Linear Classifiers in Python
by Mike Gelbart

Linear classifiers are a type of machine learning algorithm used for classification tasks, where the goal is to assign input data points to different classes or categories. Linear classifiers work by learning a linear decision boundary that separates different classes in the feature space. The decision boundary is defined by a linear combination of the input features, and it helps in making predictions about the class label of new, unseen data points.

Here are some key points about linear classifiers:

1. **Linear Decision Boundary:** Linear classifiers assume that the decision boundary that separates classes is a linear function of the input features. For example, in a two-dimensional feature space, the decision boundary could be a straight line.

2. **Binary and Multiclass Classification:** Linear classifiers can be used for both binary classification (two classes) and multiclass classification (more than two classes). For binary classification, a single decision boundary is learned. For multiclass classification, multiple decision boundaries are learned, each corresponding to a different class.

3. **Feature Transformation:** Linear classifiers can perform well when the features are linearly separable in the given feature space. If the data is not linearly separable, feature transformation techniques can be applied to map the data to a higher-dimensional space where linear separability might be possible.

4. **Examples of Linear Classifiers:** Some popular linear classifiers include:
   - **Perceptron:** An early algorithm for binary classification that updates the decision boundary based on misclassified examples.
   - **Logistic Regression:** A probabilistic linear classifier that models the probability of a data point belonging to a certain class.
   - **Support Vector Machines (SVM):** A linear classifier that aims to find the hyperplane that maximally separates classes while considering a margin between them.
   - **Linear Discriminant Analysis (LDA):** A technique that finds a linear combination of features that best separates classes while minimizing the variance within each class.

5. **Loss Functions:** Linear classifiers use loss functions to quantify the

difference between predicted and actual class labels. The goal is to find the model parameters (coefficients) that minimize the loss function.

6. **Training:** Training a linear classifier involves finding the optimal values for the coefficients of the linear equation. This is often done using optimization techniques such as gradient descent, which iteratively updates the coefficients to minimize the loss function.

7. **Interpretability:** Linear classifiers are often considered interpretable because the coefficients provide insight into the relative importance of different features in making classification decisions.

8. **Limitations:** While linear classifiers are effective for many problems, they might not perform well when the decision boundary is highly nonlinear. In such cases, more complex classifiers like nonlinear SVMs or decision trees might be more suitable.

Overall, linear classifiers are a fundamental and widely used class of algorithms for classification tasks, offering simplicity, interpretability, and good performance on many types of data.Linear Classifiers in Python

supervised learning refers to learning a relationship from examples of input-output pairs usually called X and y
example - fitting and predicting

```
import sklearn.datasets
newsgroups = sklearn.datasets.fetch_20newsgroups_vectorized()
X, y = newsgroups.data, newsgroups.target
X.shape
output > (11314, 130107)
y.shape
output > (11314,)
```

what this says?
we have 11,000 training examples, each with about 130,000 features
in this dataset, features are derived from the words appearing in each news article
y-values are the article topics (this is also what we are trying to predict)

```
from sklearn.neighbors import KNeighborsClassifier
#instantiate and add in desired hyperparameters
knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X,y)
#once instantiated and fit, we can make predictions on any data
including the original data set which in this case and most cases is X
y_pred = knn.predict(X)
```

#y_pred no contains one entry per row of X with the prediction from the trained classifier
#running an accuracy score now
knn.score(X, y)
output > 0.99991
pretty darn good but not particularly meaningful because this is done on the ‘seen’ or training data
**the key is to see how well the model generalizes to ‘unseen’ data
how do we do this?
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y)
#test_size defaults to 25% of the examples
**key to refit model to training data
running a repeat score shows that the model runs only on the training set
knn.fit(X_train, y_train)
knn.score(X_test, y_test)
output > 0.66
score is clearly lower showing us that the training error was definitely a poor representation of the model’s ability to classify new data
the actual accuracy of 66% may be great or poor, this all depends on the situation

Logistic Regression
remember is a linear classifire
example - format on built-in wine dataset
import sklearn.datasets
wine = sklearn.datasets.load_wine()
from sklearn.linear_model import LogisiticRegression
lr = LogisticRegression()
lr.fit(wine.data, wine.target)
lr.score(wine.data, wine.target)
output > 0.966
#with logistic regression we do not have to do hard predictions we can also output confidence scores
can do this wih predict_proba function
lr.predict_proba(wine.data[:1])
output > array([[9.966e-01, 2.740e-03, 6.787e-04]])
*remember little is scientific notation and represents 10 to the power of
can interpret first probability as 9.9 x10 to power of -1 or > .99 ie 99%

Using LinearSVC
basic SVM classifier
works exactly like LogisiticRegression
import sklearn.datasets

```
wine = sklearn.datasets.load_wine()
from sklearn.svm import LinearSVC
svm = LinearSVC()
svm.fit(wine.data, wine.target)
svm.score(wine.data, wine.target)
output > 0.955
```

Using SVC
fits a nonlinear SVM by default

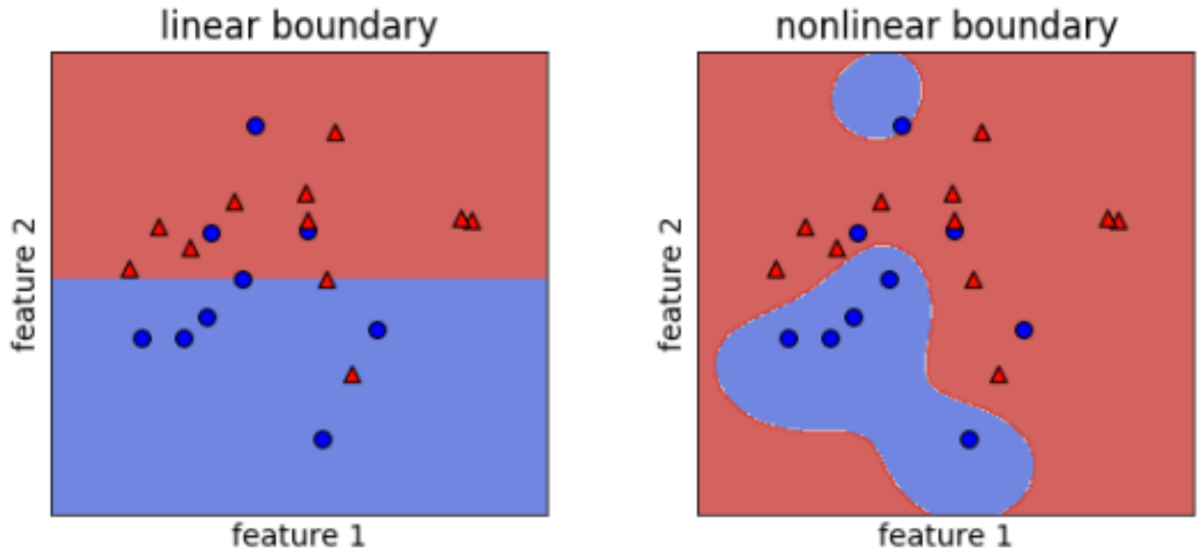
```
from sklearn.svm import SVC
svm = SVC()
svm.fit(wine.data, wine.target)
svm.score(wine.data, wine.target)
output > 0.708
```

*be aware that though this accuracy is not initially high we could tune the hyperparameters to achieve 100% training accuracy
*be aware the risk always with more complex models is overfitting

example

```
from sklearn import datasets
digits = datasets.load_digits()
X_train, X_test, y_train, y_test = train_test_split(digits.data, digits.target)

# Apply logistic regression and print scores
lr = LogisticRegression()
lr.fit(X_train, y_train)
print(lr.score(X_train, y_train))
print(lr.score(X_test, y_test))

# Apply SVM and print scores
svm = SVC()
svm.fit(X_train, y_train)
print(svm.score(X_train, y_train))
print(svm.score(X_test, y_test))

# Instantiate logistic regression and train
lr = LogisticRegression()
lr.fit(X, y)

# Predict sentiment for a glowing review
review1 = "LOVED IT! This movie was amazing. Top 10 this year."
review1_features = get_features(review1)
print("Review:", review1)
print("Probability of positive review:", lr.predict_proba(review1_features)[0,1])

# Predict sentiment for a poor review
review2 = "Total junk! I'll never watch a film by that director again, no matter how good the reviews."
review2_features = get_features(review2)
print("Review:", review2)
print("Probability of positive review:", lr.predict_proba(review2_features)[0,1])
```

Linear decision boundaries



a nonlinear boundary can lead to non-contiguous regions of certain prediction
*in their basic forms, logistic regression and SVMs are linear classifiers
*they do have nonlinear versions

Important vocabulary
‘classification’ is supervised learning when the y-values are categories
**in contrast with regression, where we are trying to predict a continuous value
‘decision boundary’ the surface separating different predicted classes
‘linear classifier’ a classifier that learn linear decision boundaries (examples - logistic regression, linear SVM, etc)
‘linearly separable’ a data set can be perfectly explained by a linear classifier

Sidebar - reminder on differences between classification and regression
Classification and regression are two fundamental types of supervised machine learning tasks that aim to make predictions based on input data and corresponding target variables. Here's a synopsis of the key differences between classification and regression:

1. Objective:
   - Classification: The main objective of classification is to categorize input data into predefined classes or categories. The target variable in classification is discrete and represents the class labels. Examples of classification tasks include spam email detection (categorizing emails into "spam" or "not spam") or image recognition (identifying objects in an image as "cat," "dog," "car," etc.).
   - Regression: The main objective of regression is to predict a continuous numeric value based on input data. The target variable in regression is continuous, and the model's output is a numeric value. Examples of regression tasks include predicting house prices based on features like size, number of rooms, and location or forecasting future sales based on historical data.

2. Output:
   - Classification: The output of a classification model is a discrete class label or a probability score for each class. The model assigns each data point to the class with the highest probability or a class based on a defined threshold.
   - Regression: The output of a regression model is a continuous value. The model generates a numeric prediction that represents the estimated value of the target variable.

3. Evaluation:
   - Classification: Classification models are evaluated using metrics like accuracy, precision, recall, F1-score, and area under the Receiver Operating Characteristic (ROC) curve, depending on the specific problem and class balance.
   - Regression: Regression models are evaluated using metrics such as mean squared error (MSE), root mean squared error (RMSE), mean absolute error (MAE), and R-squared (coefficient of determination).

4. Examples:
   - Classification: Examples of classification problems include sentiment analysis, disease diagnosis (positive/negative), customer churn prediction (stay/leave), and image classification.
   - Regression: Examples of regression problems include house price prediction, stock price forecasting, temperature prediction, and sales forecasting.

In summary, classification and regression are two distinct types of supervised learning tasks. Classification deals with discrete class labels and aims to assign input data to predefined categories, while regression deals with continuous numeric values and aims to predict a numerical outcome based on input features.

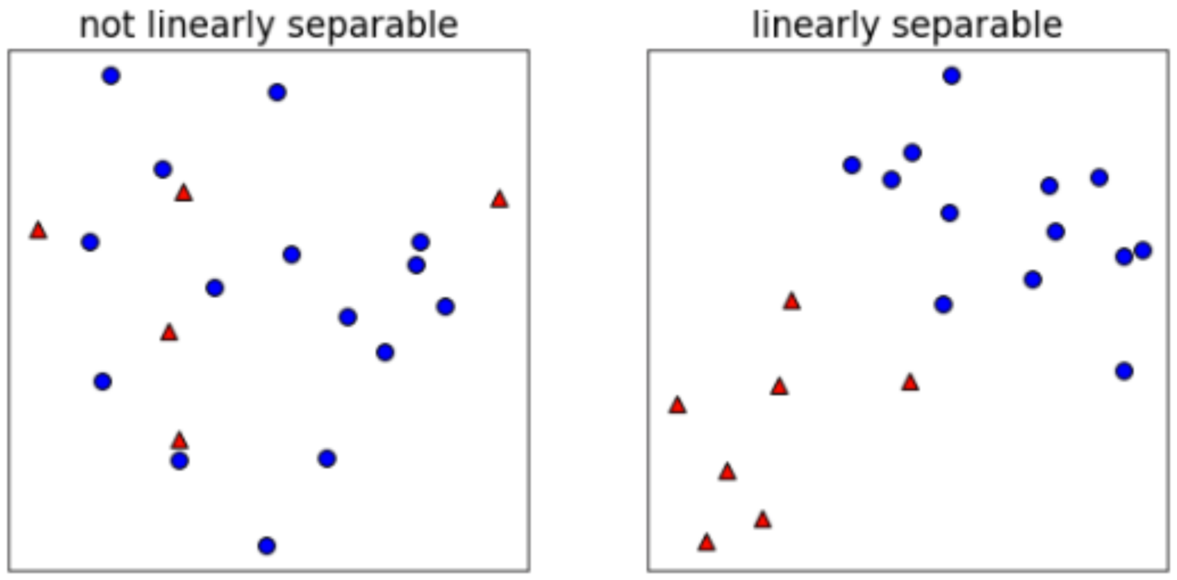
Sidebar - reminder on differences between linear and logistic regression
The main difference between linear regression and logistic regression lies in their objectives and the type of output they produce:

1. Linear Regression:
   - Objective: Linear regression is used for predicting a continuous target variable (numeric values). It models the relationship between the input features (independent variables) and the continuous target variable (dependent variable) by fitting a linear equation.
   - Output: The output of linear regression is a continuous value, which represents the predicted value of the target variable.

2. Logistic Regression:
   - Objective: Logistic regression is used for binary classification tasks, where the target variable has only two possible classes (e.g., 0 or 1, True or False). It models the probability of the binary outcome as a function of the input features.
   - Output: The output of logistic regression is the predicted probability of belonging to the positive class (class 1). To obtain the final binary classification, a threshold is applied to these probabilities (e.g., 0.5). If the predicted probability is above the threshold, the data point is classified as belonging to class 1; otherwise, it is classified as belonging to class 0.

In summary, the primary difference between linear regression and logistic regression is their modeling objective and the type of output they produce. Linear regression is used for predicting continuous numerical values, while logistic regression is used for binary classification tasks, providing predicted probabilities that are then converted to binary class labels using a threshold. Both linear and logistic regression can handle various types of data, including continuous and categorical variables.

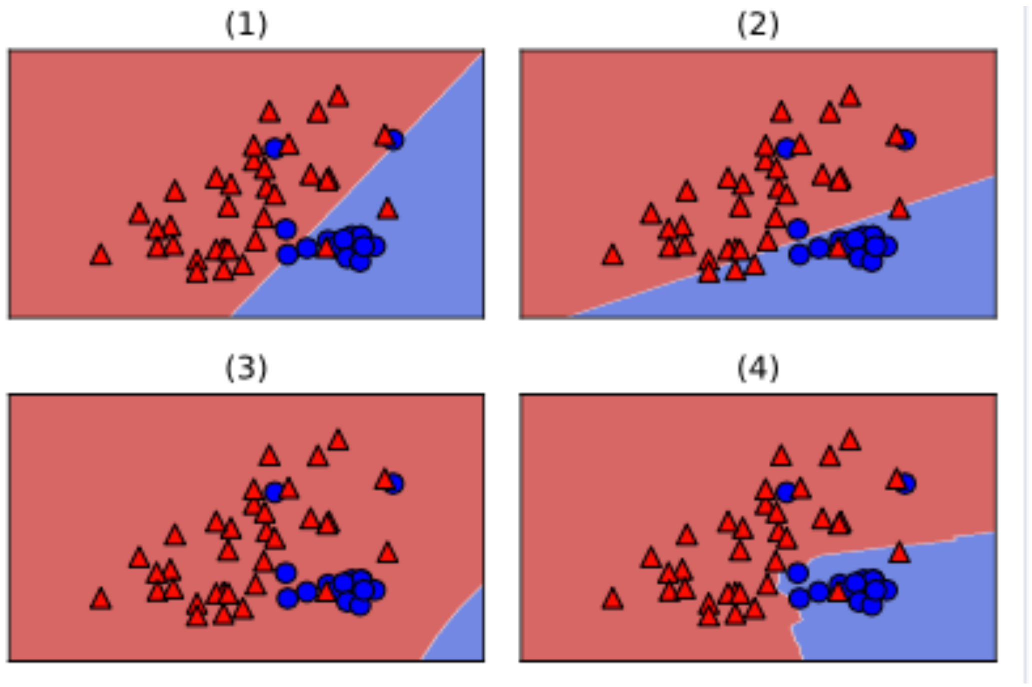
Back to linear classifiers



nice example

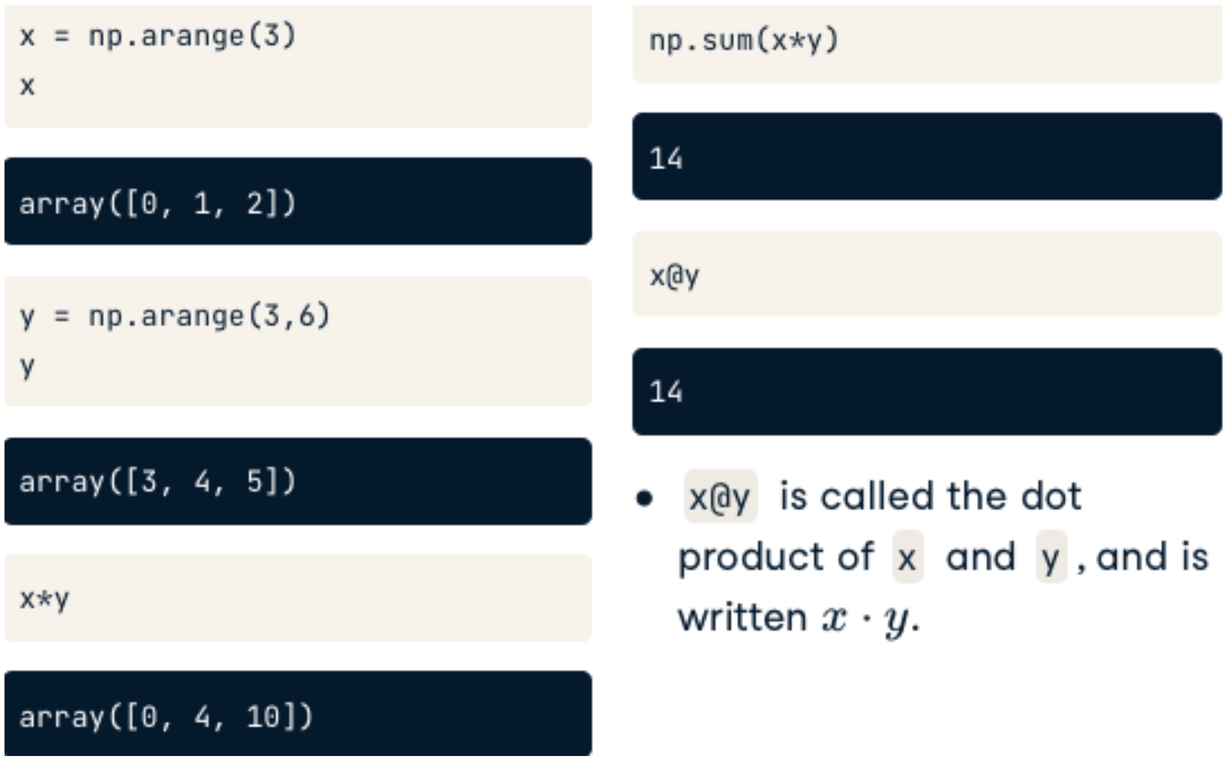
```
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC, LinearSVC
from sklearn.neighbors import KNeighborsClassifier

# Define the classifiers
classifiers = [LogisticRegression(), LinearSVC(), SVC(), KNeighborsClassifier()]

# Fit the classifiers
for c in classifiers:
    c.fit(X,y)

# Plot the classifiers
plot_4_classifiers(X, y, classifiers)
plt.show()
```

output>



shows clearly that knn is not a linear classifier
*quick side not in this example either is SVC() but that is not clearly seen with this visual

Digging deeper into understanding logistic regression and SVMs
defining a dot product
example



```
x = np.arange(3)
x
```

```
array([0, 1, 2])
```

```
y = np.arange(3,6)
y
```

```
array([3, 4, 5])
```

```
x*y
```

```
array([0, 4, 10])
```

```
np.sum(x*y)
```

```
14
```

```
x@y
```

```
14
```

- `x@y` is called the dot product of `x` and `y`, and is written $x \cdot y$.

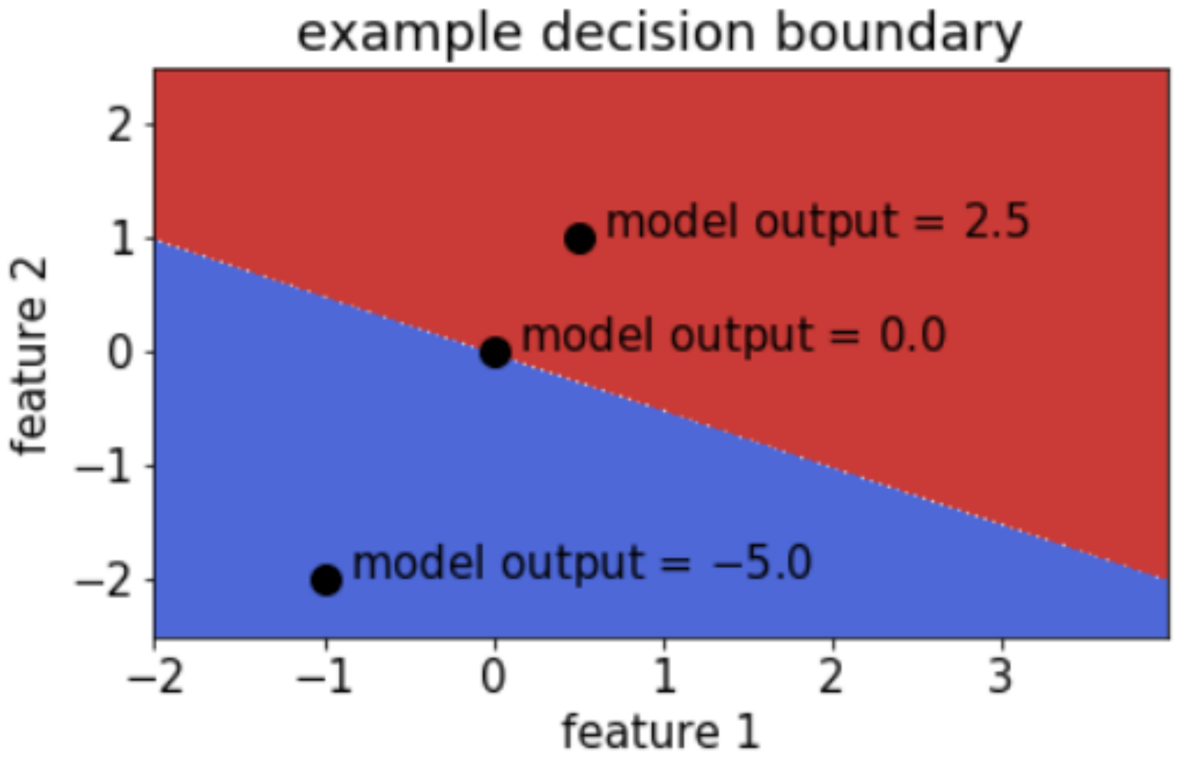
matrix multiplication
or multiplication in higher dimensions

***Linear classifier prediction
using dot products we can express how linea classifiers make predictions
start with the raw model output
raw model output = coefficients dot features + intercept
then
take or check the sign of the quantity (positive or negative)
positive predicts one class and negative predicts the other class
**this pattern is the same for logistic regression and linear SVM
so logistic regression and linear SVM use the same predict function in scikit
**their difference is in the fit function
this is related to loss functions

example of what this looks like using breast cancer classification data set
#create logistic regression object
lr = LogisiticRegression()
#fit it to the data
lr.fit(X, y)
#look at predictions on examples 10 and 20
lr.predict(X)[10]
output > 0
lr.predict(X)[20]

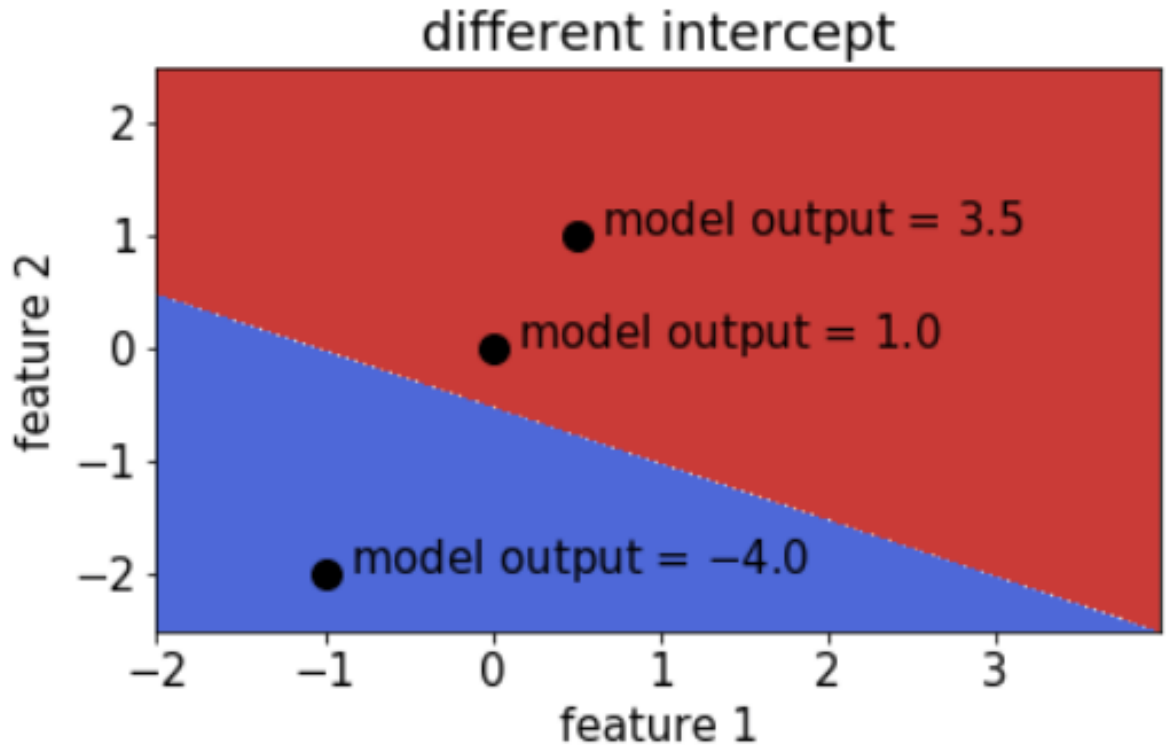
output > 1
let’s look under the hood
#break down the raw model output
lr.coef_ @ X[10] + lr.intercept_
ouput > array([-33.785..])
what matters?
output is negative, so this is attached to the ‘negative’ class which we have defined as 0
the next one
lr.coef_ @ X[20 + lr.intercept_
output > array([0.081...])
output is positive, so this is attached to the ‘positive’ class which we have defined as 1

Visualize

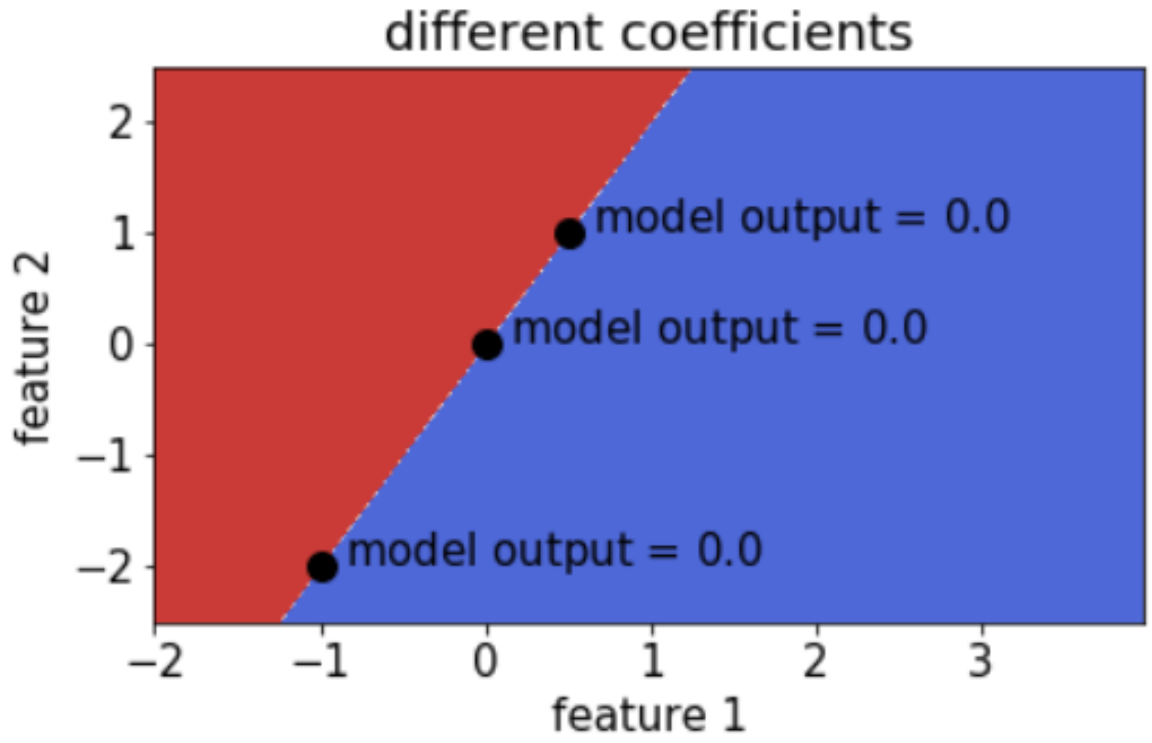


2D example
positive or negative tells you what side of the decision boundary you are on and thus your prediction
*the values of the coefficients and intercept determine the boundary

**boundary position shifts depending on the intercept

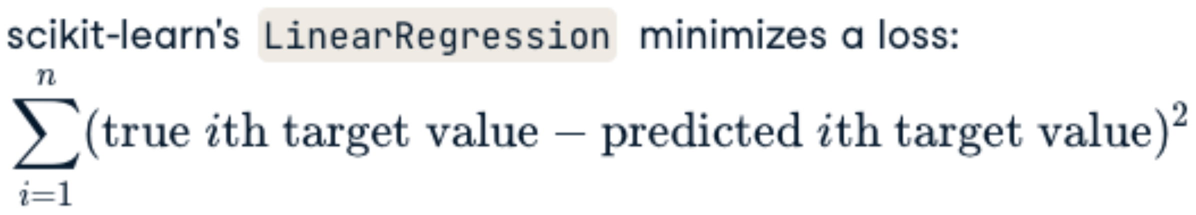


**boundary orientation shifts depending on the coefficients



What is a loss function?

involves minimizing a loss
example least squares linear regression
which minimizes the sum of squares of the errors made on your training set



scikit-learn's `LinearRegression` minimizes a loss:

$$\sum_{i=1}^{n}(\text{true } i\text{th target value} - \text{predicted } i\text{th target value})^2$$

**here, error is defined as the difference between the true target value and the predicted target value
can ‘jiggle’ the coeffs or parameters to minimize loss function
loss function is a penalty score that tells us how precise the model is doing on the training data
**visualize ’fit’ function as runnign coe that minimizes the loss
*do not mix up the score and fit function (although they can be the same sometimes)
loss is used to fit the model on the data and the score is used to see how well we are doing

Sidebar - error term and loss function
No, the error term and the loss function are not the same thing, although they are related concepts in the context of machine learning and statistics.

Error Term:
- In the context of linear regression, the error term (also known as the residual) represents the difference between the actual observed value and the predicted value generated by the regression model.
- The error term accounts for the discrepancy between the model's predictions and the true outcomes in the training data.
- The goal in linear regression is to minimize the sum of squared error terms to find the best-fitting line that minimizes the overall prediction error.

Loss Function:
- The loss function is a mathematical function that quantifies how well a machine learning model's predictions match the true outcomes.
- The loss function measures the error or discrepancy between the predicted values and the actual values in the training data.
- In addition to linear regression, loss functions are used in various machine learning algorithms, including classification, regression, and neural networks.
- The choice of loss function depends on the nature of the problem. For example, Mean Squared Error (MSE) is a common loss function for regression tasks, while Cross-Entropy is used for classification tasks.

In summary, the error term specifically refers to the difference between observed and predicted values in linear regression, while the loss function is a broader concept used to measure the overall error or discrepancy between predicted and actual values across various types of machine learning algorithms.

Classification errors
squared error is not appropriate for classification problems
why?
because y-values are categories, not numbers
Squared error is not typically used for classification problems because it is not well-suited to the nature of classification tasks. Here's why squared error is not a suitable choice for classification problems:

1. Nature of Class Labels: In classification, the output values are discrete class labels (e.g., "0" or "1" for binary classification). Squaring the difference between predicted and actual class labels doesn't make intuitive sense, as class labels are not continuous and do not have a meaningful ordinal relationship.

2. Lack of Gradient Information: Squared error loss provides gradient information that leads to updates in the model's parameters during training. However, in classification, the gradient information from squared error loss can be problematic. It can cause slow or unstable convergence and lead to poor model performance.

3. Imbalanced Classes: In many classification problems, the classes may be imbalanced, meaning one class has significantly more samples than the other(s). Squared error loss treats all errors equally, which can disproportionately penalize misclassifications of the minority class, leading to biased predictions.

4. Outliers: Squaring the errors can heavily penalize outliers, which might not be the desired behavior for classification problems. Misclassifying outliers might not be as critical as in regression tasks.

Instead of squared error, classification problems often use specialized loss functions that are better suited to the problem's characteristics, such as:

- Cross-Entropy (Log Loss): A common choice for binary and multiclass classification tasks. It measures the dissimilarity between predicted class probabilities and actual class labels.

- Hinge Loss (SVM Loss): Used for Support Vector Machines and is suitable for problems involving margin-based classification.

- F1 Score, Precision, Recall, etc.: These metrics evaluate the performance of classification models in terms of precision, recall, and the trade-off between them.

These loss functions and metrics take into account the discrete nature of class labels, handle imbalanced classes, and provide meaningful guidance for updating model parameters during training.

Classificaton errors cont'd
natural quantity to think about is the number of errors we've made
goal is to make this as small as possible
number of errors might be a good loss function
we refer to this loss function as 0-1 loss
defined as 0 for correct prediction and 1 for wrong prediction
then sum it up over all training examples
*the problem with this loss function is that it is hard to minimize

Minimizing a loss
example
from scipy.optimize import minimize
minimize(np.square, 2).x
what is this? > above is equivalent to y = x^2, this is computed with np.square
second argument is our initial guess
'.x' grabs the input value that makes the function as small as possible
output > array([-1.888…e-08])

example

```
# The squared error, summed over training examples, fin the values of the weights (w)
# s is the placeholder for the sum of squared errors (**this forms the basis of the loss function)
def my_loss(w):
    s = 0
    for i in range(y.size):
        # Get the true and predicted target values for example 'i'
        y_i_true = y[i]
        y_i_pred = w@X[i]
        s = s + (y_i_true - y_i_pred)**2
    return s

# Returns the w that makes my_loss(w) smallest
w_fit = minimize(my_loss, X[0]).x
print(w_fit)
```

```
# Compare with scikit-learn's LinearRegression coefficients
lr = LinearRegression(fit_intercept=False).fit(X,y)
print(lr.coef_)
```

Sidebar - understanding w@X[i]
In the context of the given code, `w @ X[i]` represents the dot product between the weight vector `w` and the feature vector `X[i]` of the `i`-th training example.

Let's break it down:

- `w`: This is a weight vector containing coefficients for each feature in your dataset. It represents the parameters you are trying to optimize in order to find the best-fitting linear model.

- `X[i]`: This is the feature vector of the `i`-th training example. It contains the feature values for that specific example.

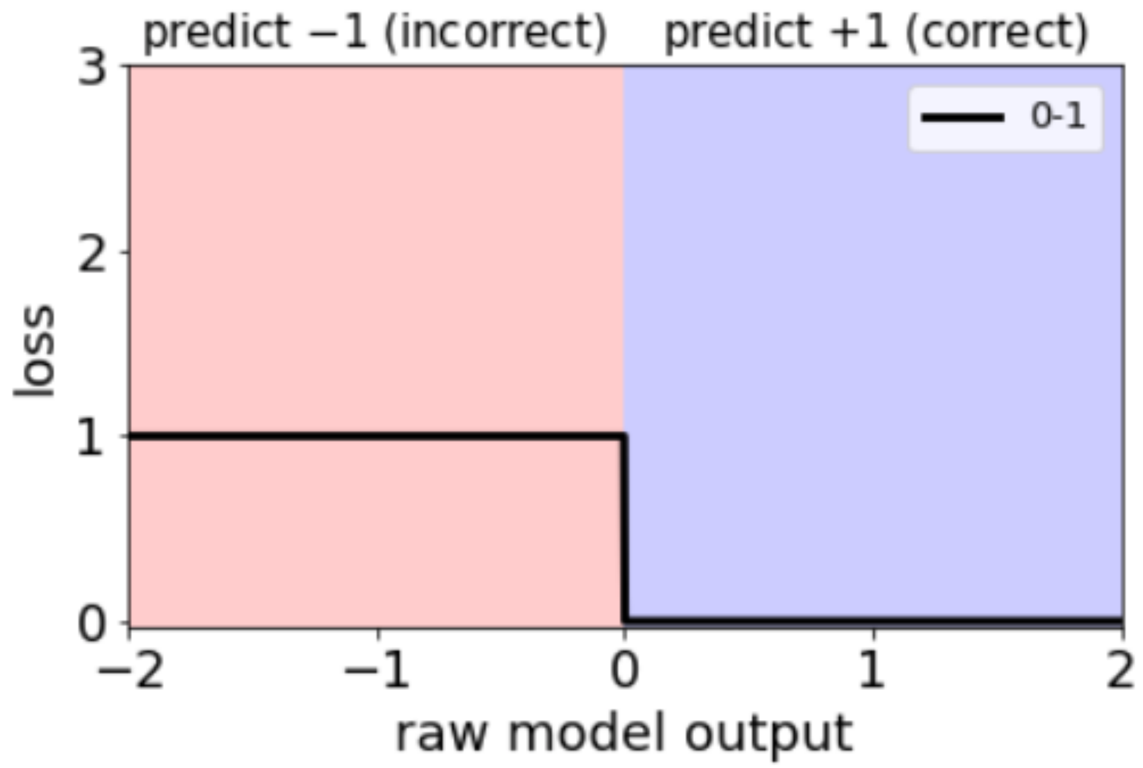
The dot product `w @ X[i]` is computed by multiplying each corresponding feature value from `X[i]` with its corresponding weight from `w`, and then summing up these products. Mathematically, it can be expressed as:

```
w @ X[i] = w[0]*X[i][0] + w[1]*X[i][1] + ... + w[n]*X[i][n]
```

Here, `w[0]` is the first weight, `X[i][0]` is the first feature value of the `i`-th example, and so on.

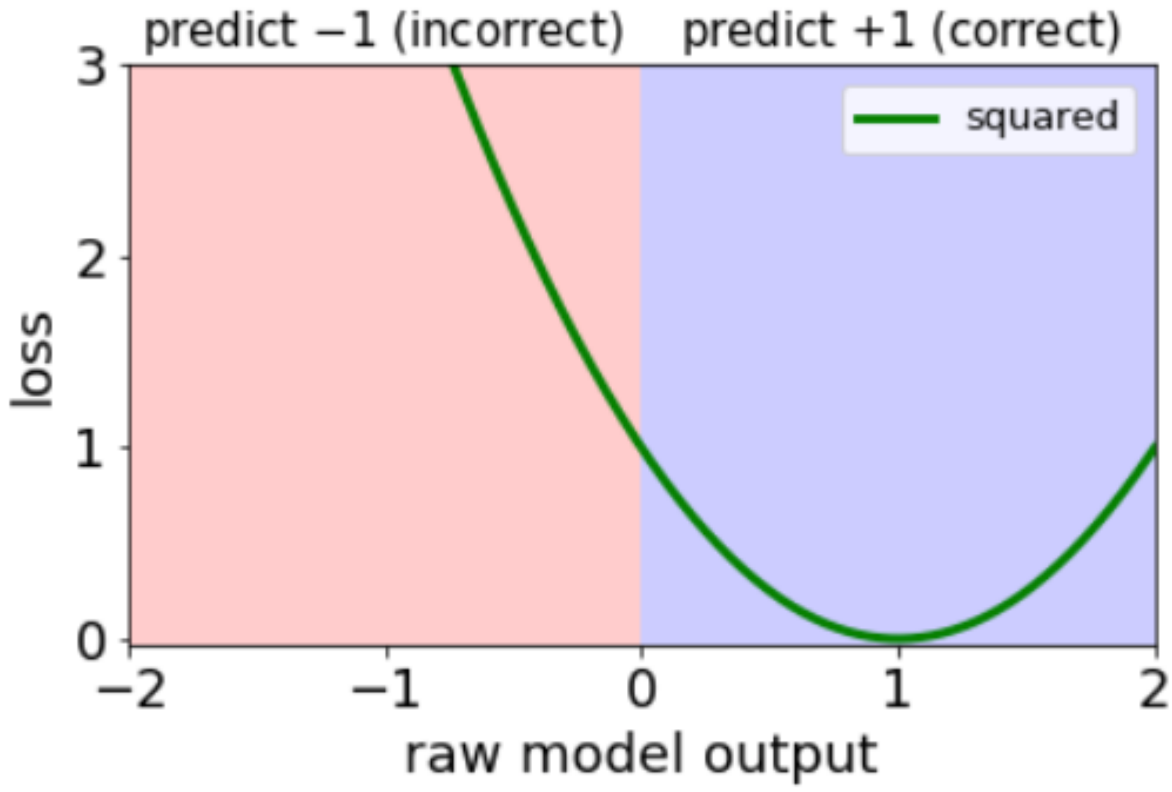
So, `w @ X[i]` gives you the linear combination of the feature values in `X[i]` weighted by the coefficients in `w`. It's the predicted target value for the `i`-th example using the current weight vector `w`.

Loss function diagrams
0-1 loss diagram



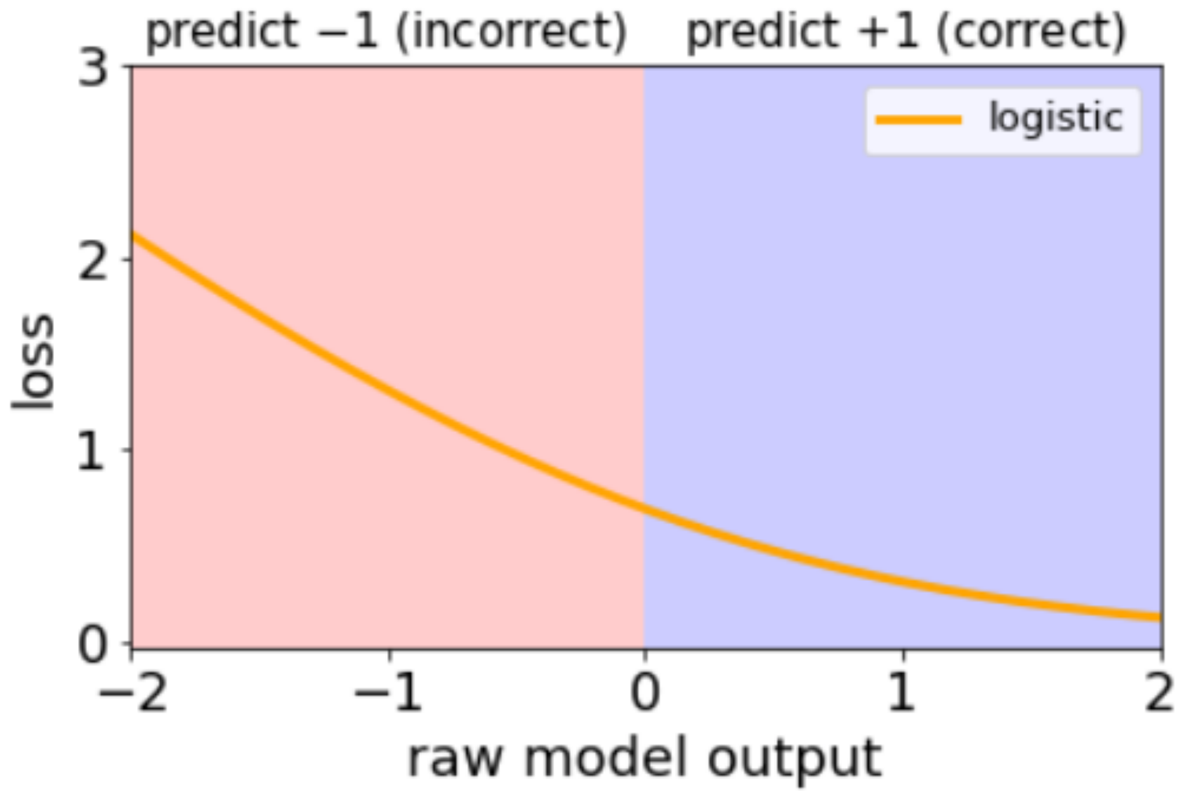
*just a reminder this diagram does not show a decision boundary
the axes here are not two features
also this only shows the loss of the example not the whole sample

Linear regression loss diagram



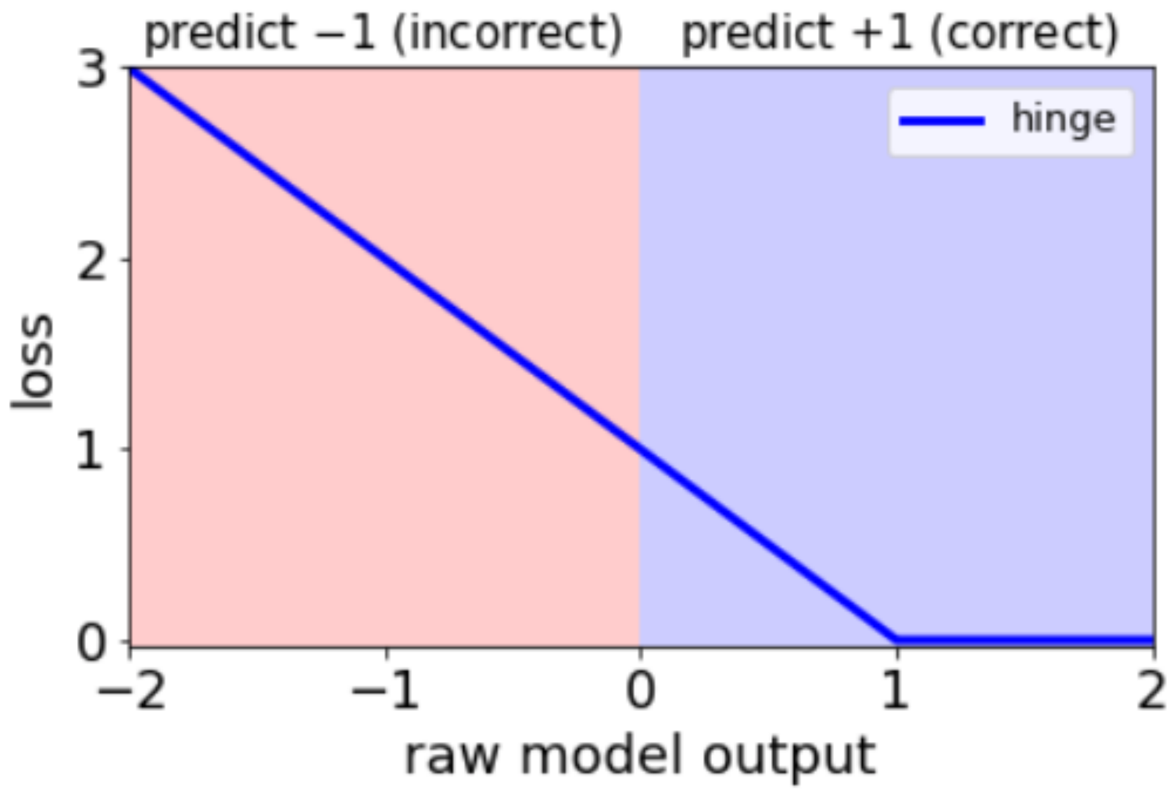
a diagram of least squares linear regression
a squared or quadratic function
this does not work for linear classifiers
here the loss is higher as the prediction is further away from the true target value
which makes sense for linear regression
but for our purposes (linear classifiers) it does not make sense
for us the distance from the true value doesn’t matter as long as we get the sign right
with this model loss is not minimized on the right arm despite those models being perfectly fit models
this is why we need specialized loss functions for classification

Logistic loss diagram



thought of as a smooth version of the 0-1 loss
has properties we want, as you move to the right towards the correct predictions, loss goes down

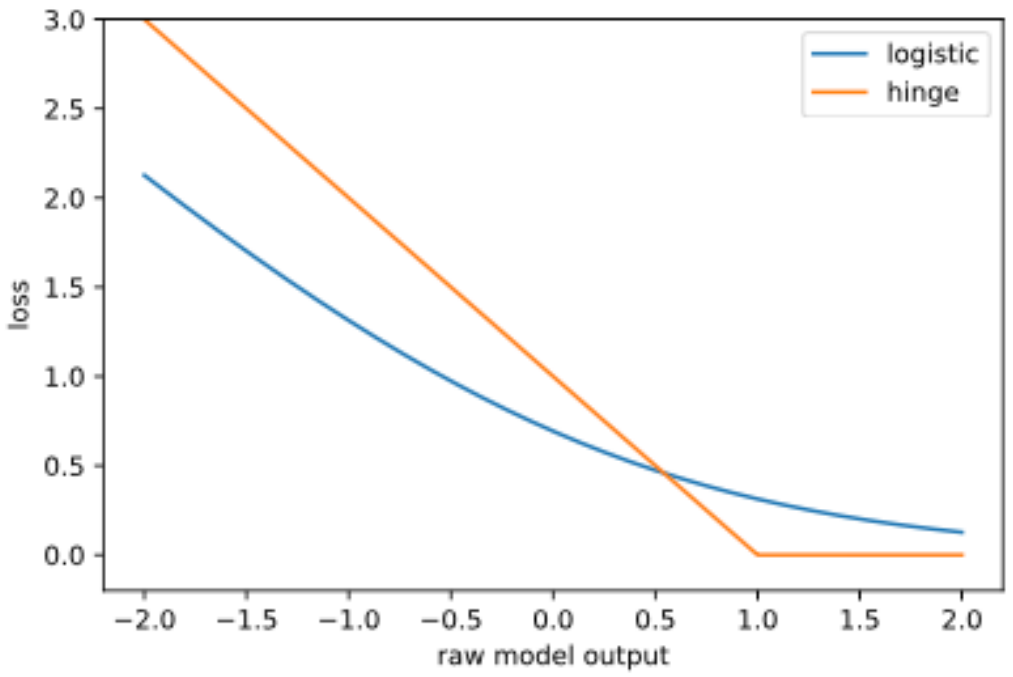
Hinge loss diagram



hinge loss is used in SVMs
penalizes incorrect predictions

Example

```
# Mathematical functions for logistic and hinge losses
def log_loss(raw_model_output):
   return np.log(1+np.exp(-raw_model_output))
def hinge_loss(raw_model_output):
   return np.maximum(0,1-raw_model_output)

# Create a grid of values and plot
grid = np.linspace(-2,2,1000)
plt.plot(grid, log_loss(grid), label='logistic')
plt.plot(grid, hinge_loss(grid), label='hinge')
plt.legend()
plt.show()
```



Sidebar - why do we add 1 to the exponential function?
The addition of 1 in the logistic loss function, specifically in the term `log(1 + exp(-raw_model_output))`, serves two primary purposes:

1. **Numerical Stability**: The logistic loss function involves the calculation of the exponential function, which can lead to numerical instability for very large or very negative values of `raw_model_output`. Adding 1 to the exponent helps mitigate this issue. When `raw_model_output` is significantly positive or negative, `exp(-raw_model_output)` might underflow to zero, leading to issues in calculations. By adding 1, you ensure that the result is at least 1, which prevents these extreme cases.

2. **Smoothness**: The addition of 1 smooths the behavior of the loss function. Without the addition of 1, the loss function would be undefined for negative values of `raw_model_output`, which could lead to problems during optimization. By adding 1, the loss function becomes smooth and continuous across all values of `raw_model_output`.

In summary, adding 1 to the exponential term in the logistic loss function improves its numerical stability and ensures that the loss function is well-behaved for a wide range of inputs. This modification is a common practice in logistic regression and other algorithms that use the logistic loss function.

Sidebar - why do we feed the negative of our parameter into the exponential function?
In logistic regression, we use the negative of the raw model output (also known as

logits) as an input to the logistic loss function. This is a fundamental component of the logistic loss function and is essential for the function to work properly. Let's break down why this is done:

1. **Sigmoid Transformation**: In logistic regression, the raw model output is not directly interpretable as probabilities. To map the raw output to a valid probability range between 0 and 1, we apply the sigmoid function (also called the logistic function) to it. The sigmoid function transforms the raw output into a probability value.

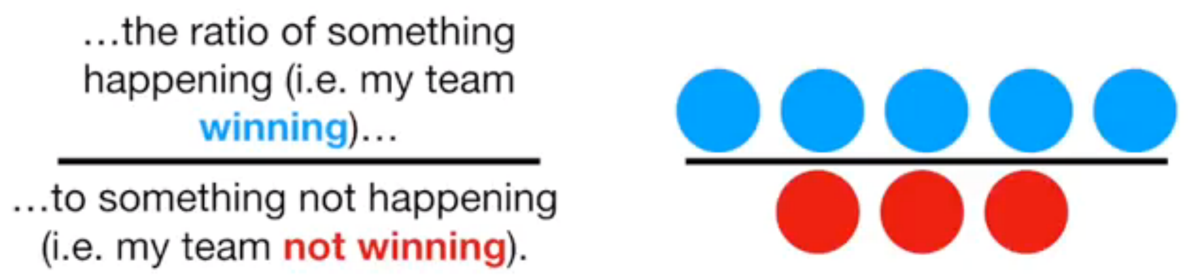
2. **Log-Odds Interpretation**: The raw model output represents the log-odds or log-likelihood of the positive class. Taking the negative of the raw output corresponds to negating these log-odds. The purpose of this is to have a value that can be compared to probabilities. Negative log-odds are more intuitive to work with in terms of probabilities.

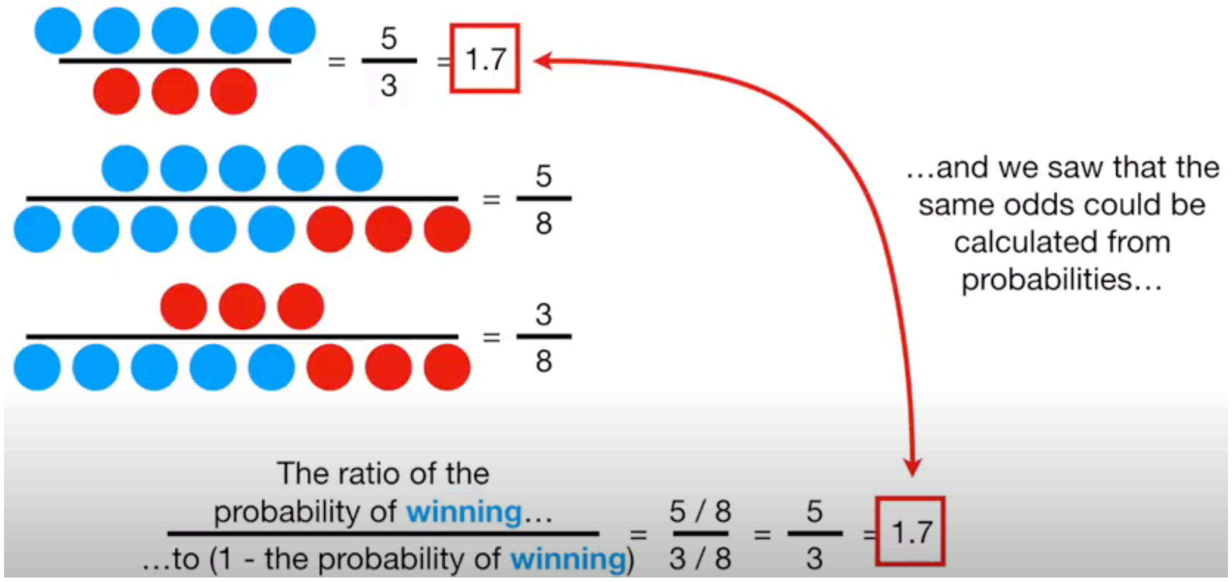
3. **Loss Function Formulation**: The logistic loss function is designed to measure the difference between the predicted probabilities and the true labels. To use the sigmoid-transformed probabilities in the loss function, we need to negate the log-odds and pass it through the loss function. This allows us to compare the predicted probabilities to the true labels and penalize the model for incorrect predictions.

Mathematically, the logistic loss function involves the negative of the raw model output because this combination, when passed through the sigmoid function and into the loss formula, effectively computes the discrepancy between predicted probabilities and true labels. The negative sign ensures that the loss increases as the predicted probabilities deviate from the true labels.

In summary, using the negative of the raw model output is a critical step in logistic regression to transform log-odds into probabilities and to formulate the logistic loss function that measures the discrepancy between predicted and actual outcomes.

Sidebar - refresher on odds and Log-odds
Odds are:


…the ratio of something happening (i.e. my team winning)…
…to something not happening (i.e. my team not winning).


= 5/3 = 1.7
= 5/8
= 3/8
…and we saw that the same odds could be calculated from probabilities…
The ratio of the probability of winning…
…to (1 - the probability of winning)
= 5 / 8 / 3 / 8 = 5/3 = 1.7

Log-odds, also known as log-odds ratios or logits, play a significant role in various fields such as statistics, machine learning, and epidemiology. Here's a quick synopsis of what log-odds are:

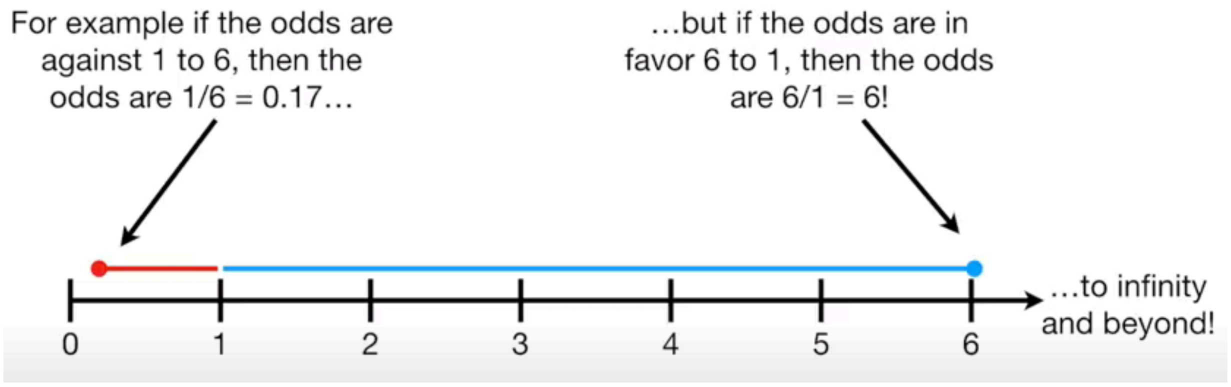
1. Odds and Probability:
In the context of probability and statistics, "odds" refer to the ratio of the probability of an event occurring to the probability of the event not occurring. The odds can be expressed as "p/(1-p)", where "p" is the probability of the event occurring.

2. Log-Odds: (log of the ratio of the probabilities is called the logit function, which forms the basis for logitstic regression

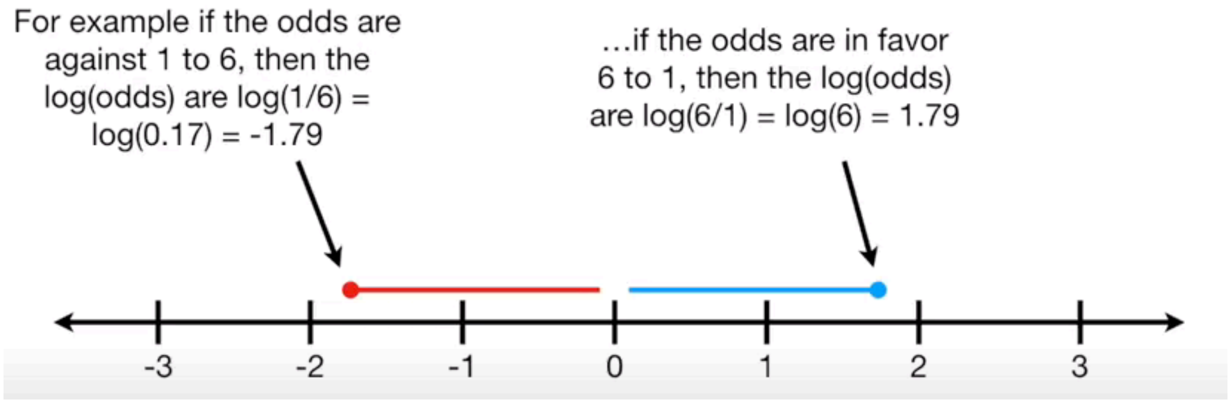
$$\log(\text{odds}) = \log\left(\frac{5}{3}\right) = \boxed{\log\left(\frac{p}{(1-p)}\right)} = \log(1.7) = 0.53$$

Log-odds, or the logarithm of odds, are a way to transform odds into a linear scale. This transformation can be particularly useful because it maps probabilities, which are bounded between 0 and 1, to a range from negative infinity to positive infinity. The log-odds transformation helps in creating a linear relationship between variables that might not have a linear relationship in their original form.

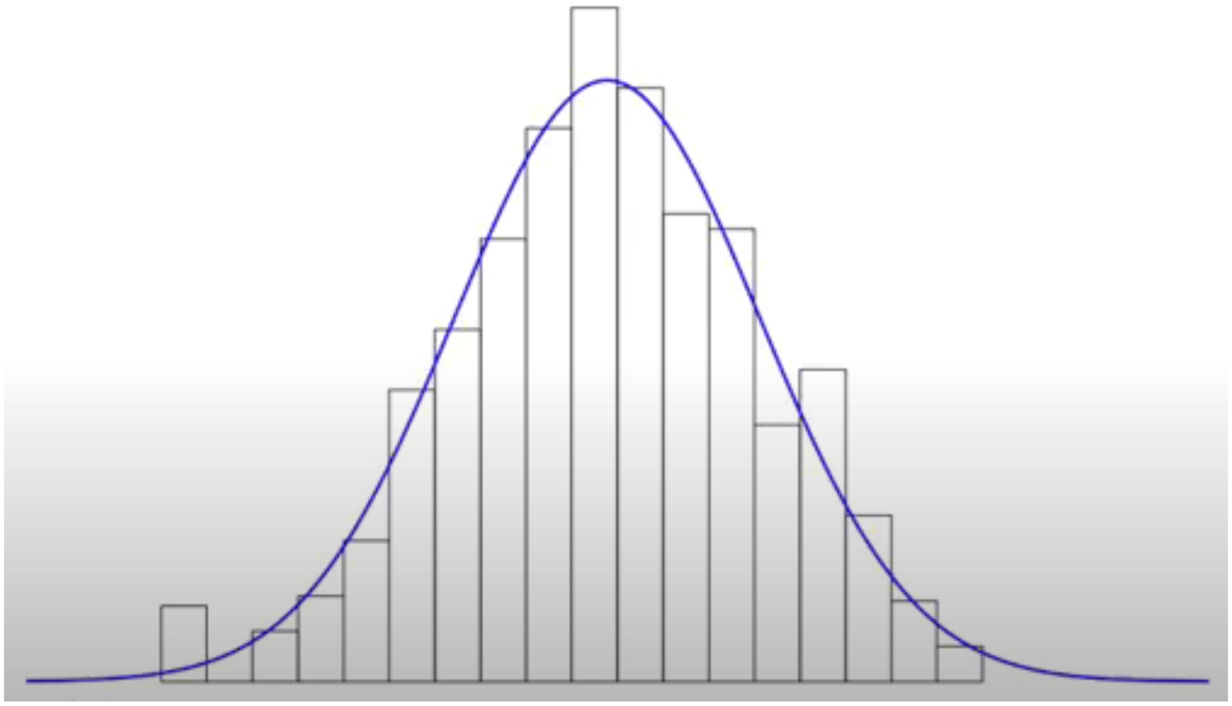
Visuals:


For example if the odds are against 1 to 6, then the odds are 1/6 = 0.17…
…but if the odds are in favor 6 to 1, then the odds are 6/1 = 6!
0
1
2
3
4
5
6
…to infinity and beyond!

*Odds for losing are bound between 0 and 1, but log(odds) is bound between negative infinity and positive infinity.


For example if the odds are against 1 to 6, then the log(odds) are log(1/6) = log(0.17) = -1.79
…if the odds are in favor 6 to 1, then the log(odds) are log(6/1) = log(6) = 1.79
-3
-2
-1
0
1
2
3

This is powerful because now if you pick pairs of random numbers that add up to 100 (for example) and use them to calculate the log(odds), and then plot them to a histogram, you will get a normal distribution.



**Now this makes log(odds) useful for solving certain statistics problems, specifically trying to determine probabilities about winning/losing, or yes/no, or true/false types of situations.

3. Interpretation:
Log-odds have a meaningful interpretation when it comes to categorical data or binary outcomes. For example, in the context of logistic regression, the log-odds can be interpreted as the natural logarithm of the ratio of the probability of a positive outcome to the probability of a negative outcome. In other words, it quantifies the change in the odds of the event occurring for a one-unit change in the predictor variable.

4. Use Cases:
Log-odds are commonly used in logistic regression, which is a statistical method for modeling the relationship between a binary outcome variable and one or more predictor variables. In logistic regression, the linear combination of predictor variables is transformed into log-odds using the sigmoid (logistic) function. This transformation allows us to predict probabilities of the binary outcome.

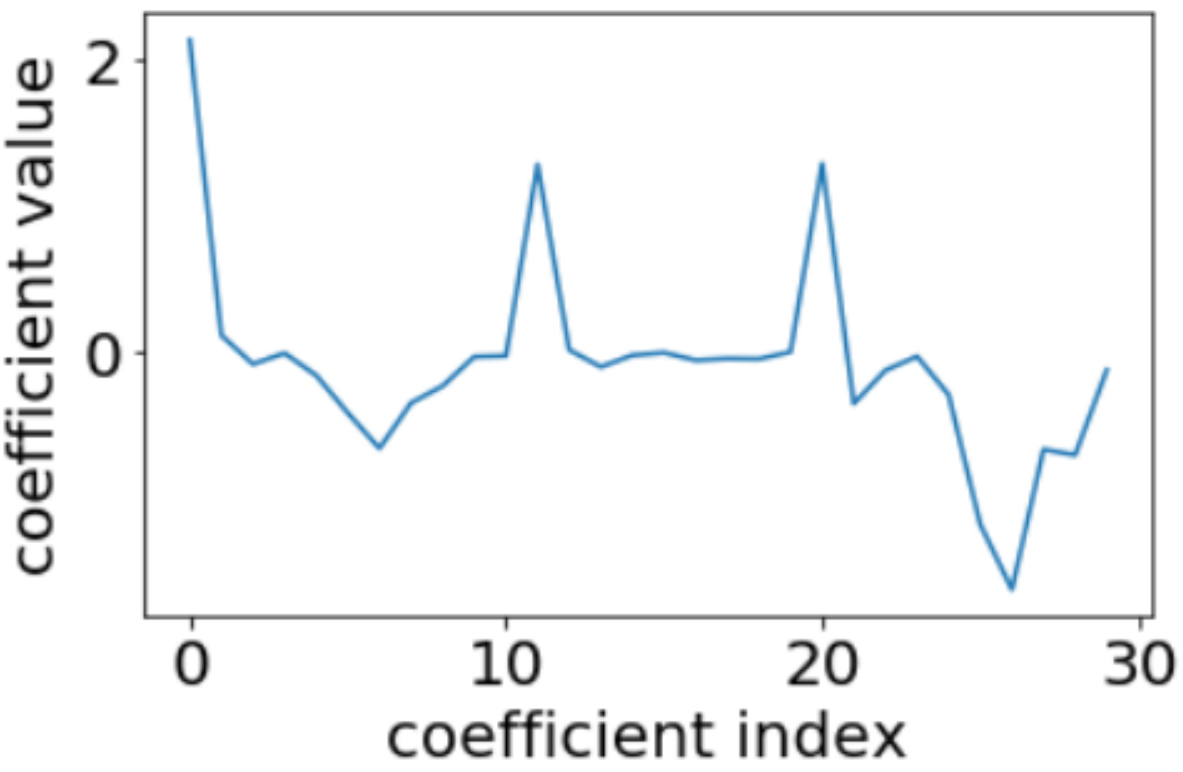
5. Conversion to Probability:
To convert log-odds back to probabilities, we use the inverse of the sigmoid function. The sigmoid function maps the log-odds to a probability range between 0 and 1. This conversion is important when interpreting model outputs or making predictions in logistic regression.

In summary, log-odds are a fundamental concept in statistics and machine learning, particularly in the context of logistic regression. They provide a way to represent probabilities of binary outcomes on a linear scale, enabling us to model and analyze relationships between variables and make predictions about binary events.

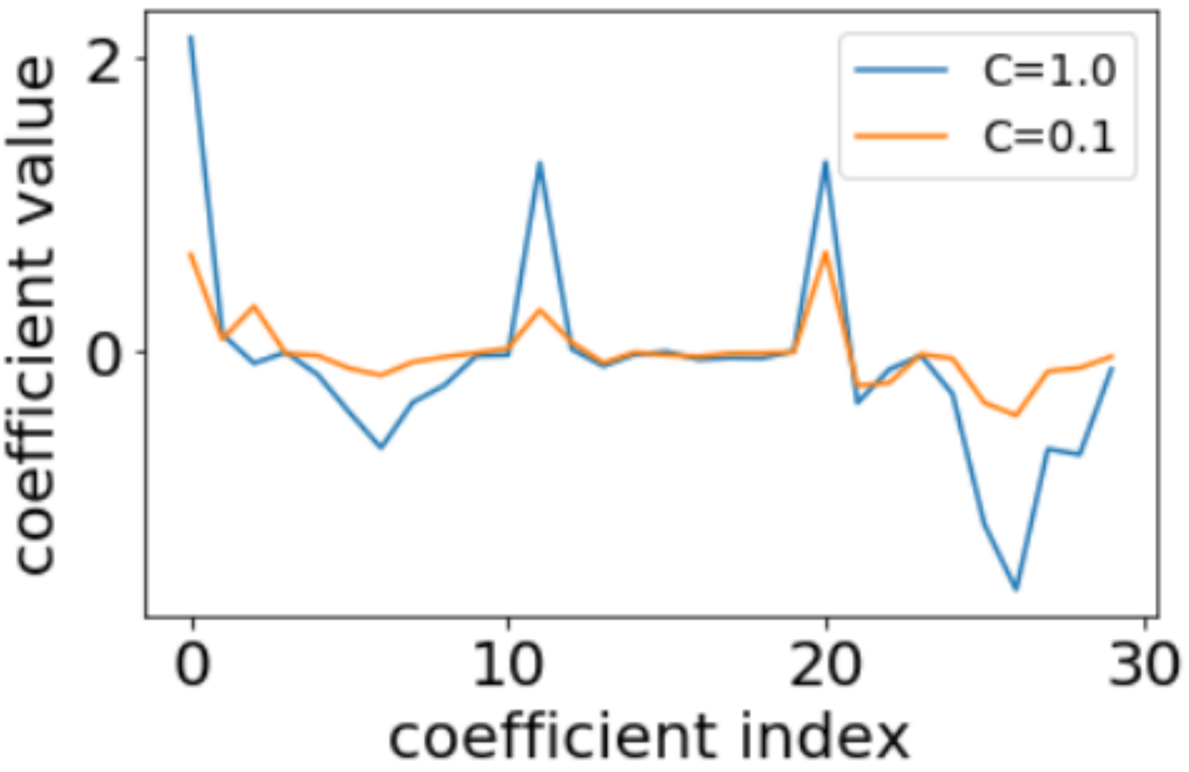
example

```
# The logistic loss, summed over training examples
def my_loss(w):
    s = 0
    for i in range(0,569):
        raw_model_output = w@X[i]
        s = s + log_loss(raw_model_output * y[i])
    return s

# Returns the w that makes my_loss(w) smallest
w_fit = minimize(my_loss, X[0]).x
print(w_fit)

# Compare with scikit-learn's LogisticRegression
lr = LogisticRegression(fit_intercept=False, C=1000000).fit(X,y)
print(lr.coef_)
```

Logistic regression and regularization
regularization combats overfitting by making the model coefficients smaller
visual example of default regularizes logistic regression



in scikit the hyperparameter ‘C’ is the inverse of the regularization strength
larger C means less regularization and smaller C means more regularization



How regularization influences training and test accuracy?

```
lr_weak_reg = LogisticRegression(C=100)
lr_strong_reg = LogisticRegression(C=0.01)

lr_weak_reg.fit(X_train, y_train)
lr_strong_reg.fit(X_train, y_train)

lr_weak_reg.score(X_train, y_train)
lr_strong_reg.score(X_train, y_train)
```

```
1.0
0.92
```

$$\text{regularized loss} = \text{original loss} + \text{large coefficient penalty}$$

- more regularization: lower training accuracy

why is this?
regularization is an extra term added to the original loss function
regularization penalizes large coefficients
the loss function without regularization (added penalties) maximizes the training accuracy metric
remember the smaller we set C, the larger the regularization penalty
the larger the regularization the more we deviate from our goal of maximizing training accuracy

How does regularization affect testing accuracy?

```
lr_weak_reg.score(X_test, y_test)
```

```
0.86
```

```
lr_strong_reg.score(X_test, y_test)
```

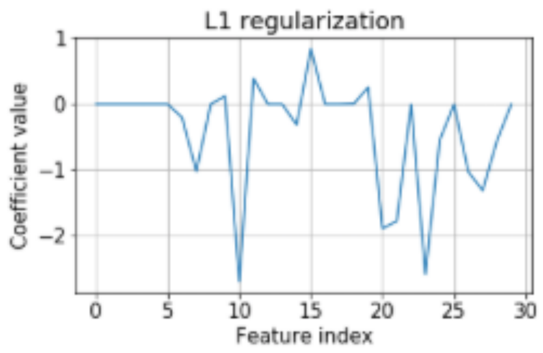
```
0.88
```

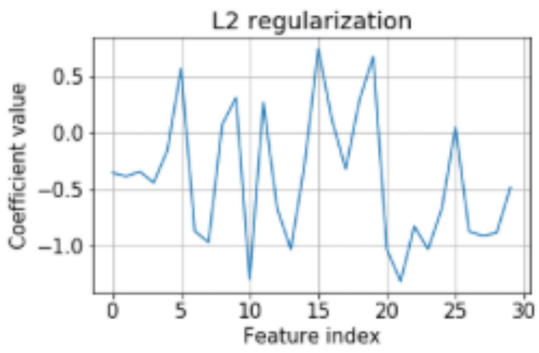
More regularization almost always higher test accuracy

**Regularizing (makes your coefficient smaller) is like a compromise between not using the feature at all (setting the coefficient to zero) and fully using it (the un-regularized coefficient value
*using a feature too heavily causes overfitting, regularization causes you to ‘fit less’
features are coefficients, coefficients are constant multipliers
coefficients represent forces of the environment (the gaunlet) that each obsvervation must run through
calculating all of these is impossible
and there in lies the art of machine learning and statistical models
like a set of dials being fine-tuned in order to optimize the output

L1 vs L2 regularization
to delineate the two different types of regularization within linear regression we use Ridge and Lasso
Ridge is L2
Lasso is L1
Lasso also performs features selection
*important to scale features prior to applying regularization
example - compare L1 and L2 on breast cancer dataset
#solver argument controls the optimization method used to find the coefficients
#need to set it for L1 because default solver is not compatible with L1 regularization
lr_l1 = LogisticRegression(solver=‘liblinear’, penalty=‘l1’)
#default solver is compatible for L2
#default penalty argument is L2

```
lr_l2 = LogisticRegression()
lr_L1.fit(X_train, y_train)
lr_L2.fit(X_train, y_train)
plt.plot(lr_L1.coef_.flatten())
plt.plot(lr_L2.coef_.flatten())
output>
```

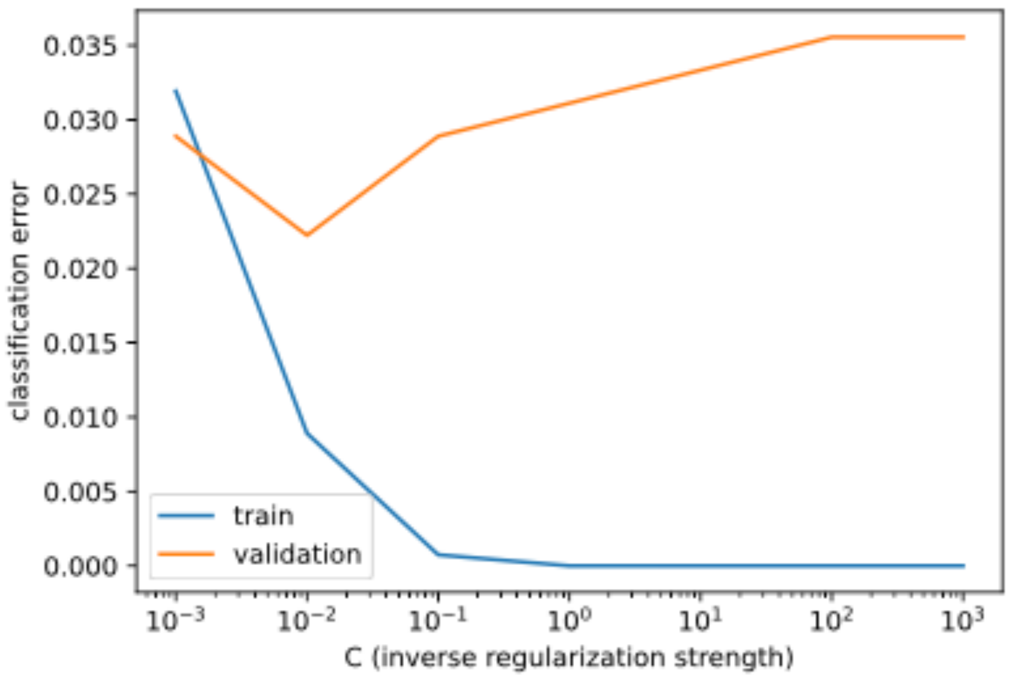




L1 (Lasso) sets many of the coefficients to zero (ie ignores them)
L1 performs feature selection for us
L2 (Ridge) shrinks the coefficients to be smaller

Example

```
# Train and validaton errors initialized as empty list
train_errs = list()
valid_errs = list()

# Loop over values of C_value
for C_value in [0.001, 0.01, 0.1, 1, 10, 100, 1000]:
    # Create LogisticRegression object and fit
    lr = LogisticRegression(C=C_value)
    lr.fit(X_train, y_train)

    # Evaluate error rates and append to lists
    train_errs.append( 1.0 - lr.score(X_train, y_train) )
    valid_errs.append( 1.0 - lr.score(X_valid, y_valid) )

# Plot results
plt.semilogx(C_values, train_errs, C_values, valid_errs)
plt.legend(("train", "validation"))
plt.show()
output>
```



Example

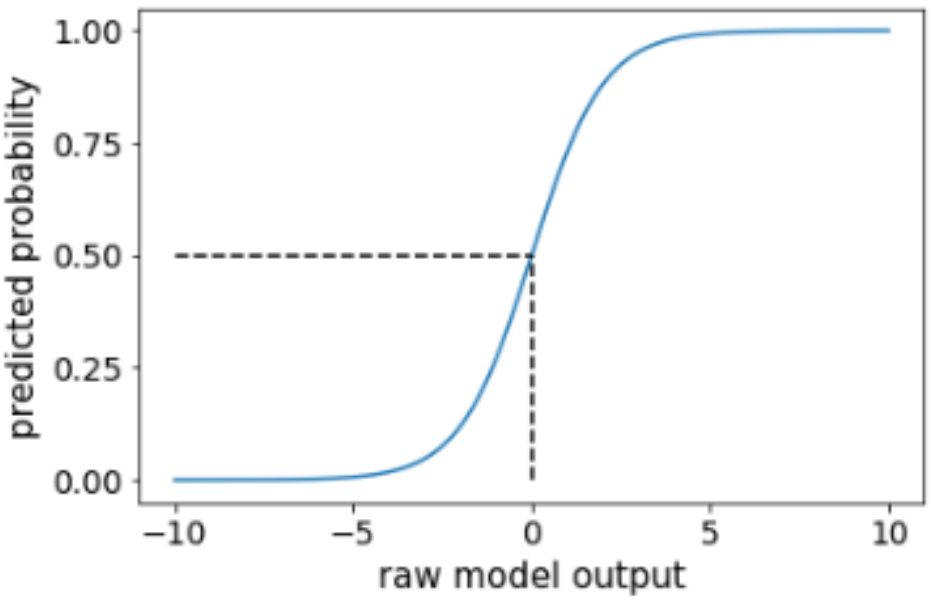
```
# Specify L1 regularization
lr = LogisticRegression(solver='liblinear', penalty='l1')

# Instantiate the GridSearchCV object and run the search
searcher = GridSearchCV(lr, {'C':[0.001, 0.01, 0.1, 1, 10]})
searcher.fit(X_train, y_train)

# Report the best parameters
print("Best CV params", searcher.best_params_)

# Find the number of nonzero coefficients (selected features)
best_lr = searcher.best_estimator_
coefs = best_lr.coef_
print("Total number of features:", coefs.size)
print("Number of selected features:", np.count_nonzero(coefs))
```

flatten() method converts a multi-dimensional array into a one-dimensional array
this is useful when you want to reshape or simplify the structure of your data

Logistic regression probabilities
with the predict_proba function
decision boundary goes across 0.5
ie more than 50% one answer
ie less than 50% another answer

with regularization coeffs will be small (ie less confident predictions)
with regularization probabilities are closer to .5
regularization combats overfitting so this makes sense, we are less confident in our predictions
*ratio of the coefficients gives us the slope of the line
magnitude of the coefficients gives us our confidence level
*regularization also affects the orientation of the boundary
more confident in our predictions the farther away from our boundary

How are probabilities computed?
use the raw model output
probabilities must be between 0 and 1
need to ‘squash’ the raw model output to be between 0 and 1
do this with the sigmoid function



when positive > predict the positive class
when negatvie > predict the negative class

Example
# Set the regularization strength
model = LogisticRegression(C=0.1)

# Fit and plot
model.fit(X,y)
plot_classifier(X,y,model,proba=True)

# Predict probabilities on training points
prob = model.predict_proba(X)

print("Maximum predicted probability", np.max(prob))
#messing around > smaller values of C lead to less confident predictions

Example format
lr = LogisticRegression()
lr.fit(X,y)

# Get predicted probabilities
proba = lr.predict_proba(X)

# Sort the example indices by their maximum probability
proba_inds = np.argsort(np.max(proba,axis=1))

# Show the most confident (least ambiguous) digit
show_digit(proba_inds[-1], lr)

# Show the least confident (most ambiguous) digit
show_digit(proba_inds[0], lr)

Multi-class logistic regression
First approach
One-vs-rest
train a series of binary classifiers for each class
example
lr0.fit(X, y==0)
**this says return an array the same size as y that's True when y is 0 and False otherwise
with this the classifier learns to predict these true/false values
ie we've turned it into a binary classifier learning to discriminate between class 0 or not class 0
do this for the other classes
here we have additional class 1 and 2
lr1.fit(X, y==1)
lr2.fit(X, y==2)
we want the class that gives the largest raw model output (in scikit this is called decision_function)
lr0.decision_function(X)[0]
lr1.decision_function(X)[0]
lr2.decision_function(X)[0]
#in our sample classifier 0 has the largest raw model output
this means that it is more confident that the class is 0 when comparing to the other classes
so we predict class 0

**we can also do this more directly with scikit
lr = LogisticRegression(multi_class='ovr')
lr.fit(X, y)
lr.predict(X)[0]

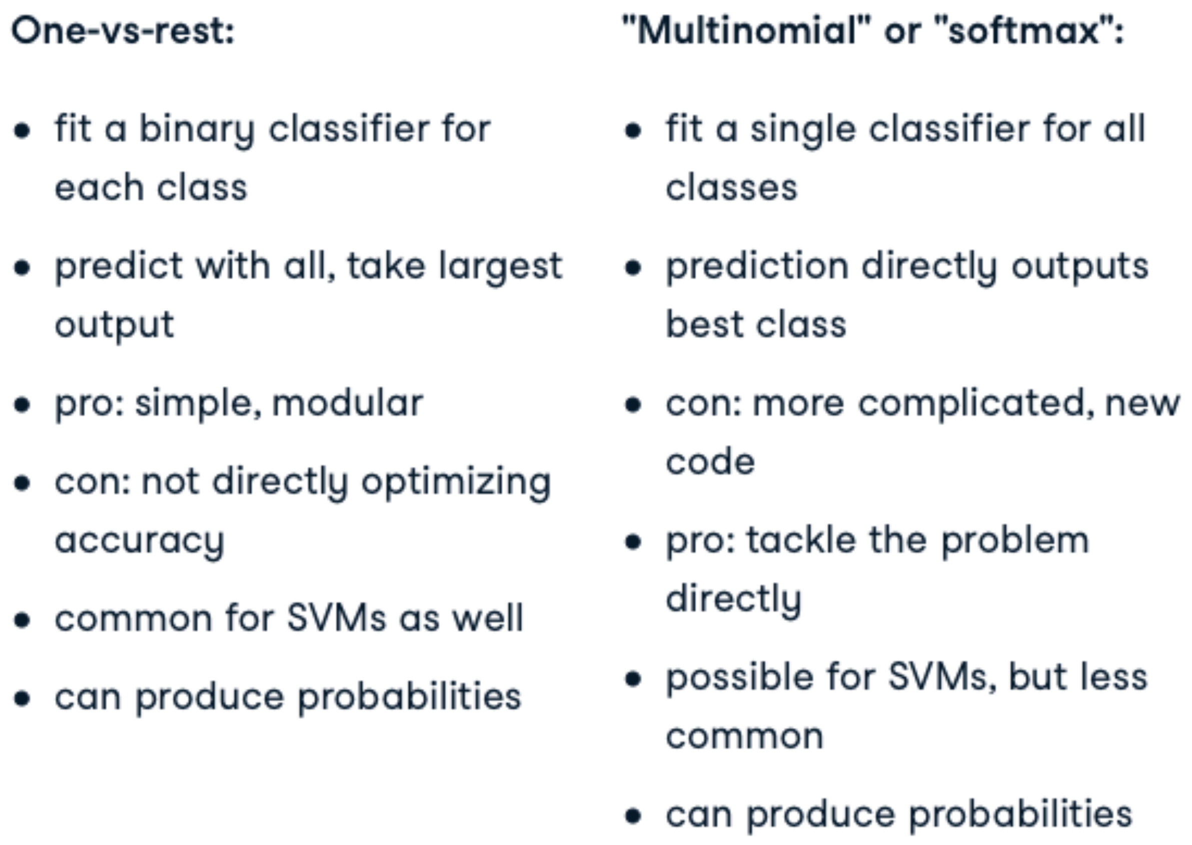
Sidebar - more on multi-class argument of LogisticRegression()
1. `'ovr'` (One-vs-Rest): This is the default strategy. It fits a separate binary logistic regression classifier for each class while considering it as the positive class and the other classes as the negative class. This results in as many binary classifiers as there are classes. During prediction, the class with the highest confidence is chosen.

2. `'multinomial'`: This strategy uses the multinomial logistic regression algorithm to handle multiple classes directly. It creates a single model that takes into account all classes simultaneously. This method can be more computationally efficient than `'ovr'` when the number of classes is large.

3. `'auto'`: This strategy is set by default. It automatically selects between `'ovr'` and `'multinomial'` based on the nature of the problem and the solver chosen.

The choice of `multi_class` should depend on the nature of your dataset, the number of classes, and the solver you intend to use. Note that the `solver` argument also interacts with `multi_class`, as not all solvers support all strategies.

Second approach to achieve multi-class classification with LogisticRegression()
called multinomial or softmax or cross-entropy loss
softmax is popular when working with neural nets
this approach modifies the loss function so that it directly tires to optimize accuracy on the mulit-class problem
Comparison ovr and multinomial



One-vs-rest:

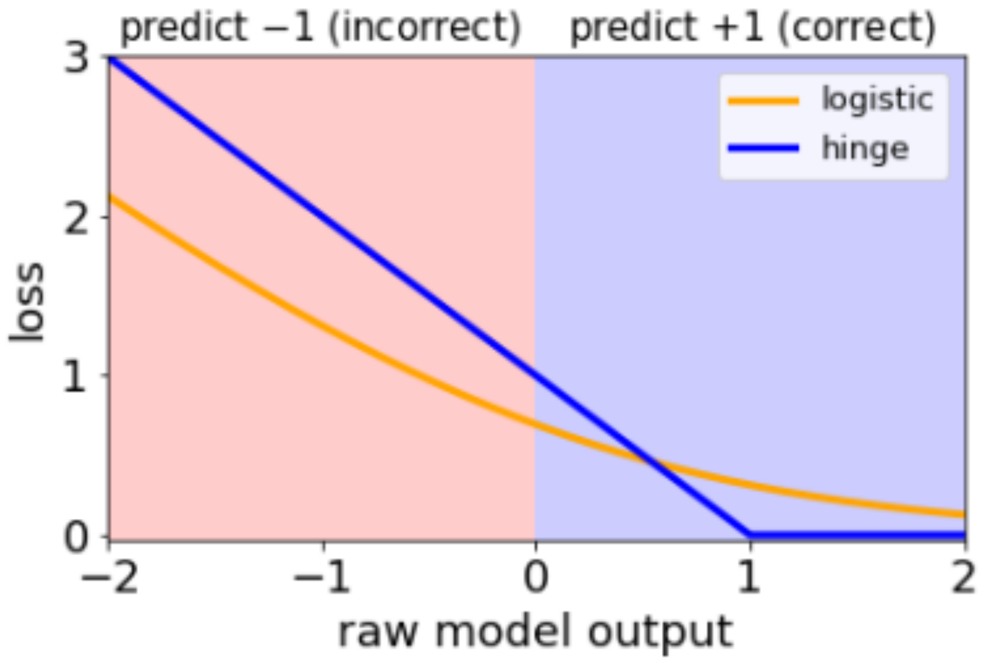
- fit a binary classifier for each class
- predict with all, take largest output
- pro: simple, modular
- con: not directly optimizing accuracy
- common for SVMs as well
- can produce probabilities

"Multinomial" or "softmax":

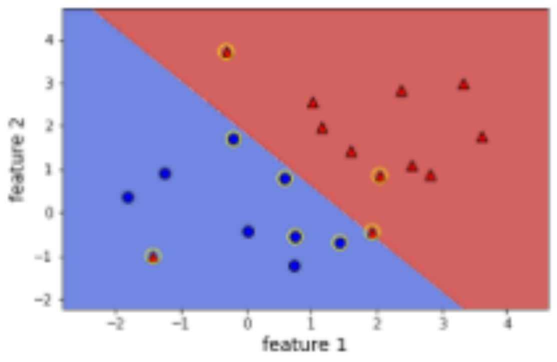
- fit a single classifier for all classes
- prediction directly outputs best class
- con: more complicated, new code
- pro: tackle the problem directly
- possible for SVMs, but less common
- can produce probabilities

Model coefficients for multi-class
for ovr or binary classifiers
we end up with one coeff per feature per class and one intercept per class
for multinomial or non-binary classification
we end up with the same set of coeffs
these two approaches work differently but they learn the same number of parameters and roughly have the same interpretations

Support Vectors
a linear classifier that uses the hinge loss function
standard version uses L2 regularization



*key part to remember with the hinge loss function
is the flat part which occurs when the raw model output is greater than 1
meaning you predicted an example correctly beyond some margin of error
this leads to the key part of an SVM
if a training example falls in this area (“zero loss” region), it doesn’t contribute to the fit
if we remove this example, then nothing would change
support vectors are defined as examples that are not in the flat part (‘zero loss’ region)
support vectors are shown with yellow circles around them:



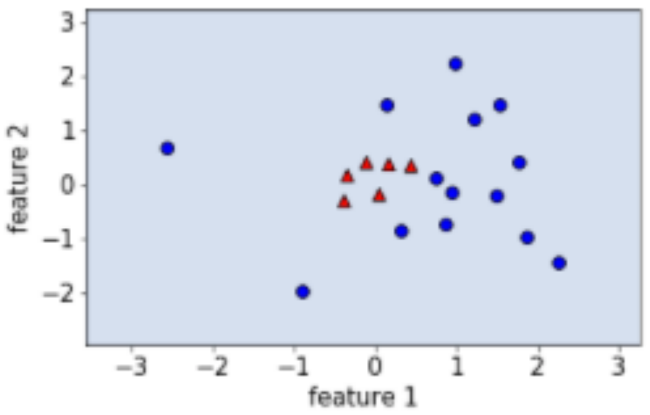
**support vectors are defined by incorrectly classified examples as well as correctly classified examples that are close to the boundary
*how close is considered close is controlled by the regularization strength
*support vectors are the examples that matter to your fit
if an example is not a support vector, removing it has no effect on the model

the reason > these non-support vectors loss was already zero
these non-support vectors are what makes the hinge loss different from logistic regression
*for logistic regression all points matter to the fit
SVMs are popular because they are suprisingly fast to fit and predict
*a big part of this are algorithms whose running time only scales with the number of support vectors, rather than the total number of training examples
'maximize the margin' is sometimes considered a part of SVM
however this concept is advantageous to linearly separable datasets (ie datasets that can accomplish a training accuracy of 100%)
**unfortunately most dataset will not be linearly separable

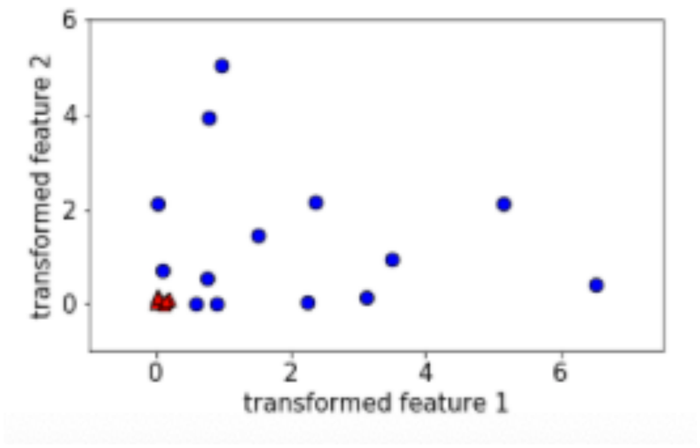
Example

```
# Train a linear SVM
svm = SVC(kernel="linear")
svm.fit(X, y)
plot_classifier(X, y, svm, lims=(11,15,0,6))

# Make a new data set keeping only the support vectors
print("Number of original examples", len(X))
print("Number of support vectors", len(svm.support_))
X_small = X[svm.support_]
y_small = y[svm.support_]

# Train a new SVM using only the support vectors
svm_small = SVC(kernel="linear")
svm_small.fit(X_small, y_small)
plot_classifier(X_small, y_small, svm_small, lims=(11,15,0,6))
```

Kernel SVMs
how to fit nonlinear boundaries using linear classifiers?



here clearly not able to fit a linear boundary that perfectly classifies all the points
however here you can see that all the red points are around coordinates (0, 0)
we can use this to our advantage, we can create two new features by squaring feature 1 and squaring feature 2
this will make values near 0 become small and values far from zero large for both positive and negative values
we have 'transformed' the features
now lets look at the data >



In this transformed space we can make a clear linear boundary
What does this look like in a the original space?
An ellipse
What does this mean?
**fitting a linear model in a transformed space corresponds to fitting a nonlinear model in the original space
*key to remember, transformation does not always have to be squared and boundary isn't always going to be an ellipse
often the new space will have a different number of dimensions from the original space

More on Kernel SVMs
from sklearn.svm import SVC
svm = SVC(gamma=1)
*default kernel is 'rbf'
RBF is radial basis function
very complicated algorithm
but ultimately what it does is a complicated transformation of the features then fits a linear boundary in the new space
C hyperparameter controls regularization
gamma hyperparameter controls the smoothness of the boundary
the higher the gamma the more complex the boundaries and the higher the training accuracy

with gamma we can reach 100% training accuracy by creating little islands around the separable data
obviously the problem here is overfitting

Example

```
# Instantiate an RBF SVM
svm = SVC()

# Instantiate the GridSearchCV object and run the search
parameters = {'gamma':[0.00001, 0.0001, 0.001, 0.01, 0.1]}
searcher = GridSearchCV(svm, parameters)
searcher.fit(X_train, y_train)

# Report the best parameters
print("Best CV params", searcher.best_params_)

# Instantiate an RBF SVM
svm = SVC()

# Instantiate the GridSearchCV object and run the search
parameters = {'C':[0.1, 1, 10], 'gamma':[0.00001, 0.0001, 0.001, 0.01, 0.1]}
searcher = GridSearchCV(svm, parameters)
searcher.fit(X_train, y_train)

# Report the best parameters and the corresponding score
print("Best CV params", searcher.best_params_)
print("Best CV accuracy", searcher.best_score_)

# Report the test accuracy using these best parameters
print("Test accuracy of best grid search hypers:", searcher.score(X_test, y_test))
```

Comparing logistic regression and support vector machines
both are linear classifiers and can be extended to multi-class
differences:
LR > outputs meaningful probabilities, all data points affect fit, and can be L1 or L2 regularization
SVM > does not naturally output probabilities, only support vectors affect fit, and usually L2 regularization and hinge loss

SGDClassifier
stands for stochastic gradient descent
scales well to large datasets

```
from sklearn.linear_model import SGDClassifier
```

allows you to easily switch between types of linear classifers
example
logreg = SGDClassifier(loss=‘log_loss’)
linsvm = SGDClassifier(loss=‘hinge’)
this emphasizes that the model is the same, and only the loss changes
*one key trip up to be aware of is SGDClassifer’s regularization hyperparameter is ‘alpha’
alpha is the inverse of C
meaning the bigger the alpha the more regularization

Example

```
# We set random_state=0 for reproducibility
linear_classifier = SGDClassifier(random_state=0)

# Instantiate the GridSearchCV object and run the search
parameters = {'alpha':[0.00001, 0.0001, 0.001, 0.01, 0.1, 1],
             'loss':['hinge', 'log_loss']}
searcher = GridSearchCV(linear_classifier, parameters, cv=10)
searcher.fit(X_train, y_train)

# Report the best parameters and the corresponding score
print("Best CV params", searcher.best_params_)
print("Best CV accuracy", searcher.best_score_)
print("Test accuracy of best grid search hypers:", searcher.score(X_test, y_test))
```