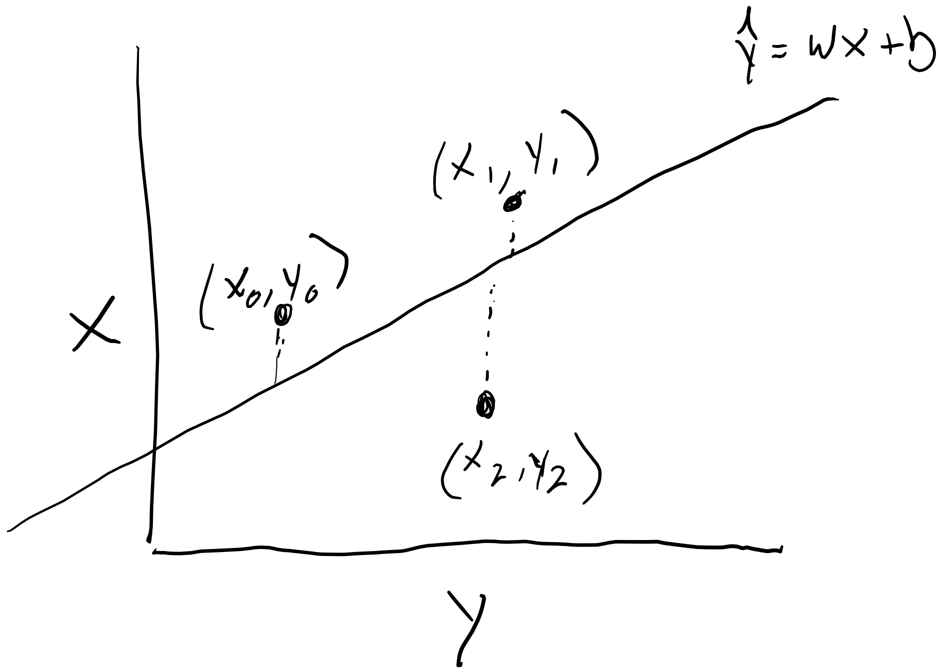
We generally articulate this goal with a **loss function** that describes the value we're trying to minimize with our choice of W and b.

AKA "Objective function"

It's very typical to minimize **squared loss** between expected and true output: $\sum_i (\hat{y}_i - y_i)^2$

That would give us a loss function of:

$$L(W, b) = \sum_i (\hat{y}_i - y_i)^2$$
$$= (\hat{y}_0 - y_0)^2 + (\hat{y}_1 - y_1)^2 + (\hat{y}_2 - y_2)^2$$
$$= (Wx_0 + b - y_0)^2 + (Wx_1 + b - y_1)^2 + (Wx_2 + b - y_2)^2$$



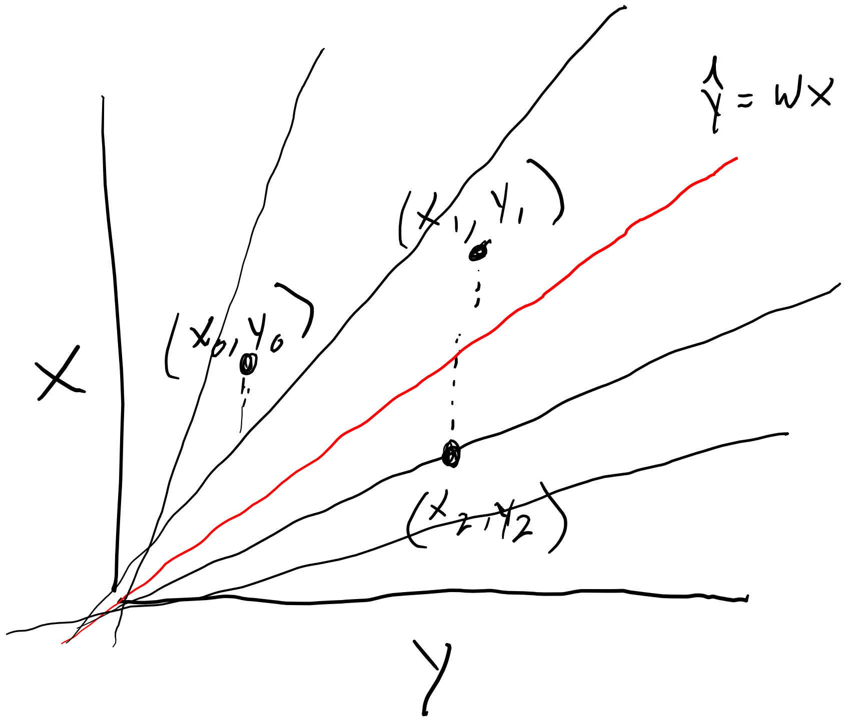
Simple example

To show how we can solve this, I'll use a simple example with **no intercept** (b)

So the loss function is:

$$L(W,b) = \sum_i (\hat{y}_i - y_i)^2$$
$$= (\hat{y}_0 - y_0)^2 + (\hat{y}_1 - y_1)^2 + (\hat{y}_2 - y_2)^2$$
$$= (Wx_0 - y_0)^2 + (Wx_1 - y_1)^2 + (Wx_2 - y_2)^2$$
$$= (W^2{x_0}^2 - 2Wx_0y_0 + {y_0}^2) + (W^2{x_1}^2 - 2Wx_1y_1 + {y_1}^2) + (W^2{x_2}^2 - 2Wx_2y_2 + {y_2}^2)$$

$$= W^2({x_0}^2 + {x_1}^2 + {x_2}^2) - 2W(x_0y_0 + x_1y_1 + x_2y_2) + ({y_0}^2 + {y_1}^2 + {y_2}^2)$$

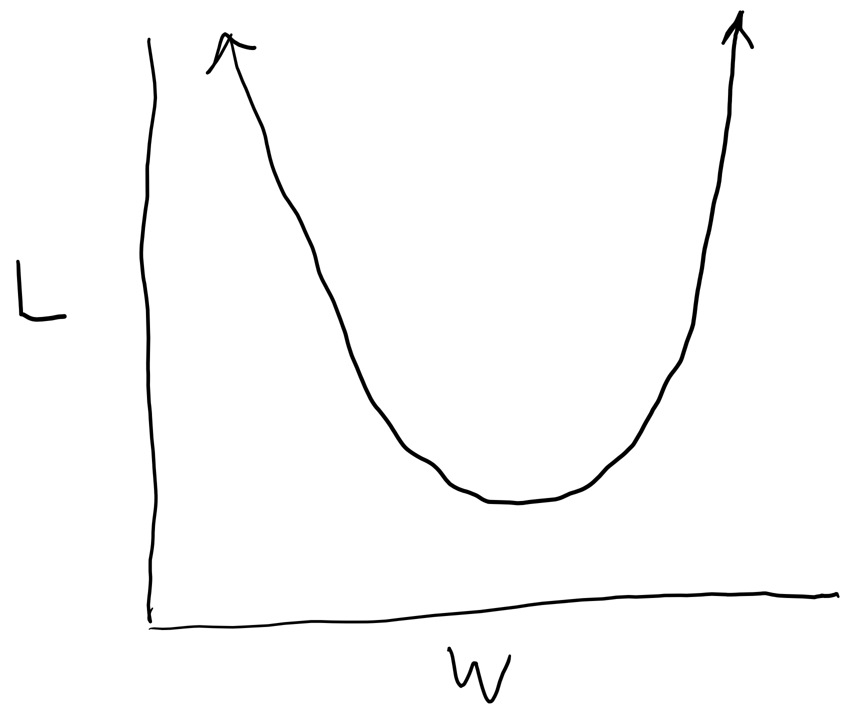


Simple example

If the loss function for W is this:

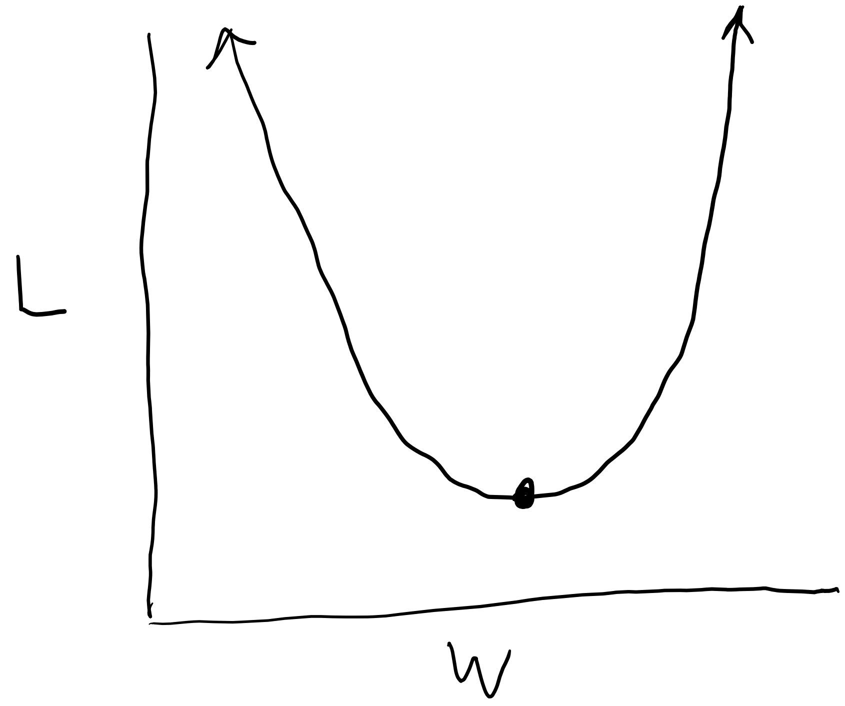
$$L(W) = W^2({x_0}^2 + {x_1}^2 + {x_2}^2)$$
$$-2W(x_0y_0 + x_1y_1 + x_2y_2)$$
$$+({y_0}^2 + {y_1}^2 + {y_2}^2)$$

Then the graph of L as a function of W looks like this:



Simple example

It's pretty obvious what value of *W* will minimize the loss here.



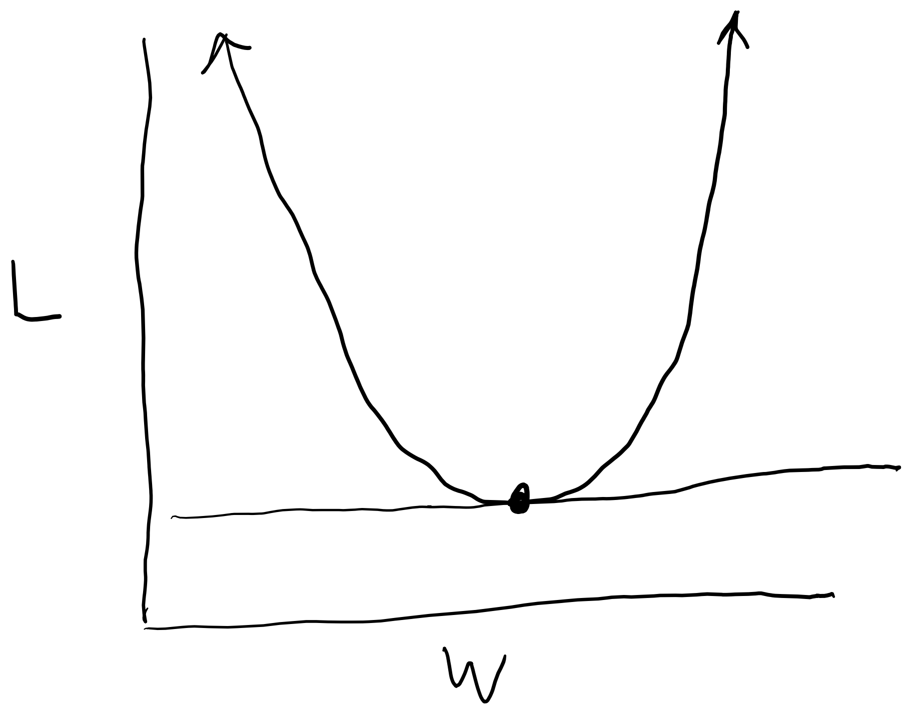
Simple example

What may be less obvious is that this also happens to be the point where the derivative of L with respect to W is 0.

$$\frac{dL}{dW} = 0$$

In some sense it is the bottom of a pit.

Gradient descent is the process of gradually following the slope of the function down to these pit-bottoms

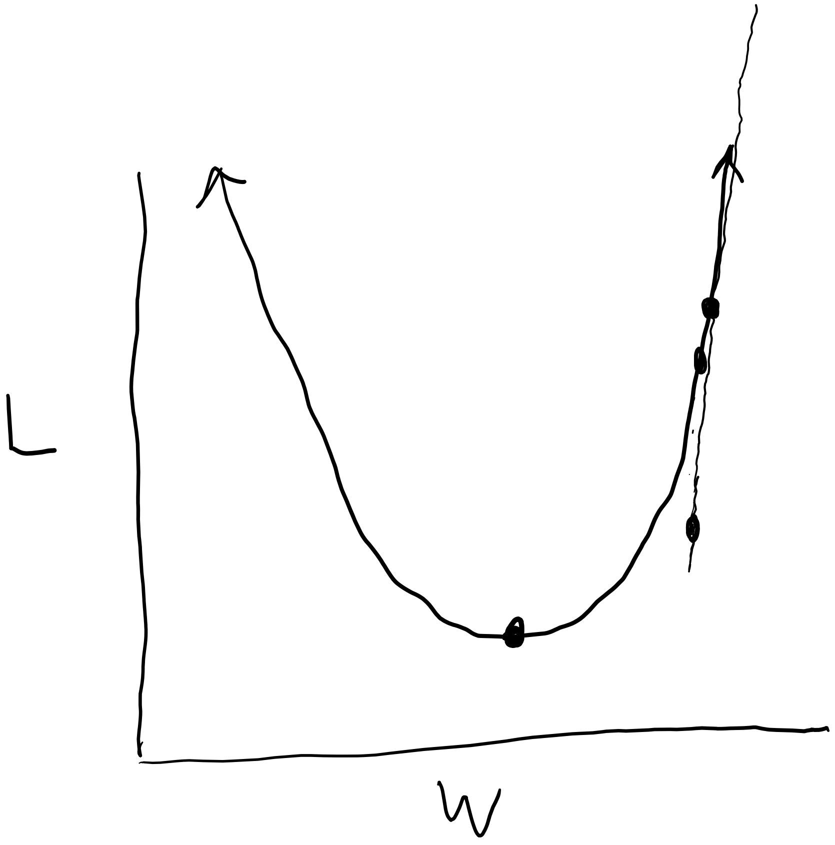


Simple example

In the most simple case, we pick a random point on the function and find the slope (derivative)

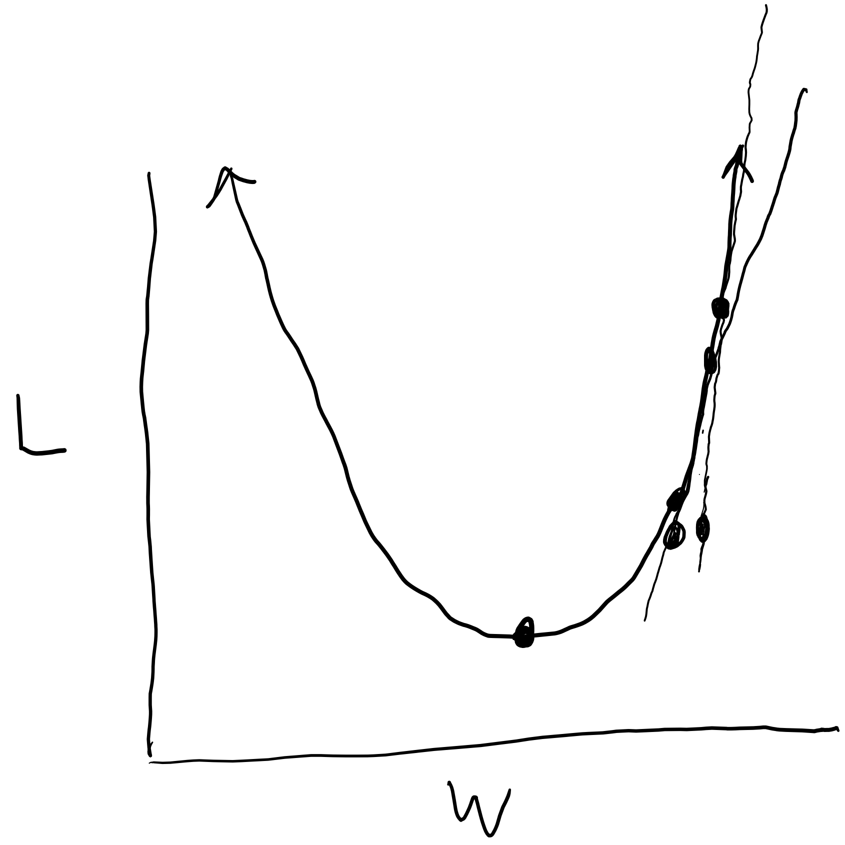
Then we move some incremental distance in the direction that **reduces** the value of L (left in this case)

This increment that we move each step is called the **learning rate**



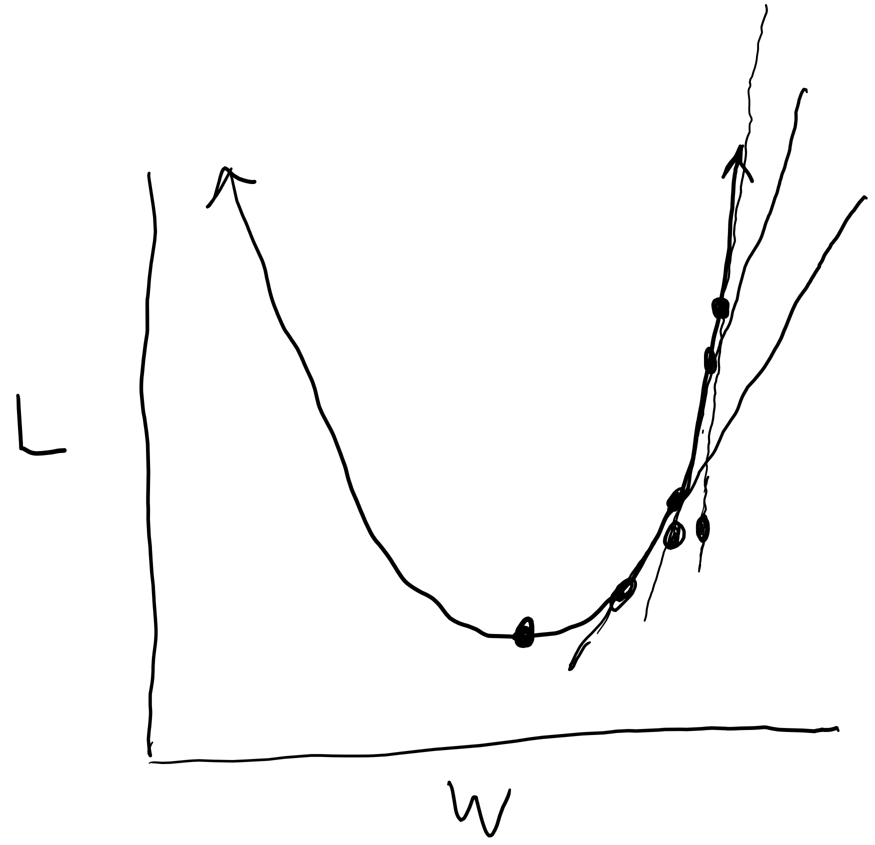
Simple example

Then we calculate the slope again at this new point and move one increment in the reducing-L direction (still left).



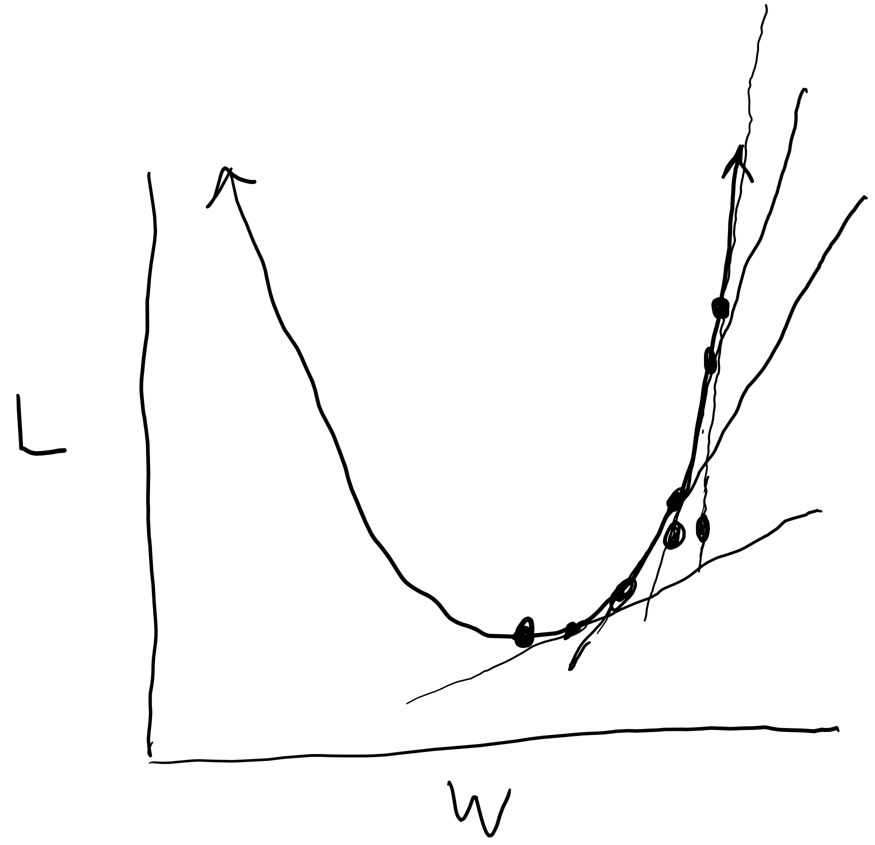
Simple example

And we keep doing that…



Simple example

And keep doing that…



Simple example

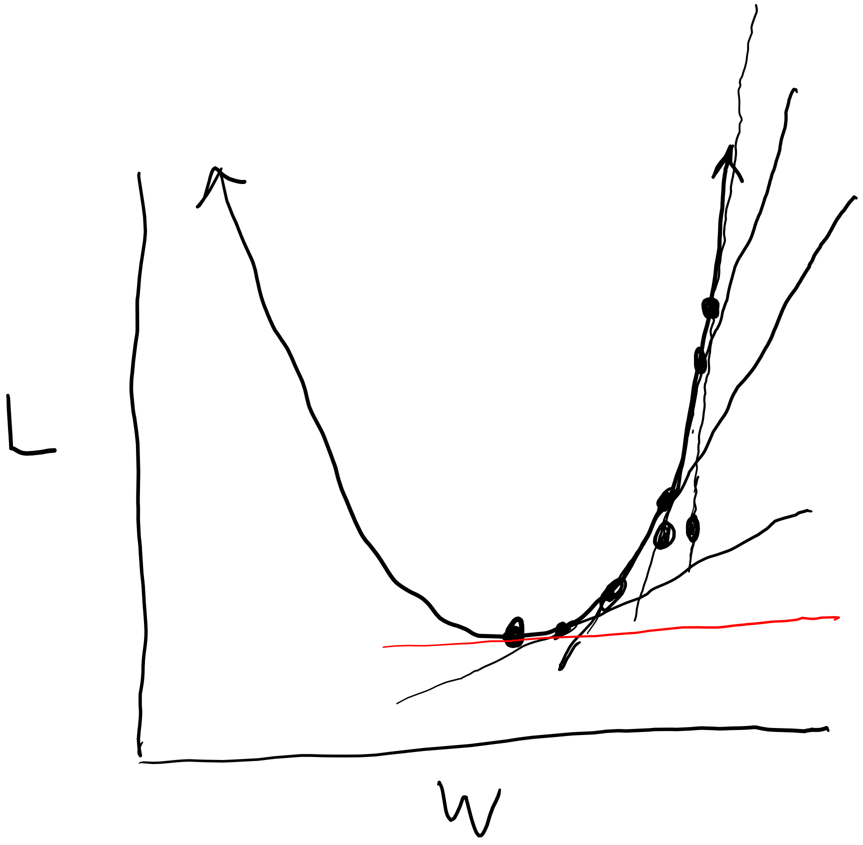
Until we hit a point on W where the slope seems to have levelled out

That is, $\frac{dL}{dW} = 0$

And we conclude that we've found the value of W that minimizes L

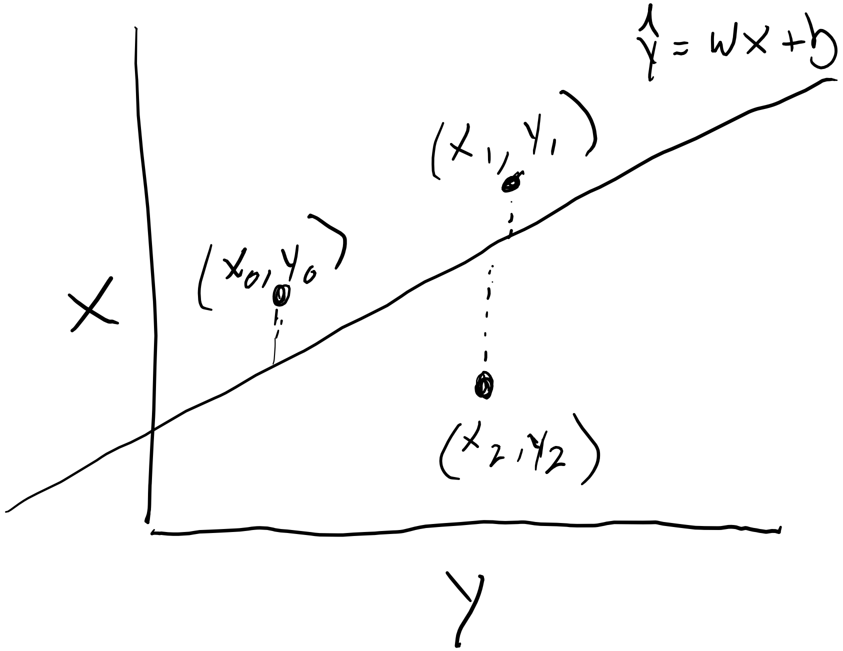
Challenge question:

- What if the learning rate is too high?
- What if it is too low?



Adding back the intercept

So what if our function **does** have an intercept?



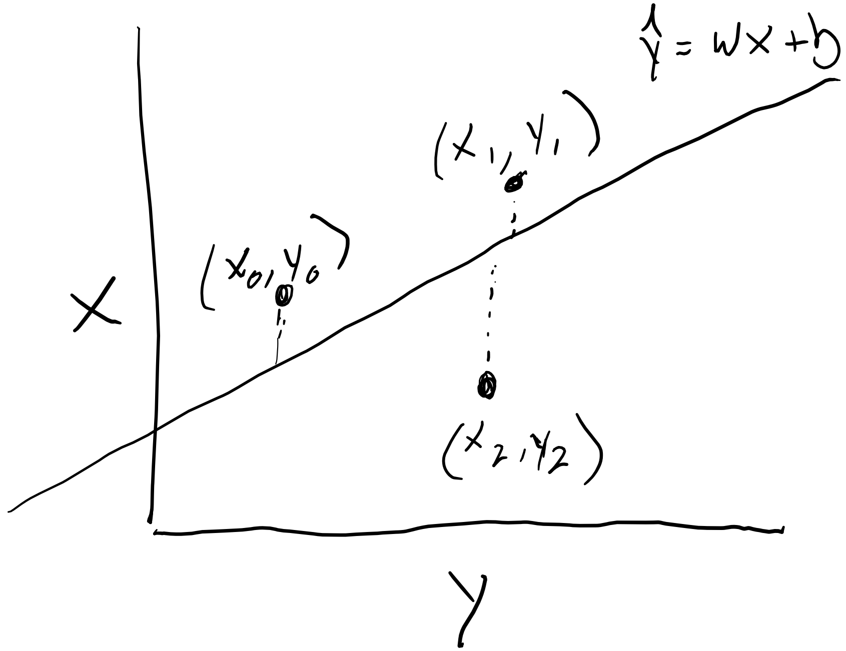
$$L(W, b) = \sum_i (\hat{y}_i - y_i)^2$$
$$= (\hat{y}_0 - y_0)^2 + (\hat{y}_1 - y_1)^2 + (\hat{y}_2 - y_2)^2$$
$$= (Wx_0 + b - y_0)^2 + (Wx_1 + b - y_1)^2 + (Wx_2 + b - y_2)^2$$
$$= (W^2{x_0}^2 - 2Wx_0y_0 + 2Wbx_0 - b2y_0 + {y_0}^2 + b^2)$$
$$+(W^2{x_1}^2 - 2Wx_1y_1 + 2Wbx_1 - b2y_1 + {y_1}^2 + b^2)$$
$$+(W^2{x_2}^2 - 2Wx_2y_2 + 2Wbx_2 - b2y_2 + {y_2}^2 + b^2)$$

$$= W^2({x_0}^2 + {x_1}^2 + {x_2}^2)$$
$$-2W(x_0y_0 + x_1y_1 + x_2y_2)$$
$$+2Wb(x_0 + x_1 + x_2)$$
$$-2b(y_0 + y_1 + y_2)$$
$$+({y_0}^2 + {y_1}^2 + {y_2}^2)$$
$$+3b^2$$

Adding back the intercept

So what if our function **does** have an intercept?

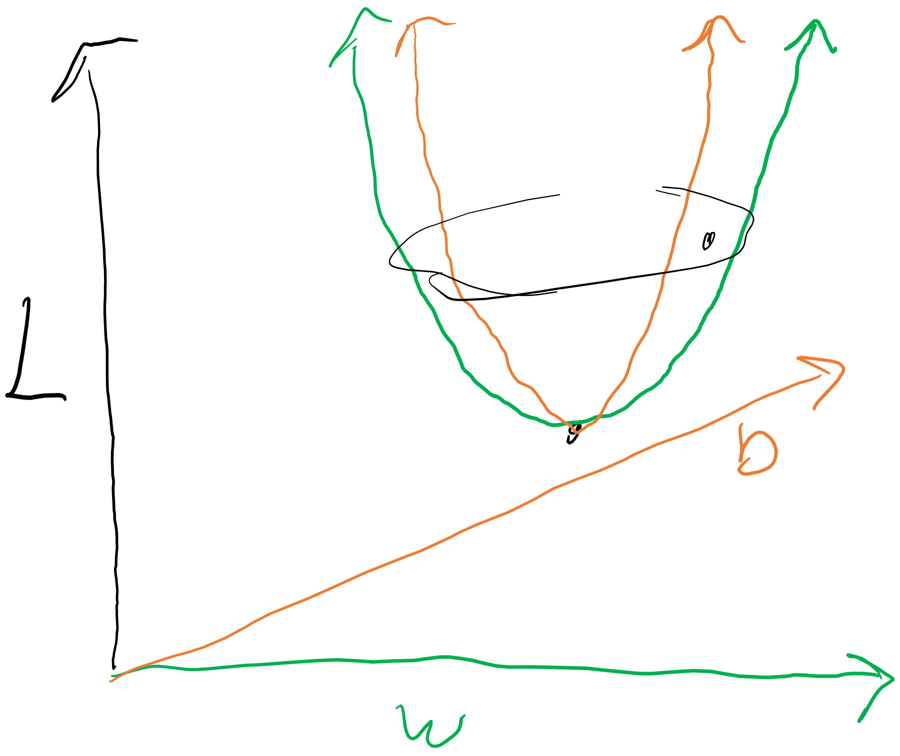
$$
\begin{aligned}
L(W,b) = {} & W^2(x_0{}^2 + x_1{}^2 + x_2{}^2) \\
& -2W(x_0y_0 + x_1y_1 + x_2y_2) \\
& +2Wb(x_0 + x_1 + x_2) \\
& -2b(y_0 + y_1 + y_2) \\
& +(y_0{}^2 + y_1{}^2 + y_2{}^2) \\
& +3b^2
\end{aligned}
$$



More complicated, but the key thing is that L is still just a quadratic function of W and b

Adding back the intercept

So the loss becomes a 2-dimensional function, and we're trying to find a value for W and a value for b, which, taken together, minimize L



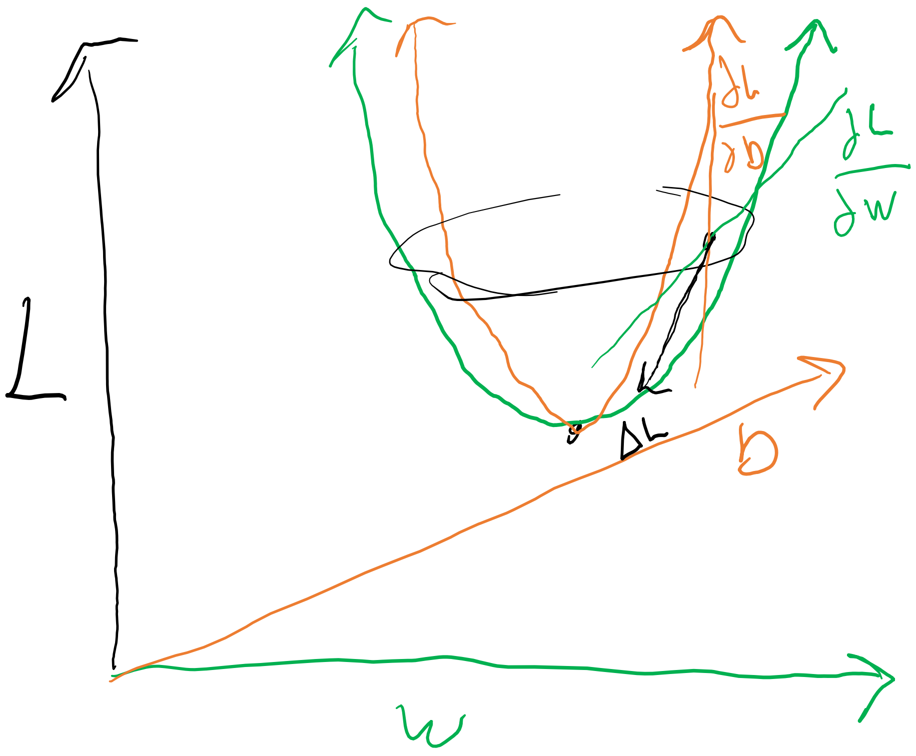
Adding back the intercept

We can still use gradient descent though!

Only now, instead of following the **derivative** $\frac{dL}{dW}$ of L with respect to W to the bottom…

We now follow a **vector** composed of the partial derivates of L with respect to W and b: $\left(\frac{\partial L}{\partial W}, \frac{\partial L}{\partial b}\right)$

We call this vector the **gradient** of L with respect to W and b, and usually denote it with the Δ symbol, e.g. Δ_L(W,b)



Gradient descent

Gradient descent is the method by which all neural nets are trained.

It works* in any situation where it is possible to calculate the gradient of the loss with respect to the model parameters

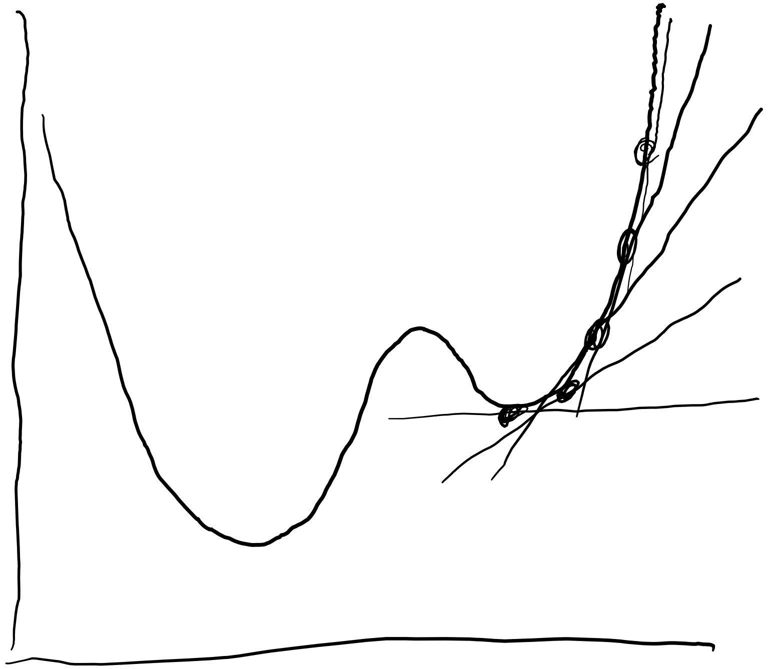
- $\hat{y} = Wx + b$ has two parameters: W and b
- ChatGPT has 175 billion parameters

*: it works more or less well depending on the shape of the loss function.

- If the function is a nice **convex** "bucket", then it will **always** find the global minimum.
- But this is not usually true

Local minima

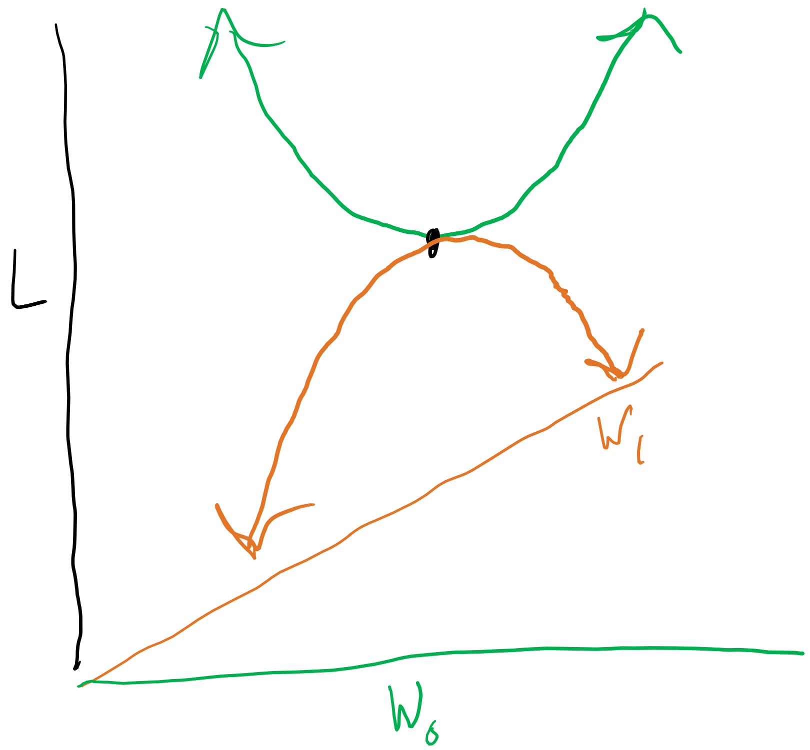
One issue in gradient descent is “local minima” which are false “dips” the gradient descent can get stuck in.



Saddle points

Another issue is “saddle points” which represent a minimum for one parameter but a maximum for a different one.

The gradient for both can be zero here… but it’s not necessarily a very good solution.



Advanced gradient descent

Advanced gradient descent algorithms have various tricks to help them avoid local minima and other issues

> arXiv
> https://arxiv.org › cs
>
> [1412.6980] Adam: A Method for Stochastic Optimization - arXiv
>
> by DP Kingma · 2014 · Cited by 137055 — Abstract: We introduce **Adam**, an algorithm for first-order gradient-based optimization of stochastic objective functions, based on adaptive...
>
> Cite as: arXiv:1412.6980

Most popular: **Adam**

- Uses "momentum" to learn adaptive learning rate for each parameter
- Generally the default choice for optimizing any arbitrary neural net

Long story short: just use Adam for everything

- Unless you have a good reason not to

Logistic regression

Logistic regression

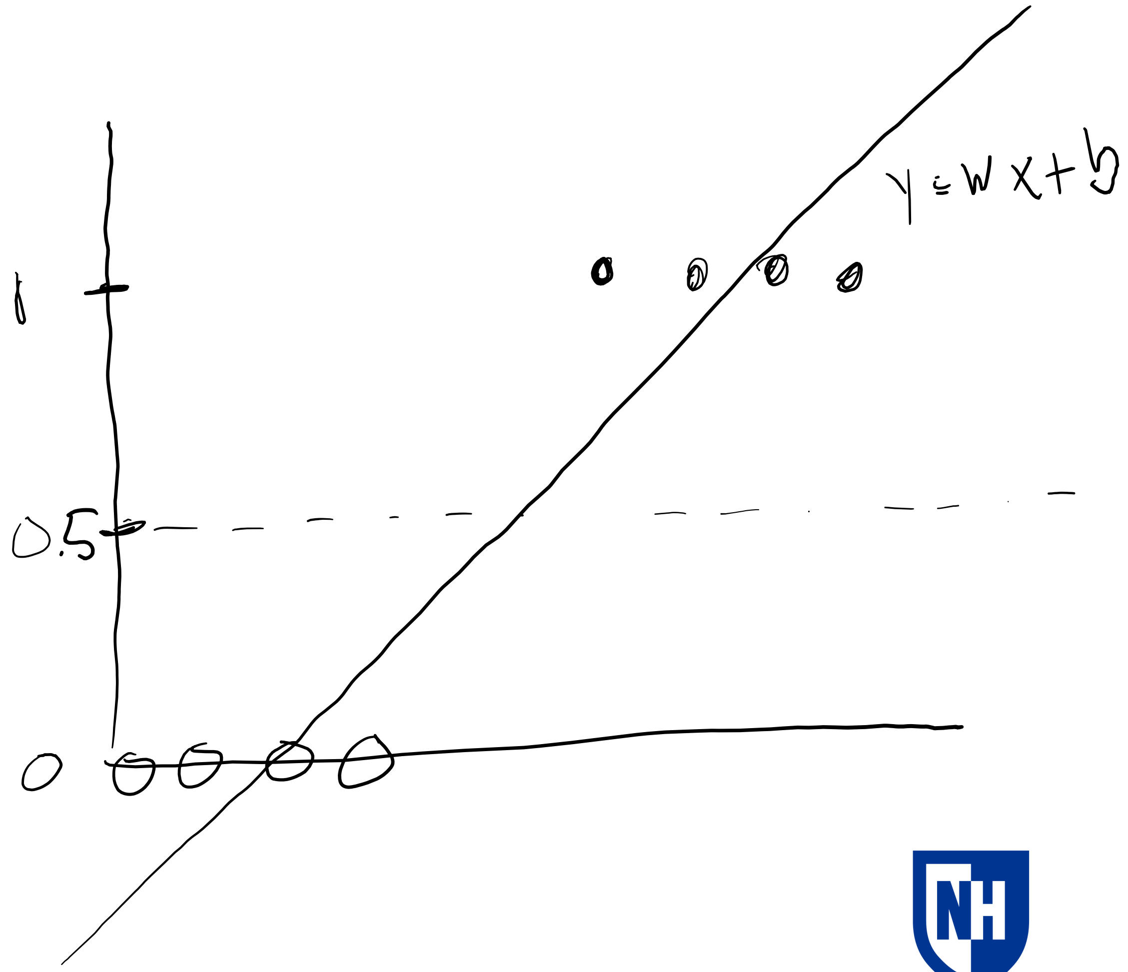
Generally in NLP we're more interested in **classification** than regression

- Mapping input x's to a discrete category rather than a continuous value

Linear regression not ideal for this

We can do it hackily with a threshold on the predicted value

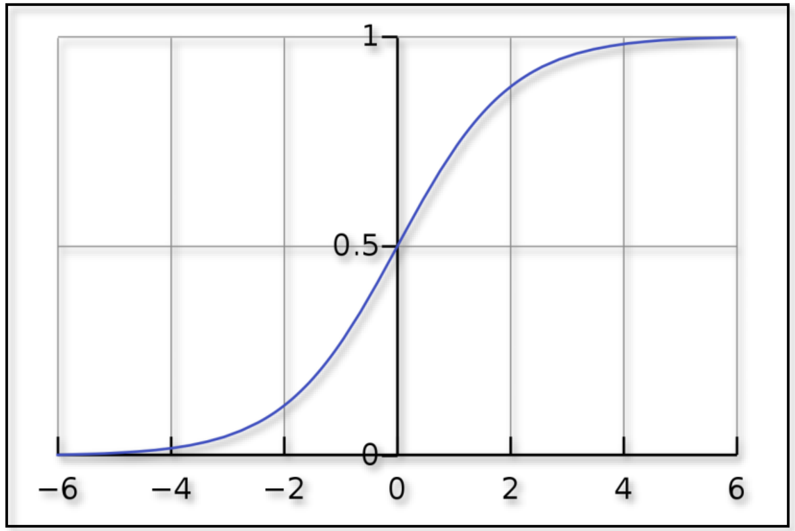
- But this has problems



Logistic function

To solve this problem, we are going to wrap our original function, which we will call *f*, in a **logistic function**

$$\sigma(x) = \frac{1}{1+e^{-x}} \qquad \sigma(f(x)) = \frac{1}{1+e^{-(Wx+b)}}$$



https://en.wikipedia.org/wiki/Logistic_function

One nice thing about it is that it is easy to differentiate because of the property that:

$$\frac{d}{dx}\sigma(x) = \sigma(x)(1-\sigma(x))$$

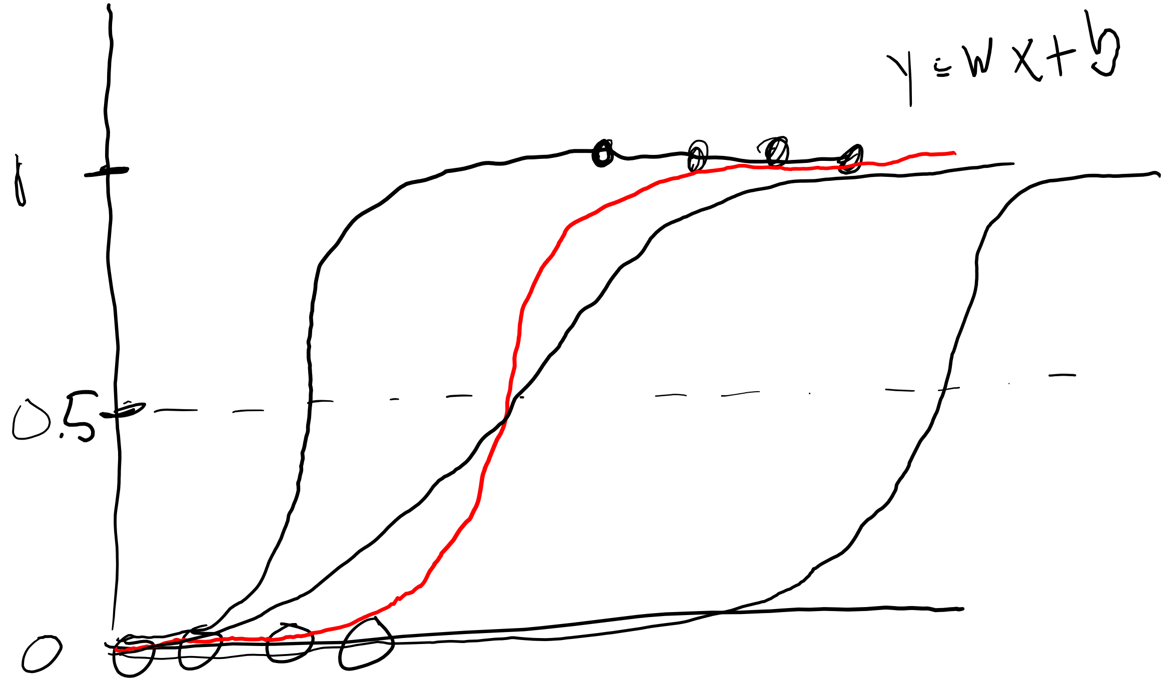
So if we call our original function *f*:

$$\frac{d}{dx}\sigma(f(x)) = \sigma(f(x))\big(1-\sigma(f(x))\big)f'(x) = W\sigma(Wx+b)(1-\sigma(Wx+b))$$

Logistic regression

So now instead of trying to fit a straight line to the data, we're trying to choose W and b to fit this S-shaped logistic curve to the data

Different choices for W and b change how steep the curve is and where it is centered.



Gradient descent for logistic regression

I won't do the full derivation, but:

- The function we're trying to fit is differentiable
- Which means we can create a differentiable loss function
- Which means we can do gradient descent!

However: mean squared error is not always convex for logistic regression

So we typically use **cross-entropy** loss as our objective:

$$L(y, \hat{y}) = \sum_{c} y_c \log(\hat{y}_c)$$

- Sum across possible classes of true value for that class multiplied by predicted log-probability of that class

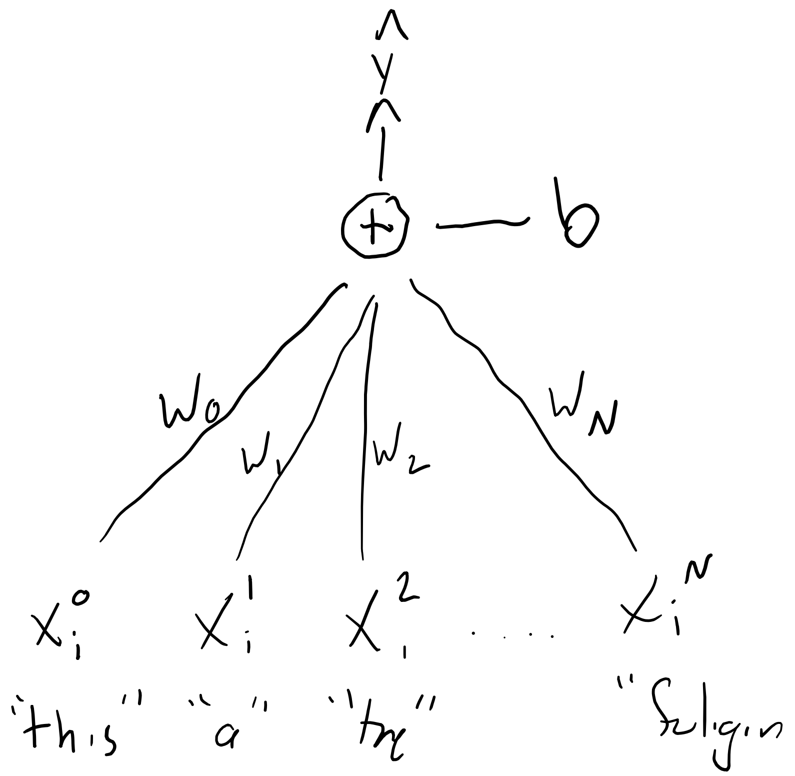
More detailed discussion available in Speech and Language Processing chapter 5: https://web.stanford.edu/~jurafsky/slp3/5.pdf

Visualizing linear regression

You can think of linear regression as a vector operation between matrices of x’s, W’s and ys

$$\begin{bmatrix} x_0^0 & x_0^1 & x_0^2 & \cdots & x_0^N \\ x_1^0 & x_1^1 & x_1^2 & \cdots & x_1^N \\ & & \ddots & & \\ x_M^0 & x_M^1 & x_M^2 & \cdots & x_M^N \end{bmatrix} \begin{bmatrix} W_0 \\ W_1 \\ \vdots \\ W_N \end{bmatrix} + b = \begin{bmatrix} \hat{y}_0 \\ \hat{y}_1 \\ \vdots \\ \hat{y}_M \end{bmatrix}$$

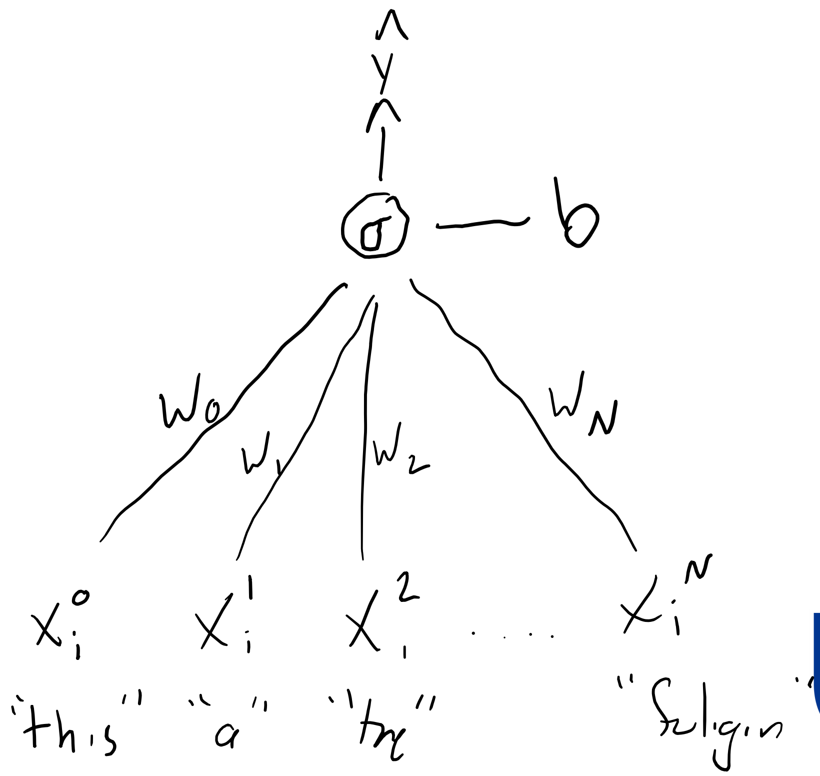
Or in a graphical form which shows how the individual x’s come together to form $\hat{y}$



Visualizing logistic regression

You can do the same thing for logistic regression by adding the σ function

$$\sigma\left(\begin{bmatrix} x_0^0 & x_0^1 & x_0^2 & \cdots & x_0^N \\ x_1^0 & x_1^1 & x_1^2 & \cdots & x_1^N \\ & & \cdots & & \\ x_M^0 & x_M^1 & x_M^2 & \cdots & x_M^N \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_N \end{bmatrix} + b\right) = \begin{bmatrix} \hat{y}_0 \\ \hat{y}_1 \\ \vdots \\ \hat{y}_M \end{bmatrix}$$



Visualizing logistic regression

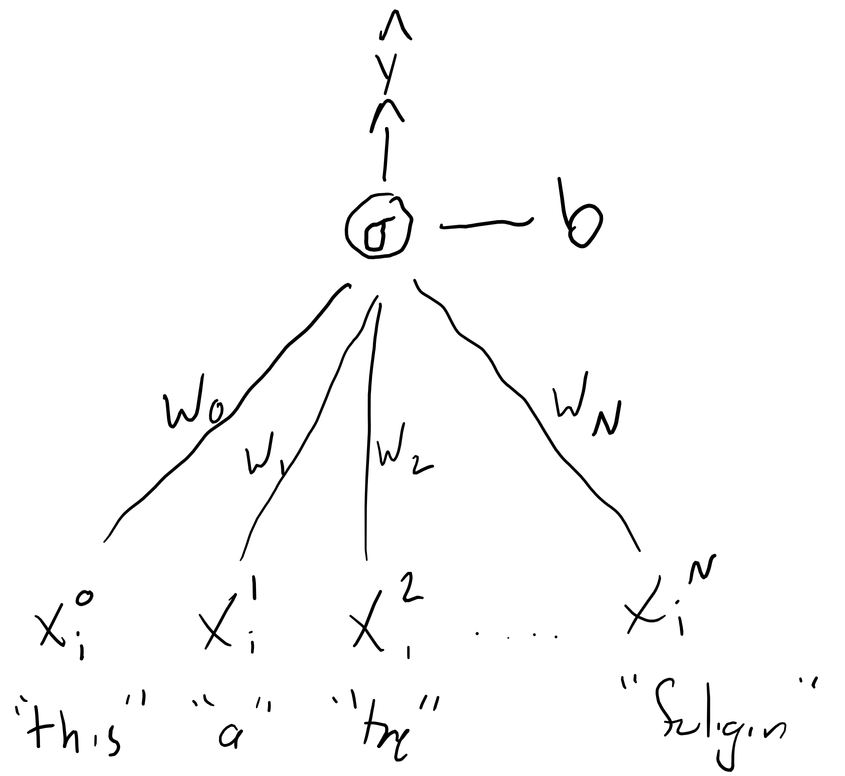
When we think of the logistic function as a final step being placed on top of the weighted sum $Wx + b$ in order to squeeze it down to [0,1], then we call it an **activation function**

There are a bunch of activation functions commonly used in neural nets:

- Rectified linear (relu), tanh, etc…

https://pytorch.org/docs/stable/nn.html#non-linear-activations-weighted-sum-nonlinearity

But logistic is the classic one



Linear vs logistic regression in practice

Read the SST-2 dataset

```
display(dev_df)
```

	sentence	label
0	it 's a charming and often affecting journey .	1
1	unflinchingly bleak and desperate	0
2	allows us to hope that nolan is poised to emba...	1
3	the acting , costumes , music , cinematography...	1
4	it 's slow -- very , very slow .	0
...	...	...
867	has all the depth of a wading pool .	0
868	a movie with a real anarchic flair .	1
869	a subject like this should inspire reaction in...	0
870	... is an arthritic attempt at directing by ca...	0
871	looking aristocratic , luminous yet careworn i...	1

872 rows × 2 columns

Preprocessing and vectorizing the data

```python
# SST-2 is actually already lowercased and tokenized, but it's good to get in
# the habit of always doing these during preprocessing
stemmer = PorterStemmer()
def preprocess(s):
    return ' '.join([stemmer.stem(token) for token in word_tokenize(s.lower())])
```

```python
train_df['preprocessed'] = train_df['sentence'].apply(preprocess)
dev_df['preprocessed'] = dev_df['sentence'].apply(preprocess)
```

```python
display(dev_df)
```

	sentence	label	preprocessed
0	it 's a charming and often affecting journey .	1	it 's a charm and often affect journey .
1	unflinchingly bleak and desperate	0	unflinchingli bleak and desper
2	allows us to hope that nolan is poised to emba...	1	allow us to hope that nolan is pois to embark ...
3	the acting , costumes , music , cinematography...	1	the act , costum , music , cinematographi and ...
4	it 's slow -- very , very slow .	0	it 's slow -- veri , veri slow .
...	...	...	...
867	has all the depth of a wading pool .	0	ha all the depth of a wade pool .
868	a movie with a real anarchic flair .	1	a movi with a real anarch flair .
869	a subject like this should inspire reaction in...	0	a subject like thi should inspir reaction in i...
870	... is an arthritic attempt at directing by ca...	0	... is an arthrit attempt at direct by calli k...
871	looking aristocratic , luminous yet careworn i...	1	look aristocrat , lumin yet careworn in jane h...

872 rows × 3 columns

Preprocessing and vectorizing the data

```python
from sklearn.feature_extraction.text import CountVectorizer
```

```python
vectorizer = CountVectorizer()
train_X = vectorizer.fit_transform(train_df['preprocessed'])
display(train_X)
```

```
<67349x10106 sparse matrix of type '<class 'numpy.int64'>'
	with 535539 stored elements in Compressed Sparse Row format>
```

```python
dev_X = vectorizer.transform(dev_df['preprocessed'])
display(dev_X)
```

```
<872x10106 sparse matrix of type '<class 'numpy.int64'>'
	with 12939 stored elements in Compressed Sparse Row format>
```

Linear regression - Training

```python
from sklearn.linear_model import LinearRegression
```

```python
# As with most cases, it's very easy to instantiate and train a linear regression model
# in scikit-learn

# Note: SST-2 is NOT a regression task. It is a classification task.

# However, because the label is either 0 or 1, we can still sort of treat it as
# a regression task, by treating those as target values for the model.

# This only works for ordinal classification tasks, where there's an easy
# way of converting from labels to numbers

lin_reg_model = LinearRegression()
lin_reg_model.fit(train_X, train_df['label'])
```

```
LinearRegression()
```

Linear regression - Training

```python
# Notice that because it is a regression model, the predictions are continuous
# values that aren't necessarily bounded between 0 and 1
train_df['lin_reg_prediction'] = lin_reg_model.predict(train_X)
dev_df['lin_reg_prediction'] = lin_reg_model.predict(dev_X)

display(dev_df)
```

	sentence	label	preprocessed	lin_reg_prediction
0	it 's a charming and often affecting journey .	1	it 's a charm and often affect journey .	1.156662
1	unflinchingly bleak and desperate	0	unflinchingli bleak and desper	-0.069508
2	allows us to hope that nolan is poised to emba...	1	allow us to hope that nolan is pois to embark ...	0.919836
3	the acting , costumes , music , cinematography...	1	the act , costum , music , cinematographi and ...	0.934645
4	it 's slow -- very , very slow .	0	it 's slow -- veri , veri slow .	0.031911
...	...	...	...	...
867	has all the depth of a wading pool .	0	ha all the depth of a wade pool .	0.514821
868	a movie with a real anarchic flair .	1	a movi with a real anarch flair .	1.922209
869	a subject like this should inspire reaction in...	0	a subject like thi should inspir reaction in i...	0.914590
870	... is an arthritic attempt at directing by ca...	0	... is an arthrit attempt at direct by calli k...	0.326833
871	looking aristocratic , luminous yet careworn i...	1	look aristocrat , lumin yet careworn in jane h...	0.098285

872 rows × 4 columns

Linear regression - Evaluation

```python
# We can't use classification metrics when we do regression, because those only work
# for discrete categorical values

# Instead, we measure the average discrepancy between target value and true value
# The two most common metrics are mean squared error (MSE) and mean absolute error (MAE)

from sklearn.metrics import mean_squared_error, mean_absolute_error

def evaluate_regression(y, py):
  print(f'Mean squared error: {mean_squared_error(y,py):.3f}')
  print(f'Mean absolute error: {mean_absolute_error(y,py):.3f}')
```

```python
# Smaller values are better. Notice how overfitted we are to the training data

print('Train linear regression results:')
evaluate_regression(train_df['label'], train_df['lin_reg_prediction'])
print('\nDev linear regression results:')
evaluate_regression(dev_df['label'], dev_df['lin_reg_prediction'])
```

```
Train linear regression results:
Mean squared error: 0.078
Mean absolute error: 0.219

Dev linear regression results:
Mean squared error: 0.389
Mean absolute error: 0.472
```

Linear regression - Evaluation

```python
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
import numpy as np

# 0.5 is a reasonable value for the threshold in this circumstance
def evaluate_thresholded_regression(y, py, threshold=0.5):

    # Numpy vectors/pandas Series can be compared to scalars, producing a bool
    # vector, which the classification metric functions can handle
    t_y = (y >= threshold)
    t_py = (py >= threshold)

    print(f'Accuracy: {accuracy_score(t_y, t_py):.3f}')
    print(f'Precision: {precision_score(t_y, t_py):.3f}')
    print(f'Recall: {recall_score(t_y, t_py):.3f}')
    print(f'F1: {f1_score(t_y, t_py):.3f}')
```

```python
# We were doing better with Naive Bayes a couple weeks ago.
print('Thresholded train linear regression results:')
evaluate_thresholded_regression(train_df['label'], train_df['lin_reg_prediction'])
print('\nThresholded dev linear regression results:')
evaluate_thresholded_regression(dev_df['label'], dev_df['lin_reg_prediction'])
```

```
Thresholded train linear regression results:
Accuracy: 0.927
Precision: 0.919
Recall: 0.953
F1: 0.936

Thresholded dev linear regression results:
Accuracy: 0.744
Precision: 0.735
Recall: 0.779
F1: 0.756
```

Linear regression - Global explanations

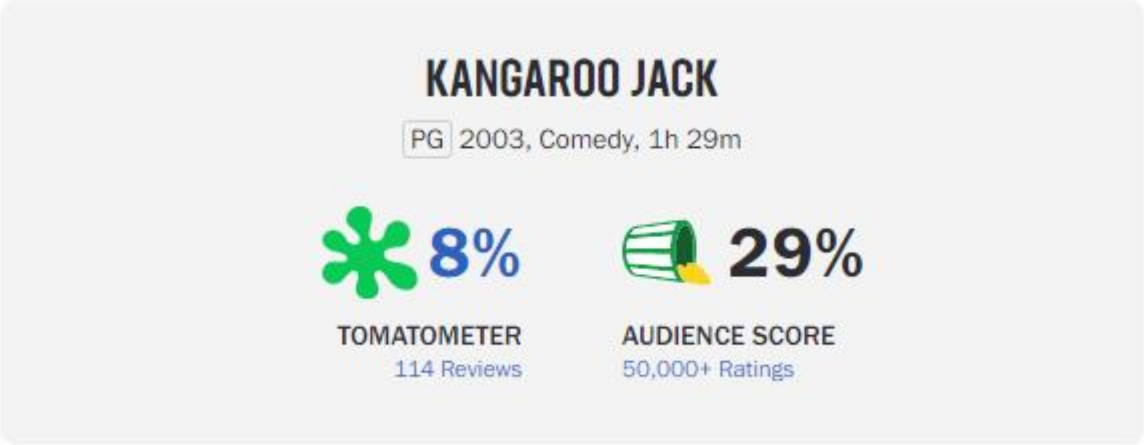
```
def explain_binary_linear_model(model, vocabulary, k=10):
  # This is necessary because the coefs for logistic regression can be a 2D matrix
  # (though they won't be in this notebook)
  model_coefs = np.squeeze(model.coef_)
  sorted_coef_indices = np.argsort(np.abs(model_coefs))
  # Grab the indices of the top k indices and flip their order to descending
  top_k_indices = sorted_coef_indices[:-k:-1]
  # Numpy vectors can be indexed by multiple indices at once
  top_k_coefs = model_coefs[top_k_indices]
  # Python lists aren't so flexible, so we have to use a comprehension
  top_k_words = [vocabulary[index] for index in top_k_indices]
  for word, coef in zip(top_k_words, top_k_coefs):
    print(f'\tWord: {word} - Coef: {coef:.3f}')
```

```
# These words make... a lot less sense than the ones we got back for Naive Bayes
# Why might that be?
print(f'Top 10 coefficients in our linear regression model:')
explain_binary_linear_model(lin_reg_model, vocabulary)
```

```
Top 10 coefficients in our linear regression model:
	Word: edmund - Coef: -2.093
	Word: embed - Coef: 1.660
	Word: schtick - Coef: 1.619
	Word: interchang - Coef: 1.581
	Word: drung - Coef: 1.561
	Word: purgatori - Coef: -1.548
	Word: size - Coef: -1.380
	Word: wire - Coef: 1.357
	Word: kangaroo - Coef: -1.353
```

“Kangaroo” and “Edmund”

“Kangaroo”



“Edmund” ???

Linear regression – Local explanations

```python
# We can also generate explanations for individual predictions using similar logic

def explain_binary_linear_model_prediction(input,model, vectorizor):

    # This is necessary because the coefs for logistic regression can be a 2D matrix
    # (though they won't be in this notebook)
    model_coefs = np.squeeze(model.coef_)

    # Assume the input hasn't been preprocessed, so do that and then vectorize it
    preprocessed = preprocess(input)
    tokens = preprocessed.split(' ')
    input_X = vectorizer.transform([preprocessed])

    py = model.predict(input_X)[0]
    print(f'Prediction: {py}')

    print(f'Word coefficients:')
    for token in tokens:
        if token in vectorizer.vocabulary_: # Skip any tokens that are not in the vectorizer vocab
            token_index = vectorizer.vocabulary_[token]
            token_coef = model_coefs[token_index]
            print(f'\tWord: {token} - Coef: {token_coef:.3f}')

    print(f'Model intercept: {model.intercept_}')
```

Linear regression – Local explanations

```
# We can see in this example that the prediction is just below 0.5
# "poorly" is considered much more negative than "well" was positive,
# but the high intercept almost pushes the total sum above 0.5

explain_binary_linear_model_prediction('the movie was well written but poorly acted.',
                                       lin_reg_model,
                                       vectorizer)
```

```
Prediction: 0.47644153366011277
Word coefficients:
	Word: the - Coef: -0.005
	Word: movi - Coef: 0.002
	Word: wa - Coef: -0.079
	Word: well - Coef: 0.161
	Word: written - Coef: 0.087
	Word: but - Coef: 0.047
	Word: poorli - Coef: -0.337
	Word: act - Coef: -0.001
Model intercept: 0.6021082139311205
```

Binary logistic regression – Training

```python
from sklearn.linear_model import LogisticRegression
```

```python
log_reg_model = LogisticRegression()
log_reg_model.fit(train_X, train_df['label'])
```

```
/usr/local/lib/python3.8/dist-packages/sklearn/linear_model/_logistic.py:814: ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
LogisticRegression()
```

Linear regression – Training

```
train_df['log_reg_prediction'] = log_reg_model.predict(train_X)
dev_df['log_reg_prediction'] = log_reg_model.predict(dev_X)

display(dev_df)
```

	sentence	label	preprocessed	lin_reg_prediction	log_reg_prediction
0	it 's a charming and often affecting journey .	1	it 's a charm and often affect journey .	1.156662	1
1	unflinchingly bleak and desperate	0	unflinchingli bleak and desper	-0.069508	0
2	allows us to hope that nolan is poised to emba...	1	allow us to hope that nolan is pois to embark ...	0.919836	1
3	the acting , costumes , music , cinematography...	1	the act , costum , music , cinematographi and ...	0.934645	1
4	it 's slow -- very , very slow .	0	it 's slow -- veri , veri slow .	0.031911	0
...	...	...	...	...	...
867	has all the depth of a wading pool .	0	ha all the depth of a wade pool .	0.514821	0
868	a movie with a real anarchic flair .	1	a movi with a real anarch flair .	1.922209	1
869	a subject like this should inspire reaction in...	0	a subject like thi should inspir reaction in i...	0.914590	0
870	... is an arthritic attempt at directing by ca...	0	... is an arthrit attempt at direct by calli k...	0.326833	0
871	looking aristocratic , luminous yet careworn i...	1	look aristocrat , lumin yet careworn in jane h...	0.098285	1

872 rows × 5 columns

Logistic regression – Evaluation

```
# Now we don't have to do any thresholding of our own, so we can
# just use the classification metrics as-is
def evaluate_classification(y, py):
  print(f'Accuracy: {accuracy_score(y, py):.3f}')
  print(f'Precision: {precision_score(y, py):.3f}')
  print(f'Recall: {recall_score(y, py):.3f}')
  print(f'F1: {f1_score(y, py):.3f}')
```

```
# Looking better! This is pretty similar to what we were getting with Naive Bayes
# Still some overfitting happening though.
print('Train logistic regression results:')
evaluate_classification(train_df['label'], train_df['log_reg_prediction'])
print('\nDev logistic regression results:')
evaluate_classification(dev_df['label'], dev_df['log_reg_prediction'])
```

```
Train logistic regression results:
Accuracy: 0.926
Precision: 0.927
Recall: 0.940
F1: 0.934

Dev logistic regression results:
Accuracy: 0.808
Precision: 0.794
Recall: 0.842
F1: 0.817
```

```
Thresholded train linear regression results:
Accuracy: 0.927
Precision: 0.919
Recall: 0.953
F1: 0.936

Thresholded dev linear regression results:
Accuracy: 0.744
Precision: 0.735
Recall: 0.779
F1: 0.756
```

Logistic regression – global explanations

```
# These look better
print(f'Top 10 coefficients in our logistic regression model:')
explain_binary_linear_model(log_reg_model, vocabulary)
```

```
Top 10 coefficients in our logistic regression model:
        Word: lack - Coef: -4.389
        Word: worst - Coef: -4.069
        Word: anatom - Coef: 4.015
        Word: devoid - Coef: -3.871
        Word: failur - Coef: -3.703
        Word: refresh - Coef: 3.654
        Word: stupid - Coef: -3.521
        Word: assum - Coef: -3.437
        Word: mess - Coef: -3.407
```

Logistic regression – Local explanations

```python
# We can explain individual predictions the same way too

# Interestingly, we see the same effect with "was" as we observed when we
# did Naive Bayes
explain_binary_linear_model_prediction('the movie was well written but poorly acted.',
                                       log_reg_model,
                                       vectorizer)
```

```
Prediction: 0
Word coefficients:
	Word: the - Coef: -0.023
	Word: movi - Coef: -0.054
	Word: wa - Coef: -0.903
	Word: well - Coef: 1.488
	Word: written - Coef: 0.483
	Word: but - Coef: 0.077
	Word: poorli - Coef: -2.898
	Word: act - Coef: -0.061
Model intercept: [0.36910711]
```

Concluding thoughts

Linear regression

- Learn $Wx + b$ from data
- Predict continuous values
- Optimize mean squared error

Logistic regression

- Learn $\sigma(Wx + b)$ from data
- Predict (close to) 0 or 1
- Optimize cross-entropy

Key concepts:

- Loss function
    - I.e. objective function
- Gradient of loss with respect to parameters
- Gradient descent
- Activation function